Data Analysis for Economic Research

Julian F. Ludwig (Texas Tech University)

July 18, 2023

Contents

Preface **4**
Introduction . . . 4
Author . . . 5

Module I **6**

Introduction to R for Economic Research **6**

1 Software Overview **7**

2 Software Installation **8**
2.1 Install R . . . 8
2.2 Install RStudio . . . 9
2.3 Install R Markdown . . . 9
2.4 Install LaTeX . . . 9
2.5 Install R Packages . . . 9
2.6 18-Step Test . . . 10

3 RStudio Interface **15**
3.1 RStudio Sections . . . 15
3.2 R Scripts . . . 16

4 R Data Types and Structures **16**
4.1 Scalar . . . 17
4.2 Vector . . . 19
4.3 Matrix (`matrix`) . . . 21
4.4 List (`list`) . . . 22
4.5 Data Frame (`data.frame`) . . . 23
4.6 Tibble (`tbl_df`) . . . 25
4.7 Data Table (`data.table`) . . . 27
4.8 Extensible Time Series (`xts`) . . . 29

5 Importing Data in R **30**
5.1 Working Directory . . . 30
5.2 Yield Curve . . . 31
5.3 Michigan Survey . . . 37
5.4 Tealbook . . . 41

6 Downloading Data in R **49**
6.1 Web API . . . 49

6.2 Using `getSymbols()` . . . 49
6.3 Using `Quandl()` . . . 52
6.4 Saving Data . . . 54
6.5 Summary and Resources . . . 56

7 Writing Reports with R Markdown **56**
7.1 Creating an R Markdown Document . . . 56
7.2 YAML Header . . . 58
7.3 Markdown Syntax . . . 58
7.4 R Chunks . . . 58
7.5 Embedding R Variables into Text . . . 60
7.6 LaTeX Syntax for Math . . . 60
7.7 Printing Tables . . . 61
7.8 Summary and Resources . . . 62

8 Learning R with DataCamp **62**
8.1 Preparing for the Course . . . 62
8.2 XP System and Course Completion . . . 63
8.3 Developing Your Skills . . . 63

9 Data Report on Yield Curve **64**
9.1 Preparations . . . 64
9.2 Data Guidelines . . . 64
9.3 R Markdown Guidelines . . . 64

Module II 65

Traditional Economic Indicators **65**

10 Economic Indicators **66**

11 Data Categorization **67**
11.1 Qualitative vs. Quantitative . . . 68
11.2 Discrete vs. Continuous . . . 69
11.3 Levels of Measurement . . . 69
11.4 Index vs. Absolute Data . . . 72
11.5 Stock vs. Flow . . . 72
11.6 Data Dimensions . . . 73
11.7 Conclusion . . . 79

12 Data Transformation **80**
12.1 Example Data . . . 80
12.2 Growth . . . 83
12.3 Differentiation . . . 87
12.4 Natural Logarithm . . . 89
12.5 Ratio . . . 93
12.6 Gap . . . 99
12.7 Filtering . . . 99

13 Data Aggregation **101**

14 Data Source **101**
14.1 Bureau of Economic Analysis (BEA) . . . 102
14.2 U.S. Census Bureau . . . 102

14.3 Bureau of Labor Statistics (BLS) . . . 102
14.4 Federal Reserve Board of Governors . . . 102
14.5 U.S. Department of the Treasury . . . 102
14.6 Financial Market Exchanges . . . 103
14.7 Federal Reserve Economic Data (FRED) . . . 103
14.8 U.S. Department of Agriculture Economic Research Service (ERS) . . . 103
14.9 Bloomberg . . . 103
14.10Yahoo Finance . . . 103

15 Temporal Patterns **103**
15.1 Trends . . . 104
15.2 Business Cycles . . . 104
15.3 Seasonal Cycles . . . 104

16 Regional Patterns **104**
16.1 County-Level Patterns . . . 104
16.2 State-Level Economic Patterns . . . 104
16.3 Country-Level Economic Patterns . . . 105

17 Causal Relationships **105**

18 Data Report on Traditional Economic Indicators **105**
18.1 Preparations . . . 105
18.2 Data Guidelines . . . 106
18.3 R Markdown Guidelines . . . 106

Module III 106

Survey- and Text-Based Economic Indicators **106**

19 Why Subjective Information Matters **107**

20 Michigan Consumer Survey **107**

21 Real-Time and Forecaster Data **108**

22 Text-Based Economic Indicators **108**

23 Data Report on Survey- and Text-Based Economic Indicators **108**
23.1 Preparations . . . 108
23.2 Data Guidelines . . . 108
23.3 R Markdown Guidelines . . . 108

Preface

Introduction

This book is your guide to preparing for a career in Economics and Finance. These fields need you to be great at understanding and explaining data. This book will help you improve your data science and analytics skills, and use them to answer questions about business cycles, economic growth, and financial markets.

A significant part of this book is about teaching you how to program in R, a popular language used for data analysis and graphics. You'll also learn how to use complementary software like RStudio, R Markdown, and LaTeX to craft professional reports that showcase your data analyses.

But it's not all about coding. We'll also explore economic indicators to help you understand business cycles and regional differences in economic performance. You'll learn about traditional measures like GDP growth,

inflation, money supply, and interest rates. We also touch on newer, survey- and text-based indicators, such as consumer confidence indices and news-based indicators of economic uncertainty, which come from language analysis of newspapers, social media, or Google searches.

The book is divided into three modules:

- *Module I: Introduction to R for Economic Research*
- *Module II: Traditional Economic Indicators*
- *Module III: Survey- and Text-Based Economic Indicators*

Each module builds on the previous one, providing you with a coherent learning journey.

By the time you finish this book, you'll be ready to analyze a wide range of economic and financial data. Your learning progress will be assessed based on the completion of DataCamp courses and the production of three data reports, one for each module.

Author

Julian F. Ludwig
Assistant Professor
Department of Economics
Texas Tech University
253 Holden Hall
Lubbock, TX 79409

Website: www.julianfludwig.com
Email: julian.ludwig@ttu.edu

Education

- Ph.D., Economics, University of Texas at Austin, 2019
- M.S., Economics, University of Texas at Austin, 2016
- M.S., International and Monetary Economics, University of Bern in Switzerland, 2014
- B.S., Economics, University of Bern in Switzerland, 2012

Research Interests

- Macroeconomics
- Time Series Econometrics

Courses Offered

- ECO 5316 Time Series Econometrics
- ECO 5311 Macroeconomic Theory and Policy
- ECO 4306 Economic and Business Forecasting
- ECO 4300 Economic Research: Data-Driven Analysis

- ECO 3323 Principles of Money, Banking and Credit

Module I

Introduction to R for Economic Research

The first module provides an overview of freely accessible software widely used for data analysis in economic research. The central focus is R, a programming language specifically developed for statistical computing and graphics. Complementary software - RStudio, R Markdown, and LaTeX - are introduced as supportive tools for dynamic document creation based on R. This module will guide you through the installation process and will familiarize you with the utilization of these tools by exploring key syntax and its application to financial and economic data. Furthermore, the module discusses valuable resources such as DataCamp, an online learning platform, to enhance your expertise.

Module Overview This module consists of nine chapters:

- Chapter 1: "Software Overview" provides an overview of software commonly used for data analysis in economic research.
- Chapter 2: "Software Installation" gives instructions for the installation of these software tools.
- Chapter 3: "RStudio Interface" guides you through the interface of RStudio, a widely used IDE for R.
- Chapter 4: "R Data Types and Structures" discusses the fundamental data types and structures in R.
- Chapter 5: "Importing Data in R" explains how to import datasets into R for analysis.
- Chapter 6: "Downloading Data in R" instructs on how to download data directly within the R environment.
- Chapter 7: "Writing Reports with R Markdown" covers the use of R Markdown for creating dynamic, reproducible data reports.
- Chapter 8: "Learning R with DataCamp" introduces DataCamp as an essential online resource for learning R.
- Chapter 9: "Data Report on Yield Curve" provides an assessment to test the skills and knowledge gained throughout the module, centered around a data report on the yield curve.

In addition, the following DataCamp courses are integral to supplementing the content in Module I:

- Introduction to R for Finance
- Intermediate R for Finance

Refer to Chapter 8 for guidelines on how to optimize your learning experience from these DataCamp courses.

Learning Objectives By the end of this module, you should be able to:

1. Identify the role and importance of software tools such as R, RStudio, R Markdown, and LaTeX in conducting economic research.
2. Install and set up necessary software including R, RStudio, R Markdown, and LaTeX for economic data analysis.
3. Navigate the RStudio interface, understanding its various panels and functionalities.
4. Understand and apply basic R data types and structures for data storage and manipulation.
5. Import external datasets into R for analysis, and download data directly within the R environment.
6. Create professional data reports using R Markdown, incorporating R code, text, plots, and tables seamlessly.
7. Utilize online learning resources like DataCamp to enhance R skills and knowledge further.
8. Apply the knowledge and skills acquired to write a detailed data report on yield curves, demonstrating proficiency in data import, manipulation, and presentation, as well as interpretation of economic data.

Learning Activities & Assessments for Module I Throughout this module, you'll participate in the following activities:

1. **Reading Material**: Read the nine chapters in Module I. Each chapter covers different topics, including the importance of specific software tools, the installation process for these tools, using RStudio, understanding R's data types and structures, importing and downloading data in R, creating reports with R Markdown, and learning R via online resources like DataCamp.
2. **Applying Reading Material**: Implement the knowledge you've gained from the chapters by replicating the steps outlined. This includes software installation, working with different data types, importing and downloading data, and visualizing results. To ensure understanding, I recommend copying any provided code into your RStudio environment and attempting to reproduce the same output.
3. **Online Learning with DataCamp**: Complete the DataCamp courses Introduction to R for Finance and Intermediate R for Finance. These courses will bolster the concepts discussed in this module and introduce you to new data analysis techniques.

Your learning will be assessed based on the following:

1. **DataCamp Course Completion**: Completion of the designated DataCamp courses: Introduction to R for Finance and Intermediate R for Finance is a necessity and part of your overall assessment.
2. **Data Report on Yield Curve**: Create a comprehensive data report on yield curves utilizing R Markdown. This report will act as a practical demonstration of your understanding of R, RStudio, and R Markdown. Your data report should showcase your proficiency in importing and manipulating data, visualizing the results, and delivering a sensible interpretation of yield curves. The quality of your report will form a significant part of your module assessment. For more details about this assignment, refer to Chapter 9.

Remember, mastery of R programming and data analysis is a step-by-step process. Make sure to thoroughly understand each data handling and manipulation technique before moving on to the next. If any challenges arise during your learning journey, do not hesitate to seek help. Good luck with your exploration of R!

1 Software Overview

The following software and programming languages are commonly used for conducting economic analyses:

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 for conducting economic research - 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.

All the software and programming languages mentioned above are **open-source**, meaning they are freely available and actively developed by a community of contributors. By mastering these tools, you will have the necessary skills to perform data analysis, create reproducible reports, and effectively communicate your findings in the field of economics.

2 Software Installation

For data analysis, RStudio will serve as your environment for writing code and text. However, as an Integrated Development Environment (IDE), RStudio is not a standalone program; it depends on R, R Markdown, and a LaTeX processor installed on your system. 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.

2.1 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.

2.2 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.

2.3 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.

2.4 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.

2.5 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")
getSymbols(Symbols="^GSPC")
```

```
## [1] "GSPC"
```

```
head(GSPC)
```

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

2.6 18-Step Test

To ensure that `R`, `RStudio`, `R Markdown`, and `LaTeX` are installed properly, you can follow the 18-step test provided below. 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 Chapter 2.4 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 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")
```

Figure 1: R Plot

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:

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 2023-06-01 3.75.

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).

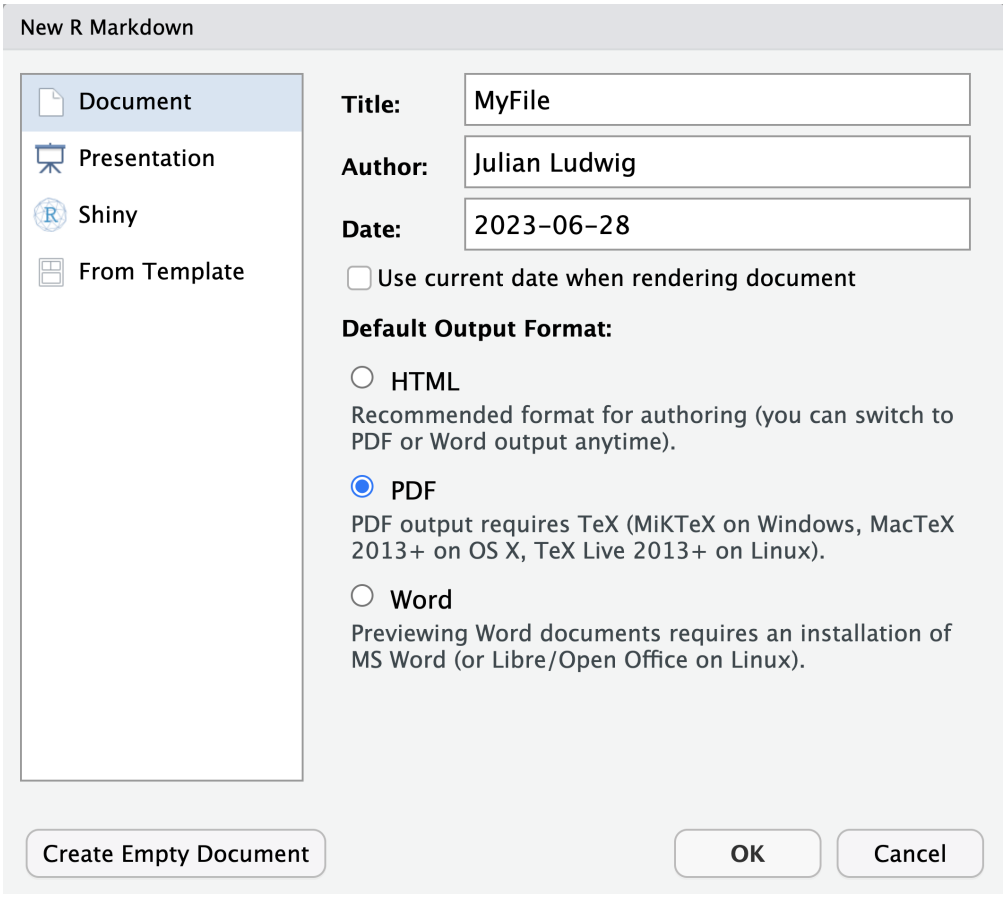


Figure 2: New R Markdown

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: PDF File Produced with R Markdown

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`:  you have options to 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.

3 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.

3.1 RStudio Sections

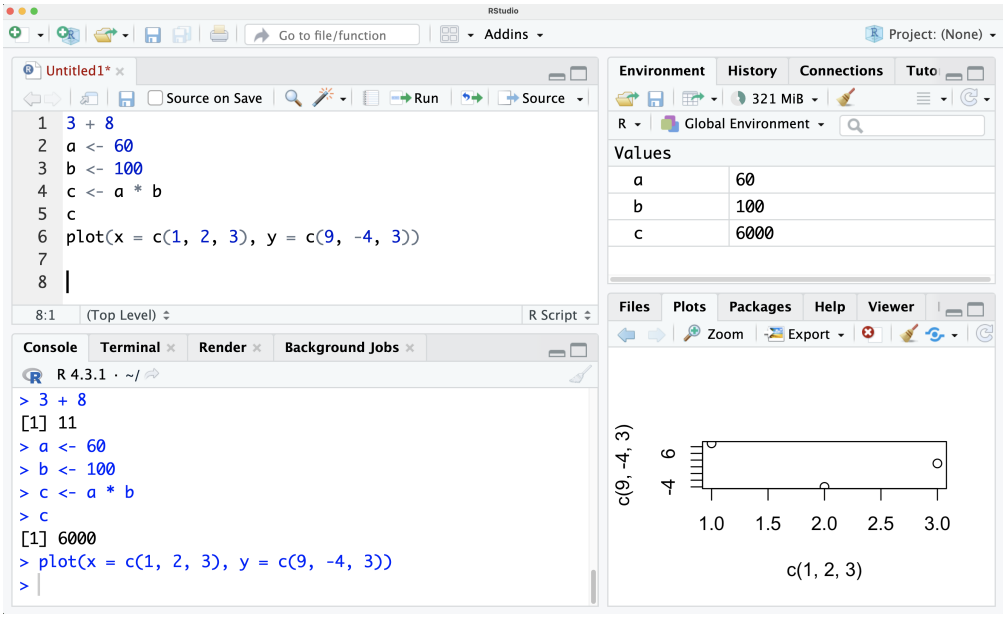
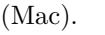


Figure 4: RStudio Interface

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  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  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, 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.

3.2 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.

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 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.

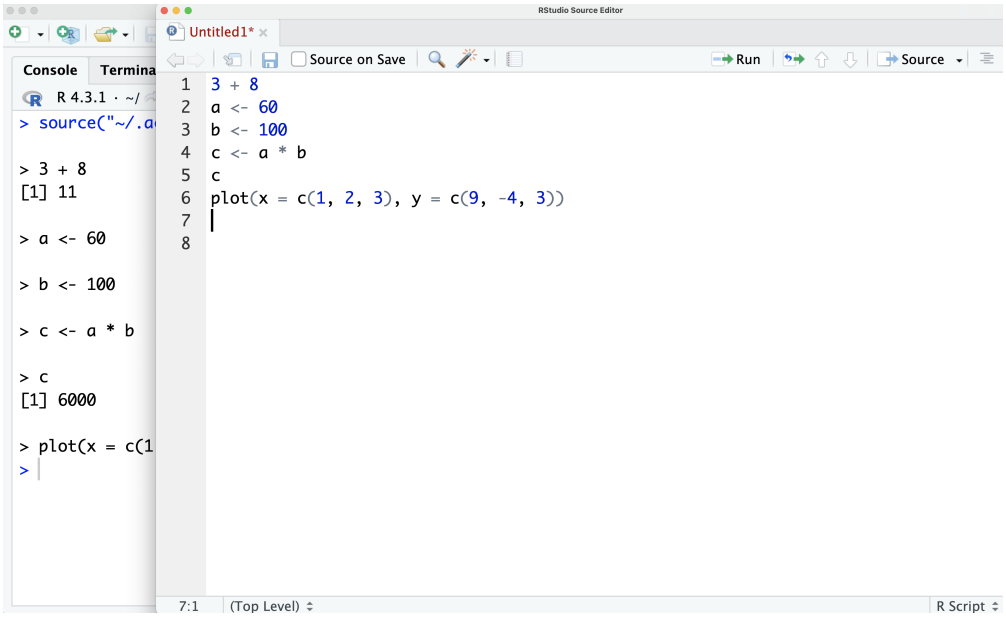


Figure 5: RStudio Interface with Separate R Script Window

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 chapter, 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** (`tbl_df`): 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.

4.1 Scalar

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

```r
# 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".
```

Numbers, Characters, and Logical Values

```
## [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:

```r
# 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 When conducting economic research, it is common to deal with data types specifically designed for storing date and time information:

```r
# 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"
```

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

```r
# 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"
```

```r
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:

Here are some example operations for `Date` objects:

```r
# 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
```

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

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

4.2 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:

Table 1: Syntax for Date Format

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	

```r
# 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:

```r
# 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 `sec()` function creates a sequence of numbers or dates:

```r
# 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"
## [5] "2004-09-01" "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
```

4.3 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"
```

```r
typeof(numeric_matrix)  # Returns "numeric".
```

```
## [1] "integer"
```

```r
# 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"
```

```r
class(character_matrix)  # Returns "matrix".
```

```
## [1] "matrix" "array"
```

```r
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.

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

```
## [1] "d"
```

```r
# Print the second row:
character_matrix[2, ]
```

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

```r
# 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
```

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

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

4.4 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.

```r
# 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
```

4.5 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")
```

```r
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
```

```r
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:

```r
# 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"
```

```r
# 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"
```

```r
# 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
```

```r
# 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.

4.6 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    66 M
## 2 b         4    32 F
## 3 c         3    34 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    66 M     0.0303
## 2 b         4    32 F     0.125
## 3 c         3    34 F     0.0882
```

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:

```r
library("tidyverse")
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
```

```r
# 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.

4.7 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:

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

# 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  36   M
## 2:    b  2  66   F
## 3:    c  3  60   F
```

```r
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:

```r
# 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  36   M
```

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

```
##    name id age sex
## 1:    a  4  36   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  36
## 2:   F  66
## 3:   F  60
```

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  36   M  0.11111111          144          40
## 2:    b  2  66   F  0.03030303          132          68
## 3:    c  3  60   F  0.05000000          180          63
```

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 36
## 2:   F 63
```

```
# 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            36
## 2:      F            63
```

```
# 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
## 1:    a  4  36   M  0.11111111          144          40
```

```
## 2:    b  2  66   F  0.03030303          132          68
## 3:    c  3  60   F  0.05000000          180          63
##    age_by_sex
## 1:         36
## 2:         63
## 3:         63
```

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

4.8 Extensible Time Series (`xts`)

`xts` (**e**x**t**ensible **t**ime **s**eries) 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:

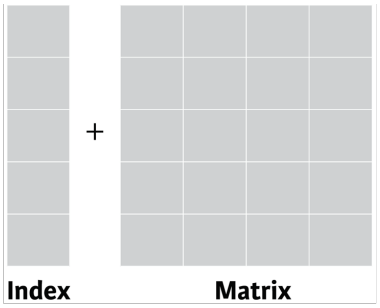


Figure 6: Data with Index. *Source*: [DataCamp](https://learn.datacamp.com/courses/manipulating-time-series-data-with-xts-and-zoo-in-r).

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 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.

5 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 chapter 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.

5.1 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/daer"
```

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 (\).

5.2 Yield Curve

This chapter demonstrates how to import Treasury yield curve rates data from a CSV file. Treasury yield curve data represents interest rates on U.S. government loans for different maturities. By comparing rates at various maturities, such as 1-month, 1-year, 5-year, 10-year, and 30-year, we gain insights into market expectations. An upward sloping yield curve, where interest rates for longer-term loans (30-year) are significantly higher than those for short-term loans (1-month), often indicates economic growth. Conversely, a downward sloping or flat curve may suggest the possibility of a recession. This data is crucial in making informed decisions regarding borrowing, lending, and understanding the broader economic landscape.

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. If the "Download CSV" button is not available, click on the link called Download interest rates data archive. Then, select yield-curve-rates-1990-2021.csv to download the daily yield curve data from 1990-2021.
6. To add the more recent yield curve data since 2022, go back to the previous page, choose the desired years (e.g., "2022", "2023") under the "Select Time Period," and click "Apply."

7. Manually copy the additional rows of yield curve data and paste them into the yield-curve-rates-1990-2021.csv file.
8. Save the file as "yieldcurve.csv" in a location of your choice, ensuring that it is saved in a familiar folder for easy access.

Following these steps will allow you to obtain the yield curve data, including both historical and recent data, in a single CSV file named "yieldcurve.csv."

5.2.1 Import CSV 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

To import the 'yieldcurve.csv' CSV file in R, 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)

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

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`
##    <chr>    <chr>  <chr>   <dbl> <chr>   <dbl>  <dbl>  <dbl>  <dbl>
##  1 01/02/~ N/A    N/A      7.83 N/A      7.89   7.81   7.87   7.9
##  2 01/03/~ N/A    N/A      7.89 N/A      7.94   7.85   7.94   7.96
##  3 01/04/~ N/A    N/A      7.84 N/A      7.9    7.82   7.92   7.93
##  4 01/05/~ N/A    N/A      7.79 N/A      7.85   7.79   7.9    7.94
##  5 01/08/~ N/A    N/A      7.79 N/A      7.88   7.81   7.9    7.95
##  6 01/09/~ N/A    N/A      7.8  N/A      7.82   7.78   7.91   7.94
##  7 01/10/~ N/A    N/A      7.75 N/A      7.78   7.77   7.91   7.95
##  8 01/11/~ N/A    N/A      7.8  N/A      7.8    7.77   7.91   7.95
##  9 01/12/~ N/A    N/A      7.74 N/A      7.81   7.76   7.93   7.98
## 10 01/16/~ N/A    N/A      7.89 N/A      7.99   7.92   8.1    8.13
## # i 8,372 more rows
## # i 5 more variables: `5 Yr` <dbl>, `7 Yr` <dbl>, `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 Chapter 4.6. 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
## "character" "character" "character"   "numeric" "character"
##        6 Mo        1 Yr        2 Yr        3 Yr        5 Yr
##   "numeric"   "numeric"   "numeric"   "numeric"   "numeric"
##        7 Yr       10 Yr       20 Yr       30 Yr
##   "numeric"   "numeric" "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] "01/02/90" "01/03/90" "01/04/90" "01/05/90" "01/08/90"
## [6] "01/09/90"
```

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

# Sort data according to date
yc <- yc[order(yc$Date), ]
```

```r
# Print first 5 observations of date column
head(yc$Date)
```

```
## [1] "1990-01-02" "1990-01-03" "1990-01-04" "1990-01-05"
## [5] "1990-01-08" "1990-01-09"
```

```r
# Print last 5 observations of date column
tail(yc$Date)
```

```
## [1] "2023-06-23" "2023-06-26" "2023-06-27" "2023-06-28"
## [5] "2023-06-29" "2023-06-30"
```

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

```r
yc
```

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

5.2.2 Plotting Historical 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:

```r
# 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".

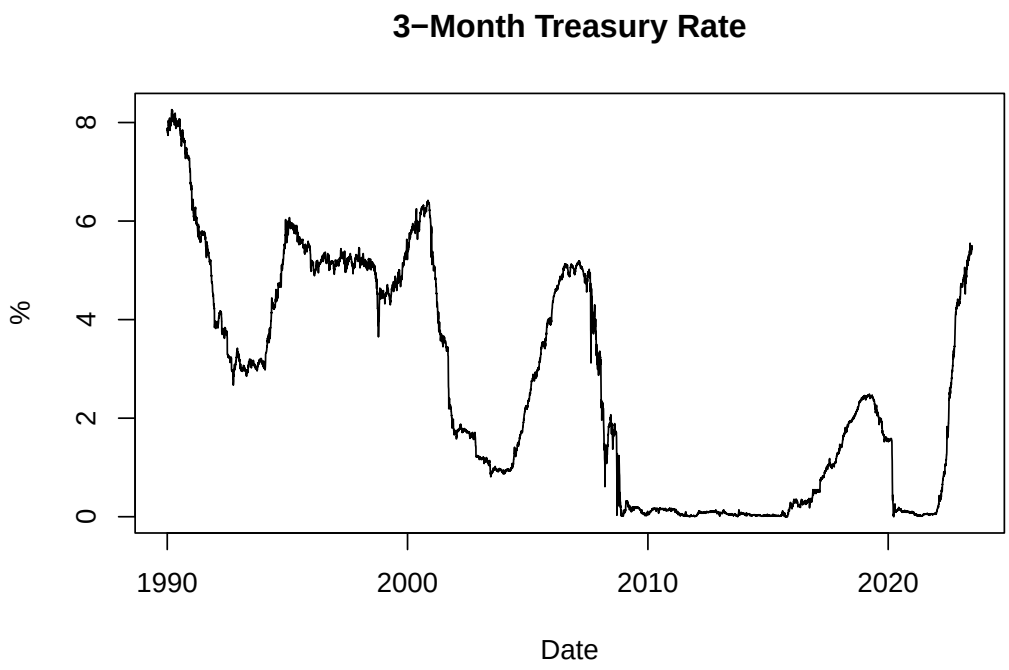


Figure 7: 3-Month Treasury Rate

The resulting plot, shown in Figure 7, 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.

5.2.3 Plotting 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(tail(yc[, -1], 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
## [13] 3.85
```

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

```
yc_most_recent_labels
```

```
##  [1] "1 Mo"  "2 Mo"  "3 Mo"  "4 Mo"  "6 Mo"  "1 Yr"  "2 Yr"
##  [8] "3 Yr"  "5 Yr"  "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(tail(yc$Date, 1), 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(tail(yc$Date, 1), 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.

The resulting plot, shown in Figure 8, 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

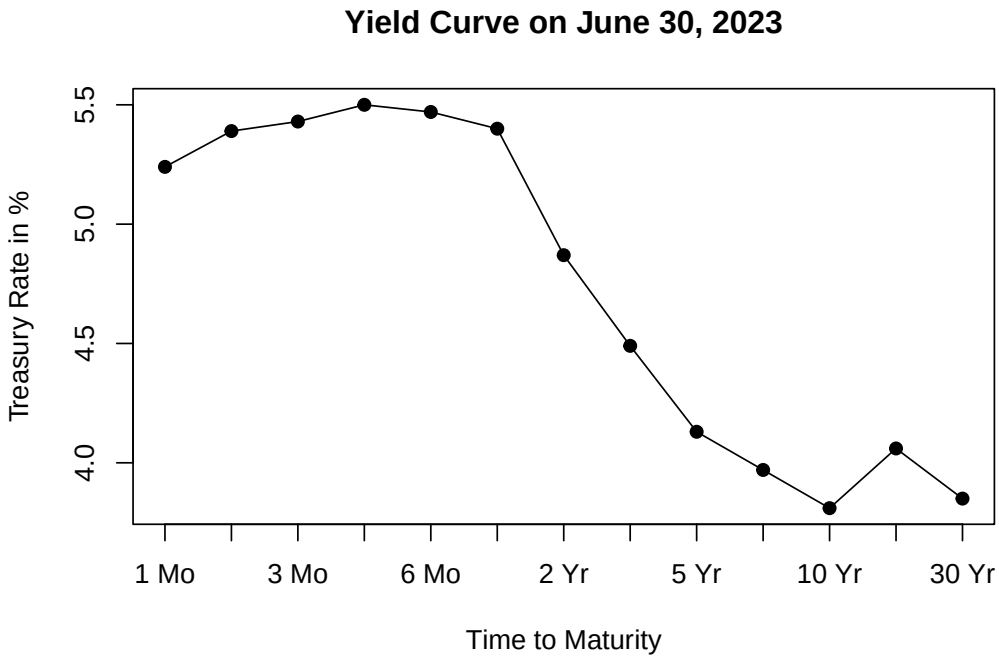


Figure 8: Yield Curve on June 30, 2023

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.

5.3 Michigan Survey

In this chapter, we'll demonstrate how to import a TSV file using real-world consumer survey data collected by the University of Michigan. This data is gathered through surveys that ask people about their opinions and feelings regarding the economy. It helps to understand how consumers perceive the current economic conditions and their expectations for the future. The data provides valuable insights into consumer behavior and helps economists, policymakers, and businesses make informed decisions.

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-Deliminated (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.

5.3.1 Import TSV File

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

To import the consumer survey TSV file, 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)

# 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")
```

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
##    <dbl> <dbl>   <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
##  1     1  1978    83.7        NA        NA        NA      93.7
##  2     2  1978    84.3        NA        NA        NA      99.7
##  3     3  1978    78.8        NA        NA        NA      91.7
##  4     4  1978    81.6        NA        NA        NA      91.8
##  5     5  1978    82.9        NA        NA        NA      95.1
##  6     6  1978    80          NA        NA        NA      91.7
##  7     7  1978    82.4        NA        NA        NA      92.2
##  8     8  1978    78.4        NA        NA        NA      87.8
##  9     9  1978    80.4        NA        NA        NA      86.6
## 10    10  1978    79.3        NA        NA        NA      90.6
## # i 535 more rows
## # i 353 more variables: ics_a3554 <dbl>, 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>, ...
```

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"
## [5] "1978-05-01" "1978-06-01"
```

5.3.2 Plotting 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 9, 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 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.

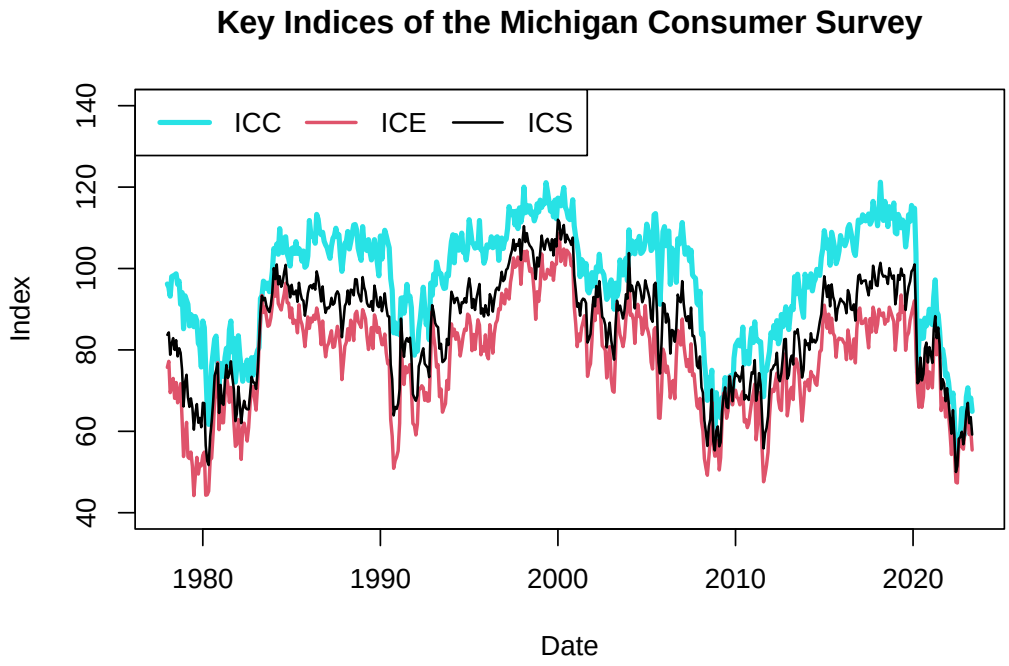


Figure 9: Key Indices of the Michigan Consumer Survey

5.4 Tealbook

In this chapter, we'll demonstrate how to import an Excel workbook (`xls` or `xlsx`) by importing projections data from Tealbooks (formerly known as Greenbooks). Tealbooks are an important resource as they consist of economic forecasts created by the research staff at the Federal Reserve Board of Governors. These forecasts are prepared before each meeting of the Federal Open Market Committee (FOMC), which is a key decision-making body responsible for determining U.S. monetary policy. The Tealbooks' projections provide valuable insights into the expected performance of the economy, including indicators such as inflation, GDP growth, unemployment rates, and interest rates. The data from Tealbooks are made available to the public with a five-year lag, allowing researchers and analysts to examine past economic forecasts and compare them to actual outcomes. By importing and analyzing this data, we can gain a deeper understanding of the economic outlook as perceived by the Federal Reserve's research staff and the FOMC prior to their policy meetings.

To obtain the Tealbook forecasts, perform the following steps:

1. Visit the website of the Federal Reserve Bank of Philadelphia by clicking here.
2. Navigate to SURVEYS & DATA, select Real-Time Data Research, choose Tealbook (formerly Greenbook) Data Sets, and finally select Philadelphia Fed's Tealbook (formerly Greenbook) Data Set.
3. On the data page, download Philadelphia Fed's Tealbook/Greenbook Data Set: Row Format.
4. Save the Excel workbook as 'GBweb_Row_Format.xlsx' in a familiar location for easy retrieval.

The Excel workbook contains multiple sheets, each representing different variables. The sheet `gRGDP`, for example, holds real GDP growth forecasts. Columns `gRGDPF1` to `gRGDPF9` represent one- to nine-quarter-ahead forecasts, `gRGDPF0` represents the nowcast, and `gRGDPB1` to `gRGDPB4` represent one- to four-quarter-behind backcasts.

5.4.1 Import Excel Workbook

An Excel workbook (`xlsx`) is a file that contains one or more spreadsheets (worksheets), which are the separate "pages" or "tabs" within the file. An Excel worksheet, or simply a sheet, is a single spreadsheet within an Excel workbook. It consists of a grid of cells formed by intersecting rows (labeled by numbers) and

columns (labeled by letters), which can hold data such as text, numbers, and formulas.

To import the Tealbook, which is an Excel workbook, install and load the `readxl` package by running `install.packages("readxl")` in the console and include the package at the beginning of your R script. You can then use the `excel_sheets()` function to print the names of all Excel worksheets in the workbook, and the `read_excel()` function to import a particular worksheet:

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

# Get names of all Excel worksheets
sheet_names <- excel_sheets(path = "files/GBweb_Row_Format.xlsx")
sheet_names
```

```
##  [1] "Documentation" "gRGDP"         "gPGDP"
##  [4] "UNEMP"         "gPCPI"         "gPCPIX"
##  [7] "gPPCE"         "gPPCEX"        "gRPCE"
## [10] "gRBF"          "gRRES"         "gRGOVF"
## [13] "gRGOVSL"       "gNGDP"         "HSTART"
## [16] "gIP"
```

```
# Import one of the Excel worksheets: "gRGDP"
gRGDP <- read_excel(path = "files/GBweb_Row_Format.xlsx", sheet = "gRGDP")
gRGDP
```

```
## # A tibble: 466 x 16
##     DATE gRGDPB4 gRGDPB3 gRGDPB2 gRGDPB1 gRGDPF0 gRGDPF1 gRGDPF2
##    <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
##  1 1967.     5.9     1.9     4       4.5     0.5     1        NA
##  2 1967.     1.9     4       4.5     0       1.6     4.1      NA
##  3 1967.     1.9     4       4.5    -0.3     1.4     4.7      NA
##  4 1967.     1.9     4       4.5    -0.3     2.3     4.5      NA
##  5 1967.     3.4     3.8    -0.2     2.4     4.3    NA        NA
##  6 1967.     3.4     3.8    -0.2     2.4     4.2     4.5      NA
##  7 1967.     3.4     3.8    -0.2     2.4     4.8     6.4      NA
##  8 1967.     3.4     3.8    -0.2     2.4     4.9     6.4      NA
##  9 1967.     3.8    -0.2     2.4     4.2     6.1    NA        NA
## 10 1967.     3.8    -0.2     2.4     4.2     5.1    NA        NA
## # i 456 more rows
## # i 8 more variables: gRGDPF3 <dbl>, gRGDPF4 <dbl>,
## #   gRGDPF5 <dbl>, gRGDPF6 <dbl>, gRGDPF7 <dbl>, gRGDPF8 <dbl>,
## #   gRGDPF9 <dbl>, GBdate <dbl>
```

To import all sheets, use the following code:

```
# Import all worksheets of Excel workbook
TB <- lapply(sheet_names, function(x)
    read_excel(path = "files/GBweb_Row_Format.xlsx", sheet = x))
names(TB) <- sheet_names

# Summarize the list containing all Excel worksheets
summary(TB)
```

```
##               Length Class  Mode
## Documentation  2     tbl_df list
## gRGDP         16     tbl_df list
## gPGDP         16     tbl_df list
```

```
## UNEMP          16     tbl_df list
## gPCPI          16     tbl_df list
## gPCPIX         16     tbl_df list
## gPPCE          16     tbl_df list
## gPPCEX         16     tbl_df list
## gRPCE          16     tbl_df list
## gRBF           16     tbl_df list
## gRRES          16     tbl_df list
## gRGOVF         16     tbl_df list
## gRGOVSL        16     tbl_df list
## gNGDP          16     tbl_df list
## HSTART         16     tbl_df list
## gIP            16     tbl_df list
```

In the provided code snippet, the `lapply()` function is used to apply the `read_excel()` function to each element in the `sheet_names` list, returning a list that is the same length as the input. Consequently, the `TB` object becomes a list in which each element is a tibble corresponding to an individual Excel worksheet. For instance, to access the worksheet that contains unemployment forecasts, you would use the following code:

```
# Print one of the worksheets (unemployment forecasts):
TB[["UNEMP"]]
```

```
## # A tibble: 466 x 16
##     DATE UNEMPB4 UNEMPB3 UNEMPB2 UNEMPB1 UNEMPF0 UNEMPF1 UNEMPF2
##    <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
##  1 1967.     3.8     3.8     3.8     3.7     3.8     4.1      NA
##  2 1967.     3.8     3.8     3.7     3.7     4       4.1      NA
##  3 1967.     3.8     3.8     3.7     3.7     3.9     4        NA
##  4 1967.     3.8     3.8     3.7     3.7     3.9     3.9      NA
##  5 1967.     3.8     3.7     3.7     3.8     3.9    NA        NA
##  6 1967.     3.8     3.7     3.7     3.8     3.9     3.8      NA
##  7 1967.     3.8     3.7     3.7     3.8     3.8     3.6      NA
##  8 1967.     3.8     3.7     3.7     3.8     3.8     3.6      NA
##  9 1967.     3.7     3.7     3.8     3.9     3.8    NA        NA
## 10 1967.     3.7     3.7     3.8     3.9     4      NA        NA
## # i 456 more rows
## # i 8 more variables: UNEMPF3 <dbl>, UNEMPF4 <dbl>,
## #   UNEMPF5 <dbl>, UNEMPF6 <dbl>, UNEMPF7 <dbl>, UNEMPF8 <dbl>,
## #   UNEMPF9 <dbl>, GBdate <chr>
```

5.4.2 Nowcast and Backcast

Let's explore the forecaster data, interpreting the information provided in the Tealbook Documentation file. This file is accessible on the same page as the data, albeit further down. The data set extends beyond simple forecasts, encompassing nowcasts and backcasts as well.

The imported Excel workbook `GBweb_Row_Format.xlsx` includes multiple sheets, each one representing distinct variables. Take, for instance, the sheet `gPGDP`, which contains forecasts for inflation (GDP deflator). Columns `gPGDPF1` through `gPGDPF9` represent forecasts for one to nine quarters ahead respectively, while `gPGDPF0` denotes the nowcast. The columns `gPGDPB1` to `gPGDPB4`, on the other hand, symbolize backcasts for one to four quarters in the past.

```
# Print column names of inflation back-, now-, and forecasts
colnames(TB[["gPGDP"]])
```

```
##  [1] "DATE"    "gPGDPB4" "gPGDPB3" "gPGDPB2" "gPGDPB1" "gPGDPF0"
##  [7] "gPGDPF1" "gPGDPF2" "gPGDPF3" "gPGDPF4" "gPGDPF5" "gPGDPF6"
```

```
## [13] "gPGDPF7" "gPGDPF8" "gPGDPF9" "GBdate"
```

A **nowcast** (`gPGDPF0`) effectively serves as a present-time prediction. Given the delay in the release of economic data, nowcasting equips economists with the tools to make informed estimates about what current data will ultimately reveal once it is officially published.

Conversely, a **backcast** (`gPGDPB1` through `gPGDPB4`) is a forecast formulated for a time period that has already transpired. This might seem counterintuitive, but it's necessary because the initial estimates of economic data are often revised significantly as more information becomes available.

It is noteworthy that financial data typically doesn't experience such revisions, since interest rates or asset prices are directly observed on exchanges, eliminating the need for revisions. In contrast, inflation is not directly observable and requires deduction from a wide array of information sources. These sources can encompass export data, industrial production statistics, retail sales data, employment figures, consumer spending metrics, and capital investment data, among others.

In summary, while forecasting seeks to predict future economic scenarios, nowcasting aspires to deliver an accurate snapshot of the current economy, and backcasting strives to enhance our understanding of historical economic conditions.

5.4.3 Information Date and Realization Date

The imported Excel workbook `GBweb_Row_Format.xlsx` is structured in a **row format** as opposed to a column format. As clarified in the Tealbook Documentation file, this arrangement means that (1) each row corresponds to a specific Tealbook publication date, and (2) the columns represent the forecasts generated for that particular Tealbook. The forecast horizons include, at most, the four quarters preceding the nowcast quarter, the nowcast quarter itself, and up to nine quarters into the future. Thus, all the values in a single row refer to the same forecast date, yet different forecast horizons.

Let's define **information date** as the specific date when a forecast is generated, and **realization date** as the exact time period the forecast is referring to. In the row format data structure, each row in the data set corresponds to a specific information date, with the columns indicating different forecast horizons, and hence, different realization dates. This allows a comparative view of how the forecast relates to both current and past data. For instance, if a GDP growth forecast exceeds the nowcast, it suggests an optimistic outlook on that particular information date.

However, it is often useful to compare different information dates that all correspond to the same realization date. For example, the forecast error is calculated by comparing a forecast from an earlier information date with a backcast created at a later information date, while both the forecast and backcast relate to the same realization date. If the data is organized in **column format**, then each row pertains to a certain realization date, and the columns correspond to different information dates. Regardless of whether we import the data in row or column format, we need to be able to manipulate the data set to compare both different information and realization dates with each other.

5.4.4 Plotting Forecasts

Let's use the `plot()` function to visualize some of the real GDP growth forecasts. We will plot the Tealbook forecasts for three different information dates, where the x-axis represents the realization date and the y-axis represents the real GDP growth forecasts made at the information date:

```
# Extract the real GDP growth worksheet
gRGDP <- TB[["gRGDP"]]

# Create a date column capturing the information date
gRGDP$date_info <- as.Date(as.character(gRGDP$GBdate), format = "%Y%m%d")

# Select information dates to plot
dates <- as.Date(c("2008-06-18", "2008-12-10", "2009-08-06"))
```

```
# Plot forecasts of second information date
plot(x = as.yearqtr(dates[2]) + seq(-4, 9) / 4,
     y = as.numeric(gRGDP[gRGDP$date_info == dates[2], 2:15]),
     type = "l", col = 5, lwd = 5, lty = 1, ylim = c(-7, 5),
     xlim = as.yearqtr(dates[2]) + c(-6, 8) / 4,
     xlab = "Realization Date", ylab = "%",
     main = "Back-, Now-, and Forecasts of Real GDP Growth")
abline(v = as.yearqtr(dates[2]), col = 5, lwd = 5, lty = 1)

# Plot forecasts of first information date
lines(x = as.yearqtr(dates[1]) + seq(-4, 9) / 4,
      y = as.numeric(gRGDP[gRGDP$date_info == dates[1], 2:15]),
      type = "l", col = 2, lwd = 2, lty = 2)
abline(v = as.yearqtr(dates[1]), col = 2, lwd = 2, lty = 2)

# Plot forecasts of third information date
lines(x = as.yearqtr(dates[3]) + seq(-4, 9) / 4,
      y = as.numeric(gRGDP[gRGDP$date_info == dates[3], 2:15]),
      type = "l", col = 1, lwd = 1, lty = 1)
abline(v = as.yearqtr(dates[3]), col = 1, lwd = 1, lty = 1)

# Add legend
legend(x = "bottomright", legend = format(dates, format = "%b %d, %Y"),
       title="Information Date",
       col = c(2, 5, 1), lty = c(2, 1, 1), lwd = c(2, 5, 1))
```

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

- `x = as.yearqtr(dates[2]) + seq(-4, 9) / 4`: This argument specifies the x-values for the line plot. `as.yearqtr(dates[2])` converts the second date in the `dates` vector into a quarterly format. `seq(-4, 9) / 4` generates a sequence of quarterly offsets, which are added to the second date to generate the x-values.
- `y = as.numeric(gRGDP[gRGDP$date_info == dates[2], 2:15])`: This argument specifies the y-values for the line plot. It selects the back-, now-, and forecasts (columns 2 through 15) from the `gRGDP` dataframe for rows where the `date_info` column equals the second date in the `dates` vector.
- `type = "l"`: This argument specifies that the plot should be a line plot.
- `col = 5`: This argument sets the color of the line plot. `5` corresponds to magenta in R's default color palette.
- `lwd = 5`: This argument specifies the line width. The higher the value, the thicker the line.
- `lty = 1`: This argument sets the line type. `1` corresponds to a solid line.
- `ylim = c(-7, 5)`: This argument sets the y-axis limits.
- `xlim = as.yearqtr(dates[2]) + c(-6, 8) / 4`: This argument sets the x-axis limits. The limits are calculated in a similar way to the x-values, but with different offsets.
- `xlab = "Realization Date"` and `ylab = "%"`: These arguments label the x-axis and y-axis, respectively.
- `main = "Back-, Now-, and Forecasts of Real GDP Growth"`: This argument sets the main title of the plot.
- `abline(v = as.yearqtr(dates[2]), col = 5, lwd = 5, lty = 1)`: This function adds a vertical line to the plot at the second date in the `dates` vector. The color, line width, and line type of the vertical line are specified by the `col`, `lwd`, and `lty` arguments, respectively.
- `lines`: This function is used to add additional lines to the plot, corresponding the the first and third date in the `dates` vector.
- `legend`: This function adds a legend to the plot. The `x` argument specifies the position of the legend,

the `legend` argument specifies the labels, and the `title` argument specifies the title of the legend. The color, line type, and line width of each label are specified by the `col`, `lty`, and `lwd` arguments, respectively.

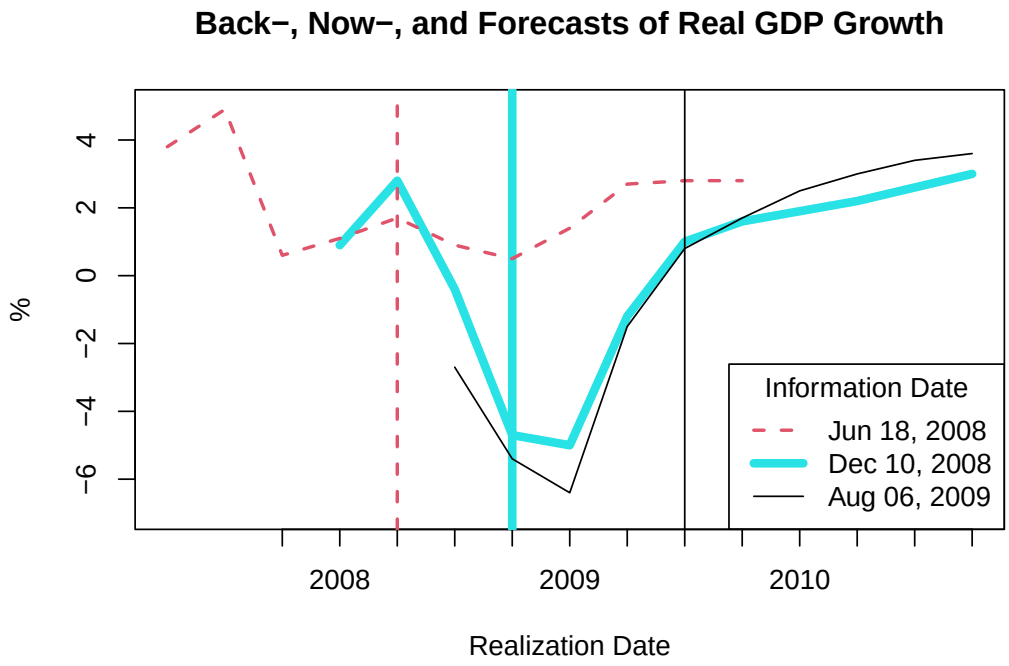


Figure 10: Back-, Now-, and Forecasts of Real GDP Growth

The resulting plot, as shown in Figure 10, exhibits backcasts, nowcasts, and forecasts derived at three distinct time periods:

1. The dashed red line corresponds to the backcasts, nowcasts, and forecasts made on June 18, 2008.
2. The thick magenta line represents the backcasts, nowcasts, and forecasts made on December 10, 2008.
3. The solid black line signifies the backcasts, nowcasts, and forecasts made on August 06, 2009.

The vertical lines depict the three information dates.

This figure indicates that considerable revisions occurred between June 18 and December 10, 2008, implying that the Federal Reserve did not anticipate the Great Recession. While the revisions between December 10, 2008 and August 06, 2009 are less drastic, it's noteworthy that there are considerable revisions in the December 10, 2008 nowcasts and backcasts. Consequently, the Federal Reserve had to operate based on incorrect information about growth in the current and preceding periods.

5.4.5 Plotting Forecast Errors

The forecast error for a given period is computed by subtracting the forecasted value from the actual value. This error is determined using an h-step ahead forecast, represented as follows:

$$e_{t,h} = y_t - f_{t,h}$$

In this equation, $e_{t,h}$ stands for the forecast error, y_t is the actual value, $f_{t,h}$ is the forecasted value, and t designates the realization date. The information date of the h-step ahead forecast $f_{t,h}$ is $t-h$, and for y_t it is either t or $t+k$ with $k>0$ depending on whether the value is observed like asset prices or revised post realization like inflation or GDP growth.

It's essential to note that the forecast error compares the same realization date with the values of two different information dates. For simplification, we assume that $y_t = f_{t,-1}$; that is, the 1-step behind backcast is a good approximation of the actual value.

When utilizing the Tealbook inflation data in row format, one might infer that the forecast error is computed by subtracting `gPGDPB1` from `gPGDPF1` for each row. However, this would be incorrect as it results in $y_{t-1} - f_{t+1,h}$, where the information date for both variables is t, but the realization dates are $t-1$ and $t+1$ respectively, and thus do not match.

Therefore, we need to shift the backcast `gPGDPB1` one period forward, and the one-step ahead forecast `gPGDPF1` one period backward to compute the one-step ahead forecast error. However, as the realization periods are quarterly, and there are eight FOMC (Federal Open Market Committee) meetings per year, each necessitating Tealbook forecasts, we need to aggregate the information frequency to quarterly as well.

The R code for performing these operations and computing the forecast error is as follows:

```
# Extract the GDP deflator (inflation) worksheet
gPGDP <- TB[["gPGDP"]]

# Create a date column capturing the information date
gPGDP$date_info <- as.Date(as.character(gPGDP$GBdate),
                           format = "%Y%m%d")

# Aggregate information frequency to quarterly
gPGDP$date_info_qtr <- as.yearqtr(gPGDP$date_info)
gPGDP_qtr <- aggregate(
    x = gPGDP,
    by = list(date = gPGDP$date_info_qtr),
    FUN = last
)

# Lag the one-step ahead forecasts
gPGDP_qtr$gPGDPF1_lag1 <- c(
    NA,
    gPGDP_qtr$gPGDPF1[-nrow(gPGDP_qtr)]
)

# Lead the one-step behind backcasts
gPGDP_qtr$gPGDPB1_lead1 <- c(gPGDP_qtr$gPGDPB1[-1], NA)

# Compute forecast error
gPGDP_qtr$error_F1_B1 <- gPGDP_qtr$gPGDPB1_lead1 -
    gPGDP_qtr$gPGDPF1_lag1
```

To visualize the inflation forecast errors, we use the `plot()` function as demonstrated below:

```
# Plot inflation forecasts for each realization date
plot(x = gPGDP_qtr$date, y = gPGDP_qtr$gPGDPF1_lag1,
     type = "l", col = 5, lty = 1, lwd = 3,
     ylim = c(-3, 15),
     xlab = "Realization Date", ylab = "%",
     main = "Forecast Error of Inflation")
abline(h = 0, lty = 3)

# Plot inflation backcasts for each realization date
lines(x = gPGDP_qtr$date, y = gPGDP_qtr$gPGDPB1_lead1,
      type = "l", col = 1, lty = 1, lwd = 1)
```

```r
# Plot forecast error for each realization date
lines(x = gPGDP_qtr$date, y = gPGDP_qtr$error_F1_B1,
      type = "l", col = 2, lty = 1, lwd = 1.5)

# Add legend
legend(x = "topright",
       legend = c("Forecasted Value",
                  "Actual Value (Backcast)",
                  "Forecast Error"),
       col = c(5, 1, 2), lty = c(1, 1, 1),
       lwd = c(3, 1, 1.5))
```

The customization of the plot is achieved using a variety of parameters, as discussed earlier.

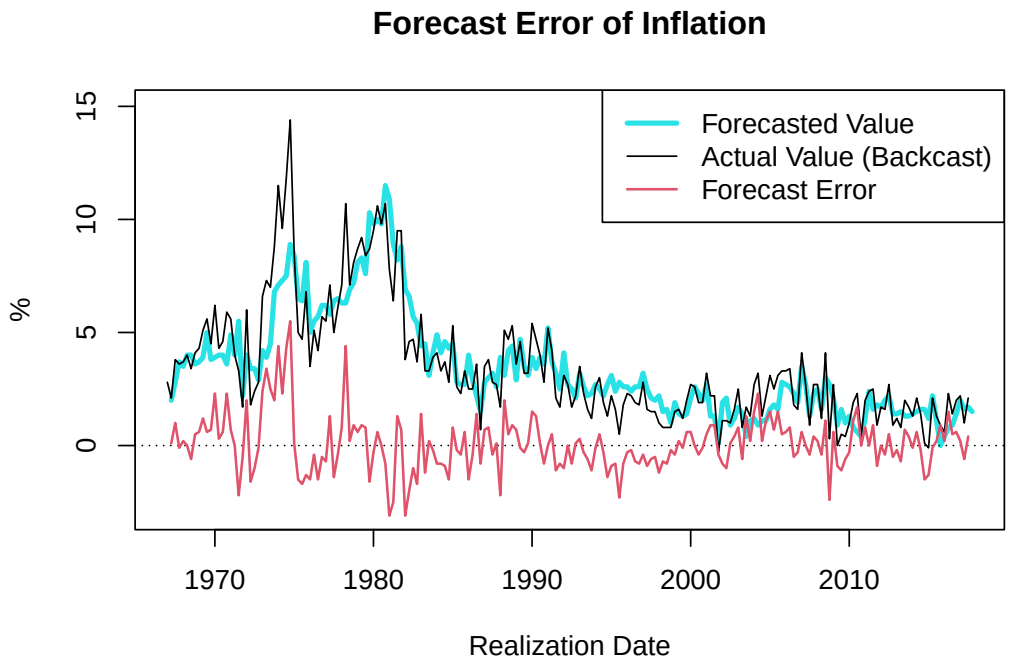


Figure 11: Forecast Error of Inflation

The resulting plot, shown in Figure 11, showcases the one-quarter ahead forecast error of inflation (GDP deflator) since 1967. A notable observation is the decrease in the forecast error variance throughout the "Great Moderation" period, which spanned from the mid-1980s to 2007.

The term "Great Moderation" was coined by economists to describe the substantial reduction in the volatility of business cycle fluctuations, which resulted in a more stable economic environment. Multiple factors contributed to this trend. Technological advancements, structural changes, improved inventory management, financial innovation, and, crucially, improved monetary policy, are all credited for this period of economic stability.

Monetary policies became more proactive and preemptive during this period, largely due to a better understanding of the economy and advances in economic modeling. These models, equipped with improved data, allowed for more accurate forecasts and enhanced decision-making capabilities. Consequently, policy actions were often executed before economic disruptions could materialize, thus moderating the volatility of the business cycle.

It is worth noting that the forecast error tends to hover around zero. If the forecast error were consistently smaller than zero, one could simply lower the forecasted value until the forecast error reaches zero, on average,

consequently improving the accuracy of the forecasts. Thus, in circumstances where the objective is to minimize forecast error, it is reasonable to expect that the forecast error will average out to zero over time.

The following calculations are in line with the description provided above:

```
# Compute mean of forecast error
mean(gPGDP_qtr$error_F1_B1, na.rm = TRUE)
```

```
## [1] 0.07871287
```

```
# Compute volatility of forecast error (FE)
sd(gPGDP_qtr$error_F1_B1, na.rm = TRUE)
```

```
## [1] 1.237391
```

```
# Compute volatility of FE before Great Moderation
sd(gPGDP_qtr$error_F1_B1[gPGDP_qtr$date <= as.yearqtr("1985-01-01")],
   na.rm = TRUE)
```

```
## [1] 1.728324
```

```
# Compute volatility of FE at & after Great Moderation
sd(gPGDP_qtr$error_F1_B1[gPGDP_qtr$date > as.yearqtr("1985-01-01")],
   na.rm = TRUE)
```

```
## [1] 0.8499342
```

6 Downloading Data in R

If your computer is connected to the internet, you can directly download data within R, bypassing the need to download data files and import the data in R as described in Chapter 5. This chapter focuses on how to download (and save) financial and economic data in R using an Application Programming Interface (API). Primarily, we'll be using two web APIs: `getSymbols()` from the `quantmod` package and `Quandl()` from the `Quandl` package.

6.1 Web API

A Web API or Application Programming Interface is a tool that allows software applications to communicate with each other. It enables users to download or upload data from or to a server.

There are several economic databases that provide web APIs:

- International Monetary Fund (IMF) data can be downloaded in R directly through the `imfr` package
- Bank for International Settlements (BIS) data can be accessed using the `BIS` package
- Organization for Economic Co-operation and Development (OECD) data is available through the `OECD` package

You can find more APIs for specific databases by searching "*r web api name_of_database*" in Google.

6.2 Using `getSymbols()`

The `getSymbols()` function, part of the `quantmod` package, enables access to various data sources.

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

# Download Apple stock prices from Yahoo Finance
AAPL_stock_price <- getSymbols(
  Symbols = "AAPL",
  src = "yahoo",
```

```
  auto.assign = FALSE,
  return.class = "xts")

# Display the most recent downloaded stock prices
tail(AAPL_stock_price)
```

```
##            AAPL.Open AAPL.High AAPL.Low AAPL.Close AAPL.Volume
## 2023-07-10    189.26    189.99   187.04     188.61    59922200
## 2023-07-11    189.16    189.30   186.60     188.08    46638100
## 2023-07-12    189.68    191.70   188.47     189.77    60750200
## 2023-07-13    190.50    191.19   189.78     190.54    41342300
## 2023-07-14    190.23    191.18   189.63     190.69    41573900
## 2023-07-17    191.90    194.32   191.81     193.99    50437300
##            AAPL.Adjusted
## 2023-07-10        188.61
## 2023-07-11        188.08
## 2023-07-12        189.77
## 2023-07-13        190.54
## 2023-07-14        190.69
## 2023-07-17        193.99
```

The arguments in this context are:

- `Symbols`: Specifies the instrument to import. This could be a stock ticker symbol, exchange rate, economic data series, or any other identifier.
- `auto.assign`: When set to FALSE, data must be assigned to an object (here, `AAPL_stock_price`). If not set to FALSE, the data is automatically assigned to the `Symbols` input (in this case, `AAPL`).
- `return.class`: Determines the type of data object returned, e.g., `ts`, `zoo`, `xts`, or `timeSeries`.
- `src`: Denotes the data source from which the symbol originates. Some examples include "yahoo" for Yahoo! Finance, "google" for Google Finance, "FRED" for Federal Reserve Economic Data, and "oanda" for Oanda foreign exchange rates.

To find the correct symbol for financial data from **Yahoo! Finance**, visit finance.yahoo.com and search for the company of interest, e.g., Apple. The symbol is the series of letters inside parentheses. For example, for `Apple Inc. (AAPL)`, use `Symbols = "AAPL"` (and `src = "yahoo"`). Yahoo! Finance provides stock price data in various columns, such as `AAPL.Open`, `AAPL.High`, `AAPL.Low`, `AAPL.Close`, `AAPL.Volume`, and `AAPL.Adjusted`. These values represent the opening, highest, lowest, and closing prices for a given market day, the trade volume for that day, and the adjusted value, which accounts for stock splits. Typically, the `AAPL.Adjusted` column is of most interest.

For economic data from **FRED**, visit fred.stlouisfed.org, and search for the data series of interest, e.g., unemployment rate. Click on one of the suggested data series. The symbol is the series of letters inside parentheses. For example, for `Unemployment Rate (UNRATE)`, use `Symbols = "UNRATE"` (and `src = "FRED"`).

The downloaded data is returned as an `xts` object, a format we discuss in Chapter 4.8. Let's visualize the Apple stock prices using the `plot()` function:

```
# Extract adjusted Apple share price
AAPL_stock_price_adj <- Ad(AAPL_stock_price)

# Plot adjusted Apple share price
plot(AAPL_stock_price_adj, main = "Apple Share Price")
```

Observe that this plot differs from those created in previous chapters where data was stored as a tibble (`tbl_df`) object. This discrepancy arises because `plot()` is a **wrapper function**, changing its behavior based on the type of object used. As `xts` is a time series object, the `plot()` function when applied to an `xts` object is optimized for visualizing time series data. Therefore, it's no longer necessary to specify it as a

Figure 12: Apple Share Price

line plot (`type = "l"`), as most time series are visualized with lines. To explicitly use the `xts` type `plot()` function, you can use the `plot.xts()` function, which behaves identically as long as the input object is an `xts` object. If not, the function will attempt to convert the object first.

As we've explored in Chapter 4.8, an `xts` object is an enhancement of the `zoo` object, offering more features. My personal preference leans towards the `zoo` variant of `plot()`, `plot.zoo()`, over `plot.xts()` because it operates more similarly to how `plot()` does when applied to data frames. To utilize `plot.zoo()`, apply it to the `xts` object like so:

```
# Plot adjusted Apple share price using plot.zoo()
plot.zoo(AAPL_stock_price_adj,
         main = "Apple Share Price",
         xlab = "Date", ylab = "USD")
```

In this context, the inputs are as follows:

- `AAPL_stock_price_adj` is the `xts` object containing Apple Inc.'s adjusted stock prices, compatible with `plot.zoo()` thanks to `xts`'s extension of `zoo`.
- `main` sets the plot's title, in this case, "Apple Share Price".
- `xlab` and `ylab` respectively label the x-axis as "Date" and the y-axis as "USD", denoting stock prices in U.S. dollars.

The usage of `plot.zoo()` provides a more intuitive plotting method for some users, replicating `plot()`'s behavior with data frames more closely compared to `plot.xts()`.

The resulting plot, shown in Figure 13, visualizes the Apple share prices over the last two decades. Stock prices often exhibit exponential growth over time, meaning the value increases at a rate proportional to its current value. As such, even large percentage changes early in a stock's history may appear minor due to a lower initial price. Conversely, smaller percentage changes at later stages can seem more significant due to a higher price level. This perspective can distort the perception of a stock's historical volatility, making past changes appear smaller than they were. To more accurately represent these proportionate changes, many analysts prefer using logarithmic scales when visualizing stock price data:

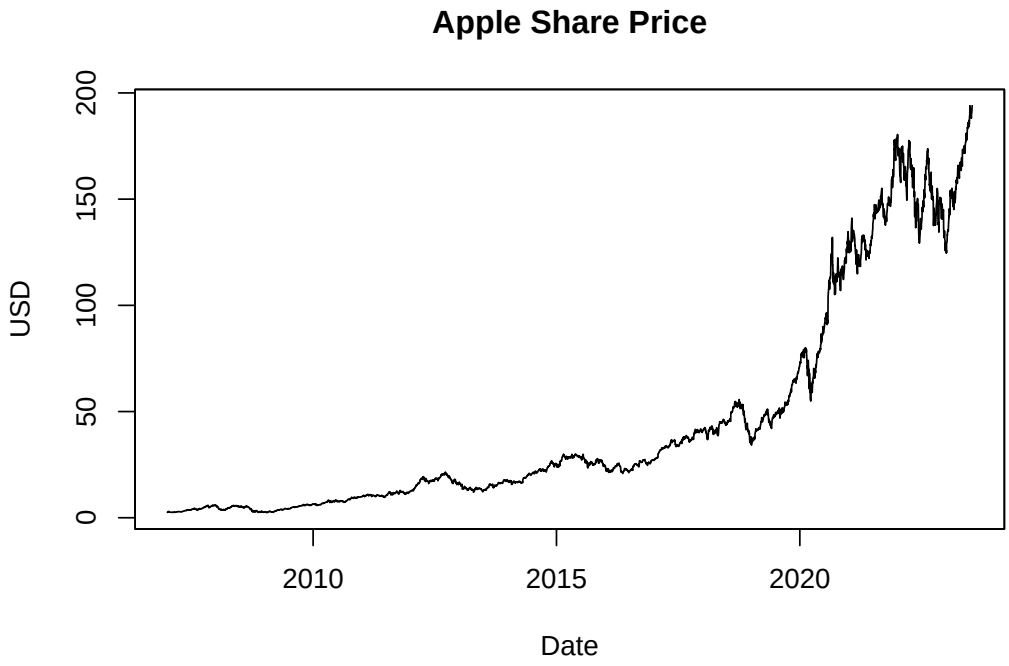


Figure 13: Apple Share Price Visualized with plot.zoo()

```
# Compute the natural logarithm (log) of Apple share price
log_AAPL_stock_price_adj <- log(AAPL_stock_price_adj)

# Plot log share price
plot.zoo(log_AAPL_stock_price_adj,
         main = "Logarithm of Apple Share Price",
         xlab = "Date", ylab = "Log of USD Price")
```

Figure 14 displays the natural logarithm of the Apple share prices. This transformation is beneficial because it linearizes the exponential growth in the stock prices, with the slope of the line corresponding to the rate of growth. More importantly, changes in the logarithm of the price correspond directly to percentage changes in the price itself. For example, if the log price increases by 0.01, this equates to a 1% increase in the original price. This property allows for a direct comparison of price changes over time, making it easier to interpret and understand the magnitude of these changes in terms of relative growth or decline.

6.3 Using `Quandl()`

Quandl is a data service that provides access to many financial and economic data sets from various providers. It has been acquired by Nasdaq Data Link and is now operating through the Nasdaq data website: data.nasdaq.com. Some data sets require a paid subscription, while others can be accessed for free. Quandl provides an API that allows you to import data from a wide variety of languages, including Excel, Python, and R.

To use Quandl, first get a free Nasdaq Data Link API key at data.nasdaq.com/sign-up. After that, install and load the Quandl package, and provide the API key with the function `Quandl.api_key()`:

```
# Load Quandl package
library("Quandl")

# Provide API key
api_quandl <- "CbtuSe8kyR_8qPgehZw3"  # Replace this with your API key
```

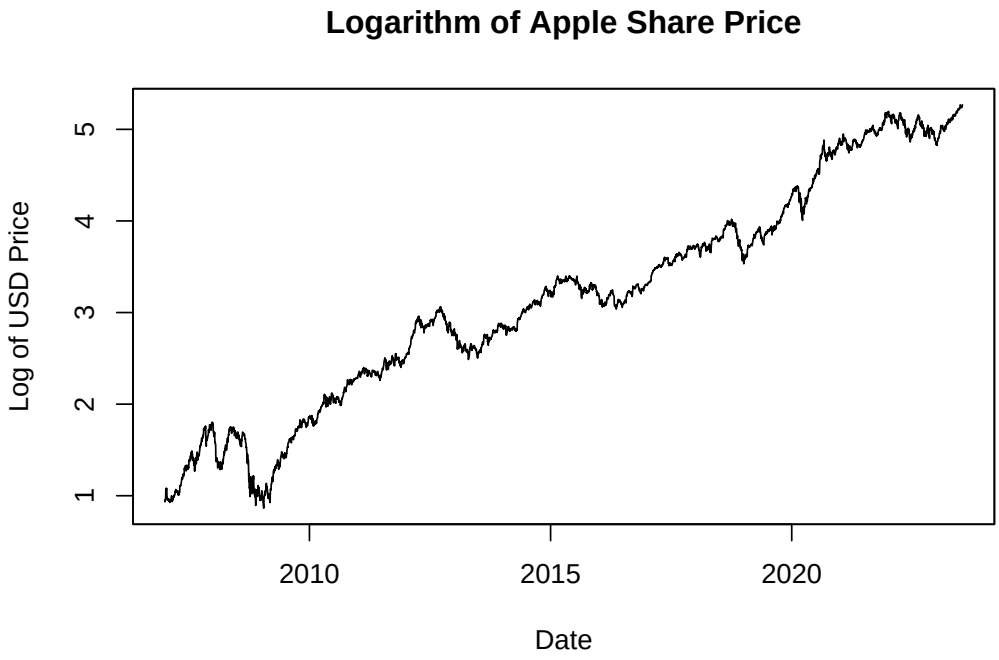


Figure 14: Logarithm of Apple Share Price

```
Quandl.api_key(api_quandl)
```

Then, you can download data using `Quandl()`:

```
# Download crude oil prices from Nasdaq Data Link
OPEC_oil_price <- Quandl(code = "OPEC/ORB", type = "xts")

# Display the downloaded crude oil prices
tail(OPEC_oil_price)
```

```
##             [,1]
## 2023-07-07 78.43
## 2023-07-10 79.09
## 2023-07-11 79.68
## 2023-07-12 80.64
## 2023-07-13 81.53
## 2023-07-14 82.06
```

Here, the `code` argument refers to both the data source and the symbol in the format "`source/symbol`". The `type` argument specifies the type of data object: `data.frame`, `xts`, `zoo`, `ts`, or `timeSeries`.

To find the correct code for the Quandl function, visit data.nasdaq.com, and search for the data series of interest (e.g., oil price). The code is the series of letters listed in the left column, structured like this: `.../...`, where `...` are letters. For example, for the OPEC crude oil price, use `code = "OPEC/ORB"`.

Now, we'll use the `plot.zoo()` function to graph the crude oil price:

```
# Plot crude oil prices
plot.zoo(OPEC_oil_price,
         main = "OPEC Crude Oil Price",
         xlab = "Date",
         ylab = "USD per Barrel (159 Litres)")
```

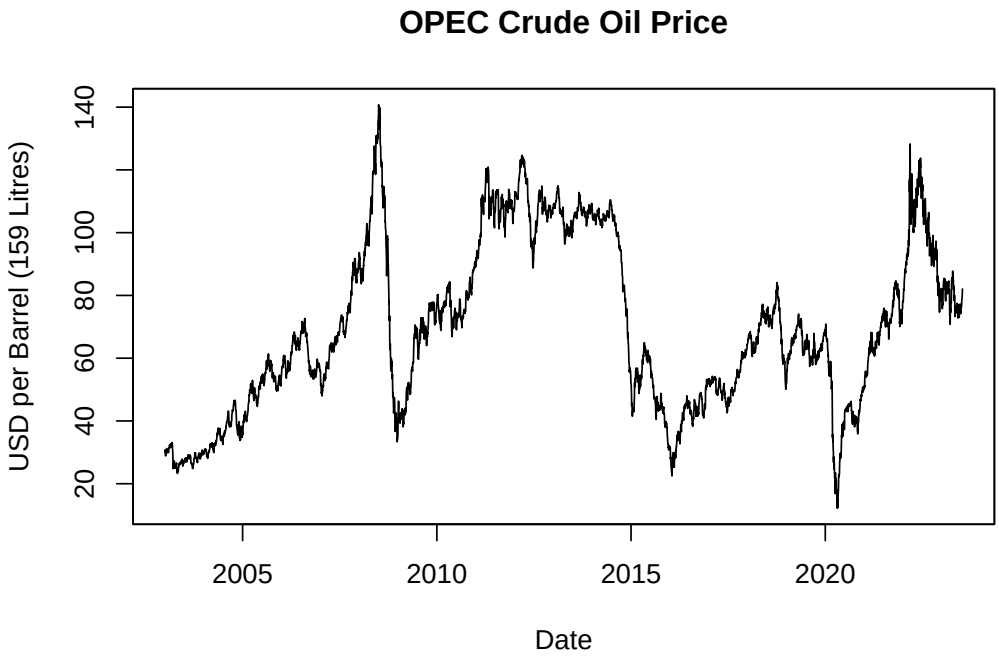


Figure 15: OPEC Crude Oil Price

Figure 15 displays the OPEC crude oil prices over the last two decades. While stock prices often exhibit long-term exponential growth due to company expansions and increasing profits, oil prices don't inherently follow the same pattern. The primary drivers of oil prices are supply and demand dynamics, shaped by a myriad of factors such as geopolitical events, production changes, advancements in technology, environmental policies, and shifts in global economic conditions. Oil prices, as a result, can undergo substantial fluctuations, exhibiting high volatility with periods of significant booms and busts, rather than consistent exponential growth.

Visualizing oil prices over time can benefit from the use of a logarithmic scale:

```
# Compute the natural logarithm (log) of crude oil prices
log_OPEC_oil_price <- log(OPEC_oil_price)

# Plot log crude oil prices
plot.zoo(log_OPEC_oil_price,
         main = "Logarithm of OPEC Crude Oil Price",
         xlab = "Date",
         ylab = "Log of USD per Barrel (159 Litres)")
```

The natural logarithm of crude oil price, depicted in Figure 16, offers a clearer view of relative changes over time. It does this by rescaling the data such that equivalent percentage changes align to the same distance on the scale, regardless of the actual price level at which they occur. This transformation means a change in the log value can be read directly as a percentage change in price. So, a decrease of 0.01 in the log price corresponds to a 1% decrease in the actual price, whether the initial price was \$10, \$100, or any other value. This makes the data more intuitive and straightforward to interpret, particularly when tracking price trends and volatility across various periods.

6.4 Saving Data

To ensure data preservation, it is recommended to save the data both before and after any transformations. Follow these steps to create a folder and save the objects:

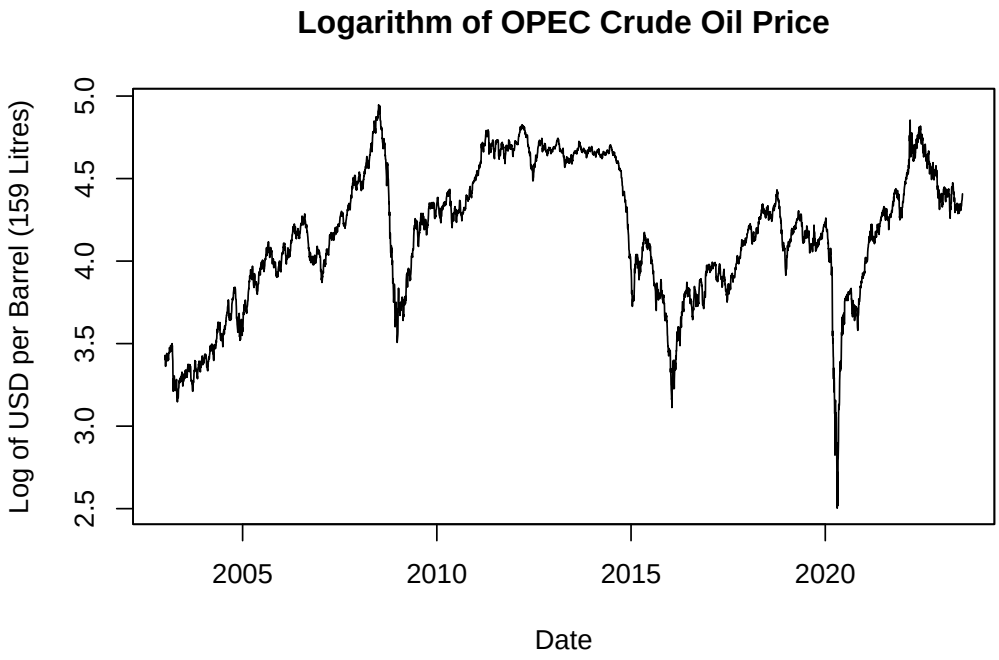


Figure 16: Logarithm of OPEC Crude Oil Price

```
# Create a folder named "downloads" for storing the files
dir.create("downloads")
```

Now, you can save the data in the `downloads` folder using the `RData` format, which is the default data format in R:

```
# Save the data object as an RData file
save(AAPL_stock_price, file = "downloads/AAPL.RData")
```

Once saved, `AAPL_stock_price` can be loaded into a different R session using the `load()` function:

```
load(file = "downloads/AAPL.RData")
```

This function creates an `xts` object named `AAPL_stock_price`, identical to the one we saved earlier.

For wider accessibility, especially for those not using R, the data can be saved as CSV, TSV, or Excel files:

```
# Load necessary packages
library("readr")  # For write_csv(), write_tsv(), and write_delim()
library("readxl") # For write_excel_csv()

# Convert the xts object to a data frame for saving
AAPL_df <- as.data.frame(AAPL_stock_price)

# Make column with date index, currently saved as row names
AAPL_df$Date <- rownames(AAPL_df)

# Position the date column first
AAPL_df <- AAPL_df[, c(ncol(AAPL_df), seq(1, ncol(AAPL_df) - 1))]

# Save data in various formats
write_csv(x = AAPL_df, file = "downloads/AAPL.csv", na = "NA")
write_tsv(x = AAPL_df, file = "downloads/AAPL.txt", na = "MISSING")
```

```
write_delim(x = AAPL_df, file = "downloads/AAPL.txt", delim = ";")
write_excel_csv(x = AAPL_df, file = "downloads/AAPL.xlsx", na = "")
```

These examples demonstrate how to save the `AAPL_df` data in:

1. CSV format, where missing values are labeled as NA (default).
2. TSV format, where missing values are labeled as MISSING.
3. A custom format with ";" as the delimiter.
4. Excel format, with missing cells left blank.

To save the file in a directory other than your current working directory (which can be determined using `getwd()`), replace `"AAPL.csv"` with the full path. For instance, you might use `file = "/Users/yourname/folder/AAPL.csv"`.

Refer to Chapter 5 for guidance on importing these CSV, TSV, and Excel data files into R.

For more in-depth understanding and examples, consider taking DataCamp's Introduction to Importing Data in R course.

6.5 Summary and Resources

This chapter has provided insights into multiple ways of downloading and saving data using various APIs. We specifically focused on the use of `getSymbols()` and `Quandl()` for downloading financial and economic data directly within R, then saving the data in different formats for future use.

To further enhance your knowledge and skills in working with data in R, consider the following DataCamp courses:

- Importing and Managing Financial Data in R: Gain a solid understanding of how to import and manage financial data in R.
- Intermediate Importing Data in R: Build upon your data importing skills and learn how to handle larger and more complex datasets.
- Working with Web Data in R: Learn to process and analyze data from the web using R.

7 Writing Reports with R Markdown

R Markdown is a tool for creating documents that combine text, R code, and the results of that R code. It simplifies the process of incorporating graphs and other data outputs into a document, removing the need for separate R and word processing operations. It allows for the automation of data retrieval and updating, making it useful for maintaining up-to-date financial reports, among other applications. With R Markdown, you can produce documents in various formats, including HTML, PDF, and Word, directly from your R code. Markdown facilitates the formatting of text in a plain text syntax, while embedded R code chunks ensure the reproducibility of analysis and reports.

7.1 Creating an R Markdown Document

Here is the step-by-step guide to create a new R Markdown document in RStudio:

1. Click on the top-left plus sign  , then select `R Markdown...`
2. In the dialog box that appears, select `Document` and choose `PDF`, then click `OK`.
3. You should now see a file populated with text and code. Save this file by clicking `File -> Save As...` and select an appropriate folder.
4. To generate a document from your R Markdown file, click `Knit`: 

(or use the shortcut Ctrl+Shift+K or Cmd+Shift+K).
5. Lastly, the `Knit` drop-down menu 

lets you export your file in different formats, such as HTML or Word, in addition to PDF.

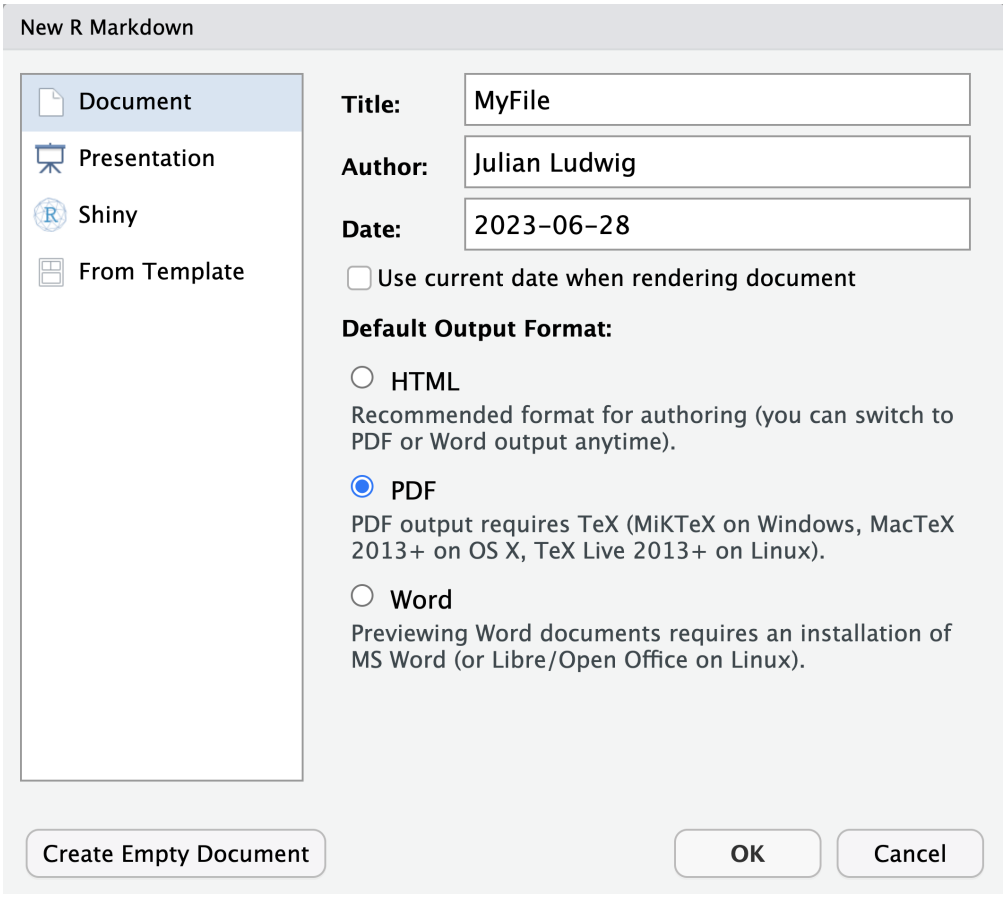


Figure 17: New R Markdown

The R Markdown template includes:

- A **YAML header**, enclosed by `---`, which holds the document's metadata, such as the title, author, date, and output format.
- Examples of **Markdown syntax**, demonstrating how to use it.
- Examples of **R code chunks**, showing how to write and utilize them in your document.

The R code chunks are enclosed by ```` ```{r} ```` at the beginning and ```` ``` ```` at the end, such as:

````
```{r cars}
summary(cars)
```
````

Anything written within these markers is evaluated as R code. On the other hand, anything outside these markers is considered text, formatted using Markdown syntax, and, for mathematical expressions, LaTeX syntax.

7.2 YAML Header

The YAML header at the top of the R Markdown document, enclosed in `---`, specifies high-level metadata and options that influence the whole document. It might look like this:

```
---
title: "My Document"
author: "Your Name"
date: "1/1/2023"
output: html_document
---
```

In this YAML header, the `title`, `author`, and `date` fields define the title, author, and date of the document. The `output` field specifies the output format of the document (which can be `html_document`, `pdf_document`, or `word_document`, among others).

7.3 Markdown Syntax

Markdown is a user-friendly markup language that enables the addition of formatting elements to plain text documents. The following are some fundamental syntax elements:

- Headers: `#` can be used for headers. For instance, `# Header 1` is used for a primary header, `## Header 2` for a secondary header, and `### Header 3` for a tertiary header, and so forth.
- **Bold**: To make text bold, encapsulate it with `**text**` or `__text__`.
- *Italic*: To italicize text, use `*text*` or `_text_`.
- Lists: For ordered lists, use `1.`, and for unordered lists, use `-` or `*`.
- Links: Links can be inserted using `[Link text](url)`.
- Images: To add images, use `![alt text](url)` for online images or `![alt text](path)` for local images, where `path` is the folder path to an image saved on your computer.

7.4 R Chunks

In R Markdown, you can embed chunks of R code. These chunks begin with ```` ```{r} ```` and end with ```` ``` ````. The code contained in these chunks is executed when the document is rendered, and the output (e.g., plots, tables) is inserted into the final document.

Following the `r` in the chunk declaration, you can include a variety of options in a comma-separated list to control chunk behavior. For instance, ```` ```{r, echo = FALSE} ```` runs the code in the chunk and includes its output in the document, but the code itself is not printed in the rendered document.

Here are some of the most commonly used chunk options:

- `echo`: If set to `FALSE`, the code chunk will not be shown in the final output. The default is `TRUE`.

- `eval`: If set to `FALSE`, the code chunk will not be executed. The default is `TRUE`.
- `include`: If set to `FALSE`, neither the code nor its results are included in the final document. The default is `TRUE`.
- `message`: If set to `FALSE`, suppresses all messages in the output. The default is `TRUE`.
- `warning`: If set to `FALSE`, suppresses all warnings in the output. The default is `TRUE`.
- `fig.cap`: Adds a caption to graphical results. For instance, `fig.cap="My Plot Caption"`.
- `fig.align`: Aligns the plot in the document. For example, `fig.align='center'` aligns the plot to the center.
- `out.width`: Controls the width of the plot output. For example, `out.width="50%"` will make the plot take up 50% of the text width.
- `collapse`: If `TRUE`, all the code and results in the chunk are rendered as a single block. If `FALSE`, each line of code and its results are rendered separately. The default is `FALSE`.
- `results`: The `results` argument provides options to control the display of chunk output in the final document. When set to `results='hide'`, the text output is concealed, while `results='hold'` displays the output after the code. Additionally, `results='asis'` allows direct inclusion of unmodified output, ideal for text or tables. `results='markup'` formats output as Markdown, for seamless integration into surrounding text, particularly useful when the R output is written in Markdown syntax. `results='verbatim'` displays the output as plain text, which is useful when the text includes special characters.
- `fig.path`: Specifies the directory where the figures produced by the chunk should be saved.
- `fig.width` and `fig.height`: Specifies the width and height of the plot, in inches. For example, `fig.width=6, fig.height=4` will make the plot 6x4 inches.
- `dpi`: Specifies the resolution of the plot in dots per inch. For example, `dpi = 300` will generate a high-resolution image.
- `error`: If `TRUE`, any error that occurs in the chunk will stop the knitting process. If `FALSE`, errors will be displayed in the output but will not stop the knitting process.

Here's an example:

````
```{r, echo=FALSE, fig.cap="Title", out.width = "50%", fig.align='center', dpi = 300}
plot(cars)
```
````

This chunk will create a plot, add a caption to it, set the width of the plot to 50% of the text width, align the plot to the center of the document, and output the plot with a resolution of 300 DPI. The actual R code will not be displayed in the final document.

Instead of specifying options for each code chunk, you can modify the default settings for all code chunks in your document using the `knitr::opts_chunk$set()` function. For instance, I often include the following code at the start of an R Markdown document, right after the YAML header:

````
```{r}
knitr::opts_chunk$set(echo = FALSE, message = FALSE, warning = FALSE,
                      fig.align = "center", out.width = "60%")
```
````

The aforementioned code modifies the default settings for all chunks in the document, as described below:

- `echo = FALSE`: Each chunk's code will be omitted from the final document, a sensible practice for official documents, as recipients don't require visibility of code used for graph creation.
- `message = FALSE`: All messages generated by code chunks will be muted.
- `warning = FALSE`: Warnings produced by code chunks will be silenced.
- `fig.align = "center"`: All generated figures will be centrally aligned.
- `out.width = "60%"`: The width of any generated figures will be set to 60% of the text width.

7.5 Embedding R Variables into Text

A key strength of R Markdown is the ability to incorporate R variables directly within the Markdown text. This enables a dynamic text where the values are updated as the variables change. You can accomplish this by using the `` `r variable` `` syntax. Furthermore, you can format these numbers for enhanced readability.

To insert the value of an R variable into your text, you encase the variable name in backticks and prepend it with `r`. Here's an illustration:

```
# R variable defined inside R chunk
my_var <- 123234.53983
```

To refer to this variable in your Markdown text, you can write the following text (outside of an R chunk):

```
The total amount is `r my_var` USD.
```

The output will be: "The total amount is 1.2323454×10^5 USD."

That's because when the R Markdown document is knitted, `` `r my_var` `` will be replaced by the current value of `my_var` in your R environment, dynamically embedding the value of `my_var` into your text.

Additionally, you can format numbers for better readability by avoiding scientific notation, rounding, and adding a comma as a thousands separator. To do this, you can use the `formatC()` function in R as follows:

```
# R variable with formatting, defined inside R chunk
my_var_formatted <- formatC(my_var, format = "f", digits = 2, big.mark = ",")
```

Then, in your text:

```
The total amount is `r my_var_formatted` USD.
```

The output will be: "The total amount is 123,234.54 USD."

In this case, `format = "f"` ensures fixed decimal notation, `digits = 2` makes sure there are always two decimal places, and `big.mark = ","` adds comma as the thousand separator.

By properly formatting your numbers in your R Markdown documents, you enhance their clarity and make your work more professional and easier to read.

7.6 LaTeX Syntax for Math

LaTeX is a high-quality typesetting system that is widely used for scientific and academic papers, particularly in mathematics and engineering. LaTeX provides a robust way to typeset mathematical symbols and equations. Thankfully, R Markdown supports LaTeX notation for mathematical formulas, which is rendered in the HTML output.

In R Markdown, you can include mathematical notation within the text by wrapping it with dollar signs (`$`). For example, `$a^2 + b^2 = c^2$` will be rendered as $a^2 + b^2 = c^2$.

Here are some basic LaTeX commands for mathematical symbols:

- Subscripts: To create a subscript, use the underscore (`_`). For example, `$a_i$` is rendered as a_i.
- Superscripts: To create a superscript (useful for exponents), use the caret (`^`). For example, `$e^x$` is rendered as e^x.
- Greek letters: Use a backslash (`\`) followed by the name of the letter. For example, `$\alpha$` is rendered as α, `$\beta$` as β, and so on.
- Sums and integrals: Use `\sum` for summation and `\int` for integration. For example, `$\sum_{i=1}^n i^2$` is rendered as $\sum_{i=1}^{n} i^2$ and `$\int_a^b f(x) dx$` is rendered as $\int_a^b f(x)dx$.
- Fractions: Use `\frac{numerator}{denominator}` to create a fraction. For example, `$\frac{a}{b}$` is rendered as $\frac{a}{b}$.
- Square roots: Use `\sqrt` for square roots. For example, `$\sqrt{a}$` is rendered as $\sqrt{a}$.

If you want to display an equation on its own line, you can use double dollar signs (`$$`). For example:

```
$$
\% \Delta Y_t
\equiv 100 \left( \frac{Y_t - Y_{t-1}}{Y_{t-1}}\right)   \%
\approx 100 \left( \ln Y_t - \ln Y_{t-1} \right) \%
$$
```

This will be rendered as:

$$\%\Delta Y_t \equiv 100 \left(\frac{Y_t - Y_{t-1}}{Y_{t-1}} \right) \% \approx 100 \left(\ln Y_t - \ln Y_{t-1} \right) \%$$

LaTeX and R Markdown together make it easy to include mathematical notation in your reports. With practice, you can write complex mathematical expressions and equations using LaTeX in your R Markdown documents.

7.7 Printing Tables

The R packages `kable` and `kableExtra` are great tools for creating professionally formatted tables in your R Markdown documents. Directly printing data without any formatting is not usually advisable as it lacks professionalism and can often be challenging to read and interpret. By contrast, these packages allow you to control the appearance of your tables, leading to better readability and aesthetics.

You'll first need to install and load the necessary packages. You can do so by executing `install.packages(c("knitr", "kableExtra"))` in your console and then load the two packages in the beginning of your code:

```
library("knitr")
library("kableExtra")
```

Let's assume we have a simple dataframe `df` that we want to print:

```
df <- data.frame(
  Name = c("Alice", "Bob", "Charlie"),
  Age = c(24, 30, 18),
  Gender = c("Female", "Male", "Male")
)
df
```

```
##      Name Age Gender
## 1   Alice  24 Female
## 2     Bob  30   Male
## 3 Charlie  18   Male
```

You can create a basic table using the `kable` function from the `knitr` package:

```
kable(df)
```

Name	Age	Gender
Alice	24	Female
Bob	30	Male
Charlie	18	Male

This will generate a simple, well-formatted table. However, you can further customize the table's appearance using functions from the `kableExtra` package:

```
df %>%
  kable() %>%
  kable_styling("striped", full_width = FALSE)
```

Name	Age	Gender
Alice	24	Female
Bob	30	Male
Charlie	18	Male

This code generates a striped table, which alternates row colors for easier reading. The `full_width = FALSE` argument ensures the table only takes up as much width as necessary.

Adding a caption to your table is straightforward. Simply provide the `caption` argument to the `kable` function:

```
df %>%
  kable(caption = "A Table of Sample Data") %>%
  kable_styling("striped", full_width = FALSE)
```

Table 2: A Table of Sample Data

Name	Age	Gender
Alice	24	Female
Bob	30	Male
Charlie	18	Male

This code generates the same striped table, but now with a caption: "A table of sample data."

These are just the basics. Both `kable` and `kableExtra` provide numerous options for customizing your tables. I encourage you to explore their documentation and experiment with different settings.

7.8 Summary and Resources

R Markdown provides a powerful framework for dynamically generating reports in R. The "dynamic" part of "dynamically generating reports" means that the document is able to update automatically when your data changes. By understanding and effectively using Markdown syntax, R code chunks, chunk options, and YAML headers, you can create sophisticated, reproducible documents with ease like the document you are currently reading.

For an in-depth understanding of R Markdown, you may want to delve into R Markdown: The Definitive Guide, an extensive resource on the topic. Additionally, DataCamp's course Reporting with R Markdown provides practical lessons on how to create compelling reports using this tool.

8 Learning R with DataCamp

DataCamp is an online learning platform that offers programming courses in R, among other languages. All university students can avail themselves of a complimentary 6-month license from DataCamp. If you are enrolled in my course, you should have received this license by email, but if you haven't, please reach out to me.

To start learning, first create an account on DataCamp using the student license you received. Once set up, I recommend beginning with the Introduction to R for Finance and Intermediate R for Finance courses.

8.1 Preparing for the Course

Before beginning a course on DataCamp, consider downloading and printing out the course slides. This will allow you to add your own notes as you work through the material. You can access the slides from the course dashboard by scrolling down to any of the chapters and clicking on `Continue Chapter` or `Completed` (if

you've already finished that chapter). Then, locate the `Show Slides` icon in the top-right corner and click on the download symbol.

As you progress through the course, try to apply what you're learning to a financial or economic dataset that interests you using the RStudio software on your desktop. You could use datasets such as the yield curve data (see Chapter 5.2), Michigan consumer survey data (see Chapter 5.3), tealbook forecaster data (see Chapter 5.4), or stock price data (see Chapter 6). This immediate application of newly acquired knowledge helps solidify understanding and allows you to save useful functions in your R script for future reference. This method is particularly effective for bridging the gap between course material and your area of interest, especially in more advanced courses. While the Introduction to R for Finance and Intermediate R for Finance courses mainly focus on introducing basic functions and don't necessarily involve specific datasets, this approach is highly beneficial for most other courses.

8.2 XP System and Course Completion

In DataCamp, each course is equipped with an experience point (XP) system. XP is a measure of your progress and engagement within a course. You earn XP by successfully completing exercises, practice sessions, and assessments within a course. The more XP you gain, the more it showcases your dedication and understanding of the subject matter.

Upon successfully completing a course, you receive a certificate from DataCamp, which is a testament to your accomplishment and newly acquired skills. These certificates can be shared on professional networking sites like LinkedIn or included in your CV, showcasing your dedication to learning and skill development.

Courses are divided into chapters that each focus on a specific topic. In turn, each chapter consists of numerous exercises. You complete a course by working through each exercise and solving them successfully. When ✓`Completed` appears beneath every chapter of a course, it signifies that you have finished the course. If you find an exercise challenging, you can opt for `Take Hint` to get some help or `Show Answer` to see the correct solution. However, it's most beneficial to grasp the concepts and find solutions independently.

Importantly, DataCamp allows unlimited course retakes. If you wish to reset a course, go to the course dashboard, select `Replay Course` (or `Continue Course`), then `Course Outline`, and finally `Reset Course Progress`. This way, if you're unsatisfied with your XP score in a course, you can simply redo it. The aim is to thoroughly understand the course content and develop the associated skills, not to rush through it. Learn at your own pace and remember to enjoy the learning journey!

8.3 Developing Your Skills

After completing the initial courses, I encourage you to delve into more courses and skill tracks to uncover the wide array of possibilities that R (and Python) offer. One effective method of exploring new courses is by following career or skill tracks. These are curated collections of courses designed to provide comprehensive knowledge in a specific field. You can find these tracks by navigating to app.datacamp.com/learn, clicking on `Tracks` in the left navigation pane, and then selecting either `Career Tracks` or `Skill Tracks`. You can then choose `R` for courses based on the R software, or select other programs.

For those studying Economics and Business, the following tracks are particularly pertinent:

- The complete skill track Finance Fundamentals in R
- The complete skill track Applied Finance in R
- The complete skill track Importing & Cleaning Data with R
- The complete skill track Data Visualization with R
- The complete skill track Interactive Data Visualization in R
- The complete career track Quantitative Analyst with R

For honing report writing skills, Reporting with R Markdown is recommended, and for those keen on Excel, the Spreadsheet Fundamentals skill track is worth considering.

9 Data Report on Yield Curve

You are tasked with preparing a data report on yield curves. The report should be prepared using R Markdown and must be a minimum of two pages long, containing both graphs and text.

Before you commence with the report, ensure you have completed the preparations detailed in Chapter 9.1. Your report should adhere to the guidelines set out in Chapter 9.2 and Chapter 9.3. Please submit your completed data report through the course website (Blackboard).

9.1 Preparations

As preparation for this assignment, please work through the following chapters of this book:

- Chapter 1: Software Overview
- Chapter 2: Software Installation
- Chapter 3: RStudio Interface
- Chapter 4: R Data Types and Structures
- Chapter 5: Importing Data in R
- Chapter 6: Downloading Data in R
- Chapter 7: Writing Reports with R Markdown
- Chapter 8: Learning R with DataCamp

In addition, complete the following courses on DataCamp as per the guidelines in Chapter 8:

- Introduction to R for Finance
- Intermediate R for Finance

9.2 Data Guidelines

Chapter 5.2 provides instruction on how to import yield curve data, as well as methods for visualizing and interpreting the data. For your report, plot both the most recent yield curve and the yield curve from your date of birth. If yield curve data is unavailable for your birth date (either due to being born prior to 1990 or on a non-trading day), use the data from the next available trading day.

Perform a comparative analysis of the yield curves from your birth date and the most recent one available. Leverage the insights provided in Chapter 5.2 to interpret the shapes of these yield curves. If the concept of yields and yield curves is new to you, consider conducting further online research. A good starting point for your research would be the U.S. Treasury website, from where you sourced your data.

9.3 R Markdown Guidelines

Your assignment is to write a data report on yield curves using R Markdown. If you need an introduction to R Markdown, you can refer to Chapter 7. Your final output, produced by knitting your R Markdown file, should be a professional, dynamic PDF document, following these standards:

- The final submission should be a PDF file. Submissions in R, R Markdown, or Microsoft Word formats will not be accepted.
- The PDF document should be directly generated from RStudio, not converted from another format such as a Microsoft Word file.
- Even though the report is dynamically generated using R chunks, the reader should not be aware of this. Hence, take care of the following:
  - To prevent the display of R messages (e.g., *Loading required package: xts*), set `message = FALSE` and `warning = FALSE` as default options. See Chapter 7.4 for more details.
  - To avoid showing R code in the PDF file (e.g., `PV <- FV / (1 + YTM)^4`), set `echo = FALSE` as the default option. Further information is provided in Chapter 7.4.
  - Avoid the use of hash tags in your document (e.g., ## 123.45), which usually appear when you print numbers or tables directly from an R chunk. Instead, integrate numbers within the Markdown

text as described in Chapter 7.5. And when it comes to printing tables, use the `kable` function, as explained in Chapter 7.7.
- Every number, plot, and table must be accurately labeled and accompanied by a descriptive text.
- All sentences should be grammatically correct and complete, with no missing verbs or unfinished thoughts.

Please follow these guidelines closely to ensure your report is professional, understandable, and visually appealing, offering an insightful presentation of the yield curve data.

Module II

Traditional Economic Indicators

The second module dives into various traditional economic indicators, such as GDP growth, inflation, and interest rates. It highlights their critical role in gauging the economic performance of a region over time and facilitating cross-regional performance comparisons.

Module Overview This module consists of nine chapters:

- Chapter 10: "Economic Indicators" provides an overview of key economic indicators such as GDP growth, inflation, and interest rates.
- Chapter 11: "Data Categorization" delves into different ways to classify and categorize economic data.
- Chapter 12: "Data Transformation" explores methods for modifying and manipulating data, to make it more suitable for analysis or to reveal hidden patterns.
- Chapter 13: "Data Aggregation" explains how to compile data from a lower frequency to a higher one, such as aggregating monthly data into quarterly or annual data.
- Chapter 14: "Data Source" provides an overview of various data providers and distributors, guiding you on where and how to access reliable economic and financial data.
- Chapter 15: "Temporal Patterns" illustrates how economic performance indicators can be used to discern economic business cycles, where periods of economic boom are followed by recessions.
- Chapter 16: "Regional Patterns" shows how economic performance indicators can be employed to compare different countries or regions within a country.
- Chapter 17: "Causal Relationships" explores how to identify and interpret cause-effect relationships in economic data.
- Chapter 18: "Data Report on Traditional Economic Indicators" serves as an assessment to examine the skills and knowledge gained throughout the module, focused around a data report on traditional economic indicators of your choice.

In addition, the following DataCamp courses supplement the content in Module II:

- Introduction to the Tidyverse
- Data Manipulation with dplyr
- (Optional: Reporting with R Markdown)

For guidelines on how to optimize your learning experience with these DataCamp courses, refer to Chapter 8.

Learning Objectives By the end of this module, you should be able to:

1. Understand the key macroeconomic indicators such as GDP growth, inflation, and interest rates.
2. Analyze time-series economic data to identify patterns and trends.
3. Perform cross-sectional comparisons of economic indicators to analyze differences between regions or countries.
4. Create a data report on traditional economic indicators using R and R Markdown.

Learning Activities & Assessments for Module II During this module, you will engage in the following activities:

1. **Reading Material**: Explore and understand the content of the nine chapters in Module II. These chapters will provide you with a firm understanding of key economic indicators and how they can be used to analyze economic performance.
2. **DataCamp Courses**: Complete the DataCamp courses Introduction to the Tidyverse and Data Manipulation with dplyr. These courses will equip you with skills to manipulate and visualize data effectively, enabling you to make insightful regional comparisons.
3. **Data Report**: Craft a comprehensive data report on traditional economic indicators of your choice using R and R Markdown. This task allows you to apply and demonstrate the knowledge and skills you've acquired throughout the module.

Your learning progress will be evaluated based on:

1. **DataCamp Course Completion**: Ensure completion of the assigned DataCamp courses, namely Introduction to the Tidyverse and Data Manipulation with dplyr. The progress and completion of these courses will be tracked and contribute to your overall assessment.
2. **Data Report on Traditional Economic Indicators**: Develop a comprehensive data report on traditional economic indicators of your choice using R Markdown. This report should demonstrate your understanding of R, RStudio, R Markdown, as well as key macroeconomic indicators. It should effectively illustrate your ability to analyze business cycles using time-series data and make regional comparisons using cross-sectional data. The quality and comprehensiveness of your report will form a significant part of your module assessment. Refer to Chapter 18 for more guidance on this task.

Keep in mind, gaining a solid understanding of economic indicators requires time and patience. Ensure to fully grasp each economic concept and its implications before advancing to the next. If you encounter any difficulties, remember, assistance is always available. Best of luck with your delve into economic indicators!

10 Economic Indicators

In the fields of economics and finance, various types of indicators provide valuable insights into the state and trajectory of economies and markets. These indicators range from macroeconomic measures that reflect the overall performance of an economy, to sentiment indices that capture the mood of investors:

- **Macroeconomic Indicators**: Macroeconomic indicators capture the overall performance and trends of an entire economy or a significant segment of it. These indicators include measures such as GDP, inflation rates, interest rates, national income, employment levels, and government debt. Macroeconomic indicators provide policymakers, analysts, and stakeholders with a broad understanding of the health and stability of an economy, enabling them to make informed decisions and formulate effective policies.
- **Microeconomic Indicators**: Microeconomic indicators focus on specific entities within the economy, such as individual firms, industries, or households. They provide insights into micro-level economic phenomena, such as market shares, production levels, consumer spending patterns, cost structures, and individual income levels. Microeconomic indicators help analyze the performance and dynamics of individual economic agents such as firms and consumers.
- **Market Indicators**: Market indicators track the movements of various types of market-based measures, such as equity prices, bond prices, commodity prices, and indices like the S&P 500 or FTSE 100. These indicators provide insights into the dynamics and sentiments prevailing in financial markets and the economy as a whole. Commodity prices, as an example, can signal trends in global economic health and are especially critical for countries heavily dependent on commodity exports or imports.
- **Sentiment Indices**: Sentiment indices reflect the overall attitude of investors towards a particular market or economy. These indices, which can be based on surveys or derived from market data, provide insights into the mood of investors and their expectations for the future. Examples include the Consumer

Confidence Index, the Investor Sentiment Index, and the Fear & Greed Index. By capturing the "mood" of the market, sentiment indices can help predict market trends and identify potential turning points.

- **Financial Indicators**: Financial indicators revolve around the functioning, stability, and performance of financial systems and markets. These indicators include measures such as stock market indices, exchange rates, bond yields, credit ratings, bank lending rates, and financial market volatility indices. They are crucial in assessing the strength and resilience of financial systems, identifying potential risks, and monitoring the effectiveness of monetary and regulatory policies. While indicators related to other non-financial industries, such as the production output of the manufacturing sector, typically fall under microeconomic indicators, financial indicators often overlap with macroeconomic indicators because they reflect conditions that affect all industries. That's because all sectors depend on the financial market for lending, borrowing, and liquidity.
- **Demographic Indicators**: Demographic indicators focus on population characteristics and dynamics. They encompass measures such as population size, age distribution, fertility rates, life expectancy, migration patterns, and labor force participation rates. Demographic indicators provide valuable insights into population trends, labor market dynamics, social challenges, and the potential impact on economic growth and development.
- **Environmental Indicators**: Environmental indicators highlight the relationship between economic activities and the environment. They encompass measures such as greenhouse gas emissions, energy consumption, water usage, waste generation, and natural resource depletion. Environmental indicators help assess the sustainability of economic development, the impact of industrial processes, and the potential risks posed by environmental degradation.
- **Social Indicators**: Social indicators focus on the well-being and quality of life of individuals within an economy. They encompass measures such as poverty rates, education levels, healthcare access, income inequality, and social mobility. Social indicators provide insights into the distribution of resources, social cohesion, and the effectiveness of social policies.

By understanding these indicators and how they interact, analysts, policymakers, and investors can make informed decisions and predictions.

11 Data Categorization

Before we delve into various economic indicators, it is essential to build a vocabulary of how data is categorized. This categorization is primarily based on the type of measurement and the dimensions of data:

- Qualitative vs. Quantitative: This refers to the nature of the data being collected, whether it's descriptive (qualitative) or numerical (quantitative).
- Discrete vs. Continuous: This classification is based on the quantifiable values that the data can take on, either as distinct and countable (discrete) or as an infinite range of values within a certain scale (continuous).
- Levels of Measurement: Data can also be classified based on the scale of measure it uses, such as nominal, ordinal, interval, or ratio.
- Index vs. Absolute Data: This categorization depends on whether the data values are meaningful in relative terms or absolute terms. Index data is meaningful in relation to other values of the same index, whereas absolute data values have meaning on their own, independently of other values.
- Stock vs. Flow: This dichotomy classifies data based on whether they represent a quantity at a specific point in time (stock) or a rate over a period (flow).
- Data Dimensions: This categorization refers to the way data is organized across temporal and spatial dimensions, including cross-sectional, time-series, panel, spatial, and clustered types of data.

Understanding these categories will allow us to more effectively interpret and analyze economic indicators.

11.1 Qualitative vs. Quantitative

In the world of data analysis and research, data is typically classified into two broad categories: Qualitative and Quantitative.

Qualitative data, often called categorical data, refers to non-numerical information that expresses the descriptive, subjective, or explanatory attributes of the variables under study. This data type includes factors such as colors, gender, nationalities, opinions, or any other attribute that does not have a natural numerical representation. In other words, qualitative data deals with characteristics and descriptors that can't be easily measured, but can be observed subjectively.

On the other hand, quantitative data is numerical and lends itself to mathematical and statistical manipulation. This data type involves numbers and things measurable in a quantifiable way. Examples of quantitative data include height, weight, age, income, temperature, or any variable that can be measured or counted. Quantitative data forms the backbone of any statistical analysis and tends to be more structured than qualitative data.

As an example, we can consider the `Affairs` dataset from the `AER` (Applied Econometrics with R) package, which is utilized for teaching and learning applied econometrics.

The `Affairs` dataset comes from a study by Ray C. Fair in 1978, published in the Journal of Political Economy, titled "A Theory of Extramarital Affairs". The study attempted to model the time spent in extramarital affairs as a function of certain background variables.

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

# Load the Affairs dataset
data("Affairs", package = "AER")

# Display a few rows of the Affairs dataset
tail(Affairs)
```

```
##      affairs gender age yearsmarried children religiousness
## 1935       7   male  47         15.0      yes             3
## 1938       1   male  22          1.5      yes             1
## 1941       7 female  32         10.0      yes             2
## 1954       2   male  32         10.0      yes             2
## 1959       2   male  22          7.0      yes             3
## 9010       1 female  32         15.0      yes             3
##      education occupation rating
## 1935        16          4      2
## 1938        12          2      5
## 1941        18          5      4
## 1954        17          6      5
## 1959        18          6      2
## 9010        14          1      5
```

This dataset comprises 601 observations and nine variables, with a mixture of four qualitative and five quantitative variables:

1. `affairs`: Quantitative - how much time is spent in extramarital affairs.
2. `gender`: Qualitative - gender of the respondents.
3. `age`: Quantitative - age of the respondents.
4. `yearsmarried`: Quantitative - number of years married.
5. `children`: Qualitative - whether the respondents have children or not.
6. `religiousness`: Qualitative - rating of religiousness, 1 being not religious at all, and 5 being very religious.

7. `education`: Quantitative - level of education in years of schooling.
8. `occupation`: Qualitative - category of the respondent's occupation.
9. `rating`: Quantitative - a self-rating of the marriage, 1 indicates very unhappy, and 5 indicates very happy.

Although certain qualitative variables are numerically coded, they remain qualitative in nature. They signify varying categories instead of directly measuring an attribute.

11.2 Discrete vs. Continuous

When dealing with quantitative data, it is further divided into two categories: Discrete and Continuous. The distinction between these two lies in the nature of the values or observations that the variables can assume.

Discrete data, as the name suggests, can only take certain, separated values. These values are distinct and countable, and there are no intermediate values between any two given values. Examples of discrete data include the number of students in a class, the number of cars in a parking lot, or any other count that cannot be divided into smaller parts.

Contrarily, continuous data can take any value within a given range or interval. Here, the data points are not countable as they may exist at any point along a continuum within a given range. This means that the variables have an infinite number of possibilities. Examples of continuous data include the height of a person, the weight of an animal, or the time it takes to run a race. In these examples, the measurements can be broken down into smaller units, and a great number of precise values are possible.

For instance, let's examine the `mtcars` dataset that comes with the R programming language. This dataset provides information on various attributes of 32 automobiles from the 1970s.

```
# View the first few rows of the mtcars dataset
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3
##                   carb
## Mazda RX4            4
## Mazda RX4 Wag        4
## Datsun 710           1
## Hornet 4 Drive       1
## Hornet Sportabout    2
## Valiant              1
```

One of the variables in this dataset, `mpg` (miles per gallon), is an example of a continuous variable. Continuous variables, such as `mpg`, can take on any value within a range. In this context, a car's fuel efficiency (measured in miles per gallon) can theoretically take any non-negative value. This makes `mpg` a continuous variable.

On the other hand, the `cyl` variable, representing the number of cylinders in a car's engine, is a discrete variable. Discrete variables can only take certain, distinct values. In the case of `cyl`, cars can have a whole number of cylinders—usually 4, 6, or 8. This characteristic makes `cyl` a discrete variable.

11.3 Levels of Measurement

Qualitative and quantitative data can be classified into four levels of measurement: nominal, ordinal, interval, and ratio These levels represent an increasing degree of precision in measurement. The first two levels,

nominal and ordinal, are applicable to qualitative data, while the last two levels, interval and ratio, are relevant to (discrete or continuous) quantitative data.

11.3.1 Nominal Scale Variables

Nominal scale variables are a type of **categorical variable** that represent distinct groups or categories without an inherent order or ranking. These variables essentially "name" the attribute and don't possess any quantitative significance. In the field of economics, an example of a nominal variable could be the industry sector of a company, such as technology, healthcare, manufacturing, and so on. These are distinct categories with no implied order or priority.

In R, nominal variables can be represented using the `factor()` function. Here's an example of how this can be done:

```
# Create a factor vector to represent industry sectors
sectors <- factor(c("technology", "healthcare", "manufacturing", "technology", "manufacturing"),
                  levels = c("technology", "healthcare", "manufacturing"))

# Display the created factor vector
print(sectors)
```

```
## [1] technology    healthcare    manufacturing technology
## [5] manufacturing
## Levels: technology healthcare manufacturing
```

In the above code snippet, the `factor()` function is used to convert a character vector into a factor vector, with the distinct industry sectors specified as the `levels`. This approach can enhance data analysis efficiency, as R internally assigns a distinct integer to each level. Hence, when dealing with large datasets, R can quickly refer to these numerical assignments instead of the corresponding character strings, improving processing speed.

The `levels()` function can be used to extract the defined levels of a factor vector, while the `as.numeric()` function can convert the factor levels to their corresponding numeric codes, as shown below:

```
# Extract defined levels of the factor vector
levels(sectors)
```

```
## [1] "technology"    "healthcare"    "manufacturing"
```

```
# Convert factor levels to their corresponding numeric codes
as.numeric(sectors)
```

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

In the output of this code, the `levels()` function will display the distinct industry sectors, while the `as.numeric()` function will present the numerical code assigned to each sector in the original vector.

11.3.2 Ordinal Scale Variables

Ordinal scale variables, like nominal variables, are categorical. However, ordinal scale variables have a clear ordering of the variables. An example in economics might be a credit rating (e.g., AAA, AA, A, BBB, BB). Here's an example:

```
# Creating a factor with ordered levels
credit_rating <- factor(c("AAA", "AA", "AAA", "BBB", "BB"),
                        levels = c("C", "B", "BB", "BBB", "A", "AA", "AAA"),
                        ordered = TRUE)
print(credit_rating)
```

```
## [1] AAA AA  AAA BBB BB
```

```
## Levels: C < B < BB < BBB < A < AA < AAA
```

In this code, the `ordered = TRUE` argument tells R that the levels should be considered in a specific order.

11.3.3 Interval Scale Variables

Interval scale variables are a type of **numerical variable** where the distance between two values is meaningful. However, they lack a true zero point. An example in economics is the difference between two credit scores. The difference between a score of 800 and 700 is the same as the difference between 700 and 600, but a score of 0 does not imply the absence of creditworthiness, it's just a lower bound.

Interval data lacks the absolute zero point, which makes direct comparisons of magnitude impossible. A person with a credit score of 800 is not twice as creditworthy as someone with a score of 400.

```
# Creating a numeric vector of credit scores
credit_scores <- c(800, 750, 700, 650, 600)
print(credit_scores)
```

```
## [1] 800 750 700 650 600
```

Another example of an interval scale variable is temperature when measured in Celsius or Fahrenheit. For instance, 20 degrees Celsius is not twice as hot as 10 degrees Celsius. Furthermore, 0 degrees Celsius doesn't imply the absence of temperature. Thus, the Celsius scale is an interval scale where the differences between values are meaningful, but it doesn't have a true zero point or a ratio relationship between different temperatures.

11.3.4 Ratio Scale Variables

Ratio scale variables, like interval scale variables, are numeric, but they have a clear definition of zero. When the value of a ratio scale variable is zero, it means the absence of the quantity. Examples in economics include income, annual sales, market share, unemployment rate, and GDP. When GDP is zero it means that no output is produced in that country during the year.

```
# Creating a numeric vector of GDP values in billions of dollars
gdp_billions <- c(19362, 21195, 22675, 21433, 22770)
print(gdp_billions)
```

```
## [1] 19362 21195 22675 21433 22770
```

The presence of a meaningful zero point allows us to make valid comparisons using ratios. For instance, one country's GDP being twice as large as another's is a meaningful statement.

Under normal circumstances, ratio scale variables cannot assume negative values, as this would imply a quantity less than nothing, which is nonsensical. Variables such as age, height, weight, or income are all examples of ratio scale variables as they can be compared using ratios (someone can be twice as old or earn thrice as much income as someone else), and their zero points are meaningful (zero age means no time has passed since birth, zero height or weight means an absence of these attributes, and zero income means no money has been earned). Therefore, negative values for these variables would not make sense.

However, there are certain exceptions where variables typically considered ratio scale variables can indeed be negative. These exceptions include instances such as net income, inflation, and interest rates. Here, a "negative" value does not imply less than nothing but rather denotes a particular condition or state. Negative net income reflects a state of indebtedness, negative inflation indicates a deflation, and negative interest rates imply a penalty for storing money at a bank. Despite being negative, these values can still be meaningfully compared to each other using ratios. For instance, 4% inflation is twice as high as 2% inflation, and deflation of 2% (equivalent to -2% inflation) is twice as high as deflation of 1% (or -1% inflation). However, one cannot meaningfully compare positive inflation to deflation using ratios. For example, 4% inflation cannot be compared using ratios to 2% deflation.

11.4 Index vs. Absolute Data

The field of economics and finance often uses two types of data for analysis: index data and absolute data. Unlike absolute data, which provides valuable insights on its own, index data is only meaningful when considered in relative terms.

11.4.1 Index Data

Index data, often referred to as **indices** or **indexes**, and not to be confused with indicators, are measures which values have no meaning on their own. An index value only becomes meaningful as comparison to a value at a different time period.

Stock market indices are prime examples of index data. These indices combine various stock prices into a normalized index. As part of this process, an arbitrary value, such as 100, is assigned to the index at a given point in time (e.g., 2010). If the index subsequently rises to 110, this indicates a 10% increase in the total price of the underlying stocks. However, without the context provided by the previous index value (100), the new value (110) lacks meaningful interpretation.

The reason for creating an index lies in the complexity of making raw values meaningful on their own. Simply adding all share prices together wouldn't provide valuable insights, as the share price of a particular company may be much higher than others due to arbitrary factors like the number of shares they've issued. Instead, stock market indices typically aggregate share prices by normalizing their contributions to the index based on factors such as the size of the company. During this aggregation process, the relative weights of the stocks carry importance, but the sum of these weights — whether it's 1, 100, or 34 — is irrelevant. This results in an index that lacks inherent meaning on its own, and thus, it only carries significance in relative terms. After completing this aggregation process, the index is standardized to an arbitrary value, often set at 100 at a specific moment in time.

11.4.2 Absolute Data

On the other hand, absolute data refers to data that has inherent meaning on its own, without needing comparison to other data.

A fitting example of absolute data is the Gross Domestic Product (GDP), which measures the total market value of all finished goods and services produced within a country's borders over a specific time period. A GDP value, for instance, $24,000 billion USD, provides a self-explanatory measurement. It indicates that, during that year, the country produced goods and services worth $24,000 billion USD, irrespective of previous values.

In summary, while index data offers relative comparisons that highlight trends and changes over time, absolute data also provides measurements that are meaningful on their own.

11.5 Stock vs. Flow

Economic and financial data can be broadly categorized into two types of variables: stock and flow.

11.5.1 Stock Variables

A **stock variable** represents a quantity measured at a single specific point in time. In a sense, they provide a snapshot of a specific moment. Here are examples of stock variables:

1. Wealth: The total accumulation of assets a person owns at a given point in time.
2. Debt: The total amount owed by a person, business, or country at a certain moment.
3. Population: The total number of people in a country or region at a given time.
4. Capital Stock: The total value of assets held by a firm at a point in time.
5. Unemployment Level: The total number of people unemployed at a specific point in time.
6. Inventory: The total amount of goods a company has in stock at a given time.
7. Reserves: The total amount of currency held by a central bank at a specific time.

8. Households' savings: The total amount of unconsumed income of a household at a point in time.

11.5.2 Flow Variables

A **flow variable**, on the other hand, is measured over an interval of time. Therefore, a flow would be measured per unit of time (say a year or a month). Here are examples of flow variables:

1. Income: Money that an individual or business receives over a period of time.
2. Spending: Money spent by individuals or businesses over a certain period.
3. Production: The total amount of goods and services produced over a time period.
4. Consumption: The total goods and services consumed over a period of time.
5. Investment: Money invested by a business over a time period.
6. Imports and Exports: Goods and services brought into or sent out of a country over a time period.
7. Government spending: Total expenditure by the government over a certain period.
8. Changes in inventory: The difference in inventory levels between two points in time.

11.5.3 Accounting

The concepts of stock and flow are also evident in the financial statements of companies. The **Balance Sheet** presents the stock of what a company owns (assets) and owes (liabilities) at a specific point in time, while the **Income Statement** depicts the flow of revenues and expenses that result in net income or loss over a particular period. This clear separation aids in understanding a company's financial position (stock) and performance (flow) separately, helping various stakeholders, including investors, creditors, and management, make informed decisions.

In conclusion, understanding the difference between stock and flow variables is crucial in economics, finance, and accounting, as it provides insights into a wide range of issues, from personal financial planning to the evaluation of a country's economic performance or a company's financial health.

11.6 Data Dimensions

Understanding the dimensions of a dataset is crucial in data analysis and econometrics. The dimensions of a dataset refer to how the observations are organized across different dimensions such as time and space.

There are five principal arrangements of data across dimensions in economic data:

1. Cross-Sectional
2. Time Series
3. Panel
4. Spatial
5. Clustered

Each type of data arrangement presents its own unique characteristics, challenges, and opportunities for data analysis.

11.6.1 Cross-Sectional Data

Cross-sectional data refer to information collected from multiple subjects at a specific point in time. In such data, each row represents a unique observation or individual. Examples of cross-sectional data are surveys and administrative records of persons, households, or firms. When analyzing this data, the standard assumption is that observations are independent, meaning that the ordering of observations does not matter.

Consider the `Ecdat` package in R, which contains numerous datasets used in econometrics textbooks. A good example of cross-sectional data is the `Wages` dataset from the `Ecdat` package. The dataset provides information about workers' wages and relevant characteristics.

```
# Load the Ecdat package
library(Ecdat)
```

```r
# Load the Wages dataset
data(Wages)

# Display the first few rows of the Wages dataset
head(Wages)
```

```
##   exp wks bluecol ind south smsa married  sex union ed black
## 1   3  32      no   0   yes   no     yes male    no  9    no
## 2   4  43      no   0   yes   no     yes male    no  9    no
## 3   5  40      no   0   yes   no     yes male    no  9    no
## 4   6  39      no   0   yes   no     yes male    no  9    no
## 5   7  42      no   1   yes   no     yes male    no  9    no
## 6   8  35      no   1   yes   no     yes male    no  9    no
##     lwage
## 1 5.56068
## 2 5.72031
## 3 5.99645
## 4 5.99645
## 5 6.06146
## 6 6.17379
```

In the `Wages` dataset, each row corresponds to a unique worker, providing details such as their education, work experience, region, gender, and union membership. The order of workers does not affect the interpretation of the data, thereby affirming the dataset's cross-sectional nature.

Another example of cross-sectional data is the `Hdma` dataset from the `Ecdat` package. The `Hdma` dataset comprises of information regarding home loans in Boston, as gathered by the Home Mortgage Disclosure Act (HMDA).

```r
# Load the Hdma dataset
data(Hdma)

# Display the first few rows of the Hdma dataset
head(Hdma)
```

```
##     dir   hir       lvr ccs mcs pbcr dmi self single uria
## 1 0.221 0.221 0.8000000   5   2   no  no   no     no  3.9
## 2 0.265 0.265 0.9218750   2   2   no  no   no    yes  3.2
## 3 0.372 0.248 0.9203980   1   2   no  no   no     no  3.2
## 4 0.320 0.250 0.8604651   1   2   no  no   no     no  4.3
## 5 0.360 0.350 0.6000000   1   1   no  no   no     no  3.2
## 6 0.240 0.170 0.5105263   1   1   no  no   no     no  3.9
##   comdominiom black deny
## 1           0    no   no
## 2           0    no   no
## 3           0    no   no
## 4           0    no   no
## 5           0    no   no
## 6           0    no   no
```

The `Hdma` dataset presents each row as a loan applicant, including details about the applicant's income, gender, race, loan specifications, and the decision made by the bank. As the ordering of applicants does not affect data interpretation, this dataset also qualifies as cross-sectional data.

Finally, consider the built-in `state.x77` dataset in R, sourced from the 1977 Statistical Abstract of the United States. The dataset consists of various economic and demographic measures of the U.S. states.

```r
# Display the US State Facts
head(state.x77)
```

```
##            Population Income Illiteracy Life Exp Murder HS Grad
## Alabama          3615   3624        2.1    69.05   15.1    41.3
## Alaska            365   6315        1.5    69.31   11.3    66.7
## Arizona          2212   4530        1.8    70.55    7.8    58.1
## Arkansas         2110   3378        1.9    70.66   10.1    39.9
## California      21198   5114        1.1    71.71   10.3    62.6
## Colorado         2541   4884        0.7    72.06    6.8    63.9
##            Frost   Area
## Alabama       20  50708
## Alaska       152 566432
## Arizona       15 113417
## Arkansas      65  51945
## California    20 156361
## Colorado     166 103766
```

The `state.x77` dataset comprises 50 rows and 8 columns, each row corresponding to one of the 50 US states. The columns represent a variety of measurements including the population, income, illiteracy rates, life expectancy, murder rates, high school graduation rates, frost frequency, and land area. The data is ordered alphabetically according to the state's name, but changing the order wouldn't affect interpretation, making this a cross-sectional dataset. Despite the geographic attributes of each state, the absence of specific geographic information (such as latitude and longitude) qualifies this dataset as cross-sectional rather than spatial data.

11.6.2 Time Series Data

Time series data are observations that are indexed by time. Each row in a time series dataset typically represents an observation at a distinct time point. Examples of this type of data are annual GDP, daily interest rates, and the hourly share price of Tesla. The key characteristic of time series data is that the ordering of observations does matter, and observations are not assumed to be independent of one another.

The `datasets` package in R, which is automatically installed and loaded with R, includes several built-in time series datasets that are commonly used for illustrative purposes. Consider the `AirPassengers` dataset as an example:

```r
# Air Passenger Monthly Totals (in thousands) from 1949 to 1960
AirPassengers
```

```
##      Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
## 1949 112 118 132 129 121 135 148 148 136 119 104 118
## 1950 115 126 141 135 125 149 170 170 158 133 114 140
## 1951 145 150 178 163 172 178 199 199 184 162 146 166
## 1952 171 180 193 181 183 218 230 242 209 191 172 194
## 1953 196 196 236 235 229 243 264 272 237 211 180 201
## 1954 204 188 235 227 234 264 302 293 259 229 203 229
## 1955 242 233 267 269 270 315 364 347 312 274 237 278
## 1956 284 277 317 313 318 374 413 405 355 306 271 306
## 1957 315 301 356 348 355 422 465 467 404 347 305 336
## 1958 340 318 362 348 363 435 491 505 404 359 310 337
## 1959 360 342 406 396 420 472 548 559 463 407 362 405
## 1960 417 391 419 461 472 535 622 606 508 461 390 432
```

The `AirPassengers` dataset includes monthly totals of international airline passengers from 1949 to 1960. Each row represents a particular month, and the order of the data matters due to potential temporal

dependencies, such as seasonality or trends.

Another example of a time series dataset is `EuStockMarkets` which includes daily closing prices of major European stock indices from 1991 to 1998:

```
# Load EuStockMarkets dataset
data(EuStockMarkets)

# Display the first few rows, and include the Date variable
head(data.frame(Date = time(EuStockMarkets), EuStockMarkets))
```

```
##       Date     DAX    SMI    CAC   FTSE
## 1 1991.496 1628.75 1678.1 1772.8 2443.6
## 2 1991.500 1613.63 1688.5 1750.5 2460.2
## 3 1991.504 1606.51 1678.6 1718.0 2448.2
## 4 1991.508 1621.04 1684.1 1708.1 2470.4
## 5 1991.512 1618.16 1686.6 1723.1 2484.7
## 6 1991.515 1610.61 1671.6 1714.3 2466.8
```

In this dataset, each row represents a trading day, and the order of the days is crucial to understanding trends and volatilities in the stock markets.

Lastly, consider the `longley` dataset, available in R's built-in datasets:

```
# Longley's Economic Regression Data (built-in dataset)
head(longley)
```

```
##      GNP.deflator     GNP Unemployed Armed.Forces Population Year
## 1947         83.0 234.289      235.6        159.0    107.608 1947
## 1948         88.5 259.426      232.5        145.6    108.632 1948
## 1949         88.2 258.054      368.2        161.6    109.773 1949
## 1950         89.5 284.599      335.1        165.0    110.929 1950
## 1951         96.2 328.975      209.9        309.9    112.075 1951
## 1952         98.1 346.999      193.2        359.4    113.270 1952
##      Employed
## 1947   60.323
## 1948   61.122
## 1949   60.171
## 1950   61.187
## 1951   63.221
## 1952   63.639
```

The `longley` dataset includes annual observations on six macroeconomic variables for the United States, spanning from 1947 to 1962. Each row stands for a year, and the ordering of the years carries substantial information about the economic progression of the country.

Aggregate economic data such as the `longley` dataset is typically available at low frequency (annual, quarterly, or perhaps monthly) so the sample size is quite small, while financial data such as the `EuStockMarkets` dataset is typically available at high frequency (weekly, daily, hourly, or per transaction).

In all of these examples, the order of observations matters significantly because it allows for the analysis of trends, cycles, and other time-dependent structures within the data. Proper handling of these elements is crucial for effective time series analysis. This often involves the application of statistical methods designed specifically for time series data, such as autoregressive integrated moving average (ARIMA) models and vector autoregression (VAR) models. These models take into account the temporal dependencies and provide a more reliable analysis of the data.

11.6.3 Panel Data

Panel data, also known as **longitudinal data** or **cross-sectional time series data**, combines elements of both cross-sectional and time series data. In panel data, observations are collected on multiple entities (such as individuals, households, firms, or countries) over multiple time periods. This data structure provides unique opportunities to analyze both individual and time effects, as well as their interaction.

To illustrate panel data, we can explore some examples using R datasets.

One example is the `Produc` dataset, which is available in the `plm` package:

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

# Load the Produc dataset
data("Produc", package = "plm")

# Display a few rows of the Produc dataset
Produc[15:20, ]
```

```
##      state year region     pcap     hwy   water    util       pc
## 15 ALABAMA 1984      6 19257.47 8655.94 2235.16 8366.37 59446.86
## 16 ALABAMA 1985      6 19433.36 8726.24 2253.03 8454.09 60688.04
## 17 ALABAMA 1986      6 19723.37 8813.24 2308.99 8601.14 61628.88
## 18 ARIZONA 1970      8 10148.42 4556.81 1627.87 3963.75 23585.99
## 19 ARIZONA 1971      8 10560.54 4701.97 1627.34 4231.23 24924.82
## 20 ARIZONA 1972      8 10977.53 4847.84 1614.58 4515.11 26058.65
##      gsp    emp unemp
## 15 45118 1387.7  11.0
## 16 46849 1427.1   8.9
## 17 48409 1463.3   9.8
## 18 19288  547.4   4.4
## 19 21040  581.4   4.7
## 20 23289  646.3   4.2
```

The `Produc` dataset contains state-level data for the United States from 1970 to 1986. It includes variables such as state GDP, capital stock, employment, and wages. With data from 48 states, each state having 17 yearly observations, this dataset combines both cross-sectional and time-series components. This panel structure allows for analyzing the relationship between these variables over time, while also accounting for heterogeneity across states.

Another example is the `EmplUK` dataset from the `plm` package:

```
# Load the EmplUK dataset
data("EmplUK", package = "plm")

# Display a few rows of the EmplUK dataset
EmplUK[19:24, ]
```

```
##    firm year sector    emp    wage capital   output
## 19    3 1981      7 19.570 24.8714  6.2136  99.5581
## 20    3 1982      7 18.125 24.8447  5.7146  98.6151
## 21    3 1983      7 16.850 28.9077  7.3431 100.0301
## 22    4 1977      8 26.160 14.8283  8.4902 118.2223
## 23    4 1978      8 26.740 14.8379  8.7420 120.1551
## 24    4 1979      8 27.280 14.8756  9.1869 118.8319
```

The `EmplUK` dataset contains information on employment in different industries in the United Kingdom from 1977 to 1983. It includes variables such as employment, wages, and output. This dataset allows for analyzing

how employment and wages vary across industries and over time.

In panel data analysis, various econometric techniques can be applied to account for time-varying effects, individual-specific heterogeneity, and potential endogeneity. Common methods include fixed effects models, random effects models, and dynamic panel data models. These approaches help uncover relationships that may be obscured in cross-sectional or time series analysis alone.

11.6.4 Spatial Data

Spatial data are observations that are indexed by geographical locations or coordinates. Each row in a spatial dataset typically represents an observation at a distinct spatial unit. Examples of this type of data are housing prices across different districts, climate data from different weather stations, and population densities across various regions. The key characteristic of spatial data is that the geographical arrangement of observations does matter, and observations are not assumed to be independent of one another, often exhibiting spatial autocorrelation where observations close in space tend to be more similar.

The `house` dataset, available in the `spdep` package, serves as a fitting example for spatial data analysis:

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

# Load the house dataset
data(house)

# Display a few rows and columns of the house dataset
head(house[, 1:6])
```

```
##            coordinates  price yrbuilt  stories  TLA     wall beds
## 1 (484668.1, 195270.3) 303000    1978 one+half 3273  partbrk    4
## 2 (484875.6, 195301.3)  92000    1957      one  920 metlvnyl    2
## 3 (485248.4, 195353.8)  90000    1937      two 1956 stucdrvt    3
## 4 (485764.2, 196177.5) 330000    1887 one+half 1430     wood    4
## 5 (488149.8, 196591.4) 185000    1978      two 2208  partbrk    3
## 6   (485525.7, 196660) 100000    1989      one 1232     wood    1
```

The `house` dataset consists of details pertaining to 25,357 single-family homes sold in Lucas County, Ohio, between 1993 and 1998. It includes particulars such as sale prices, locations, and other relevant features. Additionally, the dataset reveals the GPS coordinates of each house in the `coordinates` column. The first value in the `coordinates` column denotes the northern coordinate, while the second one refers to the western coordinate. Researchers can employ these GPS coordinates to conduct spatial analyses, examining housing market dynamics in Lucas County while taking into account spatial autocorrelation and spatial dependence.

It's important to note that the regional datasets previously discussed in the context of cross-sectional and panel data can also be classified as spatial data. To perform spatial data analysis on these regional datasets, one needs to merge them with another dataset that includes GPS or longitude and latitude data for each region. This geographical data is often available online; for instance, one can find the longitude and latitude for each U.S. state or county with relative ease.

By integrating geographic information and taking into account spatial dependencies, economists can improve the accuracy of their empirical results, gaining deeper insights into the influence of geographical factors on economic outcomes. Common methods include spatial lag models, spatial error models, and spatial autoregressive models. These approaches help reveal relationships that may be hidden when using cross-sectional or time series analysis alone.

11.6.5 Clustered Data

Clustered data is a form of data that may resemble panel or spatial data, as it groups observations based on criteria such as geographical location, time period, or subject. This clustering implies that observations

within each group are likely to exhibit a degree of correlation, meaning they are more likely to resemble each other than observations in different clusters.

However, the approach to clustered data contrasts with panel or spatial data in an important way: in clustered data, the specific nature of the correlation within each cluster is not typically explicitly modeled. For example, while we might expect houses sold within the same county (a cluster) to have more similar characteristics than houses sold across different counties, we don't explicitly model the specific relationships among the houses within each county so that the order of observations within clusters doesn't matter.

On the other hand, panel data has a time-dependent structure as it involves repeated observations collected from the same subjects over time. Similarly, spatial data has a location-dependent structure, where observations are based on their geographical locations. In both these cases, the dependencies within the data are explicitly incorporated into the data analysis.

Consider the `EmplUK` dataset as an example of clustered data. This dataset contains panel data on employment and wages for different industries in the United Kingdom from 1977 to 1987, as discussed earlier in the panel data section:

```
# Display a few rows of the EmplUK dataset
EmplUK[19:25, ]
```

```
##    firm year sector    emp    wage capital   output
## 19    3 1981      7 19.570 24.8714  6.2136  99.5581
## 20    3 1982      7 18.125 24.8447  5.7146  98.6151
## 21    3 1983      7 16.850 28.9077  7.3431 100.0301
## 22    4 1977      8 26.160 14.8283  8.4902 118.2223
## 23    4 1978      8 26.740 14.8379  8.7420 120.1551
## 24    4 1979      8 27.280 14.8756  9.1869 118.8319
## 25    4 1980      8 27.830 15.2332  9.4036 111.9164
```

The `EmplUK` dataset is clustered by industry, suggesting that observations within the same industry may exhibit similar employment patterns or be influenced by industry-specific factors. Depending on the research question and context, the `EmplUK` dataset could be treated as either panel data or clustered data.

When analyzing the `EmplUK` dataset to understand employment and wage variations across different industries over time, treating it as panel data is appropriate. In this context, each industry is a distinct unit in the panel data framework, and using panel data models allows researchers to study variations in employment and wages over time and across industries, accounting for both time effects and industry-specific effects.

In a different context, the `EmplUK` dataset could be treated as clustered data. Here, the interest lies in understanding the variation and potential correlations within each industry. The clusters here are the industries themselves, and observations within each industry might be more similar to each other due to industry-specific factors such as common economic conditions, regulations, or labor market dynamics. Treating the data as clustered acknowledges the correlation within each cluster (industry) but does not model this correlation explicitly.

Hence, the choice between treating the `EmplUK` dataset as panel data or clustered data depends on the research question.

When working with clustered data, it is vital to use statistical techniques that account for the correlation within clusters. Common techniques include the use of cluster-robust standard errors, fixed effects models, or random effects models.

11.7 Conclusion

In this chapter, we've developed a comprehensive vocabulary for understanding and categorizing data. The classifications include whether the data is described in words or numbers (qualitative or quantitative), the quantifiable values the data can assume (discrete or continuous), the scale of measurement (nominal, ordinal, interval, or ratio), its relational form (index vs. absolute data), its temporal nature (stock or flow), and its

dimensions (cross-sectional, time-series, panel, spatial, and clustered). Together, these classifications provide us with a robust framework to approach and interpret different types of data that we encounter in economic analysis.

Each of these categories has unique characteristics that require specific analytical methods and considerations. For example, time-series data are analyzed differently from cross-sectional data, considering their inherent temporal order and potential autocorrelation. Similarly, data transformations, such as logarithmic or difference transformations, may be more applicable or meaningful to certain types of data than others.

In the following chapters, we will apply this vocabulary to economic indicators and the transformations thereof. We will discuss how each economic indicator fits into these categories and how specific data transformations can enhance our understanding of these indicators. Remember, the appropriate transformation depends on the specific type of data and the research question at hand, so our newly established vocabulary will prove invaluable.

12 Data Transformation

In Economics and Finance, it is common to transform raw data into meaningful indicators. One common transformation is per capita GDP, where a country's GDP is divided by its population. This gives a more comparable measure of economic performance across different countries. Similarly, inflation transforms the price index into a growth rate of prices, which helps us understand price distortions, interest rates, and evaluate monetary policy. While these transformations enhance data comparability, they can also introduce complexity. This chapter explores common data transformations in Economics and Finance and offers insights on their interpretation.

Throughout this chapter, we will explore the following transformed measures:

- Growth: This term refers to how an economic variable changes compared to its original value over a set period. Often, it's expressed as a percentage.
- Differentiation: Differentiation measures the change between an economic variable's current value and its preceding value. This offers insights into the degree and direction of change.
- Natural Logarithm (Log): Logarithms have many uses, like simplifying data that shows exponential growth or quantifying proportional changes.
- Ratio: Ratio measures calculate the ratio between two variables, such as per capita GDP or unemployment rate. They provide information on the relative size or magnitude of one variable in relation to another.
- Gap: Gap measures calculate the distance between two variables, highlighting disparities.
- Filtering: These measures remove noise or specific fluctuations from the data, providing a clearer view of specific patterns.

By the end of this chapter, you will gain a solid understanding of these transformations and learn how to correctly interpret each one. But before we dive into the details, let's first prepare some example data that we'll use to illustrate these transformations throughout this chapter.

12.1 Example Data

For our example data, we leverage key macroeconomic indicators from the U.S. economy, which we source from the FRED database maintained by the Federal Reserve Bank of St. Louis. These indicators provide critical insights into the economic state of the nation, informing various financial, investment, and policy decisions.

The following indicators will be used as examples, where the symbols in parenthesis represent the FRED symbols for downloading the data in R:

- **Gross Domestic Product (GDP)** (`GDP`): This metric denotes the total market value of all the finished goods and services produced in the United States in a specific time period.

- **Unemployment Rate** (`UNRATE`): This measures the percentage of the total U.S. labor force that is unemployed but actively seeking employment and willing to work.
- **GDP Deflator (Aggregate Price Level)** (`GDPDEF`): This is a measure of the price level of all domestically produced, final goods and services in the United States.
- **3-Month Treasury Bill Rate** (`TB3MS`): This represents the return on investment for the U.S. government's short-term debt obligation that matures in three months.
- **NASDAQ Composite Index** (`NASDAQCOM`): This encompasses a broad range of the securities listed on the NASDAQ stock market and is typically viewed as an indicator of the performance of technology and growth companies.
- **Population** (`B230RC0Q173SBEA`): This refers to the total number of individuals residing within the United States.

We employ the `getSymbols()` function from the `quantmod` package in R to access the data. This function enables direct data download as delineated in Chapter 6. To use the `getSymbols()` function for data download, set the `Symbols` parameter of the function as `"GDP"` or `"UNRATE"` (the characters enclosed within the parentheses) and the `src` (source) parameter as `"FRED"`, corresponding to the FRED website:

```
# Load quantmod package
library("quantmod")

# Start date
start_date <- as.Date("1971-02-05")

# Download data
getSymbols("GDP", src = "FRED", from = start_date)
getSymbols("UNRATE", src = "FRED", from = start_date)
getSymbols("GDPDEF", src = "FRED", from = start_date)
getSymbols("TB3MS", src = "FRED", from = start_date)
getSymbols("NASDAQCOM", src = "FRED", from = start_date)
getSymbols("B230RC0Q173SBEA", src = "FRED", from = start_date)

# De-annualize GDP & use quarterly instead of monthly index
GDP <- GDP / 4
index(GDP) <- as.yearqtr(index(GDP))

# Remove missing NASDAQCOM values
NASDAQCOM <- na.omit(NASDAQCOM)
```

```
## [1] "GDP"
## [1] "UNRATE"
## [1] "GDPDEF"
## [1] "TB3MS"
## [1] "NASDAQCOM"
## [1] "B230RC0Q173SBEA"
```

The code block above, as discussed in Chapter 6, is responsible for data acquisition. The line `start_date <- as.Date("1971-02-05")` initializes a variable `start_date` with the date value of February 5, 1971. The function `as.Date()` converts the string input `"1971-02-05"` into a format that R can interpret and manipulate as a date object. This conversion process is explained in Chapter 4. Within the `getSymbols()` function, the argument `from = start_date` stipulates that data should be retrieved starting from the date encapsulated in the `start_date` variable, which in this instance is from February 5, 1971 onward. The data fetched by this function is returned as `xts` objects, a time series data format described in Chapter 4.8.

The following code block visualizes the data:

```
# Put all plots into one
par(mfrow = c(rows = 3, columns = 2),
```

```
    mar = c(b = 2, l = 4, t = 2, r = 1))

# Plot U.S. macro indicators, SA = seasonally adjusted
plot.zoo(x = GDP, main = "Gross Domestic Product (GDP)",
         xlab = "", ylab = "Billion-USD, SA")
plot.zoo(x = UNRATE, main = "Unemployment Rate",
         xlab = "", ylab = "Percent, SA")
plot.zoo(x = GDPDEF, main = "GDP Deflator (Aggregate Price Level)",
         xlab = "", ylab = "Index, 2012 = 100, SA")
plot.zoo(x = TB3MS, main = "3-Month Treasury Rate",
         xlab = "", ylab = "Percent")
plot.zoo(x = NASDAQCOM, main = "Nasdaq Composite Index",
         xlab = "", ylab = "Index, Feb 2, 1971 = 100")
plot.zoo(x = B230RC0Q173SBEA, main = "Population",
         xlab = "", ylab = "Thousands")
```

The above code block uses two main functions:

1. `par()`: This is a base R function used to set or query graphical parameters. Parameters can be set for the duration of a single R session and influence the visual aspect of your plots. Here, it's being used in two ways:
    - `mfrow = c(rows = 3, columns = 2)`: This sets up the plotting area as a matrix of 3 rows and 2 columns. This means up to six plots can be displayed on the same plotting area, arranged in three rows and two columns.
    - `mar = c(b = 2, l = 4, t = 2, r = 1)`: This is setting the margin sizes around the plots. The `mar` parameter takes a numeric vector of length 4, indicating the margin size for the bottom, left, top, and right sides of the plot. The numbers are the line widths for the margins.
2. `plot.zoo()`: This function comes from the `zoo` package in R, which is a package for working with ordered observations like time series data such as `xts` objects. The `plot.zoo()` function is used to create time series plots.
    - `x =`: This parameter is used to pass the time series data you want to plot. For example, `x = GDP` means it's plotting the GDP data.
    - `main =`: This is used to set the title of each plot.
    - `xlab = ""` and `ylab =`: These are used to set the labels for the x and y axes. In this case, the x-axis labels are left blank (`""`), and the y-axis labels are set to various units of measurement relevant to the data being plotted.

In summary, this script is creating a 3×2-matrix of time series plots for six different macroeconomic indicators with specific title and y-axis labels, and specific margin sizes around each plot.

Figure 18 displays the six economic indicators selected to demonstrate the various transformations covered in this chapter. The Gross Domestic Product (`GDP`) quantifies the U.S. total economic output in billions of U.S. dollars. This GDP series undergoes seasonal adjustment (SA), a transformation discussed in Chapter 12.7.3. The Unemployment Rate (`UNRATE`), depicted as a percentage, signifies the fraction of the U.S. workforce actively seeking employment but currently unemployed; this series is also seasonally adjusted (SA). The GDP deflator (`GDPDEF`) measures the aggregate price level in the U.S. and is conveyed as an index number standardized to the average of 2012 Q1 - Q4 equating to 100, which is also seasonally adjusted (SA). See Chapter @ref(index-vs.-absolute-data) for an overview on index measures. The 3-Month Treasury Bill Rate (`TB3MS`), measured as an annual percentage rate, represents the yield on investments in short-term U.S. government debt obligations. The NASDAQ Composite Index (`NASDAQCOM`) captures the performance of over 3,000 listed equities on the NASDAQ stock market, reflected through a market capitalization-weighted index

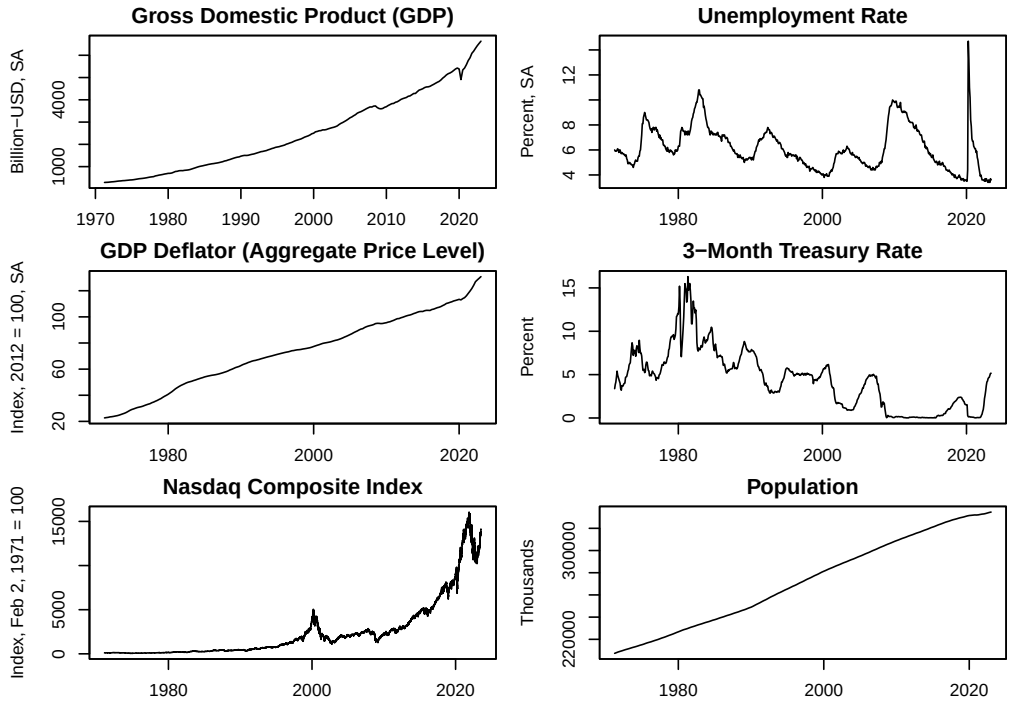


Figure 18: Example Data

value, where the index is normalized to 100 on February 2, 1971. Lastly, Population (`B230RC0Q173SBEA`) lists the number of people living in the United States, reported in thousands of people.

12.2 Growth

12.2.1 Definition

The **growth rate** or **%-change** is the relative increase or decrease in a variable from one time period to the next, typically expressed as a percentage:

$$\%\Delta x_t = \frac{x_t - x_{t-1}}{x_{t-1}} = 100\left(\frac{x_t - x_{t-1}}{x_{t-1}}\right)\%$$

Here, t represents the current period, and $t-1$ denotes the preceding period.

Consider the U.S. GDP growth rate in 2023 Q1 as an example. It's calculated as follows:

$$\begin{aligned}\text{GDP Growth}_{2023Q1} &= \frac{\text{GDP}_{2023Q1} - \text{GDP}_{2022Q4}}{\text{GDP}_{2022Q4}} \\ &= \frac{6632 \text{ Billion USD} - 6534 \text{ Billion USD}}{6534 \text{ Billion USD}} \\ &= 1.50\%\end{aligned}$$

This illustrates that growth rates aren't influenced by the units of measurement: billion USD, a point we'll delve into later in Chapter 12.2.3.

12.2.2 Log Approximation

Growth rates can be approximated using the difference in the natural logarithm of the series:

$$\frac{x_t - x_{t-1}}{x_{t-1}} \approx \ln(x_t) - \ln(x_{t-1}) = 100\Big(\ln(x_t) - \ln(x_{t-1})\Big)\%$$

This approximation is accurate as long as growth rates remain small. However, the approximation loses precision when growth rates exceed 20% or -20%.

To illustrate this, let's plot the growth rate of GDP alongside the difference in its log values in percent:

```
# Compute GDP growth in percent and its log approximation
GDP_growth <- 100 * (GDP - lag.xts(GDP)) / lag.xts(GDP)
GDP_logdiff <- 100 * (log(GDP) - lag.xts(log(GDP)))

# Plot the two series
plot.zoo(x = merge(GDP_growth, GDP_logdiff), plot.type = "single",
         col = c(5, 1), lwd = c(5, 1.5), lty = c(1, 2),
         xlab = "", ylab = "%")
legend("topleft", legend = c("Real GDP Growth", "Log Difference"),
       col = c(5, 1), lwd = c(5, 1.5), lty = c(1, 2),
       horiz = TRUE, bty = 'n')
```

This code block computes two different forms of growth for the GDP time series and plots them together for comparison:

- `GDP_growth <- 100 * (GDP - lag.xts(GDP)) / lag.xts(GDP)`: This line of code computes the percentage growth of the GDP, which is the difference between the current and previous GDP values, divided by the previous GDP value. The `lag.xts(GDP)` function generates a new series where each data point is shifted one time unit into the future, effectively getting the GDP value from the previous time period. This is then multiplied by 100 to convert it into a percentage. The result is stored in the variable `GDP_growth`.
- `GDP_logdiff <- 100 * (log(GDP) - lag.xts(log(GDP)))`: This line of code calculates the log difference of GDP. It first applies a natural logarithm transformation to the GDP (`log(GDP)`) and then computes the difference with the lagged log-transformed GDP (`lag.xts(log(GDP))`). This log difference approximates the growth rate when the changes in GDP are relatively small. The result is then multiplied by 100 to convert it into a percentage. The result is stored in the variable `GDP_logdiff`.
- `plot.zoo()`: This function is used to create a time series plot. The `x = merge(GDP_growth, GDP_logdiff)` argument tells the function to plot the two series together. The `plot.type = "single"` argument makes both series appear on a single plot. The `col = c(5, 1)` argument sets the color of the two lines (5 for magenta and 1 for black), `lwd = c(5, 1.5)` sets the line width, and `lty = c(1, 2)` sets the line type (1 for solid, 2 for dashed). The `xlab` and `ylab` arguments are used to label the x and y-axes, respectively.
- `legend()`: This function adds a legend to the plot. The `"topleft"` argument places the legend at the top left corner of the plot. The `legend = c("Real GDP Growth", "Log Difference")` argument specifies the names of the series. The remaining arguments (`col`, `lwd`, `lty`, `horiz`, and `bty`) set the color, line width, line type, horizontal layout, and box type of the legend, respectively.

Figure 19 generated from this code block shows the real GDP growth and the log difference of GDP over time, providing a visual comparison of these two methods of computing growth. As quarterly GDP growth in the U.S. has remained relatively stable, the log approximation aligns almost perfectly with the actual growth rate.

12.2.3 Relativity

Growth rates **eliminate** the **units of measurement**, making it easier to compare variables of different scales or units. When we examine economic performance using GDP growth, for example, we don't need to standardize each country's GDP into a single currency. This is because growth rates focus on the proportional changes, not the absolute values. Thus, regardless of whether the GDP is measured in US dollars, Euros, or any other currency, the GDP growth rate can be directly compared across countries.

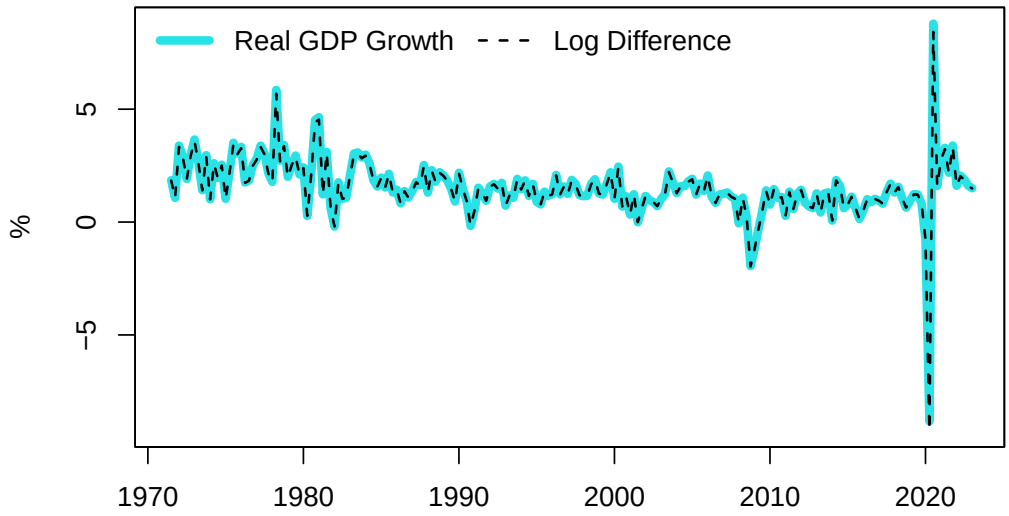


Figure 19: GDP Growth and Log Approximation

Additionally, growth rates provide a **relative measure**, measuring the change with respect to an initial level. This feature makes it an effective tool for comparing economies of different sizes. For instance, a small economy might have a lower absolute GDP than a larger one, but it could be growing at a significantly faster pace. Therefore, using growth rates allows us to observe this relative performance more clearly, irrespective of the initial level of GDP.

12.2.4 Return and Inflation

Note that when the concept of "growth rate" is applied to prices, it's often referred to as "return" or "inflation." The choice of terminology depends on the context. In the financial world, when we discuss the percentage increase in the price of a financial asset such as a stock or a bond over time, we usually refer to the growth rate as a **return**. This usage reflects the perspective of investors who buy assets with the expectation that their value will increase over time, thus yielding a positive return on their investment.

The term **inflation** refers to the overall increase in prices in an economy. When economists calculate the rate at which the general level of prices for goods and services is rising, and subsequently, the purchasing power of currency is falling, they refer to this as the inflation rate.

Despite this terminology, both return and inflation fundamentally represent a form of growth rate, but applied in different contexts.

Now, let's compute and plot the growth rates of the prices from the example data using their log approximation:

```
# Compute growth rates of prices using log approximation
Inflation <- 100 * (log(GDPDEF) - lag.xts(log(GDPDEF)))
Nasdaq_return <- 100 * (log(NASDAQCOM) - lag.xts(log(NASDAQCOM)))

# Put all plots into one
par(mfrow = c(rows = 2, columns = 1),
    mar = c(b = 2, l = 4, t = 2, r = 1))

# Plot growth rates of prices
plot.zoo(x = Inflation, main = "U.S. Inflation",
         xlab = "", ylab = "%")
plot.zoo(x = Nasdaq_return, main = "Nasdaq Stock Market Return",
         xlab = "", ylab = "%")
```

The functions used in the above code chunk have been explained earlier in Chapters 12.2.1 and 12.2.2.

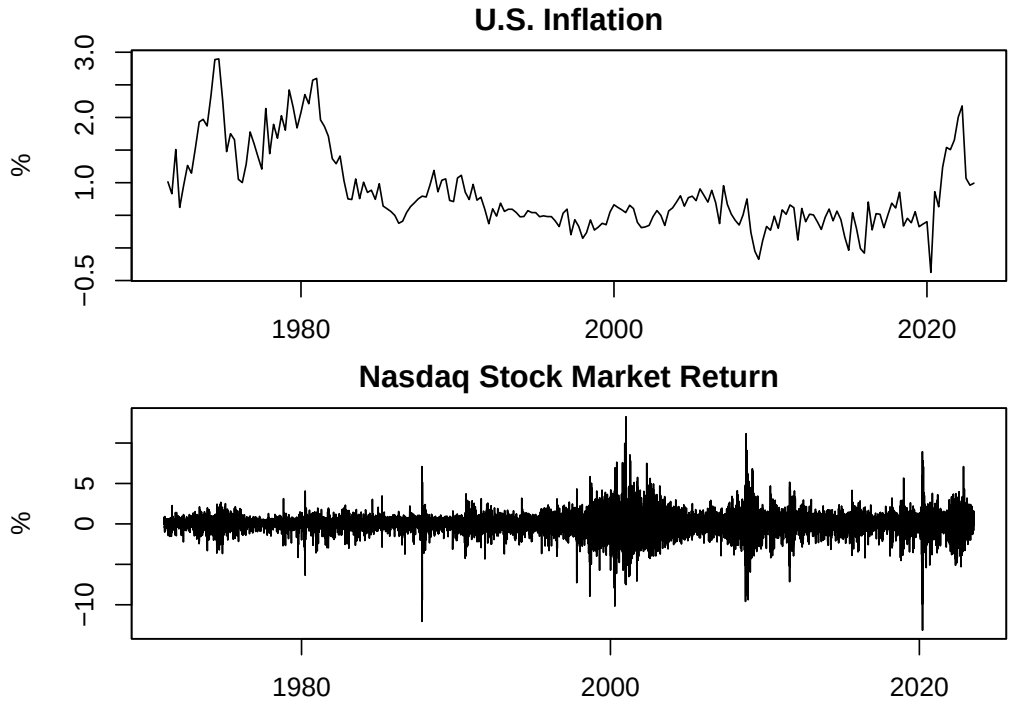


Figure 20: Growth Rate of Prices

Figure 20 is the output of this code block. It displays the growth rate of two price indices: the GDP deflator and the Nasdaq composite index. Since the GDP deflator measures overall price changes in an economy, its growth rate is referred to as inflation. Since the Nasdaq indices represent assets that can be invested in, the growth rates of these prices are termed as returns.

12.2.5 Applicability

Not all data are suitable for transformation into growth rates. Growth rates are most meaningful for ratio scale variables, which measure quantities that have a clear, absolute zero and uniform intervals between numbers (see Chapter 11.3 for an overview on nominal, ordinal, interval, and ratio scale variables). In such cases, growth rates offer insights about the pace of a variable's change over time. This is why we commonly see growth rates for economic output (GDP), prices, population, etc., quantities for which it makes sense to ask, "By what percentage did this quantity change?"

Conversely, some variables, often percentages or rates themselves, are not meaningful to describe in terms of growth rates. Their movements over time are better described in terms of differentiation. This includes variables like unemployment rates, interest rates, or inflation rates.

Specifically, if variables exhibit one or more of the following characteristics, computing growth rates may lead to more confusion than insight:

- They are expressed as a ratio or percentage.
- They don't have a meaningful absolute zero.
- The difference between two values isn't meaningful.
- The ratio of two values is usually not meaningful or not interpreted.

This is a general guideline, and exceptions certainly exist, particularly when delving into specialized domains or specific use cases.

Since the U.S. unemployment rate and the 3-month Treasury bill (T-bill) rate from the example data are

expressed in percentages, their growth rates are not calculated here. However, we will compute their first differences in Chapter 12.3. Interestingly, the 3-month T-bill rate is, in fact, already a growth rate: the values in the time series represent the (annualized) returns of holding a 3-month T-bill until maturity at different dates.

12.3 Differentiation

12.3.1 Definition

The **first difference** signifies the variation in a variable from one time point to the subsequent one:

$$\Delta x_t = x_t - x_{t-1}$$

In this equation, t symbolizes the current period, while $t-1$ signifies the preceding one.

Take the first difference of U.S. GDP in 2023 Q1 for instance. It is computed as follows:

$$\begin{aligned}\Delta\text{GDP}_{2023Q1} &= \text{GDP}_{2023Q1} - \text{GDP}_{2022Q4} \\ &= 6632 \text{ Billion USD} - 6534 \text{ Billion USD} \\ &= 97.95 \text{ Billion USD}\end{aligned}$$

This shows that in contrast with growth rates, differentiation does not remove the units of measurement, with the first difference still presented in billion USD. This could pose some challenges when the underlying unit is a percentage, as outlined in Chapter 12.3.2.

The **second difference** is the variation of the first difference:

$$\Delta^2 x_t = \Delta x_t - \Delta x_{t-1}$$

and the k**th difference** is the difference of the $(k-1)$th difference:

$$\Delta^k x_t = \Delta^{k-1} x_t - \Delta^{k-1} x_{t-1}$$

Differentiation beyond the second difference is not commonly observed in Economics and Finance. Differentiation is frequently employed to eliminate trends in data to concentrate on business cycles.

12.3.2 Percentage Points (pp)

As noted, the units of measurements do not disappear when differentiating. This could create confusion when the underlying unit of measurement is a percentage, for example, when obtaining the first difference of an unemployment rate. This confusion occurs because if the unemployment rate increases by 5%, it implies that the growth is 5%, rather than the difference being 5%. That's why the term **percentage points** was devised, to avert this confusion. Adding "points" to "percentage" clarifies that it is not 5% growth but rather the first difference is 5%.

Formally, a **percent change (%-change)** is a relative measure, demonstrating the change in one value relative to its initial value (see Chapter 12.2):

$$\%\Delta Y_t = 100\left(\frac{Y_t - Y_{t-1}}{Y_{t-1}}\right)\% = 100\left(\frac{5\% - 4\%}{4\%}\right)\% = 25\%$$

while **percentage point change (pp-change)** is an absolute measure, reflecting the difference between two percentage values:

$$\Delta Y_t = Y_t - Y_{t-1} = 5\% - 4\% = 1pp$$

In this context, Y_t is a measure represented in percentages, like the unemployment rate, inflation, GDP growth, stock return, or interest rate. The example illustrates that moving from 4% to 5% is a 25% increase, but only a 1pp increase; therefore, confusing these two units of measurements can cause significant errors.

To illustrate, let's calculate both the %-change and pp-change of the unemployment rate from the example-data, and compare the two measures. Note, however, that as per Chapter 12.2, it is unusual to compute growth rates (%-change) of measures expressed in percent; hence, when writing about %-change of unemployment, the reader will be confused, and likely believe you mixed up %-change with pp-change; thus, I do not recommend actually calculating the %-change of the unemployment rate in practice. This is purely for illustrative purposes:

```
# Compute %-change of unemployment rate
UNRATE_percent_pp <- 100 * (UNRATE - lag.xts(UNRATE)) / UNRATE
names(UNRATE_percent_pp) <- "percent"

# Compute pp-change of unemployment rate
UNRATE_percent_pp$pp <- UNRATE - lag.xts(UNRATE)

# Put all plots into one
par(mfrow = c(rows = 2, columns = 1),
    mar = c(b = 2, l = 4, t = 2, r = 1))

# Plot %-change and pp-change of unemployment rate
plot.zoo(x = UNRATE_percent_pp, plot.type = "single",
         xlab = "", ylab = "% vs. pp", main = "Change in Unemployment Rate",
         ylim = c(-22, 22), col = c(5, 1), lwd = c(1.5, 1))
legend(x = "topleft", legend = c("%-change", "pp-change"),
       col = c(5, 1), lwd = c(1.5, 1), horiz = TRUE)

# Zoom in using the ylim-input
plot.zoo(x = UNRATE_percent_pp,  plot.type = "single",
         xlab = "", ylab = "% vs. pp", main = "(Zoomed In)",
         ylim = c(-2.2, 2.2), col = c(5, 1), lwd = c(1.5, 1))
legend(x = "topleft", legend = c("%-change", "pp-change"),
       col = c(5, 1), lwd = c(1.5, 1), horiz = TRUE)
```

The functions used in the above code chunk have been explained earlier in Chapters 12.2.1 and 12.2.2. What we haven't used before is the `ylim` input, here `ylim = c(-22, 22)`, which defines the limits of the y-axis and allows to zoom in. Note that the `xlim` input does the same for limiting the x-axis.

Figure 21 plots the %-change and pp-change of the unemployment rate. The variance of the %-change is significantly larger than that of the pp-change because the unemployment rates are one or two-digit numbers, so an increase results in a larger relative increase than an absolute increase. Note that if the underlying time series were three or higher digit numbers, it would be reversed: the variance of the pp-change would be larger than that of the %-change.

12.3.3 Basis Points (bp)

Changes in economic and financial data, particularly at a high frequency, often appear in small increments. For instance, monthly changes in interest rates often involve subtle shifts like 0.04pp or -0.24pp. For clarity and ease of reading, these changes are commonly expressed in basis points (bp) rather than percentage points (pp), with 1pp equivalent to 100bp.

To formalize, a **basis point (bp) change** is calculated by multiplying the difference between two percentage values by 100:

$$\Delta Y_t = 100\,(Y_t - Y_{t-1}) = 5\% - 4\% = 1pp = 100bp$$

In this formula, Y_t represents a measure in percentages, like stock returns or interest rates.

Now, let's compute the basis point change for the 3-month Treasury bill rate and visualize it:

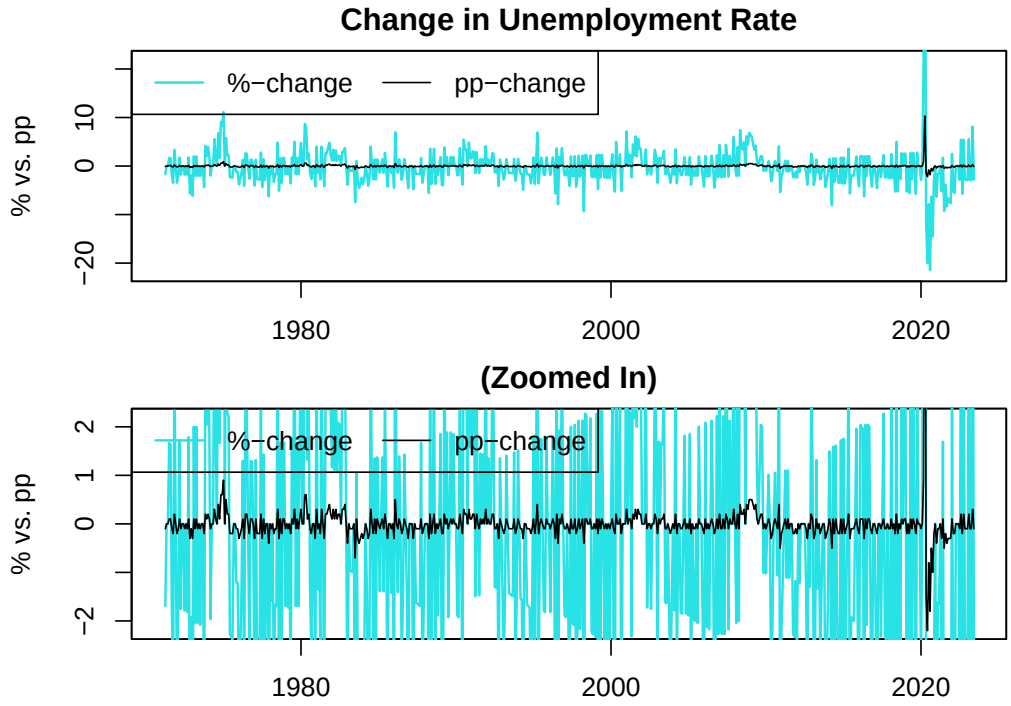


Figure 21: Percent vs. Percentage Point Change of Unemployment Rate

```r
# Calculate basis point (bp) change in 3-month T-bill rate
dTB3MS_bp <- 100 * (TB3MS - lag.xts(TB3MS))

# Plot the time series
plot.zoo(x = dTB3MS_bp["1990/"], plot.type = "single",
         xlab = "", ylab = "Basis Point (bp)", ylim = c(-150, 150),
         main = "Change in 3-Month Treasury Bill Rate")
```

As shown in Figure 22, the bp-change of the 3-month T-bill rate demonstrates that U.S. interest rates typically exhibit gradual increases during economic boom periods and sharp declines during recessions, leading to larger negative changes.

12.4 Natural Logarithm

12.4.1 Definition

The **natural logarithm**, often written as $\ln(x)$, $\log_e(x)$, or $\log(x)$, is the inverse operation to exponentiation with base e, where e is the **natural base** or **Euler's number** and approximately equal to 2.7182818. In other words, if $y = e^x$, then $x = \ln(y)$, which defines the natural logarithm.

This relationship can be expressed as:

$$\ln(e^x) = x$$

and

$$e^{\ln(x)} = x$$

These equations hold true for any positive number x. The natural logarithm function $\ln(x)$ is the power to which e must be raised to get x.

Here's an example of how you can calculate the natural logarithm in R:

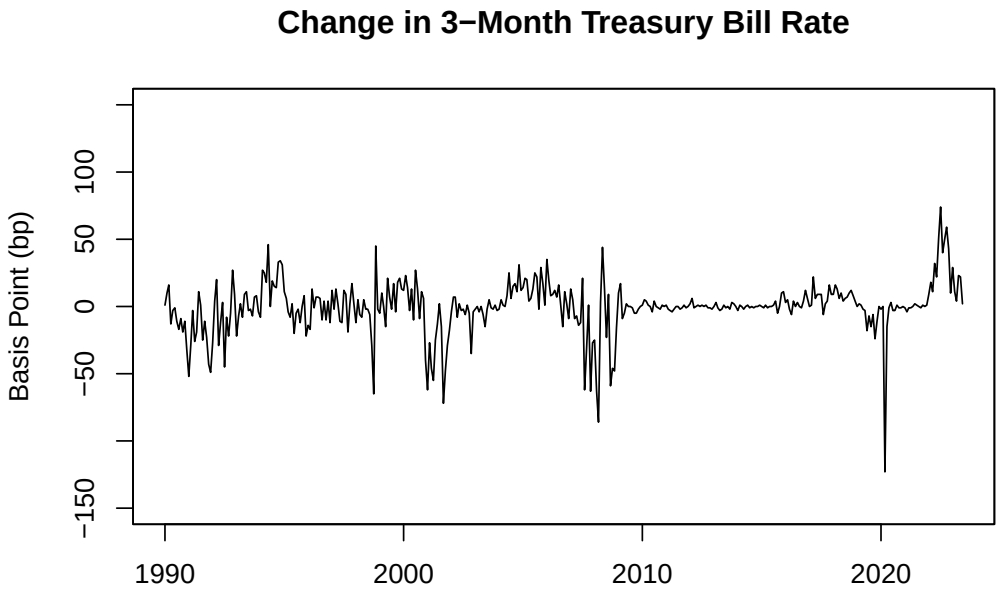


Figure 22: Basis Point Changes in 3-Month Treasury Bill Rate

```
# Calculate the natural logarithm of e (which should be 1)
log(exp(1))
```

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

And here's how you could calculate the natural logarithm of an array of numbers:

```
# Create an array of numbers
numbers <- c(1, 2, 3, 4, 5)

# Calculate the natural logarithm of the array of numbers
log(numbers)
```

```
## [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
```

Now, consider trying to find the natural logarithm of zero, $\ln(0)$. Is there a power to which you can raise e to get zero? The answer is no. Even if we raise e to a very large negative power, the result approaches zero, but never quite gets there. This is why $\ln(0)$ is considered undefined or negative infinity if such number is defined:

```
# Calculate the natural logarithm of zero
log(0)
```

```
## [1] -Inf
```

Next, consider trying to find the natural logarithm of a negative number. The constant e raised to any real number always results in a positive value. Therefore, there's no real number that you can raise e to that would result in a negative number. Because of this, the natural logarithm of a negative number is considered undefined when dealing with real numbers:

```
# Calculate the natural logarithm of a negative number
log(-5)
```

```
## Warning in log(-5): NaNs produced
```

```
## [1] NaN
```

The natural logarithm has many useful properties. For example, one such property is that the natural

logarithm of a product equals the sum of the natural logarithms of each individual number:

$$\ln(ab) = \ln(a) + \ln(b)$$

```
# ln(ab) = ln(a) + ln(b)
log(3 * 4)
log(3) + log(4)
```

```
## [1] 2.484907
## [1] 2.484907
```

Another property is that the natural logarithm of a number raised to a power is the product of the power and the natural logarithm of the number:

$$\ln(a^n) = n \ln(a)$$

```
# ln(a^n) = n * ln(a)
log(3^4)
4 * log(3)
```

```
## [1] 4.394449
## [1] 4.394449
```

These unique properties of natural logarithm make it particularly valuable in simplifying mathematical expressions and solving exponential and logarithmic equations.

12.4.2 Motivation

Logarithmic measures aid in making the visualization and interpretation of exponentially growing data simpler. For instance, stock prices, which often exhibit exponential growth, can make long-term historical data appear flat when graphed. Using a logarithmic transformation can convert this exponential growth into linear growth, thus making patterns and shifts over time more evident. Logarithmic measures are also easy to interpret as changes in the log approximate the growth rate (as discussed in Chapter 12.2.2).

To illustrate this, let's graph the Nasdaq composite index from the example-data and its natural log:

```
# Calculate natural logarithm of Nasdaq composite index
Nasdaq_log <- log(NASDAQCOM)

# Set up subplots
par(mfrow = c(rows = 2, columns = 1),
    mar = c(b = 2, l = 4, t = 2, r = 1))

# Plot original time series
plot.zoo(x = NASDAQCOM,
         xlab = "", ylab = "Index, Feb 2, 1971 = 100",
         main = "Nasdaq Composite Index")

# Plot natural log of time series
plot.zoo(x = Nasdaq_log,
         xlab = "", ylab = "Log",
         main = "Log of Nasdaq Composite Index")
```

Figure 23 shows that the Nasdaq stock market index appears stagnant in the early sample. However, the logarithmic transformation uncovers significant growth and contraction periods throughout the sample, highlighting the changes in the earlier years. This is because stock prices often demonstrate exponential growth, with value increasing proportional to the current value. Therefore, even substantial %-changes early

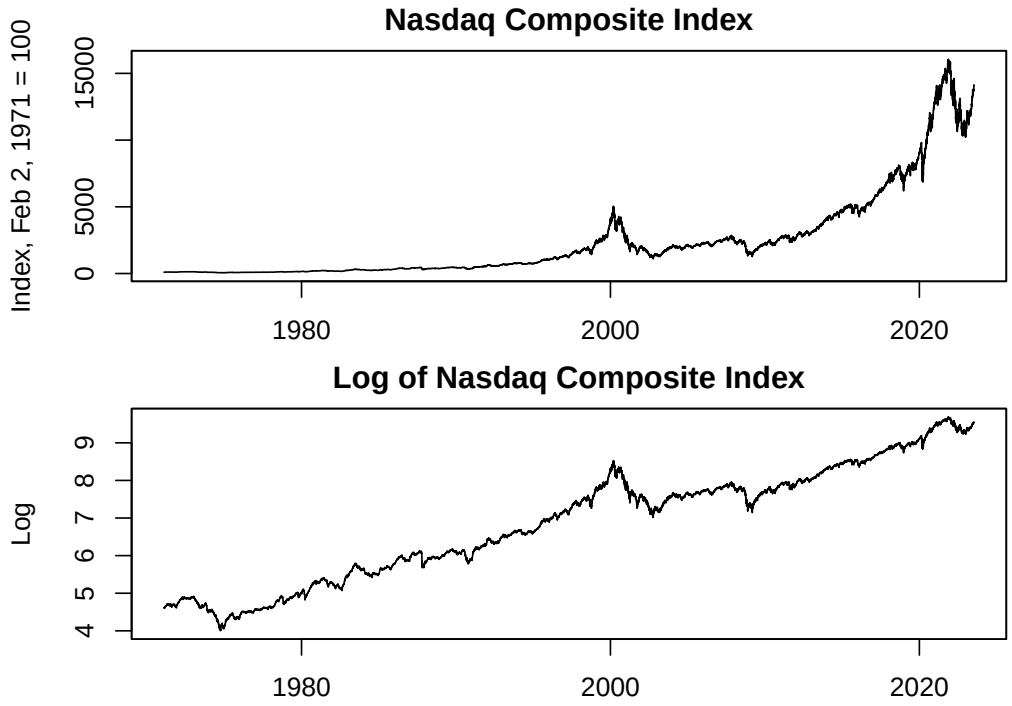


Figure 23: Log Transformation of Nasdaq Composite Index

in a stock's history may appear small due to a lower starting price. In contrast, smaller %-changes later on appear large because they started from a higher price. This can distort how we view a stock's past ups and downs.

The second panel of Figure 23 presents the natural logarithm of the Nasdaq stock market index. This transformation helps visualize the exponential growth in stock prices as a linear increase, with the slope of the line representing the growth rate. Crucially, changes in the log of the price correspond directly to percentage changes in the price. For instance, a 0.01 increase in the log price equates to a 1% rise in the original price (see Chapter 12.2.2). This characteristic enables a direct comparison of price changes over time, aiding in the interpretation of these shifts in terms of relative growth or contraction.

You might wonder why we plot the log of a series, which interprets change as growth rate, instead of directly plotting the growth rate. This is because the natural logarithm of a variable often offers more insight into long-term trends than the growth rate itself. To understand this better, use the fact that changes in the log approximate the growth rate. As a result, the log series embodies the **cumulated growth rates** – the sum of all growth rates from the beginning of the dataset to the present. When the growth rate fluctuates extensively, it can be tough to discern whether positive or negative growth rates predominate over time. However, when you plot the logarithm, these growth rates are cumulative, making it clear whether a variable is growing or not.

Let's compare the growth rate and the natural logarithm of the Nasdaq composite index from the example-data to demonstrate this:

```
# Calculate growth rate of Nasdaq composite index
Nasdaq_return <- 100 * (log(NASDAQCOM) - lag.xts(log(NASDAQCOM)))

# Set up subplots
par(mfrow = c(rows = 2, columns = 1),
    mar = c(b = 2, l = 4, t = 2, r = 1))

# Plot growth rate
```

```r
plot.zoo(x = Nasdaq_return,
         xlab = "", ylab = "%",
         main = "Nasdaq Stock Market Return (Growth Rate)")

# Plot natural log
plot.zoo(x = Nasdaq_log,
         xlab = "", ylab = "Log",
         main = "Log of Nasdaq Composite Index (Cumulated Growth Rate)")
```

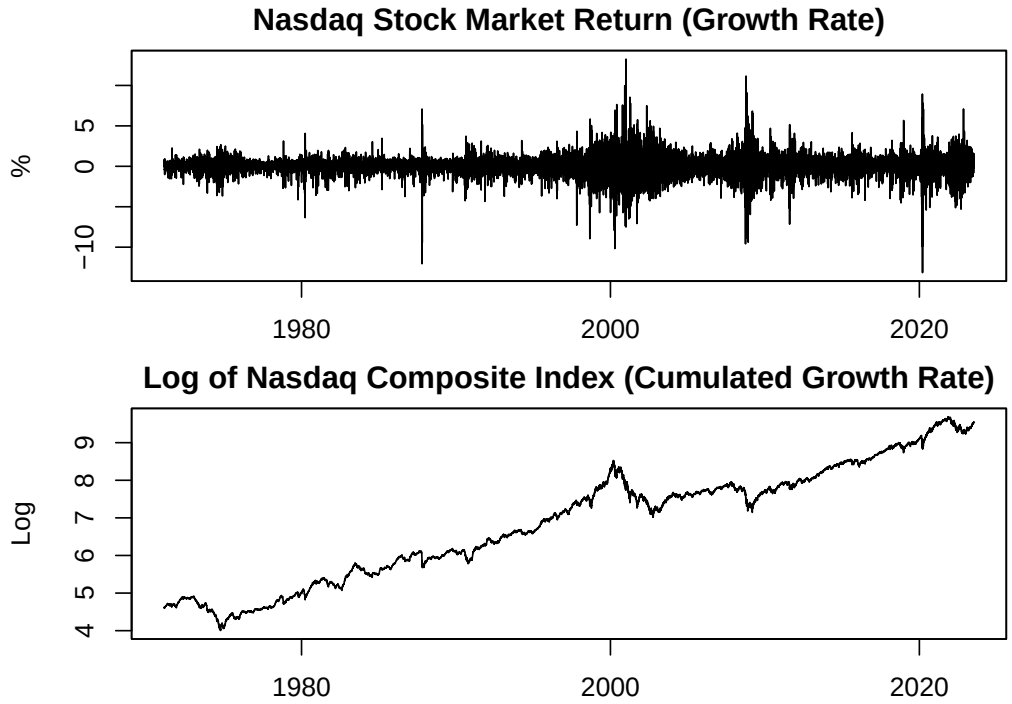


Figure 24: Growth vs. Log of Nasdaq Composite Index

The first panel of figure 24 shows the highly volatile stock market returns (growth rate), making it difficult to assess overall performance. The second panel, on the other hand, depicts the log transformation of the Nasdaq stock market index. This transformation accumulates the stock market returns from the first panel from the beginning to the current period on the x-axis. It clearly shows whether stock prices grow or contract over time, providing a more reliable perspective on the market's performance.

12.4.3 Applicability

Not all data are suitable for transformation into natural logarithm. Essentially, if the concept of a growth rate is nonsensical or irrelevant for the data at hand, then it would also be inappropriate to compute its natural logarithm. This is because the natural logarithm essentially represents cumulative growth rates, and therefore presupposes that a concept of 'growth' is meaningful for the data in question. Specific cases where the application of the growth rate may or may not be appropriate are discussed in Chapter 12.2.5.

12.5 Ratio

12.5.1 Definition

A **ratio** is a mathematical expression that represents the relationship between two quantities or variables, represented as a fraction. It's often used to provide insights into relative magnitudes or proportions between the two variables.

Ratio measures can be calculated as follows:

$$\text{Ratio Measure} = \frac{\text{Variable 1}}{\text{Variable 2}}$$

Ratio measures can provide useful insights into how the two variables relate to each other. For instance, a company's debt-to-equity ratio compares its total debt to its total equity, offering a sense of the company's financial leverage. A high debt-to-equity ratio might indicate a risky financial situation, whereas a lower ratio may suggest a more stable financial position.

In the financial sector, another common ratio measure is the price-to-earnings (P/E) ratio. This ratio compares a company's stock price to its earnings per share, giving investors a sense of how much they're paying for each dollar of earnings.

It's important to clarify that the term "ratio" defined in this context is not to be confused with "ratio scale" variables defined in Chapter 11.3.4. The "ratio" in the current context refers to the mathematical comparison of two quantities, "a to b". On the other hand, a "ratio scale" variable refers to a type of variable where the difference between any two values is meaningful and it has a true zero point. This means that zero indicates the absence of the quantity being measured, and ratios between numbers on the scale are meaningful. Therefore, despite sharing a common term, these two concepts have fundamental differences. However, calculating a ratio measure only makes sense if the underlying variables are indeed ratio scale variables.

12.5.2 Per Capita

Per capita measures are a specific type of ratio measure that calculate the average value of a particular variable per person within a given population. These measures are typically obtained by dividing a macroeconomic variable, such as Gross Domestic Product (GDP) or income, by the population size of a region:

$$\text{Per Capita Measure} = \frac{\text{Variable}}{\text{Population Size}}$$

Per capita measures play a significant role in economics and finance. One prominent example is GDP per capita, which calculates the average economic output per person in a country. This measure enables meaningful comparisons of the average standard of living and economic well-being across nations, regardless of population size. It provides insights into the relative well-being of individuals within different countries.

In the field of finance, per capita measures are also valuable. For instance, credit card debt per capita offers insights into the average debt burden carried by individuals in a specific region. By dividing the total credit card debt by the population, analysts can assess the average level of indebtedness and make comparisons across different areas. This measure helps in understanding the credit behavior and financial health of individuals within various regions.

To calculate GDP per capita in the U.S., we divide the GDP series (`GDP`) of the example-data by the population (`B230RC0Q173SBEA`) of the country:

```
# Compute GDP per capita
GDP_per_capita <- ((10^9) * GDP) / (1000 * B230RC0Q173SBEA)
```

For example, U.S. GDP per capita in 2023 Q1 is calculated as follows:

$$\begin{aligned}
\text{GDP Per Capita}_{2023Q1} &= \frac{\text{GDP}_{2023Q1}}{\text{Population}_{2023Q1}} \\
&= \frac{6,632 \text{ Billion USD}}{334,641 \text{ Thousands}} \\
&= \frac{6,632,443,500,000 \text{ USD}}{334,641,000} \\
&= 19,819.58 \text{ USD}
\end{aligned}$$

This division yields the average economic output per person in USD. Therefore, in the U.S., the average income per person was 19,819.58 USD in 2023 Q1, taking into account all sources of income, including wages, dividends, and profits.

Next, let's visualize GDP per capita:

```
# Plot GDP per capita
plot.zoo(x = GDP_per_capita,
         xlab = "Date", ylab = "USD",
         main = "GDP Per Capita in the United States")
```

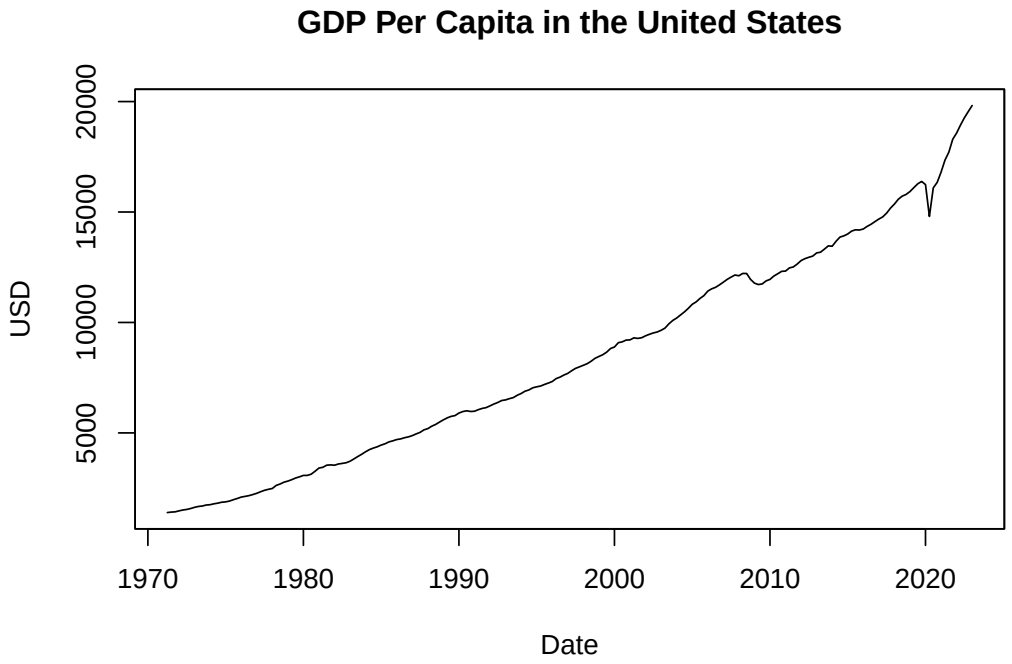


Figure 25: GDP Per Capita in the United States

Figure 25 illustrates GDP per capita in the United States, representing the average income per person in USD over time. The plot shows a consistent increase in average income, indicating economic growth. However, it's important to note that prices have also steadily increased during this period. Therefore, to accurately evaluate changes in the standard of living, it becomes necessary to consider these price changes. In the next chapter, we will delve into the concept of real measures, which account for such changes and provide a more accurate representation of living standards.

12.5.3 Real vs. Nominal

Real measures are ratio measures that adjust for changes in prices by dividing the variable by the price index. In contrast, **nominal measures** are the counterparts of real measures, remaining unadjusted for inflation:

$$\text{Real Measure} = 100 \left(\frac{\text{Nominal Measure}}{\text{Price Index}} \right)$$

In this equation, the ratio is multiplied by 100 due to the normalization of price indices at 100 during a specific period. For example, if the price index is normalized at 100 in 2012, then the real measure is interpreted as the nominal measure in terms of 2012 prices.

Real variables measure the physical quantity of goods and services by accounting for changes in the price level. They provide a more accurate representation of economic trends. Examples include real GDP, real income, and real wages. These measures eliminate the impact of price changes, enabling comparisons of economic performance over time or across regions.

On the other hand, nominal measures, which are not adjusted for price changes, represent the monetary value of goods and services at current prices. Hence, unlike real values that adjust for inflation, nominal values do not.

It's essential to distinguish between the "nominal measure" discussed here and the "nominal scale" variables outlined in Chapter 11.3.1. In this context, "nominal measure" refers to values not adjusted for price changes, whereas "nominal scale" variables are a type of categorical data where different categories do not indicate any order or hierarchy.

To calculate real GDP in the U.S., the GDP series (`GDP`) of the example-data is divided by a price index - specifically, the U.S. GDP deflator index (`GDPDEF`):

```
# Compute real GDP
RGDP <- 100 * GDP / GDPDEF
```

The resulting data is returned as an `xts` object. You can compare the real GDP measure `RGDP` with the nominal measure `GDP`, which is the GDP measure unadjusted for price changes:

```
# Merge nominal and real GDP
GDP_nominal_real <- merge(GDP, RGDP)

# Print merged data
tail(GDP_nominal_real)
```

```
##               GDP    GDP.1
## 2021 Q4 6087.280 5001.545
## 2022 Q1 6185.120 4981.011
## 2022 Q2 6312.119 4973.815
## 2022 Q3 6430.985 5013.671
## 2022 Q4 6534.498 5045.633
## 2023 Q1 6632.444 5070.675
```

Visualize the two GDP series using the `plot.zoo()` function:

```
# Plot nominal vs. real GDP
plot.zoo(x = GDP_nominal_real, plot.type = "single",
         col = c(1, 2), lwd = 2, lty = c(1, 2),
         main = "Nominal vs. Real GDP in the United States",
         xlab = "Date", ylab = "Billions of USD")
legend(x = "topleft", legend = c("Nominal GDP", "Real GDP"),
       col = c(1, 2), lwd = 2, lty = c(1, 2))
```

Figure 26 displayes U.S. GDP for each quarter since 1947, contrasting the nominal and real measures. In 2012, nominal and real GDP align because the real GDP is defined in terms of 2012 prices, and the nominal GDP (measured in current prices) is also based on 2012 prices for that year. The fact that nominal GDP has grown more than real GDP reflects the influence of inflation, which causes nominal GDP to inflate while real GDP provides a more accurate measure of economic growth.

In summary, **nominal GDP** quantifies the total value of goods and services produced within an economy, calculated using the market prices at the time of measurement. As such, if prices increase due to inflation, nominal GDP might rise even without an actual increase in goods and services produced. Conversely, **real GDP** adjusts for inflation and represents economic output in terms of "constant prices" from a chosen base year - like calculating today's economic output as if prices had remained at their 2012 levels. This inflation-adjusted measure enables more accurate comparisons of economic growth over different periods.

12.5.4 Growth of Ratios

The **growth rate of a ratio** can be calculated based on the growth rate of the variables that form it.

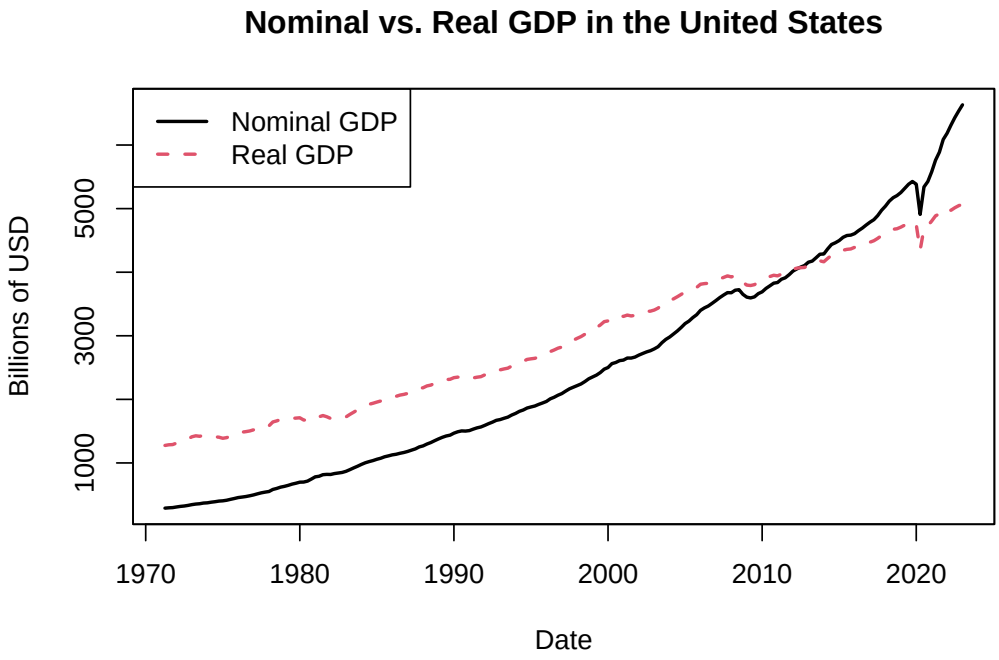


Figure 26: Nominal vs. Real GDP in the United States

Define $x_t = y_t/p_t$ as the ratio of interest (e.g., real GDP), y_t as the numerator (e.g., nominal GDP), and p_t as the denominator (e.g., the price level). The growth rate of the ratio can then be formulated as follows:

$$\%\Delta x_t = \frac{x_t - x_{t-1}}{x_{t-1}} = \frac{\frac{y_t}{p_t} - \frac{y_{t-1}}{p_{t-1}}}{\frac{y_{t-1}}{p_{t-1}}} = \frac{p_{t-1}}{p_t}\frac{y_t}{y_{t-1}} - 1 = \frac{1 + \%\Delta y_t}{1 + \%\Delta p_t} - 1$$

This formula links the growth rate of the ratio measure $\%\Delta x_t$ with the growth rates of the constituent variables $\%\Delta y_t$ and $\%\Delta p_t$.

Consider the case of real versus nominal growth, yielding the following equation:

$$1 + \text{Real Growth}_t = \frac{1 + \text{Nominal Growth}_t}{1 + \text{Inflation}_t}$$

Another approach to approximate the growth rate of a ratio is through the use of logarithms:

$$\begin{aligned}
\%\Delta x_t &\approx \ln(x_t) - \ln(x_{t-1}) \\
&= \ln\left(\frac{y_t}{p_t}\right) - \ln\left(\frac{y_{t-1}}{p_{t-1}}\right) \\
&= \Big(\ln(y_t) - \ln(y_{t-1})\Big) - \Big(\ln(p_t) - \ln(p_{t-1})\Big) \\
&= \%\Delta y_t - \%\Delta p_t
\end{aligned}$$

The growth rate of the ratio measure is approximately the difference between the growth rates of the two underlying variables. In terms of real growth, this yields the following approximation:

$$\text{Real Growth}_t \approx \text{Nominal Growth}_t - \text{Inflation}_t$$

To illustrate, consider the interest rate as a nominal growth variable. The interest rate is a growth rate when it serves as a discount rate or yield to maturity, as illustrated by the 3-month Treasury bill data from the example-data section, which captures the total return on a security over a year. However, if the interest rate is understood as a coupon rate that doesn't account for capital gain, then the interest rate is not a growth rate. The above formula can then be used to calculate the (ex-post) **real interest rate**, representing the return on the security in real terms:

$$r_t \approx i_t - \pi_t$$

Where r_t is the real interest rate, i_t is the (nominal) interest rate, and π_t is the inflation rate. The real interest rate is essentially the quantity of goods and services one could purchase in a year by holding this security for that duration.

Applying this to data:

```
# Compute ex post real interest rates
Inflation_annual_rate <- 4 * Inflation
EXPOSTREAL <- TB3MS - Inflation_annual_rate

# Plot ex post real interest rate
plot.zoo(x = na.omit(merge(TB3MS, Inflation_annual_rate, EXPOSTREAL)),
         plot.type = "single", ylim = c(-16, 16),
         lwd = c(4, 2, 1.5), lty = c(1, 3, 1), col = c(5, 2, 1),
         ylab = "", xlab = "",
         main = "Real 3-Month Treasury Bill Rate")
legend(x = "bottomleft",
       legend = c("Nominal Rate", "Inflation", "Real Rate"),
       lwd = c(4, 2, 1.5), lty = c(1, 3, 1), col = c(5, 2, 1),
       bty = 'n', horiz = TRUE)
abline(h = 0, lty = 2)
```

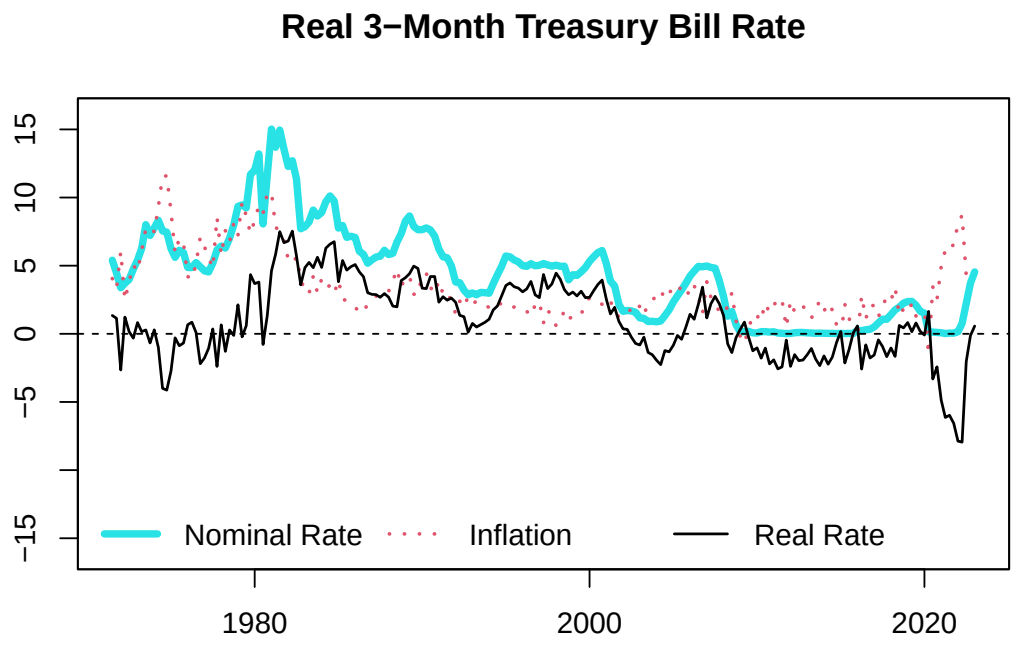


Figure 27: Real 3-Month Treasury Bill Rate

Figure **??** shows that real and nominal rates typically move in tandem, but not always. For example, in the 1970s, even though the nominal rates in the U.S. were high, the real rates were surprisingly low, and sometimes even negative. If you just looked at the high nominal rates, you might think it was tough to get

credit because it was costly to borrow. However, the negative real rates tell us that this was not actually the case.

12.6 Gap

A **gap** is a term that refers to the difference between two variables. You can calculate a gap measure as follows:

$$\text{Gap Measure} = \text{Variable 1} - \text{Variable 2}$$

This gap measure might be referred to by different names, depending on the context. For example, in Finance, the term **spread** is used to indicate the gap between two interest rates. In the field of Economics, the term **disparity** is often used to describe the gap between the highest and lowest income levels within a population or group, while the gap between revenues and expenditures is known as **net** revenue.

12.7 Filtering

Filtering techniques play a vital role in the fields of economics and finance by helping to extract meaningful insights from noisy data. They are primarily used to isolate certain components of time-series data such as trends, cycles, or seasonal patterns. This chapter will introduce and delve into the concepts of filtering specific frequencies, detrending, and seasonal adjustment.

12.7.1 Frequency Filtering

In some instances, analysts are interested in isolating components of a time series that oscillate at specific frequencies. For example, business cycle analysis often focuses on fluctuations that recur every 2 to 8 years. To isolate these components, one can apply filters that attenuate (reduce the amplitude of) frequencies outside this range.

A well-known filter in economics is the band-pass filter, which only allows frequencies within a certain band to pass through. The Baxter-King filter, for instance, is a widely used band-pass filter in economics. Another popular approach is the wavelet transform, which can provide a time-varying view of different frequencies.

12.7.2 Detrending

Detrending is another important tool for analyzing time series data. In many economic series, there is often a long-term trend or direction in which the series is headed, such as the general upward trend of GDP or stock market indices. Detrending is the process of removing this underlying trend to study the cyclical and irregular components of the series.

Several methods exist for detrending data, from simple approaches such as subtracting the linear trend estimated by least squares, to more sophisticated methods like the Hodrick-Prescott (HP) filter, which estimates and removes a smooth trend component.

12.7.3 Seasonal Adjustment

In many economic time series, patterns tend to repeat at regular intervals. This repetition is referred to as seasonality. Examples include increased retail sales during the holiday season, or fluctuations in employment rates due to seasonal industries. Seasonal adjustment is the process of removing these recurring patterns to better understand the underlying trend and cyclical behavior of the series.

Methods for seasonal adjustment include moving-average methods, like the Census Bureau's X-13ARIMA-SEATS, and model-based methods such as the popular STL (Seasonal and Trend decomposition using Loess) procedure. By implementing these techniques, researchers and policymakers can make more accurate comparisons over time and across different series, free from the distortion of seasonal effects.

For instance, consider **seasonally adjusted (SA) GDP**. It is a modified GDP measure that eliminates seasonal variation effects like holidays, weather patterns, or specific events. These adjustments allow economists

to focus on analyzing underlying economic trends and business cycles. For example, in a seasonally adjusted GDP series, a significant increase in output from Q3 to Q4 can be attributed to actual economic growth rather than the higher consumption typically associated with holiday seasons.

Seasonally adjusted GDP data can be obtained from various sources, including government statistical agencies or international organizations. In the United States, the Bureau of Economic Analysis (BEA) provides seasonally adjusted GDP data, which can be accessed through the BEA website or economic data platforms like FRED. In R, there are packages available that perform such seasonal adjustment. The `stats` package offers functions such as `decompose()` and `stl()` for seasonal decomposition and filtering, while the `seasonal` package provides tools for estimating and removing seasonal components from time series data. These packages enable users to perform seasonal adjustment by analyzing the time series patterns of the same series or incorporating information from related series, such as export or import data. However, it is recommended to first check for pre-existing seasonally adjusted data available from the BEA or other reliable sources before attempting to create your own adjustments, as they may have access to more advanced tools and methodologies.

In the example data, the GDP series (`GDP`) is already seasonally adjusted, the common default option, as most economists focus more on business cycles than seasonal cycles. To access non-seasonally adjusted U.S. GDP data, visit FRED and search "U.S. GDP". The first suggestion would be "Gross Domestic Product" subtitled "Billions of Dollars, Quarterly, Seasonally Adjusted Annual Rate". Although this is the correct variable, we require the original, not seasonally adjusted (NSA) GDP series. To retrieve this, click on "6 other formats", and select "Quarterly, Millions of Dollars, Not Seasonally Adjusted". The resulting graph will be titled "Gross Domestic Product (NA000334Q)". To download the data, use the `getSymbols()` function with `Symbols` parameter as `"NA000334Q"` and the `src` (source) parameter as `"FRED"`:

```
# Download GDP: Quarterly, Millions of Dollars, Not Seasonally Adjusted
getSymbols("NA000334Q", src = "FRED")
```

```
## [1] "NA000334Q"
```

After rescaling the non-seasonally adjusted GDP to billions, we can compare the seasonally adjusted (SA) GDP measure `GDP` with the non-seasonally adjusted (NSA) measure `NA000334Q`:

```
# Rescale the not seasonally adjusted GDP to billions
GDP_nsa <- NA000334Q / 1000

# Merge seasonally adjusted and not seasonally adjusted GDP
GDP_sa_nsa <- merge(GDP, GDP_nsa)

# Print merged data
tail(GDP_sa_nsa)
```

```
##              GDP NA000334Q
## 2021 Q4 6087.280  6203.369
## 2022 Q1 6185.120  6010.733
## 2022 Q2 6312.119  6352.982
## 2022 Q3 6430.985  6439.154
## 2022 Q4 6534.498  6655.020
## 2023 Q1 6632.444  6481.468
```

Visualization of both GDP series:

```
# Plot U.S. seasonally adjusted vs. not seasonally adjusted GDP
plot.zoo(x = GDP_sa_nsa["1970/"], plot.type = "single",
         col = c(5, 1), lwd = c(4, 1),
         main = "SA vs. NSA U.S. GDP",
         xlab = "Date", ylab = "Billions of USD")
legend(x = "topleft", legend = c("Seasonally Adjusted (SA)", "Not Seasonally Adjusted (NSA)"),
```

```
       col = c(5, 1), lwd = c(4, 1))
```

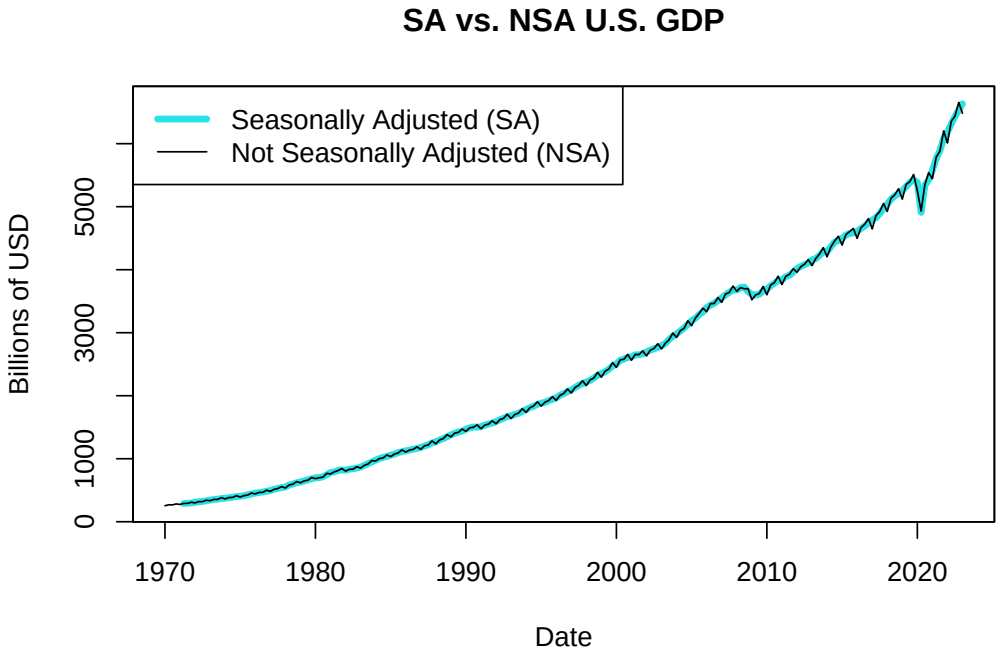


Figure 28: SA vs. NSA U.S. GDP

Figure 28 displays U.S. GDP for each quarter since 1970, comparing the seasonally adjusted with the not seasonally adjusted measure. The seasonally adjusted GDP series removes the effects of seasonal variations, allowing for a clearer analysis of underlying trends and business cycles. On the other hand, the not seasonally adjusted GDP series reflects the raw, unadjusted data and includes the impact of seasonal variations. This series provides a more detailed view of the quarterly fluctuations in economic activity, which can be influenced by factors like holiday spending or seasonal industries.

13 Data Aggregation

The content for this chapter is still under development and will be available soon. This chapter will not be relevant for ECO 4300-D02 students enrolled in the 2023 Summer II term.

14 Data Source

This chapter provides an overview of the primary U.S. institutions that either collect economic data (data suppliers) or compile and disseminate data from various sources (data distributors).

Data suppliers:

- Bureau of Economic Analysis (BEA)
- U.S. Census Bureau
- Bureau of Labor Statistics (BLS)
- Federal Reserve Board of Governors
- U.S. Department of the Treasury
- Financial Market Exchanges

Data distributors:

- Federal Reserve Economic Data (FRED)

- U.S. Department of Agriculture Economic Research Service (ERS)
- Bloomberg
- Yahoo Finance

14.1 Bureau of Economic Analysis (BEA)

The Bureau of Economic Analysis (BEA) is part of the U.S. Department of Commerce and is primarily responsible for providing comprehensive and extensive economic statistics. Key reports include:

- Gross Domestic Product (GDP)
- Personal Income and Outlays
- Corporate Profits
- U.S. International Trade in Goods and Services

14.2 U.S. Census Bureau

The U.S. Census Bureau collects demographic, social, and economic data. This information, which includes the decennial census, informs everything from legislative representation to public policy decisions. Key data includes:

- Population and Housing Census
- American Community Survey
- Economic Census
- Annual Surveys of Manufacturers

14.3 Bureau of Labor Statistics (BLS)

The Bureau of Labor Statistics is the principal federal agency responsible for measuring labor market activity, working conditions, and price changes in the economy. Key data includes:

- Employment Situation
- Consumer Price Index
- Producer Price Index
- Real Earnings

14.4 Federal Reserve Board of Governors

The Federal Reserve Board of Governors provides data on the functions, operations, and structure of the Federal Reserve System. Key data includes:

- Interest Rates
- Monetary Aggregates
- Exchange Rates
- Industrial Production and Capacity Utilization

14.5 U.S. Department of the Treasury

The U.S. Department of the Treasury collects data related to the U.S. government's finances and debt. Key data includes:

- Daily Treasury Yield Curve Rates
- Treasury Auctions
- U.S. International Reserve Position
- The Monthly Treasury Statement (receipts and outlays of the federal government)

14.6 Financial Market Exchanges

Exchanges such as the New York Stock Exchange (NYSE), NASDAQ, and Chicago Mercantile Exchange (CME) provide extensive data on financial securities. Key data includes:

- Equity prices
- Trading volumes
- Derivatives data (futures and options)
- Index values

14.7 Federal Reserve Economic Data (FRED)

FRED, maintained by the Federal Reserve Bank of St. Louis, offers a wealth of economic data from multiple sources in an easily accessible format. It includes but is not limited to:

- Interest Rates (sourced from Federal Reserve)
- Unemployment Rate (sourced from BLS)
- Consumer Price Index (CPI) (sourced from BLS)
- Money Stock Measures (sourced from Federal Reserve)

14.8 U.S. Department of Agriculture Economic Research Service (ERS)

The ERS is a primary source of economic information and research in the field of agriculture. Key data includes:

- Agricultural Outlook
- Crop and Livestock Production
- Food Prices, Expenditures, and Establishments
- International Agricultural Trade

14.9 Bloomberg

Bloomberg provides financial software tools such as an analytics and equity trading platform, data services, and news to financial companies and organizations. Key data includes:

- Stock Market Data
- Economic Indicators
- Commodities Prices
- Currency Exchange Rates

Bloomberg data requires a Bloomberg license and installation of Bloomberg software.

14.10 Yahoo Finance

Yahoo Finance is a media property that provides financial news, data and commentary including stock quotes, press releases, financial reports, and original content. Key data includes:

- Stock Quotes
- Market Trends
- Financial News
- Economic Calendar

Unlike Bloomberg data, data from Yahoo Finance can be accessed for free.

15 Temporal Patterns

In Economics, time-series data often exhibit three key features: a trend, a business cycle, and a seasonal cycle. Understanding these features is crucial to analyzing and interpreting economic data.

15.1 Trends

A trend is the overall direction in which a series of data is moving over time. For example, an upward trend in GDP might indicate a growing economy, while a downward trend could signal a contraction.

Trends can be identified visually by plotting the data over time and observing the general pattern. More sophisticated techniques include using a moving average or applying statistical methods like regression analysis to estimate the trend line.

15.2 Business Cycles

A business cycle refers to fluctuations in economic activity around the trend. These cycles typically consist of periods of expansion (growth in real output) and recession (decline in real output). The peak of the cycle represents the end of an expansion and the start of a recession, while the trough marks the end of a recession and the beginning of an expansion.

Recession periods are a crucial part of economic history. They are often associated with periods of high unemployment, decreased consumer spending, and declines in the stock market. Understanding when recessions have occurred can provide insight into the causes and consequences of economic downturns.

Recession shades are often used to highlight periods of economic downturn. These are typically shown as gray bars on a time-series graph. The start of a recession shade marks the peak of the business cycle, and its end marks the trough. The National Bureau of Economic Research (NBER) is a primary source of information on business cycle dates in the U.S.

15.3 Seasonal Cycles

Seasonal cycles are predictable patterns that recur each year and are driven by the seasons. For instance, retail sales may spike during the holiday season, or construction activity might decrease in winter due to weather conditions.

To analyze the underlying trend and business cycles, economists often use seasonally adjusted data, which remove these seasonal effects.

16 Regional Patterns

The economy does not operate uniformly across regions. Variations in economic performance can be observed at different geographical levels: from county to state, and state to country. This chapter delves into understanding these regional patterns and the factors influencing them.

16.1 County-Level Patterns

At the county level, economic performance can vary significantly due to factors such as the availability of natural resources, the local industrial mix, population demographics, and the quality of local institutions. For instance, counties with abundant natural resources may experience economic booms when commodity prices are high. On the other hand, counties with a high concentration of manufacturing jobs might suffer during a downturn in the industrial sector.

County-level economic data can help local policymakers understand the economic conditions of their jurisdictions and develop strategies to promote economic growth and resilience. Examples of relevant metrics at this level include employment rates, median income, poverty rates, and business activity.

16.2 State-Level Economic Patterns

State-level economic patterns can be shaped by a variety of factors including state tax policies, quality of infrastructure, education levels, and the presence of large corporations. For instance, states with favorable

business climates may attract more companies, leading to higher employment rates and stronger economic growth.

Metrics commonly used to analyze state-level economic performance include GDP per capita, unemployment rate, and measures of income inequality. Comparing these metrics across states can highlight disparities and potential areas for policy intervention.

16.3 Country-Level Economic Patterns

At the country level, economic performance is influenced by factors such as national policies, international trade relations, technological advancement, and geopolitical stability. Differences in these factors can lead to wide disparities in economic outcomes across countries.

Key metrics for analyzing country-level economic performance include GDP growth rate, inflation rate, balance of trade, and Human Development Index (HDI). Additionally, tools like the GINI index are used to compare income inequality across countries.

International organizations such as the World Bank, the International Monetary Fund (IMF), and the Organisation for Economic Co-operation and Development (OECD) regularly publish country-level economic data. These resources can be valuable for comparing economic performance across countries and understanding the global economic landscape.

17 Causal Relationships

The content for this chapter is still under development and will be available soon. This chapter will not be relevant for ECO 4300-D02 students enrolled in the 2023 Summer II term.

18 Data Report on Traditional Economic Indicators

The objective of this assignment is to develop a data report on economic indicators. The report should be prepared using R Markdown and must be a minimum of four pages long, containing both graphs and text.

Before you embark on your report, ensure you have thoroughly completed the preparations outlined in Chapter 18.1. Your report should comply with the guidelines articulated in Chapter 18.2 and Chapter 18.3. Upon completion, please submit your data report via the course website (Blackboard).

18.1 Preparations

To lay the groundwork for this assignment, work through the following chapters of this book:

- Chapter 10: Economic Indicators
- Chapter 11: Data Categorization
- Chapter 12: Data Transformation
- Chapter 13: Data Aggregation
- Chapter 14: Data Source
- Chapter 15: Temporal Patterns
- Chapter 16: Regional Patterns
- Chapter 17: Causal Relationships
- Chapter 18: Data Report on Traditional Economic Indicators

Additionally, complete the following DataCamp courses as guided by Chapter 8:

- Introduction to the Tidyverse
- Data Manipulation with dplyr
- (Optional: Reporting with R Markdown)

18.2 Data Guidelines

For the report, choose at least two of the economic indicators discussed in this module. Conduct a business cycle analysis and a regional comparison for the selected indicators.

18.3 R Markdown Guidelines

Your task is to construct a data report on economic indicators using R Markdown. If you need a primer on R Markdown, refer to Chapter 7. The final product, rendered by knitting your R Markdown file, should be a professional, dynamic PDF document that adheres to the same standards outlined in Chapter 9.3 for the data report on yield curves.

Module III

Survey- and Text-Based Economic Indicators

The third module of this book emphasizes the importance of subjective information captured through survey and text-based economic indicators. Traditional economic analysis often relies on objective measurements such as sales and prices. However, subjective information also plays an important role. The perceptions, expectations, and opinions of individuals, commonly collected from surveys and text-based data, guides people's decision-making processes. These decisions, in turn, dictate consumption, saving, and investment behaviors, thereby influencing economic performance. Consequently, such subjective data sources emerge as key indicators of both current economic activity and future trends. This module familiarizes readers with these unique data sources and provides them with the necessary tools and insights for effective analysis and interpretation.

Module Overview This module consists of five chapters:

- Chapter 19: "Why Subjective Information Matters" explores the significant role subjective information plays in economics, arguing that economics is not merely an analysis of exogenous processes, but involves individuals making decisions based on subjective information, which subsequently impacts economic performance.
- Chapter 20: "Michigan Consumer Survey" elaborates on the consumer survey conducted by the University of Michigan and illustrates how this survey can generate meaningful economic indicators.
- Chapter 21: "Real-Time and Forecaster Data" covers how back-, now-, and forecasts can offer insights into decision-making processes and the creation of potent indicators. This chapter introduces resources such as Philadelphia Fed's real-time data, Survey of Professional Forecasters, and Tealbook data.
- Chapter 22: "Text-Based Economic Indicators" introduces key text-based economic indicators, derived from word counts in newspapers, social media sites, and Google searches.
- Chapter @ref(data-report-on-survey--and-text-based-economic-indicators): "Data Report on Survey- and Text-Based Economic Indicators" provides an assessment that tests the skills and knowledge acquired throughout the module, focusing on a data report on survey- and text-based economic indicators of your choice.

In addition to the chapters, the following DataCamp courses complement the content in this module:

- Introduction to Data Visualization with ggplot2
- Joining Data with dplyr
- (Optional: Intermediate Data Visualization with ggplot2)
- (Optional: Skill Track: Text Mining with R)

Refer to Chapter 8 for tips on how to optimize your learning experience from these DataCamp courses.

Learning Objectives Upon completing Module III, you should be able to:

1. Understand the importance of subjective information in economics and its impact on economic performance.
2. Recognize the purpose and utility of survey-based economic indicators, specifically using the example of the Michigan Consumer Survey.
3. Analyze real-time and forecaster data to construct meaningful economic indicators and to understand economic decision-making processes.
4. Identify and utilize text-based economic indicators, derived from word count analyses of newspapers, social media sites, and Google searches.
5. Navigate and utilize the `ggplot2` and `dplyr` libraries in R to manipulate and visualize data effectively, with additional skills in text mining if the optional DataCamp skill track on Text Mining with R is taken.
6. Apply the skills learned in this module to create a comprehensive data report on survey- and text-based economic indicators.

Learning Activities & Assessments for Module III Throughout this module, you will engage in the following activities:

1. **Reading Material**: Absorb and comprehend the content of the five chapters in Module III, providing you with a firm grasp of the importance of subjective information in economics, and how survey- and text-based indicators can provide meaningful insights.
2. **DataCamp Courses**: Complete the DataCamp courses Introduction to Data Visualization with ggplot2 and Joining Data with dplyr to learn how to visualize and merge data effectively. The optional courses, Intermediate Data Visualization with ggplot2 and Skill Track: Text Mining with R, can further enhance your skills in data visualization and text mining.
3. **Data Report**: Create a data report on survey- and text-based economic indicators of your choice using R and R Markdown.

Your progress will be evaluated based on:

1. **DataCamp Course Completion**: Completion of the designated DataCamp courses, specifically Introduction to Data Visualization with ggplot2 and Joining Data with dplyr, forms a key part of your assessment.
2. **Data Report on Survey- and Text-Based Economic Indicators**: Craft a data report on survey- and text-based economic indicators of your choice using R Markdown. This report will serve as a practical demonstration of your grasp of R, RStudio, R Markdown, and the survey- and text-based indicators. The report should exhibit your ability to analyze and interpret survey- and text-based data, and the quality of your report will form a significant part of your module assessment. Refer to Chapter @ref(data-report-on-survey--and-text-based-economic-indicators) for guidelines on this task.

Recall that comprehending the intricacies of survey- and text-based economic indicators is a slow process. Make sure to immerse yourself fully in each concept before progressing to the next. Never hesitate to ask for help if you face any challenges during your learning process. Good luck with your exploration of survey- and text-based economic indicators!

19 Why Subjective Information Matters

Content Coming Soon

20 Michigan Consumer Survey

Content Coming Soon

21 Real-Time and Forecaster Data

Content Coming Soon

22 Text-Based Economic Indicators

Content Coming Soon

23 Data Report on Survey- and Text-Based Economic Indicators

The goal of this assignment is to create a data report on survey- and text-based economic indicators. The report should be prepared using R Markdown and must be a minimum of four pages long, containing both graphs and text.

Before starting your report, ensure that you have adequately completed the preparations outlined in Chapter 23.1. Your report should align with the guidelines provided in Chapter 23.2 and Chapter 23.3. Upon finishing, please submit your data report via the course website (Blackboard).

23.1 Preparations

To set a solid foundation for this assignment, review the following chapters of this book:

- Chapter 19: Why Subjective Information Matters
- Chapter 20: Michigan Consumer Survey
- Chapter 21: Real-Time and Forecaster Data
- Chapter 22: Text-Based Economic Indicators
- Chapter @ref(data-report-on-survey--and-text-based-economic-indicators): Data Report on Survey- and Text-Based Economic Indicators

In addition, complete the following DataCamp courses as advised by Chapter 8:

- Introduction to Data Visualization with ggplot2
- Joining Data with dplyr
- (Optional: Intermediate Data Visualization with ggplot2)
- (Optional: Skill Track: Text Mining with R)

23.2 Data Guidelines

Select one of the survey- or text-based indicators explored in this module for your report. Furthermore, use the data discussed in this module to create your own survey- or text-based indicator. Illustrate why this new indicator could yield valuable insights. Conduct a business cycle analysis for both the chosen and your newly created survey- or text-based indicators.

23.3 R Markdown Guidelines

Your assignment involves drafting a data report on survey- and text-based economic indicators using R Markdown. If you need an introduction to R Markdown, you can refer to Chapter 7. The final output, produced by knitting your R Markdown file, should be a polished, dynamic PDF document that follows the same standards detailed in Chapter 9.3 for the data report on yield curves.