

A Minimal Book Example

John Doe

2023-07-01

Contents

1	About	5
1.1	Usage	5
1.2	Render book	5
2	Software Overview	7
3	Software Installation	9
3.1	Install R	9
3.2	Install RStudio	10
3.3	Install R Markdown	10
3.4	Install LaTeX	11
3.5	Install R Packages	11
3.6	18-Step Test	12
4	Getting Started	19
4.1	RStudio Interface	19
4.2	R Data Types and Structures	22
4.3	Importing Data in R	39
5	DataCamp	57
6	Assignment	59

Chapter 1

About

This is a *sample* book written in **Markdown**. You can use anything that Pandoc’s Markdown supports; for example, a math equation $a^2 + b^2 = c^2$.

1.1 Usage

Each **bookdown** chapter is an .Rmd file, and each .Rmd file can contain one (and only one) chapter. A chapter *must* start with a first-level heading: **# A good chapter**, and can contain one (and only one) first-level heading.

Use second-level and higher headings within chapters like: **## A short section** or **### An even shorter section**.

The **index.Rmd** file is required, and is also your first book chapter. It will be the homepage when you render the book.

1.2 Render book

You can render the HTML version of this example book without changing anything:

1. Find the **Build** pane in the RStudio IDE, and
2. Click on **Build Book**, then select your output format, or select “All formats” if you’d like to use multiple formats from the same book source files.

Or build the book from the R console:

```
# ## Render book
#
# You can render the HTML version of this example book without changing anything:
```

```
#  
# 1. Find the Build pane in the RStudio IDE, and  
#  
# 1. Click on Build Book, then select your output format, or select "All formats"  
#  
# Or build the book from the R console:  
  
bookdown::render_book()
```

This module provides an overview of the software used in this course, guides you through the installation process, and helps you become familiar with their usage. All the software used in this course is freely available.

Chapter 2

Software Overview

This course makes use of various software tools and programming languages that are essential for conducting economic analyses. The following software and programming languages will be used:

1. **R:** R is a programming language designed for statistical computing and graphics. This language is widely used by data scientists and researchers for a range of tasks such as data processing, visualization, model estimation, and performing predictive or causal inference. For instance, one can use R to import GDP data, plot the data, compute the GDP growth rate from this data, and finally, apply time-series modeling techniques to predict future GDP growth.
2. **LaTeX:** LaTeX is a powerful document preparation system widely used for typesetting scientific and technical documents. Similar to Microsoft Word, LaTeX is a text formatting software, but it offers advanced support for mathematical equations, cross-references, bibliographies, and more. LaTeX is particularly useful for creating professional-looking PDF documents with complex mathematical notation.
3. **Markdown:** Markdown is designed for simple and easy formatting of plain text documents. It uses plain text characters and a simple syntax to add formatting elements such as headings, lists, emphasis, links, images, and code blocks. Markdown allows for quick and readable content creation without the need for complex formatting options. It is often used for creating documentation, writing blog posts, and formatting text in online forums. Markdown documents can be easily converted to other formats, making it highly portable.
4. **R Markdown:** R Markdown combines R with Markdown, LaTeX, and Microsoft Word. This fusion creates an environment where data scientists and researchers can combine text and R code within the same document,

eliminating the process of creating graphs in R and then transferring them to a Word or LaTeX document. An R Markdown document can be converted into several formats, including HTML, PDF, or Word. To generate a PDF, R Markdown initially crafts a LaTeX file which it then executes in the background. Thanks to the embedded R code in the R Markdown document, it's possible to automate data downloading and updating to ensure a financial report remains up-to-date. In fact, the text you're reading now was crafted with R Markdown.

5. **RStudio:** RStudio is an **Integrated Development Environment (IDE)** for R. An IDE is a software application that combines multiple programs into a single, user-friendly platform. Think of RStudio as the all-in-one tool you'll use in this course - it will handle all tasks, running R, Markdown, and LaTeX in the background for you. However, for RStudio to work, R, R Markdown, and a LaTeX processor must be installed on your computer, so that RStudio can use these programs in the background.
6. **R packages:** R provides a rich set of basic functions that can be extended with R packages. These packages are a collection of functions written by contributors for specific tasks. For example, the **quantmod** package provides functions for financial quantitative modeling.

These are all **open-source** programs, meaning they are freely available to users and their development is driven by a community of developers who voluntarily contribute their expertise to improve their functionalities.

By gaining proficiency in these software tools and programming languages, you will have a solid foundation for performing data analysis, creating reproducible reports, and effectively communicating your findings in a professional manner.

Chapter 3

Software Installation

For this course, you will only use RStudio for writing code and text. However, as RStudio is an IDE rather than a standalone program, it relies on the presence of R, R Markdown, and a LaTeX processor on your computer. RStudio interacts with these programs in the background to generate an output. Below, you will find the installation instructions for each of these programs. Additionally, a set of 18 steps is provided to help you verify whether R, RStudio, R Markdown, and LaTeX have been installed correctly.

3.1 Install R

To install R on your computer, follow the instructions below:

For MacOS:

1. To download R for MacOS, visit the R project website: www.r-project.org.
2. Click CRAN mirror and choose your preferred mirror. It doesn't really matter which mirror you choose, simply choose a location close to you, e.g. National Institute for Computational Sciences, Oak Ridge, TN.
3. Select Download R for macOS.
4. Under "Latest release", read the first paragraph to check whether the program is compatible with your operating system (OS) and processor. To find your computer's OS and processor, click the top left Apple icon, and click "About this Mac." Under "macOS", you will see both the name (e.g. "Ventura", "Catalina", "Monterey") and the number (e.g. "Version 13.4.1") of the OS, and under "Processor" you will either see that your computer is run by an Intel processor or an Apple silicon (M1/M2) processor.
5. If the operating system (OS) and the processor are compatible, click on the first R-X.X.X.pkg (where X represents the R version numbers). Otherwise,

if you have an older OS or an Intel processor, click on a version further down that is compatible with your system.

6. Once the file has downloaded, click it to proceed to installation, leaving all default settings as they are.

For Windows:

1. To download R for Windows, visit the R project website: www.r-project.org.
2. Click CRAN mirror and choose your preferred mirror. It doesn't really matter which mirror you choose, simply choose a location close to you, e.g. Revolution Analytics, Dallas, TX.
3. Select Download R for Windows.
4. Select "base", and read whether the program is compatible with your Windows version.
5. If it is compatible, click Download R-X.X.X for Windows (X are numbers), and otherwise click here for older versions.
6. Once the file has downloaded, click it to proceed to installation, leaving all default settings as they are.

3.2 Install RStudio

1. Visit the RStudio website: www.rstudio.com and navigate to the download page.
2. Click DOWNLOAD.
3. Scroll down to "All Installers" section.
4. Choose the download that matches your computer. If you have a Mac, it's most likely "macOS 10.15+"; then click the download link (e.g. "RStudio-2022.07.1-554.dmg"). If you have a Windows, it's most likely "Windows 10/11" and click the download link (e.g. "RStudio-2022.07.1-554.exe").
5. Open the file when it has downloaded, and install with the default settings.

3.3 Install R Markdown

R Markdown can be installed from inside the RStudio IDE.

1. To download R Markdown, open RStudio, after you have successfully installed R and RStudio.
2. In RStudio, find the "Console" window.
3. Type the command `install.packages("rmarkdown")` in the console and press Enter.

3.4 Install LaTeX

When it comes to installing LaTeX, there are several software options available. While most options work well, I recommend using **TinyTeX**. **TinyTeX** as it is an easy-to-maintain LaTeX distribution. Other good alternatives include **MacTeX** and **MiKTeX**. LaTeX is the underlying program responsible for word processing and generating PDF reports within RStudio.

To install **TinyTeX** using **RStudio**, follow these steps:

1. Open **RStudio** after successfully installing **R**, **RStudio**, and **R Markdown**.
2. Locate the “Console” window within **RStudio**.
3. Type `install.packages("tinytex")` and press Enter.
4. Type `tinytex::install_tinytex()` and press Enter.
5. Type `install.packages("knitr")` and press Enter.

3.5 Install R Packages

R provides a set of basic functions that can be extended using packages. To install a package (e.g., `quantmod`), follow these steps:

1. Open **RStudio**.
2. In the **RStudio** window, find the “Console” window.
3. Type the command `install.packages("quantmod")` in the console and press Enter.
4. Wait for the installation process to complete. R will download and install the package from the appropriate repository.

After installation, you can use the package in your script by including the line `library("quantmod")` at the beginning. Remember to execute the `library("quantmod")` command each time you want to use functions from the `quantmod` package in your code.

It is common practice to load the necessary packages at the beginning of your script, even if you don’t use all of them immediately. This ensures that all the required functions and tools are available when needed and promotes a consistent and organized approach to package management in your code.

As a side note, the `quantmod` package includes the `getSymbols` function, which is commonly used to download financial data, such as the S&P 500 index (GSPC):

```
library("quantmod")
```

```
## Loading required package: xts
```

```
## Loading required package: zoo
```

```
##
```

```
## Attaching package: 'zoo'
```

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

## Loading required package: TTR

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

getSymbols(Symbols="^GSPC")

## [1] "GSPC"

head(GSPC)

##           GSPC.Open GSPC.High GSPC.Low GSPC.Close GSPC.Volume GSPC.Adjusted
## 2007-01-03   1418.03   1429.42   1407.86    1416.60  3429160000    1416.60
## 2007-01-04   1416.60   1421.84   1408.43    1418.34  3004460000    1418.34
## 2007-01-05   1418.34   1418.34   1405.75    1409.71  2919400000    1409.71
## 2007-01-08   1409.26   1414.98   1403.97    1412.84  2763340000    1412.84
## 2007-01-09   1412.84   1415.61   1405.42    1412.11  3038380000    1412.11
## 2007-01-10   1408.70   1415.99   1405.32    1414.85  2764660000    1414.85
```

Here, the `getSymbols` function retrieves the historical data for the S&P 500 index from Yahoo Finance, and stores it in the `GSPC` object. The `head` function then displays the first few rows of the downloaded data.

R packages provide a wealth of specialized functions for specific tasks. To use a function from a particular package, you can indicate the package by preceding the function with the package name followed by a double colon `::`. For example, `quantmod::getSymbols()` specifies the `getSymbols()` function from the `quantmod` package. This practice helps to avoid conflicts when multiple packages provide functions with the same name. It also allows users to easily identify the package associated with the function, promoting clarity and reproducibility in code.

3.6 18-Step Test

To ensure that R, RStudio, R Markdown, and LaTeX are installed properly, you can follow the 18-step test provided in the module. This test will help verify the functionality of the installed programs and identify any potential issues or errors.

During this process, you may encounter the following issues:

- **Issue with Generating PDF:** If you are unable to generate a PDF file in step 15, it is likely due to an issue with the installation of LaTeX. In such cases, please revisit the instructions for installing LaTeX in Section

3.4 and ensure you have followed them correctly. Alternatively, you can consider installing MacTeX or MiKTeX instead of TinyTeX.

- **Non-Latin Alphabet Language:** If your computer language is not based on the Latin alphabet (e.g., Chinese, Arabic, Farsi, Russian, etc.), additional instructions may be required. You can refer to this video for specific guidance: youtu.be/pX_fy2fyM30.


I encourage you to persist and do your best to install all the required software, even if it takes some time. Downloading and installing programs is a critical skill that is essential in almost every profession today. This is an excellent opportunity to acquire this skill.

Keep going and don't hesitate to seek additional support or resources if needed. It's common to encounter challenges when installing software, and resources like [google.com](https://www.google.com) and stackoverflow.com can provide helpful answers and suggestions. If you encounter an error, simply copy and paste the error message into a search engine, and you'll likely find solutions and guidance from the community.


If you fail to install R, RStudio, and LaTeX, I recommend using RStudio Cloud, an online platform where you can perform all the necessary tasks directly in your web browser. You can access RStudio Cloud at rstudio.cloud. While signing up is free, please note that some features may require a fee.

Make a Plot

To continue with the test, make sure you have R, RStudio, R Markdown, and LaTeX installed and are connected to the internet. Follow the steps below in RStudio:

1. Type and execute `install.packages("quantmod")` in the RStudio console.
2. Click on the top-left plus sign  then click **R Script**.
3. Click **File - Save As...** then choose a familiar folder.
4. Copy and paste the following R code into your R Script:

```
library("quantmod")
treasury10y <- getSymbols(Symbols = "GS10", src = "FRED", auto.assign = FALSE)
plot(treasury10y, main = "10-Year Treasury Rate")
```

5. Click on **Source:**  **Source** (or use the shortcut Ctrl+Shift+Enter or Cmd+Shift+Return).

You should now see a plot of the 10-year Treasury rate on your screen. Compare it to the rate displayed on fred.stlouisfed.org/series/GS10.

Save Plot as PDF


Continue with the following steps in RStudio:



Figure 3.1: R Plot


6. Add the line: `pdf(file="myplot.pdf",width=6,height=4)` before the plot function, and add `dev.off()` after the plot:

```
library("quantmod")
treasury10y <- getSymbols(Symbols = "GS10", src = "FRED", auto.assign = FALSE)
pdf(file = "myplot.pdf", width = 6, height = 4)
plot(treasury10y, main = "10-Year Treasury Rate")
dev.off()
```

7. Click on **Source**:  **Source** (or use the shortcut Ctrl+Shift+Enter or Cmd+Shift+Return).
8. Now navigate to the same folder on your computer where you saved the R script.
9. There should be a file called `myplot.pdf` - open it.

You should now see the PDF version of the plot displaying the Treasury rate. If you encounter no error message but cannot locate the `myplot.pdf` file, it's possible that R saved it in a different folder than where the R script is located. To check where R saves the plot, type `getwd()` in the console, which stands for "get working directory." If you want to change the working directory and have R save the files in a different folder, type `setwd("/Users/.../...")`, replacing `"/Users/.../..."` with the path to the desired folder.

Run Marked Code

To run only one line or one variable, mark it and then click Run:  Run (or use the shortcut Ctrl+Enter or Cmd+Return). Follow these steps in RStudio:

10. Mark the variable `treasury10y`:


```
2 library("quantmod")
3 treasury10y <- getSymbols(Symbols="GS10",src="FRED",auto.assign=FALSE)
4 pdf(file="myplot.pdf",width=6,height=4)
5 plot(treasury10y,main="10-Year Treasury Rate")
6 dev.off()
```

11. Click Run:  Run (or use shortcut Ctrl+Enter or Cmd+Return)


You should see the data displayed in your console, ending with 2023-05-01 3.57.

Create PDF with R Markdown

Next, let's ensure that R Markdown is working. If you have installed LaTeX and `knitr`, follow these steps in RStudio:

12. Click on the top-left plus sign  then click R Markdown...
13. A dialog box will appear - select Document and choose PDF, then click OK:

You should now see a file with text and code.

14. Click File - Save As... and choose a familiar folder to save the file.
15. Click Knit:  Knit (or use the shortcut Ctrl+Shift+K or Cmd+Shift+K).

A PDF file should appear on your screen and also in your chosen folder.

16. Next, locate the following lines:

```
16 ```{r cars}
17 summary(cars)
18 ```
```


17. Replace these lines with the following (do not copy the line numbers):

```
16 ```{r, message=FALSE,warning=FALSE,echo=FALSE}
17 library("quantmod")
18 treasury10y <- getSymbols(Symbols="GS10",src="FRED",auto.assign=FALSE)
19 plot(treasury10y,main="10-Year Treasury Rate")
20 ```
```

18. Click Knit:  Knit (or use the shortcut Ctrl+Shift+K or Cmd+Shift+K).

You should now see a file that looks similar to this:

Hint: You can set `echo=TRUE` to include R code in your report.

You can now change the title of the file and the text to create a professional report. If you click the arrow next to Knit:  Knit you have options to

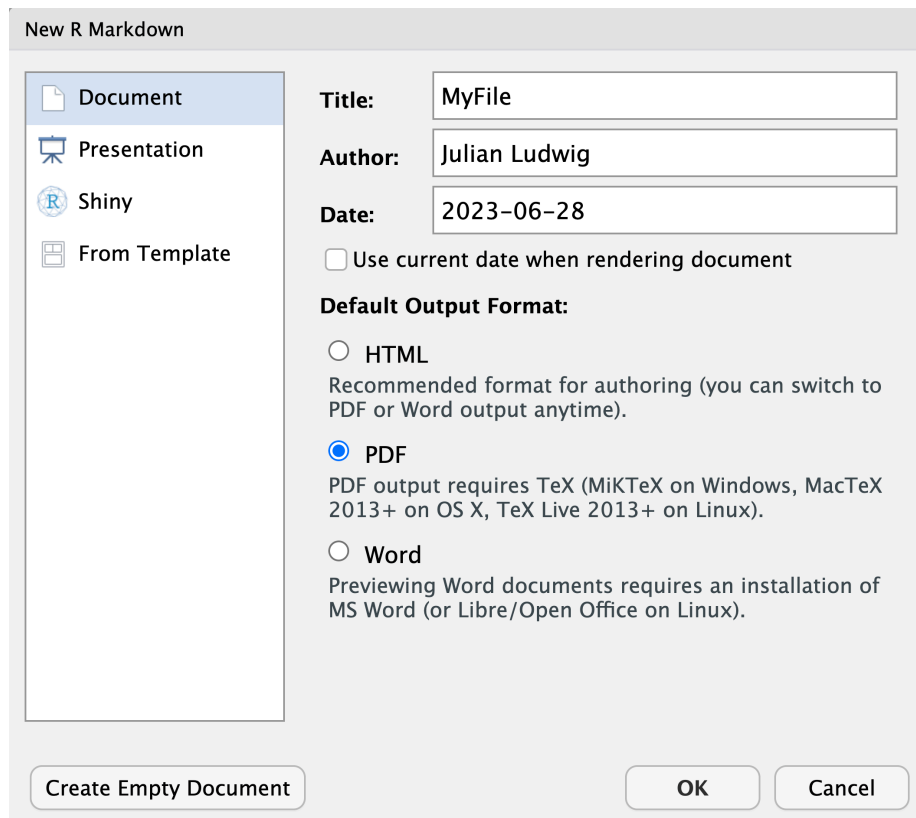


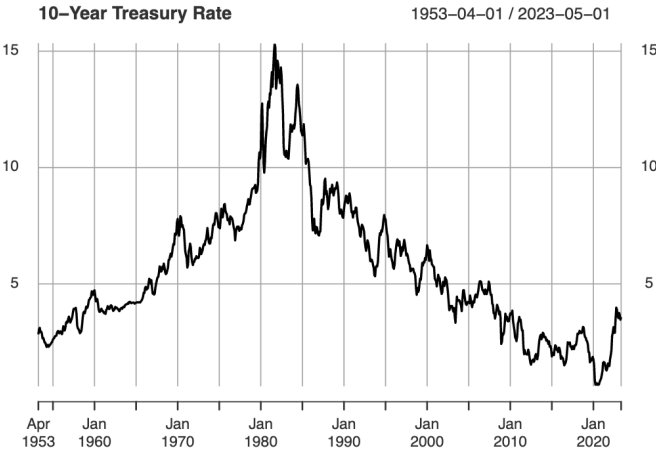
Figure 3.2: New R Markdown

MyFile

Julian Ludwig

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.
When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

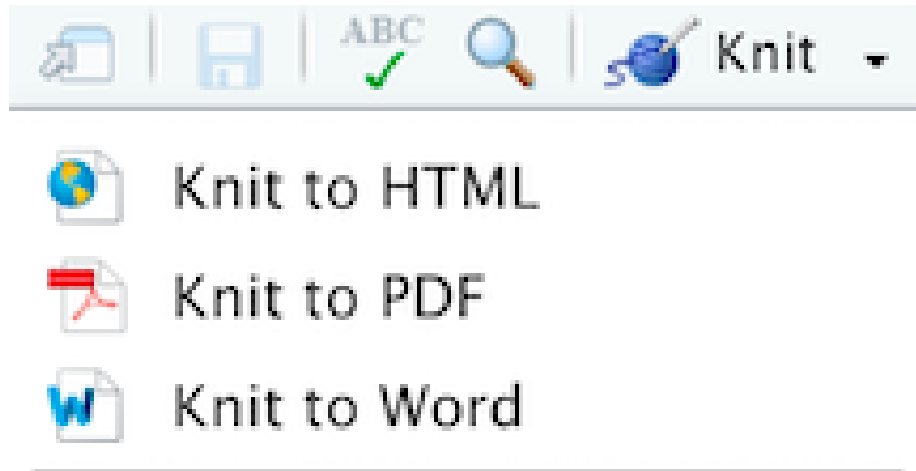


Including Plots

You can also embed plots, for example:

Figure 3.3: PDF File Produced with R Markdown

export your file as an HTML or Word document instead of a PDF document, which is convenient when designing a website or writing an app:



Troubleshooting

That's it! If everything worked as expected, you're good to go. If not, continue troubleshooting until it works.

Chapter 4



Getting Started

R is a programming language and free software environment for statistical computing and graphics supported by the R Foundation for Statistical Computing.

4.1 RStudio Interface

After launching RStudio on your computer, navigate to the menu bar and select “File,” then choose “New File,” and finally click on “R Script.” Alternatively, you can use the keyboard shortcut `Ctrl+Shift+N` (Windows/Linux) or `Cmd+Shift+N` (Mac) to create a new R script directly.

Once you have opened a new R script, you will notice that RStudio consists of four main sections:

1. **Source** (top-left): This section is where you write your R scripts. Also known as **do-files**, R scripts are files that contain a sequence of commands which can be executed either wholly or partially. To run a single line in your script, click on that line with your cursor and press the  **Run** button. However, to streamline your workflow, I recommend using the keyboard shortcut `Ctrl+Enter` (Windows/Linux) or `Cmd+Enter` (Mac) to run the line without reaching for the mouse. If you want to execute only a specific portion of a line, select that part and then press `Ctrl+Enter` or `Cmd+Enter`. To run all the commands in your R script, use the  **Source** button or the keyboard shortcut `Ctrl+Shift+Enter` (Windows/Linux) or `Cmd+Shift+Enter` (Mac).
2. **Console** (bottom-left): Located below the Source section, the Console is where R executes your commands. You can also directly type commands into the Console and see their output immediately. However, it is advisable to write commands in the R Script instead of the Console. By doing so,

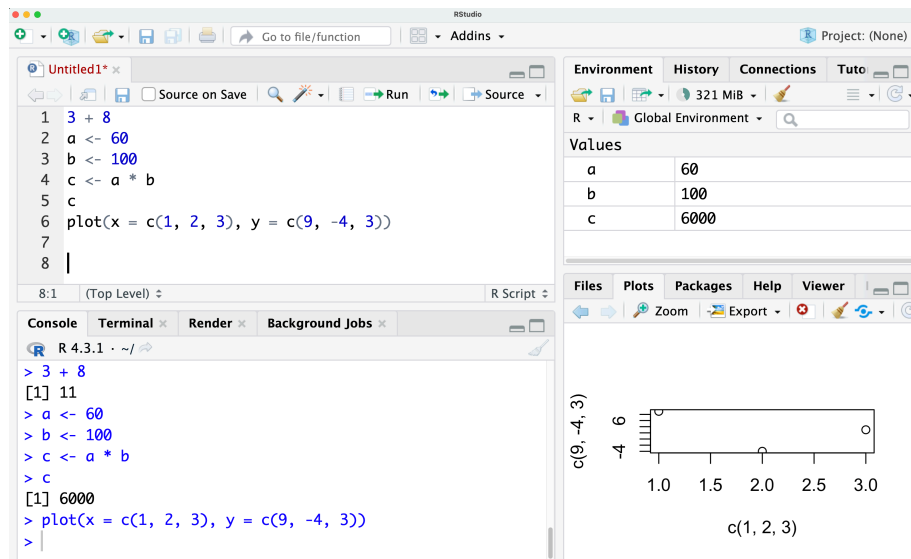


Figure 4.1: RStudio Interface

you can save the commands for future reference, enabling you to reproduce your results at a later time.

3. **Environment** (top-right): In the upper-right section, the Environment tab displays the current objects stored in memory, providing an overview of your variables, functions, and data frames. To create a variable, you can use the assignment operator `<-` (reversed arrow). Once a variable is created and assigned a numeric value, it can be utilized in arithmetic operations. For example:

```

a <- 60
a + 20

```

```
## [1] 80
```

4. **Files/Plots/Packages/Help/Viewer** (bottom-right): The bottom-right panel contains multiple tabs:
 - Files: displays your files and folders
 - Plots: displays your graphs
 - Packages: lets you manage your R packages
 - Help: provides help documentation
 - Viewer: lets you view local web content

These four main sections of RStudio provide a comprehensive environment for writing, executing, and managing your R code efficiently.

R Scripts


An R script is a text file that contains your R code. You can execute parts of the script by selecting a subset of commands and pressing Ctrl+Enter or Cmd+Enter, or run the entire script by pressing Ctrl+Shift+Enter.

Any text written after a hashtag (#) in an R Script is considered comments and is not executed as code. Comments are valuable for providing explanations or annotations for your commands, enhancing the readability and comprehensibility of your code.

```
# This is a comment in an R script
x <- 10 # Assign the value 10 to x
y <- 20 # Assign the value 20 to y
z <- x + y # Add x and y and assign the result to z
print(z) # Print the value of z
```

```
## [1] 30
```

The output displayed after two hashtags (##) in the example above: ## [1] 30, is not part of the actual R Script. Instead, it represents a line you would observe in your console when running the R Script. It showcases the result or value of the variable z in this case.

To facilitate working with lengthy R scripts, it is recommended to use a separate window. You can open a separate window by selecting  in the top-left corner.

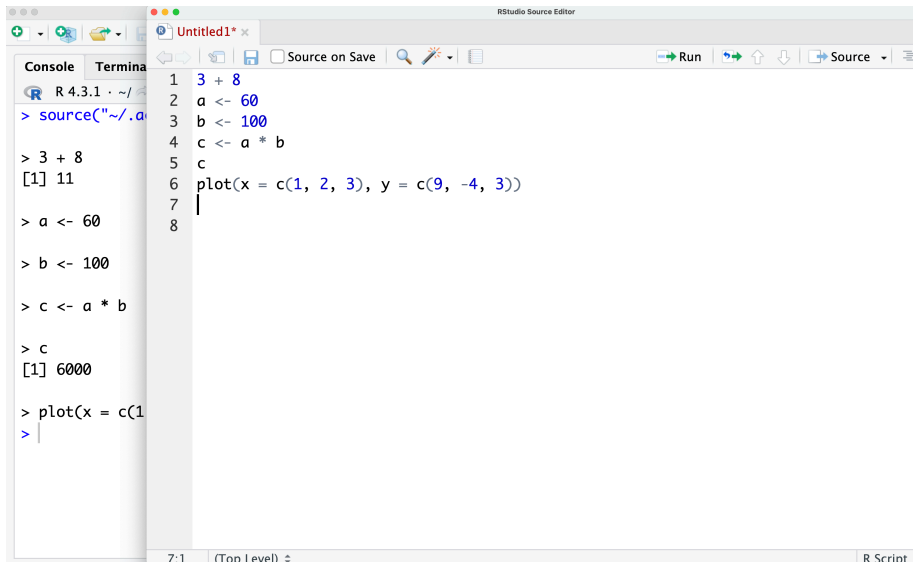


Figure 4.2: RStudio Interface with Separate R Script Window

When the R Script is in a separate window, you can easily switch between the R Script window and the Console/Environment/Plot Window by pressing Ctrl+Tab or Cmd+Tab. This allows for convenient navigation between different RStudio windows.

4.2 R Data Types and Structures

The R language supports a broad array of operations such as mathematical calculations, logical analyses, and text manipulation. However, the applicability of a function to a variable depends on the variable's data type. For instance, an arithmetic function to add two variables won't work if the variables store text.

R supports various data types, including numeric (`x <- 15`), character (`x <- "Hello!"`), and logical (`x <- TRUE` or `x <- FALSE`). In addition to single values (scalars), R allows variables to hold collections of numbers or strings using vectors, matrices, lists, or data frames. Advanced data structures such as tibbles, data tables, and `xts` objects provide additional features beyond traditional data frames.

In this section, we will explore the following data types:

1. **Scalar**: A single data element, such as a number or a character string.
2. **Vector**: A one-dimensional array that contains elements of the same type.
3. **Matrix** (`matrix`): A two-dimensional array with elements of the same type.
4. **List** (`list`): A one-dimensional array capable of storing various data types.
5. **Data Frame** (`data.frame`): A two-dimensional array that can accommodate columns of different types.
6. **Tibble** (`tibble`): An enhanced version of data frames, offering user-friendly features.
7. **Data Table** (`data.table`): An optimized data frame extension designed for speed and handling large datasets.
8. **Extensible Time Series** (`xts`): A time-indexed data frame specifically designed for time series data.

Understanding the data type of variables is crucial because it determines the operations and functions that can be applied to them.

It's worth noting that R provides so-called **wrapper functions**, which are functions that have the same name but perform different actions depending on the data object. These wrapper functions adapt their behavior based on the input data type, allowing for more flexible and intuitive programming. For example, the `summary()` function in R is a wrapper function. When applied to a numeric vector, it provides statistical summaries such as mean, median, and quartiles. However, when applied to a data frame, it gives a summary of each variable, including the minimum, maximum, and quartiles for numerical

variables, as well as counts and levels for categorical variables.

4.2.1 Scalar

Scalars in R are variables holding single objects. You can determine an object's type by applying the `class()` function to the variable.

Numbers, Characters, and Logical Values

```
# Numeric (a.k.a. Double)
w <- 5.5 # w is a decimal number.
class(w) # Returns "numeric".

# Integer
x <- 10L # The L tells R to store x as an integer instead of a decimal number.
class(x) # Returns "integer".

# Complex
u <- 3 + 4i # u is a complex number, where 3 is real and 4 is imaginary.
class(u) # Returns "complex".

# Character
y <- "Hello, World!" # y is a character string.
class(y) # Returns "character".

# Logical
z <- TRUE # z is a logical value.
class(z) # Returns "logical".

## [1] "numeric"
## [1] "integer"
## [1] "complex"
## [1] "character"
## [1] "logical"
```

An object's type dictates which functions can be applied. For example, mathematical functions are applicable to numbers but not characters:

```
# Mathematical operations
2 + 2 # Results in 4.
3 * 5 # Results in 15.
(1 + 2) * 3 # Results in 9 (parentheses take precedence).

# Logical operations
TRUE & FALSE # Results in FALSE (logical AND).
TRUE | FALSE # Results in TRUE (logical OR).
```

```
# String operations
paste("Hello", "World!") # Concatenates strings, results in "Hello World!".
nchar("Hello") # Counts characters in a string, results in 5.

## [1] 4
## [1] 15
## [1] 9
## [1] FALSE
## [1] TRUE
## [1] "Hello World!"
## [1] 5
```

Dates and Times

In this course, we also deal with data types specifically designed for storing date and time information:

```
# Date
v <- as.Date("2023-06-30") # v is a Date.
# The default input format is %Y-%m-%d, where
# - %Y is year in 4 digits,
# - %m is month with 2 digits, and
# - %d is day with 2 digits.
class(v) # Returns "Date".

## [1] "Date"

# POSIXct (Time)
t <- as.POSIXct("2023-06-30 18:47:10", tz = "CDT") # t is a POSIXct.

## Warning in strptime(xx, f, tz = tz): unknown timezone 'CDT'
## Warning in as.POSIXct.POSIXlt(x): unknown timezone 'CDT'
## Warning in strptime(x, f, tz = tz): unknown timezone 'CDT'
## Warning in as.POSIXct.POSIXlt(as.POSIXlt(x, tz, ...), tz, ...): unknown
## timezone 'CDT'

# The default input format is %Y-%m-%d %H:%M:%S, where
# - %H is hour out of 24,
# - %M is minute out of 60, and
# - %S is second out of 60.
# The tz input is the time zone, where CDT = Central Daylight Time.
class(t) # Returns "POSIXct".

## [1] "POSIXct" "POSIXt"
```

The default input format, `%Y-%m-%d` or `%Y-%m-%d %H:%M:%S`, can be changed by specifying a format input. The output format can be adjusted by applying the

`format()` function to the object:

```
# Date with custom input format:
v <- as.Date("April 6 -- 23", format = "%B %d -- %y")
v # Returns default output format: %Y-%m-%d.

## [1] "2023-04-06"

format(v, format = "%B %d, %Y") # Returns a custom output format: "%B %d, %Y".

## [1] "April 06, 2023"
```

The syntax for different date formats can be found by typing `?strptime` in the R console. Some of the most commonly used formats are outlined in the table below:

Specification	Description	Example
%a	Abbreviated weekday	Sun, Thu
%A	Full weekday	Sunday, Thursday
%b or %h	Abbreviated month	May, Jul
%B	Full month	May, July
%d	Day of the month, 0-31	27, 07
%j	Day of the year, 001-366	148, 188
%m	Month, 01-12	05, 07
%U	Week, 01-53, with Sunday as first day of the week	22, 27
%w	Weekday, 0-6, Sunday is 0	0, 4
%W	Week, 00-53, with Monday as first day of the week	21, 27
%x	Date, locale-specific	
%y	Year without century, 00-99	84, 05
%Y	Year with century, on input: 00 to 68 prefixed by 20, 69 to 99 prefixed by 19	1984, 2005
%C	Century	19, 20
%D	Date formatted %m/%d/%y	5/27/84
%u	Weekday, 1-7, Monday is 1	7, 4
%n	Newline on output or arbitrary whitespace on input	
%t	Tab on output or arbitrary whitespace on input	

Here are some example operations for `Date` objects:

```
# Date Operations
date1 <- as.Date("2023-06-30")
date2 <- as.Date("2023-01-01")

# Subtract dates to get the number of days between
days_between <- date1 - date2
days_between

## Time difference of 180 days
```

```
# Add days to a date
date_in_future <- date1 + 30
date_in_future
```

```
## [1] "2023-07-30"
```

4.2.2 Vector

In R, a vector is a homogeneous sequence of elements, meaning they must all be of the same basic type. As such, a vector can hold multiple numbers, but it cannot mix types, such as having both numbers and words. The function `c()` (for combine) can be used to create a vector:

```
# Numeric vector
numeric_vector <- c(1, 2, 3, 4, 5)
class(numeric_vector) # Returns "numeric".

# Character vector
character_vector <- c("Hello", "World", "!")
class(character_vector) # Returns "character".

# Logical vector
logical_vector <- c(TRUE, FALSE, TRUE)
class(logical_vector) # Returns "logical".
```

```
## [1] "numeric"
## [1] "character"
## [1] "logical"
```

The function `c()` can also be used to add elements to a vector:

```
# Add elements to existing vector:
x <- c(1, 2, 3)
x <- c(x, 4, 5, 6)
x
```

```
## [1] 1 2 3 4 5 6
```

The `seq()` function creates a sequence of numbers or dates:

```
# Create a sequence of numbers:
x <- seq(from = 1, to = 1.5, by = 0.1)
x
```

```
## [1] 1.0 1.1 1.2 1.3 1.4 1.5
```

```
# Create a sequence of dates:
x <- seq(from = as.Date("2004-05-01"), to = as.Date("2004-12-01"), by = "month")
x
```

```
## [1] "2004-05-01" "2004-06-01" "2004-07-01" "2004-08-01" "2004-09-01"
## [6] "2004-10-01" "2004-11-01" "2004-12-01"
```

Missing data is represented as NA (not available). The function `is.na()` indicates the elements that are missing and `anyNA()` returns TRUE if the vector contains any missing values:

```
x <- c(1, 2, NA, NA, 4, 9, 12, 5, 4, NA)
is.na(x)
```

```
## [1] FALSE FALSE TRUE TRUE FALSE FALSE FALSE FALSE FALSE TRUE
anyNA(x)
```

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

A generalization of logical vectors are **factors**, which are vectors that restrict entries to be one of predefined categories:

```
# Unordered factors, e.g. categories "Male" and "Female":
gender_vector <- c("Male", "Female", "Male", "Male", "Male", "Female", "Male")
factor_gender_vector <- factor(gender_vector)
factor_gender_vector
```

```
## [1] Male Female Male Male Male Female Male
## Levels: Female Male
```

```
# Ordered factors, e.g. categories with ordering Low < Medium < High:
temperature_vector <- c("High", "Low", "Low", "Low", "Medium", "Low", "Low")
factor_temperature_vector <- factor(temperature_vector,
                                   order = TRUE,
                                   levels = c("Low", "Medium", "High"))
factor_temperature_vector
```

```
## [1] High Low Low Low Medium Low Low
## Levels: Low < Medium < High
```

4.2.3 Matrix (matrix)

A matrix in R (`matrix`) is a two-dimensional array that extends atomic vectors, containing both rows and columns. The elements within a matrix must be of the same data type.

```
# Create a 3x3 numeric matrix, column-wise:
numeric_matrix <- matrix(1:9, nrow = 3, ncol = 3)
numeric_matrix
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```

class(numeric_matrix) # Returns "matrix".

## [1] "matrix" "array"
typeof(numeric_matrix) # Returns "numeric".

## [1] "integer"
# Create a 2x3 character matrix, row-wise:
character_matrix <- matrix(letters[1:6], nrow = 2, ncol = 3, byrow = TRUE)
character_matrix

##      [,1] [,2] [,3]
## [1,] "a"  "b"  "c"
## [2,] "d"  "e"  "f"

class(character_matrix) # Returns "matrix".

## [1] "matrix" "array"
typeof(character_matrix) # Returns "character".

## [1] "character"

To select specific elements, rows, or columns within a matrix, square brackets are
used. The cbind() and rbind() functions enable the combination of columns
and rows, respectively.

# Print element in the second row and first column:
character_matrix[2, 1]

## [1] "d"

# Print the second row:
character_matrix[2, ]

## [1] "d" "e" "f"

# Combine matrices:
x <- matrix(1:4, nrow = 2, ncol = 2)
y <- matrix(101:104, nrow = 2, ncol = 2)
rbind(x, y) # Combines matrices x and y row-wise.

##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
## [3,]  101  103
## [4,]  102  104

cbind(x, y) # Combines matrices x and y column-wise.

##      [,1] [,2] [,3] [,4]

```

```
## [1,]    1    3 101 103
## [2,]    2    4 102 104
```

4.2.4 List (list)

A list (list) in R serve as an ordered collection of objects. In contrast to vectors, elements within a list are not required to be of the same type. Moreover, some list elements may store multiple sub-elements, allowing for complex nested structures. For instance, a single element of a list might itself be a matrix or another list.

```
# List
my_list <- list(1, "a", TRUE, 1+4i,
               c(1, 2, 3), matrix(1:8, 2, 4), list("c",4))
names(my_list) <- c("num_1", "char_a", "log_T", "complex_1p4i",
                  "vec", "mat", "list")
my_list

## $num_1
## [1] 1
##
## $char_a
## [1] "a"
##
## $log_T
## [1] TRUE
##
## $complex_1p4i
## [1] 1+4i
##
## $vec
## [1] 1 2 3
##
## $mat
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
##
## $list
## $list[[1]]
## [1] "c"
##
## $list[[2]]
## [1] 4
class(my_list) # Returns "list".
```

```
## [1] "list"
```

The content of elements can be retrieved by using double square brackets:

```
# Select second element:
my_list[[2]]
```

```
## [1] "a"
```

```
# Select element named "mat":
my_list[["mat"]]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
```

4.2.5 Data Frame (`data.frame`)

A data frame (`data.frame`) in R resembles a matrix in its two-dimensional, rectangular structure. However, unlike a matrix, a data frame allows each column to contain a different data type. Therefore, within each column (or vector), the elements must be homogeneous, but different columns can accommodate distinct types. Typically, when importing data into R, the default object type used is a data frame.

```
# Vectors
student_names <- c("Anna", "Ella", "Sophia")
student_ages <- c(23, 21, 25)
student_grades <- c("A", "B", "A")
student_major <- c("Math", "Biology", "Physics")
```

```
# Data frame
students_df <- data.frame(name = student_names,
                          age = student_ages,
                          grade = student_grades,
                          major = student_major)

students_df
```

```
##      name age grade  major
## 1  Anna  23    A    Math
## 2  Ella  21    B Biology
## 3 Sophia  25    A Physics
```

```
class(students_df) # Returns "data.frame".
```

```
## [1] "data.frame"
```

Data frames are frequently used for data storage and manipulation in R. The following illustrates some common functions used on data frames:

```

# Access a column in the data frame
students_df$name

# Alternative way to access a column:
students_df[["name"]]

# Access second row in third column:
students_df[2, 3]

## [1] "Anna"    "Ella"    "Sophia"
## [1] "Anna"    "Ella"    "Sophia"
## [1] "B"

# When selecting just one column, data frame produces a vector
class(students_df[, 3])

# To avoid this, add drop = FALSE
class(students_df[, 3 , drop = FALSE])

## [1] "character"
## [1] "data.frame"

# Add a column to the data frame
students_df$gpa <- c(3.8, 3.5, 3.9)
students_df

##      name age grade  major gpa
## 1  Anna  23     A    Math 3.8
## 2  Ella  21     B Biology 3.5
## 3 Sophia 25     A Physics 3.9

# Subset the data frame
students_df[students_df$age > 22 & students_df$gpa > 3.6, ]

##      name age grade  major gpa
## 1  Anna  23     A    Math 3.8
## 3 Sophia 25     A Physics 3.9

# Number of columns and rows
ncol(students_df)
nrow(students_df)

# Column and row names
colnames(students_df)
rownames(students_df)

## [1] 5
## [1] 3
## [1] "name" "age"  "grade" "major" "gpa"

```

```
## [1] "1" "2" "3"

# Change column names
colnames(students_df) <- c("Name", "Age", "Grade", "Major", "GPA")
students_df

##      Name Age Grade  Major GPA
## 1  Anna  23    A    Math 3.8
## 2  Ella  21    B Biology 3.5
## 3 Sophia 25    A Physics 3.9

# Take a look at the data type of each column
str(students_df)

## 'data.frame':    3 obs. of  5 variables:
## $ Name : chr  "Anna" "Ella" "Sophia"
## $ Age  : num  23 21 25
## $ Grade: chr  "A" "B" "A"
## $ Major: chr  "Math" "Biology" "Physics"
## $ GPA  : num  3.8 3.5 3.9

# Take a look at the data in a separate window
View(students_df)
```

These examples illustrate just a few of the operations you can perform with data frames in R. With additional libraries like `dplyr`, `tidyr`, and `data.table`, more complex manipulations are possible.

4.2.6 Tibble (`tbl_df`)

A tibble (`tbl_df`) is a more convenient version of a data frame. It is part of the `tibble` package in the `tidyverse` collection of R packages. To use tibbles, you need to install the `tibble` package by executing `install.packages("tibble")` in your console. Don't forget to include `library("tibble")` at the beginning of your R script.

To create a tibble, you can use the `tibble()` function. Here's an example:

```
# Load R package
library("tibble")

# Create a new tibble
tib <- tibble(name = letters[1:3],
              id = sample(1:5, 3),
              age = sample(18:70, 3),
              sex = factor(c("M", "F", "F")))
tib

## # A tibble: 3 x 4
##   name    id  age sex
```



```
##   <chr> <int> <int> <fct>
## 1 a      2     37 M
## 2 b      5     54 F
## 3 c      1     51 F
```

```
class(tib)
```

```
## [1] "tbl_df"      "tbl"        "data.frame"
```

One advantage of tibbles is that they make it easy to calculate and create new columns. Here's an example:

```
tib <- tibble(tib, idvage = id/age)
tib
```

```
## # A tibble: 3 x 5
##   name    id  age sex  idvage
##   <chr> <int> <int> <fct> <dbl>
## 1 a      2    37 M    0.0541
## 2 b      5    54 F    0.0926
## 3 c      1    51 F    0.0196
```

Unlike regular data frames, tibbles allow non-standard column names. You can use special characters or numbers as column names. Here's an example:

```
tibble(`:`) = "smile", ` ` = "space", `2000` = "number")
```

```
## # A tibble: 1 x 3
##   `:` ` ` `2000`
##   <chr> <chr> <chr>
## 1 smile space number
```

Another way to create a tibble is with the `tribble()` function. It allows you to define column headings using formulas starting with `~` and separate entries with commas. Here's an example:

```
tribble(
  ~x, ~y, ~z,
  "a", 2, 3.6,
  "b", 1, 8.5
)
```

```
## # A tibble: 2 x 3
##   x      y      z
##   <chr> <dbl> <dbl>
## 1 a      2    3.6
## 2 b      1    8.5
```

For additional functions and a helpful cheat sheet on `tibble` and `dplyr`, you can refer to this [cheat sheet](#).

Tidyverse

The `tibble` package is part of the `tidyverse` environment, which is a collection of R packages with a shared design philosophy, grammar, and data structures. To install `tidyverse`, execute `install.packages("tidyverse")`, which includes `tibble`, `readr`, `dplyr`, `tidyr`, `ggplot2`, and more. Key functions in the `tidyverse` include `select()`, `filter()`, `mutate()`, `arrange()`, `count()`, `group_by()`, and `summarize()`. An interesting operator in the `tidyverse` is the pipe operator `%>`, which allows you to chain functions together in a readable and sequential manner. With the pipe operator, you can order the functions as they are applied, making your code more expressive and easier to understand. Here's an example:

```
library("tidyverse")

## -- Attaching core tidyverse packages ----- tidyverse 2.0.0 --
## v dplyr      1.1.2      v purrr      1.0.1
## v forcats    1.0.0      v readr      2.1.4
## v ggplot2    3.4.2      v stringr    1.5.0
## v lubridate  1.9.2      v tidyr      1.3.0
## -- Conflicts ----- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::first()  masks xts::first()
## x dplyr::lag()    masks stats::lag()
## x dplyr::last()   masks xts::last()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to be resolved.
x <- c(0.109, 0.359, 0.63, 0.996, 0.515, 0.142, 0.017, 0.829, 0.907)

# Apply several functions to x:
y <- round(exp(diff(log(x))), 1)
y

## [1] 3.3 1.8 1.6 0.5 0.3 0.1 48.8 1.1

# Perform the same computations using pipe operators:
y <- x %>% log() %>% diff() %>% exp() %>% round(1)
y

## [1] 3.3 1.8 1.6 0.5 0.3 0.1 48.8 1.1
```

By using the `%>` operator, each function is applied to the previous result, simplifying the code and improving its readability.

To delve deeper into the `tidyverse`, explore their official website: www.tidyverse.org. Another resource is the R-Bootcamp, available at r-bootcamp.netlify.app. Additionally, DataCamp provides a comprehensive skill track devoted to the `tidyverse`, named Tidyverse Fundamentals with R.

4.2.7 Data Table (`data.table`)

A data table (`data.table`) is similar to a data frame but with more advanced features for data manipulation. In fact, `data.table` and `tibble` can be considered competitors, with each offering enhancements over the standard data frame. While data tables offer high-speed functions and are optimized for large datasets, tibbles from the `tidyverse` are slower but are more user-friendly. The syntax used in `data.table` functions may seem esoteric, differing from that used in `tidyverse`. Like `tibble`, `data.table` is not a part of base R. It requires the installation of the `data.table` package via `install.packages("data.table")`, followed by `library("data.table")` at the beginning of your script.

To create a data table, you can use the `data.table()` function. Here's an example:

```
# Load R package
library("data.table")

##
## Attaching package: 'data.table'

## The following objects are masked from 'package:lubridate':
##
##      hour, isoweek, mday, minute, month, quarter, second, wday, week,
##      yday, year

## The following objects are masked from 'package:dplyr':
##
##      between, first, last

## The following object is masked from 'package:purrr':
##
##      transpose

## The following objects are masked from 'package:xts':
##
##      first, last

# Create a new data.table:
dt <- data.table(name = letters[1:3],
                 id = sample(1:5,3),
                 age = sample(18:70,3),
                 sex = factor(c("M", "F", "F")))
dt

##      name id age sex
## 1:    a  4  54  M
## 2:    b  5  46  F
## 3:    c  3  31  F
```

```
class(dt)
```

```
## [1] "data.table" "data.frame"
```

Columns in a data table can be referenced directly, and new variables can be created using the `:=` operator:

```
# Selection with data frame vs. data table:
df <- as.data.frame(dt) # create a data frame for comparison
df[df$sex == "M", ] # select with data frame
```

```
##   name id age sex
## 1    a  4  54  M
```

```
dt[sex == "M", ] # select with data table
```

```
##   name id age sex
## 1:    a  4  54  M
```

```
# Variable assignment with data frame vs. data table:
df$id_over_age <- df$id / df$age # assign with data frame
dt[, id_over_age := id / age] # assign with data table
```

You can select multiple variables with a list:

```
dt[, list(sex, age)]
```

```
##   sex age
## 1:   M  54
## 2:   F  46
## 3:   F  31
```

Multiple variables can be assigned simultaneously, where the LHS of the `:=` operator is a character vector of new variable names, and the RHS is a list of operations:

```
dt[, c("id_times_age", "id_plus_age") := list(id * age, id + age)]
dt
```

```
##   name id age sex id_over_age id_times_age id_plus_age
## 1:    a  4  54  M  0.07407407         216         58
## 2:    b  5  46  F  0.10869565         230         51
## 3:    c  3  31  F  0.09677419          93         34
```

Many operations in data analysis need to be done by group (e.g. calculating average unemployment by year). In such cases, data table introduces a third dimension to perform these operations. Specifically, the data table syntax is `DT[i,j,by]` with options to

- subset rows using `i` (which rows?),
- manipulate columns with `j` (what to do?), and

- group according to `by` (grouped by what?).

Here is an example:

```
# Produce table with average age by sex:
dt[, mean(age), by = sex]

##      sex    V1
## 1:    M 54.0
## 2:    F 38.5

# Do the same but name the columns "Gender" and "Age by Gender":
dt[, list(`Age by Gender` = mean(age)), by = list(Gender = sex)]

##      Gender Age by Gender
## 1:      M           54.0
## 2:      F           38.5

# Assign a new variable with average age by sex named "age_by_sex":
dt[, age_by_sex := mean(age), by = sex]
dt

##      name id age sex id_over_age id_times_age id_plus_age age_by_sex
## 1:    a  4  54  M  0.07407407           216           58      54.0
## 2:    b  5  46  F  0.10869565           230           51      38.5
## 3:    c  3  31  F  0.09677419            93           34      38.5
```

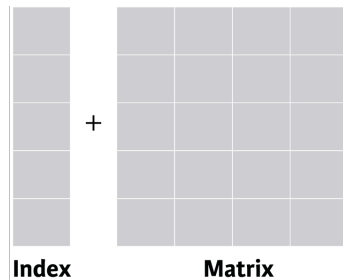
For additional information about data tables and their powerful features, check out the Intro to Data Table documentation and this cheat sheet for `data.table` functions. Furthermore, DataCamp provides several courses on `data.table`, such as:

- Data Manipulation with `data.table` in R
- Joining Data with `data.table` in R
- Time Series with `data.table` in R

4.2.8 Extensible Time Series (`xts`)

`xts` (extensible time series) objects are specialized data structures designed for time series data. These are datasets where each observation corresponds to a specific timestamp. `xts` objects attach an index to the data, aligning each data point with its associated time. This functionality simplifies data manipulation and minimizes potential errors:

The index attached to an `xts` object is usually a `Date` or `POSIXct` vector, maintaining the data in chronological order from earliest to latest. If you wish to sort data (such as stock prices) by another variable (like trade volume), you'll first need to convert the `xts` object back to a data frame, as `xts` objects preserve the time order. `xts` objects are built upon `zoo` objects (Zeileis' Ordered Observations), another class of time-indexed data structures. `xts` objects enhance these

Figure 4.3: Data with Index. *Source:* DataCamp.

base structures by providing additional features.

Like `tibble` and `data.table`, `xts` is not included in base R. To use it, you need to install the `xts` package using `install.packages("xts")`, then include `library("xts")` at the start of your script.

To create an `xts` object, use the `xts()` function which associates data with a time index (`order.by = time_index`):

```
# Load R package
library("xts")

# Create a new xts object from a matrix:
data <- matrix(1:4, ncol = 2, nrow = 2, dimnames = list(NULL, c("a", "b")))
data

##           a b
## [1,]  1 3
## [2,]  2 4

time_index <- as.Date(c("2020-06-01", "2020-07-01"))
time_index

## [1] "2020-06-01" "2020-07-01"

dxts <- xts(x = data, order.by = time_index)
dxts

##           a b
## 2020-06-01 1 3
## 2020-07-01 2 4

class(dxts)

## [1] "xts" "zoo"

tclass(dxts)

## [1] "Date"
```

```
# Extract time index
index(dxts)

## [1] "2020-06-01" "2020-07-01"

# Extract data without time index
coredata(dxts)

##      a b
## [1,] 1 3
## [2,] 2 4
```

To delve deeper into `xts` and `zoo` objects, consider reading the guides *Manipulating Time Series Data in R with xts & zoo* and *Time Series in R: Quick Reference*. Additionally, DataCamp provides in-depth courses on these topics:

- *Manipulating Time Series Data with xts and zoo in R*
- *Importing and Managing Financial Data in R*

If you're working within the `tidyverse` environment, the R package `tidyquant` offers seamless integration with `xts` and `zoo`. Lastly, this handy cheat sheet provides a quick reference on `xts` and `zoo` functions.

4.3 Importing Data in R

R is a software specialized for data analysis. To analyze data using R, the data must first be imported into the R environment. R offers robust functionality for importing data in a variety of formats. This section explores how to import data from CSV, TSV, and Excel files. Before we dive into specifics, let's ensure RStudio can locate the data stored on your computer.

4.3.1 Working Directory

The working directory in R is the folder where R starts when it's looking for files to read or write. If you're not sure where your current working directory is, you can use the `getwd()` (get working directory) command in R to find out:

```
getwd()

## [1] "/Users/julianludwig/Library/CloudStorage/Dropbox/Economics/teaching/4300_2023/module-01"
```

To change your working directory, use the `setwd()` (set working directory) function:

```
setwd("your/folder/path")
```

Be sure to replace `"your/folder/path"` with the actual path to your folder.

When your files are stored in the same directory as your working directory, defined using the `setwd()` function, you can directly access these files by their

names. For instance, `read_csv("yieldcurve.csv")` will successfully read the file if “yieldcurve.csv” is in the working directory. If the file is located in a subfolder within the working directory, for example a folder named `files`, you would need to specify the folder in the file path when calling the file: `read_csv("files/yieldcurve.csv")`.

To find out the folder path for a specific file or folder on your computer, you can follow these steps:

For Windows:

1. Navigate to the folder using the File Explorer.
2. Once you are in the folder, click on the address bar at the top of the File Explorer window. The address bar will now show the full path to the folder. This is the path you can set in R using the `setwd()` function.

An example of a folder path on Windows might look like this: `C:/Users/YourName/Documents/R`.

For MacOS:

1. Open Finder and navigate to the folder.
2. Once you are in the folder, Command-click (or right-click and hold, if you have a mouse) on the folder’s name at the top of the Finder window. A drop-down menu will appear showing the folder hierarchy.
3. Hover over each folder in the hierarchy to show the full path, then copy this path.

An example of a folder path on macOS might look like this: `/Users/YourName/Documents/R`.

Remember, when setting the working directory in R, you need to use forward slashes (/) in the folder path, even on Windows where the convention is to use backslashes (\).

4.3.2 CSV (Comma Separated Values)

CSV (Comma Separated Values) is a common file format used to store tabular data. As the name suggests, the values in each row of a CSV file are separated by commas. Here’s an example of how data is stored in a CSV file:

- Male,8,100,3
- Female,9,20,3

In this section, we’ll demonstrate how to import a CSV file using real-world Treasury yield curve rates data.

Import Yield Curve Data in CSV Format

To obtain the yield curve data, follow these steps:

1. Visit the U.S. Treasury’s data center by clicking [here](#).

2. Click on “Data” in the menu bar, then select “Daily Treasury Par Yield Curve Rates.”
3. On the data page, select “Download CSV” to obtain the yield curve data for the current year.
4. To access all the yield curve data since 1990, choose “All” under the “Select Time Period” option, and click “Apply.” Please note that when selecting all periods, the “Download CSV” button may not be available.
5. To manually save the data as a CSV file, you can copy the data by selecting it and using the Ctrl+C (or Cmd+C) command. Open an Excel file, and use the Ctrl+V (or Cmd+V) command to paste the data into the Excel file.
6. Save the Excel file as a CSV file named ‘yieldcurve.csv’ in a location of your choice, ensuring that it is saved in a familiar folder for easy access.

Next, install and load the `readr` package. Run `install.packages("readr")` in the console and include the package at the top of your R script. You can then use the `read_csv()` or `read_delim()` function to import the yield curve data:

```
# Load the package
library("readr")

# Import CSV file
yc <- read_csv(file = "files/yieldcurve.csv", col_names = TRUE)

## Warning: One or more parsing issues, call `problems()` on your data frame for details,
## e.g.:
##   dat <- vroom(...)
##   problems(dat)

## Rows: 8382 Columns: 14
## -- Column specification -----
## Delimiter: ","
## chr (6): Date, 1 Mo, 2 Mo, 4 Mo, 20 Yr, 30 Yr
## dbl (8): 3 Mo, 6 Mo, 1 Yr, 2 Yr, 3 Yr, 5 Yr, 7 Yr, 10 Yr
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

# Import CSV file using the read_delim() function
yc <- read_delim(file = "files/yieldcurve.csv", col_names = TRUE, delim = ",")

## Warning: One or more parsing issues, call `problems()` on your data frame for details,
## e.g.:
##   dat <- vroom(...)
##   problems(dat)

## Rows: 8382 Columns: 14
## -- Column specification -----
```

```
## Delimiter: ","
## chr (6): Date, 1 Mo, 2 Mo, 4 Mo, 20 Yr, 30 Yr
## dbl (8): 3 Mo, 6 Mo, 1 Yr, 2 Yr, 3 Yr, 5 Yr, 7 Yr, 10 Yr
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

In the code snippets above, the `read_csv()` and `read_delim()` functions from the `readr` package are used to import a CSV file named “yieldcurve.csv”. The `col_names = TRUE` argument indicates that the first row of the CSV file contains column names. The `delim = ","` argument specifies that the columns are separated by commas, which is the standard delimiter for CSV (*Comma Separated Values*) files. Either one of the two functions can be used to read the CSV file and store the data in the variable `yc` for further analysis.

To inspect the first few rows of the data, print the `yc` object in the console. For an overview of the entire dataset, use the `View()` function, which provides an interface similar to viewing the CSV file in Microsoft Excel:

```
# Display the data
```

```
yc
```

```
## # A tibble: 8,382 x 14
##   Date      `1 Mo` `2 Mo` `3 Mo` `4 Mo` `6 Mo` `1 Yr` `2 Yr` `3 Yr` `5 Yr` `7 Yr`
##   <chr>    <chr> <chr>  <dbl> <chr>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 01/02/~ N/A    N/A    7.83 N/A    7.89  7.81  7.87  7.9   7.87  7.98
## 2 01/03/~ N/A    N/A    7.89 N/A    7.94  7.85  7.94  7.96  7.92  8.04
## 3 01/04/~ N/A    N/A    7.84 N/A    7.9   7.82  7.92  7.93  7.91  8.02
## 4 01/05/~ N/A    N/A    7.79 N/A    7.85  7.79  7.9   7.94  7.92  8.03
## 5 01/08/~ N/A    N/A    7.79 N/A    7.88  7.81  7.9   7.95  7.92  8.05
## 6 01/09/~ N/A    N/A    7.8   N/A    7.82  7.78  7.91  7.94  7.92  8.05
## 7 01/10/~ N/A    N/A    7.75 N/A    7.78  7.77  7.91  7.95  7.92  8
## 8 01/11/~ N/A    N/A    7.8   N/A    7.8   7.77  7.91  7.95  7.94  8.01
## 9 01/12/~ N/A    N/A    7.74 N/A    7.81  7.76  7.93  7.98  7.99  8.07
## 10 01/16/~ N/A    N/A    7.89 N/A    7.99  7.92  8.1   8.13  8.11  8.18
## # i 8,372 more rows
## # i 3 more variables: `10 Yr` <dbl>, `20 Yr` <chr>, `30 Yr` <chr>
```

```
# Display the data in a spreadsheet-like format
```

```
View(yc)
```

Both the `read_csv()` and `read_delim()` functions convert the CSV file into a tibble (`tbl_df`), a modern version of the R data frame discussed in Section 4.2.6. Remember, a data frame stores data in separate columns, each of which must be of the same data type. Use the `class(yc)` function to check the data type of the entire dataset, and `sapply(yc, class)` to check the data type of each column:

```
# Check the data type of the entire dataset
class(yc)

## [1] "spec_tbl_df" "tbl_df"      "tbl"        "data.frame"

# Check the data type of each column
sapply(yc, class)

##      Date      1 Mo      2 Mo      3 Mo      4 Mo      6 Mo
## "character" "character" "character" "numeric" "character" "numeric"
##      1 Yr      2 Yr      3 Yr      5 Yr      7 Yr     10 Yr
## "numeric"   "numeric"   "numeric"   "numeric"   "numeric"   "numeric"
##      20 Yr     30 Yr
## "character" "character"
```

When importing data in R, it's possible that R assigns incorrect data types to some columns. For example, the `Date` column is treated as a character column even though it contains dates, and the `30 Yr` column is treated as a character column even though it contains interest rates. To address this issue, you can convert the first column to a date type and the remaining columns to numeric data types using the following three steps:

1. Replace “N/A” with NA, which represents missing values in R. This step is necessary because R doesn't recognize “N/A”, and if a column includes “N/A”, R will consider it as a character vector instead of a numeric vector.

```
yc[yc == "N/A"] <- NA
```

2. Convert all yield columns to numeric data types:

```
yc[, -1] <- sapply(yc[, -1], as.numeric)
```

The `as.numeric()` function converts a data object into a numeric type. In this case, it converts columns with character values like “3” and “4” into the numeric values 3 and 4. The `sapply()` function applies the `as.numeric()` function to each of the selected columns. This converts all the interest rates to numeric data types.

3. Convert the date column to a date object, recognizing that the date format is Month/Day/Year or `%m/%d/%Y`:

```
# Check the date format
head(yc$Date)

## [1] "01/02/1990" "01/03/1990" "01/04/1990" "01/05/1990" "01/08/1990"
## [6] "01/09/1990"

# Convert to date format
yc$Date <- as.Date(yc$Date, format = "%m/%d/%Y")
head(yc$Date)
```

```
## [1] "1990-01-02" "1990-01-03" "1990-01-04" "1990-01-05" "1990-01-08"
## [6] "1990-01-09"
```

Hence, we have successfully imported the yield curve data and performed the necessary conversions to ensure that all columns are in their correct formats:

```
yc
```

```
## # A tibble: 8,382 x 14
##   Date      `1 Mo` `2 Mo` `3 Mo` `4 Mo` `6 Mo` `1 Yr` `2 Yr` `3 Yr` `5 Yr`
##   <date>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 1990-01-02    NA    NA  7.83    NA  7.89  7.81  7.87  7.9   7.87
## 2 1990-01-03    NA    NA  7.89    NA  7.94  7.85  7.94  7.96  7.92
## 3 1990-01-04    NA    NA  7.84    NA  7.9   7.82  7.92  7.93  7.91
## 4 1990-01-05    NA    NA  7.79    NA  7.85  7.79  7.9   7.94  7.92
## 5 1990-01-08    NA    NA  7.79    NA  7.88  7.81  7.9   7.95  7.92
## 6 1990-01-09    NA    NA  7.8     NA  7.82  7.78  7.91  7.94  7.92
## 7 1990-01-10    NA    NA  7.75    NA  7.78  7.77  7.91  7.95  7.92
## 8 1990-01-11    NA    NA  7.8     NA  7.8   7.77  7.91  7.95  7.94
## 9 1990-01-12    NA    NA  7.74    NA  7.81  7.76  7.93  7.98  7.99
## 10 1990-01-16   NA    NA  7.89    NA  7.99  7.92  8.1   8.13  8.11
## # i 8,372 more rows
## # i 4 more variables: `7 Yr` <dbl>, `10 Yr` <dbl>, `20 Yr` <dbl>, `30 Yr` <dbl>
```

Here, `<dbl>` stands for double, which is the R data type for decimal numbers, also known as numeric type. Converting the yield columns to `dbl` ensures that the values are treated as numeric and can be used for calculations, analysis, and visualization.

Plot Historical Trends in Treasury Yields

Let's use the `plot()` function to visualize the imported yield curve data. In this case, we will plot the 3-month Treasury rate over time, using the `Date` column as the x-axis and the `3 Mo` column as the y-axis:

```
# Plot the 3-month Treasury rate over time
plot(x = yc$Date, y = yc$`3 Mo`, type = "l",
     xlab = "Date", ylab = "%", main = "3-Month Treasury Rate")
```

In the code snippet above, `plot()` is the R function used to create the plot. It takes several arguments to customize the appearance and behavior of the plot:

- `x` represents the data to be plotted on the x-axis. In this case, it corresponds to the `Date` column from the yield curve data.
- `y` represents the data to be plotted on the y-axis. Here, it corresponds to the `3 Mo` column, which represents the 3-month Treasury rate.
- `type = "l"` specifies the type of plot to create. In this case, we use `"l"` to create a line plot.
- `xlab = "Date"` sets the label for the x-axis to "Date".

- `ylab = "%"` sets the label for the y-axis to “%”.
- `main = "3-Month Treasury Rate"` sets the title of the plot to “3-Month Treasury Rate”.

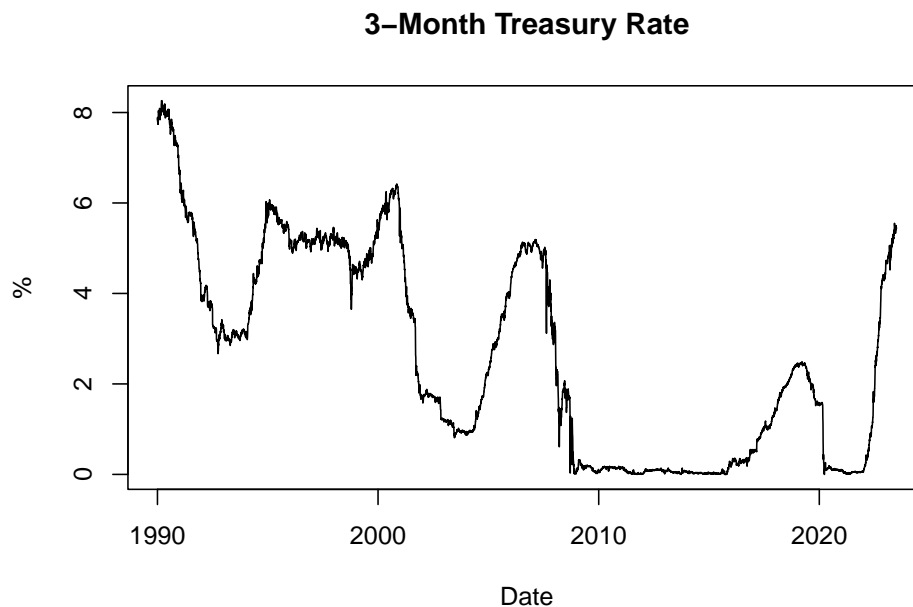


Figure 4.4: 3-Month Treasury Rate

The resulting plot, shown in Figure 4.4, displays the historical evolution of the 3-month Treasury rate since 1990. It allows us to observe how interest rates have changed over time, with low rates often observed during recessions and high rates during boom periods. Recessions are typically characterized by reduced borrowing and investment activities, leading to decreased demand for credit and lower interest rates. Conversely, boom periods are associated with strong economic growth and increased credit demand, which can drive interest rates upward.

Furthermore, inflation plays a significant role in influencing interest rates through the Fisher effect. When inflation is high, lenders and investors are concerned about the diminishing value of money over time. To compensate for the erosion of purchasing power, lenders typically demand higher interest rates on loans. These higher interest rates reflect the expectation of future inflation and act as a safeguard against the declining value of the money lent. Conversely, when inflation is low, lenders may offer lower interest rates due to reduced concerns about the erosion of purchasing power.

Plot Yield Curve

Next, let's plot the yield curve. The yield curve is a graphical representation of the relationship between the interest rates (yields) and the time to maturity of a bond. It provides insights into market expectations regarding future interest rates and economic conditions.

To plot the yield curve, we will select the most recently available data from the dataset, which corresponds to the last row. We will extract the interest rates as a numeric vector and the column names (representing the time to maturity) as labels for the x-axis:

```
# Extract the interest rates of the last row
yc_most_recent_data <- as.numeric(last(yc[, -1]))
yc_most_recent_data

## [1] 5.24 5.39 5.43 5.50 5.47 5.40 4.87 4.49 4.13 3.97 3.81 4.06 3.85

# Extract the column names of the last row
yc_most_recent_labels <- colnames(last(yc[, -1]))
yc_most_recent_labels

## [1] "1 Mo" "2 Mo" "3 Mo" "4 Mo" "6 Mo" "1 Yr" "2 Yr" "3 Yr" "5 Yr"
## [10] "7 Yr" "10 Yr" "20 Yr" "30 Yr"

# Plot the yield curve
plot(x = yc_most_recent_data, xaxt = 'n', type = "o", pch = 19,
     xlab = "Time to Maturity", ylab = "Treasury Rate in %",
     main = paste("Yield Curve on", format(last(yc$Date), format = '%B %d, %Y'))),
axis(side = 1, at = seq(1, length(yc_most_recent_labels), 1),
     labels = yc_most_recent_labels)
```

In the code snippet above, `plot()` is the R function used to create the yield curve plot. Here are the key inputs and arguments used in the function:

- `x = yc_most_recent_data` represents the interest rates of the most recent yield curve data, which will be plotted on the x-axis.
- `xaxt = 'n'` specifies that no x-axis tick labels should be displayed initially. This is useful because we will customize the x-axis tick labels separately using the `axis()` function.
- `type = "o"` specifies that the plot should be created as a line plot with points. This will display the yield curve as a connected line with markers at each data point.
- `pch = 19` sets the plot symbol to a solid circle, which will be used as markers for the data points on the yield curve.
- `xlab = "Time to Maturity"` sets the label for the x-axis to “Time to Maturity”, indicating the variable represented on the x-axis.
- `ylab = "Treasury Rate in %"` sets the label for the y-axis to “Treasury Rate in %”, indicating the variable represented on the y-axis.

- `main = paste("Yield Curve on", format(last(yc$Date), format = '%B %d, %Y'))` sets the title of the plot to “Yield Curve on” followed by the date of the most recent yield curve data.

Additionally, the `axis()` function is used to customize the x-axis tick labels. It sets the tick locations using `at = seq(1, length(yc_most_recent_labels), 1)` to evenly space the ticks along the x-axis. The `labels = yc_most_recent_labels` argument assigns the column names of the last row (representing maturities) as the tick labels on the x-axis.

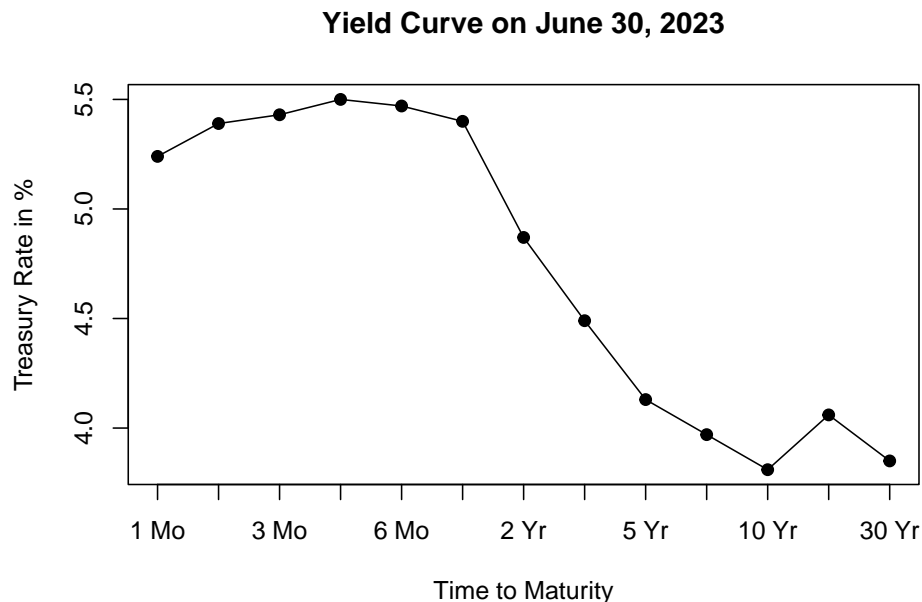


Figure 4.5: Yield Curve on June 30, 2023

The resulting plot, shown in Figure 4.5, depicts the yield curve based on the most recent available data, allowing us to visualize the relationship between interest rates and the time to maturity. The x-axis represents the different maturities of the bonds, while the y-axis represents the corresponding treasury rates.

Analyzing the shape of the yield curve can provide insights into market expectations and can be useful for assessing economic conditions and making investment decisions. The yield curve can take different shapes, such as upward-sloping (normal), downward-sloping (inverted), or flat, each indicating different market conditions and expectations for future interest rates.

An upward-sloping yield curve, where longer-term interest rates are higher than shorter-term rates, is often seen during periods of economic expansion. This shape suggests that investors expect higher interest rates in the future as the economy grows and inflationary pressures increase. It reflects an optimistic

outlook for economic conditions, as borrowing and lending activity are expected to be robust.

In contrast, a downward-sloping or inverted yield curve, where shorter-term interest rates are higher than longer-term rates, is often considered a predictor of economic slowdown or recession. This shape suggests that investors anticipate lower interest rates in the future as economic growth slows and inflationary pressures decrease. It reflects a more cautious outlook for the economy, as investors seek the safety of longer-term bonds amid expectations of lower returns and potential economic downturn.

Inflation expectations also influence the shape of the yield curve. When there are high inflation expectations for the long term, the yield curve tends to slope upwards. This occurs because lenders demand higher interest rates for longer maturities to compensate for anticipated inflation. However, when there is currently high inflation but expectations are that the central bank will successfully control inflation in the long term, the yield curve may slope downwards. In this case, long-term interest rates are lower than short-term rates, as average inflation over the long term is expected to be lower than in the short term.

4.3.3 TSV (Tab Separated Values)

TSV (Tab Separated Values) is a common file format used to store tabular data. As the name suggests, the values in each row of a TSV file are separated by tabs. Here's an example of how data is stored in a TSV file:

- Male 8 100 3
- Female 9 20 3

In this section, we'll demonstrate how to import a TSV file using real-world consumer survey data collected by the University of Michigan.

Import Michigan Consumer Survey Data in TSV Format

To obtain the Michigan consumer survey data, follow these steps:

1. Visit the website of University of Michigan's surveys of consumers by clicking [here](#).
2. Click on "DATA" in the menu bar, then select "Time Series."
3. On the data page, under Table, select "All: All Tables (Tab-delimited or CSV only)" to obtain the consumer survey data on all topics.
4. To access all the consumer survey data since 1978, type "1978" under the "Oldest Year" option.
5. Click on "Tab-Delimited (Excel)" under the "format" option.
6. Save the TSV file in a location of your choice, ensuring that it is saved in a familiar folder for easy access.

The dataset contains 360 variables with coded column names such as `ics_inc31` or `pago_dk_all`. To understand the meaning of these columns, you can visit the

same website here and click on SURVEY INFORMATION. From there, select the Time-Series Variable Codebook which is a PDF document that provides detailed explanations for all the column names. By referring to this codebook, you can gain a better understanding of the variables and their corresponding meanings in the dataset.

Next, to import the consumer survey data, you need to install and load the `readr` package if you haven't done so already. Once the package is loaded, you can use either the `read_tsv()` or `read_delim()` function to read the TSV (Tab-Separated Values) file.

```
# Load the package
library("readr")

# Import TSV file
cs <- read_tsv(file = "files/sca-tableall-on-2023-Jul-01.tsv", skip = 1)

## New names:
## Rows: 545 Columns: 360
## -- Column specification
## ----- Delimiter: "\t" dbl
## (359): Month, yyyy, ics_all, ics_inc31, ics_inc32, ics_inc33, ics_a1834,... lgl
## (1): ...360
## i Use `spec()` to retrieve the full column specification for this data. i
## Specify the column types or set `show_col_types = FALSE` to quiet this message.
## * `` -> `...360`

# Import TSV file using the read_delim() function
cs <- read_delim(file = "files/sca-tableall-on-2023-Jul-01.tsv", skip = 1, col_names = TRUE, delim = "\t")

## New names:
## Rows: 545 Columns: 360
## -- Column specification
## ----- Delimiter: "\t" dbl
## (359): Month, yyyy, ics_all, ics_inc31, ics_inc32, ics_inc33, ics_a1834,... lgl
## (1): ...360
## i Use `spec()` to retrieve the full column specification for this data. i
## Specify the column types or set `show_col_types = FALSE` to quiet this message.
## * `` -> `...360`
```

In the provided code snippets, the `file` input specifies the file path or URL of the TSV file to be imported. The `skip` input is used to specify the number of rows to skip at the beginning of the file. In this case, `skip = 1` indicates that the first line of the TSV file, which contains the title “All Tables”, should be skipped. The `col_names` input is set to `TRUE` to indicate that the second line of the TSV file (after skipping 1 row) contains the column names. Lastly, the `delim` input is set to `"\t"` to specify that the columns in the TSV file are separated by tabs, which is the standard delimiter for TSV (*Tab* Separated

Values) files.

Note that if the file is neither CSV nor TSV, but rather has an exotic format where columns are separated by a different character that is neither a comma nor a tab, such as “/”, you can use the `read_delim()` function with the `delim = "/"` argument to specify the custom delimiter.

To inspect the first few rows of the data, print the `cs` object in the console. For an overview of the entire dataset, execute `View(cs)`.

```
# Display the data
```

```
cs
```

```
## # A tibble: 545 x 360
##   Month  yyyy  ics_all  ics_inc31  ics_inc32  ics_inc33  ics_a1834  ics_a3554
##   <dbl> <dbl>   <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
## 1     1   1978    83.7      NA      NA      NA      93.7    86.7
## 2     2   1978    84.3      NA      NA      NA      99.7    82.3
## 3     3   1978    78.8      NA      NA      NA      91.7    76.8
## 4     4   1978    81.6      NA      NA      NA      91.8    79.7
## 5     5   1978    82.9      NA      NA      NA      95.1    78.9
## 6     6   1978     80      NA      NA      NA      91.7    75.7
## 7     7   1978    82.4      NA      NA      NA      92.2    78.4
## 8     8   1978    78.4      NA      NA      NA      87.8    77.2
## 9     9   1978    80.4      NA      NA      NA      86.6    83.9
## 10    10   1978    79.3      NA      NA      NA      90.6    76.7
## # i 535 more rows
## # i 352 more variables: ics_a5597 <dbl>, ics_ne <dbl>, ics_nc <dbl>,
## #   ics_s <dbl>, ics_w <dbl>, icc_all <dbl>, ice_all <dbl>, pago_f_all <dbl>,
## #   pago_s_all <dbl>, pago_u_all <dbl>, pago_dk_all <dbl>, pago_r_all <dbl>,
## #   pagorn_hy_all <dbl>, pagorn_ha_all <dbl>, pagorn_ld_all <dbl>,
## #   pagorn_ly_all <dbl>, pagorn_hp_all <dbl>, pagorn_la_all <dbl>,
## #   pagorn_hd_all <dbl>, pagorn_ny_all <dbl>, pagorn_nad_all <dbl>, ...
```

Use `sapply(cs, class)` to check the data type of each column, to make sure all columns are indeed numeric:

```
# Check the data type of each column
```

```
table(sapply(cs, class))
```

```
##
## logical numeric
##      1      359
```

Here, since there are 360 columns, the `summary()` function is applied, which reveals that there are 359 numerical columns, and 1 logical column, which makes sense.

Instead of a date column, the consumer survey has a year (`yyyy`) and a month (`Month`) column. To create a date column from the year and month columns,

combine them with the `paste()` function to create a date format of the form Year-Month-Day or %Y-%m-%d:

```
# Create date column
cs$Date <- as.Date(paste(cs$yyyy, cs$Month, "01", sep = "-"))
head(cs$Date)

## [1] "1978-01-01" "1978-02-01" "1978-03-01" "1978-04-01" "1978-05-01"
## [6] "1978-06-01"
```

Plot Consumer Indices

The Michigan Consumer Survey consists of a wide range of survey responses from a sample of households collected every month. These survey responses are gathered to produce indices about how consumers feel each period. The University of Michigan produces three main indices: the Index of Consumer Confidence (ICC), the Index of Current Economic Conditions (ICE), and the Index of Consumer Sentiment (ICS). These indices are designed to measure different aspects of consumer attitudes and perceptions regarding the economy.

1. **Index of Consumer Confidence (ICC):** The ICC reflects consumers' expectations about future economic conditions and their overall optimism or pessimism. It is based on consumers' assessments of their future financial prospects, job availability, and economic outlook. A higher ICC value indicates greater consumer confidence and positive expectations for the economy.
2. **Index of Current Economic Conditions (ICE):** The ICE assesses consumers' perceptions of the current economic environment. It reflects their evaluations of their personal financial situation, job security, and their perception of whether it is a good time to make major purchases. The ICE provides insights into the current economic conditions as perceived by consumers.
3. **Index of Consumer Sentiment (ICS):** The ICS combines both the ICC and ICE to provide an overall measure of consumer sentiment. It takes into account consumers' expectations for the future as well as their assessment of the present economic conditions. The ICS is often used as an indicator of consumer behavior and their likelihood of making purchases and engaging in economic activities.

These indices are calculated based on survey responses from a sample of households, and they serve as important indicators of consumer sentiment and economic trends. They are widely followed by economists, policymakers, and financial markets as they provide valuable insights into consumers' attitudes and perceptions, which can impact their spending behavior and overall economic activity.

Let's use the `plot()` function to visualize the imported Michigan consumer

survey data. In this case, we will plot the three key indices: ICC, ICE, and ICS over time, using the Date column as the x-axis and the three indices as the y-axis:

```
# Plot ICC, ICE, and ICS over time
plot(x = cs$Date, y = cs$icc_all, type = "l", col = 5, lwd = 3, ylim = c(40, 140),
     xlab = "Date", ylab = "Index", main = "Key Indices of the Michigan Consumer Survey",
     lines(x = cs$Date, y = cs$ice_all, col = 2, lwd = 2)
lines(x = cs$Date, y = cs$ics_all, col = 1, lwd = 1.5)
legend(x = "topleft", legend = c("ICC", "ICE", "ICS"),
      col = c(5, 2, 1), lwd = c(3, 2, 1.5), horiz = TRUE)
```

In the code snippet provided, the appearance and behavior of the plot are customized using several functions and arguments:

- **x**: This argument specifies the data to be used for the x-axis of the plot. In this case, it is `cs$Date`, indicating the “Date” column of the Michigan consumer survey data.
- **y**: This argument specifies the data to be used for the y-axis of the plot. In this case, it is `cs$icc_all`, `cs$ice_all`, and `cs$ics_all`, representing the ICC, ICE, and ICS indices from the Michigan consumer survey data.
- **type**: This argument determines the type of plot to be created. In this case, it is set to `"l"`, which stands for “line plot”. This will create a line plot of the data points.
- **col**: This argument specifies the color of the lines in the plot. In the code snippet, different colors are used for each index: 5 for ICC, 2 for ICE, and 1 for ICS.
- **lwd**: This argument controls the line width of the plot. It is set to 3 for ICC, 2 for ICE, and 1.5 for ICS, indicating different line widths for each index.
- **ylim**: This argument sets the limits of the y-axis. In this case, it is set to `c(40, 140)`, which defines the range of the y-axis from 40 to 140.
- **xlab**: This argument specifies the label for the x-axis of the plot. In the code snippet, it is set to `"Date"`.
- **ylab**: This argument specifies the label for the y-axis of the plot. In the code snippet, it is set to `"Index"`.
- **main**: This argument specifies the main title of the plot. In the code snippet, it is set to `"Key Indices of the Michigan Consumer Survey"`.
- **lines**: This function is used to add additional lines to the plot.
- **legend**: This function adds a legend to the plot. It is used to create a legend in the top-left corner (`x = "topleft"`) with labels corresponding to each index (`"ICC"`, `"ICE"`, `"ICS"`) and their respective line colors and widths.

The resulting plot, shown in Figure 4.6, displays the historical evolution of the three Michigan consumer survey indices: ICC, ICE, and ICS. These indices are considered leading indicators because they provide early signals about changes

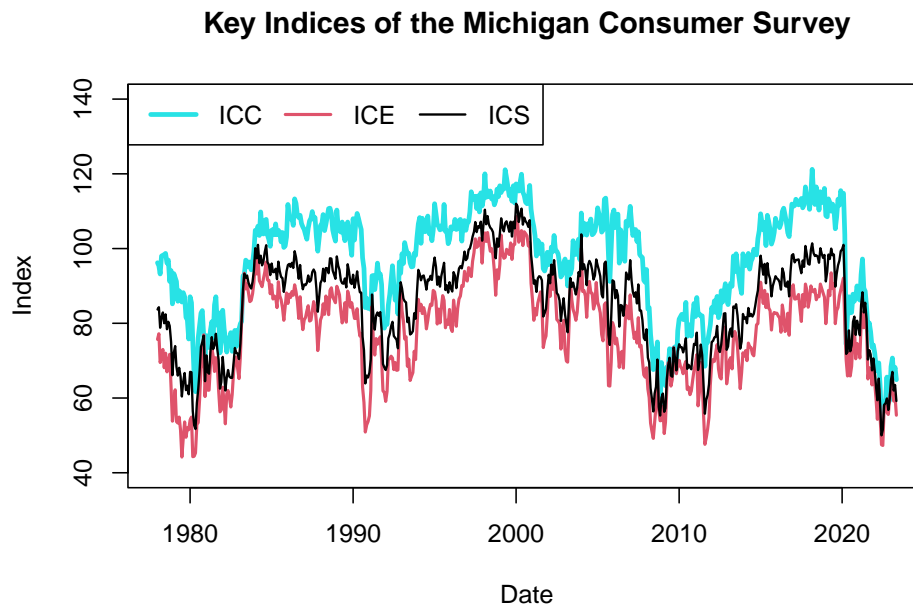


Figure 4.6: Key Indices of the Michigan Consumer Survey

in consumer sentiment and economic conditions. They often reflect consumers' expectations and attitudes before these changes are fully manifested in traditional economic indicators, such as unemployment rates or GDP growth.

Consumer sentiment plays a crucial role in shaping consumer behavior, including spending patterns, saving habits, and investment decisions. When consumer confidence is high, individuals are more likely to spend and invest, stimulating economic growth. Conversely, low consumer confidence can lead to reduced spending and investment, potentially dampening economic activity. Hence, these indices can serve as an early warning system for potential shifts in economic activity.

By incorporating the consumer survey indices alongside traditional economic indicators, policymakers and analysts can gain a more comprehensive understanding of the economic landscape. While traditional indicators like unemployment rates provide objective measures of economic conditions, the consumer survey indices offer a subjective perspective, reflecting consumers' beliefs, expectations, and intentions. This subjective insight can provide additional context and help anticipate changes in consumer behavior and overall economic activity.

Therefore, by monitoring both traditional economic indicators and the Michigan consumer survey indices, policymakers and analysts can obtain a more holistic view of the economy, enabling them to make more informed decisions and implement timely interventions to support economic stability and growth.

4.3.4 Excel File

CSV (Comma Separated Values) is a common file format used to store tabular data. As the name suggests, the values in each row of a CSV file are separated by commas. Here's an example of how data is stored in a CSV file:

- Male,8,100,3
- Female,9,20,3

In this section, we'll demonstrate how to import a CSV file using real-world Treasury yield curve rates data.

Import Yield Curve Data in CSV Format

To obtain the yield curve data, follow these steps:

1. Visit the U.S. Treasury's data center by clicking [here](#).
2. Click on "Data" in the menu bar, then select "Daily Treasury Par Yield Curve Rates."
3. On the data page, select "Download CSV" to obtain the yield curve data for the current year.
4. To access all the yield curve data since 1990, choose "All" under the "Select Time Period" option, and click "Apply." Please note that when selecting all periods, the "Download CSV" button may not be available.
5. To manually save the data as a CSV file, you can copy the data by selecting it and using the Ctrl+C (or Cmd+C) command. Open an Excel file, and use the Ctrl+V (or Cmd+V) command to paste the data into the Excel file.
6. Save the Excel file as a CSV file named 'yieldcurve.csv' in a location of your choice, ensuring that it is saved in a familiar folder for easy access.

Next, install and load the `readr` package. Run `install.packages("readr")` in the console and include the package at the top of your R script. You can then use the `read_csv()` or `read_delim()` function to import the yield curve data:

```
# Load the package
library("readr")

# Import CSV file
yc <- read_csv(file = "files/yieldcurve.csv", col_names = TRUE)
```

```
## Warning: One or more parsing issues, call `problems()` on your data frame for details.
## e.g.:
##   dat <- vroom(...)
##   problems(dat)

## Rows: 8382 Columns: 14
## -- Column specification -----
## Delimiter: ","
```

```
## chr (6): Date, 1 Mo, 2 Mo, 4 Mo, 20 Yr, 30 Yr
## dbl (8): 3 Mo, 6 Mo, 1 Yr, 2 Yr, 3 Yr, 5 Yr, 7 Yr, 10 Yr
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
# Import CSV file using the read_delim() function
yc <- read_delim(file = "files/yieldcurve.csv", col_names = TRUE, delim = ",")

## Warning: One or more parsing issues, call `problems()` on your data frame for details,
## e.g.:
##   dat <- vroom(...)
##   problems(dat)

## Rows: 8382 Columns: 14
## -- Column specification -----
## Delimiter: ","
## chr (6): Date, 1 Mo, 2 Mo, 4 Mo, 20 Yr, 30 Yr
## dbl (8): 3 Mo, 6 Mo, 1 Yr, 2 Yr, 3 Yr, 5 Yr, 7 Yr, 10 Yr
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

In the code snippets above, the `read_csv()` and `read_delim()` functions from the `readr` package are used to import a CSV file named “yieldcurve.csv”. The `col_names = TRUE` argument indicates that the first row of the CSV file contains column names. The `delim = ","` argument specifies that the columns are separated by commas, which is the standard delimiter for CSV (Comma Separated Values) files. Either one of the two functions can be used to read the CSV file and store the data in the variable `yc` for further analysis.

To import Excel files, we use the `readxl` package. To import an Excel file, execute `install.packages("readxl")` in the console, and then load the package at the start of your R script.

```
# Load the package
library("readxl")

# Print names of all worksheets
# Replace 'path' with your Excel file path
# excel_sheets(path="car.xlsx")

# Import the sheet you want
# Replace 'path' and 'sheet' with your Excel file path and sheet name
# car <- read_excel(path="car.xlsx", sheet="Statistics_A", col_names=TRUE)

# Import all sheets
# Replace 'path' with your Excel file path
```

```
# car_list <- lapply(excel_sheets(path="car.xlsx"),read_excel, path="car.xlsx",col_name
```

4.3.5 Saving Data

Once the data has been imported and possibly manipulated, it can be saved in various formats.

```
# Save data in different formats
```

```
save(yc, file="yieldcurve.RData")
```

```
write_csv(x=yc, path="yieldcurve.csv", na="NA")
```

```
## Warning: The `path` argument of `write_csv()` is deprecated as of readr 1.4.0.
```

```
## i Please use the `file` argument instead.
```

```
## This warning is displayed once every 8 hours.
```

```
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
```

```
## generated.
```

```
# Other formats can be used by replacing 'yieldcurve' with your data and 'path' with y
```

For more information on importing data in R, consider taking DataCamp's Introduction to Importing Data in R course.

Chapter 5

DataCamp

To learn more about R Markdown, consider reading R Markdown: The Definitive Guide.

- Create an account on www.datacamp.com with the free student license you receive by email
- Take Introduction to R for Finance on DataCamp
- Then take additional courses and explore some of the exciting things you can do with R (and Python)
- For Economics and Business students I recommend
 - The entire skill track Finance Fundamentals in R
 - The entire skill track Applied Finance in R
 - The entire skill track Importing & Cleaning Data with R
 - The entire skill track Data Visualization with R
 - The entire skill track Interactive Data Visualization in R
 - The entire career track Quantitative Analyst with R
 - Reporting with R Markdown to learn how to write reports
 - Spreadsheet Fundamentals to learn Excel

Chapter 6

Assignment

1. Work through all chapters of Module 1: Introduction to R.
2. DataCamp courses:
 - A
 - B
3. Create