MAPPING WORLDS WITH R

Your Journey from Beginner to Data Cartographer

MILOS POPOVIC

Contents

Dedication **1**

Before we start **3**

Quick Start: Your First Steps with R! **5**
Installing Your Tools: R and RStudio 5
Let's Install RStudio (The Dashboard) 6
Your First Inter**Action:** Running a Command 7
1. Open RStudio 7
2. Find the Console 7
3. Type Your First Command 7
4. Check the Output 8

Chapter 1: Your Mapping Cockpit: Meet R & RStudio **11**
Learning Objectives 11
Building on Previous Knowledge 11
Concepts: Understanding Your Workspace 12
R vs. RStudio: Engine and Dashboard Recap 12
The Four Panes of RStudio 12
Script Editor (or Source Pane): (Usually Top-Left) 12
Console: (Usually Bottom-Left) 13
Environment / History / Connections / ... Pane: (Usually Top-Right) . . . 13
Files / Plots / Packages / Help / Viewer Pane: (Usually Bottom-Right) . . 13
Writing and Running Your First Script 14
Staying Organized: RStudio Projects (Your Project Hub) 16
R Packages: Your Specialized Toolkits 18
Common Setup Hurdles: 20
Sneak Peek: Coordinate Reference Systems (CRS) - Why Maps Align 20
Chapter 1 Summary: Key Takeaways & Common Pitfalls 24
Key Takeaways: 24
Common Pitfalls for Beginners (and Solutions): 25
Review & Connect: What's Next? 25

Chapter 2: Understanding Spatial Data in R: Shapes, Grids, and Addresses **27**
Learning Objectives 27
Building on Previous Knowledge 28
Concepts: The Building Blocks of Digital Maps 28
Two Flavors of Spatial Data: Vector vs. Raster 28
Meet *sf* Objects: Your Spatial Data Frames 29
Coordinate Reference Systems (CRS) Revisited: The Map's Address Book 29

Why CRS is CRITICAL: 30
Identifying CRSs: EPSG Codes 30
Common Spatial File Formats 31
Common Spatial File Formats 31
Hands-On: Exploring Spatial Data Objects in R 33
Chapter 2 Summary: Key Takeaways & Common Pitfalls 41
Review & Connect: What's Next? 42

Chapter 3: Finding and Preparing Spatial Data: Your Ingredients **43**
Learning Objectives 43
Building on Previous Knowledge 43
Concepts: Finding Your Ingredients and Checking Quality 44
Where to Find Spatial Data: Reliable Sources 44
Is This Data Any Good? Metadata and Quality Checks 45
Choosing Data Formats (Recap) 46
Ethical Considerations: Data Selection & Bias 46
Hands-On: Loading and Cleaning Your Own Data! 46
The Key to Easy File Loading: Working Directories & RStudio Projects! . . 47
Loading Vector Data (`sf::st_read()`) 47
Loading Tabular Data (`readr::read_csv()`) 50
Basic Data Cleaning with `dplyr` 51
Joining Tabular Data to Spatial Data (`dplyr::left_join()`) 54
Saving Your Cleaned Data (`sf::st_write()`) 56
Chapter 3 Summary: Key Takeaways & Common Pitfalls 57
Key Takeaways 57
Common Pitfalls (and Solutions) 57
Review & Connect: What's Next? 58

Chapter 4: Creating Your First Thematic Map with ggplot2 **59**
Learning Objectives 59
Building on Previous Knowledge 59
Concepts: Telling Stories with Maps 60
What are Thematic Maps? 60
Focus: Choropleth Maps 60
Color Kitchen: Choosing the Right Palette 60
Map Design Basics: Visual Hierarchy & Accessibility 61
Hands-On: Your First `ggplot2` Thematic Map! 62
Introducing `ggplot2`: Mapping in Layers 62
Creating a Basic Choropleth Map 63
Customizing Colors (`scale_fill_*`) 65
Adding Other Map Elements (`labs`, `ggspatial`) 66
Project Exercise: Mapping African Indicators 69
Chapter 4 Summary: Key Takeaways & Common Pitfalls 73
Key Takeaways 73
Common Pitfalls (and Solutions) 73
Review & Connect: What's Next? 73

Chapter 5: Point Maps and Geographic Visualization **75**
Learning Objectives 75
Building on Previous Knowledge 75
Concepts: Working with Points 76
What is Point Data? (Recap) 76

When Are Point Maps Most Effective? . . . 76
The Overplotting Problem: Too Many Points! . . . 76
Spatial Joins: Linking Points to Polygons (`st_join`) . . . 77
Ethical Note: Point Data Privacy & Bias . . . 77
Hands-On: Plotting Points with `ggplot2` . . . 78
1. Setup: Load Packages and Get Data . . . 78
2. Creating a Simple Dot Map . . . 79
3. Creating a Bubble Map (Size by Population) . . . 81
4. Styling Points by Category (Color/Shape) . . . 83
5. Handling Labels with `ggrepel` . . . 85
6. Basic Point Pattern Analysis (Visual Only!) . . . 88
Project Exercise: Map German Cities with Population Markers . . . 88
Chapter 5 Summary: Key Takeaways & Common Pitfalls . . . 93
Key Takeaways . . . 93
Common Pitfalls (and Solutions) . . . 93
Review & Connect: What's Next? . . . 94

Chapter 6: Line and Network Maps: Tracing Paths 95
Learning Objectives . . . 95
Building on Previous Knowledge . . . 95
Concepts: Understanding Lines and Networks . . . 96
Working with Line Data (Linestrings) . . . 96
Fetching Line Data from OpenStreetMap (`osmdata`) . . . 96
Why Map Lines? Networks and Connectivity . . . 97
Smoothing the Path: Line Simplification (`sf::st_simplify`) . . . 97
Advanced Concepts (Briefly!) . . . 98
Hands-On: Working with Line Data . . . 98
1. Setup: Load Packages . . . 98
2. Fetching Line Data (`osmdata`) . . . 99
3. Styling Lines in `ggplot2` . . . 100
4. Line Simplification (`sf::st_simplify`) . . . 102
5. Creating Simple Flow Maps (Conceptual + Basic Example) . . . 105
Project Exercise: Create a Styled River or Transportation Network Map . 107
Chapter 6 Summary: Key Takeaways & Common Pitfalls . . . 109
Key Takeaways . . . 109
Common Pitfalls (and Solutions) . . . 109
Review & Connect: What's Next? . . . 111

Chapter 7: Understanding Raster Data - The World in Pixels 113
Learning Objectives . . . 113
Building on Previous Knowledge . . . 113
Concepts: Welcome to the Grid! . . . 114
Raster Data Explained: Pixels and Values . . . 114
Where Does Raster Data Come From? . . . 114
Anatomy of a Raster: What's Inside? . . . 115
Meet `terra`: Your Raster Toolkit in R . . . 116
Understanding `print()` Output for `terra` Rasters . . . 116
Hands-On: Working with Your First Raster . . . 117
1. Setup: Load Packages . . . 117
2. Loading Raster Data (`terra::rast()`) . . . 117
3. Inspecting Your Raster (`terra` functions) . . . 118
4. Basic Plotting: Seeing Your Raster . . . 120

5. Simple Raster Operations: Cropping (`terra::crop()`) 125
Project Exercise: Explore Your Region's Elevation 127
Chapter 7 Summary: Key Takeaways & Common Pitfalls 131
Key Takeaways 131
Common Pitfalls (and Solutions): Raster Roadblocks 131
Review & Connect: What's Next? 132

Chapter 8: Terrain and Elevation Visualization: Shaping the Land 133
Learning Objectives 133
Building on Previous Knowledge 133
Concepts: Revealing the Shape of the Land 134
Using `terra` for Terrain Analysis 134
Finding and Downloading Elevation Data (DEMs) 135
Ethical Considerations in Terrain Representation 135
Hands-On: Visualizing Elevation Data 136
1. Setup: Load Packages 136
2. Get Data: Elevation for Switzerland 136
3. Calculate Terrain Attributes (Slope, Aspect, Hillshade) 138
4. Creating Shaded Relief Maps with `ggplot2` (The Cool Part!) 141
5. Contour Generation and Styling 144
6. Combining Elevation with Other Features 147
7. Performance Tips (Brief Recap) 149
Project Exercise: Create a Topographic Map with Styled Elevation 149
Chapter 8 Summary: Key Takeaways & Common Pitfalls 154
Key Takeaways 154
Common Pitfalls (and Solutions): Terrain Troubles 154
Review & Connect: What's Next? 154

Chapter 9: Interactive Maps with leaflet: Bringing Your Maps to Life! 157
Learning Objectives 157
Building on Previous Knowledge 157
Concepts: Why Make Maps Interactive? 158
Static vs. Interactive Maps: When to Choose Which? 158
Meet `leaflet`: Your Interactive Mapping Toolkit 158
Hands-On: Your First Interactive Map with `leaflet` 159
1. Setup: Load Packages 159
2. Simple Start: An Empty Map with Base Tiles 159
3. Choosing Your Background: Provider Tiles 161
4. Adding Your Data: Markers and Polygons 162
5. Customizing Popups with HTML 166
6. Giving Users Control: Layers Control 168
7. Saving Your Interactive Map (`htmlwidgets::saveWidget`) 170
Project Exercise: Build an Interactive Map of Global Landmarks 172
Chapter 9 Summary: Key Takeaways & Common Pitfalls 175
Key Takeaways 175
Common Pitfalls (and Solutions): Interactive Snags 175
Review & Connect: What's Next? 175

Chapter 10: Introduction to 3D Mapping with rayshader: Adding Depth 177
Learning Objectives 177
Building on Previous Knowledge 178
Concepts: Thinking in Three Dimensions 178

Why Go 3D? . . . 178
Hardware Check: Can My Computer Handle This? . . . 179
How 2D Becomes 3D: Heightmaps and Textures . . . 179
Hands-On: Your First 3D Map with `rayshader` . . . 180
1. Setting Up `rayshader` . . . 180
2. Preparing Data: Raster to Matrix . . . 181
3. Creating Simple 3D Terrain Models . . . 182
4. Controlling the View: Camera Angle and Lighting . . . 189
5. Saving Your 3D Masterpiece! . . . 190
Project Exercise: Create Your Own 3D Elevation Model . . . 194
Chapter 10 Summary: Key Takeaways & Common Pitfalls . . . 198
Key Takeaways . . . 198
Common Pitfalls (and Solutions): Navigating 3D Challenges . . . 198
Review & Connect: What's Next? . . . 199

Chapter 11: Creating Map Animations and Time Series with gganimate 201
Learning Objectives . . . 201
Building on Previous Knowledge . . . 201
Concepts: Visualizing Change Over Time . . . 202
What is Spatial Time Series Data? . . . 202
Storytelling with Animated Maps . . . 202
Animation Tool: `gganimate` . . . 202
Hands-On: Animating `ggplot2` Maps with `gganimate` . . . 203
1. Setup: Load Packages . . . 203
2. Prepare Data: Spatial + Time Series . . . 204
3. Build the Base Static `ggplot` Map . . . 205
4. Adding Animation with `transition_time()` . . . 207
5. Smoothing Transitions (`ease_aes`) . . . 208
6. Saving Your Animation (`anim_save`) . . . 209
Project Exercise: Create an Animated Map Showing GDP Changes . . . 211
Chapter 11 Summary: Key Takeaways & Common Pitfalls . . . 213
Key Takeaways . . . 213
Common Pitfalls (and Solutions): Animation Annoyances . . . 214
Review & Connect: What's Next? . . . 214

Chapter 12: Putting It All Together: Advanced Projects 217
Learning Objectives . . . 217
Building on Previous Knowledge . . . 217
Concepts: From Techniques to Projects . . . 218
Combining Techniques: Multi-Layered Storytelling . . . 218
Map Design Revisited: Beyond the Basics . . . 218
Ethics Revisited: Deeper Considerations . . . 219
Integrating R Maps: R Markdown and Shiny . . . 219
Hands-On: Project Ideas and Structure . . . 220
Project Idea 1: Regional Environmental Analysis . . . 220
Project Idea 2: Historical Urban Change . . . 220
Project Idea 3: Travel Time / Accessibility Analysis . . . 221
Structuring Your Project . . . 221
Continuing Your R Spatial Journey . . . 222
Key Package Documentation (Tutorials/Vignettes) . . . 222
Online Books & Tutorials . . . 222
Community . . . 222

Practice! 223

Appendix **225**

Dedication

For Irene, whose boundless patience and encouragement have fueled my passion and made this book a reality.

Before we start

More than a decade ago, I embarked on my own journey into the fascinating world of mapping with R. My beginnings were humble, spending countless hours wrestling with `ggplot2`, trying to produce a simple yet meaningful choropleth map for my paper or PhD thesis. Each map I made felt like a small victory against a tide of confusing code and seemingly endless errors. But every struggle carried a lesson, every mistake a revelation.

Fast forward ten years, and those initial frustrations have transformed into joy. Today, I run ”Milos Makes Maps”, my YouTube channel dedicated to sharing advanced mapping projects and techniques. What started as personal trials and errors evolved into a vibrant community where thousands of learners now benefit from free tutorials and insights into geographic visualization using R.

Why, you might ask, should you bother learning to code for mapping when platforms like ArcGIS and QGIS exist? The answer is simple yet profound: freedom and flexibility. Coding empowers you to understand your data deeply, manipulate it precisely, and communicate it creatively. With R, you're not limited by proprietary tools or predefined options. Instead, you're given a blank canvas and infinite possibilities to craft unique, insightful maps tailored exactly to your needs.

With this book, I aimed to create the resource I wish I had when I was just starting. Each chapter is crafted to guide absolute beginners clearly and encouragingly, from installing R and RStudio to publishing polished and ethical maps. My approach is fundamentally problem-solving oriented. You'll find eleven detailed projects alongside dozens of examples, each accompanied by step-by-step explanations and full code transparency. Unlike many other guides, this book provides access to authentic, real-world datasets directly through R packages. You won't just replicate abstract examples; you'll learn to navigate the quirks and complexities of actual data.

My advice for approaching this book? Dive in headfirst—set up R and RStudio immediately and start coding. Play around with the code, experiment boldly, make mistakes, and learn how to fix them. Each chapter builds gradually in complexity, laying a solid foundation upon which you can confidently progress. To make your journey smoother, I've prepared a GitHub repository containing all the code from every chapter—be sure to explore and leverage it.

Mapping the world with R changed my career, my perspective, and my life. My hope is that this book becomes your guide, companion, and inspiration as you begin your own exciting journey into data cartography. Happy mapping!

— Milos Popovic

Quick Start: Your First Steps with R!

- **Goal:** Get R and RStudio installed, open RStudio, and run your very first command successfully in about 15-20 minutes.
- **Difficulty:** ★☆☆ (Just one star - You absolutely can do this!)
- **Featured "Map":** None yet! Just getting the tools ready.

Hello there, and welcome! 👋 I'm so excited you're starting this journey to create amazing maps using a powerful tool called R. Feeling a bit nervous because you've maybe never written code before? That is completely, 100% okay! Seriously, this book is designed just for you, the absolute beginner. We'll take everything one tiny, manageable step at a time, and I promise to be your friendly guide.

Think of it like learning to cook. Before you can make a fancy meal, you need to get your kitchen set up, right? You need an oven (that's like R, the powerful engine) and a countertop with your tools laid out (that's like RStudio, the friendly dashboard we'll use to work with R).

Our **only goal** in this super-quick section is to install these two free tools, R and RStudio, and run one single command to make sure everything's working. That's it! No complex mapping yet, just getting our "kitchen" ready.

Ready to take the very first, small step? Let's do it together!

Installing Your Tools: R and RStudio

This is the most important first step. We need both R (the engine) and RStudio (the dashboard).

1. The Super-Simple Install Plan:
   - **Goal:** Install R first, then RStudio.
   - **Where to Go:** We'll visit two websites.
   - **What to Click:** We'll guide you to the big download buttons!
   - **Installation:** Mostly just clicking "Next" or "Continue" and accepting the defaults.

2. Need Detailed Steps?
   Installing software can sometimes have little hiccups depending on your computer. If you get stuck or want screenshots for every single click, please flip to **Appendix A – Detailed Installation Guide** at the back of the book. It has step-by-step pictures for Windows and macOS.

3. Let's Install R (The Engine):

   1. Open your web browser (Chrome, Safari, Firefox, Edge).

   2. Go to the official R website: https://www.r-project.org/

   3. Look for a link like "download R" or "CRAN" (The Comprehensive R Archive Network) near the top or under "Getting Started." Click it.

   4. You'll see a list of countries ("mirrors"). Click one that's reasonably close to you geographically.

   5. Now, click the link for your operating system: "Download R for Windows" or "Download R for macOS."
      - **Windows:** Click the link for "base" or "install R for the first time." Then click the big download link (like "Download R-x.y.z for Windows"). Find the downloaded `.exe` file (usually in your Downloads folder) and double-click it. Follow the installer steps—accepting the defaults is usually perfect! Click "Finish" at the end.

      - **macOS:** Look for the `.pkg` file link that matches your Mac's chip (check **Apple Menu > About This Mac:** "Apple M..." means arm64, "Intel" means x86_64). Download the correct `.pkg` file. Find it (usually in Downloads) and double-click it. Follow the installer steps, clicking "Continue", "Agree", and "Install" (you might need your password). Click "Close" when done. You can move the installer file to the Trash.

What just happened?
You should now have R installed!
You probably won't see much happen and that's normal. R is the engine running under the hood. Now for the dashboard!

Let's Install RStudio (The Dashboard)

1. Go to the Posit website (they make RStudio): https://posit.co/
2. Look for **Download RStudio** or navigate to **Products → RStudio**.

3. Find **RStudio Desktop** and make sure you choose the **FREE / Open Source** license version. Click the main download button.
4. The site should suggest the correct installer for your system (Windows `.exe` or macOS `.dmg`). Download it.
    - **Windows:** Find the downloaded `.exe` file and double-click it. Click **Yes** if asked for permission, then **Next** → **Next** → **Install** → **Finish**.

    - **macOS:** Double-click the downloaded `.dmg`. In the window that appears, drag the **RStudio** icon onto the **Applications** folder icon. When it's done, close the window and eject the disk image.

What just happened?
Great! RStudio is installed
Find the RStudio icon (often a blue circle with an "R" inside) in **Applications** (Mac) or the **Start Menu** (Windows) and get ready to open it!

Your First InterAction: Running a Command

Let's fire up RStudio and make sure it talks to the R engine.

1. Open RStudio

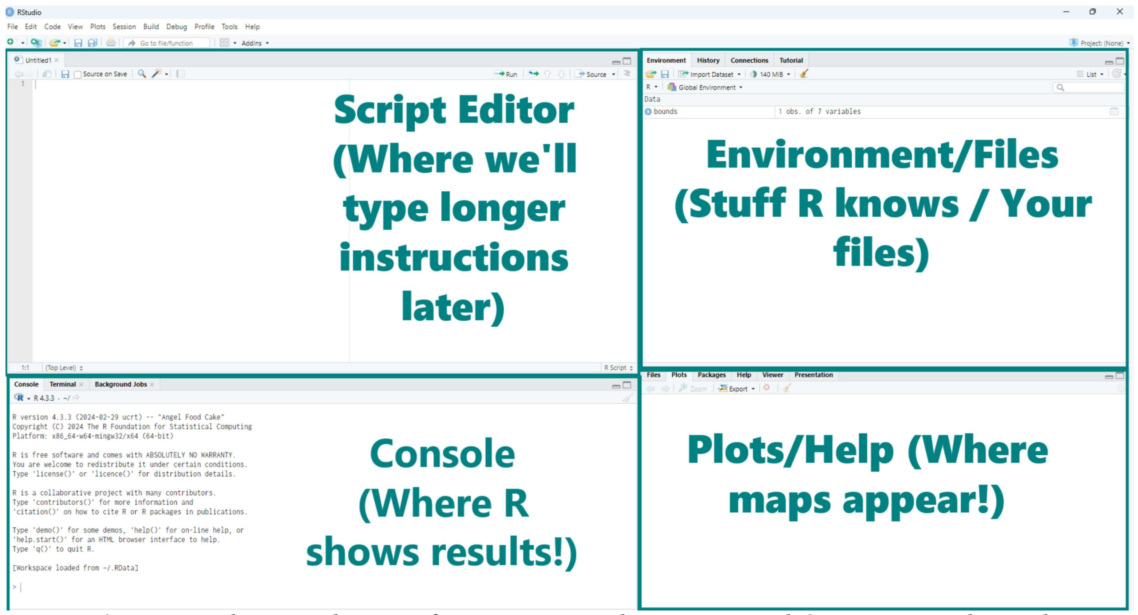
- Find the RStudio application icon you just installed (in your Applications folder on macOS, or Start Menu on Windows) and double-click it.
- **(macOS only):** It might ask, "Are you sure you want to open an application downloaded from the Internet?". Click "**Open**".

2. Find the Console

- Look at the RStudio window. The **bottom-left pane** is called the **Console**. You should see some text there already, including the version of R you installed (like "R version 4.x.x..."). This is R saying hello!
- You'll see a blinking cursor next to a > symbol. This means R is ready for your command!

3. Type Your First Command

- Click into the Console pane so your cursor is blinking next to the >.
- Carefully type the following command exactly as shown (R is case-sensitive!): `print("Hello, R Mapper!")`
- Press the **Enter** key (or **Return** key) on your keyboard.


Script Editor (Where we'll type longer instructions later)
Environment/Files (Stuff R knows / Your files)
Console (Where R shows results!)
Plots/Help (Where maps appear!)
Console
Terminal
Background Jobs
Environment
History
Connections
Tutorial
Files
Plots
Packages
Help
Viewer
Presentation

```
R version 4.3.3 (2024-02-29 ucrt) -- "Angel Food Cake"
Copyright (C) 2024 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from ~/.RData]

> |
```

Figure 1: *The RStudio interface upon initial opening, with key panes indicated.*

4. Check the Output

- Right below the command you typed, R should print back: `[1] "Hello, R Mapper!"`.

What you should see:

```
> print("Hello, R Mapper!")
[1] "Hello, R Mapper!"
>
```

The `[1]` simply marks the first element of the output.

What just happened?

- You opened RStudio, the friendly interface.
- RStudio automatically found and started the R engine in the background (you saw its version message in the Console).
- You typed a command (`print(...)`) directly into the Console.
- The `print()` command tells R to display whatever is inside the parentheses () and quotes ””.
- R executed your command and printed the text `"Hello, R Mapper!"` back to you in the Console. The `[1]` just indicates it’s the first piece of output.

Congratulations! You have successfully installed the necessary software and run your very first line of R code! That’s a huge step. Getting the setup right is often the biggest hurdle for beginners, so well done!

You now have your “mapping kitchen” ready. In Chapter 1, we’ll properly explore the

RStudio environment and start learning the basic R “ingredients” we need before we make our first actual map.

Feeling good? Ready for Chapter 1?

Chapter 1: Your Mapping Cockpit: Meet R & RStudio

- **Difficulty:** ★☆☆ (One star - Still very beginner-focused!)
- **Featured Map:** None in this chapter! Focus is on tools and basic concepts.

Learning Objectives

Okay, before we dive in, what will you be able to do after this chapter?

✓ Confidently navigate the main windows (panes) in RStudio.

✓ Understand the difference between the Script Editor and the Console.

✓ Write, run, and save simple R commands in a script file.

✓ Understand what R packages are and install your first few essential mapping toolkits.

✓ Recognize basic R concepts like variables, assignment (`<-`), and simple data types (numbers, text).

✓ Understand why using RStudio Projects is super helpful.

✓ Check the Coordinate Reference System (CRS) of spatial data (a sneak peek!).

Building on Previous Knowledge

Welcome to Chapter 1! If you completed the Quick Start, you've already installed R and RStudio and run your first command (`print("Hello, R Mapper!")`). That's awesome! If you skipped the Quick Start, please make sure you have R and RStudio installed now (see Appendix A if you need help).

In this chapter, we'll get properly acquainted with our main tool, RStudio. Think of it as learning your way around the cockpit before you fly the plane. We'll explore the different panels, learn how to give R instructions more formally using scripts, and install some special "toolkits" (called packages) that we'll need for mapping later. We'll also touch on the absolute basics of the R language itself – just enough to get started. Let's explore your mapping cockpit!

Concepts: Understanding Your Workspace

Let’s get comfortable with where we’ll be working.

R vs. RStudio: Engine and Dashboard Recap

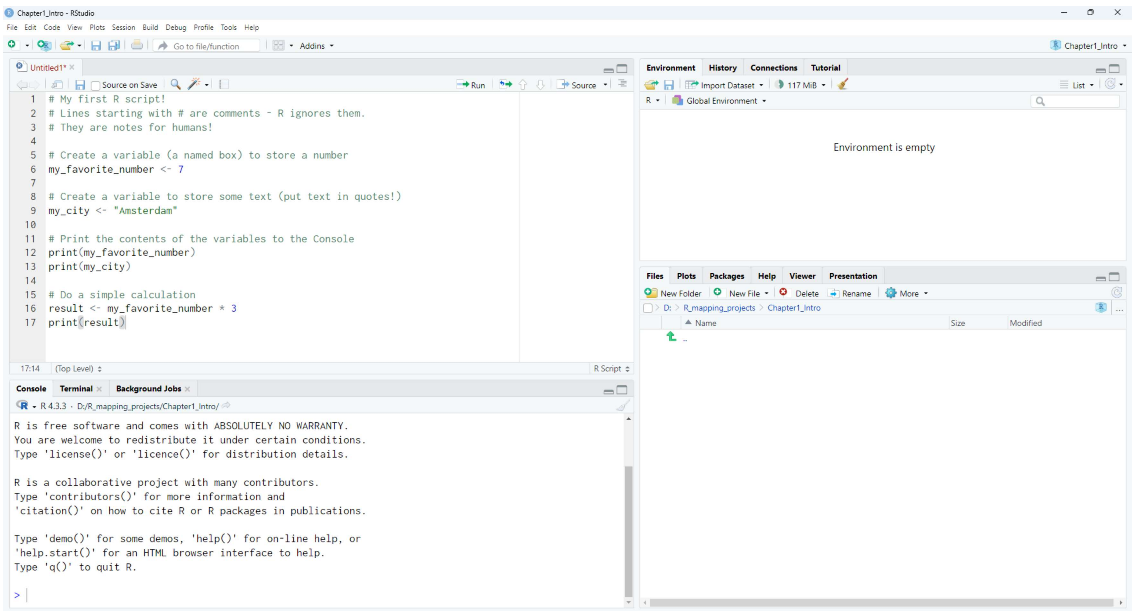
Just to quickly remind ourselves from the Quick Start:

- R: The powerful but mostly invisible engine. It understands the R language, does calculations, manages data, creates plots. We don’t usually interact with it directly.
- RStudio: The friendly dashboard or cockpit. It’s a separate application that makes working with the R engine much, much easier. It gives us organized windows (panes) for writing code, seeing results, managing files, viewing plots, etc.

When you open RStudio, you are using the dashboard to talk to the R engine.

The Four Panes of RStudio

RStudio usually shows four main windows or “panes”. Let’s look at them again:


Chapter1_Intro - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Chapter1_Intro
Untitled1*
Source on Save
Run
Source
17:14
(Top Level)
R Script
Environment History Connections Tutorial
Import Dataset
117 MiB
List
R
Global Environment
Environment is empty
Files Plots Packages Help Viewer Presentation
New Folder New File Delete Rename More
D: R_mapping_projects Chapter1_Intro
Name Size Modified
Console Terminal Background Jobs
R 4.3.3 · D:/R_mapping_projects/Chapter1_Intro/

```r
# My first R script!
# Lines starting with # are comments - R ignores them.
# They are notes for humans!

# Create a variable (a named box) to store a number
my_favorite_number <- 7

# Create a variable to store some text (put text in quotes!)
my_city <- "Amsterdam"

# Print the contents of the variables to the Console
print(my_favorite_number)
print(my_city)

# Do a simple calculation
result <- my_favorite_number * 3
print(result)
```

```
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Figure 2: *The RStudio interface revisited*

Let’s break them down:

Script Editor (or Source Pane): (Usually Top-Left)

- **What it is:** A text editor built into RStudio.

- **Why use it:** This is where you'll write and save sequences of R commands (called scripts). Think of it like writing down a recipe step-by-step. It's much better than typing directly into the Console for anything more than one simple command, because you can easily edit, save, and re-run your work.
- **How to get one:** Go to File -> New File -> R Script.

Console: (Usually Bottom-Left)

- **What it is:** The direct line to the R engine.
- **Why use it:** This is where R executes commands and displays text output, messages, and importantly, error messages. You can type commands here directly (like we did with print() in the Quick Start), but it's best for quick tests or single commands. Commands typed here aren't automatically saved.
- **Key Feature:** The > prompt means R is waiting for your next command.

Environment / History / Connections / ... Pane: (Usually Top-Right)

- **What it is:** Has multiple tabs.
- **Environment Tab:** Super useful! Shows you any objects (like data tables or variables) that R currently has stored in its memory. We'll see things appear here soon!
- **History Tab:** Shows a list of commands you've run previously. Handy for recalling steps.
- (Other tabs like Connections, Build, Git are more advanced - ignore them for now).

Files / Plots / Packages / Help / Viewer Pane: (Usually Bottom-Right)

- **What it is:** Another multi-tab area.
- **Files Tab:** Shows the files and folders in your current working directory (more on this soon!). Like a mini file explorer.
- **Plots Tab:** This is where your maps and graphs will appear! Very important for us. You can zoom, export, and navigate plots here.
- **Packages Tab:** Lists all the R "toolkits" (packages) installed on your computer. You can check which ones are currently loaded (active) here.
- **Help Tab:** Displays documentation for R functions and packages. Super helpful when you forget how a command works!
- **Viewer Tab:** Used to display local web content, like the interactive leaflet maps we'll make much later (Chapter 9).

Don't worry about memorizing everything right now! The key is knowing that the Script Editor is for writing/saving code, the Console is where R runs it and talks back, and the Plots pane is where visuals appear. You'll get comfortable navigating these as we go.

Writing and Running Your First Script

Typing directly into the Console is okay for quick tests, but for anything more, we use scripts.

1. Create a New Script: In RStudio, go to the menu: File -> New File -> R Script. The Script Editor pane (top-left) should now show an empty tab, probably named "Untitled1".

2. Type Some Commands: In the empty Script Editor, type the following lines:

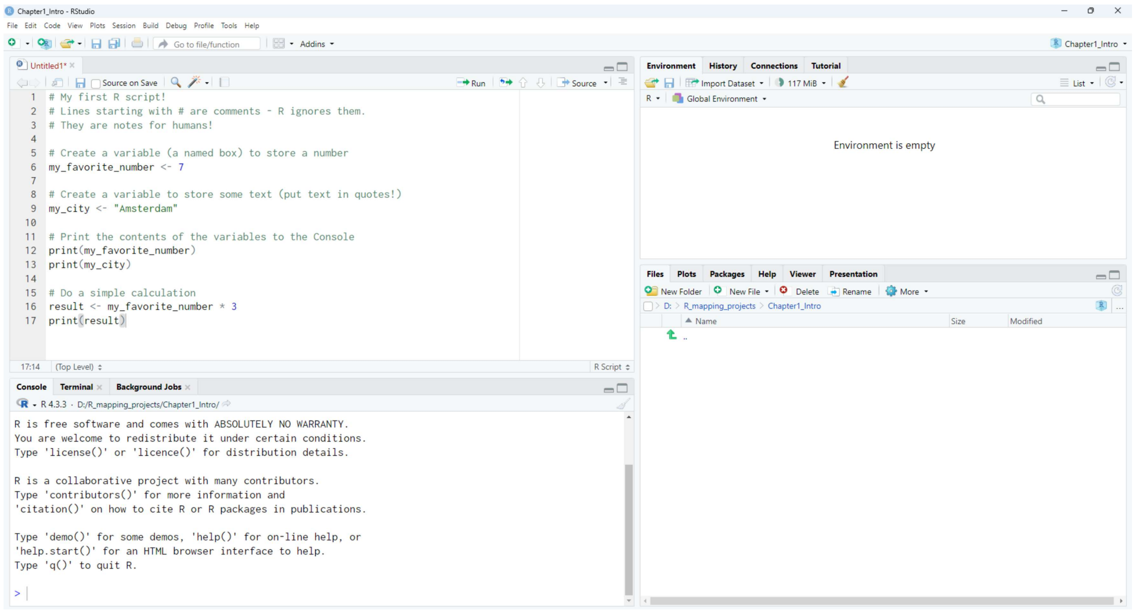
```r
# My first R script!
# Lines starting with # are comments - R ignores them.
# They are notes for humans!

# Create a variable (a named box) to store a number
my_favorite_number <- 7

# Create a variable to store some text (put text in quotes!)
my_city <- "Amsterdam"

# Print the contents of the variables to the Console
print(my_favorite_number)
print(my_city)

# Do a simple calculation
result <- my_favorite_number * 3
print(result)
```



```r
# My first R script!
# Lines starting with # are comments - R ignores them.
# They are notes for humans!

# Create a variable (a named box) to store a number
my_favorite_number <- 7

# Create a variable to store some text (put text in quotes!)
my_city <- "Amsterdam"

# Print the contents of the variables to the Console
print(my_favorite_number)
print(my_city)

# Do a simple calculation
result <- my_favorite_number * 3
print(result)
```

```
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Environment is empty

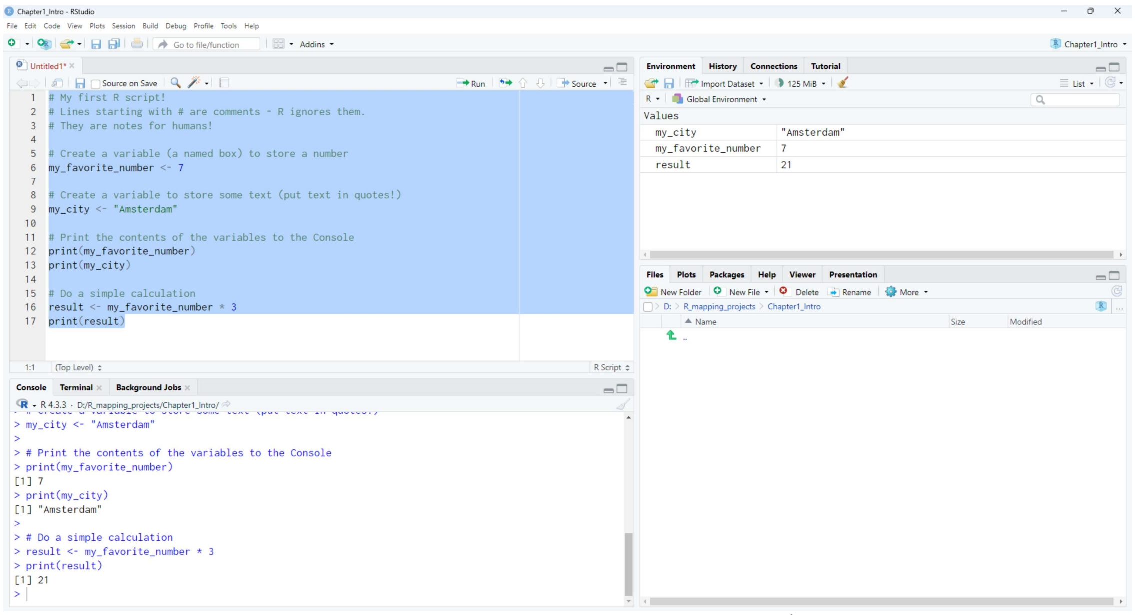
Figure 3: *Script Typed in Editor*

3. Run the Code: Now, how do we tell R to actually do what's written in the script? You

have options:

- **Run One Line:** Click anywhere on a single line of code (e.g., the line `my_favorite_number <- 7`) and then click the "Run" button (often looks like a green arrow pointing right) at the top-right of the Script Editor pane. OR, press `Ctrl+Enter` (Windows/Linux) or `Cmd+Enter` (macOS). You should see that line appear in the Console below, followed by the > prompt. Nothing else happens yet, because assigning a variable doesn't produce visible output.
- **Run Multiple Lines:** Select (highlight) several lines of code with your mouse. Then click the "Run" button or press `Ctrl+Enter` / `Cmd+Enter`. R will execute all the selected lines in order in the Console.
- **Run the Entire Script:** Click anywhere in the Script Editor, then click the "Source" button (often next to the Run button, might look like a page icon with an arrow). OR, press `Ctrl+Shift+Enter` (Windows/Linux) or `Cmd+Shift+Enter` (macOS). R will execute the entire script from top to bottom in the Console.

Try running the entire script now! (Click Source or use `Ctrl+Shift+Enter` / `Cmd+Shift+Enter`).



```
# My first R script!
# Lines starting with # are comments - R ignores them.
# They are notes for humans!

# Create a variable (a named box) to store a number
my_favorite_number <- 7

# Create a variable to store some text (put text in quotes!)
my_city <- "Amsterdam"

# Print the contents of the variables to the Console
print(my_favorite_number)
print(my_city)

# Do a simple calculation
result <- my_favorite_number * 3
print(result)
```

```
R 4.3.3 · D:/R_mapping_projects/Chapter1_Intro/
> my_city <- "Amsterdam"
>
> # Print the contents of the variables to the Console
> print(my_favorite_number)
[1] 7
> print(my_city)
[1] "Amsterdam"
>
> # Do a simple calculation
> result <- my_favorite_number * 3
> print(result)
[1] 21
>
```

```
Values
  my_city              "Amsterdam"
  my_favorite_number   7
  result               21
```

Figure 4: *Script Run Output in Console*

4. Check the Console and Environment: Look at the Console (bottom-left). You should see all the commands from your script executed, and the output from the print() commands: `[1] 7`, `[1] "Amsterdam"`, and `[1] 21`. Now look at the Environment tab (top-right). You should see three items listed under "Values": `my_city`, `my_favorite_number`, and `result`, along with their current values! R now remembers these variables.

5. Save Your Script: Your code in the Script Editor isn't saved automatically! Click the Save icon (looks like a floppy disk) at the top of the Script Editor, or go to File -> Save. RStudio will ask where to save it and what to name it. For now, maybe save it in your Documents folder and name it `my_first_script.R`. (We'll learn a better

way to organize files with Projects soon!). R script files always end with `.R`.

What just happened?

- You created an R Script file (.R) in the Script Editor.
- You wrote comments (#) to explain your code.
- You created variables (`my_favorite_number`, `my_city`, `result`) using the assignment arrow (`<-`) to store numbers and text. Think of `<-` as "gets". So, `my_favorite_number <- 7` means `"my_favorite_number gets the value 7"`.
- You used `print()` to display the values stored in your variables in the Console.
- You learned how to run code from the script (single line, multiple lines, or the whole script using Run/Source buttons or keyboard shortcuts).
- You saw the variables appear in the Environment pane, showing R remembers them.
- You saved your script file so you can open and reuse it later!

Using scripts is the standard way to work in R for anything beyond trivial tasks. It keeps your work organized, editable, and reproducible!

Staying Organized: RStudio Projects (Your Project Hub)

Okay, we just saved `my_first_script.R` somewhere like your Documents folder. That's fine for a single file, but imagine a real mapping project: you'll have multiple scripts, data files (maps, tables), output images... things can get messy fast!

Also, remember how R needs to know where to find files (file paths)? If you just tell R to load `my_data.csv`, how does it know which folder to look in? It looks in the current working directory. Trying to manage this manually can be confusing and lead to errors like "File not found!".

The Solution: RStudio Projects!

- **What it is:** An RStudio Project groups all the files related to a single analysis or project (scripts, data, outputs) into one self-contained folder.
- **The Magic:** When you open an RStudio Project (by double-clicking a special `.Rproj` file inside the project folder), RStudio automatically sets the working directory to that project folder!

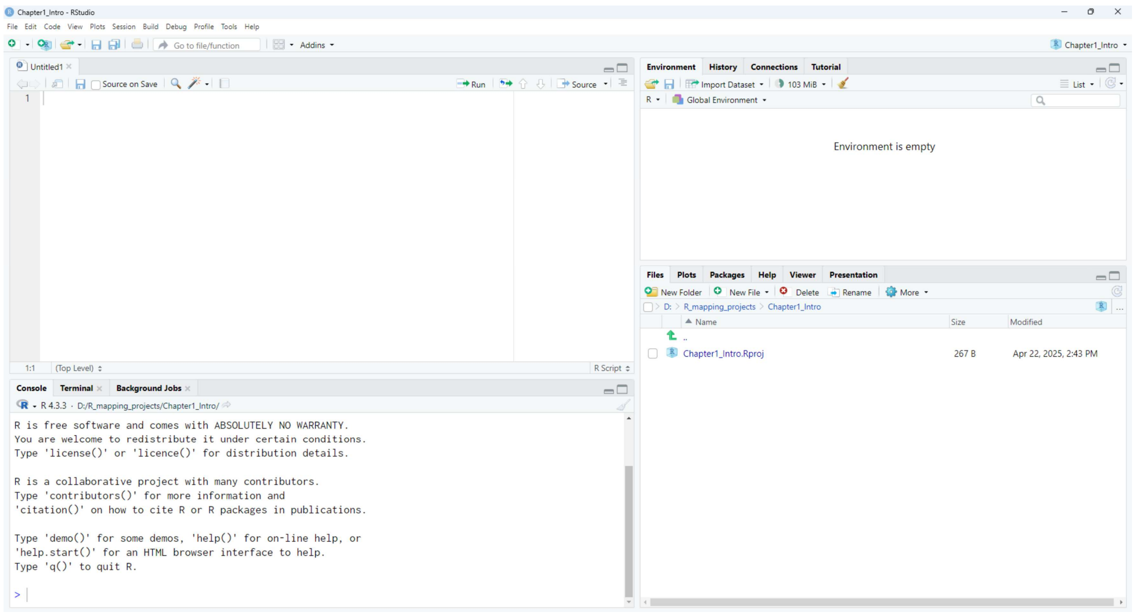
Why it's Awesome for Beginners (and Experts!):

- **No More Path Headaches:** You can load data files stored directly inside your project folder using just their simple filename (e.g., `read_csv("my_data.csv")`), because R automatically looks inside the project folder. If data is in a subfolder (e.g., `data/`), you use a simple relative path (`read_csv("data/my_data.csv")`).

- **Self-Contained & Portable:** You can move the entire project folder to a different place on your computer, or even share it with someone else, and the code using relative paths will still work!
- **Keeps Things Tidy:** Encourages you to keep all related files together.

Let's Create Your First Project:

1. Choose a Home: Decide where you'll keep your mapping projects (e.g., create a folder called `R_Mapping_Projects` inside your main Documents folder).
2. In RStudio: Go to the menu File -> New Project....
3. New Directory: Choose "New Directory" (since we're starting fresh).
4. New Project: Choose "New Project".
5. Name & Location:
   - **Directory name:** Give your project a meaningful name, like `Chapter1_Intro`. This will also be the name of the folder RStudio creates.
   - **Create project as subdirectory of:** Click "Browse..." and navigate to the `R_Mapping_Projects` folder you created in Step 1.
6. Create Project: Click the "Create Project" button.



```
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Figure 5: *RStudio Project Created*

Golden Rule: From now on, for every chapter or distinct mapping project you work on, start by creating a new RStudio Project! Save your scripts and data files inside that project folder (or organized subfolders within it). This will make your life much, much easier!

You can now re-save `my_first_script.R` inside this `Chapter1_Intro` project folder. And, btw, from now on, you will find all the scripts organized by chapter in this GitHub repo.

R Packages: Your Specialized Toolkits

R itself provides core functions, but its real power comes from **packages** (sometimes called libraries). Think of packages as specialized toolkits that other people have created and shared, adding new functions and capabilities to R.

Analogy: Base R is like a basic toolbox with a hammer, screwdriver, and wrench. Packages are like buying specialized toolsets – one for plumbing, one for electrical work, one specifically for building beautiful maps!

For mapping, we'll rely heavily on packages like:

- `sf`: For working with Simple Features (vector map data - points, lines, polygons). Absolutely essential!
- `terra`: For working with raster data (grids/pixels - like elevation or satellite images).
- `tidyverse`: Actually a collection of packages for general data science, including:
  - `ggplot2`: For creating amazing, layered plots and maps (we'll use this a LOT).
  - `dplyr`: For powerful data manipulation (filtering, selecting, joining - we've used it already!).
  - `readr`: For reading data files like CSVs.
- `geodata`, `rnaturalearth`, `osmdata`: Packages to help us easily download specific types of spatial data.
- `leaflet`, `rayshader`, `gganimate`: Packages for interactive maps, 3D maps, and animations (later chapters).

Installing vs. Loading Packages:

There's a crucial two-step process:

1. **Install:** You only need to install a package once per computer (or R installation). This downloads the package code from the internet (CRAN) and saves it in your R library. The command is `install.packages("package_name")`.
2. **Load:** You need to load a package every time you start a new R session (i.e., restart RStudio) if you want to use its functions in that session. Loading makes the package's functions available to you. The command is `library(package_name)`.

Making it Easy: The `pacman` Package

Remembering to install then load can be tedious. A helpful package called `pacman` (Package Management) simplifies this with its `p_load()` function. `p_load()` will automatically:

- Check if a package is installed.
- If not installed, it installs it (from CRAN).
- Then, it loads the package into your current session.

Let's Install and Load Our First Essential Packages using `pacman`:

Action: In a script within your `Chapter1_Intro` RStudio Project, type and run this code.

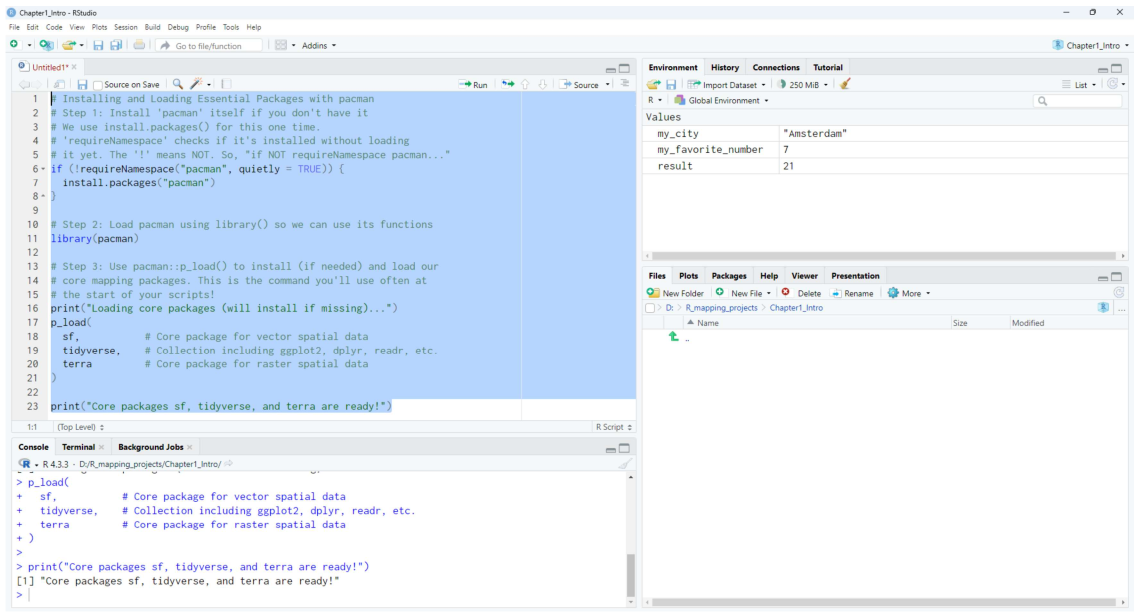
```r
# Installing and Loading Essential Packages with pacman
# Step 1: Install 'pacman' itself if you don't have it
# We use install.packages() for this one time.
# 'requireNamespace' checks if it's installed without loading
# it yet. The '!' means NOT. So, "if NOT requireNamespace pacman..."
if (!requireNamespace("pacman", quietly = TRUE)) {
  install.packages("pacman")
}

# Step 2: Load pacman using library() so we can use its functions
library(pacman)

# Step 3: Use pacman::p_load() to install (if needed) and load our
# core mapping packages. This is the command you'll use often at
# the start of your scripts!
print("Loading core packages (will install if missing)...")
p_load(
  sf,             # Core package for vector spatial data
  tidyverse,      # Collection including ggplot2, dplyr, readr, etc.
  terra           # Core package for raster spatial data
)

print("Core packages sf, tidyverse, and terra are ready!")

# You might see lots of messages if packages are being installed for
# the first time. This is normal! It might take a few minutes. You
# need an internet connection.
```


Chapter1_Intro - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Chapter1_Intro
Untitled1*
Source on Save
Run
Source
Environment History Connections Tutorial
Import Dataset
250 MiB
List
R
Global Environment
Files Plots Packages Help Viewer Presentation
New Folder New File Delete Rename More
D: R_mapping_projects Chapter1_Intro
Name Size Modified
1:1
(Top Level)
R Script
Console Terminal Background Jobs
R 4.3.3 · D:/R_mapping_projects/Chapter1_Intro/

```r
# Installing and Loading Essential Packages with pacman
# Step 1: Install 'pacman' itself if you don't have it
# We use install.packages() for this one time.
# 'requireNamespace' checks if it's installed without loading
# it yet. The '!' means NOT. So, "if NOT requireNamespace pacman..."
if (!requireNamespace("pacman", quietly = TRUE)) {
  install.packages("pacman")
}

# Step 2: Load pacman using library() so we can use its functions
library(pacman)

# Step 3: Use pacman::p_load() to install (if needed) and load our
# core mapping packages. This is the command you'll use often at
# the start of your scripts!
print("Loading core packages (will install if missing)...")
p_load(
  sf,             # Core package for vector spatial data
  tidyverse,      # Collection including ggplot2, dplyr, readr, etc.
  terra           # Core package for raster spatial data
)

print("Core packages sf, tidyverse, and terra are ready!")
```

```
Values
  my_city               "Amsterdam"
  my_favorite_number    7
  result                21
```

```
> p_load(
+   sf,             # Core package for vector spatial data
+   tidyverse,      # Collection including ggplot2, dplyr, readr, etc.
+   terra           # Core package for raster spatial data
+ )
>
> print("Core packages sf, tidyverse, and terra are ready!")
[1] "Core packages sf, tidyverse, and terra are ready!"
> 
```

Figure 6: *Package pacman in action*

What Just Happened?

- You installed the helper package pacman (if you didn't have it already).
- You loaded pacman using library().
- You used `pacman::p_load(sf, tidyverse, terra)` to ensure these three absolutely essential packages for spatial work are both installed and loaded into your current R session. R now has access to all the powerful functions within these toolkits!

Common Setup Hurdles:

- **Errors during installation:** Sometimes, spatial packages like sf or terra need extra system libraries (like GDAL, GEOS, PROJ). If you see errors mentioning these, check Appendix A or search online for "install gdal geos proj [your OS] for R sf".
- **Permission errors:** If R says it can't write to a directory, try running RStudio as Administrator (Windows, right-click icon) just for the installation, or agree if R asks to use a "personal library".
- **No internet:** `install.packages()` and `p_load()` (when installing) need an internet connection.

From now on, start most of your mapping scripts with `pacman::p_load(...)` listing the packages you need for that specific script!

Sneak Peek: Coordinate Reference Systems (CRS) - Why Maps Align

Okay, one last crucial concept before we wrap up Chapter 1. We'll dive deeper in Chapter 2, but it's vital to introduce this early because it solves a common beginner headache!

Imagine you have two maps of your city: one showing roads and one showing park boundaries. You try to plot them together in R, but they don't line up! The roads might be shifted miles away from the parks. Why?

They probably use different **Coordinate Reference Systems (CRS)**!

The Problem: The Earth is round(ish), but our computer screens and maps are flat. A CRS is a set of rules defining:

- How the Earth's shape is modeled (the Datum, e.g., WGS84 used by GPS).
- How locations from the curved model are projected onto a flat map (the Projection, e.g., Mercator, UTM, Albers).

Why it Matters: If two datasets use different rules (different CRSs), their coordinates mean different things, and they won't align geographically!

The Solution: We need to make sure all data we want to combine or plot together uses the same CRS. We often transform data from one CRS to another.

Checking CRS with sf:

The `sf` package (which you just loaded!) makes this easy. When you load spatial data into an sf object (we'll do this in Chapter 3), you can check its CRS rules.

Let's Try It (Quick 10-min Exercise):

We need some spatial data first. Let's quickly get the world map data we used before, but this time using sf functions directly.

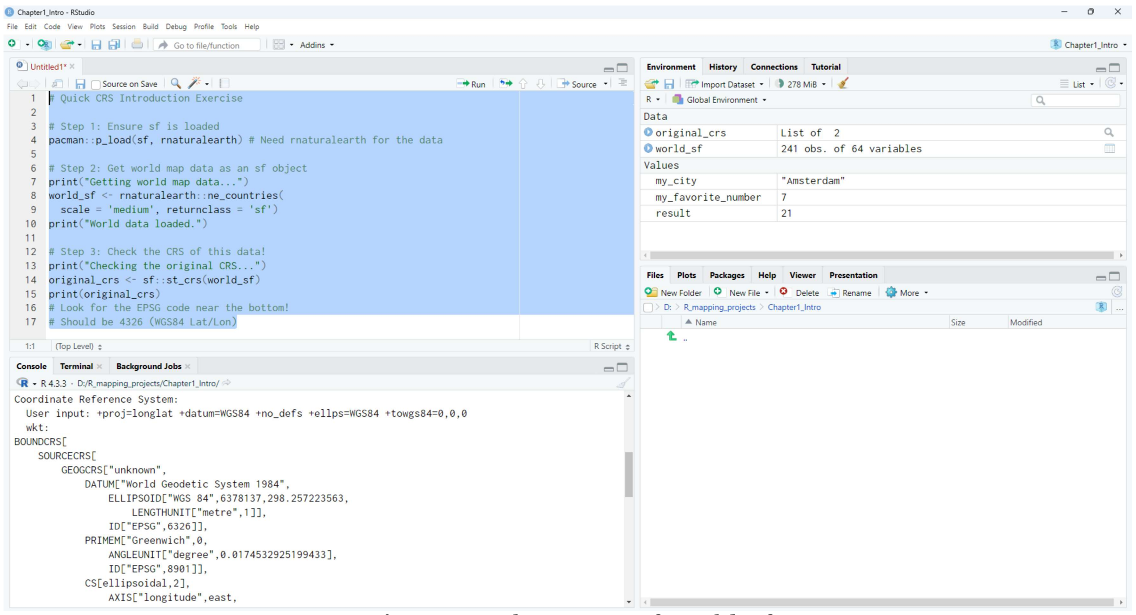
Action: Add this code to your script and run it.

```r
# Quick CRS Introduction Exercise

# Step 1: Ensure sf is loaded
pacman::p_load(sf, rnaturalearth) # Need rnaturalearth for the data

# Step 2: Get world map data as an sf object
print("Getting world map data...")
world_sf <- rnaturalearth::ne_countries(
  scale = 'medium', returnclass = 'sf')
print("World data loaded.")

# Step 3: Check the CRS of this data!
print("Checking the original CRS...")
original_crs <- sf::st_crs(world_sf)
print(original_crs)
# Look for the EPSG code near the bottom!
# Should be 4326 (WGS84 Lat/Lon)
```



```r
# Quick CRS Introduction Exercise

# Step 1: Ensure sf is loaded
pacman::p_load(sf, rnaturalearth) # Need rnaturalearth for the data

# Step 2: Get world map data as an sf object
print("Getting world map data...")
world_sf <- rnaturalearth::ne_countries(
  scale = 'medium', returnclass = 'sf')
print("World data loaded.")

# Step 3: Check the CRS of this data!
print("Checking the original CRS...")
original_crs <- sf::st_crs(world_sf)
print(original_crs)
# Look for the EPSG code near the bottom!
# Should be 4326 (WGS84 Lat/Lon)
```

```
Coordinate Reference System:
  User input: +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
  wkt:
BOUNDCRS[
    SOURCECRS[
        GEOGCRS["unknown",
            DATUM["World Geodetic System 1984",
                ELLIPSOID["WGS 84",6378137,298.257223563,
                    LENGTHUNIT["metre",1]],
                ID["EPSG",6326]],
            PRIMEM["Greenwich",0,
                ANGLEUNIT["degree",0.0174532925199433],
                ID["EPSG",8901]],
            CS[ellipsoidal,2],
                AXIS["longitude",east,
```

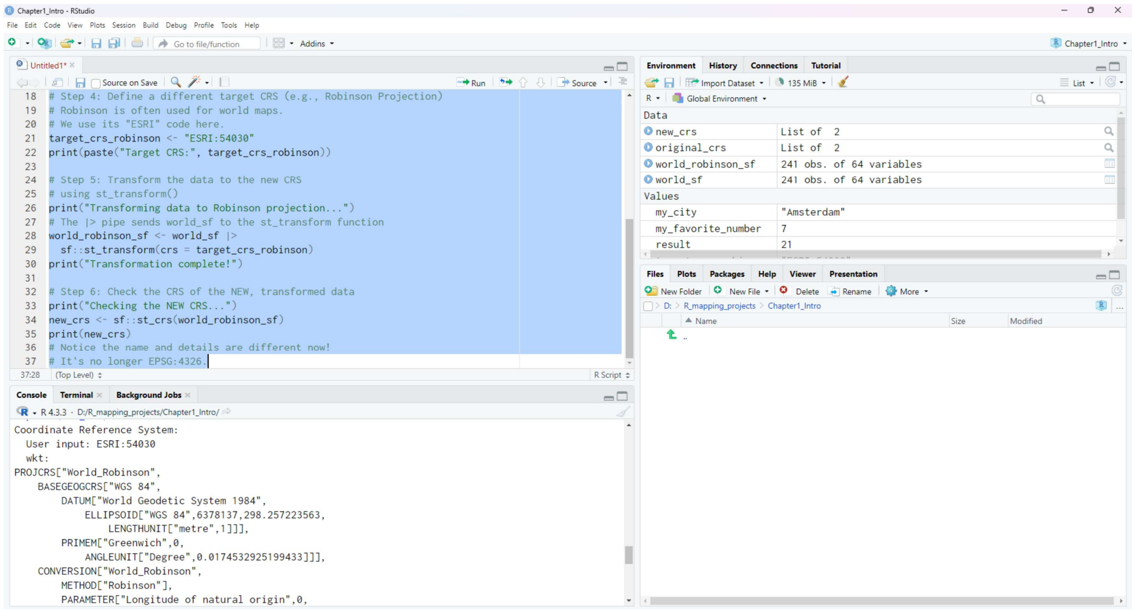
Data	
original_crs	List of 2
world_sf	241 obs. of 64 variables
Values	
my_city	"Amsterdam"
my_favorite_number	7
result	21

Figure 7: *The output of world_sf*

```r
# Step 4: Define a different target CRS (e.g., Robinson Projection)
# Robinson is often used for world maps.
# We use its "ESRI" code here.
target_crs_robinson <- "ESRI:54030"
print(paste("Target CRS:", target_crs_robinson))

# Step 5: Transform the data to the new CRS
# using st_transform()
print("Transforming data to Robinson projection...")
# The |> pipe sends world_sf to the st_transform function
world_robinson_sf <- world_sf |>
  sf::st_transform(crs = target_crs_robinson)
print("Transformation complete!")

# Step 6: Check the CRS of the NEW, transformed data
print("Checking the NEW CRS...")
new_crs <- sf::st_crs(world_robinson_sf)
print(new_crs)
# Notice the name and details are different now!
# It's no longer EPSG:4326.
```



Chapter1_Intro - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Chapter1_Intro
Untitled1*
Source on Save
Run
Source
37:28
(Top Level)
R Script
Environment History Connections Tutorial
Import Dataset
135 MiB
List
R
Global Environment
Files Plots Packages Help Viewer Presentation
New Folder New File Delete Rename More
D: > R_mapping_projects > Chapter1_Intro
Name Size Modified
Console Terminal Background Jobs
R 4.3.3 · D:/R_mapping_projects/Chapter1_Intro/

```r
# Step 4: Define a different target CRS (e.g., Robinson Projection)
# Robinson is often used for world maps.
# We use its "ESRI" code here.
target_crs_robinson <- "ESRI:54030"
print(paste("Target CRS:", target_crs_robinson))

# Step 5: Transform the data to the new CRS
# using st_transform()
print("Transforming data to Robinson projection...")
# The |> pipe sends world_sf to the st_transform function
world_robinson_sf <- world_sf |>
  sf::st_transform(crs = target_crs_robinson)
print("Transformation complete!")

# Step 6: Check the CRS of the NEW, transformed data
print("Checking the NEW CRS...")
new_crs <- sf::st_crs(world_robinson_sf)
print(new_crs)
# Notice the name and details are different now!
# It's no longer EPSG:4326.
```

```
Data
new_crs               List of  2
original_crs          List of  2
world_robinson_sf     241 obs. of 64 variables
world_sf              241 obs. of 64 variables
Values
  my_city             "Amsterdam"
  my_favorite_number  7
  result              21
```

```
Coordinate Reference System:
  User input: ESRI:54030
  wkt:
PROJCRS["World_Robinson",
    BASEGEOGCRS["WGS 84",
        DATUM["World Geodetic System 1984",
            ELLIPSOID["WGS 84",6378137,298.257223563,
                LENGTHUNIT["metre",1]]],
        PRIMEM["Greenwich",0,
            ANGLEUNIT["Degree",0.0174532925199433]]],
    CONVERSION["World_Robinson",
        METHOD["Robinson"],
        PARAMETER["Longitude of natural origin",0,
```

Figure 8: *The output of world_robinson_sf*

```r
# Step 7: Quick plot comparison
# (using base plot for simplicity here)
print(
  "Plotting comparison (Original vs. Transformed)..."
)
par(mfrow = c(1, 2)) # Arrange plots side-by-side
plot(
  sf::st_geometry(world_sf),
  main = "Original (EPSG:4326)",
  key.pos = NULL, reset = FALSE,
  border="grey"
)
plot(
  sf::st_geometry(world_robinson_sf),
  main = "Transformed (Robinson)",
  key.pos = NULL, reset = FALSE,
  border="blue"
)
par(mfrow = c(1, 1)) # Reset plot layout
print("Plots displayed. Notice the shape difference!")
```

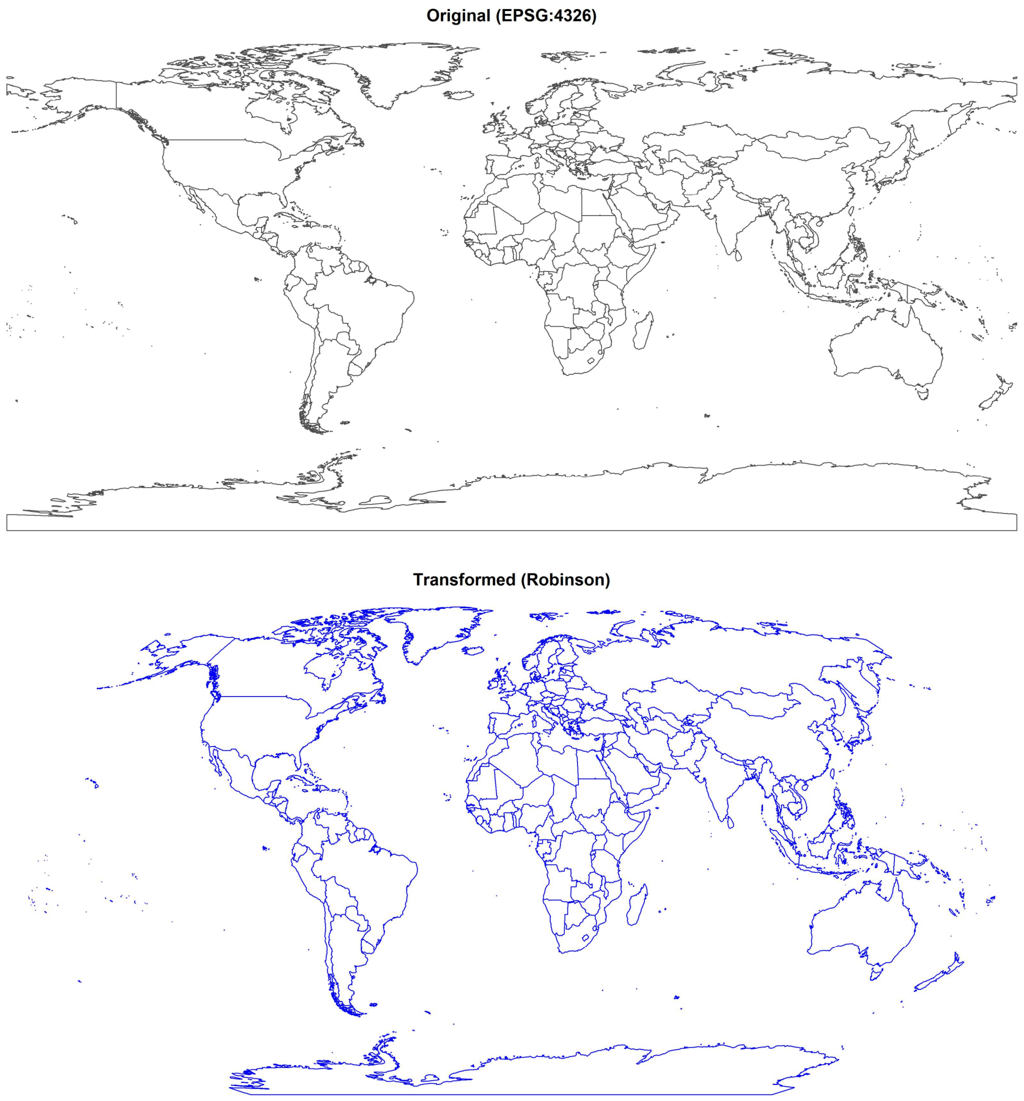


Figure 9: *One world, two projections*

What Just Happened?

- You loaded world map data into an sf object (`world_sf`).
- You used `sf::st_crs()` to check its Coordinate Reference System. The output showed lots of detail, but importantly included `ID["EPSG",4326]`, which is the code for the common WGS84 Latitude/Longitude system.
- You defined a different target CRS (`Robinson projection, ESRI:54030`).
- You used `sf::st_transform(crs = ...)` to convert your data to the new CRS, storing the result in `world_robinson_sf`. This function does the complex math to recalculate coordinates based on the new projection rules!
- You checked the CRS of the new object (`new_crs`) and saw it was indeed different.
- Plotting them side-by-side visually confirmed the change – the Robinson projection map looks different from the original Lat/Lon map!

Key Takeaway: Data needs to be in the same CRS to align correctly. `sf::st_crs()` checks the CRS, and `sf::st_transform()` changes it. We'll revisit this in detail in Chapter 2, but knowing these two functions early can save you a lot of confusion!

Chapter 1 Summary: Key Takeaways & Common Pitfalls

Fantastic work navigating your first full chapter! Let's recap the essentials:

Key Takeaways:

- **R vs. RStudio:** R is the engine, RStudio is the user-friendly dashboard/cockpit.
- **RStudio Panes:** You know the main areas: Script Editor (writing/saving code), Console (running code, seeing text output/errors), Environment (seeing variables R knows), Plots (seeing maps!).
- **Scripts (.R files):** The best way to write and save your R code for reproducibility. Use **File -> New File -> R Script**. Run code with Run/Source* buttons or Ctrl/Cmd+Enter. Save often!
- **Comments:** Use # to add notes to your code for explanation.
- **Variables & Assignment:** Store values (numbers, text) in named variables using the assignment arrow <- (e.g., `my_var <- 10`).
- **RStudio Projects (.Rproj):** The BEST way to organize your work! Automatically sets the working directory, making file paths easier and projects portable. Use `File -> New Project...`.
- **Packages:** Specialized toolkits extending R's capabilities (e.g., sf, tidyverse, terra). Install once (`install.packages()`), load each session (`library()`).
- **pacman::p_load():** A convenient function to install (if needed) AND load packages in one step. Use it at the start of your scripts!

- **CRS Basics:** Coordinate Reference Systems define how map data relates to the round Earth. Data needs the same CRS to align. `sf::st_crs()` checks CRS, `sf::st_transform()` changes it.

Common Pitfalls for Beginners (and Solutions):

Typos & Case Sensitivity: R needs commands and variable names spelled exactly right, including capitalization (my_var is different from My_Var). Double-check spelling!

- **Forgetting to Load Packages:** Seeing errors like `Error: could not find function "st_read"`? You probably forgot to load the package (sf in this case) using `library(sf)` or `pacman::p_load(sf)` at the start of your R session.
- **File Not Found Errors:** Usually means R can't find a file you asked it to load.
  - **Solution:** Are you using an RStudio Project? Is the file inside the project folder? Did you spell the filename correctly (including extension like .csv or .gpkg)? Avoid using `setwd()`!
- **Confusing <- (Assignment) and == (Comparison):** Use `<-` to store a value in a variable. Use == (double equals) later when you want to check if something is equal to something else (e.g., in a `filter()` command).
- **Forgetting Quotes Around Text:** Text values (like filenames or city names) need to be enclosed in double quotes ("like this") or single quotes ('like this'). Numbers don't need quotes (e.g., `my_number <- 10`).
- **Installation Problems:** Especially for spatial packages. See Appendix A or package documentation if `pacman::p_load()` gives errors during installation.

Review & Connect: What's Next?

You've successfully set up your mapping environment, learned how to interact with R via RStudio, handled basic commands and scripts, installed essential packages, and got your first glimpse of the crucial concept of Coordinate Reference Systems! You've built the launchpad.

In Chapter 2: Understanding Spatial Data in R, we'll dive deeper into the actual "ingredients" of our maps:

We'll properly explore the difference between vector data (points, lines, polygons - the world of `sf`) and raster data (grids - the world of `terra`).

We'll take a closer look at the structure of `sf` objects.

We'll solidify our understanding of Coordinate Reference Systems (CRS) and why projections are so fundamental to mapping.

We'll learn about common spatial data file formats you'll encounter (like Shapefiles, GeoJSON, GeoPackages).