An Amalgam of R

A Primer and Reference for Biologists

Ken Aho

2025-09-15

Contents

Preface

This book is contracted to Chapman & Hall/CRC, and will be officially published in 2026. It is currently a draft. Comments are welcome at GitHub or by email. The book is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

What this book is about

This book explores the ever expanding universe of **R**. Specifically, it considers:

- The historical development of the **R** language, the **R** engine, and the installation of **R** (Ch 1)
- **R** objects and the RStudio IDE (Ch 2)
- **R** data storage entities, and the import and export of user data files (Ch 3)
- Data management approaches using base **R** (Ch 4) and the *tidyverse* (Ch 5)
- **R** approaches to graphics, including base plotting methods (Ch 6) and the *ggplot2* package (Ch 7)
- **R** functions (Ch 8) including loops, and the creation of user-defined classes and generic methods
- Interfacing other languages (e.g., C, Fortran, C++, SQL, Python) and software (Ch 9)
- Building custom **R** packages (Ch 10)
- **R** GUIs and web applications including approaches from the packages *tcltk*, *plotly* and *shiny* (Ch 11)
- The fundamental ways that **R** interacts with your computer (Ch 12)

While this book covers a lot of ground, clearly many other topics could be considered. Subjects explored are those I have found to be particularly useful or interesting during my 20+ years of using **R** as a biologist and statistician. Chapters concerning advanced topics (i.e., Chs 8-12) are intended to be starting points for further exploration, and the reader is directed to additional resources when necessary.

This book emphasizes that **R** is an important programming language. While ignored in some (older) computer language histories (e.g., Boutin et al., 2002), **R** has had a large, devoted

following for decades and its computational engine and language can be clearly tied to core advances and important individuals in computer and data science. Further, from its inception **R** has been a tool for *metaprogramming* wherein code is shared and modified programmatically. For instance **R** has a wide variety of APIs for languages like C, Fortran, C++, Java, Python, and many others.

Individuals from the natural sciences, particularly biologists, are likely to find this book more useful than individuals from other backgrounds because coding examples and applications are generally biological. Non-biologists may find, however, that examples readily extend to other settings.

What this book is not about

Notably, although statistics is the primary focus/purpose of **R**, the primary emphasis of this book is *not* statistics. Instead I focus on the **R** language, and the characteristics, capabilities, and extensions of the **R** system. I take this approach because: 1) coverage of non-statistical topics is challenging in and of itself, and 2) the responsible introduction of statistical algorithms from any program or language (including **R**) should be accompanied by detailed information concerning the statistical procedures. Many pedagogic resources exist for the statistical application of **R**. These include: Aho (2014) (the pedagogic statistical companion to this book), Venables and Ripley (2002), (Faraway, 2004, 2016), Crawley (2012), and Fox and Weisberg (2019), among others. It should be noted that while this text does not focus on inferential statistical methods, it *does* emphasize methods for handling, summarizing and displaying empirical data, and these steps generally serve as a companion and prerequisite to formal descriptive and inferential analyses.

Distinguishing Characteristics of This Book

Many other sources have emphasized programming and data science aspects of **R**, while largely ignoring statistics, including definitive texts (e.g., Chambers, 2008, 2020; Wickham, 2016, 2021), and manuals (R Core Team, 2024a,b,c), or have focused on particular, non-statistical **R** components, including graphics (Wickham, 2016; Murrell, 2019) and web-based applications (Wickham, 2021; Sievert, 2020). This book is a brave/foolish attempt to *amalgamize* and distill what I consider important information from this collection, while occasionally emphasizing topics earlier works have ignored. For instance, Wickham (2019) admirably emphasizes many foundational and advanced programming ideas in **R**, but does not thoroughly consider some important programming extensions, including powerful syntheses with Python, Tcl, and Qt via C++. Unlike many other texts, this book also adheres to the format of a textbook, with numerous worked examples, and exercises at the end each chapter.

Conventions

This document has been created with Windows users of **R** in mind. I chose Windows as a demonstration operating system (OS) because it is currently the most widely used system manager for desktop computers and laptops by a wide margin (Wikipedia, 2025d)[1]. However, in the vast majority of cases, **R** instructions and examples provided herein will be extendable to other operating systems. In cases when this is not true I note steps to address those inconsistencies. A notable exception occurs for shell-driven creation of executable files in Ch 9. In this case, Windows command line (cmd.exe) processes will often differ fundamentally from Unix-alikes and require workarounds[2].

Several text-formatting conventions are followed throughout the book. **R** package names and important terms are *italicized*. Function names, function arguments, base types, and objects are written in blocked `Courier font`. Functions obtained from packages have their names followed by parentheses, e.g., `function.name()`. Operations from **R** and other programming languages are generally written into "chunks" whose contents can be copied to a clipboard using an icon located at the top right of the chunk (HTML versions of book only). For example:

```
print("Hello, world!")
```

The *output* from an evaluated chunk is generally printed immediately below. For example:

```
[1] "Hello, world!"
```

If you are reading an HTML version of this document generated using `bs4_book()`, then **R** function names will generally be hyperlinked to their documentation. For example: `print()` would be hyperlinked. In this case, footnotes will be inline, potentially requiring that links within footnotes be accessed using Enter + **Click**.

Acknowledgements and Corrections

I thank individuals who have reviewed/edited this book in various forms including Lauren Tucker and Adam Zambie. Corrections and comments are welcome, and can be sent using the book's GitHub site.

[1] On the other hand, Linux is the most common OS for web-servers, supercomputers, and smartphones (via the Android OS, which uses the Linux kernel) (Wikipedia, 2025d).

[2] Specific consideration of Unix/Linux implementations of **R** in high performance computing are provided in Ch 12.

Chapter 1

Welcome to R

> *"I believe that* R *currently represents the best medium for quality software in support of data analysis."*
>
> - **John Chambers**, *Developer of* S

> *"*R *is a real demonstration of the power of collaboration, and I don't think you could construct something like this any other way."*
>
> - **Ross Ihaka**, *original co-developer of* R

1.1 What is R?

R is a computer language and an open source setting for statistics, data management, computation, and graphics. The outward mien of the **R**-environment is minimalistic, with few menu-driven interactive facilities (*no* menus exist for some implementations of **R**). This is in contrast to conventional statistical software consisting of black box, menu-dominated, often inflexible tools. The simplicity of **R** allows one to easily evaluate, edit, and build procedures for data analysis, and many other purposes.

1.2 R and Biology

I am a statistical ecologist, so this book was written with natural scientists, particularly biologists, in mind. **R** is useful to biologists for three major reasons. First, it provides access to a large number of cutting edge statistical, graphical, and organizational procedures, many of which have been designed specifically for biological research. Second, biological datasets, including those from genetic and spatiotemporal research can be extensive and complex. **R** can readily manage and analyze such data. Third, analysis of biological data often requires analytical and computational flexibility. **R** allows one to "get under the hood", look at the code, and check to see what algorithms are doing. If, after examining an **R**-algorithm we are unsatisfied, we can generally modify its code or create new code to meet our specific needs.

1.3 Popularity of R

Because of its freeware status, the overall number of people using **R** is difficult to determine. Nonetheless, the **R**-consortium website estimates that there are currently more than two million active **R** users. The r4stats website houses up-to-date surveys concerning the popularity of analytical software. These surveys (accessed 10/23/2024) indicate that **R** is often preferred among data scientists for big data projects and data mining. **R** is also one of the most frequently cited statistical environments in scholarly articles, one of the most frequently used languages on the GitHub repository, and one of the most frequently discussed languages on Stack Overflow. In 2024 the **R** language was ranked 20th in the world by the Institute of Electrical and Electronics Engineers (IEEE), and was recently (10/23/2024) ranked 6th by the PopularitY of Programming Language (PYPL) index[1]. Further, in a 2017 survey, based on Stack Overflow queries, **R** was the "least disliked" programming language. The growth and popularity of **R** can be partially tied to its relatively straightforward extendability via user generated functions and packages. This characteristic prompts a strong sense of community among **R**-users, along with a practical need for the perpetuation and upkeep of the **R** system. While trailing Python, there are currently over 20000 formally contributed **R**-packages at the Comprehensive **R** Archive Network (CRAN).

1.4 A Brief History

R was created in the early 1990s by Australian computational statisticians Ross Ihaka and Robert Gentleman (Fig 1.1) to address *scope*[2] and memory use deficiencies in its primary progenitor language, **S** (Ihaka and Gentleman, 1996). Ihaka and Gentleman used the name **R** both to acknowledge the influence of **S** (because *r* and *s* are juxtaposed in the alphabet), and to celebrate their own personal efforts (because R was the first letter of their first names).

Figure 1.1: Ross Ihaka (1954 -) (L) and Robert Gentleman (1959 -) (R), the co-creators of **R**.

At the insistence of Swiss statistician Martin Maechler (Fig 1.2l), Ihaka and Gentleman distributed the **R** source code in 1995 under the Free Software Foundation's GNU general license (Ihaka, 1998). Because of its relatively easy-to-learn language, **R** was quickly extended with

[1]The PYPL index uses the search string `'X tutorial'`, as an indicator of future language usage

[2]In computer science, *scope* refers to the degree of binding between an identifier of an entity (e.g., an object name) and the entity itself (e.g., an object).

code and packages developed by its users. The rapid growth of **R** gave rise to the need for a group to guide its progress. This led, in 1997, to the establishment of the **R**-development core team[3], an international panel that modifies, troubleshoots, and manages source code (Fig 1.2).

Figure 1.2: A recent version of the **R**-core development team.

1.4.1 Development of the R Language

The **R** language is based on older languages, particularly **S**, developed at Bell Laboratories (Becker and Chambers, 1978, 1981; Becker et al., 1988), and Lisp[4] (McCarthy, 1978) and Scheme, a dialect of Lisp (Sussman and Steele Jr, 1998; Steele, 1978), which were developed at

[3]The first **R**-core consisted of: Douglas Bates, Peter Dalgaard, Robert Gentleman, Kurt Hornik, Ross Ihaka, Friedrich Leisch, Thomas Lumley, Martin Mächler, Paul Murrell, Heiner Schwarte, and Luke Tierney. Several of these individuals remain in the current **R**-core (Fig 1.2).

[4]Lisp, an abbreviation of "LISt Processor", is the second-oldest (after Fortran) high-level programming language still in common use. Lisp can be viewed as a related family of dialects rooted in MaCarthy's general programming approach, which included fully parenthesized prefix notation (wherein a function `f` with three arguments would be called using: `(f arg1 arg2 arg3))` Reilly (2003). Seminal Lisp dialects include Common Lisp, Scheme, and AutoLISP (built for the drafting software AutoCAD). Recent Lisp additions include: Hy (a Lisp dialect embedded in Python), Clojure (a dialect of Lisp for Java), and Lisp Flavored Erlang (LFE). Many important programming ideas were pioneered by Lisp, including conditional statements based on Boolean decision rules, higher-order functions (which contain other functions as arguments), recursion (which allows a function to be called by its own code), and the read–eval–print loop.

the MIT artificial intelligence laboratory in the late 1970s (Fig 1.3).

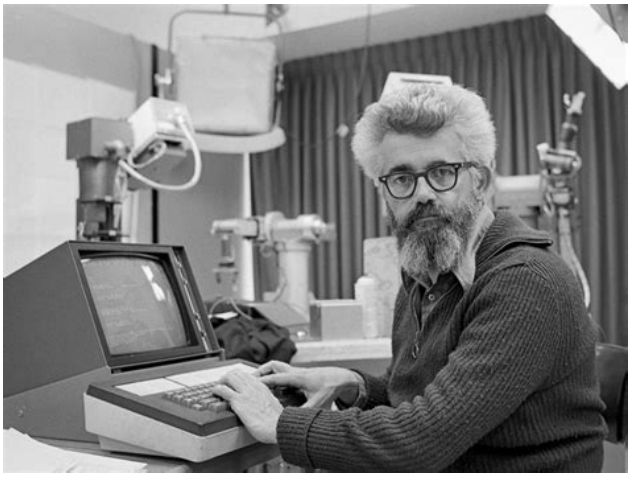


Figure 1.3: John McCarthy (1927-2011), creator of the Lisp language, and the first to coin the term *artifical intelligence*, working at the MIT AI laboratory.

In the appendix to his book *Software for Data Analysis*, John M. Chambers (Fig 1.2b), a primary developer of **S**, recounts the unique evolution and goals of **S** from its inception in 1976. Chambers notes that **S** was originally intended to be an analysis toolbox solely for the statistics research group at Bell Labs, consisting of roughly 20 people at the time. It was decided that **S** (initially known as "the system"[5]) would have fundamental *extensibility*[6], reflecting the Bell Labs' philosophy that "collaborations could actually enhance research" (Chambers, 2008)[7]. The **S** language definition, and details concerning the fitting and application of **S** statistical models are given in Becker et al. (1988) and Chambers and Hastie (1992), respectively[8].
S was designed to diminish inner functional details of its underlying C and Fortran[9] algorithms while making important higher-level processes more readily accessible and interactive. The inspiration for these goals was the exploratory data analysis approach of John Tukey (Fig 1.4), who was a contemporary of Chambers and other **S** developers[10] at Bell Labs (Chambers, 2020).

[5]The name **S** arose because all of the Bell Lab naming suggestions contained "S", and the name for the recently designed language C was a single letter (Chambers, 2008).

[6]In software engineering, *extensibility* is a design principle that allows for future growth. This allows developers to easily expand the software capabilities.

[7]Notably, although **S** was originally designed to support statistical analysis, Chambers (2020) asserted that its actual usage at Bell Labs would be viewed today as *data science*; defined as "techniques and their application to derive and communicate scientifically valid inferences and predictions based on relevant data."

[8]Becker et al. (1988) described the third version of **S**, **S3**. Chambers and Hastie (1992) introduced formula-notation using the ~ operator, dataframe objects, and modifications to object methods and classes (Wikipedia, 2024i). These publications were often referred to as the *blue book* and the *white book* by **S**-users, due to color of their covers.

[9]Fortran (FORmula TRANslator) is a computer language developed by IBM in the 1950s for science and engineering applications. Remarkably, it remains useful for many applications, including speeding up slow looping routines in interpreted languages like **R**. Fortran/**R** interfacing is formally considered in Ch 9 (Section 9.2)

[10]Other important contributors to **S** included Rick Becker, Trevor Hastie, William Cleveland, and Allan Wilks of

In a 1965 Bell Labs memo (15 years before the release of **S**) Tukey noted that modern statistician found themselves in a "peaceful collision of computing and data analysis" (Chambers, 1999).

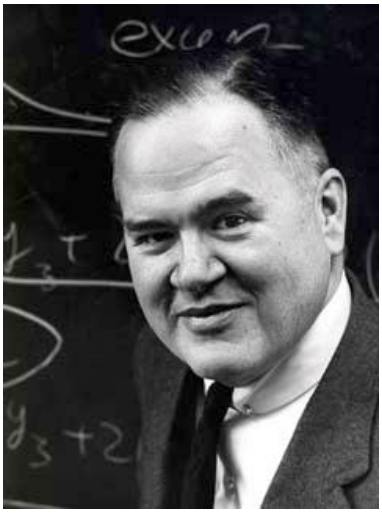


Figure 1.4: John Tukey (1915-2000), widely known for achievements in mathematical statistics, including the fast Fourier transform (Cooley and Tukey, 1965), tools in exploratory data analysis, including the boxplot (Tukey et al., 1977), and computer science, where he coined the term *bit*, as a unit of binary infomration and memory (Shannon, 1948).

An adherence of **S** to exploratory data analysis was evident in the high-quality and flexibility of its graphics devices and its easily-accessible function documentation (built-in documentation does not exist for many important languages like C and C++). The initial programmatic objectives of **S** are apparent in an early design sketch that describes an outer 'user interface' layer to core Fortran algorithms that ultimately produces an **S** object (Fig 1.5). The underlying philosophical principles and programmatic foundations of **S** have strongly affected and guided the development of **R** (Chambers, 2020).

S evolved alongside the Unix operating system (also developed at Bell Labs) which currently underlies Macintosh and Linux (free-Unix) operating systems[11]. An early inception **S** was written for Unix, allowing **S** to be *portable* to any machine using Unix. Both **S** and Unix were quickly commercially licensed by AT&T for university and third party retailers. The academic licensing and distribution of **S** attracted a large number user groups in 1980s. However, the lack of a clear open source strategy caused many early users to switch from **S** to **R** in the 1990s. **S** was purchased by Insightful® software 2004 to run the commercial software **S**-Plus®. In 2021 **S**-Plus® morphed to include TIBCO connected intelligence software, with some **R** open source applications.

Bell Labs.

[11] Unix itself was originally written in *assembly language* (a low-level programming language with a very strong correspondence between language instructions and machine/operating system instructions). Unix was later re-written in C.

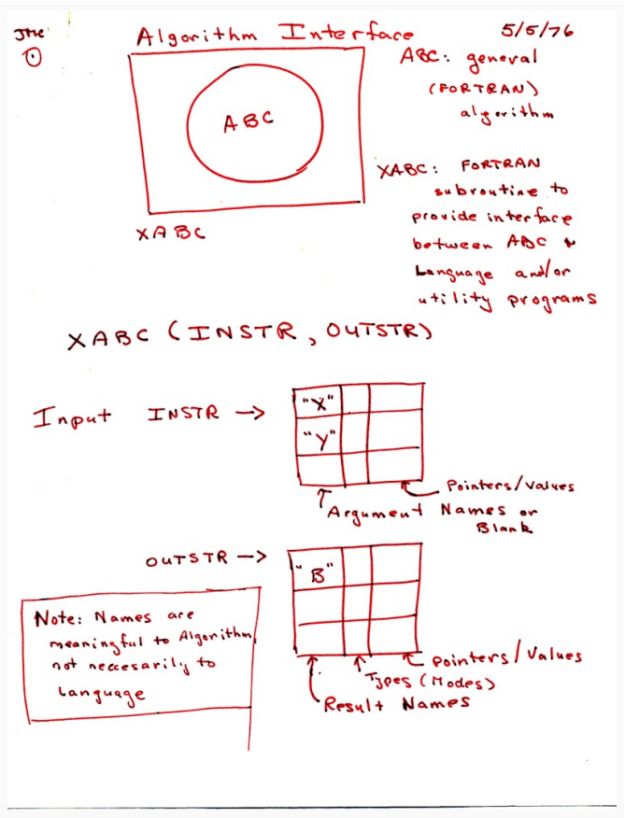


Figure 1.5: First designs for the S statistical system, *circa* 1976 (Chambers, 2008)). Written in the lower lefthand corner is the important note: 'Names are meaningful to algorithm, not necessarily to language.'

1.4.1.1 R is Born

The original Scheme-inspired **R** interpreter consisted of roughly 1000 lines of C[12] code, driven by a command line interface that used a syntax corresponding to **S**, resulting in "a free implementation of something 'close to' version 3 of the **S** language (**S3**)" (Ihaka, 1998). The **R** and **S** languages remain very similar, and code written in **S** can generally be run unaltered in **R**. The method of function implementation in **R**, however, remains more similar to Scheme. The official language definition of the current version of **R** can be found at the CRAN website, along with other sources of complementary information.

1.4.1.2 Differences of R and S

S3 and the initial release of **R** differed in two important ways (Ihaka and Gentleman, 1996)[13]. First, the **R**-environment was given a fixed amount of memory at start up. This was in contrast to **S**-engines which adjusted available memory to session needs. Among other things, this difference meant more available pre-reserved computer memory, and fewer *virtual pagination*[14] problems in **R** (Ihaka and Gentleman, 1996). It also made **R** faster than **S** for many applications (Hornik and the R Core Team, 2023). Second, **R** variable locations are *lexically scoped*. In computer science, *variables* are storage areas with identifiers, and *scope* defines the context in which a variable name is recognized. So-called *global variables* are accessible in every scope (for instance, both inside and outside functions). In contrast, *local variables* may only exist within particular localized scopes[15]. Lexical scoping allows functions in **R** access to variables that were in effect when the function was defined in a session. The characteristics of **R** functions and details concerning lexical scoping are further addressed in Ch 8.

1.4.2 Recent Developments

According to Thieme (2018), a growing component of the **R** culture includes individuals who are "Less interested in the mechanics of **R** than in what **R** allowed them to do." This group, which often includes individuals from non-**R** backgrounds (but with expertise in other languages including C, Java, CSS and HTML), and those "who may have little interest in becoming computer scientists", has been championed by Hadley Wickham (Fig 1.6), creator of the important *ggplot2* and *dplyr* **R** packages, and author of many useful books on **R** programming. A larger collection of packages supported by Wickham is referred to as the *tidyverse* (Wickham et al., 2019) (see Ch 5).

[12] C is a portable, general purpose language, initially developed by Dennis Ritchie (Kernighan and Ritchie, 2002). C, in turn, evolved from the language B, created by Ken Thompson (Thompson, 1972), which, in turn, was inspired by work on early operating system called Multics (Corbató and Vyssotsky, 1965). C/**R** interfacing is formally considered in Ch 9 (Section 9.2).

[13] For additional demonstrations of the technical differences of **R** and **S** see (Hornik and the R Core Team, 2023)

[14] *Virtual pagination* is a memory management scheme that allows a computer to store and retrieve data from secondary storage for use in main memory.

[15] Formal parameters defined in **R** functions, including arguments, are (generally) local variables, whereas variables created outside of functions are global variables (Sections 2.3.3, 8.2).

Figure 1.6: Hadley Wickham (1979 -) chief scientist at Rstudio.

1.4.3 The Future of R

It is apparent that **R** can be tied (particularly via linkages with Fortran and Lisp) to early examples of software engineering, and (via John Tukey and others) to foundational figures in data science. The future of **R** will be determined by the formal and informal community of users who have donated years of their lives to its development without monetary compensation. Importantly, the continued growth of **R** will require adaptation to the changing demography of **R**-users. Like most software endeavors, **R** has been male dominated (Fig 1.2). However, this has been changing rapidly. As an example, the **R** Ladies group, founded in 2012 by Gabriela de Queiroz (Fig 1.7), currently (8/6/2024) has 225 chapters in 65 countries, and more than 39,000 members worldwide.

Figure 1.7: Gabriela de Queiroz, chief data scientist at IBM.

1.5 Copyrights and Licenses

R is intentionally open-source and free. Thus, there are no warranties on its environment or packages. As its copyright framework **R** uses the GNU (a recursive acronym for GNU is not Unix)

General Public License (GPL). This allows users to share and change **R** and its functions. The associated legalese can read after typing `RShowDoc("COPYING")` in the **R**-console. Because its functions can be legally (and easily) recycled and altered we should always give credit to developers, package maintainers, or whomever wrote the **R** functions or code we are using.

1.6 R and Reliability

The lack of an **R** warranty has frightened away some scientists. But be assured, with few exceptions, **R** works as well or better than "top of the line" analytical commercial software. Indeed, statistical software giants SAS® and SPSS® have made **R** applications accessible from within their products (Fox, 2009), and **R** processes and files can be shared directly with Microsoft Excel® and other proprietary software. For specialized or advanced statistical techniques **R** often outperforms other alternatives because of its diverse array of continually updated applications.

The computing engines and packages that come with a conventional **R** download (see Section 3.7) meet or exceed U.S. federal analytical standards for clinical trial research (Schwartz et al., 2008). In addition, core algorithms used in **R** are based on seminal and well-trusted functions. For instance, **R** random number generators include some of the most venerated and thoroughly tested functions in computer history (Chambers, 2008). Similarly, the Linear Algebra PACKage (LAPACK) algorithms (Anderson et al., 1999), used by **R**, are among the world's most stable and best-tested software.

1.7 Installation

To install **R**, first go to the website (http://www.r-project.org/). On this page specify which platform you are using (Fig 1.8, step 1). **R** can currently be used on Unix/Linux, Windows and Mac operating systems. Once an operating system has been selected, one can click on the "base" link to download the precompiled base binaries if **R** currently exists on your computer. If **R** has not been previously installed on your computer click on "Install **R** for the first time" (Fig 1.8, step 2). You will delivered to a window containing a link to download the most recent version of **R** by clicking on the "Download" link (Fig 1.8, step 3). Two versions of **R** are generally released each year, one in April and one in October. Archived, older versions of **R** and **R** packages are also available from CRAN.

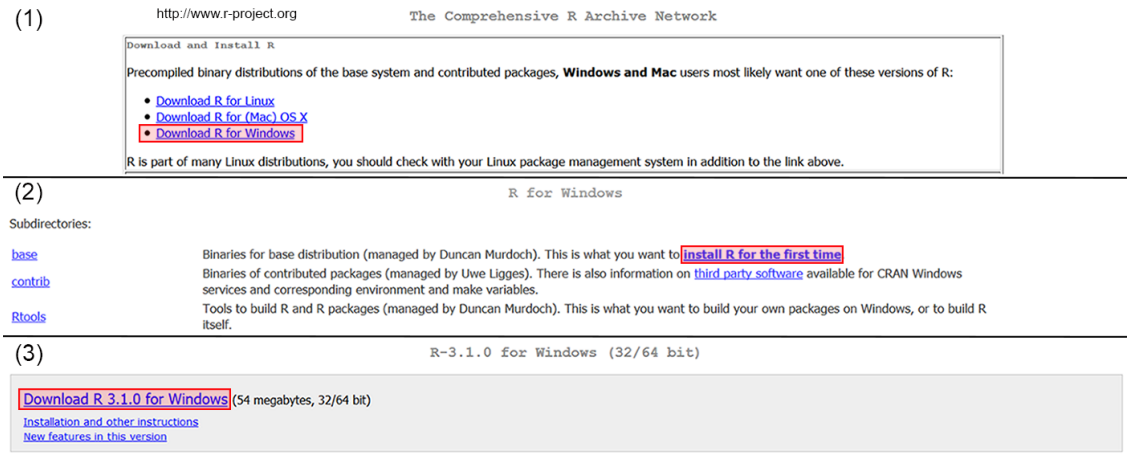


Figure 1.8: Method for installing **R** for Windows for the first time.

Exercises

1. The following questions concern the history and general characteristics of **R**.
   (a) Who were the creators of **R**?
   (b) What are some major developmental events in the history of **R**?
   (c) What languages is **R** derived from and/or most similar to?
   (d) What features distinguish **R** from other languages and statistical software?
   (e) What are the three operating systems **R** works with?
2. Briefly consider **R** in the context of major historical events in computer software and artificial intelligence.

Chapter 2

Some Basics

"Learning to write programs stretches your mind, and helps you think better."

- Bill Gates, *1955-*

2.1 First Steps

Upon opening **R** in Windows, two things will appear in the console of the **R** *Graphical User Interface* (**R**-GUI)[1]. These are the *license disclaimer* (blue text at the top of the console) and the *command line prompt*, i.e., > (Fig 2.1). The prompt indicates that **R** is ready for a command. All commands in **R** must begin at >.

The default appearance of the **R**-GUI will vary slightly among operating systems. In Windows, the command line prompt and user commands are colored red (Fig 2.1), and output, including errors and warnings, are colored blue. In Mac OS, the command line prompt will be purple, user inputs will be blue, and output will be black. In Unix/Linux, wherein **R** will generally run from a shell command line, absent of any menus, all three will be black[2]. These console defaults can often be modified/customized when using **R** with an appropriate *Integrated Development Environment* (IDE) like RStudio (Section 2.8.3).

We can exit **R** at any time by typing `q()` in the console, closing the GUI window (non-Linux only), or by selecting **Exit** from the pulldown File menu (non-Linux only).

[1] Unix/Linux operating systems require **R** to be launched from the shell command line by typing: `R`. This will begin an interactive **R** session on the system shell command line itself.

[2] A Unix/Linux GUI, similar to those in Windows and Mac OS, can be initiated by opening **R** with the commands: `R -g Tk &`.

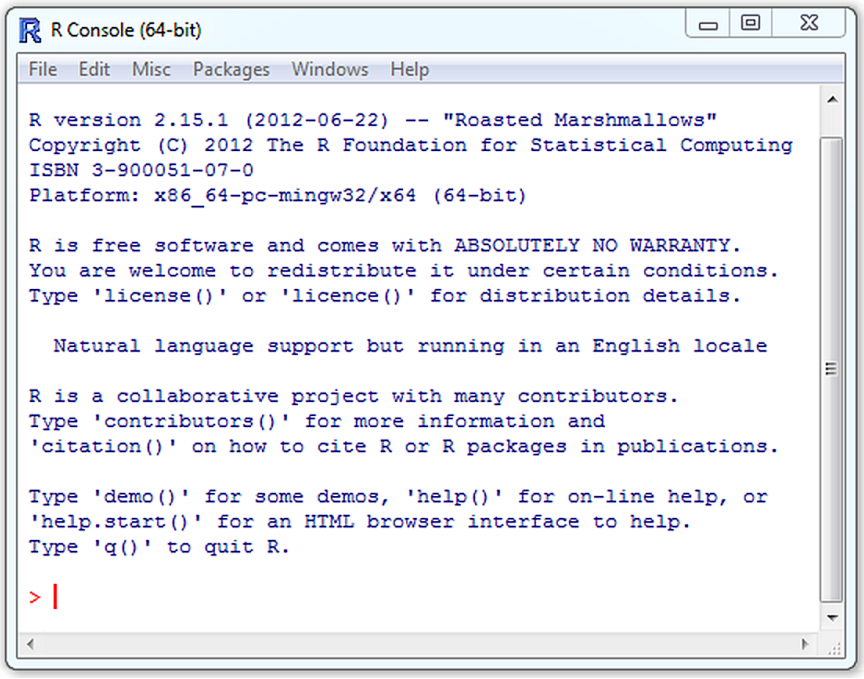


Figure 2.1: An aged, but still recognizable **R** console: **R** version 2.15.1, 'Roasted Marshmallows', ca. 2012.

2.2 First Operations

As an introduction we can use **R** to evaluate a simple mathematical expression. Type `2 + 2` and press Enter.

```
2 + 2
```

```
[1] 4
```

The output term `[1]` means, "this is the first requested element." In this case there is just one requested element, 4, the solution to $2 + 2$. If the output elements cannot be held on a single console line, then **R** would begin the second line of output with the element number comprising the first element of the new line. For instance, the command `rnorm(20)` will take 20 pseudo-random samples (see footnote in Section 9.5.11) from a standard normal distribution (see Ch 3 in Aho (2014)). We have:

```
rnorm(20)
```

```
 [1] -0.655139028 -0.321299117 -0.603446236 -0.192918230  0.008363708
 [6] -0.438218357 -0.014049221 -0.864694514 -0.301874922 -1.479418435
[11] -1.099509184  2.405779052  0.567586291 -1.184045493 -0.916286321
[16] -0.652781546  0.893402898  0.740420857 -0.183644845 -0.003615715
```

The reappearance of the command line prompt indicates that **R** is ready for another command.

Multiple commands can be entered on a single line, separated by semicolons. Note, however, that this is considered poor programming style, as it may make your code more difficult to understand by a third party.

```
2 + 2; 3 + 2
```

```
[1] 4
```

```
[1] 5
```

R commands are generally insensitive to white spaces, including tabs. This allows the use of spaces to make code more legible. To my eyes, the command `2 + 2` is simply easier to read (and potentially debug) than `2+2`.

2.2.1 Use Your Scroll Keys

As with many other command line environments, the scroll keys (Fig 2.2) provide an important shortcut in **R**. Instead of editing a line of code by tediously mouse-searching for an earlier command to copy, paste and then modify, you can simply scroll back through your earlier work using the upper scroll key, i.e., ↑ . Accordingly, scrolling down using ↓ will allow you to move forward through earlier commands.

Figure 2.2: Typical scroll direction keys on a keyboard.

2.2.2 Note to Self:

R will not recognize commands preceded by **#**. As a result this is a good way for us to leave messages to ourselves.

```
# Note at beginning of line
2 + 2
```

```
[1] 4
```

We can even place comments in the middle of an expression, as long the expression is finished on a new line.

```
2 + # Note in middle of line
+ 2
```

```
[1] 4
```

In the "best" code writing style it is recommended that one place a space after # before beginning a comment, and to insert two spaces following code before placing # in the middle of a line. This convention is followed above.

2.2.3 Unfinished Commands

R will be unable to move on to a new task when a command line is unfinished. For example, type

```
2 +
```

and press Enter. We note that the *continuation prompt*, +, is now where the command prompt should be. **R** is telling us the command is unfinished. We can get back to the command prompt by finishing the function, clicking **Misc**>**Stop current computation** or **Misc**>**Stop all computations** from the **R**-toolbar (non-Linux only), typing Ctrl + C (Linux), or by pressing the Esc key (all OS).

2.3 Expressions and Assignments

All entries in **R** are either *expressions* or *assignments*. If an entry is an *expression*, it will be evaluated, printed, and discarded. Examples include: `2 + 2`. Conversely, an *assignment* evaluates an expression, and binds the expression output to a name, thereby creating a referable **R**-*object*. This important activity has prompted the motto: "everything created or loaded in **R** is an *object*"[3].

To create an object, we use the *assignment operator*: `<-` . The operator represents an arrow that points toward it's user-defined name.

Example 2.1.
To create an **R**-object named `y`, that contains the result of the expression `2 + 2`, I can type:

```
y <- 2 + 2
```

The code: `y <- 2 + 2` literally means: "$2 + 2$ is bound to the name `y`" (Wickham, 2019).

The assignment operator can go on either side of an expression. Thus, as an alternative, I could have typed:

[3]Although everything created or loaded in **R** can be viewed as an object, not all **R** objects fit neatly into the object oriented programming (OOP) perspective of "object-oriented." This is true because **R** *base* objects (which are *not* object oriented) come from **S**, which was developed before anyone considered the need for an **S** OOP system (see Wickham (2019) and Chambers (2008)).

```
2 + 2 -> y
```

The leftward assignment operator, <-, is generally used instead of the rightward, ->, because it is easier to conceptualize the relationship `object name <- object`.

■

Results of an assignment are generally not automatically printed. However, for most common object classes (see Section 2.3.5) summaries can be easily obtained[4].

Example 2.2.
To print the result of Example 2.1 (to *see* the object bound to the name `y`), I can simply type:

```
y
```

```
[1] 4
```

or

```
print(y)
```

```
[1] 4
```

■

The mathematical equals operator, =, can also be used as an assignment operator. Like <-, = assigns from right to left.

Example 2.3.
For instance, to obtain the assignment result shown in Example 2.1, I could have typed:

```
y = 2 + 2
y
```

```
[1] 4
```

■

Notably, the equals sign has limited applicability as an assignment operator, compared to <-[5]. Thus, in this document, I use <- for object assignments, and save = for specifying arguments in **R** functions.

R objects need not be numeric. In computer programming, a *character string* or *string* is a collective sequence of characters representing text[6]. Character strings in **R** are delimited with quotes: `" "` or `' '`.

[4] *via* class `print()` or `summary()` functions (see Sections 8.7 and 8.8).
[5] See `?"<-"`.
[6] See Wikipedia (2024j) for additional information.

Example 2.4.
Here I define `y` to be a well-known character string.

```
y <- "Hello, world!"
```

```
y
```

```
[1] "Hello, world!"
```

■

2.3.1 Functions and their Arguments

Importantly, the script `print(y)`, in Example 2.2 provides one of our first clear uses of a special type of **R** object called a *function*. **R** functions generally require a user to specify *arguments* –that parameterize and control the function– within parentheses, following the function name. Thus, we use the following scripting framework to call an **R** function: `function.name(argument1, argument2, argument3, etc)`. The function `print()` only *requires* one argument: the name of the object to be printed.

A list of function arguments, and their default values, can (generally) be obtained with the function `formals()`.

```
formals(print)
```

```
$x


$...
```

The first argument in `print()`, `x`, refers to the name of the object to be printed. The second argument is the so-called triple dot placeholder, `...`. This (optional) argument, which is formally considered in Section 8.3.3, allows additional arguments to be passed from various printing methods that can be called using the *generic* function name `print()` (see Section 8.7).

Arguments in **R** functions can be set by users in two ways.

1. One can provide acceptable values for arguments, in the order that the arguments occur in the list reported by `formals()`. For example, for the function `a_function`, if I wish to assign the values `x` and `y` to the *first* and *second* arguments, respectively, I could type: `a_function(x, y)`.
2. One can an refer to an argument by its name, and specify values for the argument using the = operator. That is, for some function `a_function`, with some arguments `foo` and `bar`, that I wish to assign the values `x` and `y`, I could type: `a_function(foo = x, bar = y)`. This approach should be used if one does not remember the order of arguments

in a function (if you don't remember whether `foo` is the first or fifth argument), or if one wishes to change/specify only certain arguments from a large number of arguments.

Example 2.5.
Under approach 2, we can print the object `y`, created in Example 2.4, by typing:

```
print(x = y)
```

```
[1] "Hello, world!"
```

Of course, data object names other than `y` can be supplied to the argument `x` in `print()`. For example, to print an object named `z`, I could use either `print(z)` or `print(x = z)`.

■

One can maintain the default value for an argument by simply ignoring that argument in the function call. For example, if `a_function`, has argument defaults `foo = x`, `bar = y`, and I wished to change the value of `foo` to `baz`, while maintaining the default value for `bar`, I could type: `a_function(foo = baz)` Occasionally, a function's defaults will allow it to run without user value specifications for any arguments. In this case, I could run `a_function` by typing: `a_function()`.

2.3.2 Naming Objects

When binding an **R**-object to a name, we should try to keep the name simple, and avoid names that already represent important definitions and functions. These include: `TRUE, FALSE, NULL, NA, NaN,` and `Inf`. In addition, we cannot have names:

- beginning with a numeric value,
- containing spaces, colons, or semicolons,
- containing mathematical operators (e.g., `*`, `+`, `-`, `^`, `/`, `=`),
- containing important **R** *metacharacters* (e.g., `@`, `#`, `?`, `!`, `%`, `&`, `|`).

However, even these "forbidden" names and characters can be used if one encloses them in backticks, also called *accent grave* characters. For example, the code, `` `?` <- 2 + 2 `` will create an object named `` `?` ``, containing the number 4.

Names should, if possible, be descriptive. Thus, for a object containing 20 random observations from a normal distribution, the name `rN20` may be superior to the easily-typed, but anonymous name, `x`. Finally, we should remember that **R** is case sensitive. That is, each of the following 2^4 combinations will be recognized as distinct: `name, Name, nAme, naMe, namE, NAme, nAMe, naME, NaMe, nAmE, NamE, naME, NAMe, nAME, NaME, NAmE, NAME`.

2.3.3 Listing Objects

The lexical scoping characteristics of **R** (Section 1.4.1.2) have important consequences when considering objects and their names. An object's name will be assigned to a particular **R**

environment –a specialized storage system whose features are formally considered, alongside **R** functions, in Ch 8. By default, an object will be assigned by **R** to the environment where it was defined, although this can be modified.

Only objects in the *current* environment can be directly accessed by calling their names[7]. A list of objects assigned to particular environments can be obtained using the functions `objects()` or `ls()`.

Example 2.6.
The **R** session itself is defined to be the so-called *global* environment: `.GlobalEnv`.

```
environment()
```

```
<environment: R_GlobalEnv>
```

Object searches from `objects()` and `ls()` are limited, by default, to the current environment –which, for this document, *is* the global environment. Currently, I only have the object `y` (which has been applied and modified several times) in `GlobalEnv`.

```
objects()
```

```
[1] "y"
```

Note that in this example I run `environment()` and `objects()` without arguments.

■

2.3.4 Combining Data

To combine a collection of numbers or other data into a single entity, one can use the important **R** function `c()`, which means "combine".

Example 2.7.
To define the numbers 23, 34, and 10 collectively to be an object named `x`, I would type:

```
x <- c(23, 34, 10)
```

We could then do something like:

```
x + 7
```

```
[1] 30 41 17
```

Note that seven was added to each element in `x`.

■

[7] Names defined within **R** functions are specific (local) to those functions, just as function names from **R** packages (Section 3.7) are local to their respective package environments (see Ch 8).

2.3.5 Object Classes

Under the idiom of *object oriented programming* (*OOP*), an object may have attributes that allow it to be evaluated correctly, and associated methods appropriate for those attributes (e.g., specific functions for plotting, printing, etc.)[8].

R objects will generally have a *class*, identifiable with the function `class()`.

```
class(x)
```

```
[1] "numeric"
```

Objects in class `numeric` (and those in several other widely-used classes) can be evaluated mathematically. Some common **R** classes are shown in Table 2.1, along with several new functions used to create objects with those classes, including: `raw()`, `expression()`, `list()`, `factor()`, `function()`, `matrix()`, `array()`, and `data.frame()`. We will learn about these functions, and create objects representing all of these classes over the next few chapters. We will also learn how to create our own personalized classes and associated methods (Section 8.7).

Table 2.1: Common **R** classes for some object `x`. The listed class would be printed if one created the assignment for `x` shown in the Example, and typed `class(x)`.

Class	Example
`logical`	`x <- TRUE`
`numeric`	`x <- 2 + 2`
`integer`	`x <- 1:3`
`character`	`x <- c("a","b","c")`
`complex`	`x <- 5i`
`raw`	`x <- raw(2)`
`expression`	`x <- expression(x * 4)`
`list`	`x <- list()`
`factor`	`x <- factor("a","a","b")`
`function`	`x <- function(y)y + 1`
`matrix`	`x <- matrix(nrow = 2, rnorm(4))`
`array`	`x <- array(rnorm(8), c(2, 2, 2))`
`data.frame`	`x <- data.frame(v1 = c(1,2), v2 = c("a","b"))`

2.3.6 Object Base Types

All **R** objects will have so-called *base types* that define their underlying C language data structures. Specifically, **R** base types correspond to an underlying C-codified *typedef*, an alias framework for C data types. This internal process is referred to by the **R**-core development

[8] There are many OOP languages including **R**, C#, C++, Objective-C, Smalltalk, Java, Perl, Python and PHP. C is not considered an OOP language.

team as `SEXPTYPE`, meaning **S**-expression (`SEXP`) type (R Core Team, 2024a). There are currently 24 `SEXPTYPE` variants (R Core Team, 2024a), each corresponding to one of the 24 **R** base types (Table 2.2), and it is unlikely that more will be developed in the near future (Wickham, 2019). The meaning and usage of some of the base types may seem clear, for instance, `integer` and `character`, which are also class designations (Table 2.1). Most of the base types are specifically addressed in later chapters, including `list`, `complex`, `logical`, `integer`, `NULL`, and `symbol` (Chs 3), `character` and `language` (Chs 4 and 5), `closure`, `special`, `builtin`, `environment`, `pairlist`, S4, and `promise` (Ch 8) and `raw` and `double` (Ch 12). Base types meant for C-internal processes, i.e., `any`, `bytecode`, `promise`, `...`, `weakref`, `externalptr`, and `char`, are not easily accessible with interpreted **R** code (R Core Team, 2024b). Underlying `SEXP` types are considered only infrequently through the remainder of the book.

Table 2.2: **R** base types, examples, and corresponding `SEXP` types. The listed base type would be printed if one created the assignment for `x` shown in the Example and typed `typeof(x)`. The function `mle`, used to create the Example for base type `S4`, requires the package *stats4*.

Base type	Example	Application	SEXP
`NULL`	`x <- NULL`	vectors	`NILSXP`
`logical`	`x <- TRUE`	vectors	`LGLSXP`
`integer`	`x <- 1L`	vectors	`INTSXP`
`complex`	`x <- 1i`	vectors	`CPLXSXP`
`double`	`x <- 1`	vectors	`REALSXP`
`list`	`x <- list()`	vectors	`VECSXP`
`character`	`x <- "a"`	vectors	`STRXSP`
`raw`	`x <- raw(2)`	vectors	`RAWSXP`
`closure`	`x <- function(y)y + 1`	closure functions	`CLOSXP`
`special`	`x <- `[``	special functions	`SPECIALSXP`
`builtin`	`x <- sum`	builtin functions	`BUILTINSXP`
`expression`	`x <- expression(x * 4)`	expressions	`EXPRSXP`
`environment`	`x <- globalenv()`	environments	`ENVSXP`
`symbol`	`x <- quote(a)`	language components	`SYMSXP`
`language`	`x <- quote(a + 1)`	language components	`LANGSXP`
`pairlist`	`x <- formals(mean)`	language components	`LISTSXP`
`S4`	`x <- stats4::mle(function(x=1)x^2)`	non-simple objects	`OBJSXP`
`any`	No example	C-internal	`ANYSXP`
`bytecode`	No example	C-internal	`BCODESXP`
`promise`	No example	C-internal	`PROMSXP`
`...`	No example	C-internal	`DOTSXP`
`weakref`	No example	C-internal	`WEAKREFSXP`
`externalptr`	No example	C-internal	`EXTPTRSXP`
`char`	No example	C-internal	`CHARSXP`

Base types of numeric objects define their *storage mode*, i.e., the way **R** caches them in its primary memory[9]. Base types can be identified using the function `typeof()`.

Example 2.8.
For example, for our latest version `x` from Example 2.7 we have:

```
typeof(x)
```

```
[1] "double"
```

We see that `x` has storage mode `double`, meaning that its numeric values are stored using up to 53 bits, resulting in recognizable and distinguishable values between approximately 5×10^{-323} and 2×10^{307} (see Ch 12 for more information).

The **R** session itself (the global environment) has base type `environment`:

```
typeof(.GlobalEnv)
```

```
[1] "environment"
```

■

2.3.7 Object Attributes

Many **R**-objects will also have *attributes* (i.e., characteristics particular to the object or object class).

Example 2.9.
Typing:

```
attributes(x)
```

```
NULL
```

indicates that `x` (as defined in Example 2.7) does not have additional attributes. However, using *coercion* (Section 3.3.4) we can define `x` to be an object of class `matrix` (a collection of data in a row and column format (see Section 3.1.2)).

```
attributes(as.matrix(x))
```

```
$dim
[1] 3 1
```

[9]The functions `mode()` and `storage.mode()` are generally not appropriate for identifying **R** base types and storage modes (Wickham, 2019). In particular, the function `mode()` gives the mode of an object with respect to the **S3** system (see Becker et al. (1988)), whereas `storage.mode()` is generally used when interfacing with algorithms written in other languages, primarily C or Fortran, to check that **R** objects have the correct type for the interfaced language.

Now `x` has the attribute `dim` (i.e., dimension). Specifically, `x` is a three-celled matrix. It has three rows and one column.

■

Amazingly, underlying object characteristics allow **R** to simultaneously store and distinguish objects with the same name. For instance:

```
mean <- mean(c(1, 2, 3))
mean
```

```
[1] 2
```

```
mean(c(1, 2, 3))
```

```
[1] 2
```

In general, it is not advisable to name an object after a frequently used function. Nonetheless, the function `mean()`, which calculates the arithmetic mean of a collection of data, is distinguishable from the new user-created object `mean`, because these objects have different underlying characteristics. We can remove the user-created object `mean`, with the function `rm()`. This leaves behind only the function `mean()`, which I print below:

```
rm(mean)
mean
```

```
function (x, ...)
UseMethod("mean")
<bytecode: 0x0000012d8c1153f8>
<environment: namespace:base>
```

The capacity of **R** to track and distinguish object names is a primary focus of Section 8.8.

2.4 Getting Help

There is no single perfect source for information/documentation for all aspects of **R**. Detailed manuals from CRAN are available concerning the **R** language definition, basic operations, and package development. These resources, however, often assume a familiarity with Unix/Linux operating systems and computer science terminology. Thus, they may not be particularly helpful to biologists who are new to **R**.

2.4.1 `help()` and `?`

A comprehensive help system is available for many **R** components including operators, and loaded package dataframes and functions. The system can be accessed via the question mark, `?`, operator and the function `help()`.

Example 2.10.
For instance, if I wanted to know more about the `plot()` function, I could type:

```
?plot
```

or

```
help(plot)
```

■

Documentation for packaged **R** functions (Section 3.7) must include an annotated description of function arguments, along with other pertinent information, and documentation for packaged datasets must include descriptions of dataset variables[10]. The quality of documentation will generally be excellent for functions from packages in the default **R** download (i.e., the **R**-distribution packages, see Section 3.7), but will vary from package to package otherwise.

For help and documentation concerning programming metacharacters used in **R** (for instance `@`, `#`, `?`, `!`, `%`, `&`, `|`), one would enclose the metacharacters with quotes. For example, to find out more information about the logical operator `&` I could type `help("&")` or `? "&"`. Placing two question marks in front of a topic will cause **R** to search for help files concerning with respect to all packages in a workstation.

Example 2.11.
For instance, type:

```
??lm
```

or, alternatively

```
help.search(lm)
```

for a huge number of help files on linear model functions identified through fuzzy matching.

■

Help for particular **R**-questions can often be found online using the search engine at http://search.r-project.org/. This link is provided in the Help pulldown menu in the **R** console (non-Linux only). Helpful online discussions can also be found at Stack Overflow, and Stats Exchange.

2.4.2 `demo()` and `example()`

The function `demo()` allows one access to coded examples that developers have worked out for a particular function or topic. For instance, type:

[10] Chapter 10 provides instructions on how to develop documentation files for your own packages.

```
demo(graphics)
```

for a brief demonstration of **R** graphics. Typing

```
demo(persp)
```

will provide a demonstration of 3D perspective plots. And, typing:

```
demo(Hershey)
```

will provide a demonstration of available modifiable symbols from the *Hershey family of fonts* (see Ch 6 in Hershey (1967)). Finally, typing:

```
demo()
```

lists all of the demos available in the loaded libraries for a particular workstation. The function `example()` usually provides less involved demonstrations from the `man` package directories (short for user manual, see Ch 10) in an **R** package. For instance, type:

```
example(plotmath)
```

for a coded demonstration of mathematical graphics.

2.4.3 Vignettes

R packages often contain *vignettes*. These are short documents that generally describe the theory underlying algorithms and guidance on how to correctly use package functions. Vignettes can be accessed with the function `vignette()`. To view all vignettes for all *installed* packages (Section 3.7.1), type:

```
vignette(all = TRUE)
```

To view all vignettes available for *loaded* packages (see Section 3.7.2), type:

```
vignette(all = FALSE)
```

To view vignettes for the **R** contributed package *asbio* (following its installation), type:

```
vignette(package = "asbio")
```

To see the vignette `simpson` in package *asbio*, type:

```
vignette("simpson", package = "asbio")
```

The function `browseVignettes()` provides an HTML-browser that allows interactive vignette searches.

2.5 Keyboard Shortcuts

R contains a number of useful keyboard shortcuts. For example, At this point, it may be evident that the **R**-console can quickly become cluttered and confusing. To remove console text (without actually getting rid of any of the objects created in a session) press Ctrl + L or, from the **Edit** pulldown menu, click **Clear console** (non-Linux only). A full list of keyboard shortcuts can be obtained by typing: Alt + Shift + K (Windows and Linux) or Option + Shift + K (Mac OS). Keyboard shortcuts can often be modified, or even created, if one is running **R** from a sophisticated IDE like RStudio (Section 2.10).

2.6 Options

To enhance an **R** session, we can adjust the appearance of the **R**-console and customize options that affect expression output. These include the characteristics of the graphics devices, the width of print output in the **R**-console, and the number of print lines and print digits. Changes to some of these parameters can be made by going to **Edit**>**GUI Preferences** in the **R**-toolbar. Many other parameters can be changed using the `options()` function. To see all alterable options one can type:

```
options()
```

The resulting list is extensive. To modify options, one would simply define the desired change within parentheses following a call to `options`. For instance, to see the default number of digits, I would type:

```
options("digits")
```

```
$digits
[1] 7
```

To change the default number of digits in output from 7 to 5 in the current session, I would type:

```
options(digits = 5)
# demonstration using pi
pi
```

```
[1] 3.1416
```

One can revert back to default options by restarting an **R** session.

2.6.1 Advanced Options

To store user-defined options and start up procedures, an `.Rprofile` file will exist in your **R** program **etc** directory. This location would be something like: ...**R/R-version/etc**. **R** will silently run commands in the `.Rprofile` file upon opening. Thus, by customizing the

`.Rprofile` file one can "permanently" set session options, load installed packages, define your favorite package repository (Section 3.7), and even create aliases and defaults for frequently used functions.

The `.Rprofile` file located in the **etc** directory is the so-called `.Rprofile.site` file. Additional `.Rprofile` files can be placed in the working directory (see below). **R** will check for these and run them after running the `.Rprofile.site` file.

Example 2.12.
Here is the content of one of my current `.Rprofile` files.

```
1 options(repos = structure(c("http://ftp.osuosl.org/pub/cran/")))
2 .First <- function(){
3 library(asbio)
4 cat("\nWelcome to R Ken! ", date(), "\n")
5 }
6 .Last <- function(){
7 cat("\nGoodbye Ken", date(), "\n")
8 }
```

The command `options(repos = structure(c("http://ftp.osuosl.org/pub/cran/")))` (Line 1) defines my preferred CRAN repository mirror site (see Section 3.7). The function `.First( )` (Lines 2-5) will be run at the start of the **R** session and `.Last( )` (Lines 6-8) will be run at the end of the session. **R** functions will formally introduced in Ch 8. As we go through this book it will become clear that these lines of code force **R** to say hello, and to load the package *asbio* (**R** packages are formally considered in Section 3.7), and print the date/time (using the function `date()`) when it opens, and to say goodbye, and print the date/time when it closes (although the farewell will only be seen when running **R** from a shell interface, e.g., the Windows Command Prompt).

■

One can create `.Rprofile` files, and many other types of **R** extension files using the function `file.create()`. For instance, the code:

```
file.create("defaults.Rprofile")
```

will place an empty, editable, `.Rprofile` file called `defaults` in the working directory.

2.7 The Working Directory

By default, the **R** *working directory* is set to be the home directory of the workstation. The command `getwd()` shows the current file path for the working directory.

The working directory can be changed with the command `setwd(filepath)`, where `filepath` is the location of the desired directory, or by using pulldown menus, i.e., **File**>**Change dir**

(non-Linux only). Because **R** developed under Unix, we must specify directory hierarchies using forward slashes or by doubling backslashes.

Example 2.13.
Here is the actual working directory of this (GitHub-linked) manuscript.

```
getwd()
```

```
[1] "C:/Users/ahoken/Documents/GitHub/Amalgam"
```

To establish a working directory file path to the Windows directory: **C:\Users\User\Documents**, I would type:

```
setwd("C:/Users/User/Documents")
```

or

```
setwd("C:\\Users\\User\\Documents")
```

■

2.8 Saving and Loading Your Work

As noted in Ch 1, an **R** session is allocated with a fixed amount of memory that is managed in an on-the-fly manner. An unfortunate consequence of this is that if **R** crashes, all unsaved information from the work session will be lost. Thus, session work should be saved often. Note that **R** will not give a warning if you are writing over session files from the **R** console. The old file will simply be replaced. Three general approaches for saving non-graphics data are possible. These are: 1) saving the history, 2) saving objects, and 3) saving **R** script. All three of these operations can be greatly facilitated by using an **R** integrated development environment like RStudio (Section 2.10).

2.8.1 R History

To view the *history* (i.e., the commands that have been used in a session) one can use `history(n)` where `n` is the number of previous command lines one wishes to see[11]. For instance, to see the last three commands, one would type[12]:

[11] Importantly, the functions `savehistory()`, `loadhistory()`, and `history()` are not currently supported for Mac OS. There are ways around this. For instance, in RStudio (Section 2.10), the Mac OS command history can be obtained by clicking the **History** icon that appears on the tool bar at the top of the console window. As an additional issue, Windows and Unix-alike platforms have different implementations for `savehistory()` and `loadhistory()`. See help pages for these functions within your platform for particulars.

[12] This command will not work in an embedded Windows R GUI, like the one in RStudio.

```
history(3)
```

To save the session history in Windows one can use **File**>**Save History** or the function `savehistory()`. For instance, to save the session history to the working directory under the name `history1`, I could type:

```
savehistory(file = "history1.Rhistory")
```

We can view the code in this file from any text editor. To load the history from a previous session one can use **File**>**Load History** (non-Linux only) or the function `loadhistory()`. For instance, to load `history1` I would type:

```
loadhistory(file = "history1.Rhistory")
```

To save the history at the end of (almost) every interactive Windows or Unix-alike **R** session, one can alter the `.Rprofile` file `.Last` function to include:

```
.Last <- function() if(interactive()) try(savehistory("~/.Rhistory"))
```

2.8.2 R Objects

To save all of the objects available in the current **R**-session one can use **File**>**Save Workspace** (non-Linux only), or simply type:

```
save.image()
```

This procedure saves session objects to the working directory as a nameless file using an `.RData` extension. The file will be opened, silently, with the inception of the next **R**- session, and cause objects used or created in the previous session to be available. Indeed, **R** will automatically execute all `.RData` files in the working directory for use in a session. Stored `.RData` files can also be loaded using **File**>**Load Workspace** (non-Linux only). One can also save `.RData` objects to a specific directory location and use a specific file name using: **File**>**Save Workspace**, or with the flexible function `save()`. **R** data file formats, including .rda, and .RData, (extensions for **R** data files), and .R (the format for **R** scripts), can be read into **R** using the function `load()`. Users new to a command line environment will be reassured by typing:

```
load(file.choose())
```

The function `file.choose()` will allow one to browse interactively for files to load using dialog boxes. Detailed procedures for importing (reading) and exporting (saving) data with a row and column format, and an explicit delimiter (e.g. .csv files) are described in Ch 3.

2.8.3 R Scripts

To save an **R** *script* as an source code file, it is best to use an *Integrated Development Environment* (IDE) compatible with **R**. **R** contains its own IDE, the **R**-editor, which is useful for writing, editing, and saving scripts as .r extension files (Fig 2.3). To access the **R**-editor go to **File**>**New script** (non-Linux only) or type the shortcut Ctrl + F + N (Windows or Linux) or Cmd + F + N (Mac OS) . Code written in the **R**-editor IDE can be sent directly to the **R**-console by copying and pasting or by selecting code and using the shortcut Ctrl + R (Windows and Linux) or Cmd + R (Mac OS).

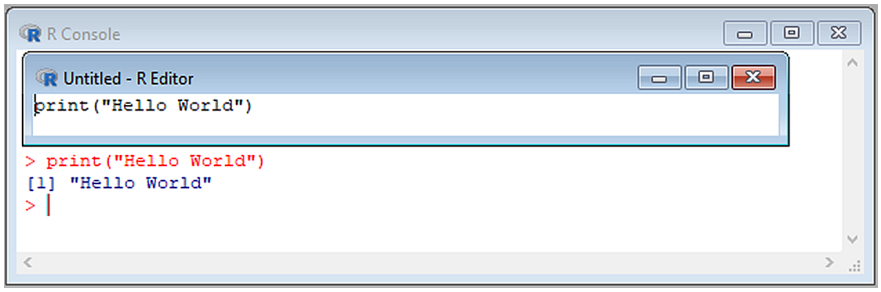


Figure 2.3: The **R**-editor providing code for a famous computational exercise.

Aside from the **R**-editor, a number of other IDEs outside of **R** allow straightforward generation of **R** script files, and a direct link between text editors, that provide syntax highlighting for **R** code, and the **R**-console itself. These include RWinEdt (an **R** package plugin for WinEdt), Tinn-R, a recursive acronym for Tinn is not Notepad, ESS (Emacs Speaks Statistics), Jupyter Notebook, a web-based IDE originally designed for Python, but useful for many languages, and particularly RStudio, which will be introduced later in this chapter[13].

Saved **R** scripts can be called and executed using the function `source()`. To browse interactively for source code files, one can type:

```
source(file.choose())
```

or go to **File**>**Source R code**.

2.9 Basic Mathematics

A large number of mathematical operators and functions are available with a conventional download of **R**.

Elementary mathematical operators, common mathematical constants, trigonometric functions, derivative functions, integration approaches, and basic statistical functions are shown in shown in Tables 2.3 - 2.9.

[13] Other text editors with at least some IDE support for **R** include, but are not limited to, NppToR in Notepad++, Bluefish, Crimson Editor, ConTEXT, Eclipse, Vim, Geany, jEdit, Kate, TextMat, gedit, and SciTE.

2.9.1 Elementary Operations

Elementary mathematical operations and functions (Table 2.3), and even those for specialized processes, can generally be applied to a wide variety of numeric object classes. For instance, the expression `log(x)` could be applied if `x` was a scalar (e.g., `x = 3`), or a collection of numbers, e.g., `x = c(3, 7, 8)`. In the latter case, the natural logarithm would be be calculated for each element in `x`, and those transformed outcomes would be returned by the function. Notably, this form of intuitive scripting is a dramatic departure from approaches used by many other computer languages[14].

[14] For instance, sums in C and Fortran are generally obtained using loops (Section 8.5). We should not forget, however, that functions like `sum()` (and even `` `+` ``) are underlain by C executables (see Section 8.1).

Table 2.3: Elementary mathematical operators and functions in **R**. For all functions `x` represents a scalar or a numeric vector.

Operation	Function/Operator	To find:	We type:
addition	`+`	$2+2$	`2 + 2`
subtraction	`-`	$2-2$	`2 - 2`
multiplication	`*`	2×2	`2 * 2`
division	`/`	$\frac{2}{3}$	`2/3`
modulo	`%%`	remainder of $\frac{5}{2}$	`5%%2`
integer division	`%/%`	$\frac{5}{2}$ without remainder	`5%/%2`
exponentiation	$\wedge$	2^3	`2^3`
$\mid x \mid$	`abs(x)`	$\mid -23.7 \mid$	`abs(-23.7)`
round x to d digits	`round(x, digits = d)`	round -23.71 to 1 digit	`round(-23.71, 1)`
round x up to closest whole num.	`ceiling(x)`	ceiling(2.3)	`ceiling(2.3)`
round x down to closest whole num.	`floor(x)`	floor(2.3)	`floor(2.3)`
$\sqrt{x}$	`sqrt(x)`	$\sqrt{2}$	`sqrt(2)`
$\log_e x$	`log(x)`	$\log_e 5$	`log(5)`
$\log_b x$	`log(x, base = b)`	$\log_{10} 5$	`log(5, base = 10)`
$x!$	`factorial(x)`	$5!$	`factorial(5)`
$\binom{n}{x} = \frac{n!}{x!(n-x)!}$	`choose(n,x)`	$\binom{5}{2}$	`choose(5,2)`
$\Gamma(x)$	`gamma(x)`	$\Gamma(3.2)$	`gamma(3.2)`
$B(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$	`beta(a,b)`	$B(3,2)$	`beta(3,2)`
$\sum_{i=1}^{n} x_i$	`sum(x)`	sum of `x`	`sum(x)`
cumulative sum	`cumsum(x)`	cum. sum of `x`	`cumsum(x)`
$\prod_{i=1}^{n} x_i$	`prod(x)`	product of `x`	`prod(x)`
cumulative product	`cumprod(x)`	cum. prod. of `x`	`cumprod(x)`

2.9.2 Associativity and Precedence

Note that the operation:

```
2 + 6 * 5
```

```
[1] 32
```

is equivalent to $2+(6\cdot 5) = 32$. This is because the * operator gets higher priority (precedence) than +. Evaluation precedence can be modified with parentheses:

```
(2 + 6) * 5
```

```
[1] 40
```

In the absence of operator *precedence*, mathematical operations in **R** are (generally) read from left to right (that is, their *associativity* is from left to right) (Table 2.4). This corresponds to the conventional order of operations in mathematics. For instance:

```
2 + 2^(2 + 1)
```

```
[1] 10
```

Table 2.4: Precedence and associativity of mathematical operators. Operators are listed from highest to lowest precendece in operations.

Precedent	Operator	Description	Associativity
1	^	exponent	right to left
2	%%	modulo	left to right
3	* /	multiplication, division	left to right
4	+ -	addition, subtraction	left to right

Example 2.14.
Here are some other simple mathematical examples. To solve $1/\sqrt{22!}$, I could type:

```
1/sqrt(factorial(22))
```

```
[1] 2.9827e-11
```

And to solve $\Gamma\left(\sqrt[3]{23\pi}\right)$, I could type:

```
gamma((23 * pi)^(1/3))
```

```
[1] 7.411
```

By default the function `log()` computes natural logarithms, i.e.,

```
log(exp(1))
```

```
[1] 1
```

The `log()` function can also compute logarithms to a particular base by specifying the base in an optional second argument called `base`. For instance, to solve the operation: $\log_{10} 3 + \log_3 5$, one could type:

```
log(3) + log(5)
```

```
[1] 2.7081
```

or

```
log(x = 3, base = 10) + log(x = 5, base = 3)
```

```
[1] 1.9421
```

■

2.9.3 Constants

R allows easy access to most conventional constants (Table 2.5).

Table 2.5: Conventional constants in **R**.

Operation	Operator/Function	To find:	We type:
$-\infty$	`-Inf`	$-\infty$	`-Inf`
∞	`Inf`	∞	`Inf`
$\pi = 3.141593\ldots$	`pi`	π	`pi`
$e = 2.718282\ldots$	`exp(1)`	e	`exp(1)`
e^x	`exp(x)`	e^3	`exp(3)`

2.9.4 Trigonometry

R assumes that the inputs for trigonometric functions are in radians. Of course degrees can be obtained from radians using $Degrees = Radians \times 180/\pi$, or conversely $Radians = Degrees \times \pi/180$ (Table 2.6). Note that there are no base-**R** functions for cotangent, secant or cosecant. However, for some angle x, measured in radians, these are readily obtained as: $\cot(x) = \cos(x)/sin(x)$, $\sec(x) = 1/\cos(x)$, and $\csc(x) = 1/\sin(x)$.

2.9.5 Derivatives

The function `D()` finds symbolic and numerical derivatives of simple expressions. It requires two arguments, 1) a mathematical function specified as an object of class `expression`, and 2) the variable name in the *differential* (the denominator in the difference quotient).

Table 2.6: Trigonometric functions in **R**. For all functions `x` represents a scalar or a numeric vector.

Operation	Operator/Function	To find:	We type:
$\cos(x)$	`cos(x)`	$\cos(3 \text{ rad.})$	`cos(3)`
$\sin(x)$	`sin(x)`	$\sin(45^\circ)$	`sin(45 * pi/180)`
$\tan(x)$	`tan(x)`	$\tan(3 \text{ rad.})$	`tan(3)`
$\text{acos}(x)$	`acos(x)`	$\text{acos}(45^\circ)$	`acos(45 * pi/180)`
$\text{asin}(x)$	`asin(x)`	$\text{asin}(3 \text{ rad.})$	`asin(3)`
$\text{atan}(x)$	`atan(x)`	$\text{atan}(45^\circ)$	`atan(45 * pi/180)`
$\cosh(x)$	`cosh(x)`	$\cosh(3 \text{ rad.})$	`cosh(3)`
$\sinh(x)$	`sinh(x)`	$\sinh(45^\circ)$	`sinh(45 * pi/180)`
$\tanh(x)$	`tanh(x)`	$\tanh(3 \text{ rad.})$	`tanh(3)`
$\cot(x)$		$\cot(3 \text{ rad.})$	`cos(3)/sin(3)`
$\sec(x)$		$\sec(3 \text{ rad.})$	`1/cos(3)`
$\csc(x)$		$\csc(3 \text{ rad.})$	`1/sin(3)`

Objects of class `expression`, can be created using the function `expression()`, and evaluated with the function `eval()`).

Example 2.15.
Here is an example of how the functions `expression()` and `eval()` can be used:

```
eval(expression(2 + 2))
```

```
[1] 4
```

Of course we wouldn't bother to use `expression()` and `eval()` in such simple applications.

■

Table 2.7 contains specific examples using `D()`.

Table 2.7: Evaluation of derivatives in **R** using `D()`.

To find:	We type:
$\frac{d}{dx}5x$	`D(expression(5 * x), "x")`
$\frac{d^2}{dx^2}5x^2$	`D(D(expression(5 * x^2), "x"), "x")`
$\frac{\partial}{\partial x}5xy + y$	`D(expression(5 * x * y + y), "x")`

Example 2.16.
Thus, to solve:

$$\frac{d}{dx}20x^{-4}$$

I could use:

```
e <- expression(20 * x^(-4))
D(e, "x")
```

```
20 * (x^((-4) - 1) * (-4))
```

Unfortunately, it is left to us to simplify the ugly output. That is,

$$\begin{aligned}\frac{d}{dx}(20x^{-4}) = & \\ &= 20 \times (x^{(-4)-1)} \times (-4)) \\ &= -80x^{-5} \\ &= -\frac{80}{x^5}\end{aligned}$$

■

Several other **R** functions provide tidier derivative results compared to `D()`, although they require the installation and loading of additional packages, not included in a conventional download of **R**. See Section 3.7 for a thorough introduction to **R** packages. For instance, the function `Deriv()`, from the package *Deriv* can be applied using two approaches[15].

- Under the first approach, a differentiable function is defined as an **R** `function` (see Ch 8) whose one argument is the variable name in the differential. This function is then used as the single required argument in `Deriv()`.
- With the second approach, a differentiable function is defined as a character string. This is then used as the first argument in `Deriv()`. The variable name in the differential is defined in a second argument.

Example 2.17.
To obtain the derivative in Example 2.16 using `Deriv()` we would first install the *Deriv* package (for instance using: `install.packages("Deriv")`) and load the package using:

```
library(Deriv) # loads Deriv
```

Under the first approach we could then type:

```
d <- Deriv(function(x) 20 * x^(-4))
d
```

```
function (x)
-(80/x^5)
```

Note that the output, `d`, is a function, allowing one to obtain instantaneous slopes for specified `x` values.

[15]The documentation for `Deriv()` actually lists six approaches (see `?Deriv`).

```
d(c(-1, 2, 3, 5.2))
```

```
[1] 80.000000 -2.500000 -0.329218 -0.021041
```

Under the second approach, we could specify

```
Deriv("20 * x^(-4)", "x")
```

```
[1] "-(80/x^5)"
```

Note that the output is a character string.

Both approaches allow one to obtain higher order derivatives and partial derivatives. For instance,

```
Deriv(d) # second derivative
```

```
function (x)
400/x^6
```

```
Deriv(Deriv("20 * x^(-4)", "x")) # second derivative
```

```
[1] "400/x^6"
```

`D()` results can also be simplified directly with function `Simplify()` from the package *Deriv*. For the current Example, one could use:

```
e <- expression(20 * x^(-4))
Simplify(D(e, "x"))
```

```
-(80/x^5)
```

■

2.9.6 Integration

The function `integrate` solves definite integrals. It requires three arguments. The first is an **R** function defining the integrand. The second and third are the lower and upper bounds of integration.

Example 2.18.
To solve:

$$\int_2^4 3x^2 dx$$

we could type:

```
f <- function(x){3 * x^2}
integrate(f, 2, 4)
```

```
56 with absolute error < 6.2e-13
```

■

R functions are explicitly addressed in Ch 8.

2.9.7 Statistics

R, of course, contains a huge number of statistical functions. These will generally require sample data for summarization. Data can be brought into **R** from spreadsheet files or other data storage files (we will learn how to do this shortly). As we have learned, data can also be assembled in **R**. For instance,

```
x <- c(1, 2, 3)
```

Statistical estimators can be separated into *point estimators*, which estimate an underlying parameter that has a single true value (from a Frequentist viewpoint), and *intervallic estimators*, which estimate the bounds of an interval that is expected, preceding sampling, to contain a parameter at some probability (Aho, 2014). Point estimators can be further classified as estimators of location, scale, shape, and order statistics (Table 2.8). Measures of *location* estimate the typical or central value from a sample. Examples include the arithmetic mean and the sample median. Measures of *scale* quantify data variability or dispersion. Examples include the sample standard deviation and the sample interquartile range (IQR). *Shape estimators* describe the shape (i.e., symmetry and peakedness) of a data distribution. Examples include the sample skewness and sample kurtosis. Finally, the kth order statistic of a sample is equal to its kth-smallest value. Examples include the data minimum, the data maximum, and other quantiles (including the median). Intervallic estimators include confidence intervals (Table 2.9). A huge number of other statistical estimating, modelling, and hypothesis testing algorithms are also available for the **R** environment. For guidance, see Venables and Ripley (2002), Aho (2014), and Fox and Weisberg (2019), among others.

Table 2.8: Simple point estimators in **R**. The term `x` represents a numeric data vector, and `y` represents a numeric data vector whose elements are paired with those in `x`. The cipher `asbio::` indicates that the function is located in the package *asbio* See Section 3.7.

Acronym	Function	Description	Estimator type
$\bar{x}$	`mean(x)`	arithmetic mean of x	location
	`mean(x, trim = t)`	trimmed mean of x for $0 \leq t \leq 1$.	location
GM	`asbio::G.mean(x)`	geometric mean of x	location
HM	`asbio::H.mean(x)`	harmonic mean of x	location
$\tilde{x}$	`median(x)`	median of x	location order statistic
$mode(x)$	`asbio::Mode(x)`	mode of x	location
s	`sd(x)`	standard deviation of x	scale
s^2	`var(x)`	variance of x	scale
$cov(x, y)$	`cov(x, y)`	covariance of x and y	scale
$r_{x,y}$	`cor(x, y)`	Pearson correlation of x and y	scale
IQR	`IQR(x)`	interquartile range of x	scale order statistic
MAD	`mad(x)`	median absolute deviation of x	scale
g_1	`asbio::skew(x)`	skew of x	shape
g_2	`asbio::kurt(x)`	kurtosis of x	shape
$min(x)$	`min(x)`	min of x	order statistic
$max(x)$	`max(x)`	max of x	order statistic
$\hat{F}^{-1}(p)$	`quantile(x, prob = p)`	quantile of x at lower-tailed probability p	order statistic

Table 2.9: Some intervallic estimators in **R**. The term `x` represents a numeric vector. The cipher `asbio::` indicates that the function is located in the package *asbio*

Function	Description
`asbio::ci.mu.z(x, conf, sigma)`	Conf. int. for μ at level `conf`. True SD = `sigma`.
`asbio::ci.mu.t(x, conf)`	Conf. int. for μ at level `conf`. σ unknown.
`asbio::ci.median(x, conf)`	Conf. int. for true median at level `conf`.

2.10 RStudio

RStudio is an open source IDE for **R** (Fig 2.4). RStudio greatly facilitates writing **R** code, saving and examining **R** objects and history, and many other processes. These include, but are not limited to, documenting session workflows (Section 2.10.2), writing **R** package documentation (Section 10.5), calling and receiving code from other languages (Section 9.1.3), and even developing web-based graphical user interfaces (Section 11.5). RStudio can currently be downloaded at (https://posit.co/products/open-source/rstudio/). Like **R** itself, RStudio can be used with Windows, Mac, and Unix/Linux operating systems. Unlike **R**, RStudio has both freeware and commercial versions[16]. We will use the former here.

Figure 2.4: The RStudio logo.

RStudio is generally implemented using a four pane workspace (Fig 2.5). These panes will contain: 1) the code editor, 2) the **R**-console, 3) the environment and histories panel, and 4) the plots and other miscellany panel. Tabs in panels may vary to a small degree depending on the underlying character of the source code being edited, and whether an RStudio project is open (Section 2.10.1).

[16] On 7/27/2022 RStudio announced it was shifting to a new name, Posit, to acknowledge its growth beyond a simple IDE for **R**. The RStudio name will be retained for RStudio Desktop, and the RStudio Server, but it will be changed for other applications including the RStudio Workbench (now Posit Workbench) and the RStudio Package Manager (now Posit Package Manager).

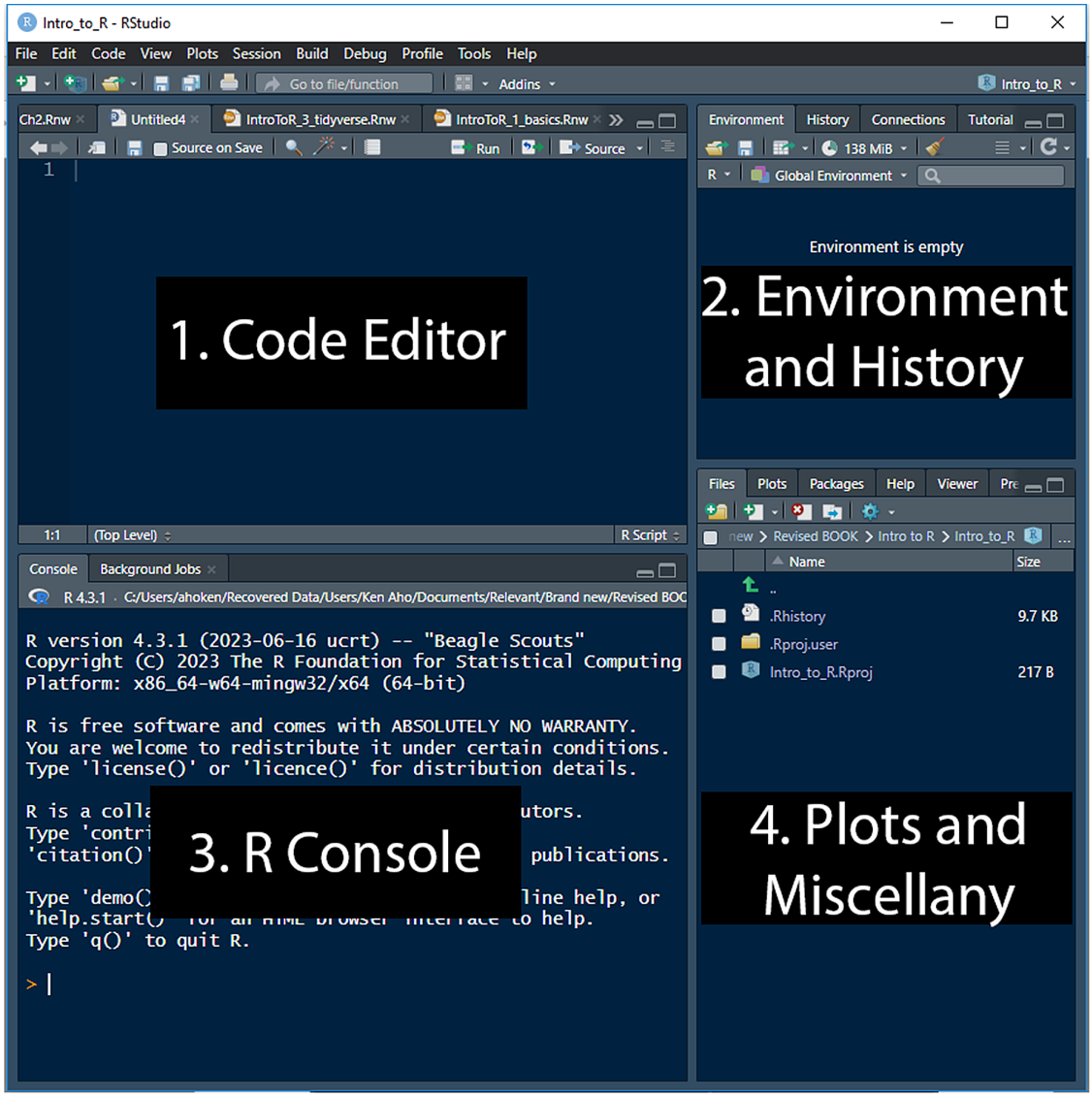


Figure 2.5: Interfaces for RStudio 2023.06.2 Build 561.

- The RStudio **Code Editor** panel (Fig 2.5, Panel 1) allows one to create **R** scripts and even scripts for other languages that can be called to and from **R** (Ch 9). The code panel can also be used to create and edit session documentation files (see Section 2.10.2 below) and other important **R** file types. A new **R** script can be created for editing within the code editor by going to **File**>**New**>**R Script**. Commands from an **R** script can be sent to the **R** console using the shortcut Ctrl + Enter (Windows and Linux) or Cmd + Enter (Mac).

- The **R-console** panel (Fig 2.5, Panel 2) by default, is identical in functionality to the **R** console of the most recent version of **R** on your workstation (assuming that all of the paths and environments are set up correctly on your computer). Thus, the console panel can be used directly for typing and executing **R** code, or for receiving commands from the code editor (Panel 1).

- The **Environments and History** panel (Fig 2.5, Panel 3) can be used to: 1) show a list of **R** objects available in your **R** session (the **Environment** tab), or 2) show, search, and select from the *history* of all previous commands (**History** tab). This panel also provides an interface for point and click import of data files including .csv, .xls, and many other file formats (**Import Dataset** pulldown within the **Environment** tab).
- The **Plots and Miscellany** panel (Fig 2.5, Panel 4) can be used to show: 1) files in the working directory, 2) a scrollable history of plots and image files, and 3) a list of available packages (via the **Packages** tab), with facilities for *updating* and *installing* packages. If a package is in the GUI list, then the package is currently *loaded*. Packages and their installation, updating, and loading are formally introduced in Section 3.7. The panel's **Files** pulldown tab allows straightforward establishment of working directories (although this can still be done at the command line using `setwd()`) (Fig 2.7). The panel's **Help** tap opens automatically when uses `?` or `help` for particular **R** topics (Section 2.4).

CAUTION!

Be very careful when managing files in the **Plots and Miscellany** panel, as you can permanently delete files without (currently) the possibility of recovery from a Recycling Bin.

2.10.1 RStudio Project

An RStudio *project* can be be created via the **File** pulldown menu (Fig 2.7). A project allows all related files (data, figures, summaries, etc.) to be easily organized together by setting the working directory to be the location of the project .Rproj file.

2.10.2 Workflow Documentation

We can document workflow and simultaneously run/test **R** session code by either:

1. Creating an **R** Markdown[17] .rmd file that can be compiled to generate an .html, .pdf, or MS Word® .doc file, or
2. Using Sweave, an approach that implements the LaTeX[18] document preparation system.

[17] Markup languages codify the structure and formatting of a document and generally, the relationships among its components including headings, paragraphs, and hyperlinks (Wikipedia, 2025b). Markup typically refers to *procedural* or *descriptive* languages in which users create code files, with text and semantic tags that will not resemble the final output, as this will requires code compilation. Examples include troff, TeX (which underlies LaTeX documents), Markdown, HTML, and XML. HyperText Markup Language (HTML) is the standard markup language for documents designed for web browser display. Markdown is a highly flexible markup language for creating formatted text using plain-text. **R** Markdown extends Markdown by allowing users to embed code chunks (from **R** and other languages) that can be be evaluated and printed during compilation of final output.

[18] LaTeX – pronounced *lay-tek* or *luh-tek*, depending on who you ask– is an open source, high-quality scientific typesetting system. The **R** packages *sweave* and *knitr* extend LaTeX markup text and formatting, by allowing users to embed **R** code chunks that can be evaluated and printed during compilation of PDF output.

2.10.2.1 R Markdown

The **R** Markdown document processing workflow in RStudio is shown Fig 2.6. These steps are highly modifiable, but can also be run in a more or less automated manner, requiring little understanding of underlying processes.

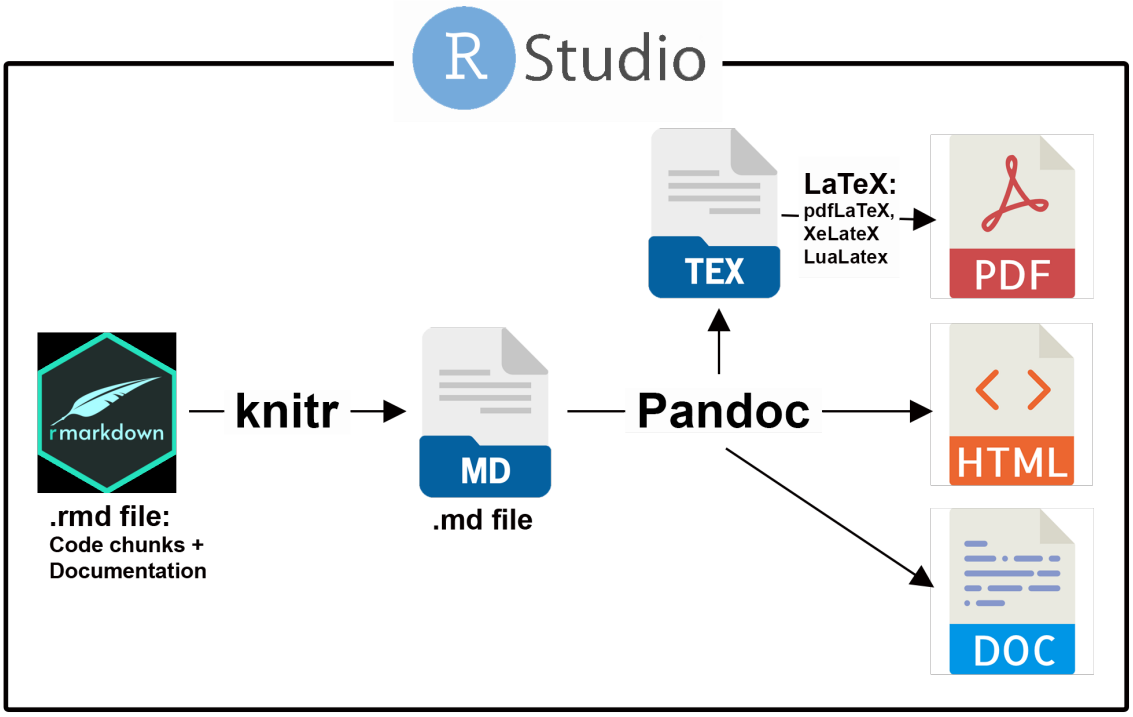


Figure 2.6: The process of document creation in **R** Markdown. Functions in the package *rmarkdown* control conversion of .rmd files to Markdown .md files, using utilities in the package *knitr*. The Pandoc program first creates a .tex file when rendering LaTeX PDF documents.

Use of **R** Markdown and .rmd files requires the package *rmarkdown* (Allaire et al., 2024), which comes pre-installed in RStudio.

As an initial step, all underlying .rmd files must include a brief YAML[19] header (see below) containing document metadata. A nice summary of YAML features and options in **R** Markdown is provided in this cheatsheet. The remainder of the .rmd document will contain text written in Markdown syntax, and code chunks. The `knit()` function from package *knitr* Xie (2015), also installed with RStudio, executes all evaluable code within chunks, and formats the code and output for processing within Pandoc, a program for converting markup files from one language to another[20]. Pandoc uses the YAML header to guide this conversion. As an example,

[19] YAML (pronounced *camel*) is a data serialization language. The YAML acronym was originally intended to mean "Yet Another Markdown Language," but more recently has been given the recursive acronym: "YAML Ain't Markup Language." **R** Markdown uses the YAML format header to communicate with Pandoc, a document converter, written in the Haskell language, embedded in RStudio, to create the desired document output.

[20] Pandoc can convert Markdown .md files, into many formats including, .rtf, .doc, and .pdf

if one has requested HTML output, the simple Markdown text: `This is a script` will be converted to the HTML formatted: `<p>This is a script</p>`. One can also write HTML script or CSS code[21] directly into an .rmd document (see Section 11.5). If the desired output is PDF, Pandoc will convert the .md file into a temporary .tex file, which is then processed by the LaTex typesetting system. Support for LaTeX can be found at its official website, and at a large number of informal user-driven venues, including Stack Exchange and Overleaf, an online LaTeX application. LaTeX will compile the .tex file into a .pdf file. In this process, the *tinytex* package (Xie, 2024), which installs the stripped-down LaTeX distribution TinyTex, can be used.

Creating an **R** Markdown document is simple in RStudio. We first open an empty .rmd document by navigating to **File** > **New File** >**R Markdown** (Fig 2.7).

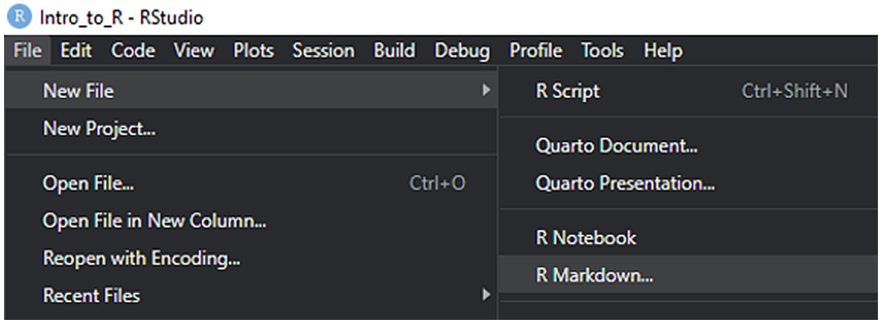


Figure 2.7: Part of the RStudio **File** pulldown menu.

You will delivered to the GUI shown in Fig 2.8. Note that by default Markdown compilation generates an HTML document.

[21] *Cascading Style Sheets* (CSS) is a language for styling output for files written in markup languages including HTML and Extensible Markup Language (XML).

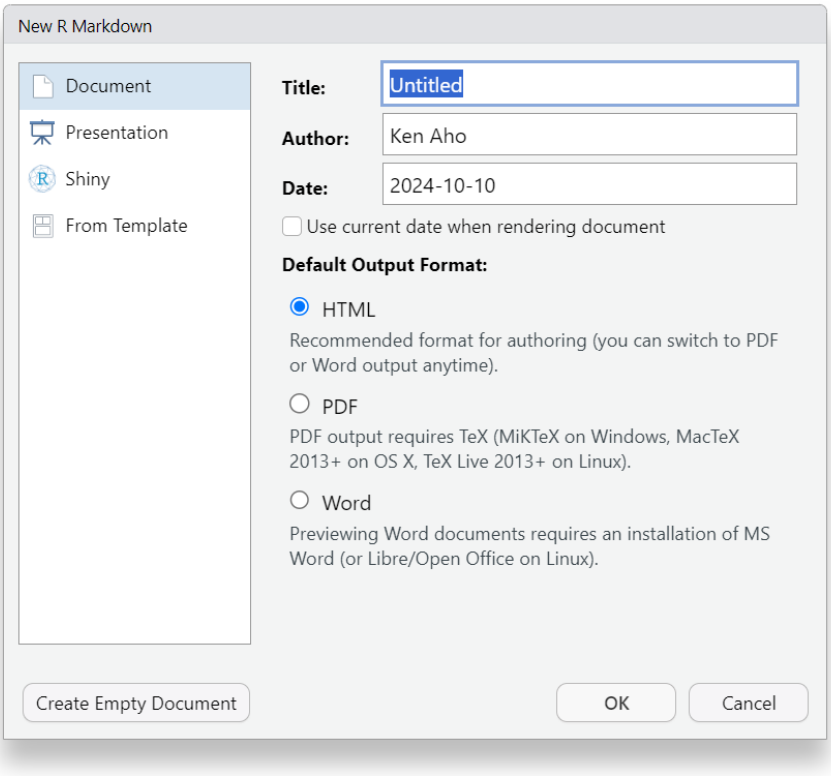


Figure 2.8: RStudio GUI for creating an **R** Markdown document.

The GUI opens a **R** Markdown (.rmd) skeleton document with a tentative YAML header.

```
---
title: "Untitled"
author: "Ken Aho"
date: "2024-10-10"
output: html_document
---
```

Figure 2.9: YAML header to an **R** Markdown (.rmd) skeleton document.

Among other options[22], the default HTML output can be changed to one of:

```
output: pdf_document
```

to create a LaTex → PDF document, or

[22] Many *rmarkdown* output formats are possible including: `html_vignette`, which provides vignette formatting appropriate for inclusion in **R** packages, `ioslides_presentation` and `slidy_presentation`, for an HTML-styled slideshow layout, `beamer_presentation`, for a PDF slideshow using LaTex → Beamer formatting, and `powerpoint_presentation` for a MS Powerpoint® slideshow.

```
output: word_document
```

to create a Word® document.

A potential concern with HTML documents is portability. Your **R** Markdown generated HTML may look fine when viewed from a browser program on the computer you used to create the document. This may not be true, however, if you export this file elsewhere, in the absence of a server host, and without a directory system containing necessary files and applications (see Garsiel (2018)) [23]. There are currently a number of inexpensive (or free) non-dynamic hosting services including GitHub.

2.10.2.1.1 Writing Text Markdown is a relatively simple *procedural markup* language, allowing unformatted text to be written directly into an **R** Markdown document. There are particular scripting procedures, however, for creating headings, formatted text, and other content.

- Pound signs (e.g., `#`, `##`, `###`) can be used as (increasingly nested) hierarchical section delimiters.
- Italic, bold, and monospace code fonts can be specified by enclosing text in asterisks, double asterisks, and back ticks, respectively. That is, `*italic*`, `**bold**`, and `` `code` `` result in: *italic*, **bold**, and `code`.
- Unordered lists can be created with newlines preceded with asterisks, `*`, and ordered lists can be specified with newlines beginning with numbers, e.g., `1.`, `2.`, etc.
- Superscripts and subscripts can be generated using: `^script^` and `~script~`, respectively. That is, `` `*r*^2^` `` and `` `CO~2~` `` produce: r^2 and CO_2.
- Footnotes can be created using the format: `` `^[footnote]` ``.
- Web hyperlinks can be created using: `` `[text](link)` ``. For instance, `` `[Amalgam of R](https://www.amalgamofr.org)` `` creates: Amalgam of R.

By default, RStudio shows **R** Markdown documents as raw source code. This format, however, can be changed to a more *presentational markup* (what you see is what you get) format by clicking on the Visual button that appears at the upper left hand side of RStudio Panel 1 (when an **R** Markdown document is open). The Visual interactive panel contains several interactive menus reminiscent of a word processor (Fig 2.10). These allow users to specify fonts, and to insert LaTeX equations (Section 2.10.2.1.3), section hierarchies, bulleted and numbered lists, and tables.

Figure 2.10: Additional RStudio menu options for an **R** Markdown document under the Visual viewing mode.

2.10.2.1.2 R Code in R Markdown Chunks The *knitr* **R** package facilitates report-building in both HTML and LaTeX → PDF formats, within the framework of the *rmarkdown* package

[23] This is an online document.

(Fig 2.6). Under *knitr*, **R** Markdown lines beginning ```{r } and ending ``` delimit an **R** code "chunk" to be potentially run in **R**.

Example 2.19.
For example, the chunk:

````
```{r }
mean(c(1,2,3))
```
````

would prompt *knitr* to: 1) show the code in an appropriate highlighted style, 2) run the code in **R** (i.e., take the mean of the three numbers), and 3) print the evaluation result into a new *output* chunk.

■

The chunk header, ```{r }, can be used to define additional options. These include the suppression of code evaluation, ```{r , eval = F}, suppression of code printing, ```{r , echo = F}, and/or elimination of the chunk from the after running, ```{r , include = F}. For a complete list of chunk options, run

```
str(knitr::opts_chunk$get())
```

If desired, global *knitr* options for chunks can be set using an initial **R** chunk or script (generally with the local chunk option `include = F`) that defines the components of `knitr::opts_chunk`.

Example 2.20.
For example, to suppress the default insertion of pound signs in lines preceding chunk evaluation output, throughout the entire knitted document, one could include the following initial chunk:

````
```{r, include = F }
knitr::opts_chunk$set(comment = NA)
```
````

■

Code chunks can be generated by going to **Code**>**Insert Chunk** or by using the RStudio shortcut Ctrl + Alt + I (Windows and Linux) or Cmd + Alt + I (Mac).

R code can also be invoked *inline* in a **R** Markdown document using the format:

```
`r some code`
```

For instance, I could seamlessly place three random numbers generated from a the continuous uniform distribution, $f(x) = UNIF(0, 1)$, inline into text using:

```
`r runif(3)`
```

Here I run an iteration using "hidden" inline **R** code: 0.04634, 0.96566, 0.14534.

2.10.2.1.3 Equations Inline equations for both **R** Markdown and Sweave (discussed below) can be specified under the LaTeX system, which uses dollar signs, `$`, to delimit equations. For instance, to obtain the inline equation: $P(\theta|y) = \frac{P(y|\theta)P(\theta)}{P(y)}$, i.e., Bayes theorem, I could type the LaTeX script into **R** Markdown:

```
$P(\theta|y) = \frac{P(y|\theta)P(\theta)}{P(y)}$
```

Display-style equations can be specified with two dollar signs, `$$`. For instance, `$$P(\theta|y) = \frac{P(y|\theta)P(\theta)}{P(y)}$$` results in:

$$P(\theta|y) = \frac{P(y|\theta)P(\theta)}{P(y)}$$

A cheatsheet for LaTeX equation writing can be found here.

2.10.2.1.4 Figures Probably the simplest way to place external figures into a document is by applying the function `knitr::include_graphics()` from within a chunk. The following **R** Markdown code would insert `Fig1.jpg` (contained in the working directory) into an **R** Markdown document.

````
```{r }
knitr::include_graphics("Fig1.jpg")
```
````

Figures can also be generated from the execution of **R** plotting functions (see Ch 6, 7). For instance, the following **R** Markdown code would place a simple **R**-generated scatterplot into the document:

````
```{r }
plot(1:10)
```
````

2.10.2.1.5 Tables **R** Markdown tables can be created by specifying the following format (outside of a chunk).

```
First Header  | Second Header
------------- | -------------
Content Cell  | Content Cell
Content Cell  | Content Cell
```

Tables, however, can also be be generated by executing **R** functions within chunks. I generally use the function `knitr::kable()` to create **R** Markdown → Pandoc → HTML tables because it is relatively simple to use, and allows straightforward tabling of **R** output.

Example 2.21.
Table 2.10, shows data from the `Loblolly` dataset in the package *datasets*. The data track the growth of loblolly pine trees (*Pinus taeda*) with respect to seed type and age. The function `head()`, nested in `kable()`, allows one to access the first or last components of an **R** data storage object. By default, `head()` returns the first six values (in this case, the first six dataframe rows).

```
knitr::kable(head(Loblolly))
```

■

I often use functions in the package *xtable* to build **R** Markdown → Pandoc → LaTeX → PDF tables. Under this approach, one could create Table 2.10 using:

```
print(xtable::xtable(head(Loblolly)))
```

This method would also require that one use the command `results = 'asis'` in the chunk options.

One can even call for different table approaches on the fly. For instance, I could use the command `eval = knitr::is_html_output()`, in the options of a Markdown chunk when using table code that optimizes HTML formatting, and use `eval = knitr::is_latex_output()` to create a table that optimizes LaTeX formatting.

Aside from `knitr::kable()` and *xtable*, there are many other **R** functions and packages that can be used to create **R** Markdown tables, particularly for HTML output. These include:

- The *kableExtra* (Zhu et al., 2022) package extends `knitr::kable()` by including styles for fonts, features for specific rows, columns, and cells, and straightforward merging and grouping of rows and/or columns. Most *kableExtra* features extend to both HTML and PDF formats.

Table 2.10: Loblolly pine data.

	height	age	Seed
1	4.51	3	301
15	10.89	5	301
29	28.72	10	301
43	41.74	15	301
57	52.70	20	301
71	60.92	25	301

- *DT* (Xie et al., 2024), a wrapper for HTML tables that uses the JavaScript (see Section 11.3) library *DataTables*. Among other features, *DT* allows straightforward implementation in interactive Shiny apps (Section 11.5).
- Like *DT*, the *reactable* package (Lin, 2023) creates flexible, interactive HTML embedded tables. As with *DT*, *reactable* tables add complications when those interactives are considered as conventional tables in **R** markdown, with captions and referable labels.

Xie et al. (2020) discuss several other alternatives.

Example 2.22.
An **R** Markdown (.rmd) skeleton file generated by RStudio (Figs 2.7-2.9) contains documentation text, interspersed with example **R** code in chunks. These been have been modified below to create a simple **R** markdown document for summarizing the `Loblolly` dataset (Fig 2.11).

````
---
title: "Loblolly Pine Analysis"
author: "Ken Aho"
date: "2024-10-10"
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

## Summary statistics
Here are some summary statistics for the lololly pine dataset.

```{r}
mean(Loblolly$height)
```

## Plots
Here is the relationship of height and age:

```{r, echo = F}
with(Loblolly, plot(age, height))
```
````

Figure 2.11: An **R** Markdown (.rmd) file with documentation text and interspersed **R** code in chunks.

Note the use of `echo = FALSE` in the final chunk to suppress printing of **R** code. A snapshot of the knitted HTML is shown in Fig 2.12.

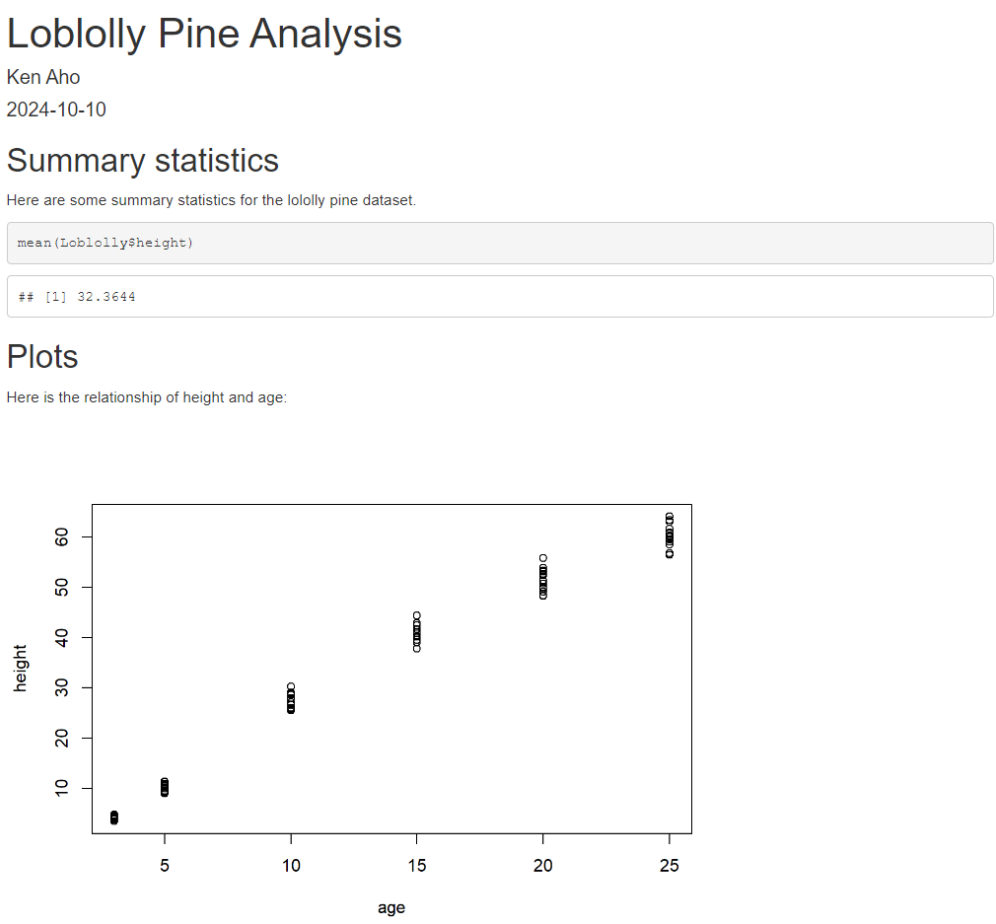


Figure 2.12: An HTML document knit from Markdown code in the previous figure. Note that code is displayed (by default) as well as executed.

■

2.10.2.1.6 bookdown A large number of useful auxiliary features are available for **R** Markdown, through the **R** package *bookdown* (Xie (2023)). These include an extended capacity for figure, table, and section numbering and referencing. The *bookdown* package is not included with RStudio, and will require installation using the code below. See Section **??** for more information on loading and installing packages.

```
install.packages("bookdown") # install bookdown package
```

To use *bookdown* we must modify the `output:` designation in the YAML header to have a *bookdown*-specific output. For instance,

```
output: bookdown::html_document2
```

to create an HTML document, or

```
output: bookdown::pdf_document2
```

to create a LaTeX → PDF document, or

```
output: bookdown::word_document2
```

to create an MS Word® document[24].

Numbering **R**-generated plots and tables in **R** in *bookdown* requires specification of a chunk label after the language reference, e.g., `r`, in the chunk generating the plot ot table. Importantly, many table generating **R** functions (e.g., `knitr::kable()` and `xtable::xtable()`, see below) also contain a `label` argument that allows referencing and numbering.

Example 2.23.
In the chunk header below I use the label `lobplot`. Note that a space is included after `r`. Captions can be specified in the chunk header using the chunk option `fig.cap` or `tab.cap` for figures and tables, respectively. The option `fig.cap` is used below:

```
```{r lobplot, echo=FALSE, fig.cap= "Loblolly pine height versus age."}
```

■

Cross-references within the text can be made using the syntax `\@ref(type:label)`, where `label` is the chunk label and `type` is the environment being referenced (e.g., `fig`, `tab`, or `eq`). For Example 2.23, we might want to type something like: "see Figure `\@ref(fig:lobplot)`." in some non-chunk component of the Markdown document.

Specification of a *bookdown* output format, will result in automated numbering of sections[25]. To turn this numbering off, one could modify the YAML output to be:

```
output:
  bookdown::html_document2:
    number_sections: false
```

The code indents shown above are important because YAML, like the language Python, uses *significant indentation*. To omit numbering for *certain* sections, one would retain the default bookdown output, and add `{-}` after the unnumbered section heading, e.g.,

```
# This section is unnumbered {-}
```

[24] Many other *bookdown* extensions to *rmarkdown* output types are possible including: `html_vignette2`, `ioslides_presentation2`, `slidy_presentation2`, `beamer_presentation2`, and `powerpoint_presentation2`. Further, there are *bookdown* formatting algorithms (e.g., `pdf_book()`, `bookdown::pdf_book()`, `epub_book()`, `bs4_book()`) that can be used to create entire books, with distinct, interacting chapters.

[25] Numbering is also possible in most **R** Markdown (non-*bookdown*) formats. Unlike *bookdown* this is not the default, and will require the specification: `number_sections: true`.
```

2.10.2.1.7 Additional Resources for R Markdown and Bookdown The Posit website houses a number of useful **R** Markdown guides, including this brief introduction. Thorough descriptions of **R** Markdown are provided in Xie et al. (2018a) and Xie et al. (2020). The latter text is currently available as an online resource. Thorough guidance for *bookdown* is provided in Xie (2016), which can be viewed as an open-source online document.

2.10.2.2 Sweave

Under the Sweave documentation approach, high quality PDF documents are generated from LaTeX .tex files, which in turn are created from Sweave .rnw files. A skeleton .rnw document can be generated in RStudio by going to **File**>**New File**>**R Sweave**[26].

2.10.2.2.1 R code in Sweave chunks Sweave chunks can be implemented using *knitr*-style formatting, or with formatting under the function `Sweave()` (Leisch, 2002). Switching between these formats in RStudio requires altering options in **Build**>**Configure Build Tools**>**Sweave**.

In RStudio, Sweave code chunks are initiated which `<<>>=`, which serves as a chunk header, and are closed with `@`.

Example 2.24.
Including the chunk below in an .rnw file would: 1) cause the **R** source code to be printed in a LaTeX-rendered PDF, 2) run the code in **R** (the mean of the three number would be calculated), and 3) print the evaluated result in the output PDF.

```
<<>>=
mean(c(1,2,3))
@
```

■

Chunk options in `Sweave()` are often similar to those in *knitr*, but are more limited (see `vignette("Sweave")`).

Example 2.25.
In Fig 2.13 I create an .rnw file, based on an RStudio skeleton, with text and analyses reflecting those used with **R** Markdown in Example 2.22. We note that instead of the Markdown YAML header, we now have lines in the preamble defining the type of desired document (e.g., article) and the LaTeX packages needed for document compilation (e.g., *amsmath*). All non-chunk text, including figure and table captions and cross-referencing must follow LaTeX guidelines.

[26]The document you are reading was either knitted from an **R** Markdown .rmd file (using *bookdown*) or a Sweave .rnw file, created in RStudio.

```
\documentclass{article}
\usepackage{amsmath}

\begin{document}
<<setup, include =FALSE>>=
knitr::opts_chunk$set(echo = TRUE)
@
\section{Summary Statistics}
Here are some summary statistics for the loblolly pine
dataset

<<>>=
mean(Loblolly$height)
@
\section{Plots}

<<echo = F>>=
with(Loblolly, plot(age, height))
@

\end{document}
```

Figure 2.13: A Sweave (.rnw) file with documentation text and interspersed code in chunks.

Fig 2.14 shows a snapshot of the result, following automated .rnw → knitr → LaTeX → .pdf compilation in RStudio.

1 Summary Statistics

Here are some summary statistics for the loblolly pine dataset

```
mean(Loblolly$height)

## [1] 32.3644
```

2 Plots

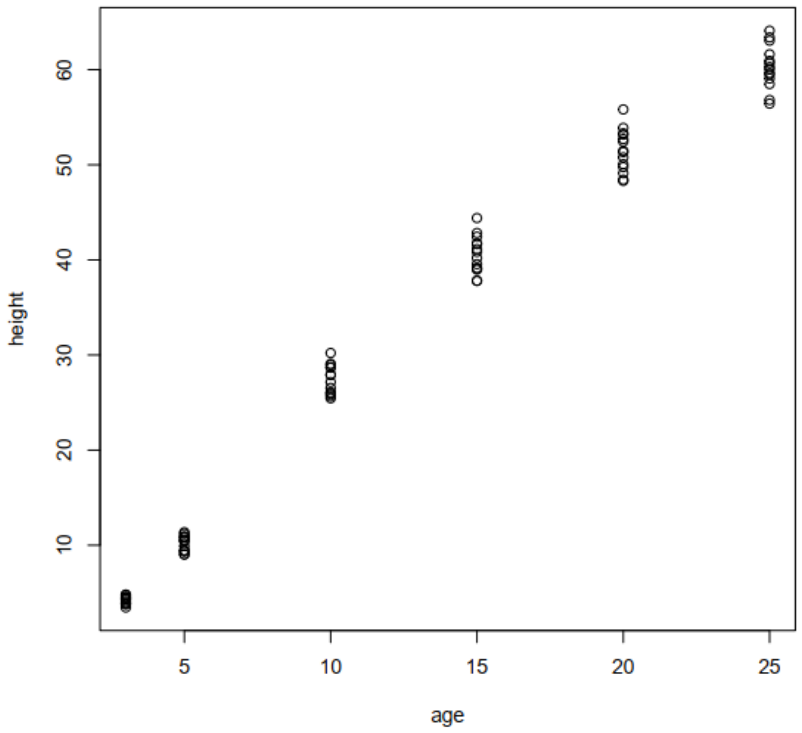


Figure 2.14: A .pdf document resulting from compilation of Sweave code in the previous figure.

■

2.10.2.3 Purl

R chunk code can be extracted from an .rmd or an .rnw file using the function `knitr::purl()`. For instance, assume that the **R** Markdown loblolly pine summary shown in Fig 2.11 is saved in the working directory under the name `lob.rmd`. Code from the file will be extracted to a script file called `lob.R`, located in the working directory, if one types:

```
purl("lob.rmd")
```

Exercises

1. Create an **R** Markdown document to contain your homework assignment. Modify the YAML header to allow numbering of figures and tables, but *not* sections. This will require use of the *bookdown* package (see Section 2.10.2.1.6). Install *bookdown* at the **R** console (not within a document chunk). To test the formatting, perform the following steps:
   (a) Create a section header called `Question 1` and a subsection header called (a). Under (a) type `"completed"`.
   (b) Under the subsection header (b), insert a chunk, and create a simple plot of points at the coordinates: $\{1, 1\}$, $\{2, 2\}$, $\{3, 3\}$, by typing the code: `plot(1:3)` in the chunk. Create a label for the chunk, and a create caption for the plot using the *knitr* chunk option, `fig.cap`.
   (c) Under the subsection header (c), create a cross reference for the plot from (b) (see Section 2.10.2.1.6).
   (d) Under the subsection header (d), write the equation, $y_i = \hat{\beta}_0 + \hat{\beta}_1 x_i + \hat{\varepsilon}_i$, using LaTeX. As noted earlier, a LaTeX equation cheatsheet can be found here.
   (e) Render (knit) the final document as either an .html file or a .doc file. *Include other assigned exercises for this Chapter as directed, using the general formatting approach given in Question 1.*

2. Perform the following operations.
   (a) Leave a note to yourself.
   (b) Create and examine an object called `x` that contains the numeric entries 1, 2, and 3.
   (c) Make a copy of `x` called `y`.
   (d) Show the class of `y`.
   (e) Show the base type of `y`.
   (f) Show the attributes of `y`.
   (g) List the current objects in your work session.
   (h) Identify your working directory.

3. Distinguish **R** expressions and assignments.

4. Sometimes **R** reports unexpected results for its classes and base types.
   (a) Create `x <- factor("a","a","b")` and show the class of `x`.
   (b) Type `?factor`. What is a `factor` in **R**?
   (c) Show the base type of `x`? Is this surprising? Why? Type `?integer`. What is an `integer` in **R**?

5. Solve the following mathematical operations using **R**.
   (a) $1 + 3/10 + 2$
   (b) $(1 + 3)/10 + 2$
   (c) $\left(4 \cdot \frac{(3-4)}{23}\right)^2$
   (d) $\log_2(3^{1/2})$

(e) $3\boldsymbol{x}^3 + 3\boldsymbol{x}^2 + 2$ where $\boldsymbol{x} = \{0, 1.5, 4, 6, 8, 10\}$
(f) $4(\boldsymbol{x} + \boldsymbol{y})$ where $\boldsymbol{x} = \{0, 1.5, 4, 6, 8\}$ and $\boldsymbol{y} = \{-2, 0.5, 3, 5, 8\}$.
(g) $\frac{d}{dx} \tan(x) 2.3 \cdot e^{3x}$
(h) $\frac{d^2}{dx^2} \frac{3}{4x^4}$
(i) $\int_3^{12} 24x + \ln(x) dx$
(j) $\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx$ (i.e., find the area under a standard normal pdf).
(k) $\int_{-\infty}^{\infty} \frac{x}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx$ (i.e., find $E(X)$ for a standard normal pdf).
(l) $\int_{-\infty}^{\infty} \frac{x^2}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx$ (i.e., find $E(X^2)$ for a standard normal pdf).
(m) Find the sum, cumulative sum, product, cumulative product, arithmetic mean, median and variance of the data `x = c(0, 1.5, 4, 6, 8, 10)`.

6. The velocity of the earth's rotation on its axis at the equator, E, is approximately 1674.364 km/h, or 1040.401 m/h[27]. We can calculate the velocity of the rotation of the earth at any latitude with the equation, $V = \cos(\text{latitude}^{\text{o}}) \times E$. Using **R**, simultaneously calculate rotational velocities for latitudes of 0,30,60, and 90 degrees north, or south, latitude (they will be the same). Remember, the function `cos()` assumes inputs are in radians, not degrees.

[27]The circumference of the earth at the equator is 40,075.02 km (24,901.5 mi). The earth completes one full rotation on its axis with respect to distant stars in 23 hours 56 minutes 4.091 seconds (a sidereal day). This means that in 24 hours, the earth rotates $\frac{24}{23+(56/60)+(4.091/60)/60} = 1.002738$ times. And this means that the velocity of the earth at the equator is $\frac{1.002738 \times 40075.02}{24} = 1674.364$ k·h^{-1}, or $0.621371 \times 1674.364 = 1040.401$ m·h^{-1}.

Chapter 3

Data Objects, Packages, and Datasets

"In God we trust. All others [must] have data."

- **Edwin R. Fisher**, *cancer pathologist*

3.1 Data Storage Objects

Depending on who you talk to[1], there are five primary types of data storage objects in **R**. These are: (atomic) vectors, matrices, arrays, dataframes, and lists[2].

3.1.1 Vectors

Historically (and confusingly), the conception of an **R** "*vector*" can be traced directly to the earliest object-class defined in the **S** language[3]. From this inception, an **R** vector is either an *atomic vector* –thus belonging one of the six atomic vector types: `logical`, `integer`, `numeric`, `complex`, `character` and `raw`– or an object of either class `expression` or class `list`. Objects of class `expression` generally contain mathematical calls or symbols that can be evaluated with the function `eval()` (see Section 2.9.5). Objects of class `list` are formally considered in Section 3.1.5.

Recall that **R** classes were introduced in Section 2.3.5 and fundamental classes were listed in Table 2.1. Because of their importance, the first eight classes shown in Table 2.1 classify vector types, and the first six specifically classify atomic vectors.

[1] For instance, Wickham (2019) views matrices, arrays, dataframes as vectors.

[2] Note that distinctions of these objects are not always clear or consistent. For instance, a `names` attribute can be given to elements of vectors and lists, and columns of dataframes. However, only names from dataframes and lists can be made visible using `attach`, or called using `$`.

[3] The class *vector* in **S3** was limited to objects that were index-able and subsettable by position (Chambers, 2008). That is, for the vector object `x`, `x[i]` would give the ith element of `x` (see Section 3.4). Although many non-vector **R** objects (e.g., matrices, and dataframes) are also index-able and subsettable, underlying components of those objects can nonetheless be viewed as atomic vectors (Fig 3.1).

3.1.1.1 Atomic vectors

Atomic vectors constitute "the essential bottom layer" of **R** data (Chambers, 2008). This characteristic is evident when viewing the relationship of atomic vectors to other data storage objects (Fig 3.1).

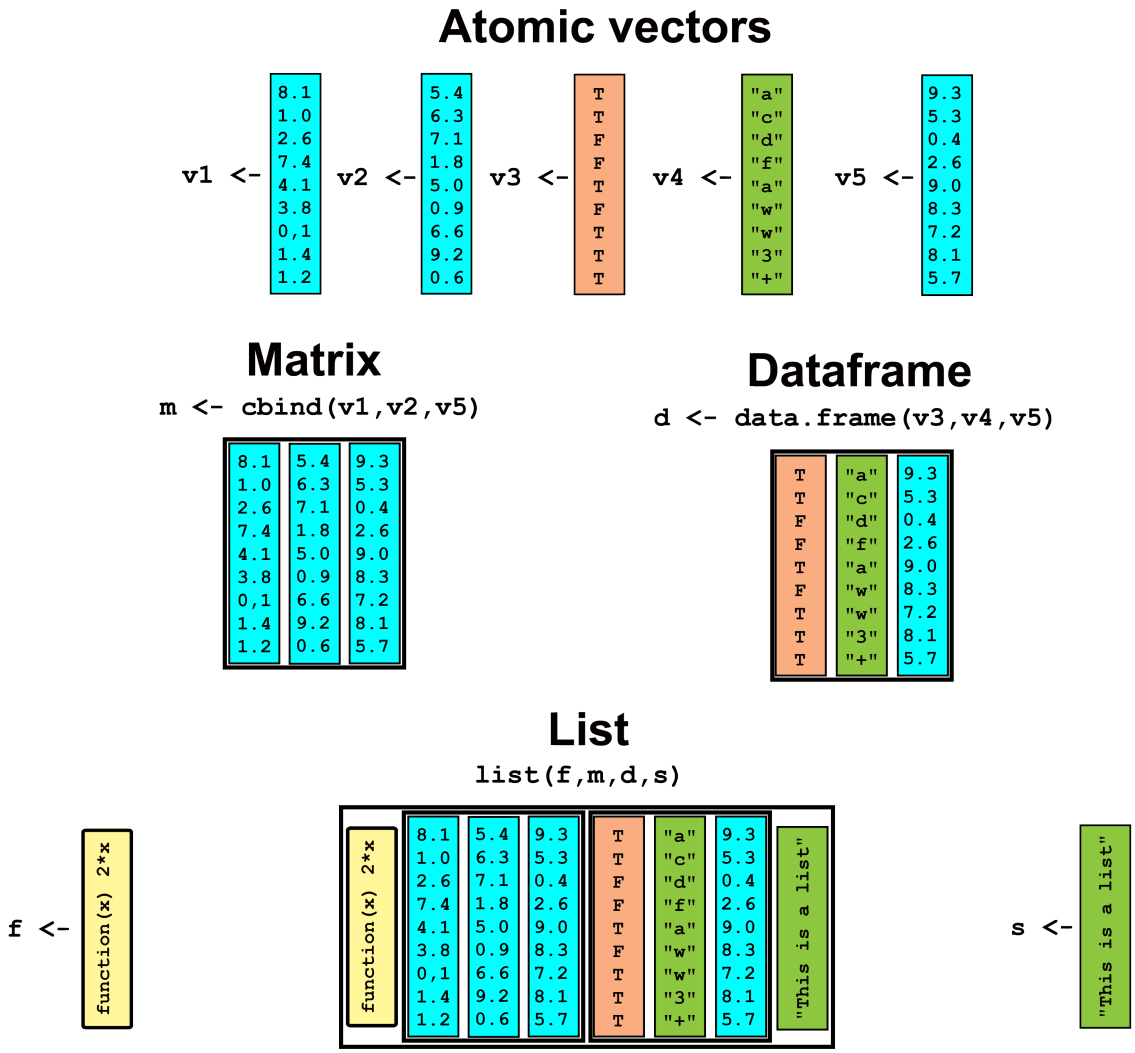


Figure 3.1: An example of **R** atomic vectors as building blocks for more complex data storage objects. Five atomic vectors are shown. Three are `numeric` (colored blue), one is `logical` (colored peach), and one is a `character` vector (light green). The `numeric` vectors are incorporated into a single matrix (which can have only one data storage mode), using `cbind()`. One of the `numeric` vectors, along with the `character` and `logical` vectors are incorporated into a dataframe (which can have multiple data storage modes). Finally, the matrix and dataframe are brought into a `list`, along with an anomolous function and character string.

Atomic vectors are simple data storage objects with a one data storage mode (base type). That is, a single atomic vector cannot contain data with both `logical`, and `character` base types (and classes), and a single atomic vector of class `numeric`, (which can have base types `integer` or `double`) cannot contain data from both of those base types.

We can create atomic vectors using the function `c()`.

Example 3.1.
Here is a `logical` atomic vector. Note that it only contains the entries `TRUE` and `FALSE`.

```
x <- c(TRUE, FALSE, TRUE)
class(x)
```

```
[1] "logical"
```

```
is.vector(x)
```

```
[1] TRUE
```

```
is.atomic(x)
```

```
[1] TRUE
```

Logical objects, and the testing of object class membership –demonstrated above with `is.logical(x)`, `is.vector(x)`, and `is.atomic(x)`– are formally introduced in Sections 3.2 and 3.3, respectively.

■

Example 3.2.
Here is an atomic vector of *character strings*. That is, a *character vector*[4]. Recall that the individual strings require quote `" "` or `' '` delimitation.

```
x <- c("string1", "string2")
class(x)
```

```
[1] "character"
```

```
is.vector(x)
```

```
[1] TRUE
```

```
is.atomic(x)
```

```
[1] TRUE
```

[4] **R** frequently uses *character vectors*, i.e., `vec <- c("a", "b", "c")`. Each entry in `vec` would be considered as a character string.

■

Example 3.3.
We can explicitly define a number, `x`, to be an an integer with the script `xL`. Thus, the code below specifies an atomic `integer` vector:

```
x <- c(1L, 3L, 7L)
class(x)
```

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

```
typeof(x)
```

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

```
is.vector(x)
```

```
[1] TRUE
```

```
is.atomic(x)
```

```
[1] TRUE
```

■

Example 3.4.
Here is a `numeric` atomic vector stored with double precision:

```
x <- c(1, 2, 3)
class(x)
```

```
[1] "numeric"
```

```
typeof(x)
```

```
[1] "double"
```

```
is.vector(x)
```

```
[1] TRUE
```

```
is.atomic(x)
```

```
[1] TRUE
```

■

Atomic vectors have order and length, but no dimension. This is clearly different from the linear algebra conception of a vector. Specifically, in linear algebra, a row vector with n elements has dimension $1 \times n$ (1 row and n columns), whereas a column vector has dimension $n \times 1$.

Example 3.5.
Consider the `numeric` atomic vector from the previous example (Example 3.4).

```
length(x)
```

```
[1] 3
```

```
dim(x)
```

```
NULL
```

The function `as.matrix(x)` (see Section 3.3.4) can be used to coerce `x` to have a matrix structure with dimension 3×1 (3 rows and 1 column). Thus, in **R** a matrix has dimension, but a vector does not.

```
dim(as.matrix(x))
```

```
[1] 3 1
```

■

Any single value object of class `numeric`, `complex`, `integer`, `logical`, or `character` is an atomic vector.

Example 3.6.
Complex numbers in **R** are defined by codifying their real parts conventionally, and their imaginary parts with `i`. Recall that the square of an imaginary number bi is $-b^2$.

```
x <- -2 + 1i^2 # -2 is real
class(x)
```

```
[1] "complex"
```

```
typeof(x)
```

```
[1] "complex"
```

```
is.vector(x)
```

```
[1] TRUE
```

■

We can add a `names` attribute to vector elements.

Example 3.7.
For example:

```
x <- c(a = 1, b = 2, c = 3)
```

```
x
```

```
a b c
1 2 3
```

Recall that the function `attributes()` can be used to list an object's attributes:

```
attributes(x)
```

```
$names
[1] "a" "b" "c"
```

The function `attr()` can be used to obtain (or set) values associated with a particular attribute.

```
attr(x, "names") # or  names(x)
```

```
[1] "a" "b" "c"
```

■

Importantly, when an element-wise operation is applied to two unequal length vectors, **R** will generate a warning and automatically recycle elements of the shorter vector.

Example 3.8.
For example,

```
c(1, 2, 3) + c(1, 0, 4, 5, 13)
```

```
Warning in c(1, 2, 3) + c(1, 0, 4, 5, 13): longer object length is not a
multiple of shorter object length
```

```
[1]  2  2  7  6 15
```

In this case, the result of the addition of the two vectors is: $1+1, 2+0, 3+4, 1+5$, and $2+13$. Thus, the first two elements in the first object are recycled in the vector-wise addition.

■

3.1.2 Matrices

Matrices are two-dimensional (row and column) data structures whose elements must all have the same data storage mode (typically `"double"`) (Fig 3.1).

The function `matrix()` can be used to create matrices.

Example 3.9.

Consider the following examples:

```
A <- matrix(ncol = 2, nrow = 2, data = c(1, 2, 3, 2))
A
```

```
     [,1] [,2]
[1,]    1    3
[2,]    2    2
```

Note that `matrix()` assumes that data are entered "by column." That is, the first two entries in the `data` argument are placed in column one, and the last two entries are placed in column two. One can enter data "by row" by adding the argument `byrow = TRUE`.

```
B <- matrix(ncol = 2, nrow = 2, data = c(1, 2, 3, 2), byrow = TRUE)
B
```

```
     [,1] [,2]
[1,]    1    2
[2,]    3    2
```

■

3.1.2.1 Matrix algebra

Matrix algebra operations can be applied directly to **R** matrices (Table 3.1). For matrices with the same dimension, the `+` and `-` operators allow elementwise addition and subtraction of matrices, and the `*` operator serves as the elementwise *Hadamard product* operator. A `non-conformable arrays` error will be given when performing elementwise addition, subtraction or multiplication of two matrices with unequal dimensions, or when the number of columns in the first matrix does not equal the number of rows in the second matrix in standard matrix multiplication using `%*%`. Recall that addition, subtraction or multiplication of two unequal length vectors will result in recycling of elements of the shorter vector (Section 3.1.1.1).

More complex matrix analyses are also possible, including spectral decomposition (function `eigen()`), and single value, QR, and Cholesky decompositions (the functions `svd()`, `qr()`, `chol()`, respectively).

Example 3.10.
In Example 3.9, matrix `A` has the form:

$$\boldsymbol{A} = \begin{bmatrix} 1 & 3 \\ 2 & 2 \end{bmatrix}.$$

Consider the operations:

Table 3.1: Simple matrix algebra operations in **R**. In all operations $\boldsymbol{A}$ (and correspondingly, `A`) is a matrix, and $\boldsymbol{a}$ (and correspondingly, `a`) is a vector.

Operator	Operation	To.find.	We.type.
`t()`	Matrix transpose	$\boldsymbol{A}^T$	`t(A)`
`+, -`	Addition or subtraction	$\boldsymbol{A} + \boldsymbol{A}$	`A+A`
`*`	Hadamard product	$\boldsymbol{A} \odot \boldsymbol{A}$	`A*A`
`outer()`	Outer product	$\boldsymbol{a} \otimes \boldsymbol{a}$	`outer(a,a)`
`%*%`	Matrix multiplication	$\boldsymbol{A} \cdot \boldsymbol{A}$	`A%*%A`
`det()`	Determinant	$\det(\boldsymbol{A})$	`det(a)`
`solve()`	Matrix inverse	$\boldsymbol{A}^{-1}$	`solve(A)`

```
t(A)
```

```
     [,1] [,2]
[1,]    1    2
[2,]    3    2
```

```
A %*% A
```

```
     [,1] [,2]
[1,]    7    9
[2,]    6   10
```

```
det(A)
```

```
[1] -4
```

```
solve(A)
```

```
     [,1]  [,2]
[1,] -0.5  0.75
[2,]  0.5 -0.25
```

■

We can use the function `cbind()` to combine vectors into matrix columns,

```
a <- c(1, 2, 3); b <- c(2, 3, 4)
cbind(a, b)
```

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

and use the function `rbind()` to combine vectors into matrix rows.

```
rbind(a,b)
```

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

3.1.3 Arrays

Arrays are one, two dimensional (matrix), or three or more dimensional data structures whose elements contain a single type of data. Thus, while all matrices are arrays, not all arrays are matrices.

```
class(A)
```

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

As with matrices, elements in arrays can have only one data storage mode.

```
typeof(A) # base type (data storage mode)
```

```
[1] "double"
```

The function `array()` can be used to create arrays. The first argument in `array()` defines the data. The second argument is a vector that defines both the number of dimensions (this will be the length of the vector), and the number of levels in each dimension (numbers in dimension elements).

Example 3.11.
Here is a $2 \times 2 \times 2$ array:

```
some.data <- c(1, 2, 3, 4, 5, 6, 7, 8)
B <- array(some.data, c(2, 2, 2))
B
```

```
, , 1

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

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

```
class(B)
```

```
[1] "array"
```

■

3.1.4 Dataframes

Like matrices, *dataframes* are two-dimensional structures. Dataframe columns, however, can have different data storage modes (e.g., `double` *and* `character`) (Fig 3.1). The function `data.frame()` can be used to create dataframes.

```
df <- data.frame(numeric = c(1, 2, 3), non.numeric = c("a", "b", "c"))
df
```

```
  numeric non.numeric
1       1           a
2       2           b
3       3           c
```

```
class(df)
```

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

Because of the possibility of different data storage modes for distinct columns, the data storage mode of a dataframe is `"list"` (see Section 3.1.5, below). Specifically, a dataframe is a two dimensional list, whose storage *elements* are columns.

```
typeof(df)
```

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

A `names` attribute will exist for each dataframe column[5].

Example 3.12.
Consider the dataframe `df`:

```
names(df)
```

```
[1] "numeric"     "non.numeric"
```

The `$` operator allows access to dataframe column names.

[5] Arrays (including matrices) will generally be numeric storage structures, and cannot have a `names` attribute. Instead, row names and column names can be applied using the functions `row.names()` and `col.names()`. These, however, cannot be made visible to search paths with `attach()` or called with `$`.

```
df$non.numeric
```

```
[1] "a" "b" "c"
```

The $ operator allows *partial matches* when specifying dataframe names:

```
df$non
```

```
[1] "a" "b" "c"
```

■

The function `attach()` allows **R** to recognize column names of a dataframe as global variables.

Example 3.13.

Following attachment of `df`, the column `non.numeric` can be directly accessed:

```
attach(df)
non.numeric
```

```
[1] "a" "b" "c"
```

The function `detach()` is the programming inverse of `attach()`.

```
detach(df)
non.numeric
```

```
Error: object 'non.numeric' not found
```

■

The functions `rm()` and `remove()` will entirely remove any **R**-object –including a vector, matrix, or dataframe– from a session. To remove *all* objects from the workspace one can use `rm(list=ls())` or (in RStudio) the "broom" button in the environments and history panel[6].

A safer alternative to `attach()` is the function `with()`. Using `with()` eliminates concerns about multiple variables with the same name becoming mixed up in functions. This is because the variable names for a dataframe specified in `with()` will not be permanently attached in an **R**-session.

Example 3.14.
Despite the removal of the `df` column `non.numeric` from the **R** search path in the second part of Example 3.13, the column can be called directly when using `with()`.

[6]All objects from a specific class can also be removed from a workspace. For example, to remove all dataframes, from a work session one could use: `rm(list=ls(all=TRUE)[sapply(mget(ls(all=TRUE)), class) == "data.frame"])`

```
with(df, non.numeric)
```

```
[1] "a" "b" "c"
```

■

3.1.5 Lists

Lists are often used to contain miscellaneous associated objects. Like dataframes, lists need not use a single data storage mode. Unlike dataframes, however, lists can include objects that do not have the same dimensionality, including functions, character strings, multiple matrices and dataframes with varying dimensionality, and even other lists (Fig 3.1). The function `list()` can be used to create lists.

Example 3.15.
Here we explore the characteristics of a simple `list`.

```
ldata <- list(first = c(1, 2, 3), second = "this.is.a.list")
ldata
```

```
$first
[1] 1 2 3

$second
[1] "this.is.a.list"
```

```
class(ldata)
```

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

```
typeof(ldata)
```

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

Note that lists are vectors:

```
is.vector(ldata)
```

```
[1] TRUE
```

Although they are not atomic vectors:

```
is.atomic(ldata)
```

```
[1] FALSE
```

■

Reflecting dataframes, objects in lists can be called with partial matching using the $ operator. Here is the character string `second` from `ldata`.

```
ldata$sec
```

```
[1] "this.is.a.list"
```

The function `str` attempts to display the *internal structure* of an **R** object. It is extremely useful for succinctly displaying the contents of complex objects like lists.

Example 3.16.
For `ldata1` we have:

```
str(ldata)
```

```
List of 2
 $ first : num [1:3] 1 2 3
 $ second: chr "this.is.a.list"
```

The output confirms that `ldata` is a list containing two objects: a sequence of numbers from 1 to 3, and a character string.

■

The underlying *vector structure* of dataframes and lists (Fig 3.1) results in a potential nested configuration of base types. In particular, although all **R** objects must have a single overarching base type, dataframe and list subcomponents may contain data with distinct base types.

Example 3.17.
For instance,

```
typeof(df)
```

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

```
typeof(ldata)
```

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

```
typeof(df$num);
```

```
[1] "double"
```

```
typeof(ldata$sec)
```

```
[1] "character"
```

■

The function `do.call()` is useful for large scale manipulations of data storage objects, particularly lists.

Example 3.18.
For example, what if you had a list containing multiple dataframes with the same column names that you wanted to bind together?

```
ldata2 <- list(df1 = data.frame(lo.temp = c(-1,3,5),
                                high.temp = c(78, 67, 90)),
               df2 = data.frame(lo.temp = c(-4,3,7),
                                high.temp = c(75, 87, 80)),
               df3 = data.frame(lo.temp = c(-0,2),
                                high.temp = c(70, 80)))
```

You could do something like:

```
do.call("rbind",ldata2)
```

```
      lo.temp high.temp
df1.1      -1        78
df1.2       3        67
df1.3       5        90
df2.1      -4        75
df2.2       3        87
df2.3       7        80
df3.1       0        70
df3.2       2        80
```

Or what if I wanted to replicate the `df3` dataframe from `ldata2` above, by binding it onto the bottom of itself three times? I could do something like:

```
do.call("rbind", replicate(3, ldata2$df3, simplify = FALSE))
```

```
  lo.temp high.temp
1       0        70
2       2        80
3       0        70
4       2        80
5       0        70
6       2        80
```

Note the use of the function `replicate()`.

■

3.2 Boolean Operations

Computer operations that dichotomously classify statements are called *logical* or *Boolean*. In **R**, a Boolean procedure will always return one of the values `TRUE` or `FALSE`. **R** logical operators are listed in Table 3.2.

Table 3.2: Logical (Boolean) operators in **R**; `x`, `y`, and `z` in columns three and four are **R** objects.

Operator	Operation	To ask:	We type:
`>`	$>$	Is x greater than y?	`x > y`
`>=`	$\geq$	Is x greater than or equal to y?	`x >= y`
`<`	$<$	Is x less than y?	`x < y`
`<=`	$\leq$	Is x less than or equal to y	`x <= y`
`==`	$=$	Is x equal to y?	`x == y`
`!=`	$\neq$	Is x not equal to y?	`x != y`
`&`	*and*	Do x and y equal z?	`x & y == z`
`&&`	*and* (control flow)	Do x and y equal z?	`x && y == z`
`|`	*or*	Do x or y equal z?	`x | y == z`
`||`	*or* (control flow)	Do x or y equal z?	`x || y == z`

Note that there are two ways to specify "and" (`&` and `&&`), and two ways to specify "or" (`|` and `||`). The longer forms of "and" and "or" evaluate queries from left to right, stopping when a result is determined. Thus, this form is more appropriate for programming control-flow operations.

Example 3.19.
For demonstration purposes, here is a simple dataframe:

```
dframe <- data.frame(
Age = c(18,22,23,21,22,19,18,18,19,21),
Sex = c("M","M","M","M","M","F","F","F","F","F"),
Weight_kg = c(63.5,77.1,86.1,81.6,70.3,49.8,54.4,59.0,65,69)
)

dframe
```

```
   Age Sex Weight_kg
1   18   M      63.5
2   22   M      77.1
3   23   M      86.1
4   21   M      81.6
5   22   M      70.3
6   19   F      49.8
7   18   F      54.4
8   18   F      59.0
```

```
9   19   F      65.0
10  21   F      69.0
```

The **R** logical operator for equals is == (Table 3.2). Thus, to identify Age outcomes equal to 21 we type:

```
with(dframe, Age == 21)
```

```
 [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE
```

The argument Age == 21 has base type logical.

```
typeof(dframe$Age == 21)
```

```
[1] "logical"
```

The *unary operator* for "not" is ! (Table 3.2). Thus, to identify Age outcomes not equal to 21 we could type:

```
with(dframe, Age != 21)
```

```
 [1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
```

Multiple Boolean queries can be made. Here we identify Age data less than 19, or equal to 21.

```
with(dframe, Age < 19 | Age == 21)
```

```
 [1]  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE  TRUE
```

Queries can involve multiple variables. For instance, here we identify males less than or equal to 21 years old that weigh less than 80 kg.

```
with(dframe, Age <= 21 & Sex == "M", weight < 80)
```

```
 [1]  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

■

3.3 Testing and Coercing Classes

3.3.1 Testing Classes

As demonstrated in Section 3.1, functions exist to logically *test* for object membership to major **R** classes. These functions generally begin with an is. prefix and include: is.atomic(), is.vector(), is.matrix(), is.array(), is.list(), is.factor(), is.double(), is.integer() is.numeric(), is.character(), and many others.

The Boolean function `is.numeric()` can be used to test if an object or an object's components behave like numbers[7].

Example 3.20.
For example,

```
x <- c(23, 34, 10)
is.numeric(x)
```

```
[1] TRUE
```

```
is.double(x)
```

```
[1] TRUE
```

Thus, `x` contains numbers stored with double precision.

■

Data objects with categorical entries can be created using the function `factor()`. In statistics the term "factor" refers to a categorical variable whose categories (factor levels) are likely replicated as treatments in an experimental design.

Example 3.21.
For example,

```
x <- factor(c(1,2,3,4))
x
```

```
[1] 1 2 3 4
Levels: 1 2 3 4
```

```
is.factor(x)
```

```
[1] TRUE
```

■

The **R** class factor streamlines many analytical processes, including summarization of a quantitative variable with respect to a factor and specifying interactions of two or more factors.

Example 3.22.
Here we see the interaction of levels in `x` with levels in another factor, `y`.

[7] The `numeric` class is often used as an alias for class `double`. In fact, `as.numeric()` is identical to `as.double()`, and `numeric()` is identical to `double()` (Wickham, 2019).

```
y <- factor(c("a","b","c","d"))
interaction(x, y)
```

```
[1] 1.a 2.b 3.c 4.d
16 Levels: 1.a 2.a 3.a 4.a 1.b 2.b 3.b 4.b 1.c 2.c 3.c 4.c 1.d 2.d ... 4.d
```

Sixteen interactions are possible, although only four actually occur when simultaneously considering `x` and `y`.

■

To decrease memory usage[8], objects of class factor have an unexpected base type:

```
typeof(x)
```

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

Despite this designation, and the fact that categories in `x` are distinguished using numbers, the entries in `x` do not have a numerical meaning and cannot be evaluated mathematically.

```
is.numeric(x)
```

```
[1] FALSE
```

```
x + 5
```

```
Warning in Ops.factor(x, 5): '+' not meaningful for factors
```

```
[1] NA NA NA NA
```

Occasionally an ordering of categorical levels is desirable. For instance, assume that we wish to apply three different imprecise temperature treatments `"low"`, `"med"` and `"high"` in an experiment with six experimental units. While we do not know the exact temperatures of these levels, we know that `"med"` is hotter than `"low"` and `"high"` is hotter than `"med"`. To provide this categorical ordering we can use `factor(data, ordered = TRUE)` or the function `ordered()`.

Example 3.23.

```
x <- factor(c("med","low","high","high","med","low"),
            levels = c("low","med","high"),
            ordered = TRUE)
x
```

[8] All `numeric` objects in **R** are stored with double-precision, and will require two adjacent locations in computer memory (see Ch 12). Numeric objects coerced to be integers (with `as.intger()`) will be stored with double precision, although one of the storage locations will not be used. As a result, integers are not conventional double precision data.

```
[1] med  low  high high med  low
Levels: low < med < high
```

```
is.factor(x)
```

```
[1] TRUE
```

```
is.ordered(x)
```

```
[1] TRUE
```

The `levels` argument in `factor()` specifies the correct ordering of levels.

■

3.3.2 `ifelse()`

The function `ifelse()` can be applied to atomic vectors or one dimensional arrays (e.g., rows or columns) to evaluate a logical argument and provide particular outcomes if the argument is `TRUE` or `FALSE`. The function requires three arguments.

- The first argument, `test`, gives the logical test to be evaluated.
- The second argument, `yes`, provides the output if the test is true.
- The third argument, `no`, provides the output if the test is false.

For instance:

```
ifelse(dframe$Age < 20, "Young", "Not so young")
```

```
 [1] "Young"        "Not so young" "Not so young" "Not so young"
 [5] "Not so young" "Young"        "Young"        "Young"
 [9] "Young"        "Not so young"
```

3.3.3 `if`, `else`, `any`, and `all`

A more generalized approach to providing a condition and then defining the consequences (often used in functions) uses the commands `if` and `else`, potentially in combination with the functions `any()` and `all()`. For instance:

```
if(any(dframe$Age < 20)) "Young" else "Not so Young"
```

```
[1] "Young"
```

and

```
if(all(dframe$Age < 20))"Young" else "Not so Young"
```

```
[1] "Not so Young"
```

3.3.4 Coercion

Objects can be switched from one class to another using *coercion* functions that begin with an `as.` prefix[9]. Analogues to the testing (`.is`) functions listed above are: `as.matrix()`, `as.array()`, `as.list()`, `as.factor()`, `as.double()`, `as.integer()`, `as.numeric()`, and `as.character()`.

Example 3.24.
For instance, a non-factor object can be coerced to have class factor with the function `as.factor()`.

```
x <- c(23, 34, 10)
is.factor(x)
```

```
[1] FALSE
```

```
y <- as.factor(x)
is.factor(y)
```

```
[1] TRUE
```

■

Coercion may result in removal and addition of attributes.

Example 3.25.
For instance, conversion from an atomic vector to a matrix below results in the loss of the vector names attribute.

```
x <- c(eulers_num = exp(1), log_exp = log(exp(1)), pi = pi)
x
```

```
eulers_num    log_exp         pi
    2.7183     1.0000     3.1416
```

```
names(x)
```

```
[1] "eulers_num" "log_exp"    "pi"
```

```
y <- as.matrix(x)
names(y)
```

```
NULL
```

■

[9] Coercion can also be implemented using class generating functions described earlier. For instance, `data.frame(matrix(nrow = 2, data = rnorm(4)))` converts a 2×2 matrix into an equivalent dataframe.

Coercion may result in very unexpected outcomes.

Example 3.26.
Here NAs (Section 3.3.5) result when attempting to coerce a object with apparent mixed storage modes to class numeric.

```
x <- c("a", "b", 10)
as.numeric(x)
```

```
Warning: NAs introduced by coercion
```

```
[1] NA NA 10
```

■

Combining **R** objects with different base types results in coercion to a single base type. See Chambers (2008) for coercion rules.

Example 3.27.
For example, combining a numeric vector with base type `double` and a `character` vector, results in an object with class and base type `character`.

```
x <- c(1.2, 3.2, 1.5)
y <- c("a", "b", "c")
z <- c(x, y)
z
```

```
[1] "1.2" "3.2" "1.5" "a"   "b"   "c"
```

```
class(z); typeof(z)
```

```
[1] "character"
```

```
[1] "character"
```

and combining a `numeric` vector with base type `double`, and a `numeric` vector with base type `integer` results in a `numeric` vector with base type `double`.

```
y <- c(1L, 2L, 3L)
z <- c(x, y)
z
```

```
[1] 1.2 3.2 1.5 1.0 2.0 3.0
```

```
class(z); typeof(z)
```

```
[1] "numeric"
```

```
[1] "double"
```

■

3.3.5 NA

R identifies *missing values* (empty cells) as NA, which means "not available." Hence, the **R** function to identify missing values is is.na().

Example 3.28.
For example:

```
x <- c(2, 3, 1, 2, NA, 3, 2)
is.na(x)
```

```
[1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
```

Conversely, to identify outcomes that are not missing, I would use the "not" operator to specify !is.na().

```
!is.na(x)
```

```
[1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE
```

■

There are a number of **R** functions to get rid of missing values. These include na.omit().

Example 3.29.
For example:

```
na.omit(x)
```

```
[1] 2 3 1 2 3 2
attr(,"na.action")
[1] 5
attr(,"class")
[1] "omit"
```

We see that **R** dropped the missing observation and then told us which observation was omitted (observation number 5).

■

Functions in **R** often, but not always, have built-in capacities to handle missing data, for instance, by calling na.omit().

Example 3.30.
Consider the following dataframe which provides plant percent cover data for four plant species at two sites. Plant species are identified with four letter codes, consisting of the first two letters of the Linnaean genus and species names.

```
field.data <- data.frame(ACMI = c(12, 13), ELSC = c(0, 4), CAEL = c(NA, 2),
                         CAPA = c(20, 30), TACE = c(0, 2))
row.names(field.data) <- c("site1", "site2")

field.data
```

```
      ACMI ELSC CAEL CAPA TACE
site1   12    0   NA   20    0
site2   13    4    2   30    2
```

The function `complete.cases()` checks for completeness of data, by row, in a data array.

```
complete.cases(field.data)
```

```
[1] FALSE  TRUE
```

If `na.omit()` is applied in this context, the entire row containing the missing observation will be dropped.

```
na.omit(field.data)
```

```
      ACMI ELSC CAEL CAPA TACE
site2   13    4    2   30    2
```

Unfortunately, this means that information about the other four species at site one will lost. Thus, it is generally more rational to remove `NA` values while retaining non-missing values. For instance, many statistical functions have to capacity to base summaries on non-`NA` data.

```
mean(as.numeric(field.data[1,]), na.rm = TRUE)
```

```
[1] 8
```

■

3.3.6 NaN

The designation `NaN` is associated with the current conventions of the IEEE 754-2008 arithmetic used by **R**. It means "not a number." Mathematical operations which produce `NaN` include:

```
0/0
```

```
[1] NaN
```

```
Inf-Inf
```

```
[1] NaN
```

```
sin(Inf)
```

```
Warning in sin(Inf): NaNs produced
```

```
[1] NaN
```

3.3.7 NULL

In object oriented programming, a *null object* has no referenced value or has a defined neutral behavior (Wikipedia, 2023c). Occasionally one may wish to specify that an **R** object is NULL. For example, a NULL object can be included as an argument in a function without requiring that it has a particular value or meaning.

Example 3.31.
It is straightforward to designate an object as NULL.

```
x <- NULL
```

The class and base type of x are NULL:

```
class(x)
```

```
[1] "NULL"
```

```
typeof(x)
```

```
[1] "NULL"
```

■

It should be emphasized that **R**-objects or elements within objects that are NA, NaN or NULL cannot be identified with the Boolean operators == or !=.

Example 3.32.
For instance:

```
x == NULL
```

```
logical(0)
```

```
y <- NA
```

```
y == NA
```

```
[1] NA
```

■

Instead, one should use `is.na()`, `is.nan()` or `is.null()` to identify `NA`, `NaN` or `NULL` components, respectively.

Example 3.33.
That is:

```
is.null(x)
```

```
[1] TRUE
```

```
!is.null(x)
```

```
[1] FALSE
```

```
is.na(y)
```

```
[1] TRUE
```

```
!is.na(y)
```

```
[1] FALSE
```

■

3.4 Accessing and Subsetting Data With []

One can subset data storage objects using *square bracket operators*, i.e., `[]`, along with a variety of functions[10]. Because of their simplicity, I focus on square brackets for subsetting here. Gaining skills with square brackets will greatly enhance your ability to manipulate datasets in **R**.

As toy datasets, here are an atomic vector (with a names attribute), a matrix, a three dimensional array, a dataframe, and a list:

```
vdat <- c(a = 1, b = 2, c = 3)
vdat
```

```
a b c
1 2 3
```

```
mdat <- matrix(ncol = 2, nrow = 2, data = c(1, 2, 3, 4))
mdat
```

```
     [,1] [,2]
[1,]    1    3
```

[10] For instance, `subset()`, `split()`, and `dplyr::filter()`.

```
[2,]    2    4
```

```
adat <- array(dim = c(2, 2, 2), data = c(1, 2, 3, 4, 5, 6, 7, 8))
adat
```

```
, , 1

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

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

```
ddat <- data.frame(numeric = c(1, 2, 3), non.numeric = c("a", "b", "c"))
ddat
```

```
  numeric non.numeric
1       1           a
2       2           b
3       3           c
```

```
ldat <- list(element1 = c(1, 2, 3), element2 = "this.is.a.list")
ldat
```

```
$element1
[1] 1 2 3

$element2
[1] "this.is.a.list"
```

To obtain the ith canonical component from an atomic vector, matrix, array, dataframe or list named `foo` we would specify `foo[i]`.

Example 3.34.
For instance, here is the first component of our toy data objects:

```
vdat[1]
```

```
a
1
```

```
mdat[1]
```

```
[1] 1
```

```
adat[1]
```

```
[1] 1
```

```
ddat[1]
```

```
  numeric
1       1
2       2
3       3
```

```
ldat[1]
```

```
$element1
[1] 1 2 3
```

Importantly, we see that dataframes and lists view their ith canonical component as the ith column and the ith `list` element, respectively.

■

We can also apply double square brackets, i.e., `[[]]` to list-type objects, i.e., atomic vectors and explicit lists, with similar results. Note, however, that the data subsets will now be missing their name attributes.

Example 3.35.
For example:

```
vdat[[1]]
```

```
[1] 1
```

```
ldat[[1]]
```

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

■

If a data storage object has a names attribute, then a name can be placed in square brackets to obtain corresponding data.

Example 3.36.
For example:

```
ddat["numeric"]
```

```
  numeric
1       1
2       2
3       3
```

The advantage of square brackets over `$` in this an application is that several components can be specified simultaneously using the former approach:

```
ddat[c("non.numeric","numeric")]
```

```
  non.numeric numeric
1           a       1
2           b       2
3           c       3
```

■

If *foo* has a row × column structure, i.e., a matrix, array, or dataframe, we could obtain the ith column from *foo* using `foo[,i]` (or `foo[[i]]`) and the jth row from *foo* using `foo[j,]`.

Example 3.37.
For example, here is the second column from `mdat`,

```
mdat[,2]
```

```
[1] 3 4
```

and the first row from `ddat`.

```
ddat[1,]
```

```
  numeric non.numeric
1       1           a
```

■

The element from `foo` corresponding to row *j* and column *i* can be accessed using: `foo[j, i]`, or `foo[,i][j]`, or `foo[j,][i]`.

Example 3.38.
For example:

```
mdat[1,2]; mdat[,2][1]; mdat[1,][2] # 1st element from 2nd column
```

```
[1] 3
```

```
[1] 3
```

```
[1] 3
```

■

Arrays may require more than two indices. For instance, for a three dimensional array, `foo`, the specification `foo[,j,i]` will return the entirety of the jth column in the ith component of the outermost dimension of `foo`, whereas `foo[k,j,i]` will return the kth element from the jth column in the ith component of the outermost dimension of `foo`.

Example 3.39.
For example:

```
adat[,2,1]
```

```
[1] 3 4
```

```
adat[1,2,1]
```

```
[1] 3
```

```
adat[2,2,1]
```

```
[1] 4
```

■

Ranges or particular subsets of elements from a data storage object can also be selected.

Example 3.40.
For instance, here I access rows two and three of `ddat`:

```
ddat[2:3,] # note the position of the comma
```

```
  numeric non.numeric
2       2           b
3       3           c
```

■

I can `drop` data object components by using negative integers in square brackets.

Example 3.41.
Here I obtain an identical result to the example above by dropping row one from `ddat`:

```
ddat[-1,] # drop row one
```

```
  numeric non.numeric
2       2           b
3       3           c
```

Here I obtain ddat rows one and three in three different ways:

```
ddat[c(1,3),]
```

```
  numeric non.numeric
1       1           a
3       3           c
```

```
ddat[-2,]
```

```
  numeric non.numeric
1       1           a
3       3           c
```

```
ddat[2, drop = TRUE]
```

```
Warning in `[.data.frame`(ddat, 2, drop = TRUE): 'drop' argument will be
ignored
```

```
  non.numeric
1           a
2           b
3           c
```

■

Square braces can also be used to rearrange data components.

```
ddat[c(3,1,2),]
```

```
  numeric non.numeric
3       3           c
1       1           a
2       2           b
```

Duplicate components:

```
ldat[c(2,2)]
```

```
$element2
[1] "this.is.a.list"

$element2
[1] "this.is.a.list"
```

Or even replace data components:

```
ddat[,2] <- c("d","e","f")
ddat
```

```
  numeric non.numeric
1       1           d
2       2           e
3       3           f
```

3.4.1 Subsetting a Factor

Importantly, the factor level structure of a factor will remain intact even if one or more of the levels are entirely removed.

Example 3.42.
For example:

```
fdat <- as.factor(ddat[,2])
fdat
```

```
[1] d e f
Levels: d e f
```

```
fdat[-1]
```

```
[1] e f
Levels: d e f
```

Note that the level `a` remains a characteristic of `fdat`, even though the cell containing the lone observation of `a` was removed from the dataset. This outcome is allowed because it is desirable for certain analytical situations (for instance, summarizations that should acknowledge missing data for some levels).

■

To remove levels that no longer occur in a factor, we can use the function `droplevels()`.

Example 3.43.
For example:

```
droplevels(fdat[-1])
```

```
[1] e f
Levels: e f
```

■

3.4.2 Subsetting with Boolean Operators

Boolean (TRUE or FALSE) outcomes can be used in combination with square brackets to subset data.

Example 3.44.
Consider the dataframe used earlier (Exercise 3.19) to demonstrate logical commands.

```
dframe <- data.frame(
Age = c(18,22,23,21,22,19,18,18,19,21),
Sex = c("M","M","M","M","M","F","F","F","F","F"),
Weight_kg = c(63.5,77.1,86.1,81.6,70.3,49.8,54.4,59.0,65,69)
)
```

Here we extract Age outcomes less than or equal to 21.

```
ageTF <- dframe$Age <= 21
dframe$Age[ageTF]
```

```
[1] 18 21 19 18 18 19 21
```

We could also use this information to obtain entire rows of the dataframe.

```
dframe[ageTF,]
```

```
   Age Sex Weight_kg
1   18   M      63.5
4   21   M      81.6
6   19   F      49.8
7   18   F      54.4
8   18   F      59.0
9   19   F      65.0
10  21   F      69.0
```

■

3.4.3 When Subset Is Larger Than Underlying Data

R allows one to make a data subset larger than underlying data itself, although this results in the generation of filler NAs.

Example 3.45.
Consider the following example:

```
x <- c(-2, 3, 4, 6, 45)
```

The atomic vector x has length five. If I ask for a subset of length seven, I get:

```
x[1:7]
```

```
[1] -2  3  4  6 45 NA NA
```

■

3.4.4 Subsetting with `upper.tri()`, `lower.tri()`, and `diag()`

We can use square brackets alongside the functions `upper.tri()`, `lower.tri()`, and `diag()` to examine the upper triangle, lower triangle, and diagonal parts of a matrix, respectively.

Example 3.46.
For example:

```
mat <- matrix(ncol = 3, nrow = 3, data = c(1, 2, 3, 2, 4, 3, 5, 1, 4))
mat
```

```
     [,1] [,2] [,3]
[1,]    1    2    5
[2,]    2    4    1
[3,]    3    3    4
```

```
mat[upper.tri(mat)]
```

```
[1] 2 5 1
```

```
mat[lower.tri(mat)]
```

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

```
diag(mat)
```

```
[1] 1 4 4
```

Note that `upper.tri()` and `lower.tri()` are used identify the appropriate triangle in the object `mat`. Subsetting is then accomplished using square brackets.

■

3.5 Object Adresses

This section and the next concern important but rather advanced explorations of memory addresses and memory usage of **R** data storage objects.

In programming, a *pointer* is a variable used to store the memory address of another variable as its value. All **R** objects will have pointers, although the addresses themselves are temporary,

and will change every time **R** is started, and memory is reallocated. Object pointer addresses can be identified using the function `obj_address()` from the **R** package *rlang* (Henry and Wickham, 2025). See Section 3.7 for a formal introduction to **R** packages.

```
# install.packages("rlang") installs rlang
library(rlang) # loads rlang

x <- c(1, 2, 3)
obj_address(x)
```

```
[1] "0x0000012db4cabc28"
```

Example 3.47.

The function `sxp()` from the **R** package *lobstr* (Wickham, 2022), will list both the address and the underlying C-codified *typedef* `SEXP` (refer to Section 2.3.6) of an object.

```
# install.packages("lobstr") installs lobstr
library(lobstr) # loads lobstr
sxp(x)
```

```
[1:0x12db4cabc28] <REALSXP[3]> (named:4)
```

The double precision numeric vector `x` is underlain by the `SEXP` type `REALSXP`.

A `dataframe` or `list` will each have its own address. However, these data containers will also have pointers for each of their nested canonical elements (column vectors for dataframes and list components for lists).

■

Example 3.48.
Consider the objects `df` and `ldata` below.

```
df <- data.frame(numeric = c(1, 2, 3), non.numeric = c("a", "b", "c"))
ldata <- list(first = c(1, 2, 3), second = "this.is.a.list")
```

We have the conceptual address structure shown in Fig 3.2.

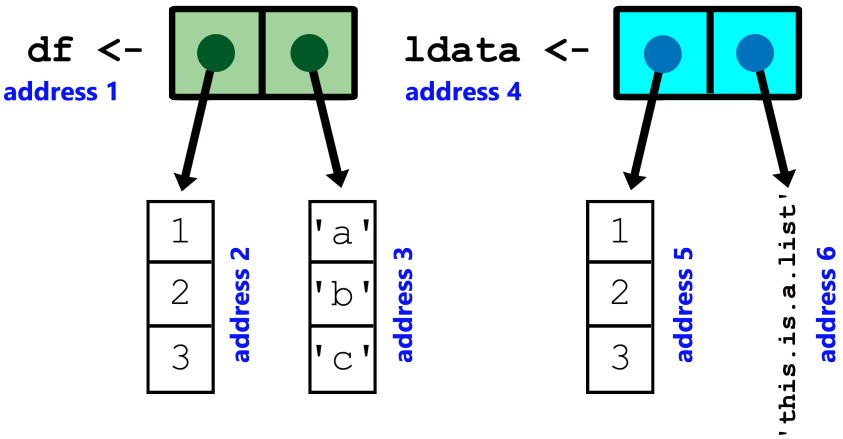


Figure 3.2: The conceptual address structure of a `list`, `ldata` and a `dataframe`, `df`. Figure follows (Wickham, 2019).

The actual **R** address structure of `df` and `ldata` can be shown with the function `lobstr::ref()`.

```
ref(df)
```

```
  [1:0x12db9b62bc8] <df[,2]>
├─numeric = [2:0x12db462be58] <dbl>
└─non.numeric = [3:0x12db462bea8] <chr>
```

```
ref(ldata)
```

```
  [1:0x12db9aaad08] <named list>
├─first = [2:0x12db456a098] <dbl>
└─second = [3:0x12db5436fc8] <chr>
```

■

3.5.1 Copy-on-Modify

In managing object addresses, **R** generally uses a method called *copy-on-modify*[11] to preserve shared address structures of objects (Wickham, 2019). Copy-on-modify semantics dramatically increase computational efficiency and reduce object memory usage.

Example 3.49.
For example, if I create a copy of an object, then both the copy and the original object will point to the same address(es) and those address(es) stored values.

[11] Ross Ihaka, in an **R**-help response in 2000 referred to this as "copy on modify (if necessary)".

```
ldata.copy <- ldata
obj_address(ldata.copy) == obj_address(ldata)
```

```
[1] TRUE
```

That is, we have the framework shown in Fig 3.3.

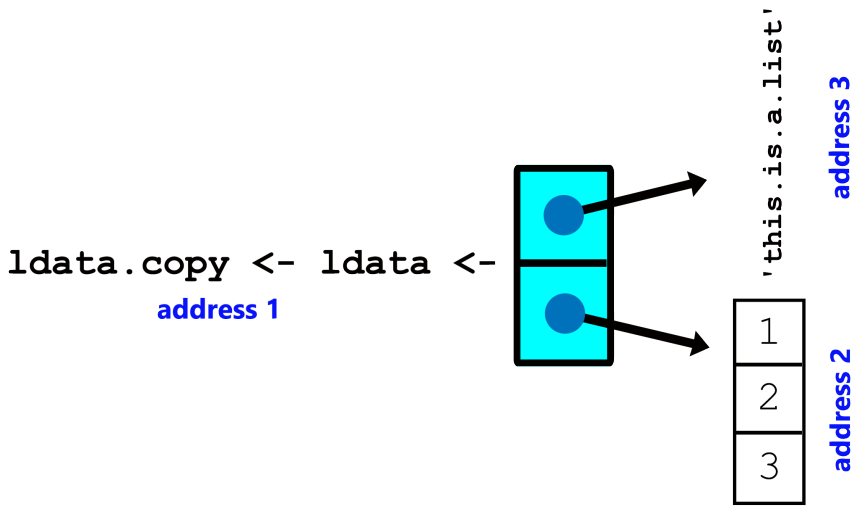


Figure 3.3: The conceptual address structure of a `list`, and its copy. Figure follows (Wickham, 2019).

The phrase "copy-on-modify" comes from the fact that even though the code `ldata.copy <- ldata` indicates that a copy of `ldata` is being made, no copying is actually being done because `ldata.copy` and `ldata` both point to a single value at the same address[12]. Copying will only occur if I indicate that one or both of the objects will be modified. For instance, below I indicate that a logical vector named `logical` should be added to `ldata.copy`:

```
ldata.copy$logical <- c(TRUE, FALSE, TRUE)
```

Then `ldata.copy` will be copied (from `ldata`) and given a new overall address:

```
obj_address(ldata.copy) == obj_address(ldata)
```

```
[1] FALSE
```

Additionally, to optimize efficiency, unmodified elements of `ldata.copy` (i.e., `first` and `second`) will still point to the same shared addresses and values defined originally in `ldata`.

```
ref(ldata, ldata.copy)
```

[12] For additional information see this exchange on stack overflow.

```
 [1:0x12db9aaad08] <named list>
├─first = [2:0x12db456a098] <dbl>
└─second = [3:0x12db5436fc8] <chr>

 [4:0x12db3028d28] <named list>
├─first = [2:0x12db456a098]
├─second = [3:0x12db5436fc8]
└─logical = [5:0x12db7b0f148] <lgl>
```

That is, we have the framework shown in Fig 3.4.

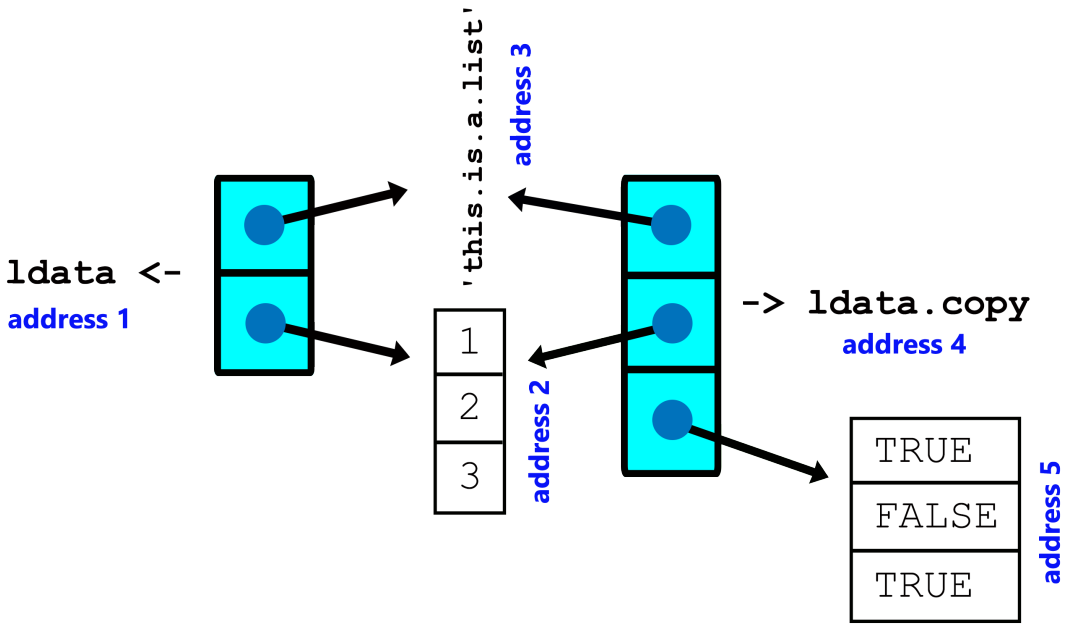


Figure 3.4: An illustration of copy-on-modify. Figure follows (Wickham, 2019).

■

Example 3.50.
Copy-on-modify semantics will also be used for most other **R** objects. For instance,

```
df.copy <- df
df.copy$logical <- c(TRUE, FALSE, TRUE)
ref(df, df.copy)
```

```
 [1:0x12db9b62bc8] <df[,2]>
├─numeric = [2:0x12db462be58] <dbl>
└─non.numeric = [3:0x12db462bea8] <chr>

 [4:0x12db245c728] <df[,3]>
```

```
├─numeric = [2:0x12db462be58]
├─non.numeric = [3:0x12db462bea8]
└─logical = [5:0x12db67385c8] <lgl>
```

■

It is worth noting that copy-on-modify procedures are followed in a `dataframe` (as shown above) only if modifications are made to columns (e.g., values are transformed) or to the column structure (columns are added or deleted). If a *row* is modified, then every column will be modified, which means that every column must be copied and given a new address.

Despite its benefits, copy-on-modify is not widely used by other languages. For instance, modification of an array in the Python language (Section 9.5) will inefficiently create an entirely new array, while destroying the old array (Pine, 2019; Haddock and Dunn, 2011).

3.5.2 Names and Symbols

An object name appears to be inextricably tied to its content. However, to access the content of an object `x`, one must go to the pointer address associated with the name `x` in computer memory. In **R**, the terms and *name* and *symbol* are analogous[13], and reveal historic ties to **S** (which uses *name*) and Lisp (which uses *symbol*). We can disentangle names/symbols from their associated content using the functions `expression()` and particularly `rlang::expr()`.

Recall (Section 2.9.5) that objects of class expression can be evaluated using the function `eval()`.

```
a <- 4
x <- expr(a)
eval(x)
```

```
[1] 4
```

Example 3.51.
Here `x` is actually defined to be the symbol name `a`.

```
x <- expr(a)
is.symbol(x)
```

```
[1] TRUE
```

```
is.name(x)
```

```
[1] TRUE
```

The symbols/names in an expression can be identified using `all.names()`.

[13] For example, the functions `is.name()` and `is.symbol()` are identical in base **R**.

```
all.names(x)
```

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

Values can be substituted for symbols using the function `substitute()`.

```
substitute(expression(a + b), list(a = 1))
```

```
expression(1 + b)
```

■

The name of function argument is a special type of symbol called a *promise*. A promise is a placeholder that delays the evaluation of a function argument until it is actually required by the function itself (see Section 8.8.2).

3.6 Memory and Objects

The memory structure of **R** objects can be complex, even for the simple examples used here. This is because object canonical components and attributes will also require pointers and `SEXP` types.

Example 3.52. Continuing Example 3.47 we have:

```
sxp(df)
```

```
[1:0x12db9b62bc8] <VECSXP[2]> (object named:9)
  numeric [2:0x12db462be58] <REALSXP[3]> (named:16)
  non.numeric [3:0x12db462bea8] <STRSXP[3]> (named:16)
  _attrib [4:0x12db559f348] <LISTSXP> (named:1)
    names [5:0x12db9b62cc8] <STRSXP[2]> (named:65535)
    class [6:0x12da5fc3a48] <STRSXP[1]> (named:65535)
    row.names [7:0x12db53a7620] <INTSXP[2]> (named:65535)
```

```
sxp(ldata)
```

```
[1:0x12db9aaad08] <VECSXP[2]> (named:10)
  first [2:0x12db456a098] <REALSXP[3]> (named:10)
  second [3:0x12db5436fc8] <STRSXP[1]> (named:13)
  _attrib [4:0x12db55a8f20] <LISTSXP> (named:1)
    names [5:0x12db9aaad48] <STRSXP[2]> (named:65535)
```

Both `df` and `ldata` use the overarching `SEXP` type `VECSXP` (even though an **R** `dataframe` is a `list` of vectors (Section 3.1.4), and not strictly a vector (but see Wickham (2019))). `REALSXP` and `STRSXP` are required for numerical and string components, respectively, of both `df` and `ldata`. Note that the `dataframe` object has additional attributes, including a `row.names` slot.

■

The size of objects in the global environment can be checked using the function `lobstr::obj_size()`.

Example 3.53.
Any vector of zero length will require 48 bytes of memory (see Wickham (2019).

```
obj_size(numeric())
```

```
48 B
```

```
obj_size(logical())
```

```
48 B
```

```
obj_size(character())
```

```
48 B
```

■

Example 3.54.
Eighty bytes (640 bits; Ch 12) are required to store a three element numeric vector in the current 64 bit version of **R**.

```
x <- c(1, 2, 3)
obj_size(x)
```

```
80 B
```

■

R lists and dataframes are very efficient storage entities because their canonical components are generally constrained to only pointers (Section 3.5).

Example 3.55.
Slightly more memory is required for storing the `vector` `x` from Example 3.54, in a `list`.

```
obj_size(list(x))
```

```
136 B
```

Dataframes are less efficient than lists, particularly for small datasets.

```
obj_size(data.frame(x))
```

```
760 B
```

This is because the SEXP structure of a dataframe is more complex than a list (Example 3.52).

■

Example 3.56.
Here I create an object y that supposedly contains four copies of x.

```
y <- list(x, x, x, x)
```

However y only requires 80 more bytes than x (while x itself requires 80 B).

```
obj_size(x)
```

```
80 B
```

```
obj_size(y)
```

```
160 B
```

A dataframe version of y is again less efficient than a list version, although this difference becomes negligible as the the number of copies of x increases (Fig 3.5).

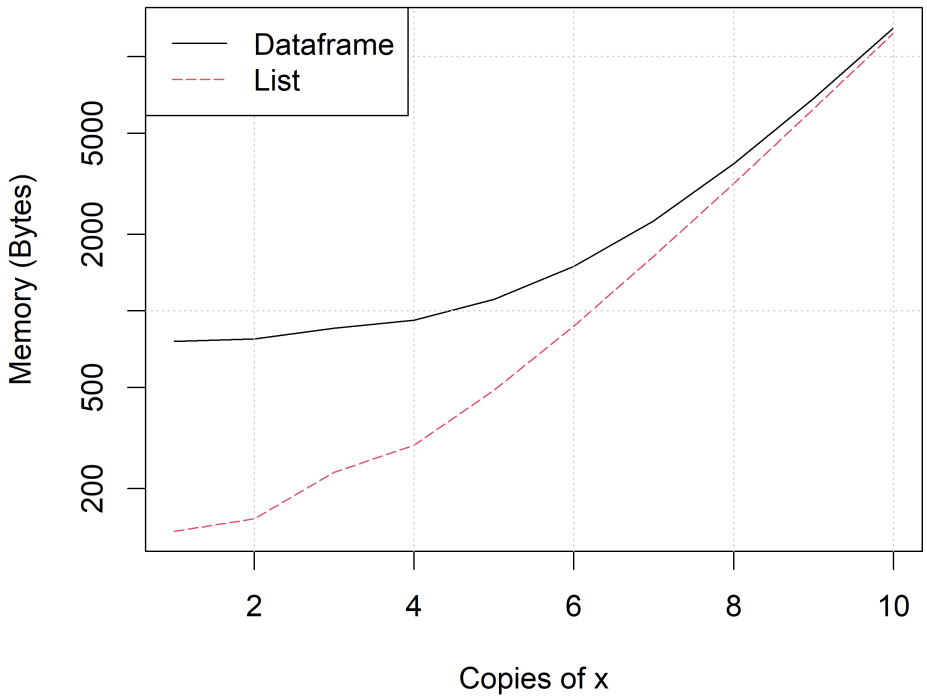


Figure 3.5: Comparison of `list` and `dataframe` memory useage given copies of `x`. Code for figure is at https://amalgamofr.org/listVSdf.R.

■

3.6.1 Global Character Pool

R memory storage for character string objects is optimized in an unexpected way. Instead of creating addresses for user strings on the fly, **R** locates addresses for single immutable string values stored within a *global string pool*. This approach has also been called *string interning* (Wikipedia, 2025c).

Example 3.57.
Thus, for the string below, we have the conceptual framework shown in Fig 3.6.

```
x <- c("a", "zyx", "PhD")
```

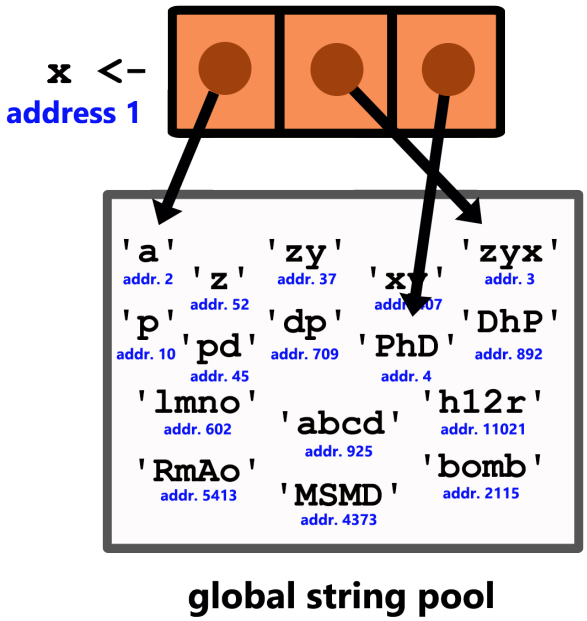


Figure 3.6: **R** character strings and the global string pool. Figure follows (Wickham, 2019).

The actual pointers to the strings in the global string pool can be identified with: `lobstr:ref()`.

```
ref(x, character = TRUE)
```

```
  [1:0x12db023f2c8] <chr>
├─[2:0x12d8c0110e0] <string: "a">
├─[3:0x12dbc8300e8] <string: "zyx">
└─[4:0x12dbc830040] <string: "PhD">
```

■

Sring interning dramatically improves computational efficiency and decreases memory usage, because string values only need to be stored once. String pools originated with Lisp, and are also used by the Java language (Wikipedia, 2025c).

3.7 Packages

An **R** *package* contains a set of related functions, documentation, (often) data files, and other miscellany that have been bundled together. The so-called **R**-*distribution packages* are included

with a conventional download of **R** (Table 3.3). These packages are directly controlled by the **R** core development team and are extremely well-vetted and trustworthy.

Packages in Table 3.4 constitute the **R**-*recommended packages*. These are not necessarily controlled by the **R** core development team, but are also extremely useful, well-tested, and stable, and like the **R**-distribution packages, are included in conventional downloads of **R**.

Aside from distribution and recommended packages, there are a large number of *contributed packages* that have been created by **R**-users ($>$ 20000 as of 9/12/2023). Table 3.5 lists a few.

3.7.1 Package Installation

Contributed packages can be *installed* from CRAN (the Comprehensive **R** Archive Network). To do this, one can go to **Packages**>Install package(s) on the **R**-GUI toolbar (non-Unix only), and choose a nearby CRAN mirror site to minimize download time. Once a mirror site is selected, the packages available at the site will appear. One can simply click on the desired packages to install them. Packages can also be downloaded directly from the command line using `install.packages("package name")`. Thus, to install the package *vegan* (see Table 3.5), I would simply type:

```
install.packages("vegan")
```

If local web access is not available, packages can be installed as compressed (.zip, .tar) files which can then be placed manually on a workstation by inserting the package files into the **library** folder within the top level **R** directory, or into a path-defined **R** library folder in a user directory.

The installation pathway for contributed packages can be identified using `.libPath()`.

```
.libPaths()
```

```
[1] "C:/Users/ahoken/AppData/Local/R/win-library/4.5"
[2] "C:/Program Files/R/R-4.5.1/library"
```

This process can be facilitated in RStudio via applications in its **Plots and Miscellany** panel (see Section 2.10).

Several functions exist for updating packages and for comparing currently installed versions of packages to their latest versions at CRAN or other repositories. The function `old.packages()` prints a list of currently installed packages that have a (suitable) later version. Here are a few of the packages I have installed that have later versions.

```
head(old.packages(repos = "https://cloud.r-project.org"))[,c(1,3,4,5)]
```

```
              Package         Installed Built   ReposVer
adehabitatLT  "adehabitatLT"  "0.3.28"  "4.5.0" "0.3.29"
animation     "animation"     "2.7"     "4.5.1" "2.8"
arcgisgeocode "arcgisgeocode" "0.2.3"   "4.5.0" "0.3.0"
```

```
batchtools    "batchtools"    "0.9.17"  "4.5.0" "0.9.18"
bookdown      "bookdown"      "0.43"    "4.5.1" "0.44"
broom         "broom"         "1.0.8"   "4.5.0" "1.0.10"
```

The function `update.packages()` will identify, and offer to download and install later versions of installed packages.

3.7.2 Loading Packages

Once a contributed package is installed on a computer it never needs to be re-installed. However, for use in an **R** session, recommended packages, and installed contributed packages will need to be *loaded* (although see Section 2.6.1 concerning customized **R** startup). This can be done using the `library()` function, or point and click tools if one is using RStudio. For example, to load the installed contributed *vegan* package, I would type:

```
library(vegan)
```

```
Loading required package: permute
```

```
Loading required package: lattice
```

We see that two other packages are loaded when we load *vegan*: *permute* and *lattice*.

To detach *vegan* from the global environment, I would type:

```
detach(package:vegan)
```

We can check if a specific package is loaded using the function `.packages()`. Most of the **R** distribution packages are loaded (by default) upon opening a session. Exceptions include `compiler`, `grid`, `parallel`, `splines`, `stats4`, and `tools`.

```
bpack <- c("base", "compiler", "datasets", "grDevices", "graphics",
           "grid", "methods", "parallel", "splines", "stats", "stats4",
           "tcltk", "tools", "translations", "utils")
sapply(bpack, function(x) (x %in% .packages()))
```

```
        base     compiler     datasets    grDevices     graphics
        TRUE        FALSE         TRUE         TRUE         TRUE
        grid      methods     parallel      splines        stats
       FALSE         TRUE        FALSE        FALSE         TRUE
      stats4        tcltk        tools translations        utils
       FALSE         TRUE        FALSE        FALSE         TRUE
```

The function `sapply()`, which allows application of a function to each element in a vector or list, is formally introduced in Section 4.1.1.

The package *vegan* is no longer loaded because of the application of `detach(package:vegan)`.

```
"vegan" %in% .packages()
```

```
[1] FALSE
```

We can get a summary of information about a session, including details about the version of **R** being used, the underlying computer platform, and the loaded packages with the function `sessionInfo()`.

```
si <- sessionInfo()
si$R.version$version.string
```

```
[1] "R version 4.5.1 (2025-06-13 ucrt)"
```

```
si$running
```

```
[1] "Windows 11 x64 (build 26100)"
```

```
head(names(si$loadedOnly))
```

```
[1] "gtable"   "xfun"     "ggplot2"  "vctrs"    "tools"    "generics"
```

This information is important to include when reporting issues to package maintainers.

Once a package is installed its functions can generally be accessed using the *double colon* metacharacter, `::`, even if the package is not actually loaded. For instance, the function `vegan::diversity()` will allow access to the function `diversity()` from *vegan*, even when *vegan* is not loaded.

```
head(vegan::diversity)[1:2]
```

```
[1] function (x, index = "shannon", groups, equalize.groups = FALSE,
[2]     MARGIN = 1, base = exp(1))
```

The *triple colon* metacharacter, `:::`, can be used to access internal package functions. These functions, however, are generally kept internal for good reason, and probably shouldn't be used outside of the context of the rest of the package.

3.7.3 Other Package Repositories

Aside from CRAN, there are currently three other extensive repositories of **R** packages. First, the Bioconductor project (http://www.bioconductor.org/packages/release/Software/html) contains a large number of packages for the analysis of data from current and emerging biological assays. Bioconductor packages are generally not stored at CRAN. Packages can be downloaded from Bioconductor using an `R` script called `biocLite`. To access the script and download the package *RCytoscape* from Biocondctor, I could type:

```
source("http://bioconductor.org/biocLite.R")
biocLite("RCytoscape")
```

Second, the *Posit Package Manager* (formerly the RStudio Package Manager) provides a repository interface for **R** packages from CRAN, Bioconductor, and packages for the Python system (see Section 9.5). Third, **R**-forge (http://r-forge.r-project.org/) contains releases of packages that have not yet been implemented into CRAN, and other miscellaneous code. Bioconductor, Posit, and **R**-forge can be specified as repositories from **Packages**>**Select Repositories** in the **R**-GUI (non-Unix only). Other informal **R** package and code repositories currently include GitHub and Zenodo.

Table 3.3: The **R**-distribution packages.

Package	Maintainer	Topic(s) addressed by package	Author/Citation
base	**R** Core Team	Base **R** functions	R Core Team (2023)
compiler	**R** Core Team	**R** byte code compiler	R Core Team (2023)
datasets	**R** Core Team	Base **R** datasets	R Core Team (2023)
grDevices	**R** Core Team	Devices for base and grid graphics	R Core Team (2023)
graphics	**R** Core Team	**R** functions for base graphics	R Core Team (2023)
grid	**R** Core Team	Grid graphics layout capabilities	R Core Team (2023)
methods	**R** Core Team	Formal methods and classes for **R** objects	R Core Team (2023)
parallel	**R** Core Team	Support for parallel computation	R Core Team (2023)
splines	**R** Core Team	Regression spline functions and classes	R Core Team (2023)
stats	**R** Core Team	**R** statistical functions	R Core Team (2023)
stats4	**R** Core Team	Statistical functions with S4 classes	R Core Team (2023)
tcltk	**R** Core Team	Language bindings to Tcl/Tk	R Core Team (2023)
tools	**R** Core Team	Tools forpackage development/administration	R Core Team (2023)
utils	**R** Core Team	**R** utility functions	R Core Team (2023)

Table 3.4: The **R**-recommended packages.

Package	Maintainer	Topic(s) addressed by package	Author/Citation
KernSmooth	B. Ripley	Kernel smoothing	Wand (2023)
MASS	B. Ripley	Important statistical methods	Venables and Ripley (2002)
Matrix	M. Maechler	Classes and methods for matrices	Bates et al. (2023)
boot	B. Ripley	Bootstrapping	Canty and Ripley (2022)
class	B. Ripley	Classification	Venables and Ripley (2002)
cluster	M. Maechler	Cluster analysis	Maechler et al. (2022)
codetools	S. Wood	Code analysis tools	Tierney (2023)
foreign	**R** core team	Data stored by non-**R** software	R Core Team (2023)
lattice	D. Sarkar	Lattice graphics	Sarkar (2008)
mgcv	S. Wood	Generalized Additive Models	Wood (2011, 2017)
nlme	**R** core team	Linear and non-linear mixed effect models	Pinheiro and Bates (2000)
nnet	B. Ripley	Feed-forward neural networks	Venables and Ripley (2002)
rpart	B. Ripley	Partitioning and regression trees	Venables and Ripley (2002)
spatial	B. Ripley	Kriging and point pattern analysis	Venables and Ripley (2002)

Table 3.5: Useful contributed **R** packages.

Package	Maintainer	Topic(s) addressed by package	Author/Citation
asbio	K. Aho	Stats pedagogy and applied stats	Aho (2023)
car	J. Fox	General linear models	Fox and Weisberg (2019)
coin	T. Hothorn	Non-parametric analysis	Hothorn et al. (2006, 2008)
ggplot2	H. Wickham	*Tidyverse* grid graphics	Wickham (2016)
lme4	B. Bolker	Linear mixed-effects models	Bates et al. (2015)
plotrix	J. Lemonetal.	Helpful graphical ideas	Lemon (2006)
spdep	R. Bivand	Spatial analysis	Bivand et al. (2013); Pebesma and Bivand (2023)
tidyverse	H. Wickham	Data science under the *tidyverse*	Wickham et al. (2019)
vegan	J. Oksanen	Multivariate and ecological analysis	Oksanen et al. (2022)

3.7.4 Accessing Package Information

Important information concerning a package can be obtained from the `packageDescription()` family of functions. Here is the version of the **R** contributed package *asbio* on my work station:

```
packageVersion("asbio")
```

```
[1] '1.11'
```

Here is the version of **R** used to build the installed version of *asbio*, and the package's build date:

```
packageDescription("asbio", fields="Built")
```

```
[1] "R 4.5.0; ; 2025-05-14 01:46:34 UTC; windows"
```

3.7.5 Accessing Datasets in R-packages

The command:

```
data()
```

results in a listing of a datasets available in a session from within **R** packages loaded in a particular **R** session. Whereas the code:

```
data(package = .packages(all.available = TRUE))
```

results in a listing of a datasets available in a session from within *installed* **R** packages.

If one is interested in datasets from a particular package, for instance the package *datasets*, one could type:

```
data(package = "datasets")
```

All datasets in the *datasets* package are read into an **R**-session automatically, upon loading of the package. This is because the package's dataframes were defined to be *lazy loaded* when the package was built (Ch 10). To access a dataset from a package that do not specify lazy loading, we must use the `data()` function with the data object name as an argument, after loading the data object's package environment.

Example 3.58.
Here I load the *asbio* package to access its dataframe `K`, which contains soil potassium measurements for "identical" soils samples, from eight soil testing laboratories.

```
library(asbio)
data(K)
```

The data are now contained in a dataframe (called K) that we can manipulate and summarize.

```
summary(K)
```

```
       K            lab
 Min.   :187   B      : 9
 1st Qu.:284   D      : 9
 Median :314   E      : 9
 Mean   :308   F      : 9
 3rd Qu.:341   G      : 9
 Max.   :413   H      : 9
               (Other):18
```

The function `summary()` provides the mean and a conventional *five number summary* (minimum, 1st quartile, median, 3rd quartile, maximum) of quantitative variables (i.e., K) and a count of the number of observations in each level of a categorical variable (i.e., lab).

■

Example 3.59.
The Loblolly data in the *datasets* package does not require use of `data()` because of its use of lazy loading. Recall that we can access the first few rows from a dataframe using the function `head()`:

```
head(Loblolly, 5)
```

```
Grouped Data: height ~ age | Seed
   height age Seed
1    4.51   3  301
15  10.89   5  301
29  28.72  10  301
43  41.74  15  301
57  52.70  20  301
```

Here we apply the `class()` function to Loblolly. The result is surprisingly complex.

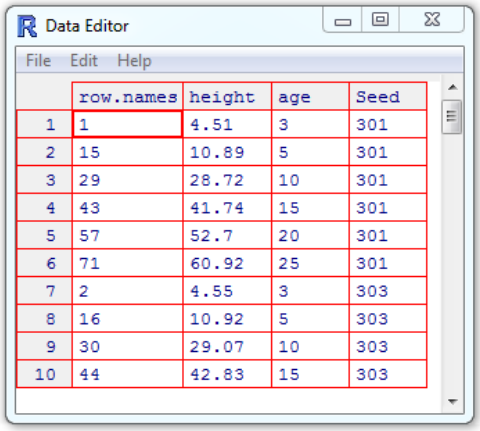
```
class(Loblolly)
```

```
[1] "nfnGroupedData" "nfGroupedData"  "groupedData"    "data.frame"
```

In addition to the `dataframe` class, there are three other classes (`nfnGroupedData, nfGroupedData, groupedData`). These allow recognition of the nested structure of the `age` and `Seed` variables (defined to `height` is a function of `age` *in* `Seed`), and facilitates the analysis of the data using mixed effect model algorithms in the package *nlme* (see `?Loblolly`).

■

R provides a spreadsheet-style data editor if one types `fix(x)`, when `x` is a dataframe or a two dimensional array. For instance, the command `fix(Loblolly)` will open the `Loblolly` pine dataframe in the data editor (Figure 3.7). When `x` is a function or character string, then a script editor is opened containing `x`. The data editor has limited flexibility compared to software whose main interface is a spreadsheet, and whose primary purpose is data entry and manipulation, e.g., Microsoft Excel®. Changes made to an object using `fix()` will only be maintained for the current work session. They will not permanently alter objects brought in remotely to a session. The function `View(x)` (RStudio only) will provide a non-editable spreadsheet representation of a dataframe or numeric array.



	row.names	height	age	Seed
1	1	4.51	3	301
2	15	10.89	5	301
3	29	28.72	10	301
4	43	41.74	15	301
5	57	52.7	20	301
6	71	60.92	25	301
7	2	4.55	3	303
8	16	10.92	5	303
9	30	29.07	10	303
10	44	42.83	15	303

Figure 3.7: The default **R** spreadsheet editor.

3.8 Facilitating Command Line Data Entry

Command line data entry is made easier with with several **R** functions. The function `scan()` can speed up data entry because a prompt is given for each data point[14], and separators are created by the function itself. Data entries can be designated using the space bar or line breaks. The `scan()` function will be terminated by a additional blank line or an end of file (EOF) signal. This will be Ctrl + D in Unix-alike operating systems and Windows.

Below I enter the numbers 1, 2, and 3 as datapoints, separated by spaces, and end data entry using an additional line break. The data are saved as the object `a`.

```
a <- scan()
1: 1 2 3
4:
Read 3 items
```

Sequences can be generated quickly in **R** using the `:` operator

[14] If one uses the default `scan()` argument: `file = ""`.

```
1:10
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

or the function `seq()`, which allows additional options:

```
seq(1, 10)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
seq(1, 10, by = 2) # 1 to 10 by two
```

```
[1] 1 3 5 7 9
```

```
seq(1, 10, length = 4) # 1 to 10 in four evenly spaced points
```

```
[1]  1  4  7 10
```

Entries can be repeated with the function `rep()`. For example, to repeat the sequence 1 through 5, five times, I could type:

```
rep(c(1:5), 5)
```

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

Note that the first argument in `rep()`, defines the thing we want to repeat and the second argument, 5, specifies the number of repetitions. I can use the argument `each` to repeat individual elements a particular number of times.

```
rep(c(1:5), each = 2)
```

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

We can use `seq()` and `rep()` simultaneously to create complex sequences. For instance, to repeat the sequence 1,3,5,7,9,11,13,15,17,19, three times, we could type:

```
rep(seq(1, 20, by = 2), 3)
```

```
 [1]  1  3  5  7  9 11 13 15 17 19  1  3  5  7  9 11 13 15 17 19  1  3  5
[24]  7  9 11 13 15 17 19
```

3.9 Importing Data Into R

While it is possible to enter data into **R** at the command line, this will normally be inadvisable except for small datasets. In general it will be much easier to import data. **R** can read data from many different kinds of formats including .txt, and .csv (comma separated) files, and files with space, tab, and carriage return datum separators. Important **R** functions for importing data

include `read.table()`, `read.csv()`, `read.delim()`, and `scan()`. The function `load()` can be used to import data files in .rda data formats, or other **R** objects. Datasets read into **R** will generally be of class `dataframe` (data storage mode `list`). ### Import Using `read.table()`, `read.csv()`, and `scan()` {#rt} The `read.table()` function can import data organized under a wide range of formats. It's first three arguments are very important.

- `file` defines the name of the file and directory hierarchy which the data are to be read from.
- `header` is a logical (`TRUE` or `FALSE`) value indicating whether `file` contains column names as its first line.
- `sep` refers to the type of data separator used for columns. Comma separated files use commas to separate columns. Thus, in this case `sep = ","`. Tab separators are specified as `"\t"`. Space separators are specified as spaces, specified as simply `" "`.

Other useful `read.table()` arguments include `row.names`, `header`, and `na.strings`. The specification `row.names = 1` indicates that the first column in the imported dataset contains row names. The specification `header = TRUE`, the default setting, indicates that the first row of data contains column names. The argument `na.strings = "."` indicates that missing values in the imported dataset are designated with periods. By default `na.strings = NA`.

Example 3.60.
As an example of `read.table()` usage, assume that I want to import a .csv file called `veg.csv` located in folder called veg_data, in my working directory. The first row of `veg.csv` contains column names, while the first column contains row names. Missing data in the file are indicated with periods. I would type:

```
read.table("veg_data/veg.csv", sep = ",", header = TRUE, row.names
= 1, na.strings = ".")
```

As before, note that as a legacy of its development under Unix, **R** locates files in directories using forward slashes (or doubled backslashes) rather than single Windows backslashes.

■

The `read.csv()` function assumes data are in a .csv format. Because the argument `sep` is unnecessary, this results in a simpler code statement.

```
read.csv("veg_data\\veg.csv", header = TRUE, row.names
= 1, na.strings = ".")
```

The function `scan()` can read in data from an essentially unlimited number of formats, and is extremely flexible with respect to character fields and storage modes of numeric data. In addition to arguments used by `read.table()`, `scan()` has the arguments

- `what` which describes the storage mode of data e.g., `"logical"`, `"integer"`, etc., or if `what` is a list, components of variables including column names (see below), and

- `dec` which describes the decimal point character (European scientists and journals often use commas).

Example 3.61.
Assume that `veg_data/veg.csv` has a column of species names, called `species`, that will serve as the dataframe's row names, and 3 columns of numeric data, named `site1, site2`, and `site3`. We would read the data in with `scan()` using:

```
scan("veg.csv", what = list(species = "", site1 = 0, site2 = 0, site3 = 0),
na.strings = ".")
```

The empty string `species = ""` in the list comprising the argument `what`, indicates that `species` contains character data. Stating that the remaining variables equal 0, or any other number, indicates that they contain numeric data.

■

The easiest way to import data, if the directory structure is unknown or complex, is to use `read.csv()` or `read.table()`, with the `file.choose()` function as the `file` argument.

Example 3.62.
For instance, by typing:

```
df <- read.csv(file.choose())
```

We can now browse for a .csv files to open that will, following import, be a dataframe with the name `df`. Other arguments (e.g., `header`, `row.names`) will need to be used, when appropriate, to import the file correctly.

■

Occasionally strange characters, e.g., `ï..`, may appear in front of the first header name when reading in files created in Excel® or other Microsoft applications. This is due to the addition of *Byte Order Mark* (BOM) characters which indicate, among other things, the Unicode character encoding of the file. These characters can generally be eliminated by using the argument `fileEncoding="UTF-8-BOM"` in `read.table()`, `read.csv()`, or `scan()`.

3.9.1 Import Using RStudio

RStudio allows direct menu-driven import of file types from a number of spreadsheet and statistical programs including Excel®, SPSS®, SAS®, and Stata® by going to **File**>**Import Dataset**. Certain restrictions may exist, however, that do not occur for `read.table()` and `read.csv()` (Table 3.6).

Table 3.6: Data import options in RStudio by data storage file type.

	CSV or Text	Excel®	SAS®, SPSS®, Stata®
Import from file system or URL	X	X	X
Change column data types	X	X	
Skip or include columns	X	X	X
Rename dataset	X	X	
Skip the first *n* rows	X	X	
Use header row for column names	X		
Trim spaces in names	X		
Change column delimiter	X		
Encodingselection	X		
Select quote identifiers	X		
Select escape identifiers	X		
Select comment identifiers	X		
Select `NA` identifiers	X	X	
Specify model file			X

3.9.2 Final Considerations

It is generally recommended that datasets imported and used by **R** be smaller than 25% of the physical memory of the computer. For instance, they should use less than 8 GB on a computer with 32 GB of RAM.

R can handle extremely large datasets, i.e. $>$ 10 GB, and $> 1.2 \times 10^{10}$ rows. In this case, however, specific **R** packages can be used to aid in efficient data handling. Parallel computing and workstation modifications may allow even greater efficiency. The actual upper physical limit for an **R** dataframe is $2 \times 10^{31} - 1$ elements. Note that this exceeds Excel® by 31 orders of magnitude (Excel® 2019 worksheets can handle approximately 1.7×10^{10} elements).

3.10 Databases

Many examples of biological data (e.g., genomes, spatial data) are extremely large and/or require multiple datasets for meaningful analyses. In this situation, storing and accessing data using a *database* may be extremely helpful. Databases can reside locally (on a user's computer) but more often are stored remotely and are accessed via internet links. This allows simultaneous access for multiple users and storage of extremely large data objects. Modern databases are often structured so that data points in distinct *tables* can be queried, assembled, and analyzed jointly. Two common formats are Relational DataBases (RDB) and Resource Description Framework (RDF) stores (Sima et al., 2019). **R** can often interface with these database systems using the Structured Query Language (SQL), often pronounced *sequel* (Chambers, 2008; Adler, 2010). Due to the need for additional background –provided in intervening chapters– this topic is formally introduced in Section 9.4.

Exercises

1. Create the following data structures:
   (a) An atomic vector object with the numeric entries `1,2,3,4`.
   (b) A matrix object with two rows and two columns with the numeric entries `1,2,3,4`.
   (c) A dataframe object with two columns; one column containing the numeric entries `1,2,3,4`, and one column containing the character entries `"a","b","c","d"`.
   (d) A list containing the objects created in (b) and (c).
   (e) Using `class()`, identify the class and the data storage mode for the objects created in problems a-d. Discuss the characteristics of the identified classes.

2. Assume that you have developed an **R** algorithm that saves hourly stream temperature sensor outputs greater than 20° from each day as separate dataframes and places them into a list container, because some days may have several points exceeding the threshold and some days may have none. Complete the following based on the list `hi.temps` given below:
   (a) Combine the dataframes in `hi.temps` into a single dataframe using `do.call()`.
   (b) Create a dataframe consisting of 10 sets of repeated measures from the dataframe `hi.temps$day2` using `do.call()`.

```
hi.temps <- list(day1 = data.frame(time = c(), temp = c()),
                 day2 = data.frame(time = c(15,16),
                                   temp = c(21.1,22.2)),
                 day3 = data.frame(time = c(14,15,16),
                                   temp = c(21.3,20.2,21.5)))
```

3. Given the dataframe `boo` below, provide solutions to the following questions:
   (a) Identify heights that are less than or equal to 80 inches.
   (b) Identify heights that are more than 80 inches.
   (c) Identify females (i.e. `F`) greater than or equal to 59 inches but less 63 inches.
   (d) Subset rows of `boo` to only contain only data for males (i.e. `M`) greater than or equal to 75 inches tall.
   (e) Find the mean weight of males who are 75 or 76 inches tall.
   (f) Use `ifelse()` or `if()` to classify heights equal to 60 inches as `"small"`, and heights greater than or equal to 60 inches as `"tall"`.

```
boo <- data.frame(height.in = c(70, 76, 72, 73, 81, 66, 69, 75,
                                80, 81, 60, 64, 59, 61, 66, 63,
                                59, 58, 67, 59),
                  weight.lbs = c(160, 185, 180, 186, 200, 156,
                                 163, 178, 186, 189, 140, 156,
                                 136, 141, 158, 154, 135, 120,
                                 145, 117),
                  sex = c(rep("M", 10), rep("F", 10)))
```

4. Create `x <- NA`, `y <- NaN`, and `z <- NULL`.
   (a) Test for the class of `x` using `x == NA` and `is.na(x)` and discuss the results.
   (b) Test for the class of `y` using `y == NaN` and `is.nan(y)` and discuss the results.
   (c) Test for the class of `z` using `z == NULL` and `is.null(z)` and discuss the results.
   (d) Discuss `NA`, `NaN`, and `NULL` designations what are these classes used for and what do they represent?

5. For the following questions, use data from Table 3.7 below.
   (a) Write the data into an **R** dataframe called `plant`. Use the functions `seq()` and `rep()` to help.
   (b) Use `names()` to find the names of the variables.
   (c) Access the first row of data using square brackets.
   (d) Access the third column of data using square brackets.
   (e) Access rows three through five using square brackets.
   (f) Access all rows *except* rows three, five and seven using square brackets.
   (g) Access the fourth element from the third column using square brackets.
   (h) Apply `na.omit()` to the dataframe and discuss the consequences.
   (i) Create a copy of `plant` called `plant2`. Using square brackets, replace the 7th item in the 2nd column in `plant2`, an `NA` value, with the value `12.1`.
   (j) Switch the locations of columns two and three in `plant2` using square brackets.
   (k) Export the `plant2` dataframe to your working directory.
   (l) Convert the `plant2` dataframe into a matrix using the function `as.matrix`. Discuss the consequences.

6. Let:
$$\boldsymbol{A} = \begin{bmatrix} 2 & -3 \\ 1 & 0 \end{bmatrix} \text{ and } \boldsymbol{b} = \begin{bmatrix} 1 \\ 5 \end{bmatrix}$$
   Perform the following operations using **R**:
   (a) $\boldsymbol{A} \otimes \boldsymbol{A}$
   (b) $\boldsymbol{A} \odot \boldsymbol{A}$
   (c) $\boldsymbol{A}\boldsymbol{b}$
   (d) $\boldsymbol{b}\boldsymbol{A}$. Was there an issue? Why?
   (e) $det(\boldsymbol{A})$
   (f) $\boldsymbol{A}^{-1}$
   (g) $\boldsymbol{A}'$

7. We can solve systems of linear equations using matrix algebra under the framework $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, and (thus) $\boldsymbol{A}^{-1}\boldsymbol{b} = \boldsymbol{x}$. In this notation $\boldsymbol{A}$ contains the coefficients from a series of linear equations (by row), $\boldsymbol{b}$ is a vector of solutions given in the individuals equations, and $\boldsymbol{x}$ is a vector of solutions sought in the system of models. Thus, for the linear equations:
$$\begin{aligned} x + y &= 2 \\ -x + 3y &= 4 \end{aligned}$$

Table 3.7: Data for Question 5.

Plant height (dm)	Soil N (%)	Water index (1-10)	Management type
22.3	12	1	A
21	12.5	2	A
24.7	14.3	3	B
25	14.2	4	B
26.3	15	5	C
22	14	6	C
31		7	D
32	15	8	D
34	13.3	9	E
42	15.2	10	E
28.9	13.6	1	A
33.3	14.7	2	A
35.2	14.3	3	B
36.7	16.1	4	B
34.4	15.8	5	C
33.2	15.3	6	C
35	14	7	D
41	14.1	8	D
43	16.3	9	E
44	16.5	10	E

we have:

$$\boldsymbol{A} = \begin{bmatrix} 1 & 1 \\ -1 & 3 \end{bmatrix}, \boldsymbol{x} = \begin{bmatrix} x \\ y \end{bmatrix}, \text{ and } \boldsymbol{b} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}.$$

Thus, we have

$$\boldsymbol{A}^{-1}\boldsymbol{b} = \boldsymbol{x} = \begin{bmatrix} 1/2 \\ 3/2 \end{bmatrix}.$$

Given this framework, solve the system of equations below with linear algebra using **R**.

$$\begin{aligned} 3x + 2y - z &= 1 \\ 2x - 2y + 4z &= -2 \\ -x + 0.5y - z &= 0 \end{aligned}$$

8. Complete the following exercises concerning the **R** contributed package *asbio*:

(a) Install[15] and load the package *asbio* for the current work session.
(b) Identify
(c) Access the help file for `bplot()` (a function in *asbio*).
(d) Load the dataset `fly.sex` from *asbio*.
(e) Obtain documentation for the dataset `fly.sex` and describe the dataset variables.
(f) Access the column `longevity` in *fly.sex* using the function `with()`.

9. Create .csv and .txt datasets, place them in your working directory, and read them into **R**.

10. How does **R** uses symbols? Create a symbol and explain its characteristics.

11. Create the lists `list1 <- list(a = c(1, 2, 3), b = c("a", "b")); list2 <- list1`.
    (a) Show that the lists and contents of these lists have the same addresses using `lobstr::rep()`.
    (b) Modify `list2` by running `list2$c <- "stuff"`. What happened to the addresses of `list1` and `list2` and their contents? Use the term *copy-on-modify* correctly in your answer.

12. Create an object containing a 1000 random outcomes, `x`, and a list, `listob`, containing `x`, using `x <- runif(10^3); listob <- list(x)`.
    (a) Find the size of `x` and `listob` using `lobstr::obj_size()`.
    (b) Create `listob2 <- list(x, x, x, x)`. Explain why the memory size of `listob` and `lostob2` are so similar despite the fact that `lostob` contains one copy of `x`, and `lostob2` contains four.

13. Define the term *global character pool*.

[15] Installation of packages while knitting of **R** Markdown or Sweave **R** code chunks is not allowed. Instead, one should install packages from the console. Required packages can (and should) be *loaded* while knitting once they are installed.

Chapter 4

Basic Data Management

"I think, therefore I R*."*

- **William B. King**, *Psychologist and* R *enthusiast*

An important characteristic of **R** is its capacity to efficiently manage and analyze large, complex, datasets. In this chapter I list a few functions and approaches useful for data management in base **R**. Data management considerations for the *tidyverse* are given in Chapter 5.

4.1 Operations on Arrays, Lists and Vectors

Operators can be applied individually to every row or column of an array, or every component of a list or atomic vector using a number of time saving methods.

4.1.1 The `apply` Family of Functions

4.1.1.1 `apply()`

Operations can be performed quickly on rows and columns of two dimensional arrays with the function `apply()`. The function requires three arguments.

- The first argument, `X`, specifies an array to be analyzed.
- The second argument, `MARGIN`, connotes whether rows or columns are to be analyzed. `MARGIN = 1` indicates rows, `MARGIN = 2` indicates columns, whereas `MARGIN = c(1, 2)` indicates rows and columns.
- The third argument, `FUN`, defines a function to be applied to the margins of the object in the first argument.

Example 4.1.
Consider the `asbio::bats` dataset which contains forearm length data, in millimeters, for northern myotis bats (*Myotis septentrionalis*), along with corresponding bat ages in in days.

```
library(asbio)
data(bats)
head(bats)
```

```
  days forearm.length
1    1           10.5
2    1           11.0
3    1           12.3
4    1           13.7
5    1           14.2
6    1           14.8
```

Here we obtain minimum values for the `days` and `forearm.length` columns.

```
apply(bats, 2, min)
```

```
          days forearm.length
           1.0           10.5
```

It is straightforward to change the third argument in `apply()` to obtain different summaries, like the mean.

```
apply(bats, 2, mean)
```

```
          days forearm.length
        13.579         23.603
```

or the standard deviation

```
apply(bats, 2, sd)
```

```
          days forearm.length
       12.4610         8.4347
```

Several summary statistical functions exist for numerical arrays that can be used in some instances in the place of `apply()`. These include `rowMeans()` and `colMeans()` which give the sample means of specified rows and columns, respectively, and `rowSums()` and `colSums()` which give the sums of specified rows and columns, respectively. For instance:

```
colMeans(bats)
```

```
          days forearm.length
        13.579         23.603
```

■

4.1.1.2 `lapply()`

The function `lapply()` allows one to sweep functions through list components. It has two main arguments:

- The first argument, `X`, specifies a list to be analyzed.
- The second argument, `FUN`, defines a function to be applied to each element in `X`.

Example 4.2.

Consider the following simple list, whose three components have different lengths.

```
x <- list(a = 1:8, norm.obs = rnorm(10),
          logic = c(TRUE, TRUE, FALSE, FALSE))
x
```

```
$a
[1] 1 2 3 4 5 6 7 8

$norm.obs
 [1] -1.28372 -0.75693  0.59059 -1.59485 -0.88787 -0.25026  1.05575
 [8] -0.43196  2.05223  0.64020

$logic
[1]  TRUE  TRUE FALSE FALSE
```

Here we sweep the function `mean()` through the list:

```
lapply(x, mean)
```

```
$a
[1] 4.5

$norm.obs
[1] -0.086682

$logic
[1] 0.5
```

Note the Boolean outcomes in `logic` have been coerced to numeric outcomes. Specifically, `TRUE = 1` and `FALSE = 0`. Here are the 1st, 2nd (median), and 3rd quartiles of `x`:

```
lapply(x, quantile, probs = 1:3/4)
```

```
$a
 25%  50%  75%
2.75 4.50 6.25
```

```
$norm.obs
     25%      50%      75%
-0.85514 -0.34111  0.62780

$logic
25% 50% 75%
0.0 0.5 1.0
```

■

4.1.1.3 sapply()

The function `sapply()` is a user friendly wrapper for `lapply()` that can return a vector or array instead of a list.

```
sapply(x, quantile, probs = 1:3/4)
```

```
       a norm.obs logic
25% 2.75 -0.85514   0.0
50% 4.50 -0.34111   0.5
75% 6.25  0.62780   1.0
```

4.1.1.4 tapply()

The `tapply()` function allows summarization of a one dimensional array (e.g., a column or row from a matrix) with respect to levels in a categorical variable. The function requires three arguments.

- The first argument, `X`, defines a one dimensional array to be analyzed.
- The second argument, `INDEX` should provide a list of one or more factors (see example below) with the same length as `X`.
- The third argument, `FUN`, is used to specify a function to be applied to `X` for each level in `INDEX`.

Example 4.3.
Consider the dataset `asbio::heart`, which documents pulse rates for twenty four subjects at four time periods following administration of a experimental treatment. These were two active heart medications and a control. Here are average heart rates for the treatments.

```
data(heart)
with(heart, tapply(rate, drug, mean))
```

```
  AX23   BWW9   Ctrl
76.281 81.031 71.906
```

Here are the mean heart rates for treatments, for each time frame. Note that the second argument is defined as a list with two components, each of which can be coerced to be a factor.

```
with(heart, tapply(rate, list(drug = drug, time = time), mean))
```

```
      time
drug      t1     t2     t3     t4
  AX23 70.50 80.500 81.000 73.125
  BWW9 81.75 84.000 78.625 79.750
  Ctrl 72.75 72.375 71.500 71.000
```

■

The function `aggregate()` can be considered a more sophisticated extension of `tapply()`. It allows objects under consideration to be expressed as functions of explanatory factors, and contains additional arguments for data specification and time series analyses.

Example 4.4.
Here we use `aggregate()` to get identical (but reformatted) results to the prior example.

```
aggregate(rate ~ drug + time, mean, data = heart)
```

```
   drug time   rate
1  AX23   t1 70.500
2  BWW9   t1 81.750
3  Ctrl   t1 72.750
4  AX23   t2 80.500
5  BWW9   t2 84.000
6  Ctrl   t2 72.375
7  AX23   t3 81.000
8  BWW9   t3 78.625
9  Ctrl   t3 71.500
10 AX23   t4 73.125
11 BWW9   t4 79.750
12 Ctrl   t4 71.000
```

Importantly, the first argument, `rate ~ drug + time` is an object of class `formula`:

```
f.rate <- with(heart, rate ~ drug + time)
class(f.rate)
```

```
[1] "formula"
```

The tilde operator, `~`, allows expression of the formulaic framework: `y ~ model`, where `y` is a response variable and `model` specifies a system of (generally) one or more predictor variables.

Objects of class `formula` have base type `language`:

```
typeof(f.rate)
```

```
[1] "language"
```

The `language` base type is used for unevaluated expressions other than constants and names. Examples include formulae, and local function assignments.

■

4.1.2 `outer()`

Another important function for matrix operations is `outer()`. This algorithm allows creation of an array that contains all possible combinations of two atomic vectors or arrays with respect to a user-specified function. The `outer()` function has three required arguments.

- The first two arguments, `X` and `Y`, define arrays or atomic vectors. `X` and `Y` can be identical if one wishes to examine pairwise operations of the array elements (see example below).
- The third argument, `FUN`, specifies a function to be used in operations.

Example 4.5.
Suppose I wish to find the means of all possible pairs of observations from a numerical vector. I could use the following commands:

```
x <- c(1, 2, 3, 5, 4)
outer(x, x, "+")/2
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]  1.0  1.5  2.0  3.0  2.5
[2,]  1.5  2.0  2.5  3.5  3.0
[3,]  2.0  2.5  3.0  4.0  3.5
[4,]  3.0  3.5  4.0  5.0  4.5
[5,]  2.5  3.0  3.5  4.5  4.0
```

The argument `FUN = "+"` indicates that we wish to add elements to each other. We divide these sums by two to obtain means. Note that the diagonal of the output matrix contains the original elements of `x`, because the mean of a number and itself is the original number. The upper and lower triangles are identical because the mean of elements a and b will be the same as the mean of the elements b and a. Note that the matrix outer product of two vectors `x` and `y` can be obtained using `outer(x, y, "*")` or simply `outer(x, y)` (Section 3.1.2.1).

```
outer(x, x, "*")
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    5    4
[2,]    2    4    6   10    8
[3,]    3    6    9   15   12
```

```
[4,]    5   10   15   25   20
[5,]    4    8   12   20   16
```

```
x %o% x
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    5    4
[2,]    2    4    6   10    8
[3,]    3    6    9   15   12
[4,]    5   10   15   25   20
[5,]    4    8   12   20   16
```

■

4.1.3 `stack()`, `unstack()` and `reshape()`

When manipulating lists and dataframes it is often useful to move between so-called "long" and "wide" data table formats. These operations can be handled with the functions `stack()` and `unstack()`. Specifically, `stack()` concatenates multiple vectors into a single vector along with a factor indicating where each observation originated, whereas `unstack()` reverses this process.

Example 4.6.
Consider the 4 x 4 dataframe below.

```
dataf <- data.frame(matrix(nrow = 4, data = rnorm(16)))
names(dataf) <- c("col1", "col2", "col3", "col4")
dataf
```

```
      col1     col2     col3     col4
1 -0.05811 0.328948  0.22311 -0.92645
2  0.52397 0.013364 -0.17729  0.78345
3  0.22395 1.218781  0.29263 -1.44949
4 -0.41296 0.499720 -0.64941  0.64079
```

Here I stack `dataf` into a long table format.

```
sdataf <- stack(dataf)
sdataf
```

```
      values  ind
1  -0.058110 col1
2   0.523970 col1
3   0.223945 col1
4  -0.412963 col1
5   0.328948 col2
6   0.013364 col2
```

```
7   1.218781 col2
8   0.499720 col2
9   0.223109 col3
10 -0.177291 col3
11  0.292631 col3
12 -0.649410 col3
13 -0.926448 col4
14  0.783446 col4
15 -1.449493 col4
16  0.640787 col4
```

Here I unstack `sdataf`.

```
unstack(sdataf)
```

```
      col1     col2     col3     col4
1 -0.05811 0.328948  0.22311 -0.92645
2  0.52397 0.013364 -0.17729  0.78345
3  0.22395 1.218781  0.29263 -1.44949
4 -0.41296 0.499720 -0.64941  0.64079
```

The function `reshape()` can handle both stacking and unstacking operations. Here I stack `dataf`. The arguments `timevar`, `idvar`, and `v.names` are used to provide recognizable identifiers for the columns in the wide table format, observations within those columns, and responses for those combinations.

```
reshape(dataf, direction = "long",
        varying = list(names(dataf)),
        timevar = "Column",
        idvar = "Column obs.",
        v.names = "Response")
```

```
    Column  Response Column obs.
1.1      1 -0.058110           1
2.1      1  0.523970           2
3.1      1  0.223945           3
4.1      1 -0.412963           4
1.2      2  0.328948           1
2.2      2  0.013364           2
3.2      2  1.218781           3
4.2      2  0.499720           4
1.3      3  0.223109           1
2.3      3 -0.177291           2
3.3      3  0.292631           3
4.3      3 -0.649410           4
1.4      4 -0.926448           1
```

```
2.4      4  0.783446          2
3.4      4 -1.449493          3
4.4      4  0.640787          4
```

■

4.2 Other Simple Data Management Functions

4.2.1 `replace()`

One use the function `replace()` to replace elements in an atomic vector based, potentially, on Boolean logic. The function requires three arguments.

- The first argument, `x`, specifies the vector to be analyzed.
- The second argument, `list`, connotes which elements need to be replaced. A logical argument can be used here as a replacement index.
- The third argument, `values`, defines the replacement value(s).

Example 4.7.
For instance:

```
Age <- c(21, 19, 25, 26, 18, 19)
replace(Age, Age < 25, "R is Cool")
```

```
[1] "R is Cool" "R is Cool" "25"        "26"        "R is Cool" "R is Cool"
```

Of course, one can also use square brackets for this operation.

```
Age[Age < 25] <- "R is Cool"
Age
```

```
[1] "R is Cool" "R is Cool" "25"        "26"        "R is Cool" "R is Cool"
```

■

4.2.2 `which()`

The function which can be used with logical commands to obtain address indices for data storage object.

Example 4.8.
For instance:

```
Age <- c(21, 19, 25, 26, 18, 19)
w <- which(Age <= 21)
w
```

```
[1] 1 2 5 6
```

Elements one, two, and five meet this criterion. We can now subset based on the index `w`.

```
Age[w]
```

```
[1] 21 19 18 19
```

To find which element in `Age` is closest to 24 I could do something like:

```
which(abs(Age - 24) == min(abs(Age - 24)))
```

```
[1] 3
```

■

4.2.3 `sort()`

By default, The function `sort()` sorts data from an atomic vector into an alphanumeric ascending order.

```
sort(Age)
```

```
[1] 18 19 19 21 25 26
```

Data can be sorted in a descending order by specifying `decreasing = TRUE`.

```
sort(Age, decreasing = T)
```

```
[1] 26 25 21 19 19 18
```

4.2.4 `rank()`

The function rank gives the ascending alphanumeric rank of elements in a vector. Ties are given the average of their ranks. This operation is important to rank-based permutation analyses (Aho, 2014, Ch 6).

```
rank(Age)
```

```
[1] 4.0 2.5 5.0 6.0 1.0 2.5
```

The second and last observations were the second smallest in `Age`. Thus, their average rank is 2.5.

4.2.5 `order()`

The function `order()` is similar to `which()` in that it provides element indices that accord with an alphanumeric ordering. This allows one to sort a vector, matrix or dataframe into an

ascending or descending order, based on one or several ordered vectors.

Example 4.9.
Consider the dataframe below which lists plant percent cover data for four plant species at three sites. In accordance with the `field.data` example from Ch 3, plant species are identified with four letter codes, corresponding to the first two letters of the taxa genus and species names.

```
field.data <- data.frame(code = c("ACMI", "ELSC", "CAEL", "TACE"),
                         site1 = c(12, 13, 14, 11),
                         site2 = c(0, 20, 4, 5),
                         site3 = c(20, 10, 30, 0))
field.data
```

```
  code site1 site2 site3
1 ACMI    12     0    20
2 ELSC    13    20    10
3 CAEL    14     4    30
4 TACE    11     5     0
```

Assume that we wish to sort the data with respect to an alphanumeric ordering of species codes. Here we obtain the ordering of the codes

```
o <- order(field.data$code)
o
```

```
[1] 1 3 2 4
```

Now we can sort the rows of `field.data` based on this ordering.

```
field.data[o,]
```

```
  code site1 site2 site3
1 ACMI    12     0    20
3 CAEL    14     4    30
2 ELSC    13    20    10
4 TACE    11     5     0
```

■

4.2.6 `unique()`

To identify unique values in dataset we can use the function `unique()`.

Example 4.10.
Below is an atomic vector listing species from a bird survey on islands in southeast Alaska.

Species ciphers follow the same coding method used in Example 4.9. Note that there are a large number of repeats.

```
AK.bird <- c("GLGU", "MEGU", "DOCO", "PAJA", "COLO", "BUFF", "COGO",
             "WHSC", "TUSW", "GRSC", "GRTE", "REME", "BLOY", "REPH",
             "SEPL", "LESA", "ROSA", "WESA", "WISN", "BAEA", "SHOW",
             "GLGU", "MEGU", "PAJA", "DOCO", "GRSC", "GRTE", "BUFF",
             "MADU", "TUSW", "REME", "SEPL", "REPH", "ROSA", "LESA",
             "COSN", "BAEA", "ROHA")

length(AK.bird)
```

```
[1] 38
```

Applying `unique()` we obtain a listing of the 24 unique bird species.

```
unique(AK.bird)
```

```
 [1] "GLGU" "MEGU" "DOCO" "PAJA" "COLO" "BUFF" "COGO" "WHSC" "TUSW" "GRSC"
[11] "GRTE" "REME" "BLOY" "REPH" "SEPL" "LESA" "ROSA" "WESA" "WISN" "BAEA"
[21] "SHOW" "MADU" "COSN" "ROHA"
```

■

4.2.7 `match()`

Given two vectors, the function `match()` indexes where objects in the second vector appear in the elements of the first vector. For instance:

```
x <- c(6, 5, 4, 3, 2, 7)
y <- c(2, 1, 4, 3, 5, 6)
m <- match(y, x)
m
```

```
[1]  5 NA  3  4  2  1
```

The number 2 (the 1st element in `y`) is the 5th element of `x`, thus the number 5 is put 1st in the vector `m` created by match. The number 1 (the 2nd element of `y`) does not occur in `x` (it is `NA`). The number 4 is the 3rd element of `y` and `x`. Thus, the number 3 is given as the third element of `m`, and so on.

Example 4.11.
The usefulness of `match()` may seem unclear at first, but consider a scenario in which I want to convert species code identifiers in field data into actual species names. The following dataframe is a species list that matches four letter species codes to scientific names. Note that the list contains more species than than the `field.data` dataset used in Example 4.9.

```
species.list <- data.frame(code = c("ACMI", "ASFO", "ELSC", "ERRY", "CAEL",
"CAPA", "TACE"), names = c("Achillea millefolium", "Aster foliaceus",
                           "Elymus scribneri", "Erigeron rydbergii",
                           "Carex elynoides", "Carex paysonis",
                           "Taraxacum ceratophorum"))

species.list
```

```
  code                   names
1 ACMI   Achillea millefolium
2 ASFO        Aster foliaceus
3 ELSC       Elymus scribneri
4 ERRY     Erigeron rydbergii
5 CAEL        Carex elynoides
6 CAPA         Carex paysonis
7 TACE Taraxacum ceratophorum
```

Here I add a column in the `field.data` containing the actual species names using `match()`.

```
m <- match(field.data$code, species.list$code)
field.data.new <- field.data # make a copy of field data
field.data.new$species.name <- species.list$names[m]
field.data.new
```

```
  code site1 site2 site3           species.name
1 ACMI    12     0    20   Achillea millefolium
2 ELSC    13    20    10       Elymus scribneri
3 CAEL    14     4    30        Carex elynoides
4 TACE    11     5     0 Taraxacum ceratophorum
```

■

4.2.8 `which()` and `%in%`

We can use the operator `%in%` in conjunction with the function `which()` to achieve the same results as `match()`.

```
m <- which(species.list$code %in% field.data$code)
field.data.new$species.name <- species.list$names[m]
field.data.new
```

```
  code site1 site2 site3           species.name
1 ACMI    12     0    20   Achillea millefolium
2 ELSC    13    20    10       Elymus scribneri
3 CAEL    14     4    30        Carex elynoides
4 TACE    11     5     0 Taraxacum ceratophorum
```

Note that the arrangement of arguments are reversed in `match()` and `which()`. In the former we have: `match(field.data$code, species.list$code)`. In the latter we have: `which(species.list$code %in% field.data$code)`.

4.3 Matching, Querying and Substituting in Strings

R contains a number of useful methods for handling character string data. Strings will have class and base type `character`.

4.3.1 `strtrim()` and `substr()`

The functions `strtrim()` and `substr()` are useful for extracting subsets from strings or string vectors.

Example 4.12.
For the taxonomic codes in the character vector below, the first capital letter indicates whether a species is a flowering plant (anthophyte) or moss (bryophyte) while the last four letters give species codes (see Example 4.9).

```
plant <- c("A_CAAT", "B_CASP", "A_SARI")
```

Assume that I want to distinguish anthophytes from bryophytes by extracting the first letter. This can be done by specifying 1 in the second `strtrim` argument, `width`.

```
phylum <- strtrim(plant, 1)
phylum
```

```
[1] "A" "B" "A"
```

```
plant[phylum == "A"]
```

```
[1] "A_CAAT" "A_SARI"
```

The function `substr()` is useful for imposing the start and end of strings to be subset. Here I extract string components 3-4 (the first two letters of the genus name).

```
substr(plant, 3, 4)
```

```
[1] "CA" "CA" "SA"
```

■

4.3.2 `strsplit()`

The function `strsplit()` splits a character string into substrings based on user defined criteria. It contains two important arguments.

- The first argument, `x`, specifies the character string to be analyzed.
- The second argument, `split`, is a character criterion that is used for splitting.

Example 4.13.
Below I split the character string `ACMI` in two, based on the space between the words `Achillea` and `millefolium`.

```
ACMI <- "Achillea millefolium"
strsplit(ACMI, " ")
```

```
[[1]]
[1] "Achillea"    "millefolium"
```

Note that the result is a list. To get back to a vector (now with two components), I can use the function `unlist()`.

```
unlist(strsplit(ACMI, " "))
```

```
[1] "Achillea"    "millefolium"
```

Here I split based on the letter `"l"`.

```
strsplit(ACMI, "l")
```

```
[[1]]
[1] "Achi"  ""      "ea mi" ""      "efo"   "ium"
```

Interestingly, letting the `split` criterion equal `NULL` results in spaces being placed between every character in a string.

```
strsplit(ACMI, NULL)
```

```
[[1]]
 [1] "A" "c" "h" "i" "l" "l" "e" "a" " " "m" "i" "l" "l" "e" "f" "o" "l"
[18] "i" "u" "m"
```

We can use this outcome to reverse the order of characters in a string.

```
sapply(lapply(strsplit(ACMI, NULL), rev), paste, collapse = "")
```

```
[1] "muilofellim aellihcA"
```

The function `rev()` provides a reversed version of its first argument, in this case a result from `strsplit()`. The function `paste()` can be use to paste together character strings.

■

Criteria for querying strings can include multiple characters in a particular order, and a particular case:

```
x <- "R is free software and comes with ABSOLUTELY NO WARRANTY"
strsplit(x, "so")
```

```
[[1]]
[1] "R is free "
[2] "ftware and comes with ABSOLUTELY NO WARRANTY"
```

Note that the `"SO"` in `"ABSOLUTELY"` is ignored because it is upper case.

4.3.3 `grep()` and `grepl()`

The functions `grep()` and `grepl()` can be used to identify which elements in a *character vector* have a specified pattern. They have the same first two arguments.

- The first argument, `pattern` specifies a patterns to be matched. This can be a character string, or object coercible to a character string, or a regular expression (see below).
- The second argument, `x`, is a character vector where matches are sought.

Example 4.14.
The function `grep()` returns indices identifying which entries in a vector contain a queried pattern. In the character vector below, we see that entries five and six have the same genus, *Carex*.

```
names = c("Achillea millefolium", "Aster foliaceus",
                          "Elymus scribneri", "Erigeron rydbergii",
                          "Carex elynoides", "Carex paysonis",
                          "Taraxacum ceratophorum")

grep("Carex", names)
```

```
[1] 5 6
```

The function `grepl()` does the same thing with Boolean outcomes.

```
grepl("Carex", names)
```

```
[1] FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE
```

Of course, we could use this information to subset `names`.

```
names[grep("Carex", names)]
```

```
[1] "Carex elynoides" "Carex paysonis"
```

We can also get `grep` to return the values directly by specifying `value = TRUE`.

```
grep("Carex", names, value = TRUE)
```

```
[1] "Carex elynoides" "Carex paysonis"
```

■

4.3.4 gsub()

The function gsub() can be used to substitute text that has a specified pattern. Several of its arguments are identical to grep() and grepl():

- As before, the first argument, pattern, specifies a pattern to be matched.
- The second argument, replacement, specifies a replacement for the matched pattern.
- The third argument, x, is a character vector wherein matches are sought and substitutions are made.

Example 4.15.
Here we substitute "C." for occurrences of "Carex" in names.

```
gsub("Carex", "C.", names)
```

```
[1] "Achillea millefolium"   "Aster foliaceus"
[3] "Elymus scribneri"       "Erigeron rydbergii"
[5] "C. elynoides"           "C. paysonis"
[7] "Taraxacum ceratophorum"
```

■

4.3.5 gregexpr()

The function gregexpr() identifies the start and end of matching sections in a character vector.

Example 4.16.
Here we examine the first two entries in names, looking for the genus *Aster*.

```
gregexpr("Aster", names[c(1:2)])
```

```
[[1]]
[1] -1
attr(,"match.length")
[1] -1
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE
```

```
[[2]]
[1] 1
attr(,"match.length")
[1] 5
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE
```

The output list is cryptic at best and requires some explanation. The first two elements in each of the two list components indicate the character number of the start and end of the matched string. For the first list component, these elements are given the identifier `-1` because `"Achillea millefolium"` does not contain the pattern `"Aster"`. For the second list component, these elements are `1` and `5` because `"Aster"` makes up the first five letters of `"Aster foliaceus"`.

■

4.3.6 Regular Expressions

A number of **R** functions for managing character strings, including `grep()`, `grepl()`, `gregexpr()`, `gsub()`, and `strsplit()`, can can incorporate *regular expressions*. In computer programming, a regular expression (often abbreviated as *regex*) is a sequence of characters that allow pattern matching in text. Regular expressions have developed within a number of programming frameworks including the *POSIX standard* (the Portable Operating System Interface standard), developed by the IEEE, and particularly the language Perl[1]. Regular expressions in **R** include *extended* regular expressions (this is the default for most pattern matching and replacement **R** functions), and *Perl-like* regular expressions.

4.3.6.1 Extended Regular Expressions

Default *extended regular expressions* in **R** use a POSIX framework for commands[2], which includes the use of particular metacharacters. These are: `\`, `|`, `( )`, `[ ]`, `^`, `$`, `.`, `{ }`, `*`, `+`, and `?`. The metacharacters will vary in meaning depending if they occur outside of square brackets, `[` and `]`, or inside of square brackets. The former usage means that they are part of a character class (see below). In non-bracketed usage, the metacharacters in the subset below have the following applications (see https://www.pcre.org/original/pcre.txt):

- `^` start of string or line.
- `$` end of string or line.
- `.` match any character except newline.

[1]The Perl programming language was introduced by Larry Wall in 1987 as a Unix scripting tool to facilitate report processing (Wikipedia, 2023d). Despite criticisms as an awkward language, Perl remains widely used for its regular expression framework and string parsing capabilities.

[2]Specifically, they use a version of the POSIX 1003.2 standard.

- | start of alternative branch.
- ( ) start and end subpattern.
- { } start and end min/max repetition specification.

Several regular expression metacharacters can be placed at the end of the end of a regular expression to specify repetition. For instance, "*" indicates the preceding pattern should be matched zero or more times, "{+}" indicates the preceding pattern should be matched one or more times, "{n}" indicates the preceding pattern should be matched exactly n more times, and "{n,}" indicates the preceding pattern should be matched n or more times.

Example 4.17.
We will use the function regmatches(), which extracts or replaces matched substrings from gregexpr() summaries, to illustrate.

```
string <- "%aaabaaab"
ID <- gregexpr("a{1}", string)
regmatches(string, ID)
```

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

```
ID <- gregexpr("a{2}", string)
regmatches(string, ID)
```

```
[[1]]
[1] "aa" "aa"
```

```
ID <- gregexpr("a{2,}", string)
regmatches(string, ID)
```

```
[[1]]
[1] "aaa" "aaa"
```

■

Example 4.18.
Metacharacters can be used together. For instance, the code below demonstrates how one might get rid of one or more extra spaces at the end of character strings.

```
string <- c("###Now is the time      ",
            "# for all  ",
            "#",
            " good men",
            "### to come to the aid of their country.       ")

out <- gsub(" +$", "", string) # drop extra space(s) at end of strings
```

```
out <- gsub("^#*","", out) # drop pound sign(s)

paste(out, collapse = "")
```

```
[1] "Now is the time for all good men to come to the aid of their country."
```

■

Example 4.19.
As a biological example, microbial "taxa" identifiers can include cryptic Amplicon Sequence Variant (ASV) codes, followed by a general taxonomic assignment. For example, here is an ASV identifier for a bacterium within the family Comamonadaceae.

```
asv <- "6abc517aa40e9e7b9c652902fe04bb1a:f__Comamonadaceae"
```

We can delete the ASV code, which ends in a colon, with:

```
gsub(".*:", "", asv)
```

```
[1] "f__Comamonadaceae"
```

The regex script in the first argument means: "match any character string occurring zero or more times that ends in `:`".

■

Example 4.20.
As another example, **R** Markdown delimits monospace font using accent grave metacharacters, `` ` ``, while LaTeX applies this font between the expression `\texttt{` and `}`. Below I convert a **R** Markdown-style character vector containing some monospace strings to a LaTeX-style character vector.

```
char.vec <- c("`+`", "addition", "$2 + 2$", "`2 + 2`")
gsub("(`)(.*)(`)","\\\\texttt{\\2}", char.vec)
```

```
[1] "\\texttt{+}"     "addition"       "$2 + 2$"        "\\texttt{2 + 2}"
```

With the code

```
"(`)(.*)(`)"
```

I subset **R** Markdown strings in `char.vec` into three potential components: 1) the `` ` `` metacharacter beginning the string, 2) the text content between `` ` `` metacharacters, and 3) the closing `` ` `` metacharacter itself. I insert the content in item 2 (indicated as `\\2`) between `\texttt{` and `}` using:

```
"\\\\texttt{\\2}"
```

■

Importantly, Example 4.20 illustrates the procedure to use if a queried character is itself a general expression metacharacter. For instance, the backslash in `\texttt`. In this case, the metacharacter must be escaped using single or double backslashes. That is, `\texttt` must be specified as `\\\\texttt` in `gsub()`.

Example 4.21.
Here I ask for a string split based on the appearance of `?` (which is a regex metacharacter) and `%` (which is not).

```
string <- "m?2%b"
strsplit(string, "[\\?%]")
```

```
[[1]]
[1] "m" "2" "b"
```

■

Character class A regular expression *character class* is comprised of a collection of characters, specifying some query or pattern, situated between quotes (single or double) and square brace metacharacters, e.g., `"["` and `"]"`. Thus, the code `"[\\?%]"` in the previous example defines a character class. Character class pattern matches will be evaluated for any single character in the specified text. The reverse will occur if the first character of the pattern is the regular expression caret metacharacter, `^`. For example, the expression `"[0-9]"` matches any single numeric character in a string, (the regular expression metacharacter `-` can be used to specify a range) and `[^abc]` matches anything *except* the characters `"a"`, `"b"` or `"c"`.

Example 4.22.
Consider the following examples:

```
string <- "a1c&m2%b"
strsplit(string, "[0-9]")
```

```
[[1]]
[1] "a"   "c&m" "%b"
```

```
strsplit(string, "[^abc]")
```

```
[[1]]
[1] "a" "c" ""  ""  ""  "b"
```

■

Example 4.23.
This regular expression will match most email addresses:

```
pattern <- "[-a-z0-9_.%]+\\@[-a-z0-9_.%]+\\.[a-z]+"
```

The expression literally reads: "1) find one or more occurrences of characters in a-z or A-Z or 0-9 or dashes or periods, followed by 2) the ampersand symbol (literally), followed by 3) one or more occurrences of characters in a-z or A-Z or 0-9 or dashes or periods, followed by 4) a literal period, followed by one or more occurrences of the letters a-z or A-Z." Here is a string we wish to query:

```
string <- c("abc_noboby@isu.edu",
            "text with no email",
            "me@mything.com",
            "also",
            "you@yourspace.com",
            "@you"
            )
```

We confirm that elements 1, 3, and 5 from `string` are email addresses.

```
grep(pattern, string, ignore.case = TRUE, value = TRUE)
```

```
[1] "abc_noboby@isu.edu" "me@mything.com"     "you@yourspace.com"
```

■

Certain character classes are predefined. These classes have names that are bounded by two square brackets and colons, and include `"[[:lower:]]"` and `"[[:upper:]]"` which identify lower and upper case letters, `"[:punct:]"` which identifies punctuation, `[[:alnum:]]`, which identifies all alphanumeric characters, and `"[[:space:]]"`, which identifies space characters, e.g., tab and newline.

```
string <- c("M2Ab", "def", "?", "%", "\n")
grepl("[[:lower:]]", string)
```

```
[1]  TRUE  TRUE FALSE FALSE FALSE
```

```
grepl("[[:upper:]]", string)
```

```
[1]  TRUE FALSE FALSE FALSE FALSE
```

```
grepl("[[:punct:]]", string)
```

```
[1] FALSE FALSE  TRUE  TRUE FALSE
```

```
grepl("[[:space:]]", string) # item five is a newline request
```

```
[1] FALSE FALSE FALSE FALSE  TRUE
```

Here I ask **R** to return elements from `string` that are three or more characters long.

```
grep("[[:alnum:]]{3}", string, value = TRUE)
```

```
[1] "M2Ab" "def"
```

4.3.6.1.1 Turning off regular expressions

For some pattern matching and replacement jobs it may be best turn off the default extended regular expressions and use exact matching by specifying `fixed = TRUE`. For example, **R** may place periods in the place of spaces in character strings and column names in dataframes and arrays.

Example 4.24.
Consider the following example:

```
countries <- c("United.States", "United.Arab.Emirates", "China", "Germany")
gsub(".", " ", countries)
```

```
[1] "              "         "                       " "      "
[4] "       "
```

Note that using `gsub(".", " ", countries)` results in the replacement of all text with spaces because of the meaning of the period metacharacter. To get the desired result we could use:

```
gsub(".", " ", countries, fixed = TRUE)
```

```
[1] "United States"        "United Arab Emirates" "China"
[4] "Germany"
```

Of course we could also double escape the period.

```
gsub("\\.", " ", countries)
```

```
[1] "United States"        "United Arab Emirates" "China"
[4] "Germany"
```

■

4.3.6.2 Perl-like Regular Expressions

The **R** character string functions `grep()`, `grepl()`, `regexpr()`, `gregexpr()`, `sub()`, `gsub()`, and `strsplit()` allow *Perl-like regular expression* pattern matching. This is done by specifying `perl = TRUE`, which switches regular expression handling to the *PRCE* package. Perl allows

handling of the POSIX predefined character classes, e.g., `"[[:lower:]]"`, along with a wide variety of other calls which are generally implemented using metacharacters and double backslash commands. Here are some examples.

- `\\d` any decimal digit.
- `\\D` any character that is not a decimal digit.
- `\\h` any horizontal white space character (e.g., tab, space).
- `\\H` any character that is not a horizontal white space character.
- `\\s` any white space character.
- `\\S` any character that is not a white space character.
- `\\v` any vertical white space character (e.g., newline).
- `\\V` any character that is not a vertical white space character.
- `\\w` any word, i.e., letter or character components separated by white space.
- `\\W` any non word.
- `\\b` a word boundary.
- `\\U` upper case character (dependent on context).
- `\\L` lower case character (dependent on context).

Note that reversals in meaning occur for capitalized and uncapitalized commands.

Example 4.25.

Here we identify string entries containing numbers.

```
string <- c("Acidobacteria", "Actinobacteria", "TM7.1", "Gitt-GS-136",
            "Chloroflexia", "Bacili")

grep("\\d", string, perl = TRUE)
```

```
[1] 3 4
```

And those containing non-numeric characters (i.e., all of the entries).

```
grep("\\D", string, perl = TRUE)
```

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

To subset non-numeric entries, one could do something like:

```
string[-grep("\\d", string, perl = TRUE)]
```

```
[1] "Acidobacteria"  "Actinobacteria" "Chloroflexia"   "Bacili"
```

■

Example 4.26.
As a slightly extended example we will count the number of words in the description of the

GNU public licences in **R** (obtained via `RShowDoc("COPYING")`). Ideas here largely follow from the function `DescTools::StrCountW()` (Signorell, 2023).

Text can be read from a connection using the function `readLines()`.

```
GNU <- readLines(RShowDoc("COPYING"))
head(GNU)
```

```
[1] "\t\t    GNU GENERAL PUBLIC LICENSE"
[2] "\t\t       Version 2, June 1991"
[3] ""
[4] " Copyright (C) 1989, 1991 Free Software Foundation, Inc."
[5] "                          51 Franklin St, Fifth Floor, Boston, MA  02110-
1301  USA"
[6] " Everyone is permitted to copy and distribute verbatim copies"
```

Note that the escaped command `\t` represent the ASCII (American character encoding standard) control character for tab return. Other useful escaped control characters include `\n`, indicating new line or carriage return.

To search for words, we will actually identify string components that are not words, identified with the Perl regex command `\\W` and word boundaries, i.e., `\\b`. We can combine these summarily as: `\\b\\W+\\b`. The call `\\W+` indicates a non-word match occurring one or more times. Here we apply this regular expression to the first element of `GNU`.

```
GNU[1]
```

```
[1] "\t\t    GNU GENERAL PUBLIC LICENSE"
```

```
gregexpr("\\b\\W+\\b", GNU[1], perl = TRUE)
```

```
[[1]]
[1] 10 18 25
attr(,"match.length")
[1] 1 1 1
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE
```

Matches occur at three locations, 10, 18, and 25, which separate the four words `GNU GENERAL PUBLIC LICENSE`. Thus, to analyze the entire document we could use:

```
sum(sapply(gregexpr("\\b\\W+\\b", GNU, perl = TRUE),
           function(x) sum(x > 0)) + 1)
```

```
[1] 3048
```

There are 3048 total words in the license description.

■

One can identify substrings by number using Perl.

Example 4.27.
In this example, I subdivide a string into two components, the first character, i.e., `"(\\w)"`, and the remaining zero or more characters: `"(\\w*)"`. These are referred to in the `substitute` argument of `gsub` as items `\\1` and `\\2`, respectively. Capitalization for these substrings are handled in different ways below.

```
string <- "achillea"
gsub("(\\w)(\\w*)", "\\U\\1\\U\\2", string, perl=TRUE) # all caps
```

```
[1] "ACHILLEA"
```

```
gsub("(\\w)(\\w*)", "\\L\\1\\U\\2", string, perl=TRUE) # low, then upper case
```

```
[1] "aCHILLEA"
```

```
gsub("(\\w)(\\w*)", "\\U\\1\\L\\2", string, perl=TRUE) # up, then lower case
```

```
[1] "Achillea"
```

The functions `tolower()` and `toupper()` provide simpler approaches to convert letters to lower and upper case, respectively.

```
toupper(string)
```

```
[1] "ACHILLEA"
```

■

4.4 Date-Time Classes

There are two basic **R** date-time classes, *POSIXlt* and *POSIXct*[3]. Class POSIXct represents the (signed) number of seconds since the beginning of 1970 (in the UTC time zone) as a numeric vector. An object of class POSIXlt will be comprised of a list of vectors with the names `sec`, `min`, `hour`, `mday` (day of month), `mon` (month), `year`, `wday` (day of week), and `yday` (day of year).

POSIX naming conventions include:

- `%m` = Month as a decimal number (01–12).
- `%d` = Day of the month as a decimal number (01–31).
- `%Y` = Year. Designations in `0:9999` are accepted.
- `%H` = Hour as a decimal number (00–23).
- `%M` = Minute as a decimal number (00–59

[3] Recall the POSIX prefix refers to the IEEE standard Portable Operating System Interface.

Example 4.28.
As an example, below are twenty dates and corresponding binary water presence measures (0 = water absent, 1 = water present) recorded at 2.5 hour intervals for an intermittent stream site in southwest Idaho (Aho et al., 2023a).

```
dates <- c("08/13/2019 04:00", "08/13/2019 06:30", "08/13/2019 09:00",
           "08/13/2019 11:30", "08/13/2019 14:00", "08/13/2019 16:30",
           "08/13/2019 19:00", "08/13/2019 21:30", "08/14/2019 00:00",
           "08/14/2019 02:30", "08/14/2019 05:00", "08/14/2019 07:30",
           "08/14/2019 10:00", "08/14/2019 12:30", "08/14/2019 15:00",
           "08/14/2019 17:30", "08/14/2019 20:00", "08/14/2019 22:30",
           "08/15/2019 01:00", "08/15/2019 03:30")

pres.abs <- c(1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1)
```

To convert the character string `dates` to a date-time object we can use the function `strptime()`. We have:

```
dates.ts <- strptime(dates, format = "%m/%d/%Y %H:%M")
class(dates.ts)
```

```
[1] "POSIXlt" "POSIXt"
```

Note that the dates can now be evaluated numerically.

```
dates.df <- data.frame(dates = dates.ts, pres.abs = pres.abs)
summary(dates.df)
```

```
     dates                         pres.abs
 Min.   :2019-08-13 04:00:00   Min.   :0.00
 1st Qu.:2019-08-13 15:52:30   1st Qu.:0.75
 Median :2019-08-14 03:45:00   Median :1.00
 Mean   :2019-08-14 03:45:00   Mean   :0.75
 3rd Qu.:2019-08-14 15:37:30   3rd Qu.:1.00
 Max.   :2019-08-15 03:30:00   Max.   :1.00
```

I can also easily extract time series components.

```
dates.ts$mday # day of month
```

```
 [1] 13 13 13 13 13 13 13 13 14 14 14 14 14 14 14 14 14 14 15 15
```

```
dates.ts$wday # day of week
```

```
 [1] 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 4 4
```

```
dates.ts$hour # hour
```

```
 [1]  4  6  9 11 14 16 19 21  0  2  5  7 10 12 15 17 20 22  1  3
```

■

Exercises

1. Using the `plant` dataset from Question 5 in the Exercises at the end of Chapter 3, perform the following operations.
   (a) Attempt to simultaneously calculate the column means for plant height and soil % N using `FUN = mean` in `apply()`. Was there an issue? Why?
   (b) Eliminate missing rows in `plant` using `na.omit()` and repeat (a). Did this change the mean for plant height? Why?
   (c) Modify the `FUN` argument in `apply()` to be: `FUN = function(x) mean(x, na.rm = TRUE)`. This will eliminate `NA`s on a column by column basis.
   (d) Compare the results in (a), (b), (c). Which is the best approach? Why?
   (e) Find the mean and variance of plant heights for each Management Type in `plant` using `tapply()`. Use the best practice approach for `FUN`, as deduced in (d).

2. For the questions below, use the list object `list.data`.
   (a) Use `sapply(list.data, FUN = length)` to get the number of components in each element of `list.data`.
   (b) Repeat (a) using `lapply()`. How is the output in (b) different from (a)?

```
list.data <- list(a = 1:9, height = rnorm(50),
                  greet = c("hello", "goodbye", "hello"))
```

3. A frequently used statistical application is the calculation of all possible mean differences. Assume that we have arithmetic means for the treatments `trt1`, `trt2`, `trt3`, `trt4` and `trt5`, given in the object `means` below.
   (a) Calculate all possible mean differences using `means` as the first two arguments in `outer()`, and letting `FUN = "-"`.
   (b) Extract meaningful and non-redundant differences by using `upper.tri()` or `lower.tri()` (Section 3.4.4). There should be $\binom{5}{2} = 10$ meaningful (not simply a mean subtracted from itself) and non-redundant differences.

```
means <- c(trt1 = 20.5, trt2 = 15.3, trt3 = 22.1, trt4 = 30.4,
           trt5 = 28)
```

4. Using the `plant` dataset from Question 5 in the Exercises for Chapter 3, perform the following operations.
   (a) Use the function `replace()` to identify samples with soil N less than 13.5% by identifying them as `"Npoor"`.

(b) Use the function `which()` to identify which plant heights are greater than or equal to 33.2 dm.
(c) Sort plant heights using the function `sort()`.
(d) Sort the `plant` dataset with respect to ascending values of plant height using the function `order()`.

5. Using `match()` or `which` and `%in%`, replace the code column names in the dataset `cliff.sp` from the package *asbio*, with the correct scientific names (genus and specific epithet) from the dataframe `sp.list` below.

```
sp.list <- data.frame(code = c("L_ASCA","L_CLCI","L_COSPP","L_COUN",
"L_DEIN","L_LCAT", "L_LCST","L_LEDI","M_POSP","L_STDR","L_THSP",
"L_TOCA","L_XAEL","M_AMSE", "M_CRFI","M_DISP","M_WECO","P_MIGU",
"P_POAR","P_SAOD"),
sci.name = c("Aspicilia caesiocineria","Caloplaca citrina",
"Collema spp.", "Collema undulatum", "Dermatocarpon intestiniforme",
"Lecidea atrobrunnea", "Lecidella stigmatea", "Lecanora dispersa",
"Pohlia sp.", "Staurothele drummondii", "Thelidium species",
"Toninia candida", "Xanthoria elegans", "Amblystegium serpens",
"Cratoneuron filicinum", "Dicranella species", "Weissia controversa",
"Mimulus guttatus", "Poa pattersonii", "Saxifraga odontoloma"))
```

6. Using the `sp.list` dataframe from the previous question, perform the following operations: (a) Apply `strsplit()` to the the column `sp.list$sci.name` to create a two column dataframe with genus and corresponding species names. (b) A two character prefix in the column `sp.list$code` indicates whether a taxon is a lichen (prefix = `"L_"`), a marchantiophyte (prefix = `"M_"`), or a vascular plant (prefix = `"P_"`). Use `grep()` to identify marchantiophytes.

7. Use the string vector `string` below to answer the following questions: (a) Use regular expressions in the `pattern` argument of `gsub()` to get rid of extra spaces at the start of string elements while preserving spaces between words. (b) Use the predefined character class `[[:alnum:]]` and an accompanying quantifier in the `pattern` argument from `grep()` to count the number of words whose length is greater than or equal to four characters.

```
string <- c("   Statistics is ", "      a ", " great topic.")
```

8. Remove the numbers from the character vector below using `gsub()` and an appropriate Perl-like regular expression.

```
x <- c("enzyme1","enzyme12","enzyme3","tRNA1","tRNA205",
   "mRNA6","mRNA17","mRNA8","mRNA100")
```

9. Consider the character vector `times` below, which has the format: `day-month-year`

`hour:minute:second`. (a) Convert `times` into an object of class `POSIXlt` called `time.pos` using the function `strptime()`. (b) Extract the day of the week from `time.pos`. (c) Sort `time.pos` using `sort()` to verify that `time.pos` is quantitative.

```
times <- c("12-12-2023 12:12:20",
       "12-01-2021 01:12:40",
       "15-10-2021 23:10:15",
       "25-07-2022 13:09:45")
```

Chapter 5

Welcome to the Tidyverse

> *"Data is like garbage. You'd better know what you are going to do with it before you collect it."*

- **Mark Twain**, *1835 - 1910*

5.1 The Tidyverse

This chapter demonstrates the data management capabilities of the *tidyverse* (Wickham et al., 2019). Thus, Chapter 5 can be considered a *tidyverse* reconsideration of Ch 4. The *tidyverse* is currently a collection of eight core packages (Fig 5.1). These are:

- *dplyr* Grammar and functions for data manipulation.
- *forcats* Tools for solving common problems with factors.
- *ggplot2* A system for "declaratively creating graphics", based on the book *The Grammar of Graphics* (Wilkinson, 2012).
- *purrr* An enhancement of **R**'s functional programming (FP) toolkit.
- *readr* Methods for reading rectangular data.
- *stringr* Functions to facilitate working with strings.
- *tibble* A "modern re-imagining of the data frame."
- *tidyr* A set of functions for "tidying data."

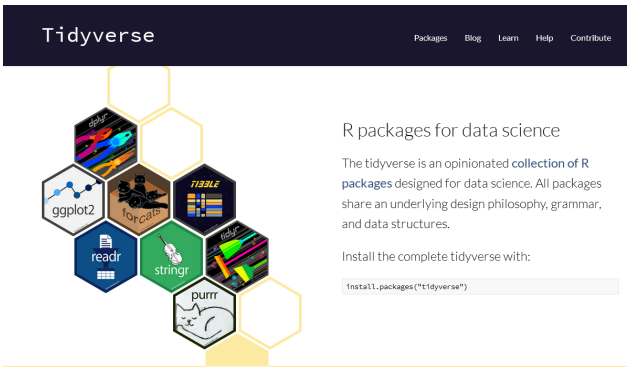


Figure 5.1: The main packages of the *tidyverse*.

The *tidyverse* library also contains several useful ancillary packages, including *lubridate*, *reshape2*, *hms*, *blob*, *margrittr*, and *glue*. While installing *tidyverse* will result in the installation of both main and ancillary packages, loading the *tidyverse* will result only in the complete loading of the eight main *tidyverse* packages.

Importantly, this chapter is not meant to be an authoritative summary of the *tidyverse*. Coverage here is mostly limited to the core data management packages *magrittr*, *tibble*, *dplyr*, *stringer*, and the ancillary packages *lubridate* and *reshape2*. The *tidyverse ggplot2* package is the major focus of Chapter 7. Wickham et al. (2023) provides a succinct but thorough introduction to the *tidyverse* in the open source book *R for Data Science*. Useful *tidyverse* "cheatsheets" can be found here.

The *tidyverse* packages can be downloaded using:

```
install.packages("tidyverse")
```

5.2 Pipes

An important convention of the *tidyverse* is the widespread use of the forward *pipe operator*: |> . In programming, a pipe is a set of commands connected in series, where the output of one command is the input of the next[1]. In many cases, use of pipes allows clearer representations of coding processes[2]. Incidentally, the |> pipe, from the *base* package, is motivated by an older forward pipe operator from the *tidyverse* package *magrittr*, %>%. As of **R** 4.1, the native pipe operator for the *tidyverse* is |> (although %>% will still work if *magrittr* is loaded)[3]. Notably, while |> is more syntactically (and algorithmically) streamlined than %>%, there are several

[1] Pipe programming dates back to early developments in Unix operating systems (Ritchie, 1984; Bell Labs, 2004), wherein pipes are codified as vertical bars "|". Along with Unix/Linux, pipes are widely used in the languages F#, Julia, and JavaScript, among others.

[2] In particular, when you see |> it is helpful to think "and then".

[3] The RStudio shortcut for %>% is **Ctrl**+Shift+m. To force RStudio to default to |> when using **Ctrl**+Shift+m (or some other keyboard shortcut), one can modify appropriate settings in **Tools**>Global Options>Code.

features available to %>% that do not exist for |>, including the potential for a placeholder operator[4]. Nonetheless, I focus on |>, not %>%, here.

Example 5.1.
Consider the circular operation: $\log_e(\exp(1))$. We could write this as,

```
1 |> exp() |> log()
```

```
[1] 1
```

Here the number 1 is piped into the function `exp()`, with the result: $\exp(1) = e^1 = e$, and this outcome is piped into the function `log()`, with the result: $\log_e e = 1$. Because the first arguments of `exp()` and `log()` are simply calls to numeric data, and these are provided by the previous pipe segment, we do not have to include information about `x` for `f(x)` operations. Thus, when functions require only the previous pipe segment result as an argument, then `x |> f()` is equivalent to $f(x)$[5]. In the case that multiple arguments need to be specified, the script `x |> f(y)` is equivalent to $f(x, y)$, and `x |> f(y) |> g(z)` describes $g(f(x, y), z)$. For instance,

```
10 |> log(base = 2)
```

```
[1] 3.3219
```

■

Example 5.2.
This example illustrates that the forward pipe works recursively from the result of the previous pipe segment, using the loblolly pine dataset.

```
head(Loblolly) # First 6 rows of data
```

```
Grouped Data: height ~ age | Seed
   height age Seed
1    4.51   3  301
15  10.89   5  301
29  28.72  10  301
43  41.74  15  301
57  52.70  20  301
71  60.92  25  301
```

[4] In general, the dot placeholder operator, `.`, from *magrittr* allows operations like $f(x, y)$ by specifying `x |> f(.,y)`. For example: `2 %>% log(10, base = .)`. In this script the number 2 will be piped into the `base` argument in the function `log()`.

[5] The %>% forward pipe does not even require the `()` *no argument* designation. That is, `x %>% f` is equivalent to $f(x)$.

```
Loblolly |>
head() |>
tail(2) # Last 2 rows from first 6 rows
```

```
Grouped Data: height ~ age | Seed
   height age Seed
57  52.70  20  301
71  60.92  25  301
```

■

Example 5.3.
We can define the result of a pipe to be a global variable (Sections 2.3.3, 8.2). Consider the script and output below (Fig 5.2).

```
x <- seq(1,10,length=100)
y <- x |> sin()
plot(x, y, type = "l", ylab = "sin(x)", xlab = "x (radians)")
```

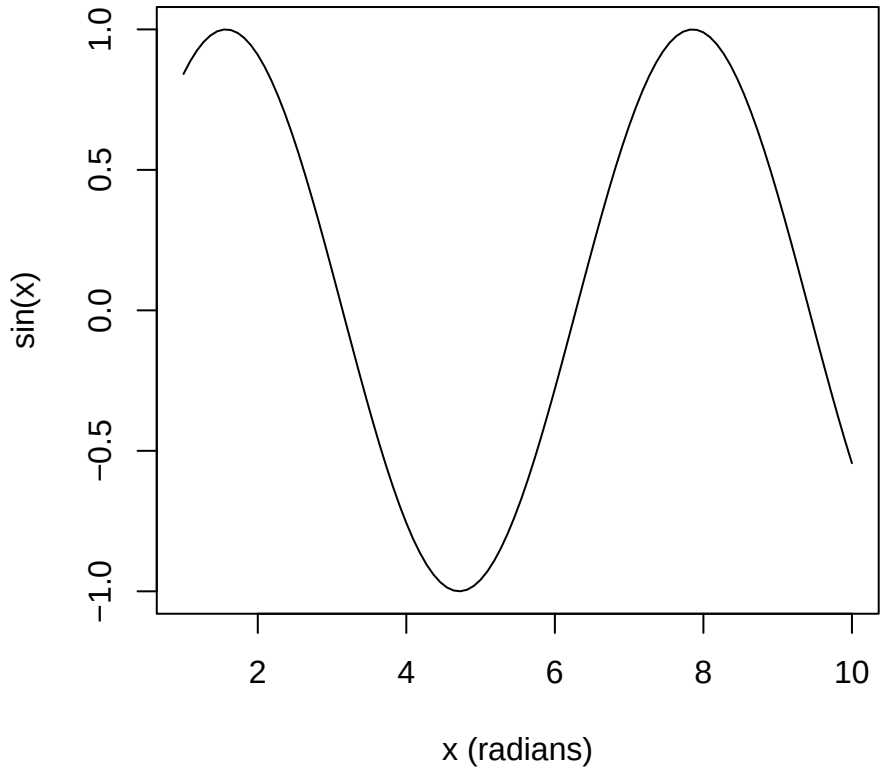


Figure 5.2: Creating a global variable (object) resulting from a pipeline.

We will formally consider the function `plot()` in Ch 6.

■

5.2.1 Other Pipes

It is worth noting that, in addition to %>%, *magrittr* contains several other potentially useful pipe operators. These include the *assignment pipe* and the *tee pipe*. The assignment pipe operator, %<>%, will pipe `x` into one or more `f(x)` expressions, and then assign the result to the name `x`[6]. The tee pipe operator %T>% works like %>%, except the return value in `x %T>% f(x)` is `x` itself. This is useful when a pipeline requires a side-effect like plotting or printing[7].

5.3 tibble

The *tidyverse* package *tibble* provides an alternative to the `data.frame` format of data storage, called a tibble. Tibbles have classes `dataframe` and `tbl_df`, allowing them to posses additional characteristics including enhanced printing (see Example 5.4 below). Additional distinguishing characteristics of tibbles include: 1) a character vector is not automatically coerced to have class `factor`, 2) recycling (see Section 3.1.1) only occurs for an input of length one, and 3) there is no partial matching when `$` is used to index by tibble columns by name[8]. The functions `tibble()` generates tibbles. The function `as_tibble()` coerces a dataframe to be a tibble.

Example 5.4.
Here we compare dataframe and tibble output of the same data.

```
data <- data.frame(numbers = 1:3, letters = c("a","b","c"),
                   date = as.Date(c("2021-12-1", "2021-12-2",
                                    "2021-12-2"),
                                  format = "%Y-%m-%d"))
data
```

```
  numbers letters       date
1       1       a 2021-12-01
2       2       b 2021-12-02
3       3       c 2021-12-02
```

```
library(tidyverse)
datat <- as_tibble(data)
datat
```

```
# A tibble: 3 x 3
  numbers letters date
    <int> <chr>   <date>
```

[6] For instance, `library(magrittr); x <- -4:4; x %<>% abs %>% sort; x` would print the pipe-modified version of `x`.

[7] For instance, `rnorm(20) |> matrix(ncol = 2) %T>% plot |> colSums`. In this case a plot and the sums of columns will both be printed (see Example 5.12).

[8] According to package *tibble*: "...tibbles are lazy and surly: they do less and complain more than base dataframes. This forces problems to be tackled earlier and more explicitly, typically leading to code that is more expressive and robust."

```
1       1 a       2021-12-01
2       2 b       2021-12-02
3       3 c       2021-12-02
```

■

5.4 dplyr

The *dplyr* package contains a collection of core *tidyverse* algorithms for data manipulation[9]. Table 5.1 lists some useful *dplyr* functions.

Table 5.1: Important *dplyr* data management functions.

Function	Usage
`summarise()`	Numerical summaries of variables.
`group_by`	Group a dataframe by a categorical variable.
`filter()`	Subset variables based on outcomes.
`arrange()`	Reorder rows in a dataframe or tibble.
`mutate()`	Creates new variables from functions of existing variables.
`select()`	Selects variables from tibbles or dataframes.

5.4.1 `summarize()`

The function `summarize()`, or equivalently `summarise()`, creates a new data frame with one row for each combination of specified grouping variables. If no groups are given (for instance, in the case that `group_by` is *not* used to group data), dataframe rows will be summaries of all observations in the required input `.data` argument.

Example 5.5.
Here we use `summarize()` to obtain means for loblolly pine height (in feet) and age (in years).

```
Loblolly |>
  summarise(mean.height.ft = mean(height), mean.age.yrs = mean(age))
```

```
  mean.height.ft mean.age.yrs
1         32.364           13
```

■

[9] *dplyr* has largely replaced the now retired *plyr* package.

5.4.2 group_by()

The group_by() function is often used in conjunction with other *dplyr* functions, including summarize(), to provide an underlying grouping framework for data summaries.

Example 5.6.
Here we use group_by() with summarize() to describe the Loblolly height data. Specifically, we will take the mean and the variance of Loblolly$height with respect to categories specified in group_by().

```
Loblolly |>
  group_by(Seed) |>
  summarise(mean.height.ft = mean(height),
            var.height.ft2 = var(height)
            ) |>
  head(5)
```

```
# A tibble: 5 x 3
  Seed  mean.height.ft var.height.ft2
  <ord>          <dbl>          <dbl>
1 329             30.3           443.
2 327             30.6           440.
3 325             31.9           468.
4 307             31.3           494.
5 331             31.0           495.
```

More than only grouping variable can be specified in group_by():

```
Loblolly |>
  group_by(Seed, age) |>
  summarise(mean.height.ft = mean(height),
            var.height.ft2 = var(height)
            ) |>
  head(5)
```

```
`summarise()` has grouped output by 'Seed'. You can override
using the `.groups` argument.
```

```
# A tibble: 5 x 4
# Groups:   Seed [1]
  Seed    age mean.height.ft var.height.ft2
  <ord> <dbl>          <dbl>          <dbl>
1 329       3           3.93             NA
2 329       5           9.34             NA
3 329      10          26.1              NA
4 329      15          37.8              NA
5 329      20          48.3              NA
```

Clearly, `group_by()` and `summarise()` allow more options than the base function `tapply()` (Section 4.1.1.4). The latter function only provides summaries of groups within a single categorical `INDEX`, with respect to a single quantitative vector, and a single user-defined function.

Starting with *dplyr* 1.1.0, we can use the `.by` argument in `summarize` to bypass `group_by()`, although this argument is experimental, and may be deprecated in the future (see `?summarise`).

```
Loblolly |>
  summarise(mean.height.ft = mean(height),
            var.height.ft2 = var(height),
            .by = Seed) |>
  head(5)
```

```
  Seed mean.height.ft var.height.ft2
1  301         33.247         512.50
2  303         34.107         552.24
3  305         35.115         572.51
4  307         31.328         493.83
5  309         33.782         535.12
```

■

5.4.3 `filter()`

The function `filter()` provides a straightforward way to extract dataframe rows based on Boolean operators.

Example 5.7.
Here we obtain rows in `Loblolly` associated with seed type 301.

```
Loblolly |>
  filter(Seed == "301")
```

```
Grouped Data: height ~ age | Seed
   height age Seed
1    4.51   3  301
15  10.89   5  301
29  28.72  10  301
43  41.74  15  301
57  52.70  20  301
71  60.92  25  301
```

Here are `Loblolly` rows associated with `height` responses greater than 60 feet.

```
Loblolly |>
  filter(height > 60)
```

```
Grouped Data: height ~ age | Seed
   height age Seed
71  60.92  25  301
72  63.39  25  303
73  64.10  25  305
75  63.05  25  309
77  60.07  25  315
78  60.69  25  319
79  60.28  25  321
80  61.62  25  323
```

■

5.4.4 `arrange()`

The function `arrange()` orders the rows of a data frame based on the alphanumeric ordering of specified data.

Example 5.8.
Here we use `arrange()` to sort the result from the previous chunk from smallest to largest loblolly pine heights.

```
Loblolly |>
  filter(height > 60) |>
  arrange(height)
```

```
Grouped Data: height ~ age | Seed
   height age Seed
77  60.07  25  315
79  60.28  25  321
78  60.69  25  319
71  60.92  25  301
80  61.62  25  323
75  63.05  25  309
72  63.39  25  303
73  64.10  25  305
```

One can use `arrange(desc())` to sort a dataframe in descending (largest-to-smallest) order.

```
Loblolly |>
  filter(height > 60) |>
  arrange(desc(height))
```

```
Grouped Data: height ~ age | Seed
   height age Seed
73  64.10  25  305
```

```
72  63.39  25  303
75  63.05  25  309
80  61.62  25  323
71  60.92  25  301
78  60.69  25  319
79  60.28  25  321
77  60.07  25  315
```

■

5.4.5 `slice_min()` and `slice_max()`

The helpful *dplyr* functions `slice_min()` and `slice_max()` allow subsetting of dataframe rows by minimum and maximum values in some column, respectively.

Example 5.9.

```
Loblolly |>
  slice_max(height, n = 5)
```

```
Grouped Data: height ~ age | Seed
   height age Seed
73  64.10  25  305
72  63.39  25  303
75  63.05  25  309
80  61.62  25  323
71  60.92  25  301
```

■

5.4.6 `select()`

The `select()` function allows one to select particular variables in a data frame.

Example 5.10.
For instance, here I select `height` from `Loblolly`.

```
Loblolly |>
  select(height) |>
  head()
```

```
   height
1    4.51
15  10.89
```

```
29  28.72
43  41.74
57  52.70
71  60.92
```

■

The `select()` function can be used in more sophisticated ways by combining it with other *dplyr* functions like `starts_with()` and `ends_with()`, or other Boolean operators.

Example 5.11.
He we select the `height` and `age` columns by calling for variable names that start with `"h"` or end with `"e"`.

```
Loblolly |>
  select(starts_with("h"), ends_with("e")) |>
  head(3)
```

```
   height age
1    4.51   3
15  10.89   5
29  28.72  10
```

■

5.4.7 `mutate()`

The function `mutate()` creates new dataframe columns that are functions of existing variables.

Example 5.12.
Below we select the `age` and `height` columns using `select()`, convert height in feet to height in meters using `mutate()`, plot the result as a side-task using the tee pipe, `%T>%` (note the use of the `.` placeholder operator) (Fig 5.3), and then take the column means of `age` and `height`. Note that by default, all columns from the previous pipe segment will be in the `mutate()` output although all columns need not be explicitly mutated. Output columns can be specified using the `mutate()` argument `.keep`.

Note that in base **R** dialect we have could have used: `with(Loblolly, plot(age, height * 0.3048, ylab = 'Height (m)', xlab = 'Age (yrs)'))` on line 6. However, this is quite a bit harder to decipher. The function `plot()` will be formally introduced in Ch 6.

```
library(magrittr) # to access tee pipe

Loblolly |>
  select(c(age, height)) |>
  mutate(height = height * 0.3048) %T>%
```

```
  plot(., ylab = "Height (m)", xlab = "Age (yrs)") |>
  colMeans()
```

```
    age  height
13.0000  9.8647
```

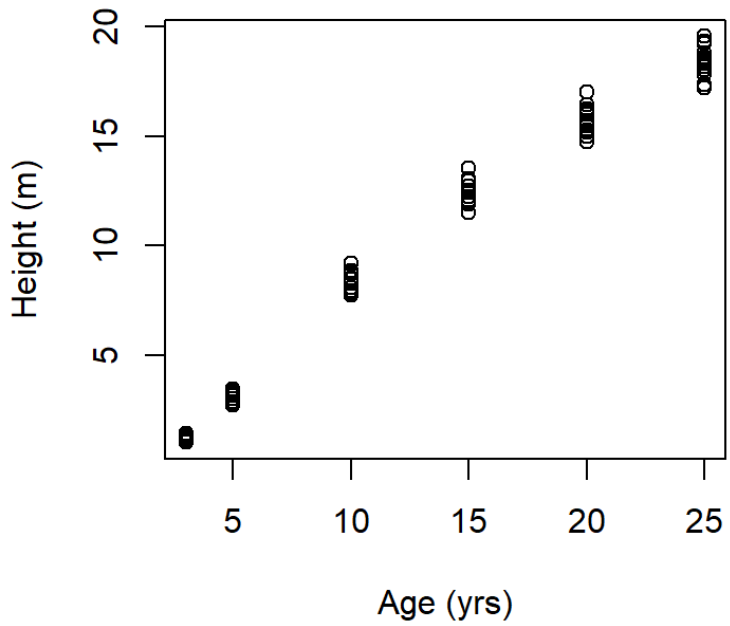


Figure 5.3: Plot of loblolly pine height as a function of age, after converting height to meters.

■

5.4.8 `across()`

The *dplyr* function `across()` allows extensions similar to those in `apply()` wherein the same function can be applied to all columns in the first argument of `across()`. Specifying the first argument in `across()` as `everything()` would allow application of a function to all columns in a dataframe.

Example 5.13.
Here we take the medians of the quantitative columns in `Loblolly` using `across()` and `summarize()`.

```
Loblolly |>
   summarise(across(c(age, height), median))
```

```
   age height
1 12.5     34
```

■

5.5 stringr

As evident in Section 4.3, use of regular expressions for matching, querying and substituting strings can be confusing. The *stringr* package attempts to simplify some of these difficulties. The *stringr* package uses processing tools from the package *stringi* (Gagolewski, 2022) for pattern searching under a wide array of potential approaches. All *stringr* functions have the prefix `str_` and take a character string or string vector as the first argument.

Consider the vector of plant scientific names used to demonstrate string management in Section 4.3.

```
names = c("Achillea millefolium", "Aster foliaceus",
                            "Elymus scribneri", "Erigeron rydbergii",
                            "Carex elynoides", "Carex paysonis",
                            "Taraxacum ceratophorum")
```

Example 5.14.
The function `str_length()` can be used to count the number of characters in a string.

```
str_length(names)
```

```
[1] 20 15 16 18 15 14 22
```

■

Example 5.15.
The function `str_detect()` tests for the presence or absence of a pattern in a string. Here I test for `"Aster"` (the genus *Aster*).

```
str_detect(names, "Aster")
```

```
[1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
```

Here are entries *not* containing *Aster*.

```
str_detect(names, "Aster", negate = TRUE)
```

```
[1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

■

Example 5.16.
Here we subset `names` using the function `stringr::str_subset()` to obtain species within the genus *Carex*.

```
str_subset(names, "Carex")
```

```
[1] "Carex elynoides" "Carex paysonis"
```

■

Example 5.17.
The function `str_replace()` is analogous to the base **R** function `gsub()`. It can be used to replace text based on a pattern.

```
str_replace(names, "Carex", "C.")
```

```
[1] "Achillea millefolium"   "Aster foliaceus"
[3] "Elymus scribneri"       "Erigeron rydbergii"
[5] "C. elynoides"           "C. paysonis"
[7] "Taraxacum ceratophorum"
```

■

Most *stringr* functions work with regular expressions (Section 4.3.6).

Example 5.18.
Here we count upper and lower case vowels with the function `stringr::str_count()` using a pattern defined by the regex character class `[AEIOUaeiou]`.

```
str_count(names, "[AEIOUaeiou]")
```

```
[1] 9 7 5 7 6 5 9
```

and use `stringr::str_extract()` to extract strings nine alphanumeric characters long, and then sort the strings with a pipe.

```
str_extract(names, "[[:alnum:]]{9}") |>
  sort()
```

```
[1] "elynoides" "foliaceus" "millefoli" "rydbergii" "scribneri" "Taraxacum"
```

■

5.6 lubridate

Base **R** approaches for handling date-time data are described in Section 4.4. The package *lubridate* (https://lubridate.tidyverse.org/) contains functions for simplifying and extending some of these operations.

Example 5.19.
As an example dataset, I will use the time series used to illustrate date-time classes in Section 4.4.

```
dates <- c("08/13/2019 04:00", "08/13/2019 06:30", "08/13/2019 09:00",
           "08/13/2019 11:30", "08/13/2019 14:00", "08/13/2019 16:30",
           "08/13/2019 19:00", "08/13/2019 21:30", "08/14/2019 00:00",
           "08/14/2019 02:30", "08/14/2019 05:00", "08/14/2019 07:30",
           "08/14/2019 10:00", "08/14/2019 12:30", "08/14/2019 15:00",
           "08/14/2019 17:30", "08/14/2019 20:00", "08/14/2019 22:30",
           "08/15/2019 01:00", "08/15/2019 03:30")

library(lubridate)
```

We will define the timezone to be timezone of our computer workstation.

```
tz <- Sys.timezone(location = TRUE)
```

The package *lubridate* contains data-time parsers that may be easier to use than the base functions `strptime` and `as.Date`. For the current example, we note that the data are in a month/day/year hour:minute format. So we can create a time series using the function `lubridate::mdy_hm`.

```
date_lub <- mdy_hm(dates, tz = tz)
date_lub
```

```
 [1] "2019-08-13 04:00:00 MDT" "2019-08-13 06:30:00 MDT"
 [3] "2019-08-13 09:00:00 MDT" "2019-08-13 11:30:00 MDT"
 [5] "2019-08-13 14:00:00 MDT" "2019-08-13 16:30:00 MDT"
 [7] "2019-08-13 19:00:00 MDT" "2019-08-13 21:30:00 MDT"
 [9] "2019-08-14 00:00:00 MDT" "2019-08-14 02:30:00 MDT"
[11] "2019-08-14 05:00:00 MDT" "2019-08-14 07:30:00 MDT"
[13] "2019-08-14 10:00:00 MDT" "2019-08-14 12:30:00 MDT"
[15] "2019-08-14 15:00:00 MDT" "2019-08-14 17:30:00 MDT"
[17] "2019-08-14 20:00:00 MDT" "2019-08-14 22:30:00 MDT"
[19] "2019-08-15 01:00:00 MDT" "2019-08-15 03:30:00 MDT"
```

Other *lubridate* parsers include `ymd()`, `ymd_hms()`, `dmy()`, `dmy_hms()`, and `mdy()`. The *lubridate* parsers can often handle mixed methods of data entry. From the `ymd()` documentation we have the following example:

```
x <- c(20090101, "2009-01-02", "2009 01 03", "2009-1-4",
       "2009-1, 5", "Created on 2009 1 6", "200901 !!! 07")
ymd(x)
```

```
[1] "2009-01-01" "2009-01-02" "2009-01-03" "2009-01-04" "2009-01-05"
[6] "2009-01-06" "2009-01-07"
```

■

Lubridate also allows extended mathematical operations for its date-time objects with the functions `duration()`, `period()`, and `interval()`.

Example 5.20.
Duration functions include `dseconds()`, `dminutes()`, `ddays()`, and `dmonths()`.

```
duration("12m", units = "seconds") # seconds in 1 year
```

```
[1] "31557600s (~1 years)"
```

```
dmonths(12)
```

```
[1] "31557600s (~1 years)"
```

```
date_lub[1]
```

```
[1] "2019-08-13 04:00:00 MDT"
```

```
date_lub[1] + ddays(1)
```

```
[1] "2019-08-14 04:00:00 MDT"
```

■

Example 5.21.
Periodic functions include `seconds()`, `minutes()`, `hours()`, and `days()`.

```
days(12) + minutes(2) + seconds(3)
```

```
[1] "12d 0H 2M 3S"
```

```
date_lub[1]
```

```
[1] "2019-08-13 04:00:00 MDT"
```

```
date_lub[1] - days(12)
```

```
[1] "2019-08-01 04:00:00 MDT"
```

■

Example 5.22.
Interval functions include `int_length()`, `int_start()`, and `int_end()`.

```
int <- interval(start = first(date_lub), end = last(date_lub))

int_length(int) |>
  duration()
```

```
[1] "171000s (~1.98 days)"
```

■

5.7 reshape2

Tidyverse functions generally require that data are in a *long table* format. That is, data are stored with columns containing all the values for a particular variable of interest. Unfortunately, this format is not conventional for many scientific applications, particularly longitudinal studies that follow experimental units over time. These will often have a *wide table* format. The *tidyverse reshape2* package contains several functions for converting dataframes from wide to a long table formats, including the functions `reshape2::melt()` and `tidyr::gather()`. The `reshape2::melt.data.frame()` function generates a `value` column based on data commonalities of outcomes given in a variable or variables defined in the `id` argument. A remaining column, if any, that captures these commonalities will be given the name `variable`. The names of the `value` and `variable` output columns can be changed with the arguments `value.name` and `variable.name`, respectively.

Example 5.23.
Consider the `asbio::asthma` dataset, which has a wide table format. The dataset documents the effect of three respiratory treatments (measured as Forced Expiratory Volume in one second (FEV1)) for 24 asthmatic patients over time (`11H - 18H`, i.e, hour 11 to hour 18). A baseline measure of FEV1 (`BASEFEV1`) was also taken 11 hours before application of the treatment.

```
data(asthma)
head(asthma)
```

```
  PATIENT BASEFEV1 FEV11H FEV12H FEV13H FEV14H FEV15H FEV16H FEV17H FEV18H
1     201     2.46   2.68   2.76   2.50   2.30   2.14   2.40   2.33   2.20
2     202     3.50   3.95   3.65   2.93   2.53   3.04   3.37   3.14   2.62
3     203     1.96   2.28   2.34   2.29   2.43   2.06   2.18   2.28   2.29
4     204     3.44   4.08   3.87   3.79   3.30   3.80   3.24   2.98   2.91
5     205     2.80   4.09   3.90   3.54   3.35   3.15   3.23   3.46   3.27
6     206     2.36   3.79   3.97   3.78   3.69   3.31   2.83   2.72   3.00
```

```
  DRUG
1    a
2    a
3    a
4    a
5    a
6    a
```

```
library(reshape2)
asthma.long <- asthma |> melt(id = c("DRUG", "PATIENT"),
                              value.name = "FEV1",
                              variable.name = "TIME")

# here I simplify the names in the TIME variable
asthma.long$TIME <- factor(asthma.long$TIME,
                           labels = c("BASE",
                                      paste("H", 11:18, sep = "")))
head(asthma.long)
```

```
  DRUG PATIENT TIME FEV1
1    a     201 BASE 2.46
2    a     202 BASE 3.50
3    a     203 BASE 1.96
4    a     204 BASE 3.44
5    a     205 BASE 2.80
6    a     206 BASE 2.36
```

In the code above, the function `reshape2::melt()` is used to convert to a long table format, and time designations are simplified using the *base* function `factor()`. The `factor()` function can be used to create a categorical variable with particular levels (Section 3.3), or to change the names of levels. The latter application is used here.

■

Exercises

1. Create a tibble from the `Downs` dataframe shown below. The data comprise part of a report summarizing Down's syndrome cases in British Columbia, compiled by the British Columbia Health Surveillance Registry (Geyer, 1991).
   (a) Examine both the original `Downs` dataframe and the tibble representation of `Downs` by printing them. Do we gain additional information from the tibble?
   (b) Find the mean and variance of the `Age` column from the `Downs` dataset using pipes and *dpylr* functions.

```
Downs <- data.frame(Age = c(17, 20.5, 21.5, 29.5, 30.5, 38.5, 39.5,
                            40.5, 44.5, 45.5, 47),
                    Births = c(13555, 22005, 23896, 15685, 13954,
                               4834, 3961, 2952, 596, 327, 249),
                    Cases = c(16, 22, 16, 9, 12, 15, 30, 31, 22, 11,
                              7)
                    )
```

2. Bring in the `world.emissions` dataset from package *asbio*.
   (a) Using the forward pipe operator, `|>`, and `filter()` from *dplyr*, create a dataframe of just US data.
   (b) Using `|>`, `filter()`, and `summarise()`, find the first and last year of emissions data for the US.
   (c) Using `|>`, `%T>%`, `filter()`, `mutate()`, and `plot()`, plot per capita CO_2 emissions for the US by year (as an intermediate pipeline step) and find the maximum CO_2 emission level. Hint: see Exercise 5.12.
   (d) Using `|>` and `filter()` create a new dataframe called `no.repeats` that eliminates rows with the entry `"redundant"` in the `world.emissions$continent` column.
   (e) With the `no.repeats` dataframe and the functions `group_by()`, and `summarise()`, get mean CO_2 levels for each country over time.
   (f) Using `|>`, `group_by()`, `summarise()` and `slice_max()`, identify the 10 countries with the highest recorded cumulative CO_2 emissions.

3. Consider the character vector `omics` below (Bonnin, 2021).
   (a) Use `stringr::str_detect()` to test for strings with the pattern `"genom"`.
   (b) Using `str_detect()`, test for strings *starting* with the pattern `"genom"` by using an extended regular expression: `^genom` in the `str_detect()` argument `pattern` (see Section 4.3.6.1).
   (c) Using `str_detect()`, test for strings *ending* with the pattern `"omics"` by using an extended regular expression (see Section 4.3.6.1).
   (d) Using `str_subset()`, subset the string vector `omics` to string entries containing the pattern `"genom"`.
   (e) Using `str_replace()`, replace the text `"omics"` with `"ome"`.

```
omics <- c("genomics", "proteomics", "proteome",
           "transcriptomics", "metagenomics", "metabolomics")
```

4. Consider the character vector `times` below, which has the format: `day-month-year hour:minute:second`.
   (a) Convert `times` into a *lubridate* date-time object using an appropriate *lubridate* function.
   (b) Add two days and seven seconds to each entry in `time` using `lubridate::days`.
   (c) Using *lubridate* functions, find the difference, in seconds, between the beginning and the end of the time series.

```
times <- c("12-12-2023 12:12:20",
           "12-01-2021 01:12:40",
           "15-10-2021 23:10:15",
           "25-07-2022 13:09:45")
```

Chapter 6

Base Graphics

> *"Mankind invented a system to cope with the fact that we are so intrinsically lousy at manipulating numbers. It's called the graph."*

- **Charlie Munger**, *businessman and philanthropist*

6.1 Introduction

An important feature of **R** is its capacity to create publication-quality graphics with tremendous user flexibility. **R** graphics are relatively non-interactive, and follow the *painters model* in which later output obscures earlier overlapping output. Thus, "removal" of a graphical feature requires the creation of an entirely new plot. This may feel like a major departure for those used to point-and-click graphics, characteristic of software like Excel® and SigmaPlot®.

There are two general graphics approaches in **R**: *base* graphics and *grid* graphics (Murrell, 2019). Base graphics are applied using the **R** distribution package *graphics*, whereas the grid graphics system relies on low level facilities in the **R** distributed *grid* package, which are generally implemented via high level functions in other packages. The base and grid graphics systems generally do not interact well, although both rely on the distributed *grDevices* package which provides the fundamental infrastructure for **R** graphics, including graphical devices.

The base graphics system is the focus of this chapter. The grid system, and its most popular adherent, the package *ggplot2*, is described in Chapter 7.

6.2 Simple Base Graphics Examples

The base graphics system allows creation of a wide variety of plots for single variables and multiple variables (see Figs 6.1 and 6.2, respectively). Approaches for making many of these example plots are elaborated later in this chapter.

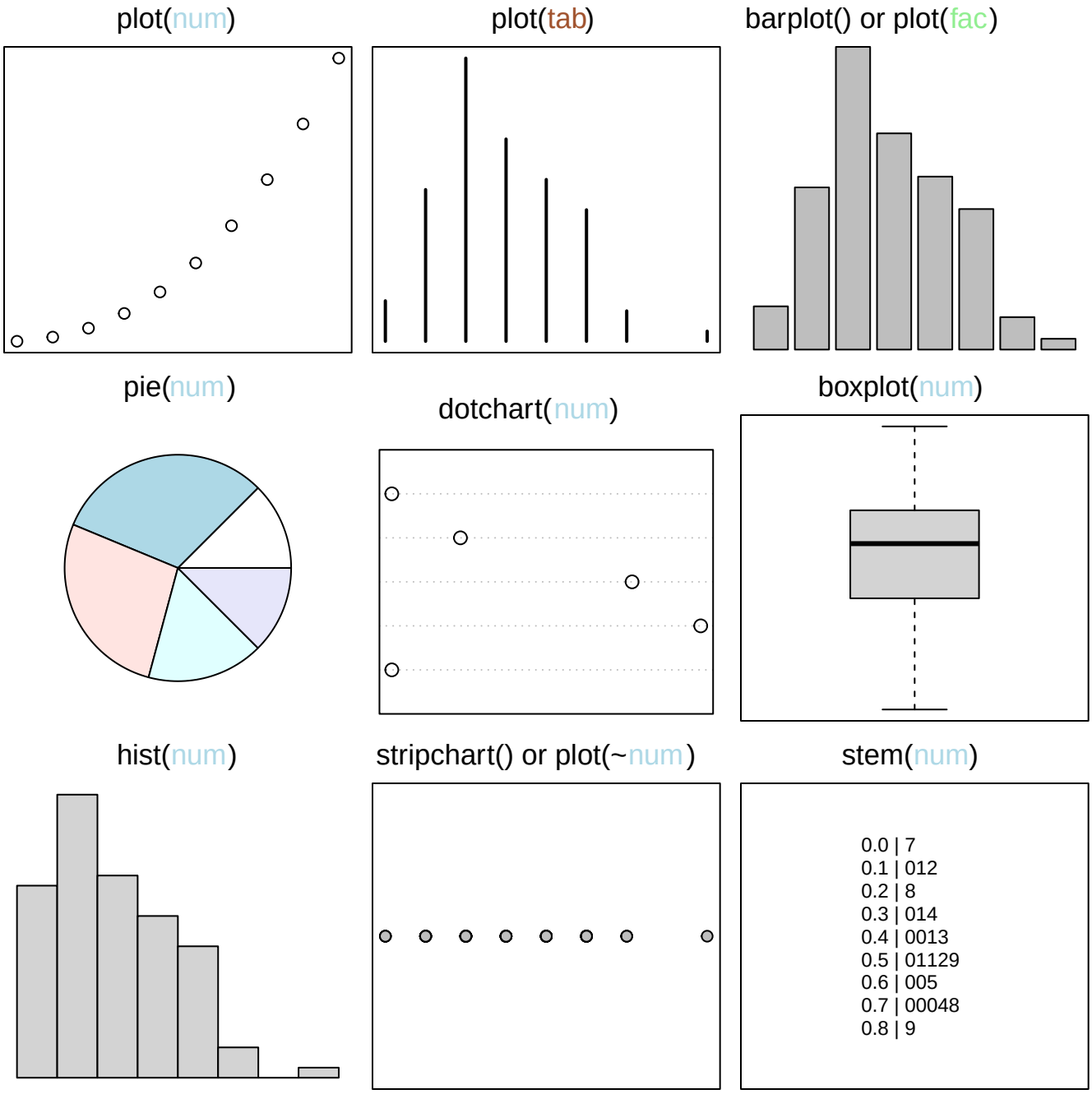


Figure 6.1: Base graphics approaches for single variables. Figure follows Murrell (2019). Classes of plotted objects are distinguished by name and color in main headings: num = `numeric`, tab = `table`, fac = `factor`. By row, from left to right, graphics are: 1) a *scatterplot* created by applying the function `plot()` to a vector of class `numeric`, 2) the `plot()` function applied to a one-dimensional object of class `table`, resulting in a distributional plot, 3) a *barplot*, useful for comparing categorical outcomes, 4) a *pie chart*, 5) a extitdotchart, which provides a dot variant of a bar plot), 6) a extitboxplot, i.e., the interquartile range (hinges) and whiskers delimiting outliers, 7) a *histogram* (the most common graphical distributional summary), 8) a *stripchart*, i.e., a one dimensional scatter plot that provides a horizontal view of distributional outcomes, and 9) a *stem chart*.

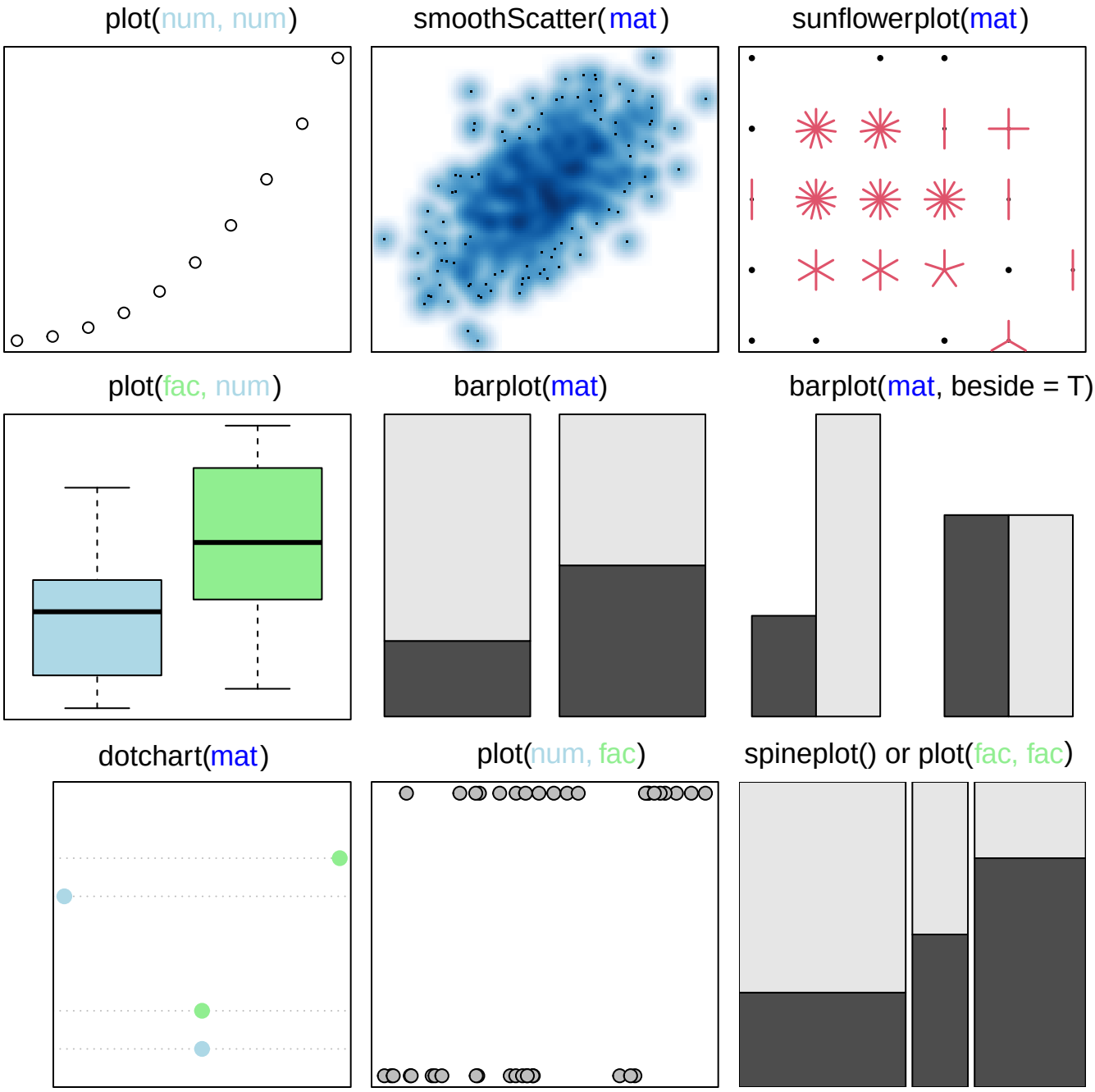


Figure 6.2: Base graphics approaches for considering multiple variables. Figure follows Murrell (2019). Classes of plotted objects are distinguished by name and color in main headings: num = `numeric`, mat = `matrix`, fac = `factor`. By row, from left to right, graphics are: 1) a *scatterplot* based on two `numeric` variables, 2) a *scatterplot* with smoothed densities, , based on a two-column `numeric` `matrix`, 3) a *sunflower plot*, which uses special symbols to indicate overplotting of points, based on a two-column `numeric` `matrix`, 4) a *boxplot* based on a `factor` (with two levels) and a `numeric` variable, 5) and 6) *stacked and beside barplots* based on a `numeric` `matrix`, 7) a dotchart, 8) a stripchart, based on two `numeric` variables, and 9) a *spineplot*, a special cases of a extitmosaic plot (obtained using `mosaicplot()`), representing a generalization of a stacked (or highlighted) bar plot.

6.2.1 plot()

The workhorse of base graphics is the function `plot()`. From Figs 6.1 and 6.2 it is evident that `plot()` can be used in a number of different ways, depending on the characteristics of data being plotted. For example, if data are two numeric vectors, then a conventional scatterplot is created. However, if the first two arguments in `plot()` call a numerical vector and a factor vector (in that order), then a boxplot is created, and if the first two arguments in `plot()` call a factor vector and a numeric vector (in that order), then a stripchart is created. Further, plotting methods for particular classes of objects can be designed that can be implemented by calling `plot()`. For instance, the dendrogram in Fig 6.3 was created using a plotting method called `plot.agnes()`, designed for objects of class `agnes`[1]. However, the function can be run using a generic call to `plot()`. See Ch 8 for additional details on plotting methods for **R** classes.

```
library(cluster)
aa.ga <- agnes(animals, method = "average")
plot(aa.ga, sub = "", main = "", which.plots = 2, xlab = "")
```

[1] An object class resulting from hierarchical agglomerative cluster analyses produced by the function `cluster::agnes`.

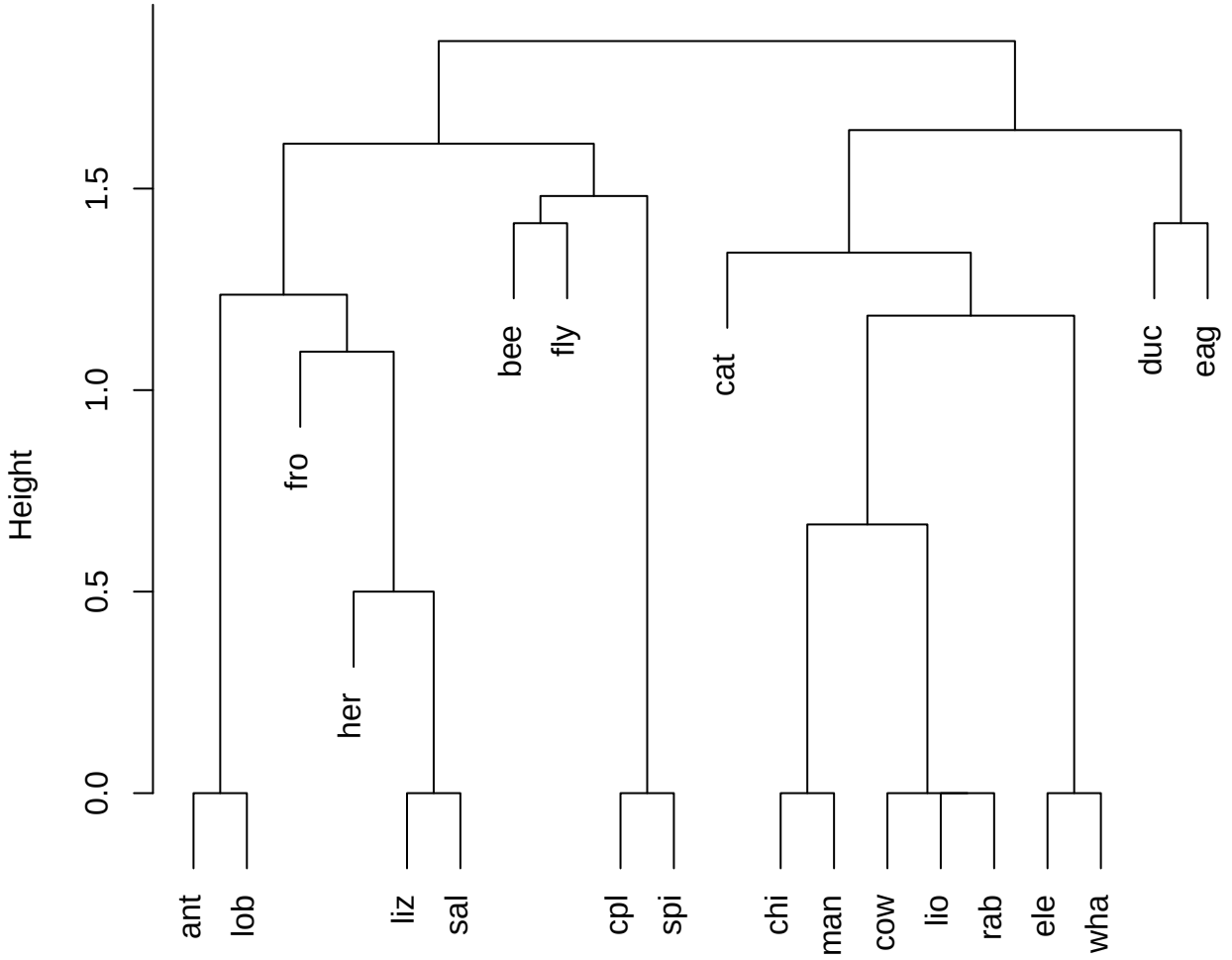


Figure 6.3: Dendrogram of an average linkage classification of animals based on six variables: warm vs. cold blooded, ability to fly, vertebrate or invertebrate, whether or not the animal is endangered, whether or not the animal lives in groups, and whether or not the animal has hair. The plotting function used, `plot.agnes()`, is called using the generic name `plot()`.

By default, the function `plot()` creates a projection at user defined Cartesian coordinates. Under this usage `plot()` has only two required arguments.

- `x` defines the x-coordinate values.
- `y` defines the y-coordinate values.

If coordinates for only one dimension, `x` are supplied, then `x` is plotted on the vertical axis against the series $1 : n$, where n is the number of points in `x`. A coordinate system can also be supplied to the argument `x` in the form of a formula, list, matrix, or dataframe.

Important optional arguments include the following:

- `pch` specifies the symbol type(s), i.e., the plotting character(s) to be used.
- `col` defines the color(s) to be used with the symbols.
- `cex` defines the size (character expansion) of the plot symbols and text.

- `xlab` and `ylab` allow the user to specify the x and y-axis labels.
- `type` allows the user to define the type of graph to be drawn. Possible types are `"p"` for scatterplot points (the default), `"l"` for a line plot, `"b"` for both, `"c"` for the line component of `"b"`, `"o"` for overplotted, `"h"` for 'histogram' like vertical lines (see middle plot in top row of Fig 6.1), `"s"` for stair steps, and `"n"` for no plotting.

Example 6.1.
We can see some symbol and color alternatives by calling them in `plot()` (Fig 6.4).

```
1 plot(1:20, 1:20, pch = 1:20, col = 1:20,
2      ylab ="Symbol number",
3      xlab = "Color number",
4      cex = 1.6, cex.lab = 1.1, cex.axis = 1.1)
```

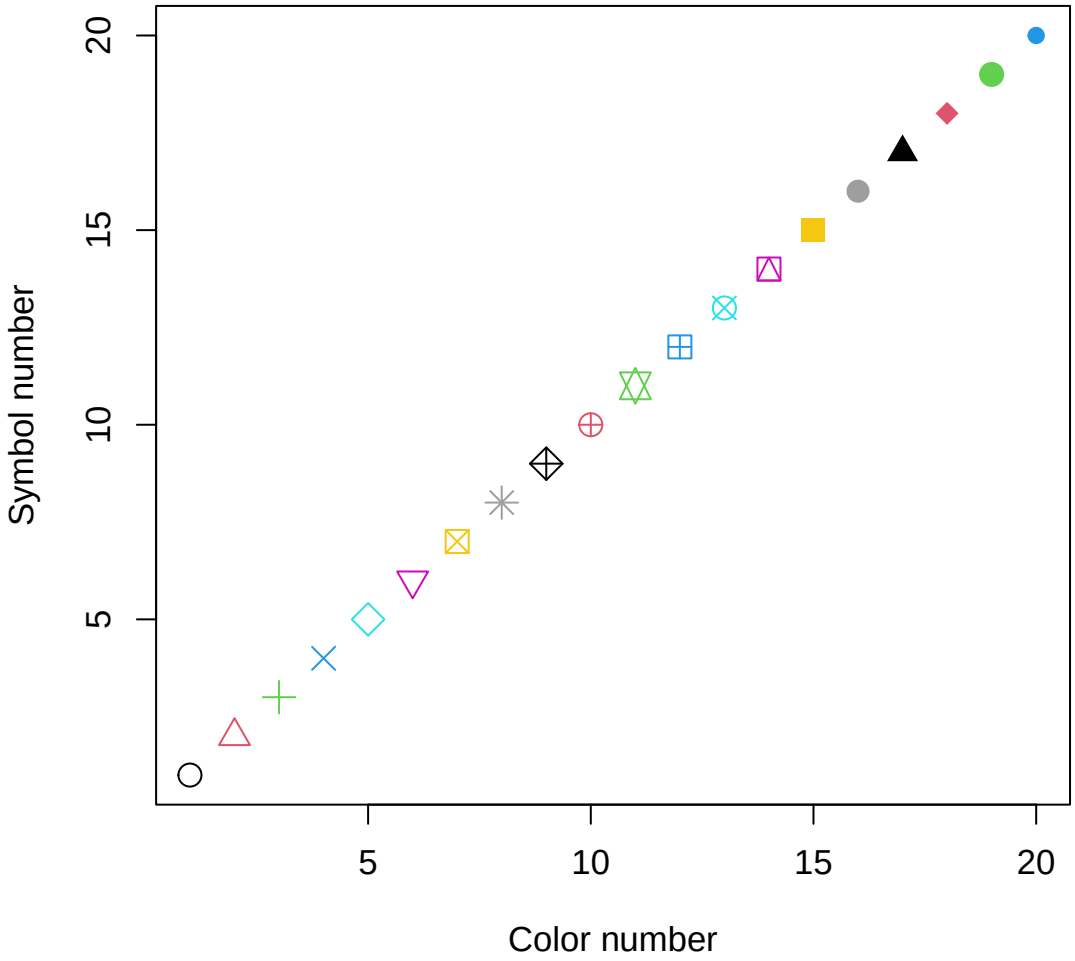


Figure 6.4: Some symbol and color plotting possibilities in `plot()`.

In Line one from the code above, the *x* and *y* coordinates are both sequences of numbers from 1 to 20 obtained from the command `1:20`. I varied symbol colors and plotting characters

(`col` and `pch`, respectively) using `1:20` as well. Thus, the combination `col = 1` and `pch = 1` results in a black open circle, whereas the combination `col = 20`, `pch = 20` results in a blue filled circle. Note that we need to enclose the axis names in quotations for `R` to recognize them as text. Symbol numbers `21–26` allow background color specification using the argument `bg`. Many other symbol types are also possible.

■

6.3 Graphical Devices

Graphics in **R** are created within *graphics devices*, encoded in the package *grDevices*. These vary with respect to storage modes, display modes, available typefaces, and other characteristics. In a basic **R** download, six graphics devices will be available:

- `windows()` is available for Windows releases of **R**. It provides on-screen rendering of graphics[2], and creates Windows metafile graphics files.
- `pdf()` renders graphics into .pdf files.
- `postscript()` renders graphics into PostScript, .ps, graphics files.
- `xfig()` renders graphics files using the Xfig graphics file format.
- `bitmap()` renders graphics into bitmap graphics files It requires the open source software ghostscript.
- `piktex()` Writes TeX/PicTeX graphics commands to a file and is of historical interest only.

A number of other graphics devices also exist, although they may return a warning if **R** was not compiled to use them upon installation.

- `cairo_pdf()`, `cairo_ps()` and `svg()` are PDF, PostScript SVG (Scalar Vector Graphics) devices based on the open source Cairo graphics.
- `bmp()`, `jpeg()`, `png()`, and `tiff()` render graphics as .bmp, .jpg, .png, and .tif bitmap files, respectively.
- `X11()` is the graphics device for the X11 windowing system, and is commonly used in Unix-alike operating systems, including MacOS.
- `quartz()` is only functional on MacOS and supports plotting to the screen (default) and to various graphics file formats. The device requires the open source software XQuartz for rendering some **R** graphical user interfaces (see Ch 11).

Multiple devices (currently up to 63) may exist simultaneously in an **R** work session, although there will only be one *active* device. To find the current (active) graphics device can type `dev.cur()`. I get:

[2] RStudio has its own native on-screen graphics device. A non RStudio graphics device can be opened (within RStudio) using `dev.new(RStudioGD = FALSE)`.

```
dev.cur()
```

```
pdf
  2
```

R tells me there are two devices open. The current device is a Windows device. The second device is the so-called "null device." The null device is always open but only serves as a placeholder. Any attempt to use it will open a new device in **R**. Occasionally, on purpose or by accident, all graphics devices (except the null device) may become turned off. A new active graphics device can be created at any time by typing:

```
dev.new()
```

One can close the current (active) device using:

```
dev.off()
```

The active device can be changed using the function `dev.set()`. For instance, if there were three or more accessible devices, and one wished to define device three as the active device, one could type:

```
dev.set(3)
```

It is possible to scroll through graphics devices using keyboard shortcuts. Specifically, let n be the current device number, then the combination Ctrl + Alt + F11 (Windows or Linux) or Cmd + Alt + F11 (Mac) shows device $n-1$, whereas Ctrl + Alt + F12 (Windows or Linux) or Cmd + Alt + F12 (Mac) shows device $n+1$.

6.4 `par()`

Parameters for a graphics device (which may contain several plots) can be accessed and modified using the function `par()`. Below are important arguments for `par()`. Some of these can also be specified as arguments in `plot()`, with different results.

- `bg` gives the background color for the graphical device. When used in `plot()` it gives the background color of plotting symbols.
- `bty` is the box-type to be drawn around the plots. If `bty` is one of `"o"` (the default), `"l"`, `"7"`, `"c"`, `"u"`, or `"]"` the resulting box resembles the corresponding upper case letter. The value `"n"` suppresses the box.
- `fg` gives the foreground color.
- `font` is an integer that specifies the font typeface. `1` corresponds to regular text (the default), `2` to bold face, `3` to italic and `4` to bold italic.
- `las` is the style of axis labels: `0` always parallel to the axis (default), `1` always horizontal, `2` always perpendicular to the axis, `3` always vertical.
- `mar` will have the form `c(bottom, left, top, right)` and gives the number of lines of margin to be specified on the four sides of the plot. The default is `c(5, 4, 4, 2) +`

`0.1`.

- `mfrow` will have the form `c(number of rows, number of columns)` and the number and position of plots in a graphical layout. Multiple graphs can also be placed into a graphical device with additional control over plot designation to multiple elements in a row and column configuration, using the function `layout()`.
- `oma` specifies the outer margins of a graphical device, given multiple plots, using a vector using a matrix of the form: `c(bottom, left, top, right)`.
- `usr` will have the form `c(x1, x2, y1, y2)` giving the extremes of the user coordinates of the plotting region.

When setting graphical parameters, it is good practice to revert back to the original parameter values. Assume that I want to background of the graphics device to be black. To set this I would type:

```
old.par <- par(no.readonly = TRUE) # save default, for resetting...
par(bg = "black") # change background parameter
```

To return to the default settings for background I would type:

```
par(old.par)
```

Defaults will also be reset by closing the current graphics device, or by opening a new device. For instance, using `dev.new()`.

Other fundamental properties of the default graphics device, such as device height, width and pointsize, can be adjusted using the `dev.new()` function. For instance, to create a graphical device 9 inches wide, and 4 inches high, I would type:

```
dev.new(width = 9, height = 4)
```

Example 6.2.
Fig 6.5 shows an example of applying background and foreground colors using the `bg` and `fg` arguments in `par()`, respectively. Note also the specification of a bold font using the `par()` argument `font = 3`, and expansion of all graphics parameters to slightly larger than their original size, using `cex = 1.1`.

```
1 old.par <- par(no.readonly = TRUE)
2 par(bg = "black", fg = "white", font = 3, cex = 1.1)
3 plot(1:10, 1:10, xlab = "x", ylab = "y",
4      col.lab = "white")
5 par(old.par)
```

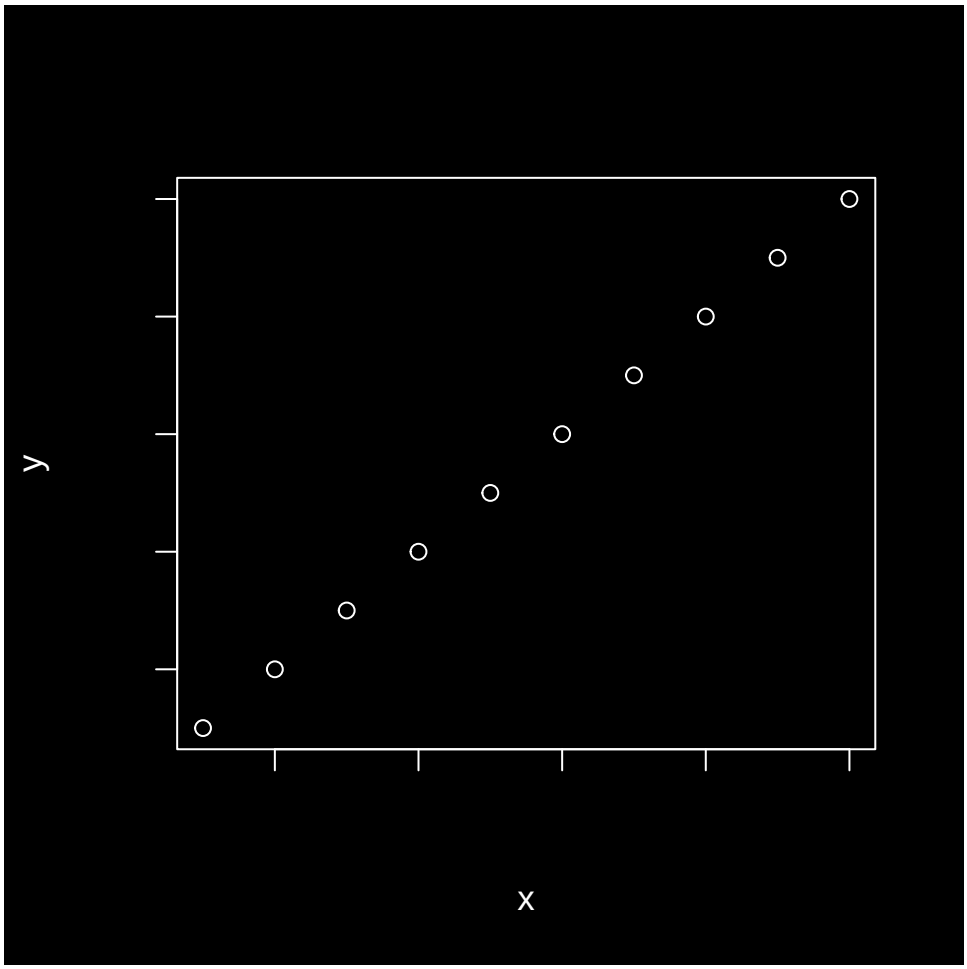


Figure 6.5: Use of `par()` to change background and foreground graphical parameters.

■

Example 6.3.
Fig 6.6 shows how one can place multiple graphs into a single graphical device using the `mfrow` argument in `par()`, and control figure margins using the `par()` argument `mar` (Line 2). It also shows some basic plot types resulting from the `type` argument in `plot()` (Lines 4-7).

```
1 old.par <- par(no.readonly = TRUE)
2 par(mfrow = c(2,2), cex = 1.1, mar = c(4,4,1,1))
3 x <- 1:10; y <- sort(rnorm(10))
4 plot(x, y)
5 plot(x, y, type = "l")
6 plot(x, y, type = "o")
7 plot(x, y, type = "h")
8 par(old.par)
```

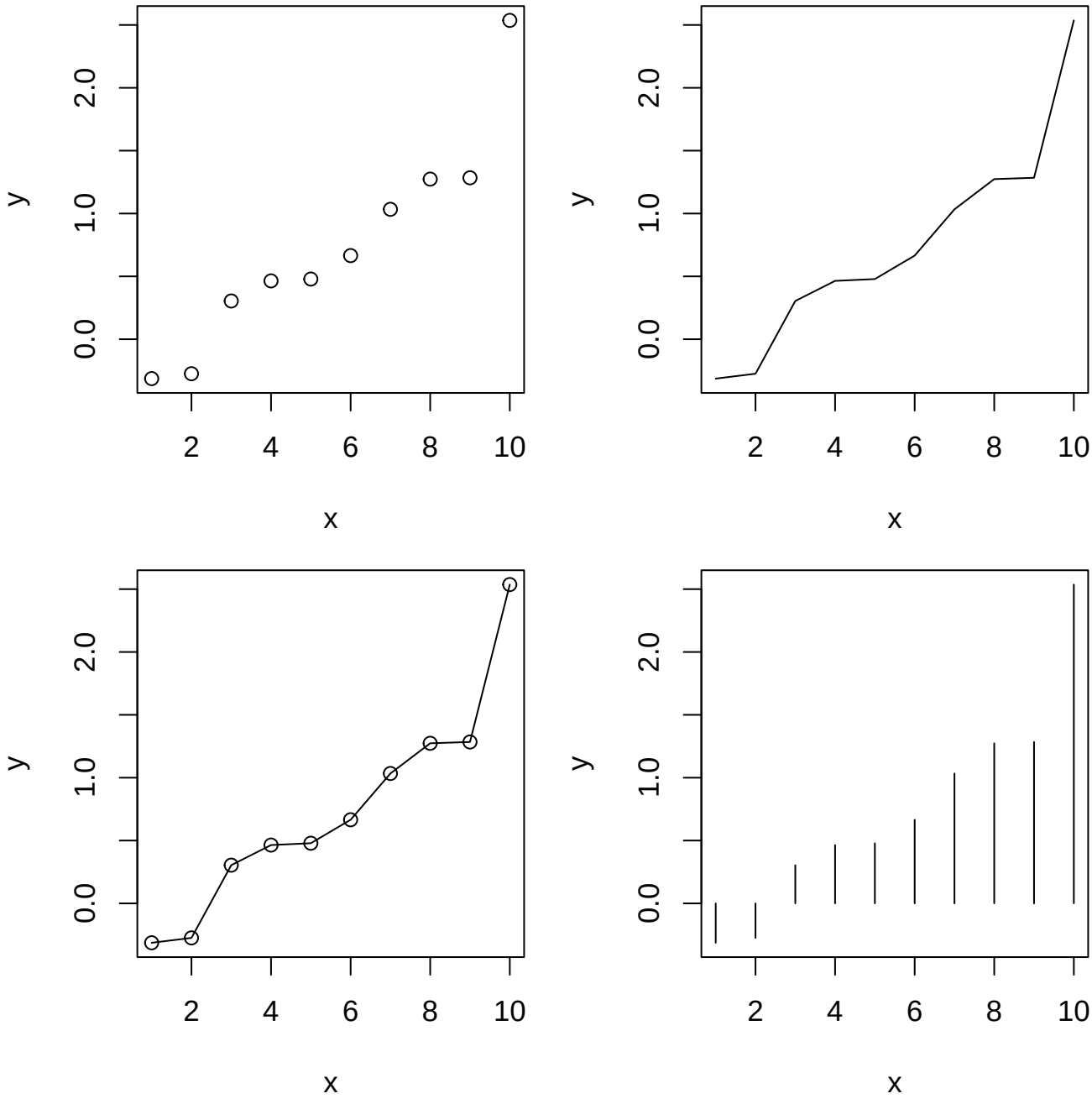


Figure 6.6: Use of `par()` to place multiple graphs into a single graphical device. The figure also demonstrates basic plot types, specified using the `plot()` argument `type`. Clockwise from the top-left these are: 1) a point plot (scatterplot), 2) a line plot, 3) a histogram-like (high density line) plot, and 4) a plot with a both points and lines.

■

6.5 Exporting Graphics

To export **R** graphics, one can generally copy snapshots to a clipboard using pull down menus on graphical device. These can then be pasted into programs (e.g., word processors) as bitmaps (a spatially mapped array of bits) or metafiles, a generic term for a file format that can store multiple types of (generally graphical) data.

To create the best possible graphs, however, one should save device output using graphical device functions.

For instance, to save a graphics device image as a pdf under the file name `example.pdf` in the working directory I would type:

```
pdf(file ="example.pdf")
```

I would then make the plot, for instance

```
plot(1:10)
```

The plot will not be shown because the `png()` graphical device is engaged. As a final step I close the device.

```
dev.off()
```

The graphics file will now be contained in the working directory. If the file argument is unspecified, `pdf()` will save a file called `Rplot.pdf`.

By default, the *bitmap graphics formats*: BMP, JPEG, PNG, and TIFF, have a width and height of 480 pixels, and a "large" point size (1/72 inch) in **R**. This results in a rather coarse 72 ppi (72 points per inch) image resolution. However, changing the `res` (resolution) argument in a graphical device function without changing the pointsize, or height and width arguments will generally result in unusable figures.

Because $500 \approx 72 \cdot 6$, one can generate a TIFF with greater than 400 ppi TIFF called `fig1.tiff` by typing:

```
tiff("fig1.tiff", res = 72 * 6, height = 480 * 6, width = 480 * 6)
plot(1:10)
dev.off()
```

With respect to graphical formats, documentation in the *grDevices* package states:

> "The PNG format is lossless[3] and is best for line diagrams and blocks of color. The JPEG format is lossy[4], but may be useful for image plots, for example. The BMP format is standard on Windows, and supported by most viewers elsewhere. TIFF is a meta-format: the default format written by the default format `tiff(compression`

[3] Lossless entails data compression without loss of information.
[4] Lossy refers to data compression in which unnecessary information is discarded.

= none) is lossless and stores RGB values uncompressed. Such files are widely accepted, which is their main virtue over PNG."

The svg(), cairo_pdf() and cairo_ps() graphical devices apply cairographics and will recognize a large number of symbols and fonts not available for document and image generation in the default setting of the Windows PostScript and PDF devices.

6.6 text(), points(), and lines()

The functions text(), points() and lines() can be used to overlay text, points and lines in a plot, respectively. As with plot() the first two arguments of these functions are the *x* and *y* coordinates for the plotted entities. Other arguments concern characteristics of the plotted items. For instance, to plot the text "example" with in an existing plot, at plot coordinates x = 0, y = 0, with a large character expansion, I could type:

```
plot(-1:1, -1:1, type = "n", axes = F, xlab = "", ylab = "") # empty plot
text(x = 0, y = 0, "example", cex = 9)
```

The result is shown in Fig 6.7.

Figure 6.7: Empty plot (even axes are suppressed) with text overlain.

The function paste() can be used to concatenate elements from text strings in plots or output. For instance, try:

```
a <- c("a", "b", "c")
b <- c("d", "e", "f")
c <- paste(a, b)
c
```

```
[1] "a d" "b e" "c f"
```

Which can be placed in a plot (Fig 6.8) using text().

```
plot(-1:1, -1:1, type = "n", axes = F, xlab = "", ylab = "")
text(x = 0, y = 0, paste(c, collapse = ' '), cex = 2)
```


Figure 6.8: An empty plot with text overlain. Note the use paste(c, collapse = ' ') to collapse the string vector c into a single entity.

To plot a dashed line between the points $(0, 0)$ and $(3, 2)$, I would type:

```
lines(x = c(0, 3), y = c(0, 2), lty = 2)
```

or

```
points(x = c(0, 3), y = c(0, 2), lty = 2, type = "l")
```

The result is shown in Fig 6.9.

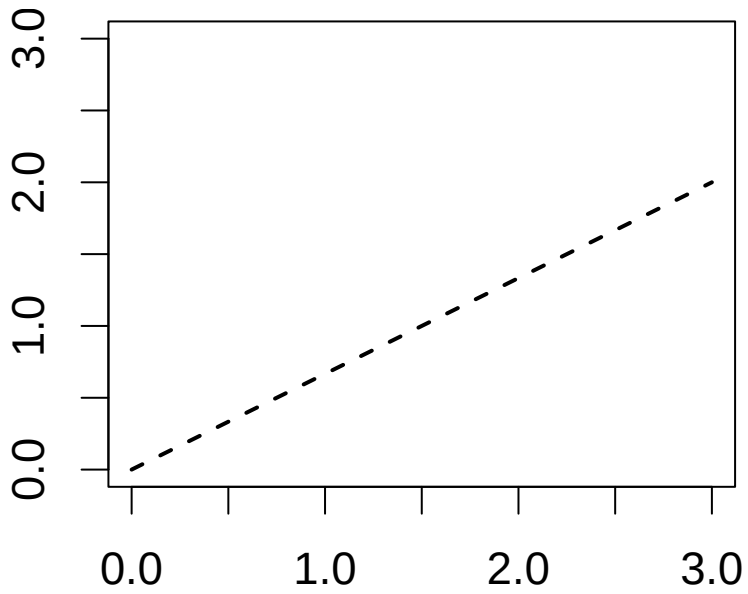


Figure 6.9: Plot with dashed line overlain.

To place a large, blue, triangle with red outline at the point (1, 1), of an existing plot I would type:

```
points(x = 0, y = 1, pch = 24, col = 2, bg = 4, cex = 8)
```

The resulting plot is shown in Fig 6.10.

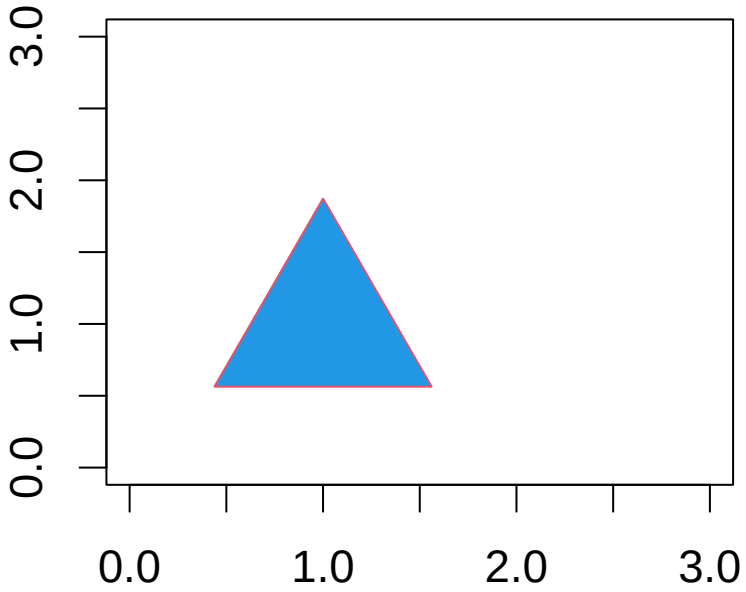


Figure 6.10: Plot with point overlain.

6.6.1 Plotting Mathematical Text

R has useful functions for the plotting of mathematical expressions. These include the Greek letters, mathematical operators, italicization, and sub- and super-scripts. mathematical text is generally called as an expression in the text argument in the functions `text()` or `mtext()`. For example, the formula for the sample variance is overlain in Fig 6.11.

```
plot(-1:1, -1:1, type = "n", axes = F, xlab = "", ylab = "")
varexp <- expression(over(sum(paste("(",italic(x[i] - bar(x)),")"^2),
                              italic(i)==1, italic(n)),(italic(n) - 1)))

text(x = 0, y = 0, varexp, cex = 3)
```

$$\frac{\sum_{i=1}^{n}(x_i-\bar{x})^2}{(n-1)}$$

Figure 6.11: Empty plot with formula for the sample variance overlain. Type `?plotmath` for more information.

6.6.2 `mtext()`

To place text in the margin of a plot we can use the function `mtext()`. As its first argument the `mtext()` function requires a character string to be written into the plot. The 2nd argument, `side` defines the plot margin to be written on: `1` = bottom, `2` =left, `3` = top, `4` = right. For instance, to place the text `"Axis 2"` on the right hand axis of an existing plot, I would type:

```
mtext("Axis 2", 4)
```

6.7 Geometric Shapes

Geometric shapes can be drawn using a number of functions including `rect()` (which draws rectangles) and `polygon()` (which draws other polygons) based on user-supplied vertices. For instance, to place a purple rectangle with vertices at $(1, 1)$, $(1, 2)$, $(2, 2)$, and $(2, 1)$, in an existing plot, I would type:

```
rect(xleft = 0, ybottom = 1, xright = 2, ytop = 2, col = 6)
```

See Fig 6.12.

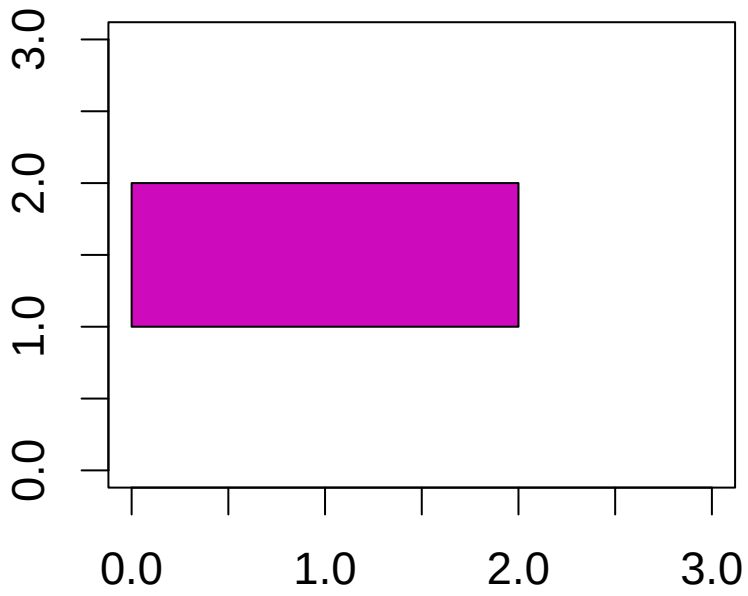


Figure 6.12: Plot with rectangle overlain.

6.8 `axis()`

The function `axis()` can be used to create new axes on a plot or to customize axis characteristics. Its first argument (`side`) specifies the side of the plot that the new axis will occupy `1` = bottom, 2 =left, 3 = top, 4 = right.

For instance, to create a right hand axis I would type:

```
axis(4)
```

Other important `axis()` arguments include a vector of axis labels (argument `labels`), and the locations of labels (argument `at`).

Example 6.4.
Here I create customized axes with rotated, x-axis labels, using `axis()` and `text()` (Fig 6.13).

```
1 plot(1:3, type = "n", axes = F, xlab = "", ylab = "")
2 axis(side = 2, at = 1:3, col = "red")
3 axis(side = 1, at = 1:3, labels = FALSE, col = "blue")
4 text(1:3, rep(.65, 3), c("Label 1", "Label 2", "Label 3"),
5      srt = 50, xpd = TRUE)
```

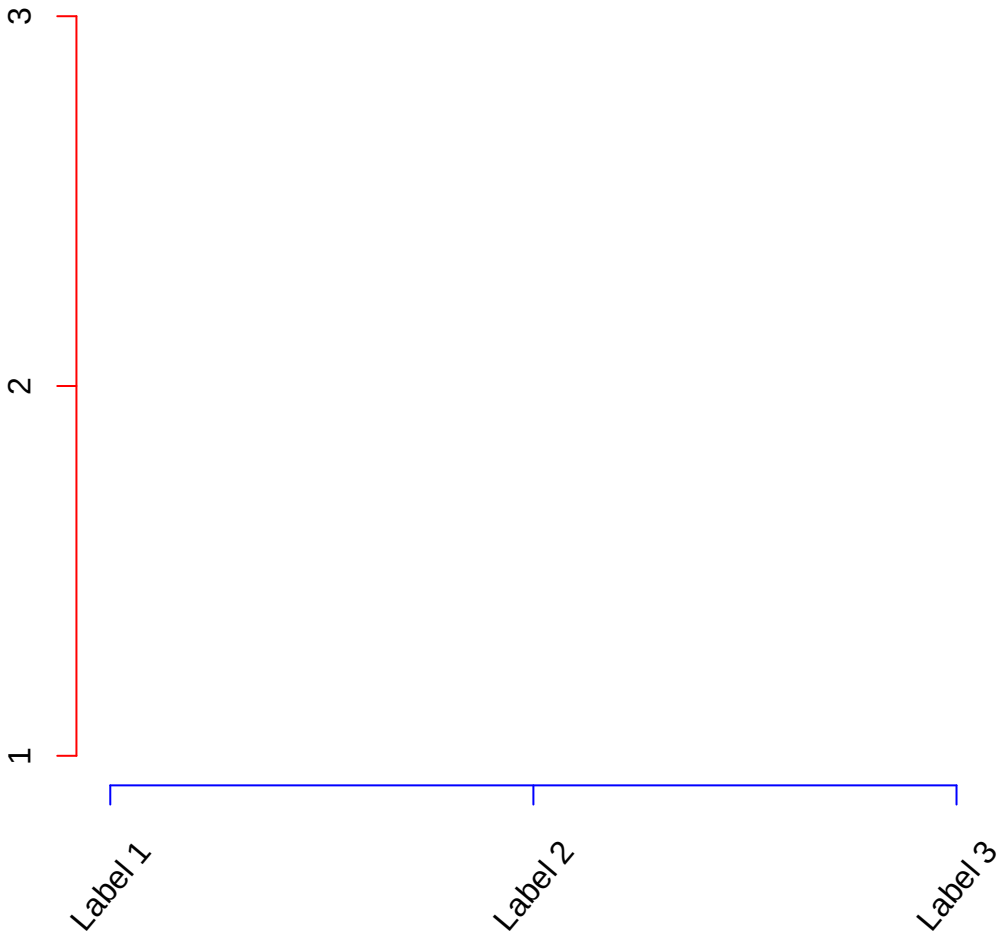


Figure 6.13: Modifying axes with `axis()`.

The argument `srt = 50` (Line 5) rotates the text 50 degrees (`srt` cannot be specified in

`mtext()` or `axis()`, hence the use of `text()` here). The specification `xpd = TRUE` in `text()` (Line 5) allows text printing to extend to the plot axis margins.

■

6.9 Font Typefaces

Font *typefaces* can be changed using a number of graphical functions, including `par()`, via the argument `family`. The general typeface families: `"serif"`, `"mono"`, and `"sans"`, and the Hershey family of fonts (type `?Hershey` for more information) are transferable across all graphics devices employed in `R`. To change the font in a graphical device from the default sans serif (similar to Arial) to serif (similar to Times New Roman) one could type:

```
par(family = "serif")
```

To use a Courier-type monospace font one would use.

```
par(family = "mono")
```

Many other typeface families are possible, although they may not be transportable to all graphical devices and graphical storage formats.

Example 6.5.
In the code below I bring in a large number of conventional font families using a function from the Foundational and Applied Statistics for R website. These typefaces (and many others) will typically be available on Windows platform machines, although not all will be supported by non-Windows graphics devices. The result can be seen in Figure 6.14 which displays text from ninety-nine Windows typefaces.

```
1  source(url("https://amalgamofr.org/win_fonts.R"))
2  png("fonts.png", res = 72 * 6, height = 480 * 6, width = 480 * 6)
3    x <- rep(c(2.8, 6.4, 9.6), each = 33)
4    y <- rep(seq(10, 0.25, by = -.2965), 3)
5    font.type <- paste(rep("f", length(fonts)), 1:length(fonts), sep = "")
6    par(mar = c(0.1,0.1,0.1,0.1), cex = 1.05)
7    plot(0:10, type = "n", xaxt= "n", yaxt = "n", xlab = "", ylab = "",
8         bty = "n")
9
10   for(i in 1:length(fonts)){
11     text(x[i],y[i], labels=fonts[i] , family = font.type[i])
12   }
13 dev.off()
```

Figure 6.14: Examples of font families that can be used in **R** graphics.

Note that on Line 2 in the code above, I use the function `png()` to generate a high resolution .png graphical file. Thus, running the entirety of the preceding code chunk will create the image file `fonts.tiff` in your working directory. To save myself from typing an inordinate amount of code, I use a for loop (see Ch 8) to place the fonts one at a time in the graphics device (Lines 9-11). Output from closing the graphical device is shown on Line 14-15.

Importantly, the typefaces imported on Line 1 in the chunk above will now be available for graphics functions using the Windows graphical device. To see the first six available Windows fonts one can type:

```
head(windowsFonts())
```

```
$serif
[1] "TT Times New Roman"

$sans
```

```
[1] "TT Arial"

$mono
[1] "TT Courier New"

$f1
[1] "Agency FB"

$f2
[1] "Albany AMT"

$f3
[1] "ALGERIAN"
```

Similarly, one can see the available fonts for PostScript and PDF graphics devices using:

```
head(names(pdfFonts()))
```

```
[1] "serif"      "sans"       "mono"       "AvantGarde" "Bookman"
[6] "Courier"
```

■

6.10 Colors

An enormous number of color choices for **R** graphics are possible and these can be specified in at least six different ways.

- First, we can specify colors with integers as I did in Figure 6.4. Additional varieties can be obtained by drawing color elements from the function `colors()` using `colors()[number]` (Fig 6.15).

```
e <- expand.grid(1:20, 1:32)
plot(e[,1], e[,2], bg = colors()[1:640], pch = 22, cex = 2.5, xaxt = "n",
     yaxt = "n", xlab = "", ylab = "")
text(e[,1], e[,2], 1:640, cex = .4)
```

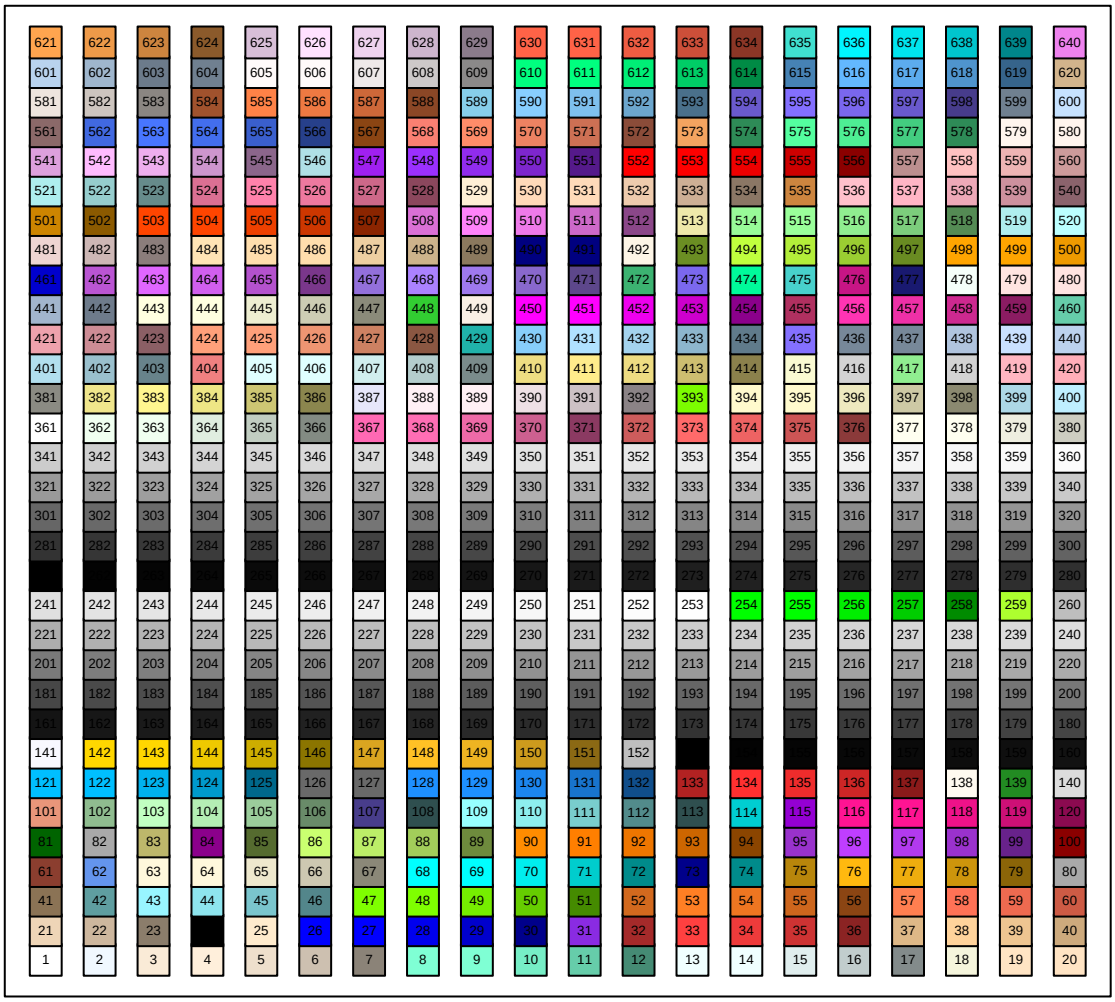


Figure 6.15: Color choices from `colors()`

The function `expand.grid()` creates a dataframe from all combinations of user-supplied supplied vectors. Note that these combinations are used as coordinates in `plot()`.

- Second, we can specify colors using actual color names, e.g., `"white"`, `"red"`, `"yellow"`. For a visual display of essentially all the available named colors in **R** type: `example(colors)`.
- Third, we can define colors by requesting red green and blue (RGB) color intensities, along with transparency, using the function `rgb()` (Fig 6.16). Usable light intensities can be made to vary individually from 0 to 255 (i.e., within an 8 bit format). Thus, there are $255^4 = 4,228,250,625$ possible `rgb()` color combinations. By default, `red` `green`, `blue`, and `alpha` (transparency) arguments in `rgb()` are defined to be in (0, 1).

```
plot(1:10, cex = 15, pch = 19, xlab = "", ylab = "",
     col = rgb(red = rep(0.2,10), green = rep(0.5,10),
               blue = rep(0.8,10),
```

```
4                 alpha = seq(0.05,1, length = 10)), axes = F)
5 box()
```

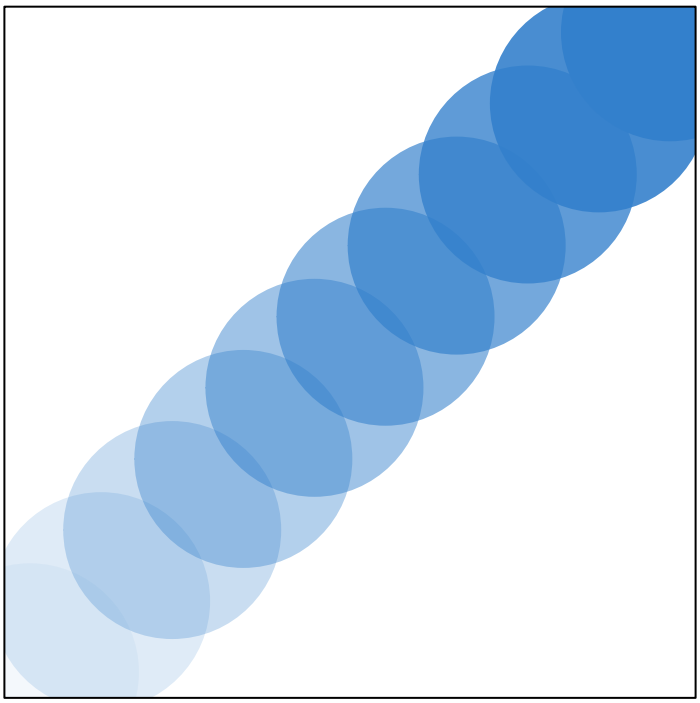


Figure 6.16: Demonstration of `rgb()`, emphasizing changes in transparency.

Note the use of `box()` on Line 5, which places a box around the plot.

- Fourth, similar to `rgb()`, we can specify colors using the function `hcl()` which controls hues, chroma, and luminescence and transparency (see Fig 6.17).
- Fifth, we can define colors using *hexadecimal codes*[5], e.g., blue = `"#0000FF"`.
- Sixth, we can specify colors using *palettes*. Figure 6.17 shows six pie plots. Each pie plot uses a different pre-defined color palette. Each pie slice from each pie represents a distinct segment of a distinct palette.

```
1 layout(matrix(seq(1,6),3,2))
2 par(mar=c(1,1,1,1))
3 pie(rep(1,12), col = rainbow(12), main = "Rainbow colors")
4 pie(rep(1,12), col = heat.colors(12), main = "Heat colors")
5 pie(rep(1,12), col = topo.colors(12), main = "Topographic colors")
6 pie(rep(1,12), col = gray(seq(0,1,1/12)), main = "Gray colors")
7 pie(rep(1,12), col = hcl(h=seq(180,0, length=12)),
8     main = "Cols from hcl hue")
```

[5]A data coding system that uses 16 symbols: the numbers 1-9, and the letters A-F. Hexadecimals are primarily used to provide a more intuitive representation of binary-coded values (see Ch 12).

```
9  pie(rep(1,12), col = hcl(h=seq(360,180,length=12)),
10     main = "Cols from chroma")
```

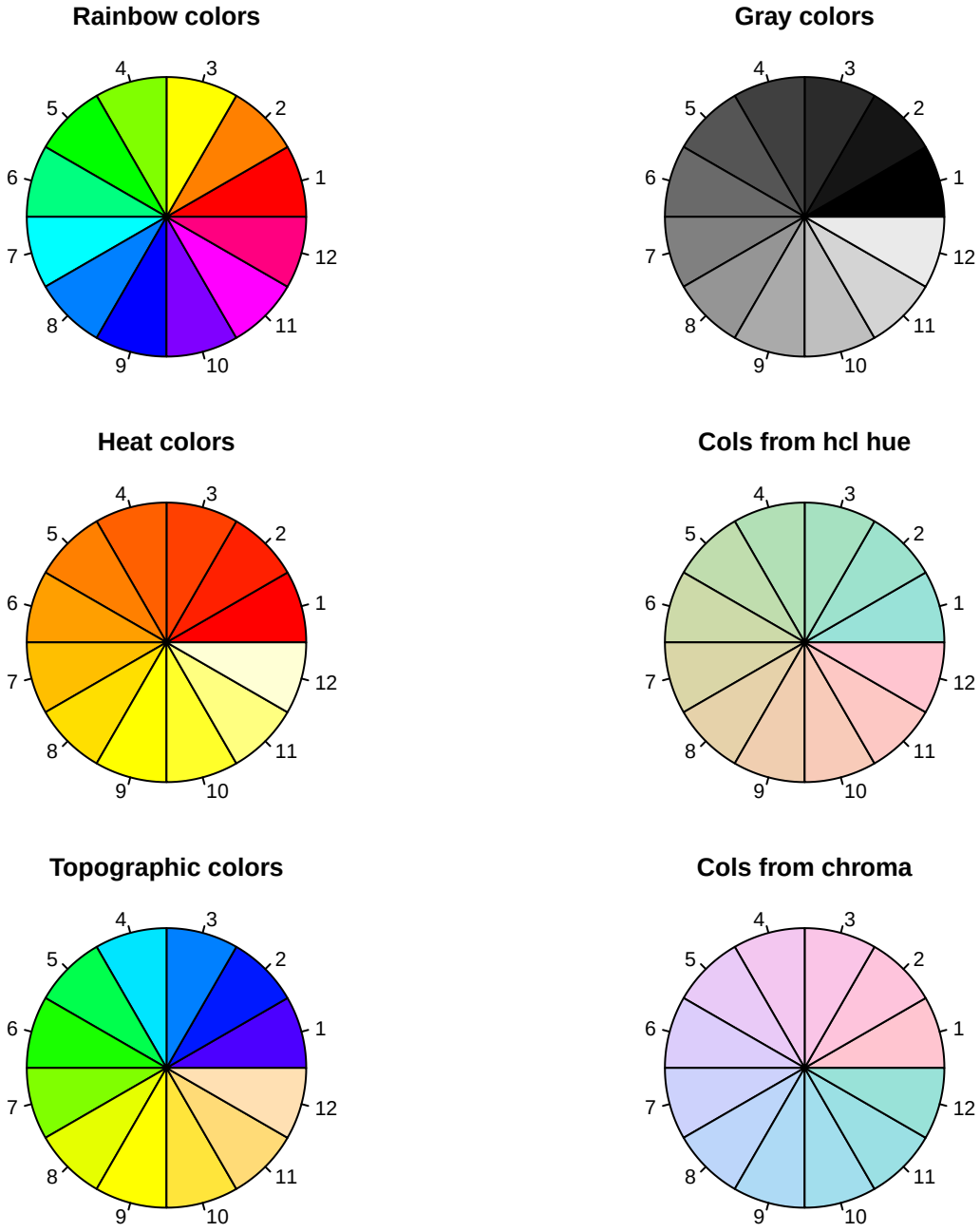


Figure 6.17: Examples of color palettes in **R**. Numbers do not correspond to actual color type designations.

Note that the functions `rainbow()`, `heat.colors()`, `topo.colors()` (Lines 3-6) only require an integer specification requesting the number of colors within a particular palette. For instance

```
rainbow(5)
```

```
[1] "#FF0000" "#CCFF00" "#00FF66" "#0066FF" "#CC00FF"
```

Note that the colors in `rainbow()` are given in a hexidecimal format.

The function `palette()` can be used to check and define a number of useful palettes. Colors in the *current* palette can be obtained by typing:

```
palette()
```

```
[1] "black"   "#DF536B" "#61D04F" "#2297E6" "#28E2E5" "#CD0BBC" "#F5C710"
[8] "gray62"
```

A list of predefined palettes in `palette()` can be obtained by typing:

```
palette.pals()
```

```
 [1] "R3"              "R4"              "ggplot2"
 [4] "Okabe-Ito"       "Accent"          "Dark 2"
 [7] "Paired"          "Pastel 1"        "Pastel 2"
[10] "Set 1"           "Set 2"           "Set 3"
[13] "Tableau 10"      "Classic Tableau" "Polychrome 36"
[16] "Alphabet"
```

To define the current palette to be the one used by the *ggplot2* package (Ch 7), I could type: `palette("ggplot2")`.

A large number of useful pre-defined palettes (including color-blind-safe palettes) can be obtained using the package *RColorBrewer* (Fig 6.18).

```
library(RColorBrewer)
display.brewer.all(n = 7, colorblindFriendly = TRUE)
```

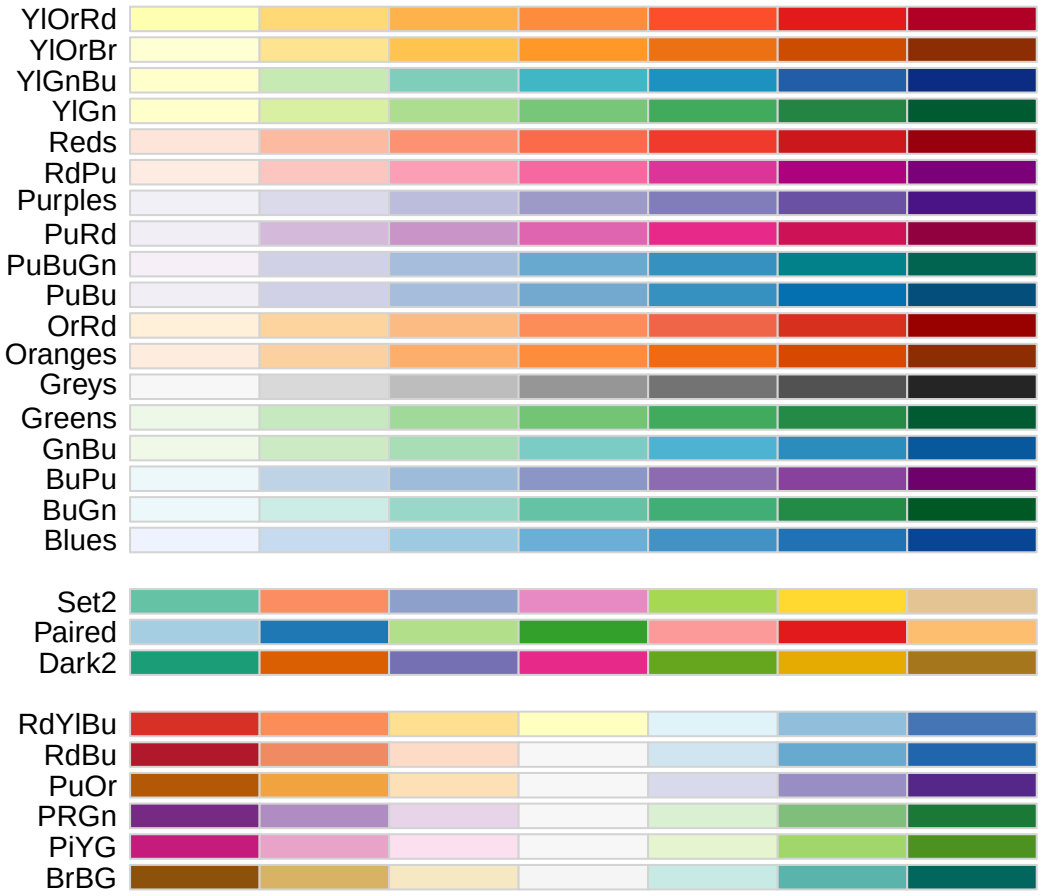


Figure 6.18: *RColorBrewer* color-blind-safe, seven category palettes. Top palettes are so-called 'sequential' palettes, middle palettes are 'qualitative', and bottom palettes are 'divergent'.

Here are the hexadecimal names for the "Set2" palette chunks in Figure 6.18.

```
brewer.pal(7, "Set2")
```

```
[1] "#66C2A5" "#FC8D62" "#8DA0CB" "#E78AC3" "#A6D854" "#FFD92F" "#E5C494"
```

Customized palettes can be generated using the `colorRamp()` function which returns functions that "interpolate a set of given colors to create new color palettes." Important `colorRampPaltette()` arguments include: two required arguments.

- `colors` defines colors to interpolate.
- `bias` a positive number that controls distinctions among interpolated colors. Larger values indicated greater differences.
- `space` one of `"RGB"` or `"Lab"`, indicating whether RGB or CIELAB[6] color spaces are to be used in interpolations.

[6]The CIELAB color space is defined by three values: `L*` for perceptual lightness and `a*` and `b*` for the four unique colors of human vision: red, green, blue and yellow. (Schanda, 2007). The CIELAB space is intended to be *perceptually uniform*. CIELAB and several other colors spaces are included in the encompassing CIECAM02 color space (Wikipedia, 2024d).

Here I generate and plot a 15 color palette interpolated from the colors red and blue (Fig 6.19).

```
1 crp <- colorRampPalette(colors = c("red", "blue"))(15)
2 plot(1:15, pch = 19, cex = 5, col = crp, axes = F, xlab = "", ylab = "")
3 box()
```

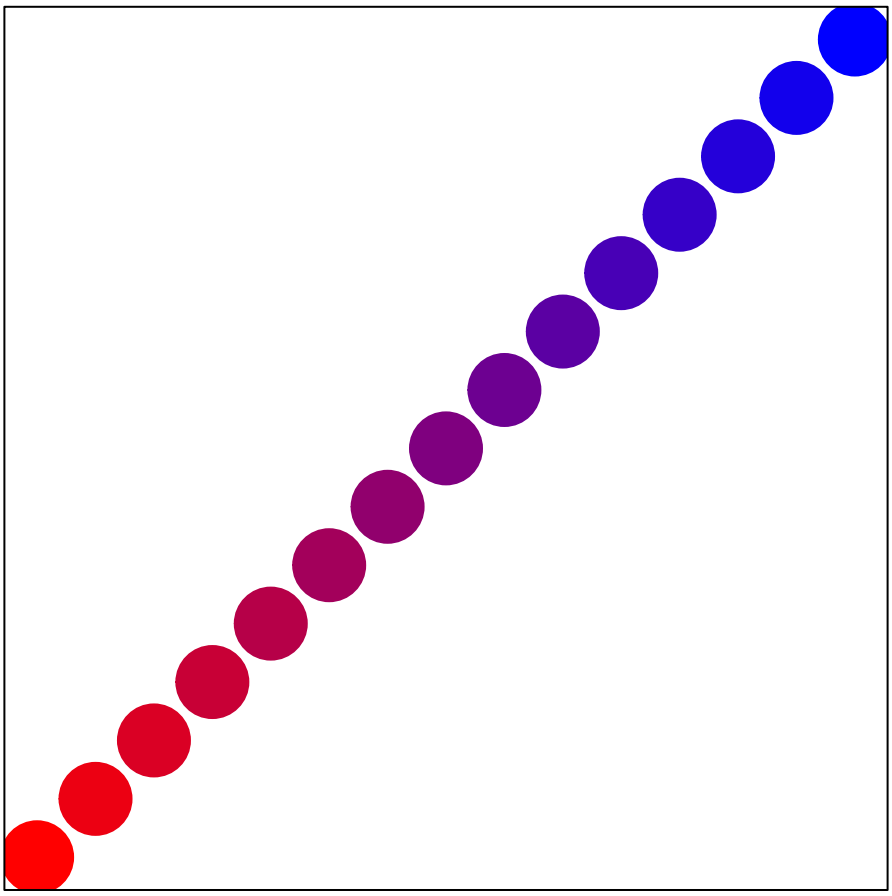


Figure 6.19: Color palette generated by the function `colorRampPalette()`.

The function `box()` (Line 3) places a box around the figure whose axes and axis labels I have intentionally omitted. There are a number of packages for the generation of customized palettes. My current favorite is *colorspace* and its interactive function `hclwizard()`, which generates the *shiny* GUI (Ch 11) shown in Fig 6.20.

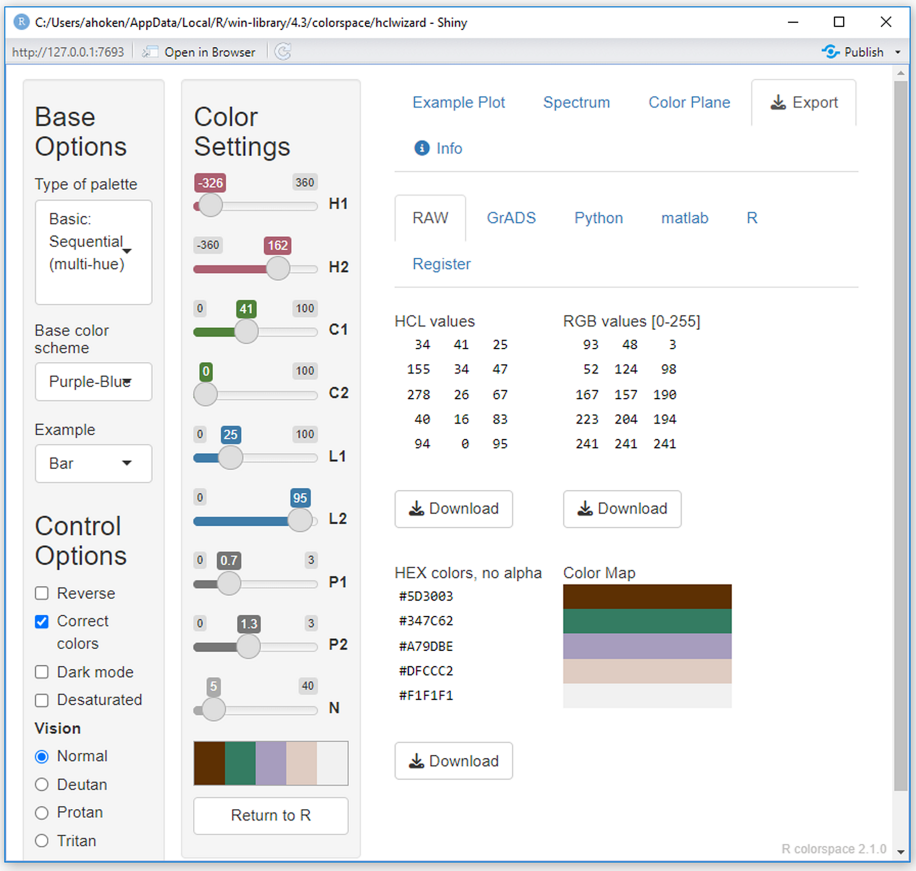


Figure 6.20: A GUI for constructed customized palettes generating by the function `colorspace::hclwizard()`.

6.11 Scatterplots

Scatterplots project points at the intersection of paired observations describing two quantitative variables. Thus, scatterplots are often presented in conjunction with simple regression analyses (Aho, 2014).

Example 6.6.
As an example of scatterplot creation we will use the `Loblolly` dataset in the package *datasets*. Figure 6.21 allows visualization of the relationship of loblolly pine tree age and tree height (also see Example 5.12).

```
with(Loblolly, plot(age, height))
```

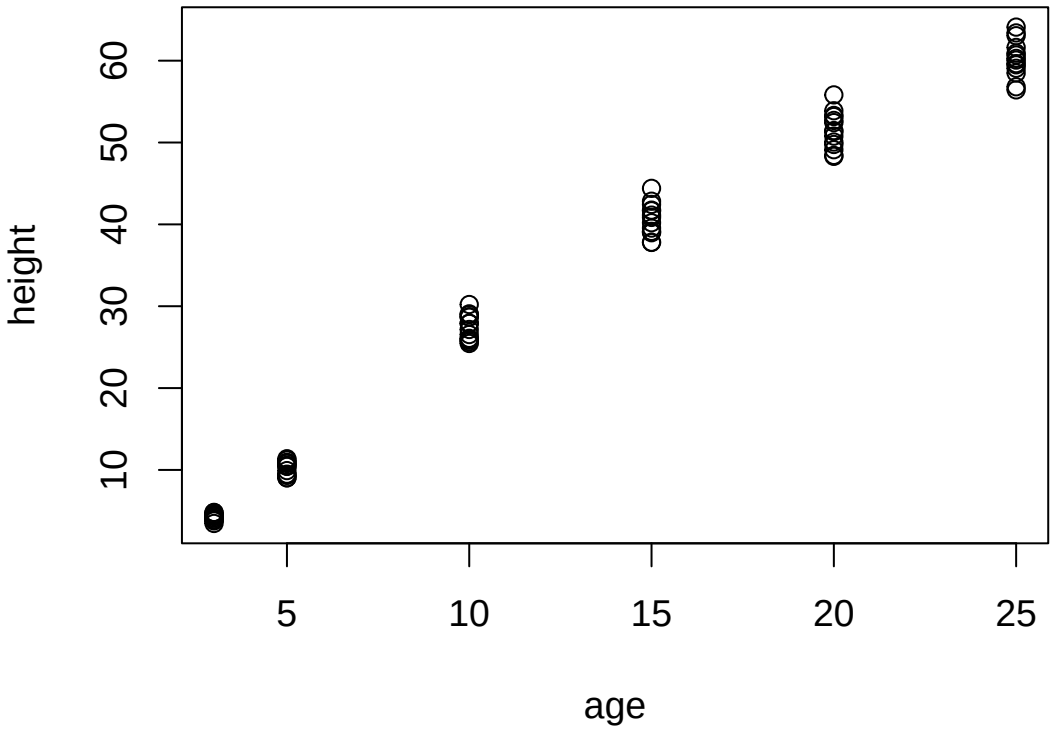


Figure 6.21: Scatterplot of `height` and `age` from the `Loblolly` pine tree dataset.

Now let's fit a simple linear regression for loblolly pine height as a function of age. A regression line will show the best possible linear fit for a response variable as a function of an quantitative explanatory variable (Aho, 2014). The **R** function for a linear model is `lm()`. It encompasses and allows a huge number of statistical procedures, including regression (see Chs. 9-11 in (Aho, 2014)). We have:

```
ha.lm <- lm(height ~ age, data = Loblolly)
```

Note that in the first argument of `lm()` we define `height` to be a function of `age` using the tilde operator. Objects of class `lm` have their own`summary` function. This can be called by simply typing:

```
summary(ha.lm)
```

```
Call:
lm(formula = height ~ age, data = Loblolly)

Residuals:
   Min     1Q Median     3Q    Max
-7.021 -2.167 -0.439  2.054  6.855

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
```

```
(Intercept)  -1.3124     0.6218   -2.11    0.038 *
age           2.5905     0.0409   63.27   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.95 on 82 degrees of freedom
Multiple R-squared:  0.98,  Adjusted R-squared:  0.98
F-statistic: 4e+03 on 1 and 82 DF,  p-value: <2e-16
```

The output shows us the Y-intercept, -1.31240, and slope, 2.59052, of the fitted regression line, and results from null hypothesis tests, along with a lot of other information.

The `abline()` function allows the plotting of a line over an existing plot. The first two arguments for `abline()` are the Y-intercept and slope (Fig 6.22).

```
with(Loblolly, plot(age,height, pch=2, col=3))
abline(-1.312396, 2.590523)
```

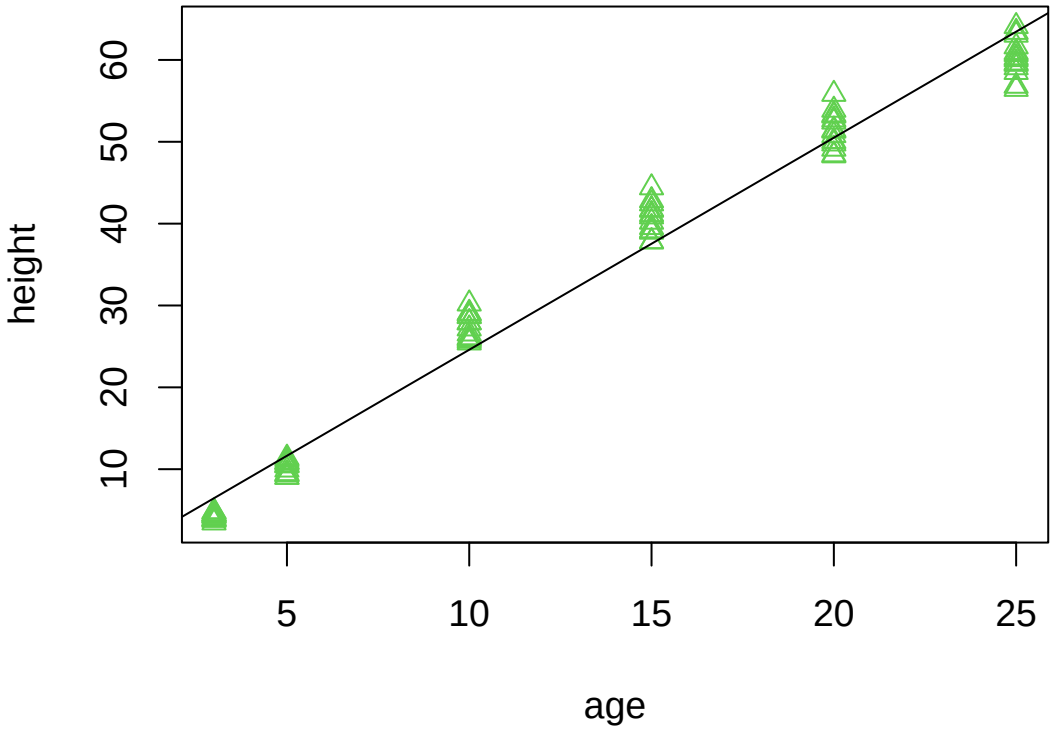


Figure 6.22: Scatterplot with fit overlain.

Note that we could have gotten the same result using `abline(ha.lm)`.

Finally, we can overlay a 95% confidence interval for the true regression fitted value (see Aho (2014), Ch 9) using the function `predict.lm()` (Fig 6.23)

```
1  ci <- predict(ha.lm, interval = "confidence")
2
3  o <- order(Loblolly$age)
4  ageo <- Loblolly$age[o]
5  cio <- ci[o,]
6
7  with(Loblolly, plot(age, height, pch=19, col=1))
8  abline(-1.312396, 2.590523)
9  points(ageo, cio[,2], type = "l", col = "gray") # lower CI bound
10 points(ageo, cio[,3], type = "l", col = "gray") # upper CI bound
```

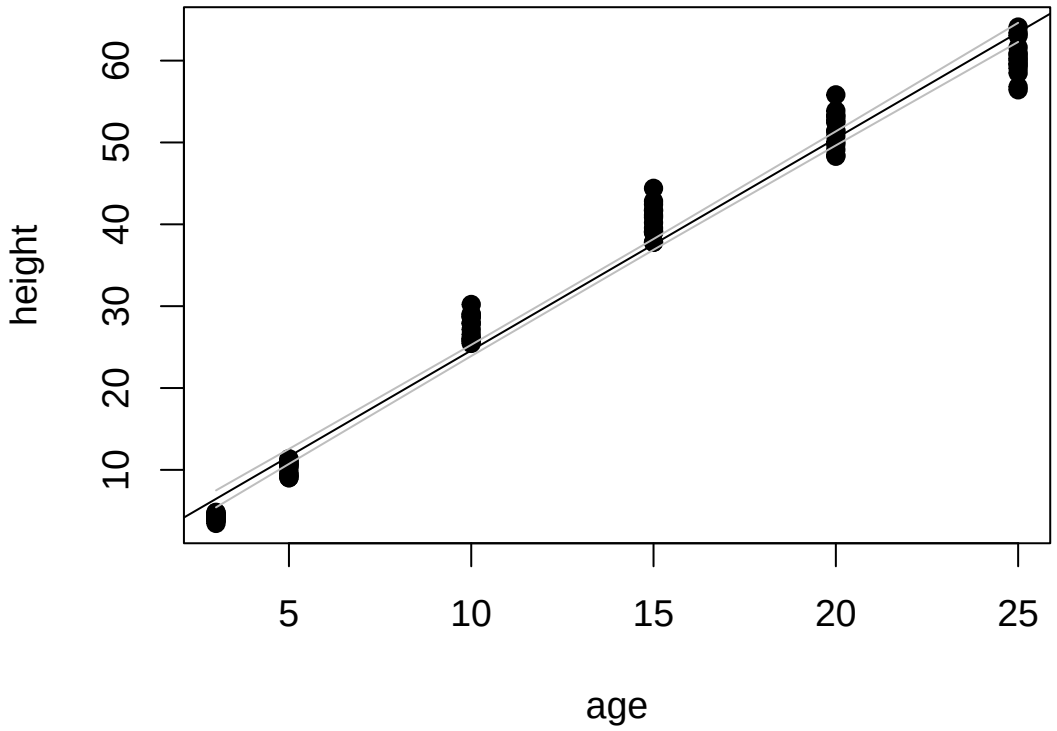


Figure 6.23: Scatterplot with confidence interval for the true mean of y given x, overlain.

The object `ci`, created on Line 1, is a dataframe containing fits, and corresponding lower and upper confidence interval bounds. The ordering of x-axis values is established on Line 3 to allow creation of lines that look like functions of x. This ordering is applied to Cartesian coordinates on Lines 4-5.

■

6.12 Transformations

Importantly, `plot()` allows straightforward application of log transformations to axes. For instance, to apply a $\log_e$ transformation to the x-axis or y-axis I could use `log = "x"` or `log`

= "y", respectively (Fig 6.24).

```
with(Loblolly, plot(age, height, log = "y"))
```

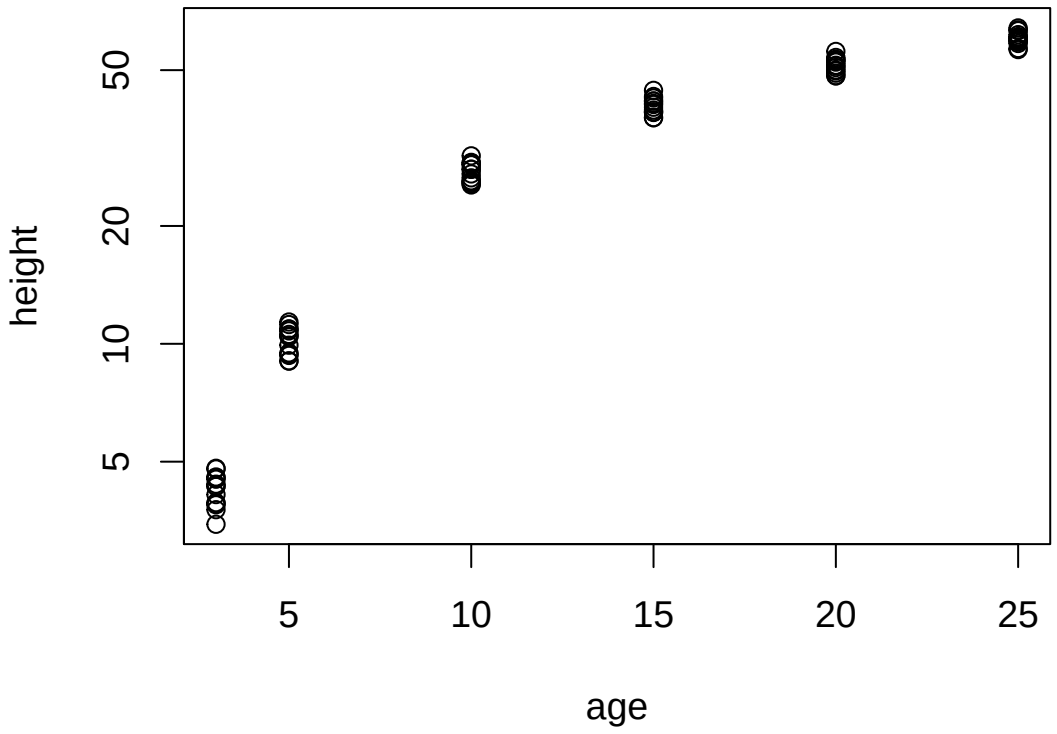


Figure 6.24: Graphical $\log_e$ transformation of the `height` axis from a scatterplot of the `Loblolly` pine tree dataset.

Example 6.7.
In this Example I will incorporate a number of the functions discussed so far, including `par()` with `usr()`, mathematical formulae with `text()`, `points()`, shapes rending with `rect()` which allows modification of plot backgrounds, `colors()` `axis()`, and `mtext()`.

The dataframe `C.isotope` in package *asbio* describes variations in δ^{14}C over time in La Jolla California. The term δ^{14}C describes the ratio of carbon 14 to carbon 12 (^{14}C is unstable, while ^{12}C is a stable isotope of carbon) compared to a standard ratio. We will create a figure with four subplots, with the following characteristics:

- It will have dimensions 8" x 7".
- The outer margins (in number of lines) will be bottom = 0.1, left = 0.1, top = 0, right = 0.
- The inner margins (for each subplot) will be bottom = 4, left = 4.4, top = 2, right = 2. The plot margins will be light gray. We can specify gray gradations with the function.
- The first plot will show δ^{14}C as a function of date. The plotting area will be dark gray, i.e., `colors()[181]`. Points will be white circles with a black border.
- The second plot will be a line plot of atmospheric carbon as a function of date. It will have a light green plotting area: `colors()[363]`.

- The third plot will be a scatterplot of $\delta^{14}C$ as a function of atmospheric CO_2. Points will be yellow circles with a black border. The plotting area will be light red: `colors()[580]`.
- The fourth plot will show the sample variance for atmospheric carbon in the time series. It will have a custom (albeit meaningless) axis, created with `axis()`, with the labels: `a`, `b`, `c`, and `d`. It will also have a horizontal axis label inserted with `mtext()`.

The result is shown in Fig 6.25.

```
library(asbio)
data(C.isotope)
dev.new(height = 8, width = 7)
op <- par(mfrow = c(2, 2), oma = c(0.1, 0.1, 0, 0), mar = c(4, 4.4, 2, 2),
          bg = gray(.97))
#------------------------------ plot 1 ------------------------------#
with(C.isotope, plot(Decimal.date, D14C, xlab = "Date", ylab =
                       expression(paste(delta^14,"C (per mille)")),
                     type = "n"))

rect(par("usr")[1], par("usr")[3], par("usr")[2], par("usr")[4],
     col = colors()[181])

with(C.isotope, points(Decimal.date, D14C, pch = 21, bg = "white"))

#------------------------------ plot 2 ------------------------------#
with(C.isotope, plot(Decimal.date, CO2, xlab = "Date", ylab =
                       expression(paste(CO[2]," (ppm)")),
                     type = "n"))

rect(par("usr")[1], par("usr")[3], par("usr")[2], par("usr")[4],
     col = colors()[363])

with(C.isotope, points(Decimal.date, CO2, type = "l"))

#------------------------------ plot 3 ------------------------------#
with(C.isotope, plot(CO2, D14C, xlab = expression(paste(CO[2], " (ppm)")),
                     ylab = expression(paste(delta^14,"C (per mille)")),
                     type = "n"))

rect(par("usr")[1], par("usr")[3], par("usr")[2], par("usr")[4],
     col = colors()[580])

with(C.isotope, points(CO2, D14C, pch = 21, bg = "yellow"))

#------------------------------ plot 4 ------------------------------#
plot(1:10, 1:10, xlab = "", ylab = "", xaxt = "n", yaxt = "n",
```

```
     type = "n")

rect(par("usr")[1], par("usr")[3], par("usr")[2], par("usr")[4],
     col = "white")
text(5.5, 5.5, expression(paste(over(
  sum(paste("(",italic(x[i] - bar(x)),")"^2),
      italic(i)==1, italic(n)),(italic(n) - 1))," = 78.4")),
  cex = 1.5)
axis(side = 1, at = c(2, 4, 6, 8), labels = c("a", "b", "c", "d"))
mtext(side = 1,
      expression(paste("Variance of ", CO[2], " concentration")),
      line = 3)
par(op)
```

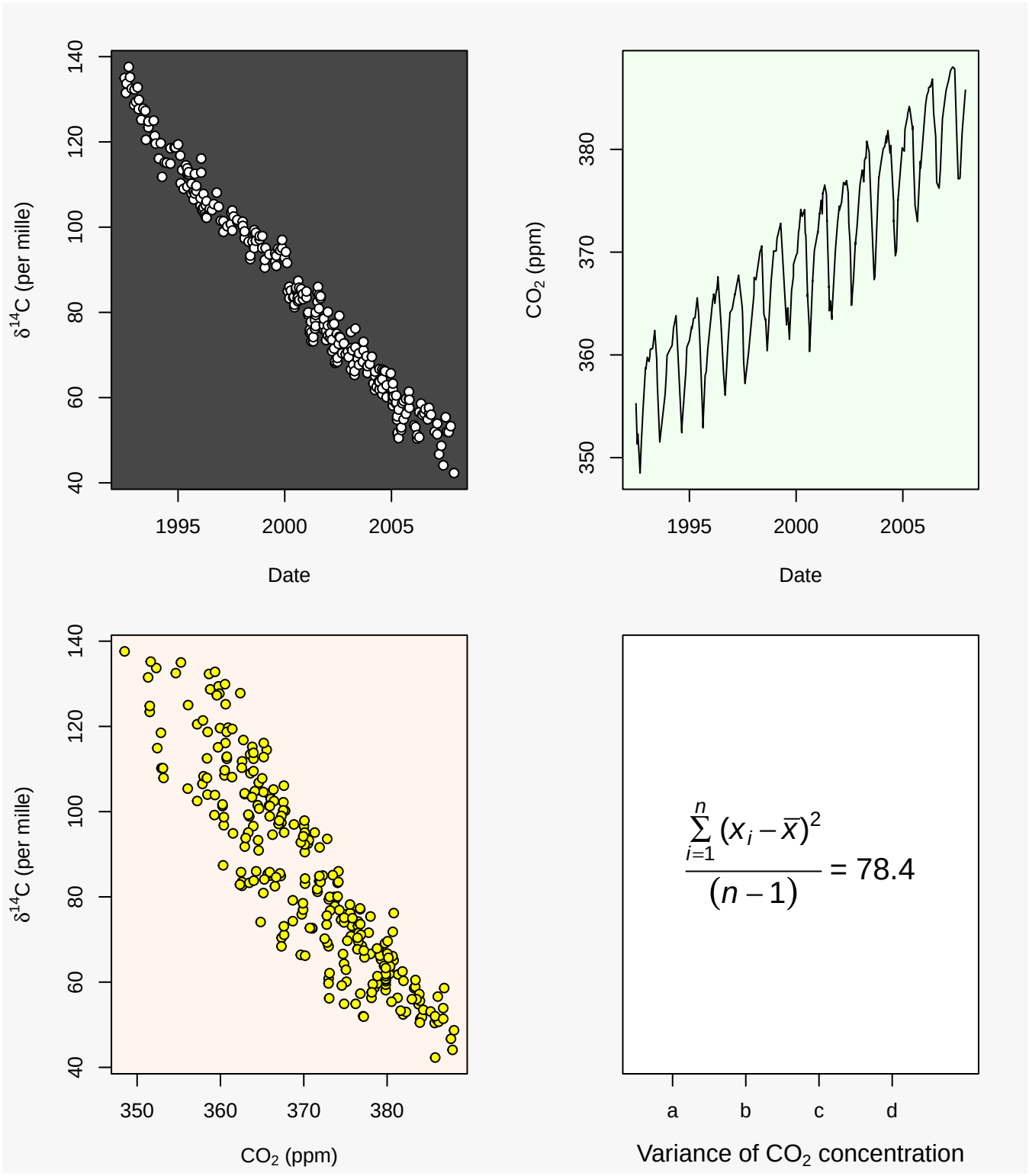


Figure 6.25: Figure resulting from summative example code.

■

6.13 Histograms

Histograms are vital for considering the distributional characteristics of quantitative data. They consist of rectangles whose height is proportional or equivalent to the frequency of particular numeric intervals (*bins*) describing that variable.

Example 6.8.
The `brycesite` dataset from package *labdsv* consists of environmental variables recorded at, or calculated for, each of 160 plots in Bryce Canyon National Park in Southern Utah.

```
library(labdsv)
data(brycesite)
```

The histogram in Fig 6.26 shows the distribution of the aspect (in degrees) of sites in the dataset.

```
with(brycesite, hist(asp, xlab = "Aspect (Degrees)", main = ""))
```

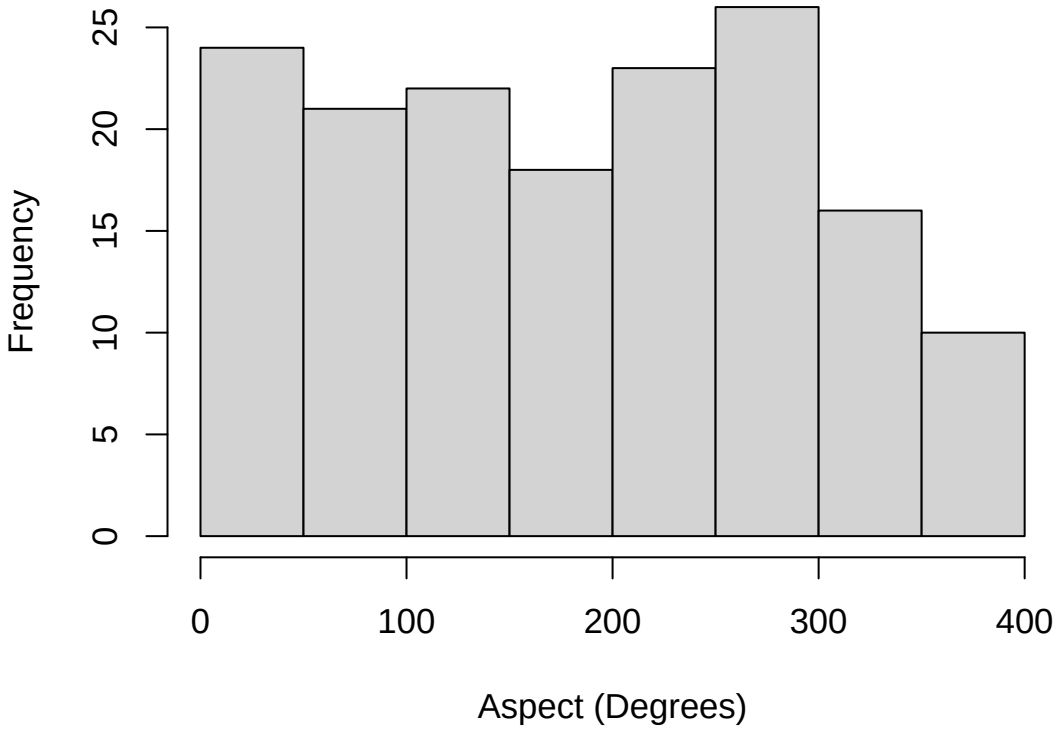


Figure 6.26: Histogram of raw aspect measures from the `brycesite` dataset.

The distribution appears remarkably uniform.

Consideration of raw aspect values in analyses is problematic because the measurements are circular. As a result the values 1 and 360 are numerically 359 units apart, although they in fact only differ by one degree. One solution is to use the transformation $[1 - \cos(\text{aspect}^{o} - 45)]/2$.

This index will have highest values on southwest slopes (at 225 degrees), and lowest values on northeast facing slopes (at 45 degrees). This acknowledges the fact that highest temperatures in the Northern Hemisphere occur on Southwest facing slopes because they receive ambient warming during the morning, coupled with late afternoon direct radiation. We have:

```
asp.val <- (1 - cos(((brycesite$asp - 45) * pi)/180))/2
```

Fig 6.27 shows the distribution of the transformed aspects which now appears bimodal.

```
hist(asp.val, xlab = "Aspect idex", main = "")
```

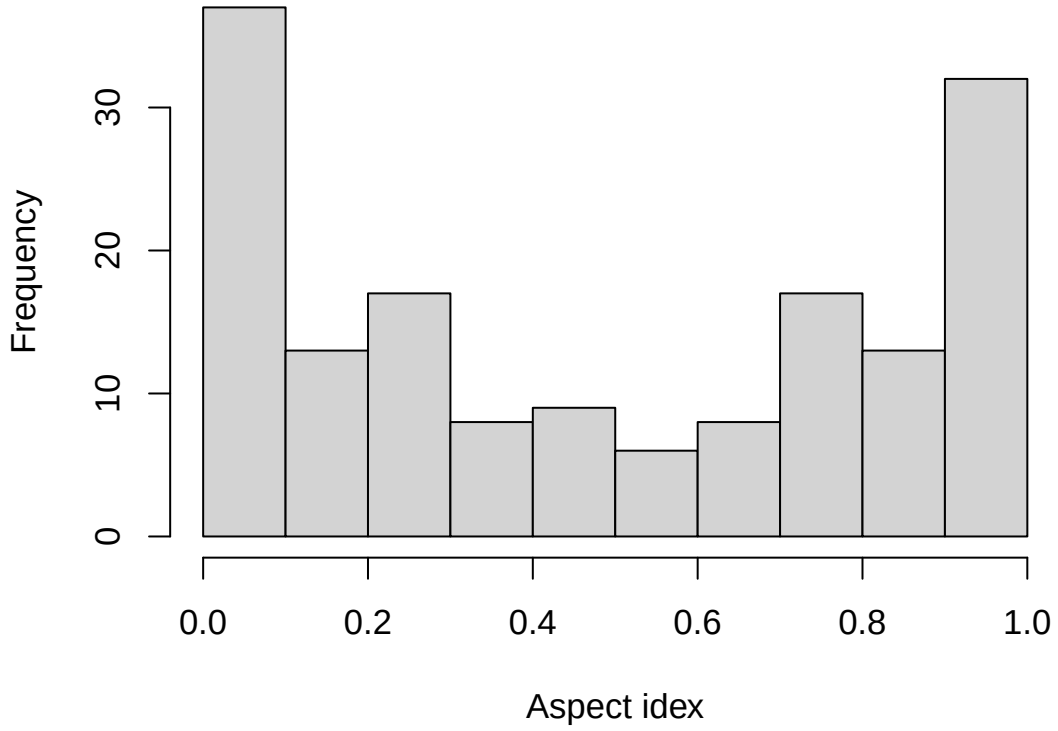


Figure 6.27: Histogram of transformed aspect measures from the `brycesite` dataset.

■

6.14 Controlling Graphical Features using Vectors

It is often useful to add information to graphical elements using a variable.

Example 6.9.
The `brycesite` contains information on incident radiation received by sites, measured in Langleys. A Langley (Ly) is a measure of energy per unit area, per unit time. To be precise, one Ly = 1 calorie m^{-2} min^{-1}. In SI units 1Ly = 41840.00 J m^{-2}. Fig 6.28 is a scatterplot of Langleys as a function of aspect index values. In addition, five topographic positions from

`brycesite$pos` are distinguished using both point color and shape. For clarity I also insert a legend. Note that ridge top sites have mostly northeastern aspect, and hence have lower radiation inputs.

```
1 with(brycesite, plot(asp.val, annrad, xlab = "Aspect value",
2                       ylab = "Annual radiation (Langleys)",
3                       col = as.numeric(pos), pch = as.numeric(pos)))
4
5 legend("bottomright", legend = levels(brycesite$pos), pch = 1:5, col = 1:5)
```

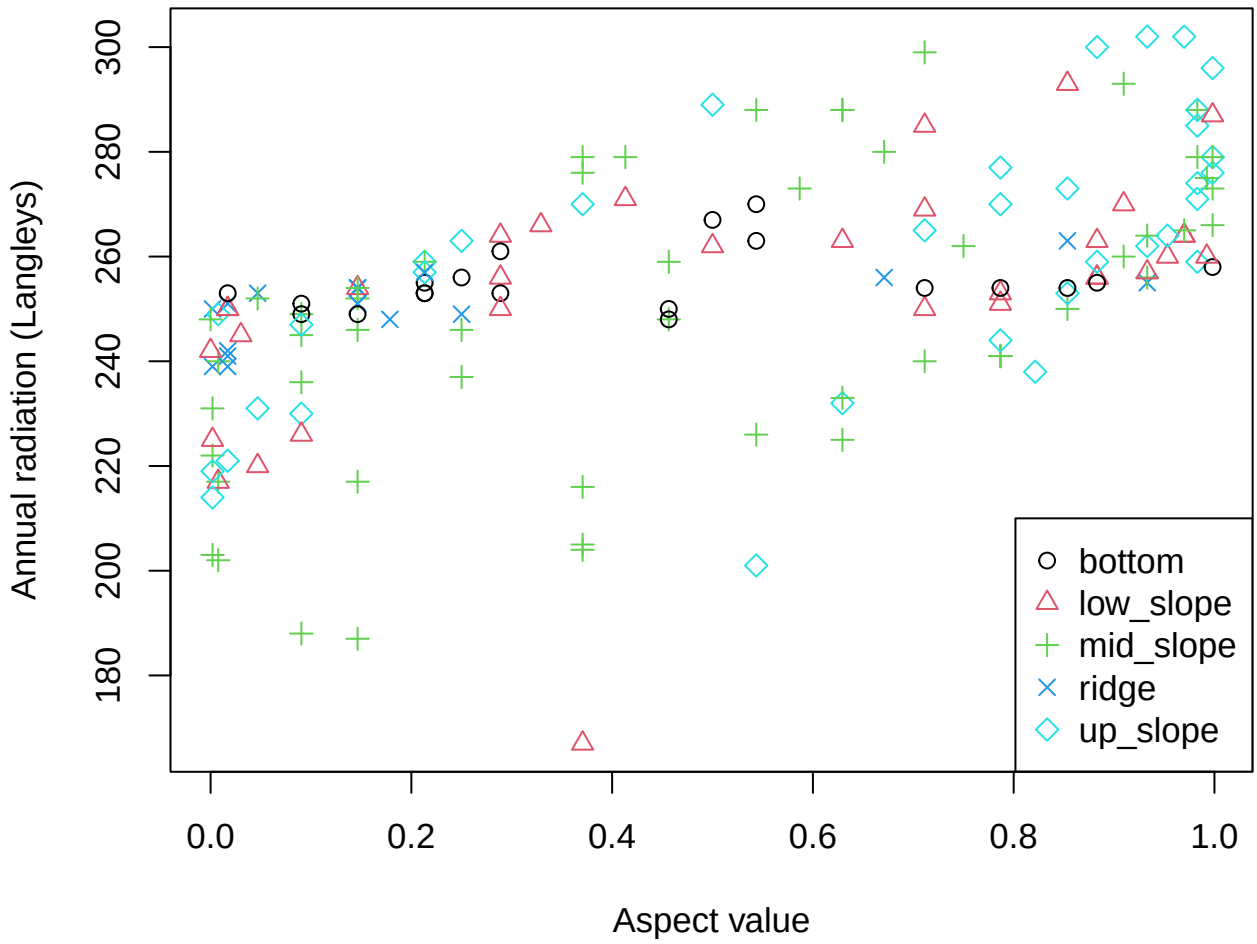


Figure 6.28: Scatterplot of aspect index value versus annual radiation with topographic positions indicated from the `brycesite` dataset.

Note that to assign colors and plotting characters appropriately, I coerce the categorical topographic position vector, `brycesite$pos`, to be numeric with `as.numeric()` (Line 3). The result is:

```
as.numeric(brycesite$pos)
```

```
  [1] 4 3 3 4 5 3 3 5 3 3 2 2 3 4 3 3 3 1 2 2 2 5 4 4 3 5 4 3 5 3 5 3 2 5 5
 [36] 4 1 1 2 4 4 3 3 3 3 4 3 5 3 3 3 2 5 3 5 3 3 5 5 4 3 3 5 2 3 3 5 2 2 5
 [71] 2 2 3 3 3 2 2 3 3 2 4 3 4 2 5 3 3 2 2 3 5 5 3 5 5 3 3 3 3 5 5 3 3 3 3
[106] 5 1 2 4 1 2 1 2 3 5 1 5 3 3 3 3 1 3 2 2 5 2 1 2 2 1 2 1 1 1 1 1 1 2 1
[141] 1 4 5 5 5 4 5 2 2 4 1 5 5 5 3 2 2 1 5 4
```

Ones correspond to the first alphanumeric level in `pos`, `bottom`, whereas fives correspond to the last alphanumeric level, `up_slope`. The color and symbols assignments are made within the plot on Line three. Base graphics legends can be created using the function `legend()` (Line 5). The first argument(s) will be a specific *x*, *y* position in the plot for the legend, or one of: `"bottomright"`, `"bottom"`, `"bottomleft"`, `"left"`, `"topleft"`, `"top"`, `"topright"`, `"right"`, or `"center"`. The `legend` argument names the categories to be depicted. The function `levels()` used in the `legend` argument lists the categories in a vector of class factor, alphanumerically.

■

6.15 Secondary Axes

For many graphical summaries it may be necessary to add additional axes. For base graphics this will involve laying one plot on top of another, by specifying `par(new = TRUE)`, and defining `axes = FALSE`, and depending on whether we want extra vertical or horizontal axes, `xlab = FALSE` or `ylab = FALSE`, and `ylab = ""` or `xlab = ""` in the arguments of the second plot.

Example 6.10.
In this example I make a scatterplot that considers both `brycesite` annual radiation and annual growing season radiation as a function of aspect value (Fig 6.29).

```
1  op <- par(mar = c(5,4.5,1,4.5), cex = 1.2)
2  with(brycesite, plot(asp.val, annrad, xlab = "Aspect value",
3                       ylab = "Annual radiation (Langleys)"))
4  par(new = TRUE)
5  with(brycesite, plot(asp.val, grorad, pch = 19, axes = FALSE, xlab = "",
6                       ylab = ""))
7  axis(4)
8  mtext(side = 4,"Growing season radiation (Langleys)", line = 3, cex = 1.2)
9  legend("bottomright", pch=c(1, 19), legend = c("Annual radiation",
10                                                "Growing season radiation"))
11 par(op)
```

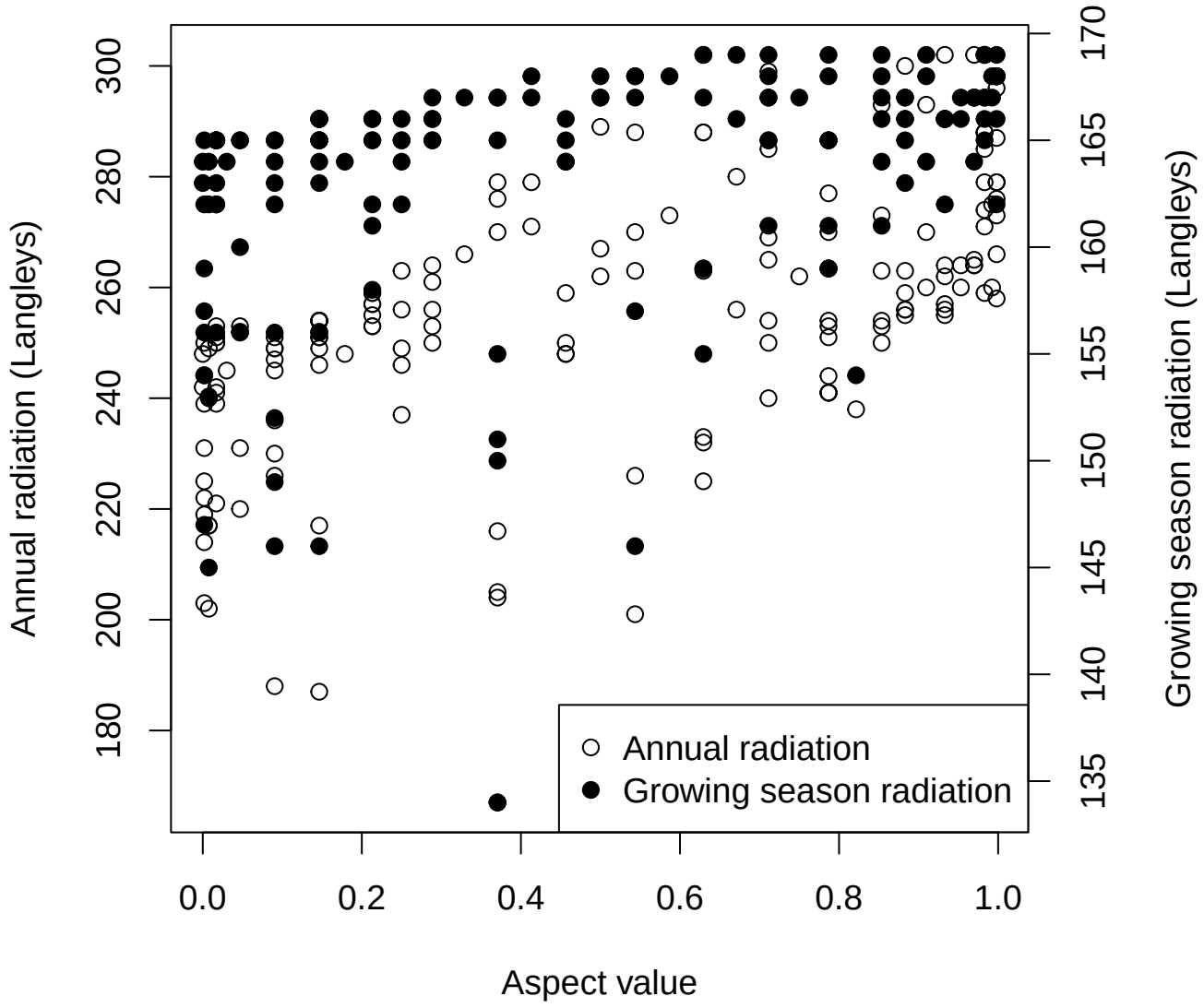


Figure 6.29: Scatterplot of annual radiation and growing season radiation as function of the aspect value index for the `brycesite` dataset.

Note the extra room given to the right hand margin (Line 1) to contain a labeling for a secondary vertical axis. The code `par(new = TRUE)` (Line 4) tells **R** not to *clean* the graphical device before drawing a new plot. The code `axis(4)` (Line 7) creates labeling for the right hand axis. The argument `axes = FALSE` in the second plot, suppresses default plot plotting of axis units on the left and bottom axes.

■

6.16 Barplots

Barplots are frequently used to compare single number summaries (e.g., sum, median, mean, etc.) of categorical levels.

Example 6.11.
Of great concern to both citizens and scientists are rising global levels of atmospheric greenhouse gasses. Atmospheric CO_2 concentrations have increased more than 40% since the start of the industrial revolution while the more potent greenhouse gasses CH_4 and NO_2 have increased approximately 150% and 23%, respectively (Brinkmann, 2009). We will take a detailed look at recent global patterns of CO_2 emissions and human population numbers in this example, while creating different sorts of barplots, and applying some of the data management techniques introduced in Chapter 4. Tidyverse data management approaches will be used for creating *ggplot2* graphics in Chapter 4. We will use the `world.emissions` dataframe from *asbio* as our data source.

```
1  library(asbio)
2  data(world.emissions)
3
4  nred <- world.emissions[world.emissions$continent != "Redundant",]
5  co2 <- with(nred, tapply(co2, country, function(x){mean(x, na.rm = T)}))
6  n <- with(nred, tapply(co2, country, function(x){length(x)}))
7
8  co2n <- data.frame(cbind(co2, n))
9  co2n.sub <- co2n[which(row.names(co2) %in% c("Canada", "China", "Finland",
10 "Japan", "Kenya", "United States")),]
11
12 labels <- paste(rownames(co2n.sub), " (", co2n.sub$n, ")", sep = "")
```

In the code above, CO_2 annual means and sample sizes for each country are computed on Lines 4-5. A subset dataset of six countries is created on Lines 7-8. Country names for this subset and the number of years of data collection are combined in an object called `labels` on Line 10. Here is the barplot code.

```
1  cols <- c("#5D3003","#347C62","#A79DBE","#DFCCC2","#994E58", "#F1F1F1")
2  barplot(co2n.sub$co2, las = 2, ylab = "", yaxt = "n", names = labels,
3          log = "y", col = cols)
4  axis(2)
5  mtext(side = 2,
6        expression(paste(CO[2], " Emissions (metric tons x ", 10^6, ")")),
7        line = 2.5)
```

The color palette on Line 1 in the code above was generated using `colorspace::hclwizard()`. Note that rotated *x*-axis labels (`las = 2`) and log-scale *y*-axis are are specified on the call to `barplot()` on Lines 2-3. A customized *y*-axis is constructed on Lines 4-7. This is done largely to force the axis tick labels to be have a default vertical format. They would be vertical otherwise because of the use of `las = 2` in `barplot()`. Figure 6.30 shows the shows the final result.

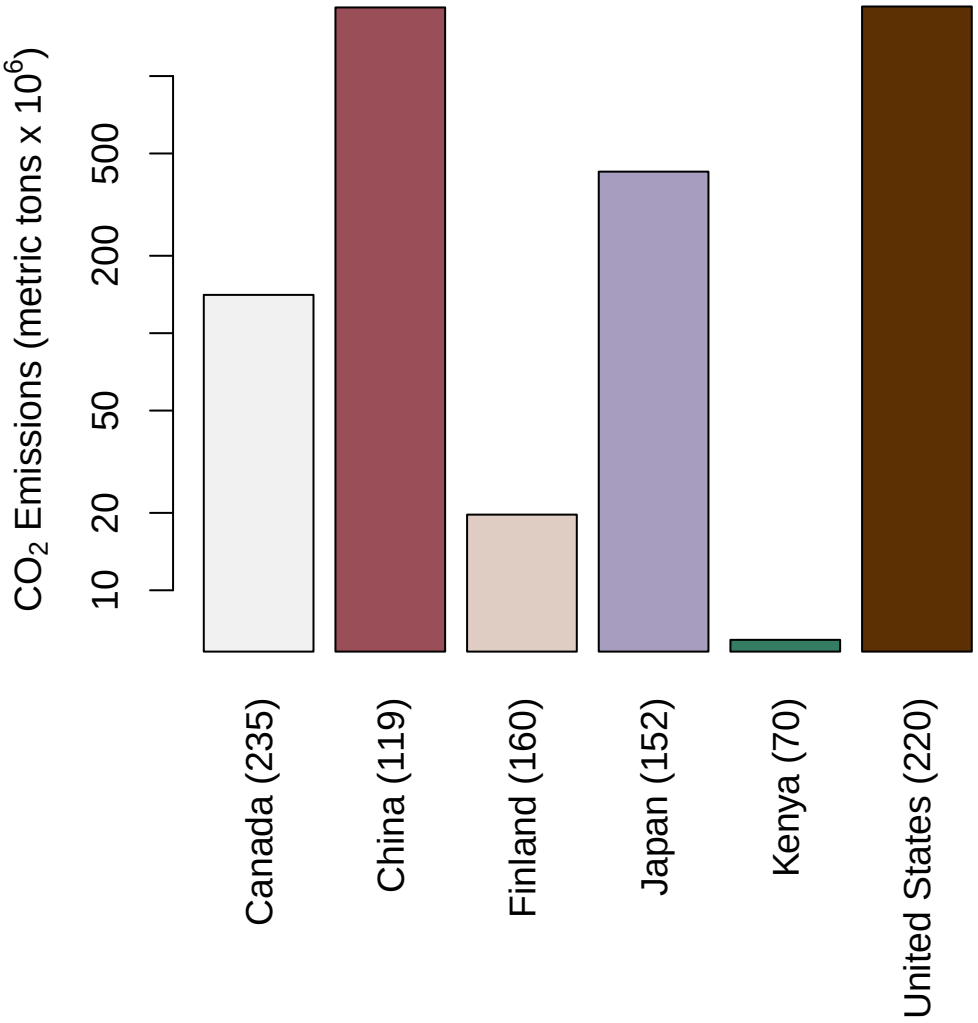


Figure 6.30: Barplot of mean annual CO_2 emmission levels for six countries. Number of years used in computing means indicated in parentheses.

To depict trends since the year 2000, we can use a stacked barplot (Fig 6.31), or a side by side barplot (Fig 6.32) by applying `barplot()` to a matrix with columns representing categories.

In the code below we subset the data by country (Lines 1-4) and year (Line 6), and create a dataframe containing CO_2 and country data (Line 8), which is converted to a wide format matrix using `unstack()` (Line 9).

```
1 csub <- world.emissions[
2   which(world.emissions$country %in%
3           c("Canada", "China", "Finland",
4             "Japan", "Kenya", "United States")),]
5
6 ysub <- csub[which(csub$year >= 2000),]
7
```

```
dat <- data.frame(co2 = ysub$co2, country = ysub$country)
dat1 <- as.matrix(unstack(dat))
```

In the code below, hexadecimal colors generated by `colorspace::hclwizard()` are brought in (Line 1) and modified, i.e., unlisted, coerced to be a character vector and reversed (Line 2), preceding creation of the barplot (Lines 4-8).

```
cols <- read.table("colormap_hex.txt") # file from hclwizard()
cols <- rev(as.character(unlist(cols)))

barplot(dat1, log = "y", col = cols, yaxt = "n", las = 2, names = labels)
axis(2)
mtext(side = 2,
      expression(paste(CO[2], " Emissions (metric tons x ", 10^6, ")")),
      line = 2.5, cex = 1.2)
```

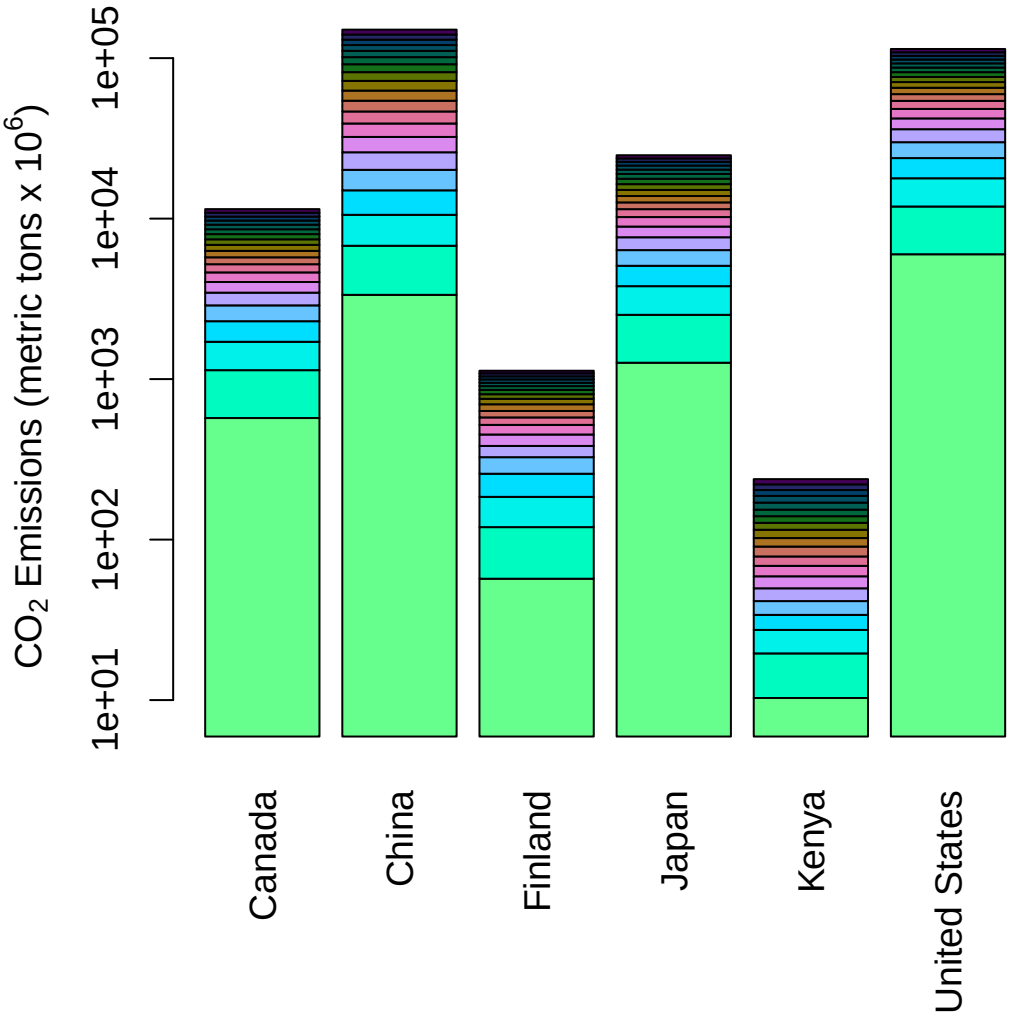


Figure 6.31: Stacked barplot of mean annual CO_2 emmission levels for six countries from 2000-2019.

Side by side barplots are generated by specifying `beside = TRUE` in `barplot()` (Line 2 in code below).

```
1 barplot(dat1, log = "y", beside = TRUE, col = cols, yaxt = "n", las = 2)
2
3 axis(2)
4 mtext(side = 2,
5       expression(paste(CO[2], " Emissions (metric tons x ", 10^6, ")")),
6       line = 2.5, cex = 1.2)
```

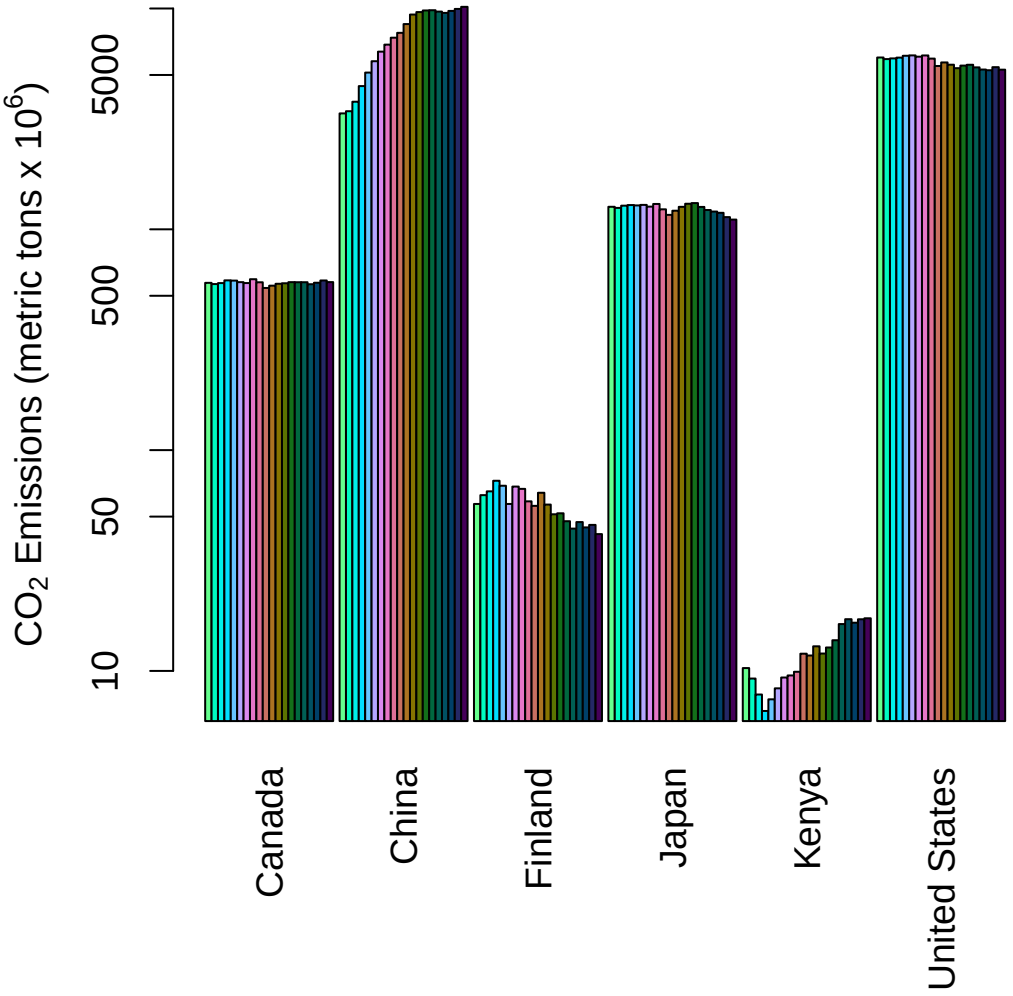


Figure 6.32: Side by side barplot of mean annual CO_2 emmission levels for six countries from 2000-2019.

■

6.17 Boxplots

Boxplots or *box and whisker plots* and their variants are an excellent way to quickly summarize and compare the distributions of levels in a categorical variable with respect to a quantitative variable. The function `boxplot()` does this by graphically providing a five number summary for factor levels (Fig 6.33). Specifically, the upper and lower hinges of boxes from boxplot show the 1st and 3rd quartiles (thus the box contains the central 50% of the data). The black stripe in the middle of each box shows the median. The whiskers extend to the most extreme data point which is no more than `coef` times the length of the box away from a hinge, where `coef` is defined in the arguments for `boxplot()` (by default `coef = 1.5`). Circle symbols outside of whiskers can be considered outliers (*cf.*, Tukey et al., 1977).

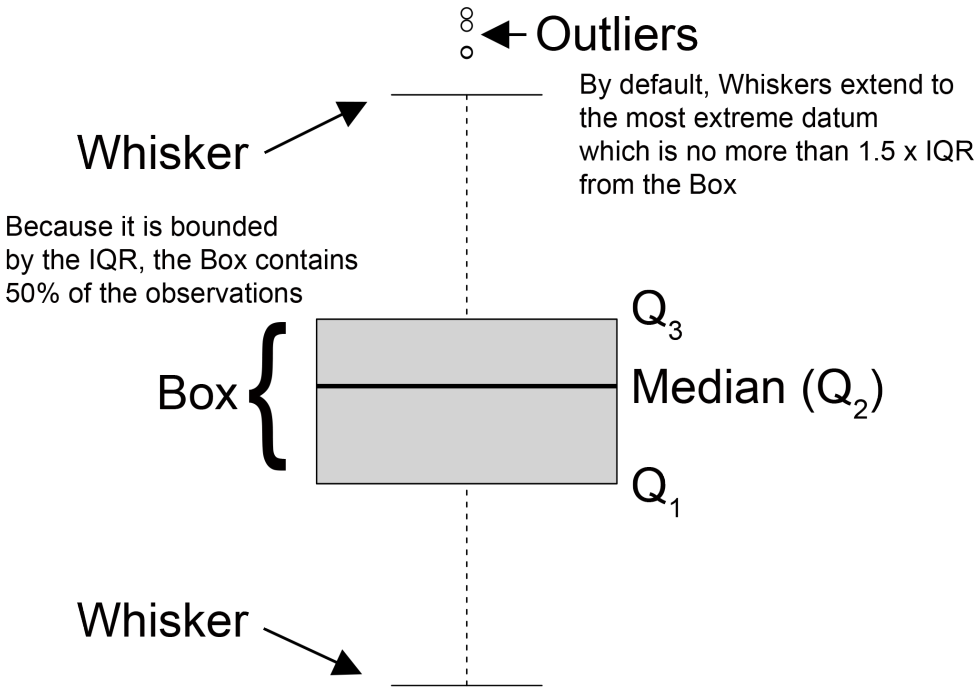


Figure 6.33: A summary of boxplot characteristics.

Example 6.12.
Here we reconsider the `world.emissions` data using boxplots. Recall our approach from the previous exercise:

```
csub <- world.emissions[
  which(world.emissions$country %in% c("Canada", "China", "Finland",
                                       "Japan", "Kenya", "United States")),]
ysub <- csub[which(csub$year >= 2000),]

dat <- data.frame(co2 = ysub$co2, country = ysub$country)
```

We can use `plot(q ~ c)` (Fig 6.2) or `boxplot(q ~ c)` to make boxplots, where `q` is a vector of quantitative data and `c` is a corresponding vector of categorical data.

```
cols1 <- rev(c("#5D3003","#347C62","#A79DBE",
               "#DFCCC2","#994E58", "#F1F1F1"))
with(dat,
     boxplot(co2 ~ country, col = cols1, las = 2, xlab = "",
             ylab =
               expression(
                 paste(CO[2], " Emissions (metric tons x ", 10^6, ")"))))
```

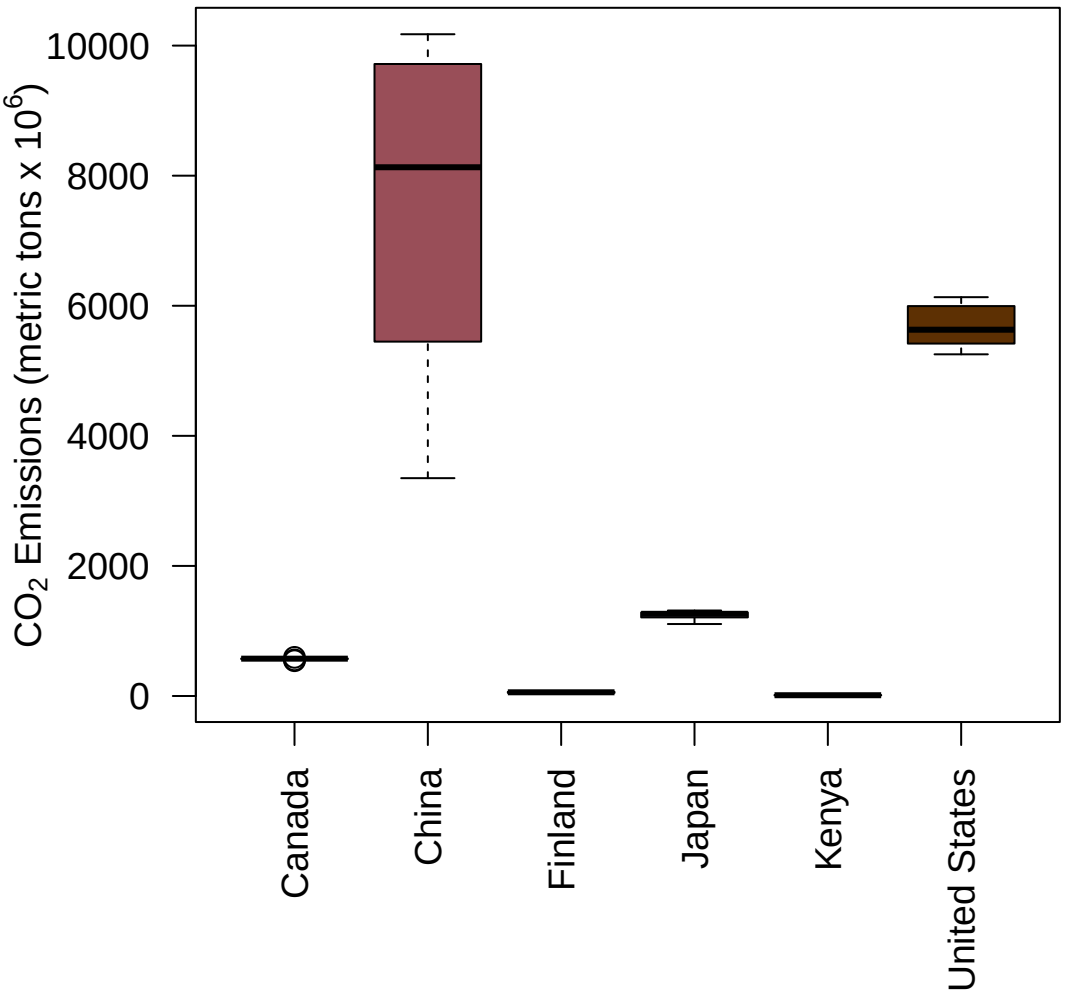


Figure 6.34: Boxplots of annual CO_2 emmission levels for six countries from 2000-2019.

■

6.17.1 Violin Plots

Another graphical tool for comparing probability distributions is a *violin plot*. It contains contains similar components to a boxplots, including designation of boxes and whiskers, along with a rotated kernel density plot on each side. Thus, it allows additional consideration of the skew, kurtosis and potential multimodality of distributions. The function `vioplot` from the package *vioplot* allows base graphics generation of violin plots.

Example 6.13.
As an example we will compare violin plots based on random sampling of a bimodal, uniform and normal distribution (see `?vioplot`). Note the kernel density generator fits a oblique sphere, although the uniform PDF is a rectangle (Fig 6.35).

```
library(vioplot)

mu <- 2
sig <- 0.6
bimodal <- c(rnorm(1000,-mu, sig), rnorm(1000, mu, sig))
uniform <- runif(2000, -4, 4)
normal <- rnorm(2000, 0, 3)
vioplot(bimodal, uniform, normal, col = cols1[1:3],
        names = "Bimodal", "Uniform", "Normal")
```

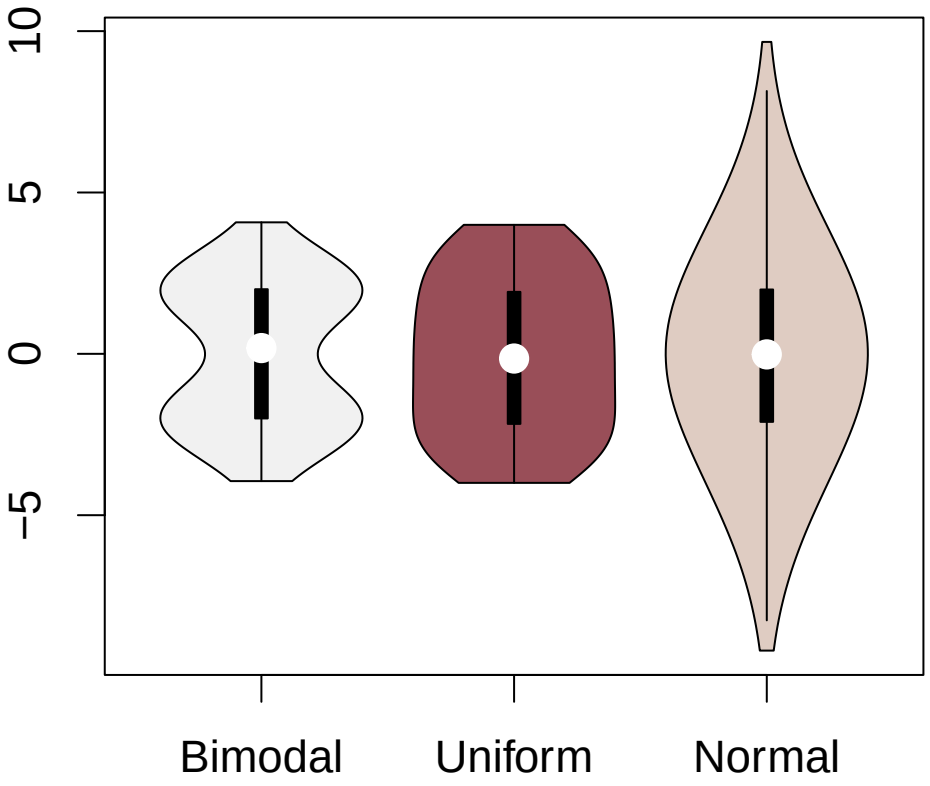


Figure 6.35: Violin plots based on random sampling from a bimodal, uniform, and normal distribution.

■

6.18 Interval Plots

Interval plots display location measures (e.g. means, medians, etc.), typically as bars, along with error bars representing measures of data dispersion (e.g., standard errors, standard deviations, confidence intervals, interquartile ranges, etc.). Thus, barplots and boxplots can be considered special types of interval plots.

Example 6.14.
As an example, we will create an interval plot by hand using a classic dataset from R.A. Fisher that records the yield of different varieties of potatoes. The data are in the dataframe `asbio::potato`. Here are the means and the standard errors of the mean.

```
data(potato)
means <- with(potato, tapply(Yield, Variety, mean))
head(means)
```

```
         Ajax Arran comrade British queen  Duke of York       Epicure
       3.3400        2.2622        3.1367        1.7778        2.1600
   Great Scot
       3.4033
```

```
ses <- with(potato,
            tapply(Yield, Variety, function(x){sd(x)/sqrt(length(x))}))
head(ses)
```

```
         Ajax Arran comrade British queen  Duke of York       Epicure
     0.305941      0.070902      0.181184      0.148313      0.146771
   Great Scot
     0.140929
```

We will plot the means using a barplot and save the horizontal locations of bars as a object `bloc`.

```
bloc <- barplot(means, las = 2, ylab = "Yield (lbs per plant)", col = cols)
```

We will then overlay error bars using the function `segments()` or `arrows()`.

```
segments(x0 = bloc, y0 = means - ses, y1 = means + ses, x1 = bloc)
```

The result is shown in Fig 6.36.

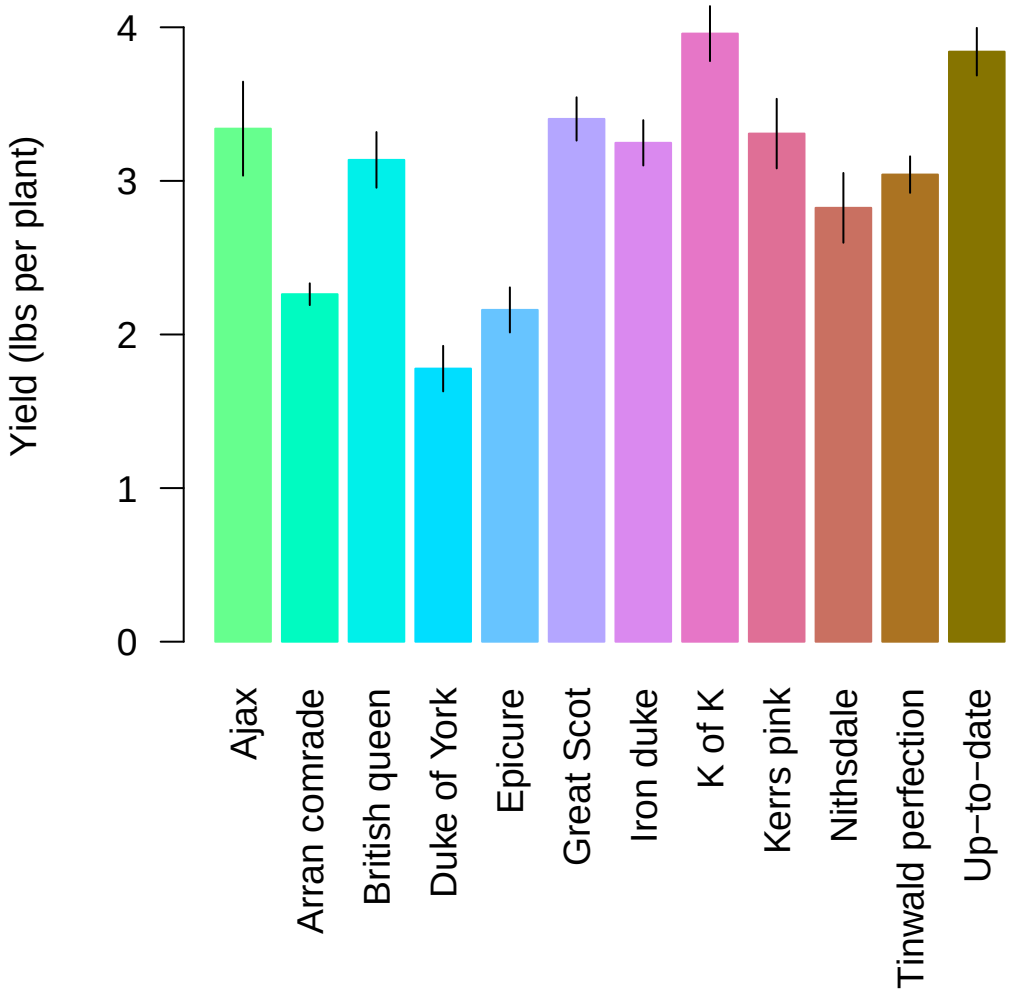


Figure 6.36: Interval plot of the Fisher potato dataset. Bar heights are means, error bars represent $\bar{x} \pm \hat{\sigma}_{\bar{x}}$.

■

The function `asbio::bplot` allows straightforward creation of interval plots from specified explanatory and response variables. A large number of location and dispersion measures can be specified in the function arguments. To recreate Fig 6.36 one could simply type:

```
with(potato, bplot(y = Yield, x = Variety, bar.col = cols, border = cols))
```

6.18.1 Pairwise Comparisons

An important component of many biological analyses are multiple pairwise comparisons of means (or other location measures). These tests will often require control of Family-Wise type I Error Rate (FWER), that is, the probability of incorrectly rejecting at least one true null hypothesis in a family of related tests. The most powerful method for controlling FWER in

a *post hoc* family of pairwise tests, following an omnibus ANalysis Of VAriance (ANOVA), is Tukey's honest significant difference (see Aho (2014)).

Example 6.15.
Zelazo et al. (1972) performed a series of experiments to determine whether certain exercises could allow infants to learn to walk at a younger age. The experimental treatments were: Active Exercise (AE), Passive Exercise (PE), Test-Only (TO), and Control (C). The data are in the dataframe `asbio::baby.walk`. For more information type `?baby.walk`.

Rejection of the omnibus ANOVA null hypothesis of no mean treatment differences, allowed pairwise comparison of treatment means using Tukey's procedure. We will use the function `asbio::pairw.anova()` to run this analysis.

```
data(baby.walk)
tukey <- with(baby.walk, pairw.anova(y = date, x = treatment))
tukey
```

```
95% Tukey-Kramer confidence intervals

              Diff    Lower    Upper  Decision Adj. p-value
muAE-muC    -2.225 -4.35648 -0.09352 Reject H0     0.038997
muAE-muPE   -0.525 -2.65648  1.60648    FTR H0     0.897224
muC-muPE      1.7 -0.52625  3.92625    FTR H0     0.172932
muAE-muTO -1.58333 -3.61562  0.44895    FTR H0     0.160457
muC-muTO   0.64167 -1.48981  2.77314    FTR H0     0.829542
muPE-muTO -1.05833 -3.18981  1.07314    FTR H0     0.513366
```

Interval plots can be used to summarize these comparisons. The `plot` method for objects of class `pairw` calls `bplot()` for this purpose. In particular, we have:

```
plot(tukey, ylab = "Months until walking", cex.lett = 1.2)
```

```
Bars are means.  Errors are SEs.

The population means of factor levels with the same letter are not
significantly different at alpha = 0.05 using the Tukey HSD method.
```

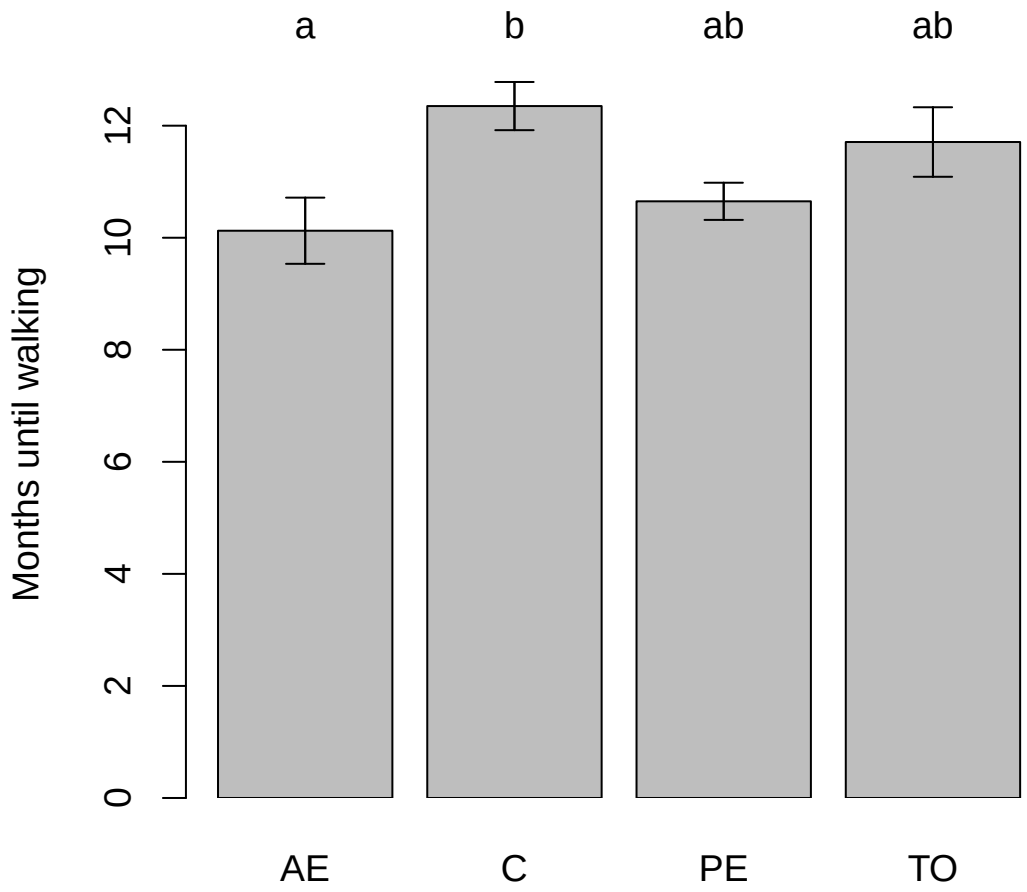


Figure 6.37: An interval plot summarizing the results of pairwise comparisons for the `baby.walk` example.

As stated in the `plot.pairw()` function output (Fig 6.37), letters above bars summarize the result of pairwise comparisons. In particular, factor levels means with the same letter are not significantly different using the conventional FWER $\alpha = 0.05$.

■

We will look at more sophisticated graphical methods for pairwise comparisons in Ch 7.

6.19 `matplot()`

The function `matplot()` allows one to plot the columns of one matrix against the columns of another. There is no clear *ggplot2* (Section 7.3) alternative to `matplot()` because *tidyverse* functions require data to be in a long table format, whereas `matplot()` works best with data in a wide table format.

Example 6.16.
To demonstrate `matplot()` we will use the `dat1` dataset, used to create Fig 6.31, which contains annual CO_2 levels from 2000-2019 for six countries.

```
1  par(mar = c(3,4.5,5,2), cex = 1.1)
2  matplot(x = 2000:2019, y = dat1, col = cols1, type = "l", lwd = 1.5,
3          log = "y",
4          ylab = expression(
5            paste(CO[2], " Emissions (metric tons x ", 10^6, ")")))
6
7  legend(x = 2001, y = 80000, xpd = TRUE,
8         lty = 1:5, ncol = 2, lwd = 1.5, bty = "n",
9         col = cols, legend =
10          c("Canada", "China", "Finland",
11            "Japan", "Kenya", "United States"))
```

In the code above, note that I allocate additional room in the top of the graph for a legend (Line 1). Note that the response variable is a matrix of CO_2 values whose columns delimit countries (Line 2). The `xpd` argument in `legend()` allows plotting to be clipped to the device region which will generally exceed the plot region (Line 6). The result is shown in Fig 6.38.

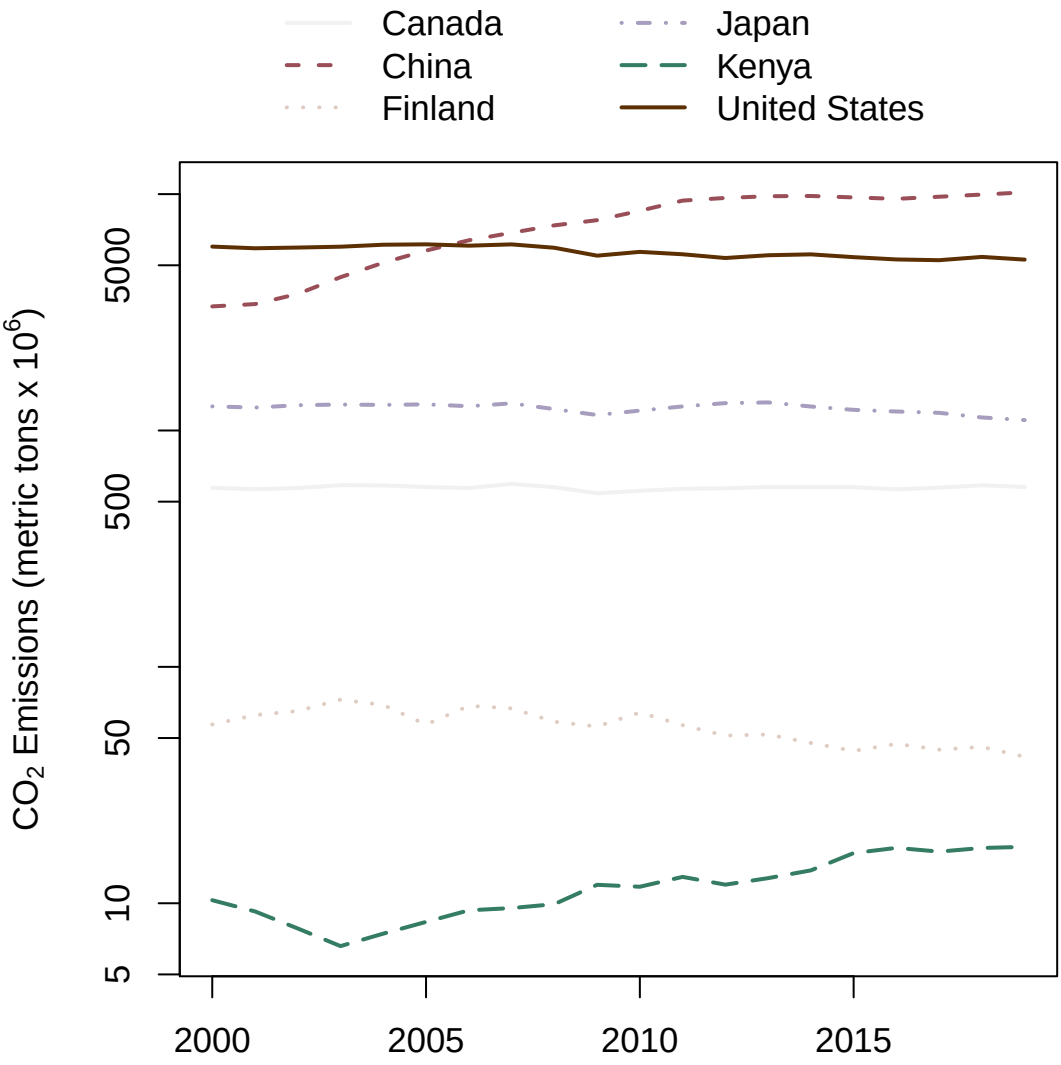


Figure 6.38: A matrix line plot.

■

6.20 Interactivity

As noted earlier, **R** graphics are generally non-interactive. Some *graphical interactivity* is allowed via the function `locator()`, which returns graphical coordinates where a mouse click occurred in plot, and `identify()`, which can be used to add labels and symbols to mouse click locations. For instance, try:

```
dev.new(RStudioGD = FALSE) # If one is using RStudio
plot(1:10)
identify(1:10, labels = 1:10)
```

For Windows, X Window, and Cairo graphics devices, more sophisticated methods exist for

interactivity. In these settings, the function `setGraphicsEventHandlers()` can be used to call functions when events such as mouse clicks or keystrokes occur. For instance, open an appropriate graphics device and run the example in:

```
?getGraphicsEvent
```

Still other interactive options are possible using animations and hand rotatable graphics. These approaches, which are often transferable to Markdown HTMLs, are considered briefly in the next two sections of this chapter. Animations using the package *ggplot2* are considered in Ch 7. GUI driven graphics interactivity is also possible, and is described in Ch 11.

6.21 Three Dimensional Graphics

It is often necessary to consider more than two variables in biological graphics. We have seen that this can be done in a number of different ways, including the use additional colors, multiple line or symbol types (Section 6.14, Fig 6.28), the use of additional axes in a two dimensional context (Section 6.15, Fig 6.29)), multiple symbol sizes, or formal three-dimensional plots.

Example 6.17.
To further consider three dimensional plotting we will use two datasets from the package *vegan* which describe taiga/tundra ecosystems at particular Scandinavian sites. Plant, moss, and lichen species abundances are given in the dataset `varespec`, and soil chemistry data for the same sites are contained in the dataset `varechem`.

```
library(vegan)
data(varespec)
data(varechem)
```

In Fig 6.39 we examine the distribution of the heath plant *Vaccinium vitis-idaea* (a common species in boreal forest understories) with respect to both pH and soil percent nitrogen. This is done by altering symbol sizes with the abundance of *V. vitis-idaea*.

```
with(varechem, plot(N, pH, xlab = "% soil N", pch = 16,
                    cex = varespec$Vaccviti/100 * 15))
```

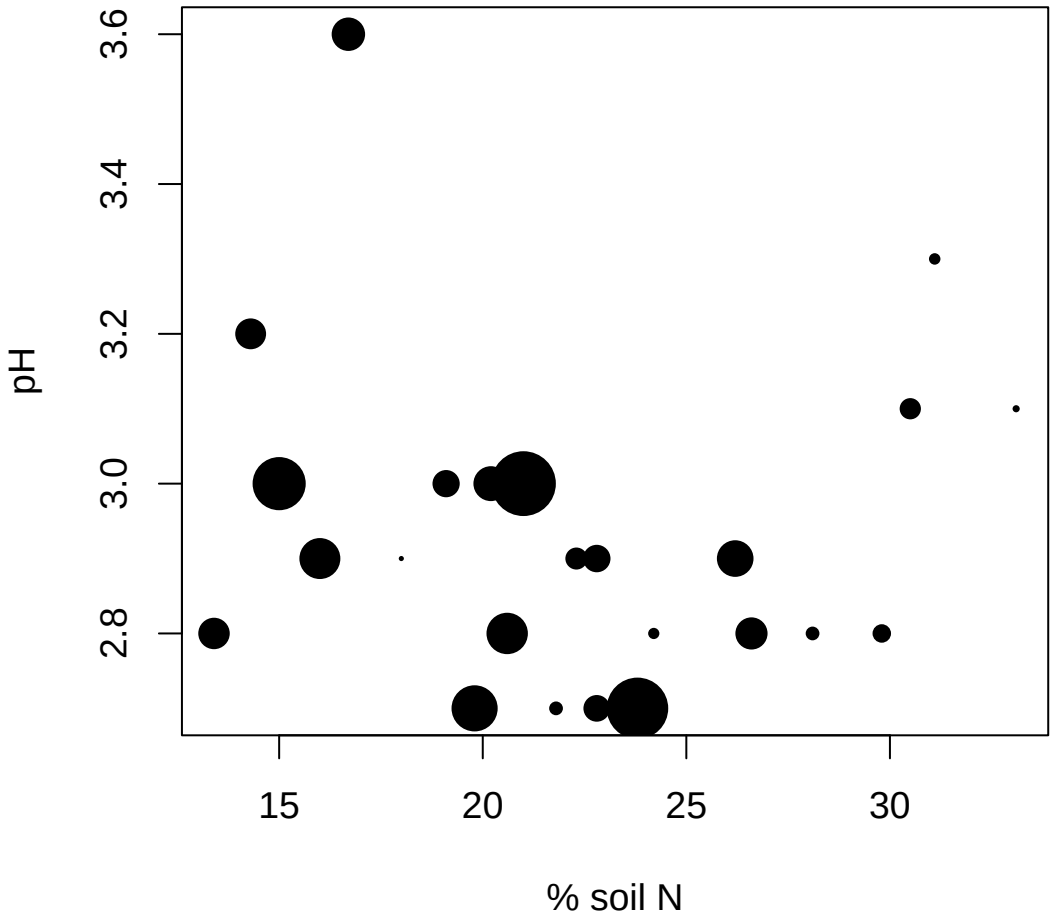


Figure 6.39: Cover of *Vaccinium vitis-idaea* with respect to pH and % soil nitrogen. Larger symbols indicate higher percent plant cover.

Vaccinium vitis-idaea appears to prefer intermediate to low levels of soil N, and acidic soils. The somewhat negative association between soil N and pH is probably due to soil leaching, because H^+ (and Al^{3+}) cations are more strongly adsorbed by soil colloids than bases in poorly drained soils.

A 3D plot of the same associations can be created using the `scatterplot3d()` function from the package *scatterplot3d* .

```
library(scatterplot3d)
Fig <- function(angle = 55){
s3d <- scatterplot3d(cbind(varechem$N, varechem$pH, varespec$Vaccviti),
type = "h", highlight.3d = TRUE, angle = angle, scale = .7, pch = 16,
xlab = "N", ylab = "pH", zlab =
  expression(paste(italic(Vaccinium), " ", italic(vitis-idaea),
                   " % cover")))

lm1 <-lm(varespec$Vaccviti ~ varechem$N + varechem$pH)
```

```
10 s3d$plane3d(lm1)
11 }
12 Fig()
```

In the code above, I define the figure to be a function (named `Fig`) to allow the angle of rotation in the 3D scatterplot to be easily changed using the `angle` argument in `Fig` (Line 2). Functions will be addressed in detail in Ch 8. By stipulating `highlight.3d = TRUE` (Line 4), objects that are closer to the viewer with respect to the `x` plane are given warmer colors. A regression "plane" is also overlaid on the graph (Lines 9-10). The fitted plane is produced from a multiple regression model created by the function `lm()`.

The result is shown in Fig 6.40.

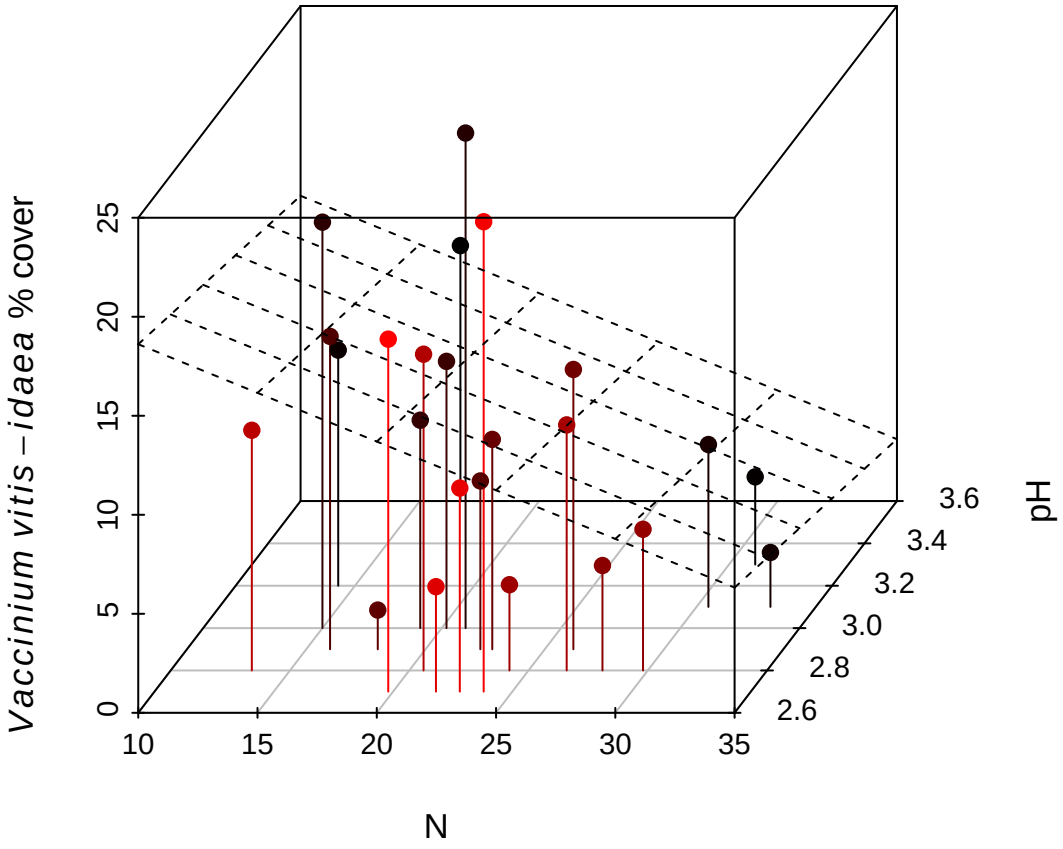


Figure 6.40: Cover of *Vaccinium vitis-idaea* with respect to pH and % soil nitrogen, depicted in a 3D scatterplot.

■

6.22 Animation

Animations can be created in **R** to illustrate a wide range of processes (Xie, 2013; Xie et al., 2018b). Functions with animation are generally based on loops (Section 8.5) with some method of slowing the loop; usually the function `Sys.sleep()`.

Example 6.18.
Here we add animation to the 3D scatterplot shown in Fig 6.40. This will be facilitated by the fact that the plot is a function with an argument whose alteration results in modification of the graph.

```
1  fig.rot <- function(){
2  lapply(seq(1, 360), function(i){
3    Fig(i)
4    Sys.sleep(.1)
5    })
6  }
7
8  fig.rot()
9  # save frames into one GIF:
10 library(animation)
11 saveGIF(fig.rot(), interval = 1, movie.name = "vaccinium.gif")
```

Recall that `lapply()` returns a list of the same length as its first argument `X`, whose elements result from applying a function, given in the second argument, to corresponding elements of `X`. In the code above, an argument-less function is created containing a loop run by `lapply()` (Lines 2 - 5). As the loop index `i` changes from `i = 1` to `i = 360` (Line 2) this changes the `angle` argument in the function `fig()`, used in creating Fig 6.40. At the end of each step in the loop, the system is paused for a tenth of second (Line 4) with the function `Sys.sleep()` to allow each "frame" of the animation to be viewed separately. In the (optional) last two lines of code, the **R** package *animation* is loaded, and the function `animation::saveGIF()` is used to save the animation in a GIF file format[7].

The animation result is shown in 6.41.

[7] Use of `saveGIF` requires installation of open source software ImageMagick or GraphicsMagick (see `?saveGIF`).

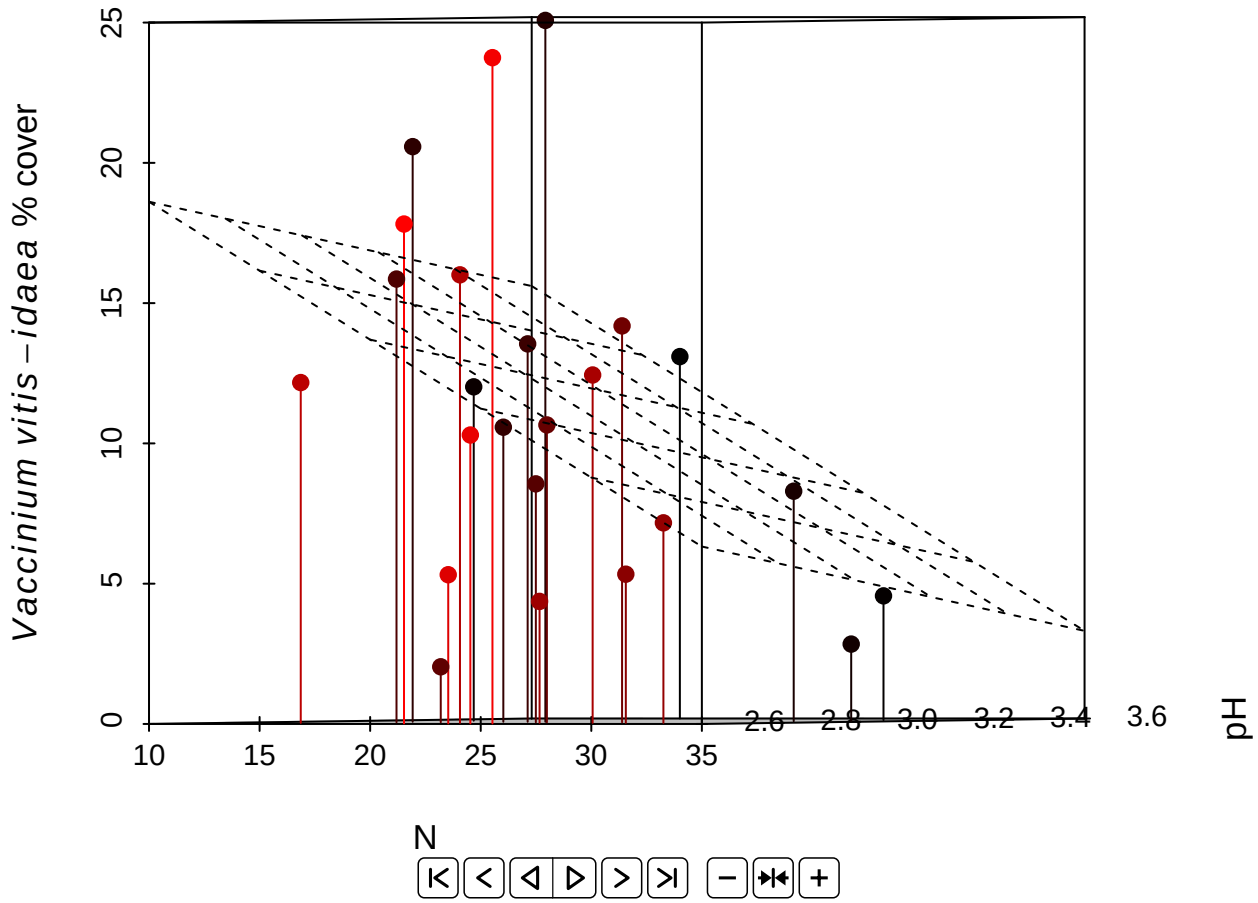


Figure 6.41: Animated version of the 3D scatterplot from Fig 6.40. Animation controls are provided by the LaTeX package *animation*.

■

Working animations generated in **R** can be placed into HTML documents created under **R** Markdown, or PDF documents under Sweave-alike approaches (Section 2.10.2). The former method currently requires installation of the *gifski* **R** package and the specification: `animation.hook = "gifski"` among the chunk options for the animation. The latter approach requires loading of the *animate* LaTeX package and using the chunk option `fig.show = "animate"`. PDF animations can viewed using a number of PDF viewers including the Foxit® and Adobe® Acrobat Readers.

Example 6.19.
We can also create hand-rotatable 3D figures under the *rgl* real-time rendering system.

```
Attaching package: 'rgl'

The following object is masked from 'package:plotrix':

    mtext3d
```

```
1 expg <- expand.grid(varechem$pH, varechem$N)
2 subs <- cbind(varechem$pH, varechem$N)
3 tf <- (expg[,1] == subs[,1]) & (expg[,2] == subs[,2])
4 y <- ifelse(tf == TRUE, varespec$Vaccviti, NA)
5 surface <- data.frame(N = expg[,1], pH = expg[,2], vac.vit = y)
6
7 library(car)
8 scatter3d(vac.vit ~ N + pH, data = surface, surface = TRUE, fit = "linear",
9 zlab = "N", xlab = "pH", ylab = "Vaccinium vitilus (% cover)")
```

In the code above, a initial surface is created that considers all possible combinations of pH and N outcomes (Line 1) and actual occurrences of `varespec$Vaccviti` at *observed* combinations (Lines 3 and 4). The function `scatter3d()` in the package *car* (Fox and Weisberg, 2019) uses tools from the *rgl* package (Murdoch and Adler, 2025) to render a three dimensional scatterplot. The scatterplot will be rotatable within an **R** session, and can be rendered as a rotatable graphic in an **R** Markdown HTML[8] and in some PDF frameworks. Plots from *rgl* can also be rendered and manipulated in Shiny apps (see Ch 11).

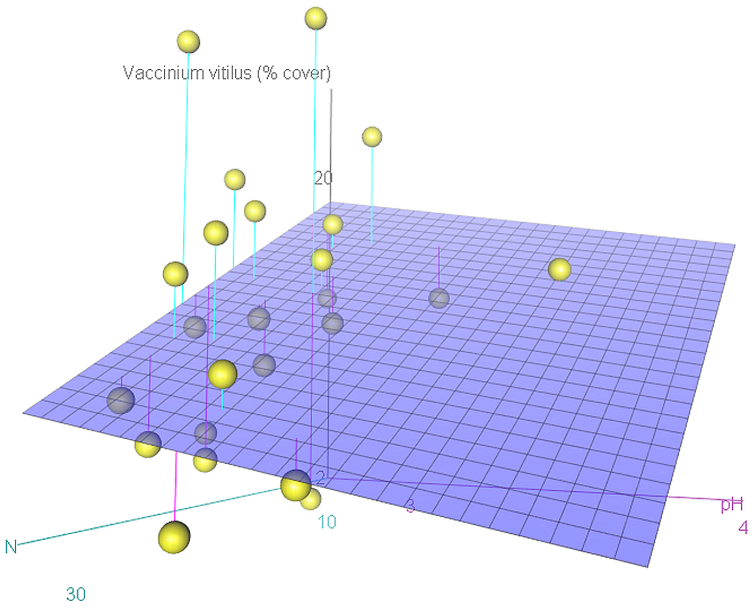


Figure 6.42: A hand rotatable graphics object (within an **R** interactive or suitable PDF/HTML environment).

■

[8] See `rgl::playwidget()` if you you are reading this document as a pdf.

Exercises

1. Consider the variables: `x <- c(1,2,2.5,3,4,3,5)` and `y <- c(6,4.3,3,3.1,2,1.7,1)`.
   (a) Make a plot with `x` defining the *x*-axis and `y` on the *y*-axis.
   (b) Make every point in the scatterplot a different color.
   (c) Make every point a different shape.
   (d) Create a legend describing all the shape and color combinations of all points one through seven (call them Point 1, Point 2, etc.).
   (e) Convert from a point to an overplotted line and point plot.
   (f) Change the label of the *x*-axis to "*Abscissa* axis" and the label of the *y*-axis to be "*Ordinate* axis" using a `plotmath` approach. This will require use of the functions `expression()`, `paste()` and `italic()`.
   (g) Place the text "y = -1.203x + 6.539" at coordinates `x = 2, y = 2.5` using the function `text()`. Italicize as indicated.
   (h) Place a line with a slope of -1.203 and an *y*-intercept of 6.539 on the plot using the function `abline()`.

2. The `Indometh` dataframe from the package *datasets* describes pharmacokinetics of the drug indomethacin following intravenous injections for six human subjects.
   (a) Create a histogram for the variable `conc`, which gives plasma concentrations of indomethacin in (mcg/ml) in subjects over time. Use an appropriate *x*-axis label.
   (b) Create a scatterplot of `conc` as a function of `time` (in hours). Create appropriate axis labels.
   (c) Change symbols and colors of points in (b) based on levels in `Subject`.
   (d) Create a wide table format for `Indometh` using: `wide <- unstack(Indometh, conc ~ Subject)` and `names(wide) <- paste("Subject", c(1,4,2,5,6,3))`.
   (e) Create a stacked barplot and a side by side barplot based on: `barplot(as.matrix(wide))`.
   (f) Use appropriate *y*-axis labels.
   (g) Create a multiple line plot (with a line for each subject) using: `time <- c(0.25, 0.50, 0.75, 1.00, 1.25, 2.00, 3.00, 4.00, 5.00, 6.00, 8.00)` and `matplot(x = time, y = wide, type = "l")`
   (h) Generate appropriate axis labels for the plot.
   (i) Create an appropriate legend for the plot created in (h). The colors and line types used by `matplot()` will be 1:6. The order of subjects is 1, 4, 2, 5, 6, 3.

3. The dataframe `life.exp` from *asbio* compares life expectancy of field mice given five different diets.
   (a) Make and interpret a boxplot showing `lifespan` as a function of `treatment`.
   (b) Make an interval plot by hand showing `lifespan` as a function of `treatment` using means as measures of location, and standard deviations to generate error bars.

4. (Advanced) Conduct an ANOVA and a *post hoc* pairwise comparison of means with Tukey's HSD using: `anova(lm(lifespan ~ treatment, data = life.exp))`, `tukey <- with(life.exp, pairw.anova(lifespan, treatment))`.
   (a) Create an interval plot summarizing these results using: `plot(tukey)`.
   (b) Interpret (a).

5. Load the `C.isotope` dataframe from the package *asbio*. Using `par(mfrow())`, create a graphical device holding three plots in a single row, i.e., the three plots will be side by side.
   (a) In the first plot, show δ^{14}C as a function of time (`decimal.date`) using a line plot. Use appropriate axis labels.
   (b) In the second plot, show CO_2 concentration as a function of time in a scatterplot.
   (c) In the third plot, show measurement precision (column four in the dataset) as a function of δ^{14}C.

6. Load the `goats` dataframe from package *asbio*.
   (a) Create a scatterplot of `NO3` as a function of `feces`.
   (b) Make a plot showing `NO3` and `organic.matter` as a simultaneous function of `feces` by adding a second y-axis.
   (c) Change symbol sizes in (a) to reflect the values in organic.matter.
   (d) Create a 3D scatterplot with `scatterplot3d::scatterplot3d`, depicting `NO3` as a function of `organic.matter` and `feces`.

Chapter 7

Grid Graphics, Including ggplot2

> *"If you think you can learn all of* R*, you are wrong. For the foreseeable future you will not even be able to keep up with the new additions."*
>
> - **Patrick Burns**, *CambR User Group Meeting, Cambridge (May 2012)*

7.1 Grid Graphics

There are a large number of auxiliary **R** packages specifically for graphics. Many of these utilize or extend the base **R** graphics approaches described in Chapter 6. Several successful newer packages, however, rely on the **R** grid graphics system (see Murrell (2019)), codified in the package *grid* (R Core Team, 2023). The grid graphics system itself provides only low-level facilities with no high-level functions to generate complete plots. Nonetheless, several successful packages have built high-level functions on grid foundations including:

- *lattice*: one of the first serious attempts to build high-level functions for grid graphs (Section 7.2).
- *gridGraphics*: converts plots drawn with the base **R** graphics, e.g., `plot()`, to identical grid output.
- *ggplot2*: a deservedly popular grid package that "… tries to take the good parts of *base* and *lattice* graphics and none of the bad" (Wickham, 2016).

Functions from *ggplot2* are the major focus of this chapter.

7.2 lattice

Among other applications, the *lattice* package (Sarkar, 2008) contains functions for implementing the trellis graphical system (Cleveland, 1993)[1], so-called because it often utilizes a rectangular array of plots resembling a garden trellis (Ryan and Nudd, 1993). Trellis plots,

[1] The **R** trellis graphics system was originally developed for **S** and **S**-Plus at Bell Labs (see Becker et al. (1996)). The *lattice* package can be considered a re-implementation of this original system.

generated from *lattice*, are an important component of several important **R** packages, including *nlme*, which allows the generation of linear and nonlinear mixed effects models (see Aho (2014), Ch 10).

Example 7.1.
A simple example of a call to trellis plotting is shown in Fig 7.1. The `datasets::Indometh` dataframe, previously used in Examples in Ch 6, records pharmacokinetics of the drug indomethacin, following intravenous injections given to human subjects. The dataframe belongs to several grouped object classes, defined in *nlme*, which have their own plotting methods (Ch 8), and are implemented through a generic call to `plot()`. We are, of course, more familiar with the use of `plot()` in base **R** graphics approaches, implemented via the *graphics* package.

```
library(nlme)
plot(Indometh)
```

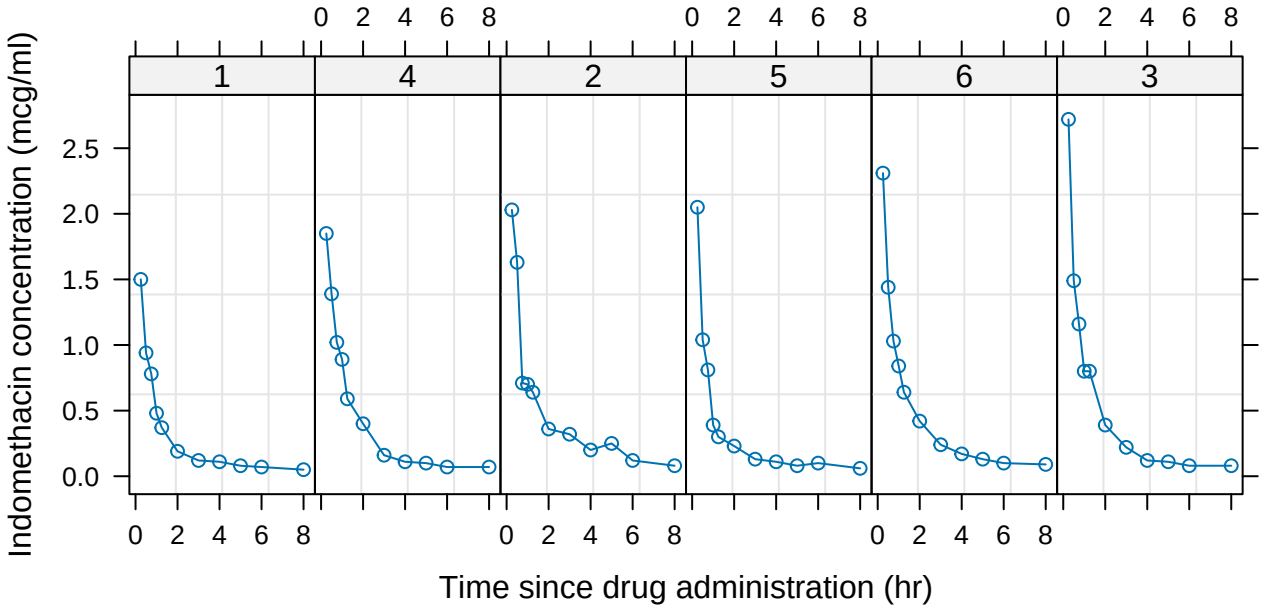


Figure 7.1: Example of a trellis plot. Indomethacin levels are tracked in six human subjects over eight hours following intravenous injections.

■

The *lattice* package contains several high level plotting functions that can be considered analogues of base **R** *graphics* functions. These include:

- `lattice::xyplot()`, which is similar to `graphics::plot()` in its default `type = "p"` mode,
- `lattice::histogram()`, which is analogous to `graphics::hist()`,
- `lattice::barchart()`, which is similar to `graphics::barplot()`,
- `lattice::levelplot()`, which is analogous to `graphics::image()`, and
- `lattice::wireframe()`, which is similar to `graphics::persp()`.

In general, trellis plots in *lattice* can be created using a conditional formula as the first argument of its functions. This will have the form `y ~ x|z`, which signifies `y` is a function of `x`, given levels in `z`.

Example 7.2.
Consider a summarization of the association of age and tobacco use (`LOW` and `HIGH`) and esophageal cancer cases using the `e.cancer` dataset (Breslow and Day, 1980) from *asbio* (Fig 7.2).

```
1  data(e.cancer)
2
3  library(tidyverse)
4  means <- e.cancer |>  # obtain means
5    group_by(age.grp, tobacco) |>
6    summarize(cases = mean(cases))
7
8  library(lattice)
9  barchart(cases ~ age.grp|tobacco, data = means, xlab = "Age",
10           ylab = "No. of cases")
```

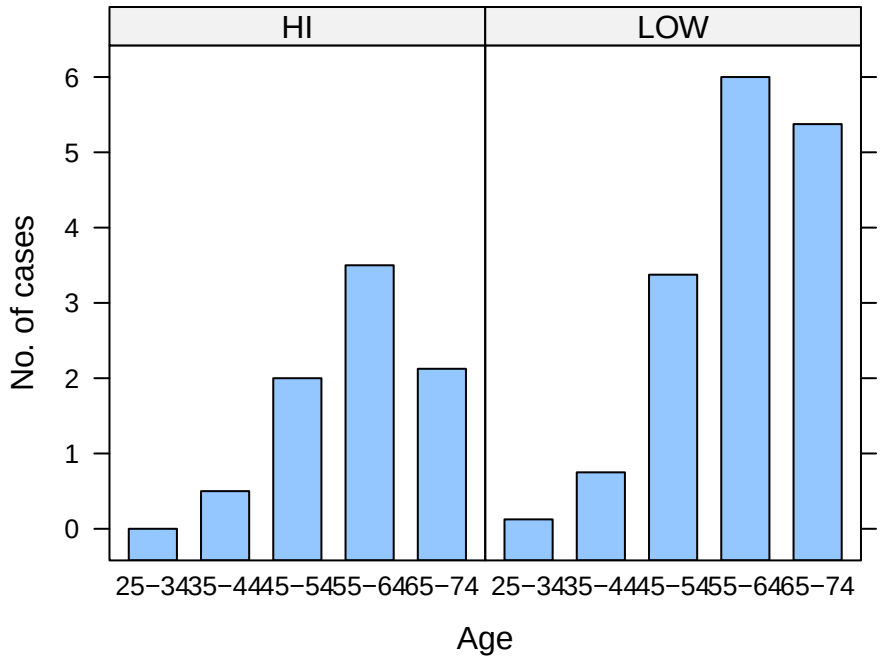


Figure 7.2: Use of `lattice::barchart()` to illustrate changes in esophogeal cancer cases with subject tobacco use and age. Bar heights are means.

Note the use of pipes and *tidyverse* functions (Ch 5) to obtain mean numbers of `cases` for combinations of levels in `age.grp` and `tobacco` (Lines 3-6). The formula`cases ~ age.grp|tobacco` (Line 8) indicates that the mean `cases` should considered as a function of levels in `age.grp`, given levels in `tobacco`.

■

Approaches in *lattice* can be used in many non-trellis applications.

Example 7.3.
Figure 7.3 provides examples of three dimensional graphics generation using the *lattice* functions `levelplot()`, `contourplot()`, and `wireframe()`. The functions are easiest to use when data are in a spatial grid format with row and column numbers defining evenly spaced intervals from some reference point, and cell responses themselves constitute "heights" for the *z* (vertical) axis. The popular `volcano` dataset, used in the figure, describes the topography of Maungawhau / Mount Eden, a scoria cone in the Mount Eden suburb of Auckland, New Zealand. In this case, rows and columns represent 10m Cartesian intervals. The first row contains elevations (in meters above sea level) for northernmost points, whereas the first column contains elevations of westernmost points. The argument `split` in `plot.trellis()` is used with both graphs in Fig 7.3 (Lines 5 and 7). It is a vector of 4 integers `c(x, y, nx, ny)` that indicate where to position the current plot at the `x`, `y` position in a regular array of `nx` by `ny` plots.

```
library(lattice)
plot(levelplot(volcano, col.regions = heat.colors, xlab = "x", ylab = "y"),
     split = c(1, 1, 1, 2), more = TRUE,
     panel.width = list(x = 5.4, units = "inches"))

plot(wireframe(volcano, panel.aspect = 0.7, zoom = 1, lwd = 0.01,
               xlab = "x", ylab = "y", zlab = "z"),
     split = c(1, 2, 1, 2), more = FALSE,
     panel.width = list(x = 5.4, units = "inches"))
```

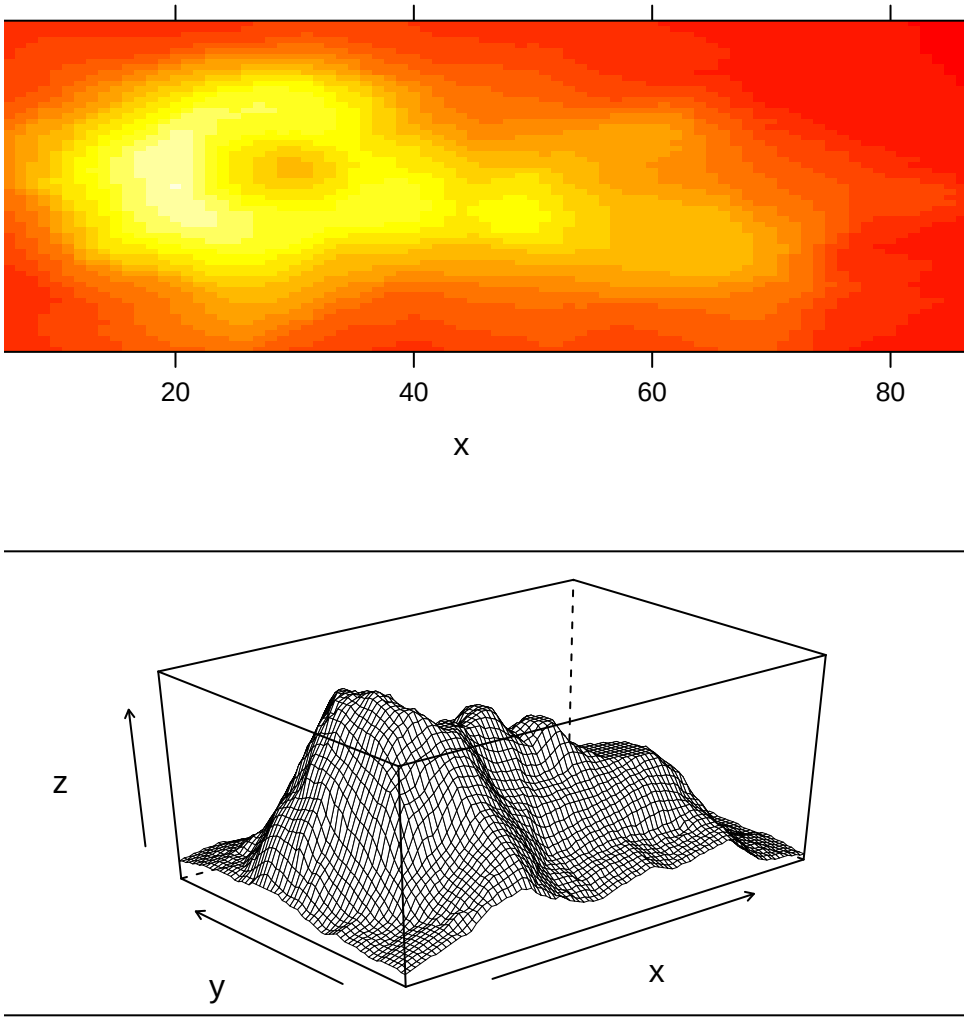


Figure 7.3: Representations of Maungawhau (Mt Eden) using *lattice* functions.

■

While *lattice* can be used to generate nice graphs, many users have found its coding requirements to be burdensome and non-intuitive. This issue, coupled with the desirable characteristics of the grid graphics system, prompted the development of the package *ggplot2*, one of the *tidyverse* collection of packages (Ch 5).

7.3 ggplot2

The *ggplot2* package (formerly *gglot*) emulates the "grammar of graphics", that underlies all statistical graphics (Wilkinson, 2012). The success of the *ggplot2* package is evident in its rich ecosystem of contributed extension packages. Detailed descriptions of the *ggplot2* package

can be found in Wickham (2010), and Wickham (2016)[2]. Helpful *ggplot2* "cheatsheets" can be found here. Like most grid implementations, *ggplot2* does not play well with base **R** graphics. In fact, *ggplot2* is based on its own unique object oriented system, the *ggproto* system [3].

7.3.1 `ggplot()`

The function `ggplot()` is used to initialize essentially all plotting procedures in *ggplot2*. There are three common approaches:

1. `ggplot(df, aes(x, y, other aesthetics))`
   - Here `df` is a tibble or dataframe. and `aes()` represents aesthetic mappings. This approach is recommended if all layers use the same data and aesthetics.
2. `ggplot(df)`
   - Here only the dataframe or tibble to be used is identified up-front. This approach is useful if graphical layers use different x and y coordinates, drawn from the same dataset, `df`.
3. `ggplot()`
   - Here a ggplot skeleton is initialized that is fleshed out as layers are added. This approach is recommended if more than more than one dataset is used in the creation of graphical layers.

One of these formats will be used as the first line of code when creating a *ggplot2* graphic. Layers will then be added representing geoms, themes, and aesthetics (see Section 7.3.2 immediately below). To clarify coding steps, this is typically done by separating layers into lines, connected with the *ggplot2* operator, `%+%`, which can be written as `+` in the context of a ggplot.

Example 7.4.
In the code below I have initiated a ggplot, under Approach 1 discussed above, using data from a dataframe called `df` that contains variables named `x` and `y`, that will define the x and y coordinates for the items in the plot. I have also added two geom layers and a theme, via the imaginary functions `geom1()`, `geom2()` and `theme1()`.

```
ggplot(df, aes(x = x, y = y)) +
  geom1() +
  geom2() +
  theme1()
```

■

[2] Although *ggplot2* is a tidyverse package, its release in 2005, greatly preceded the formal establishment of the tidyverse in 2016.

[3] For more information type: `?ggplot2::ggproto`.

7.3.2 Geoms, Aesthetics, and Themes

The *ggplot2* package facilitates the generation of overlays with *geoms*, short for "geometric objects", aesthetics, and themes. A number of *ggplot2* geom functions are shown in Table 7.1. Note that arguments in geom functions are fairly consistent. The argument `mapping` refers to aesthetic mappings, often specified with the *ggplot2* function `aes()`. A few aesthetic mapping functions are shown in Table 7.2. An explicit definition for the `stat` argument is required by several geoms, e.g., `geom_col()` and `geom_bar()`, and can take the form `stat = "identity"`, indicating that raw unsummarized data are to be plotted. The *ggplot2* package also allows specification of general graphical themes including user-defined themes, via the function `theme()`. An exhaustive list of > 90 potential `theme()` arguments can be found by typing `?theme`. Pre-defined *ggplot2* theme frameworks include `theme_gray()`, the signature *ggplot2* theme (with a grey background and white gridlines), `theme_bw()`, `theme_classic()`, `theme_dark()`, `theme_minimal()`, and many others.

Table 7.1: A few geom alternatives.

Geom function	Usage	Impt. arguments

`geom_abline()`
`geom_hline()`
`geom_vline()`
	Add reference lines	`mapping`
		`data`
		`slope`
		`intercept`

`geom_segment()`
`geom_curve()`
	Add lines and curves	`mapping`
		`data`
		`position`

`geom_area()`
`geom_ribbon()`
	Area and ribbon charts	`mapping`
		`data`
		`stat`

`geom_bar()`
`geom_col()`
	Bar charts	`mapping`
		`data`
		`stat`

`geom_bin2d()`	Heatmap of bin counts	`mapping`
		`data`
		`stat`

`geom_boxplot()`	Boxplots	`mapping`
		`data`
		`stat`

`geom_contour_filled()`	Generate 2D contours of 3D surface	`mapping`
		`data`
		`stat`

`geom_count()` `geom_sum()`	Count overlapping points	`mapping` `data`

		`stat`

`geom_crossbar()`
`geom_errorbar()`
`geom_linerange()`
`geom_pointsrange()`
	Add lines, crossbars, error bars	`mapping`
		`data`
		`stat`

`geom_density()`	Smoothed densities	`mapping`
		`data`
		`stat`

`geom_density_2d()`
`geom_density_2d_filled()`
	Contours of 2D densities	`mapping`
		`data`
		`stat`

`geom_dotplot()`	Dot plots	`mapping`
		`data`
		`position`

`geom_errorbarh()`	Horizontal error bars	`mapping`
		`data`
		`stat`

`geom_freqpoly()`
`geom_histogram()`
	Histograms	`mapping`
		`data`
		`stat`

`geom_function()`	Draw curve from function	`mapping`
		`data`
		`stat`

`geom_hex()`	Hexagonal heat map	`mapping`
		`data`
		`stat`

`geom_jitter()`	Jittered points	`mapping`
		`data`
		`stat`

`geom_text()`	Add text	`mapping`
		`data`
		`stat`

`geom_point()`	Add points	`mapping`
		`data`
		`stat`

Table 7.2: A few example of *ggplot2* aesthetic functions.

Function	Usage	Arguments
`aes()`	Aesthetics of geoms	`x`, `y`

`colour()`
`fill()`
`alpha()`
	Color related aesthetics	See `?aes_colour_fill_alpha`

`linetype()`
`size()`
`shape()`
	Line type, size, shape	See `?aes_linetype_size_shape`

`group()`	Grouping	See `?aes_group_order`

7.3.3 Boxplots

Figure 7.4 shows a boxplot of R.A. Fisher's classic potato dataset from the Rothamsted Experimental Station (Fisher and Mackenzie, 1923). There are three important coding features that should be recognized in the chunk below. First, the plot was initialized using the first approach described in the previous section: `ggplot(df, aes(x, y, other aesthetics))` (Line 2). Specifically, the dataframe to be used, `potato`, was identified, and the coordinates for the plot were defined inside `aes()`. Second, plot modifications are added with the functions `theme()` (which includes a call to the function `element_text()` to change the angle of text on the *x*-axis), `xlab()`, and finally, `geom_boxplot()` (Lines 3-5). Third, the continuation prompt, `+`, is placed at the end of lines of code to indicate that another graphical layer is being added to the plot. In *ggplot2*, `+`, is somewhat analogous to the forward pipe operator, `|>`, used in the *tiddyverse* (Ch 5). Specifically, it denotes the continuation of *ggplot2* plotting commands for a particular graphic. This continuation is broken with a line break (Line 6).

```
1 data(potato) # in asbio
2 ggplot(potato, aes(x = factor(Variety), y = Yield)) +
3   theme(axis.text.x = element_text(angle = 50, hjust = 1, vjust = 0.9)) +
4   xlab("Variety") +
5   geom_boxplot()
```

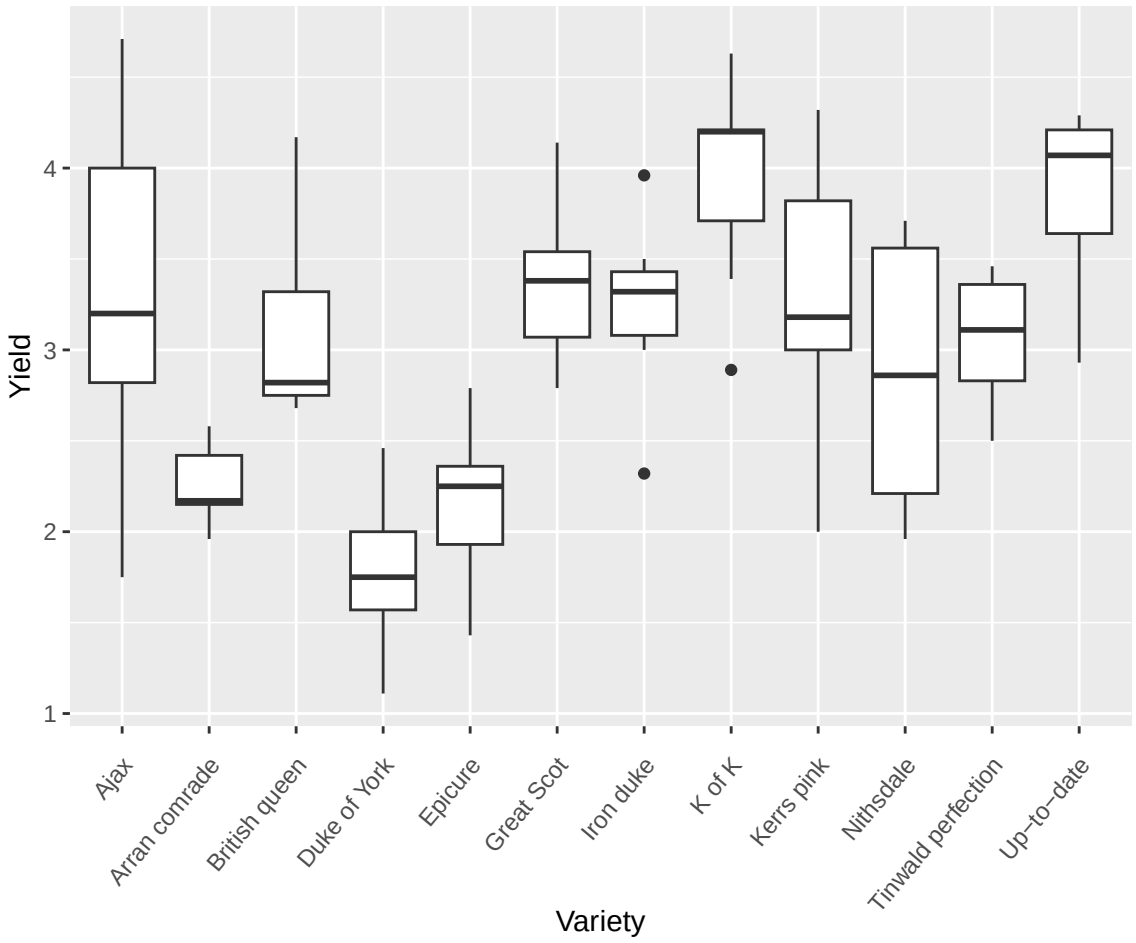


Figure 7.4: An example of a *ggplot2* boxplot. This is the signature appearance of ggplot2 graphs: a grey background and white grid lines.

7.3.4 Saving Plots

Plots can be saved using the function `ggsave()` or with a graphical device function, e.g., `pdf()`, `png()`, as described in Ch 6.

```
g <- ggplot(potato, aes(x = factor(Variety), y = Yield)) +
  theme(axis.text.x = element_text(angle = 50, hjust = 1, vjust = 0.9)) +
  xlab("Variety") +
  geom_boxplot()

pdf("potato.pdf")
print(g)
dev.off()
```

```
pdf
  2
```

Importantly, unlike `graphics::plot()`, a graphics object can be created under grid approaches. This is demonstrated on Line 1, where the ggplot is assigned to the object name `g`. In Lines 5-7, the graph is rendered, using `print.ggplot(g)` or `plot.ggplot(g)`, and compiled.

The object `g` has *ggplot2*-specific classes `"gg"` and `"ggplot"`,

```
class(g)
```

```
[1] "gg"     "ggplot"
```

and base type `list`:

```
typeof(g)
```

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

7.3.5 Line Plots

Line plots are generally rendered using the function `geom_line()`.

In Fig 7.5 we consider the Fisher's potato data under a line plot approach. This presentation allows us to consider both potato variety and fertilizer levels. Note that I distinguish categories in the variable `Fert` using `colour` and `lty` arguments in `aes()` function calls (Line 1).

```
1 ggplot(potato, aes(x = Variety, y = Yield, colour = Fert)) +
2   geom_line(aes(group = Fert, lty = Fert),
3         alpha = .7, linewidth = 1.1) +
4   theme_classic() +
5   theme(axis.text.x = element_text(angle = 50, hjust = 1, vjust = 0.9))
```

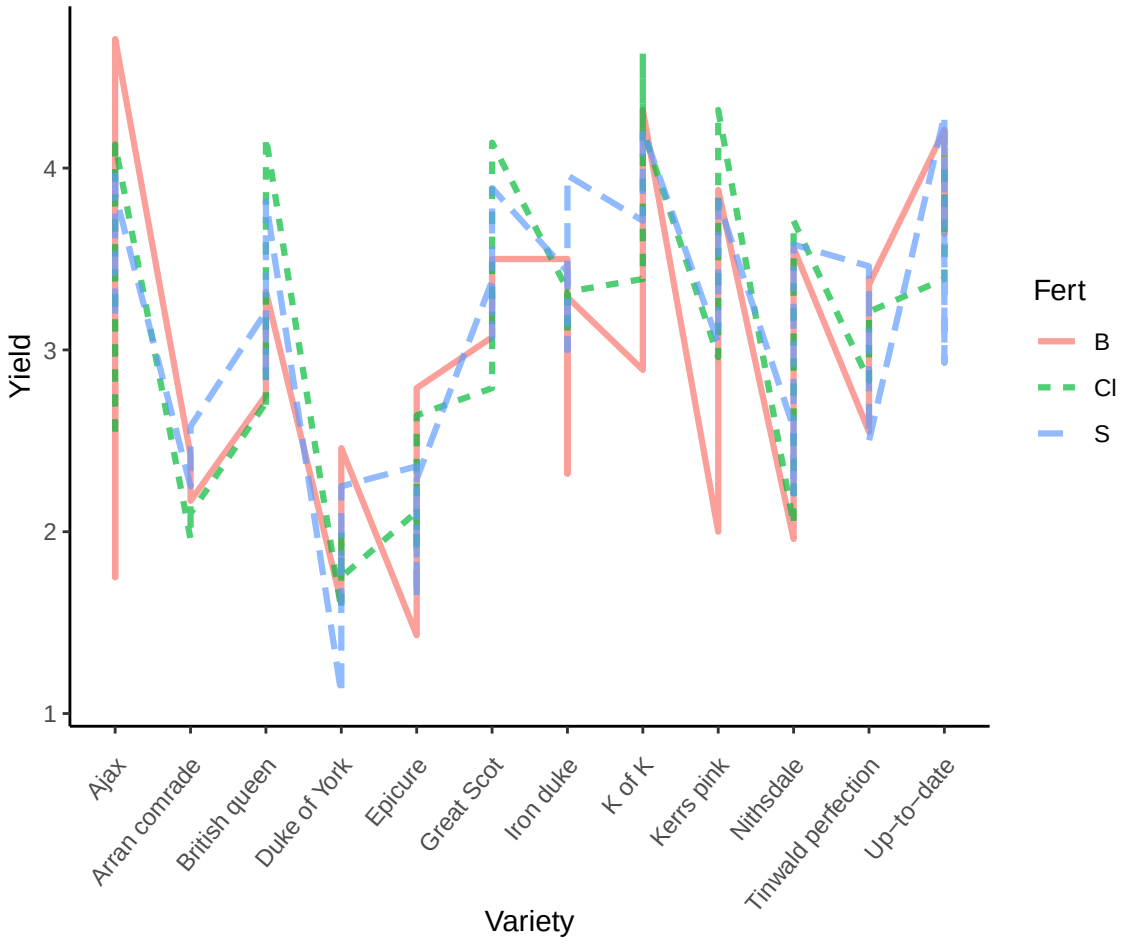


Figure 7.5: Line plot representation of the `potato` dataset.

7.3.6 Scatterplots

Here we summarize the `asbio::world.emissions` CO_2 and gross domestic product data, using a *tidyverse* approach.

```
library(asbio)
data(world.emissions)
library(dplyr)
country.data <- world.emissions |>
  filter(continent != "Redundant") |>
  group_by(country) |>
  summarize(co2 = mean(co2, na.rm = TRUE),
            gdp = mean(gdp, na.rm = TRUE))
```

I don't like the default `ggplot2` margins. Specifically, I feel that the axis label font size is too small and placed too close to the axes. Thus, prior to making a scatterplot of these variables, I make my own margin theme, as a function, that calls `theme()`.

```
margin_theme <- function(){
  theme(axis.title.x = element_text(vjust=-6, size = 12),
        axis.title.y = element_text(vjust=6, size = 12),
        axis.text = element_text(size = 10),
        plot.margin = margin(t = 7.5, r = 7.5, b = 22, l = 22))
}
```

We can call this custom theme within ggplot code (Fig 7.6).

```
g <- ggplot(country.data, aes(x = gdp, y = co2)) +
  ylab(expression
       (paste(CO[2], " Emissions (metric tons x ", 10^6, ")"))) +
  xlab("GDP (international dollars)") +
  geom_point(size = 2) +
  scale_y_continuous(trans= "log10") +
  scale_x_continuous(trans= "log10") +
  theme_classic() +
  margin_theme()
g
```

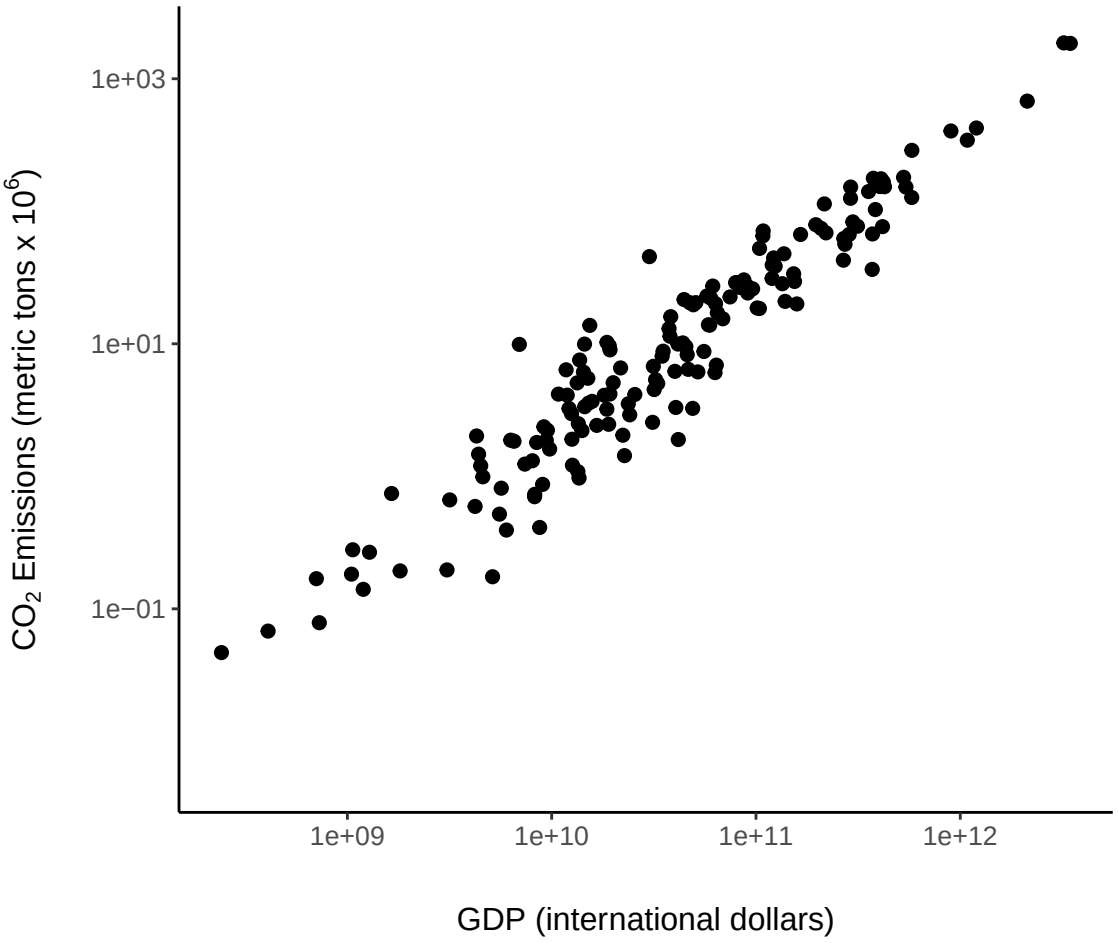


Figure 7.6: An example of a *ggplot2* scatterplot using the `world.emissions` dataset.

I also called the built-in *ggplot2* theme `theme_classic()` to generate an uncluttered graph with no grid lines.

Several other code steps are worth mentioning in Fig 7.6. First, note the use of plotmath code using calls to `expression()` in `xlab()` and `ylab()`. As an alternative, I could have used `labs(x,y)` where the arguments `x` and `y` would contain code for `xlab()` and `ylab()`. Second, $\log_{10}$ transformations were applied to both axes using the *ggplot2* functions `scale_x_continuous()` and `scale_y_continuous()`. As an alternative, I could have used the functions `scale_x_log10()` and `scale_y_log10()`.

We can also require specific tick locations using `scale_y_continuous` (Fig 7.7).

```
g + scale_y_continuous(breaks = c(1, 50, 150, 500, 750, 1600),
                       trans= "log10")
```

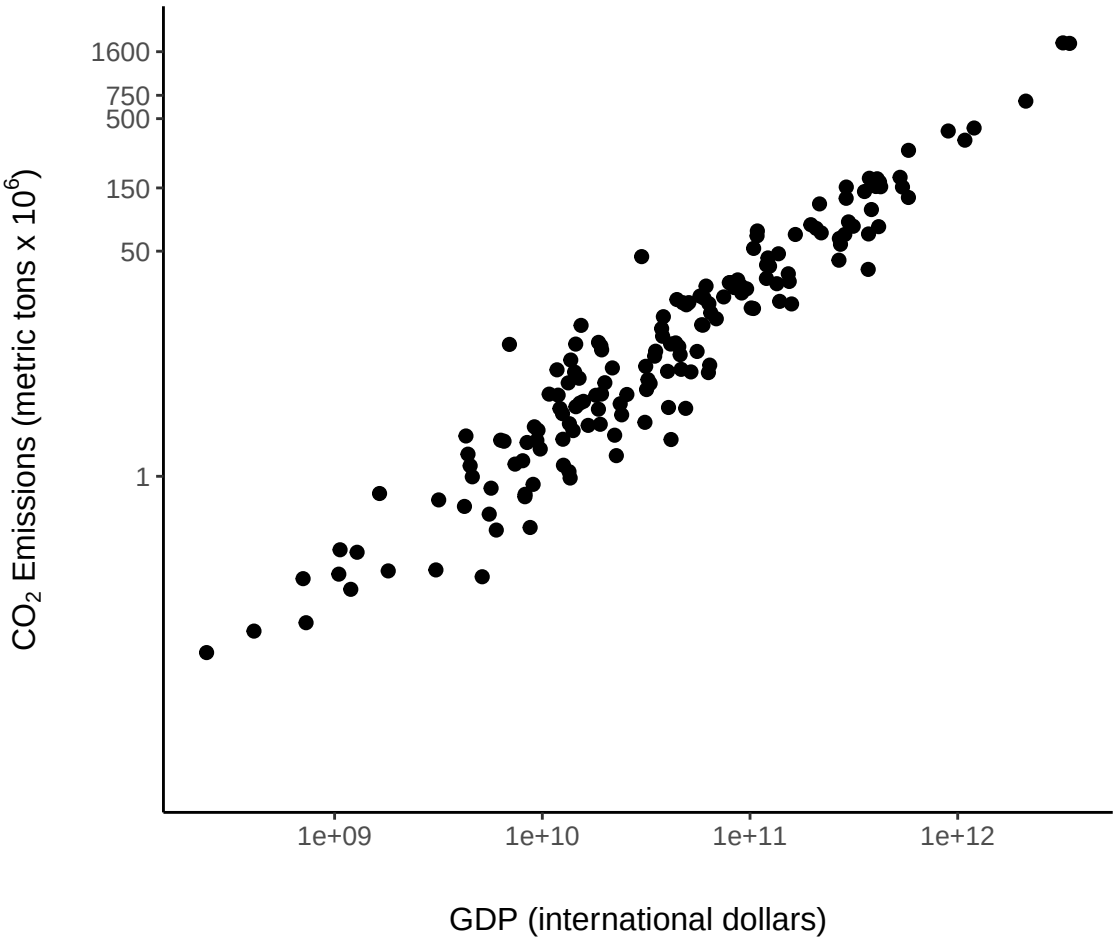


Figure 7.7: Custom tick mark locations overlaid on he y-axis of Fig 7.6.

7.3.7 Transformations

Other than $\log_{10}$ transformations (Fig 7.6), several other graphical transformation can be readily applied in the `transform` function of `scale_x_continuous()` and `scale_y_continuous()`, including `"asn"` (arcsine), `"atanh"` (the inverse hyperbolic tangent), `"boxcox"` (i.e., the optimal power transform for the response variable in a linear model (see Aho (2014))), `"log"` ($\log_e$ transform), `"log1p"` ($\log_e$ transform, following the addition of 1 to prevent undefined logarithms of zeroes), `"log2"` ($\log_2$ transform), `"logit"` (i.e., the log odds for a probability), `"pseudo_log"` ($\log_e$ transform, `NA`s resulting from undefined logarithms of zeroes, are given the value zero), `"probit"` (i.e. the inverse CDF for a standard normal distribution), `"exp"`, `"modulus"`, `"reciprocal"`, `"reverse"`, `"sqrt"`, `"date"`, `"hms"`, and `"time"`.

7.3.8 Adding Model Fits

It is straightforward to add fits from statistical models to a ggplot object, for instance the object g created in Fig 7.6. Easily accessed modeling approaches include conventional general linear

models and locally fitted models, like Generalized Additive Models (GAMs) and locally weighted scatterplot smoothers (LOWESS), that allow the association between *x* and *y* to "speak for itself" without the assumption of underlying global linear association (Aho, 2014). By default, `geom_smooth()` provides a LOWESS fit using the function `loess()` from the **R** distribution *stats* package. The code `geom_smooth(method = "lm")` fits a general linear model, in this case, a simple linear regression (Fig 7.8). By default, error polygons are included with fits that represent 95% confidence intervals for the true fitted value. These can be turned off by specifying `geom_smooth(se = FALSE)`.

```
g + geom_smooth(method = "lm") # call to lm fit
```

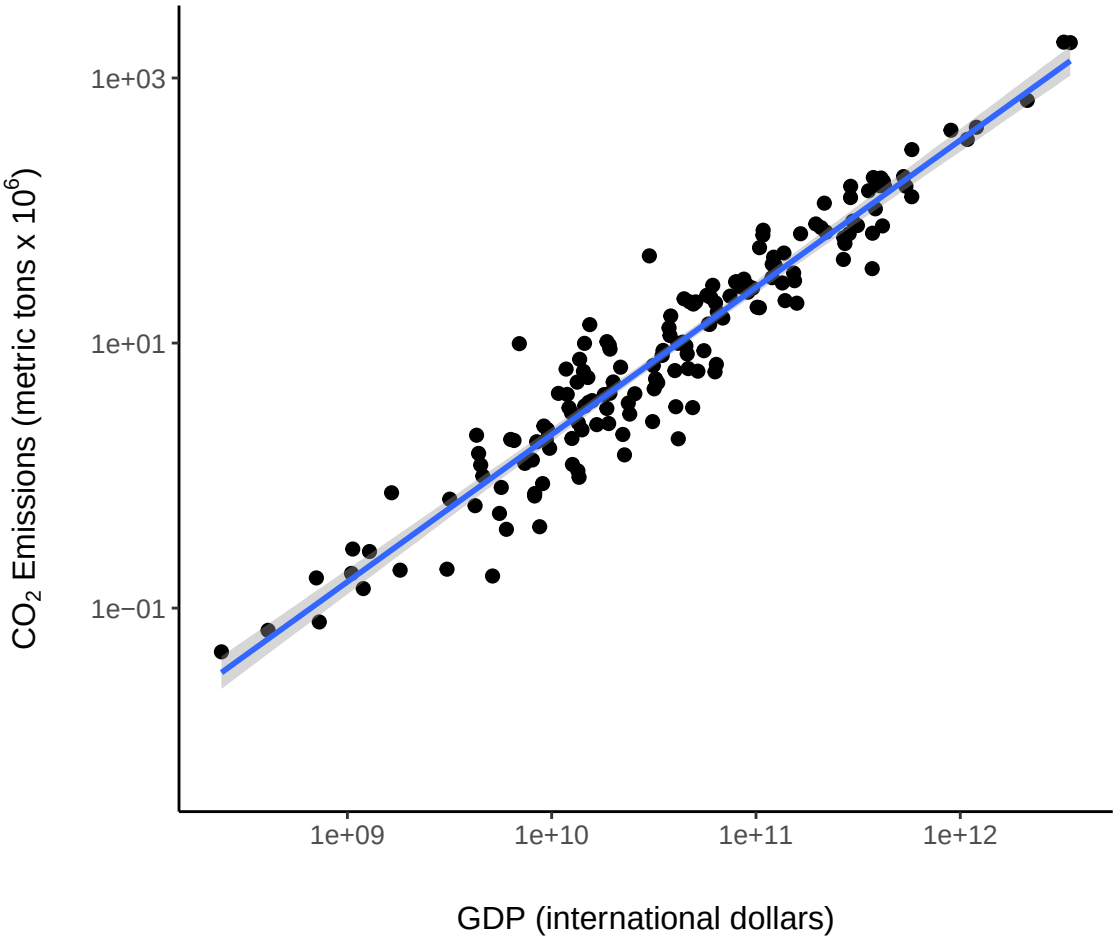


Figure 7.8: A regression model overlaid on Fig reffig:gscat1.

7.3.9 Annotations in Graphs

The *ggplot2* package has nice functions for graph annotation. In Fig 7.9 we use the function `geom_label()` to label countries. The arguments `nudge_y` and `nudge_x` allow adjustments to label locations.

```
sub <- country.data |>
  filter(country %in% c("Canada", "Finland",
"Japan", "Kenya", "United States"))

g + geom_point(size = 3, shape = 1, data = sub, col = "orange") +
  geom_label(aes(label = country), data = sub, nudge_y = .25,
             nudge_x = -.25, alpha = .9, colour  = "orange")
```

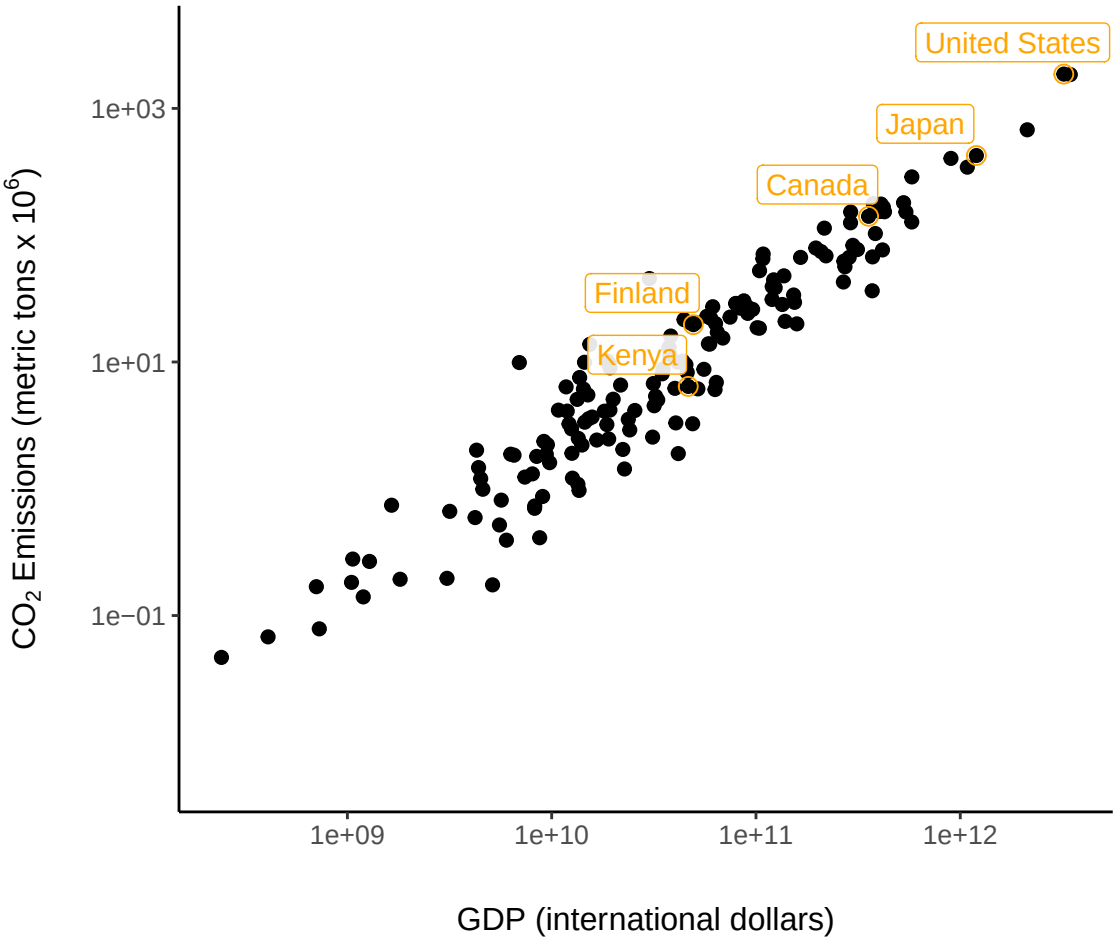


Figure 7.9: Country annotations added to Fig 7.6.

The *ggpmisc* package allows annotation of statistical models in a ggplot. This is accomplished using the function `ggpmisc::stat_poly_eq()` which fits a model using`stats::lm()`, computes model quantities and prepares tidy text summaries of the model including the model equation, test statistic values and p-values. Computed terms are called using the *ggplot2* function `after_stat()`, which delays aesthetic mapping in `aes()` until after statistic calculation.

In Fig 7.10, the equation for the `world.emissions` regression model is placed in the figure with `after_stat(eq.label)`, and the adjusted R^2 (Aho, 2014) is placed using

`after_stat(adj.rr.label)`. A complete list of available computed terms, including `eq.label` and `adj.rr.label`, is given in the documentation for `stat_poly_eq()`.

```
library(ggpmisc)

g + geom_smooth(method = "lm") +
stat_poly_eq(aes(label = paste(after_stat(eq.label),
                               after_stat(adj.rr.label),
                               sep = "*\", \"*")))
```

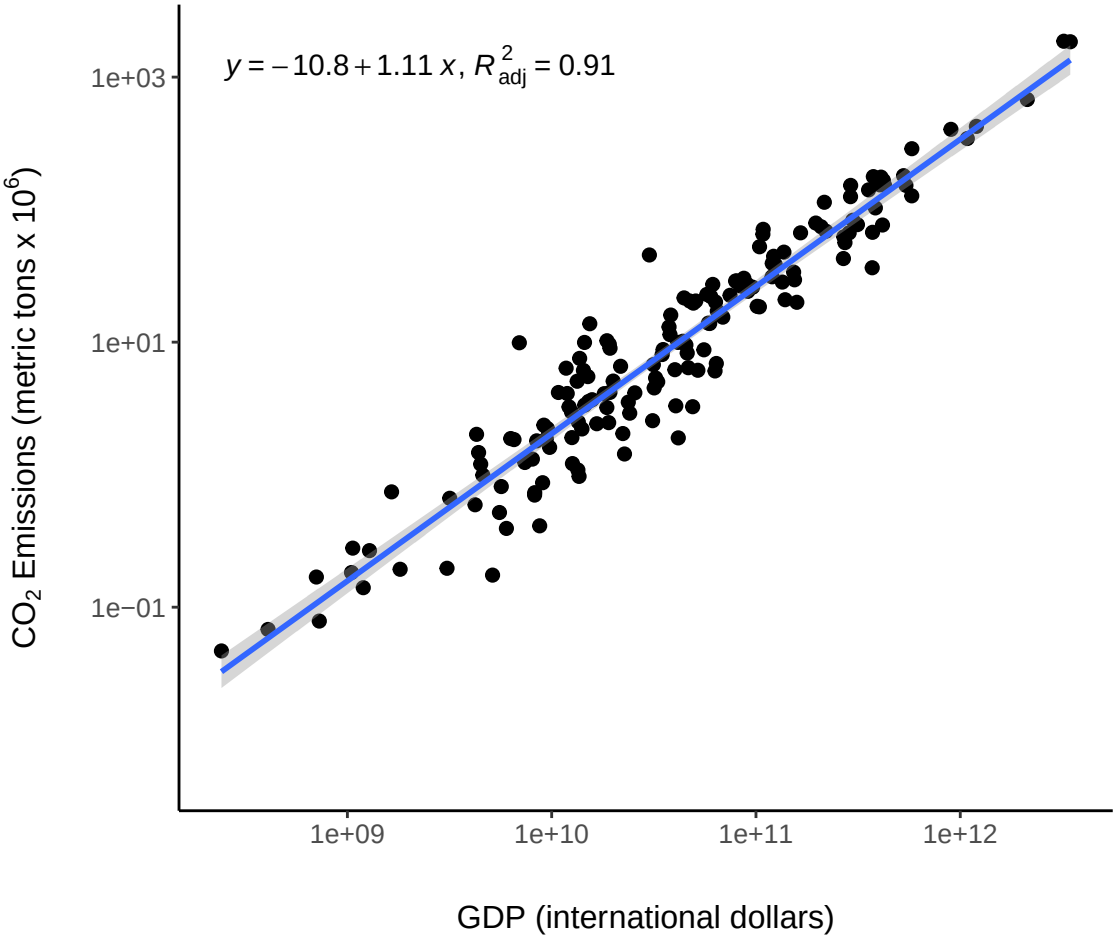


Figure 7.10: Regression model summaries overlaid on Fig 7.6.

A potential criticism of *ggplot2* is that its graphical rendering approaches are not readily accessible (as code or output), and statistical summaries are often not adequately or clearly described (in documentation or output). For instance, when using `geom_smooth()` it is unclear which default smoothing parameters are actually being used, although these can be set within `geom_smooth()`. The function `ggplot2::ggplot_build()` provides underlying plotting details. For example

```
head(ggplot_build(g)$data[[1]])
```

```
          y      x PANEL group shape colour size fill alpha stroke
1  0.40732 10.494     1    -1    19  black    2   NA    NA    0.5
2  0.51149 10.085     1    -1    19  black    2   NA    NA    0.5
3  1.63089 11.428     1    -1    19  black    2   NA    NA    0.5
4 -0.31447     NA     1    -1    19  black    2   NA    NA    0.5
5  1.00675 10.642     1    -1    19  black    2   NA    NA    0.5
6 -0.95782     NA     1    -1    19  black    2   NA    NA    0.5
```

The *gginnards* package can be used to generate accessible ggplot analytical information. This can be done by calling `gginnards::geom_debug()` within `ggpmisc::stat_poly_eq()`.

As an example, recall that a $\log_{10} - \log_{10}$ transformation was used to generate the scatterplot object, g, used to project Fig 7.6. I can summarize the linear model, overlaid in Fig 7.8, with the code:

```
library(gginnards)
g + stat_poly_eq(formula = y ~ x, geom = "debug",
                 output.type = "numeric",
                 summary.fun = function(x) x[["coef.ls"]])
```

```
[1] "PANEL 1; group(s) -1; 'draw_function()' input 'data' (head):"
  npcx npcy label
1   NA   NA
                                                                                                  coef.ls
1 -1.0801e+01, 1.1109e+00, 2.8430e-01, 2.6794e-02, -3.7990e+01, 4.1462e+01, 1.3821e-82, 3.6215e-88
            coefs r.squared rr.confint.level rr.confint.low
1 -10.8005, 1.1109   0.91388             0.95        0.89056
  rr.confint.high adj.r.squared f.value f.df1 f.df2    p.value    AIC
1         0.92912       0.91335  1719.1     1   162 3.6215e-88 30.353
     BIC   n rr.label b_0.constant     b_0    b_1 fm.method fm.class
1 39.652 164                FALSE -10.801 1.1109     lm:qr       lm
  fm.formula fm.formula.chr      x      y PANEL group orientation
1      y ~ x          y ~ x 8.3836 3.2706     1    -1           x
```

This is in accordance with the linear model $\log_{10}(CO_2) = -10.800518 + 1.110939 \log_{10}(GDP)$ obtained using the base function `lm()`.

```
model <- lm(log(co2, base = 10) ~ log(gdp, base = 10), data = country.data)
coef(model)
```

```
        (Intercept) log(gdp, base = 10)
           -10.8005                1.1109
```

7.3.10 Secondary Axes

Secondary axes can be difficult to implement in *ggplot2* because they require user specification of a one-to-one transformation, and the correct back-transformation. A simple possibility is a

linear transformation[4]. Let y represent raw, untransformed data to be plotted on a primary axis. A linear transform of y, to be plotted on the corresponding secondary axis, will have the form:

$$y' = a + b \cdot y \tag{7.1}$$

with back-transformation:

$$\frac{y' - a}{b} = y. \tag{7.2}$$

where y' denotes the transformed data, and a and b are user-defined constants.

When choosing a linear transform, it is helpful to remember that including a multiplier (allowing b to equal a number other than 1), will increase the data variance (spread) by a factor of b^2, and that adding a constant (letting a be a number other than 0), will cause the data mean to shift by a units, but will not affect the data variance (see Aho (2014)).

Example 7.5.
To demonstrate secondary axes, we will examine two datasets published by Rubino et al. (2013) concerning CO_2 and $\delta^{13}C$ trapped in Antarctic ice layers. We wish to simultaneously plot CO_2 and $\delta^{13}C$ of as a function of the age of the depositional layer. We will use the primary (left-hand) vertical axis to plot CO_2 and the use the right hand axis for $\delta^{13}C$. We first create a composite dataset for years in which both CO_2 and $\delta^{13}C$ were measured.

```
data(Rabino_CO2); data(Rabino_del13C)
# Match 1st dataset with 2nd
w <- which(Rabino_CO2$effective.age %in% Rabino_del13C$effective.age)
R.C <- Rabino_CO2[w,]
# match 2nd dataset with 1st
w <- which(Rabino_del13C$effective.age %in% R.C$effective.age)
R.d <- Rabino_del13C[w,]
data.C <- data.frame(CO2 = tapply(R.C$CO2, R.C$effective.age, mean),
                     d13C = tapply(R.d$d13C.CO2, R.d$effective.age, mean),
                     year = as.numeric(levels(factor(R.d$effective.age))))
```

For the years (ice depths) under consideration, CO_2 levels vary between approximately 271 and 368 ppm. A range of around 100 ppm.

```
data.C |>
  reframe(range_ppm = range(CO2, na.rm = T))
```

```
  range_ppm
1    277.16
```

[4]non-linear transformations include log, trigonometric, and exponential transformations.

```
2     368.02
```

Experimentation with linear transformations, Eq (7.1), reveals that a similar numeric range can be generated for δ^{13}C with the transformation: $y' = 56 \cdot y + 729$.

```
data.C |>
  reframe(range_ppm = range((d13C * 56) + 729, na.rm = T))
```

```
  range_ppm
1    277.11
2    373.34
```

Thus, we create:

```
data.C$td13C <- data.C$d13C * 56 + 730
```

and use it in the ggplot code below. A scatterplot of the data is shown in Fig 7.11.

```
ggplot(data.C, aes(x = year, y = CO2)) +
  geom_point(colour = "blue", size = 2.7, alpha = 0.2) +
  theme_classic() +
  margin_theme() +
  ylab(expression(paste("C",O[2], " (ppm)"))) +
  geom_point(data = data.C, aes(x = year, y = td13C), colour = "red",
             size = 2.7, alpha = 0.2) +
  scale_y_continuous(sec.axis =
                       sec_axis(~ (. - 730)/56,
                                name = expression(paste(delta^13,
                                                        "C (\u2030)")))) +
theme(axis.text.y.right = element_text(colour = "red")) +
theme(axis.text.y.left = element_text(colour = "blue")) +
theme(axis.title.y.right = element_text(colour = "red")) +
theme(axis.title.y.left = element_text(colour = "blue")) +
theme(axis.line.y.right = element_line(colour = "red")) +
theme(axis.line.y.left = element_line(colour = "blue")) +
theme(axis.ticks.y.right = element_line(colour = "red")) +
xlab("Year")
```

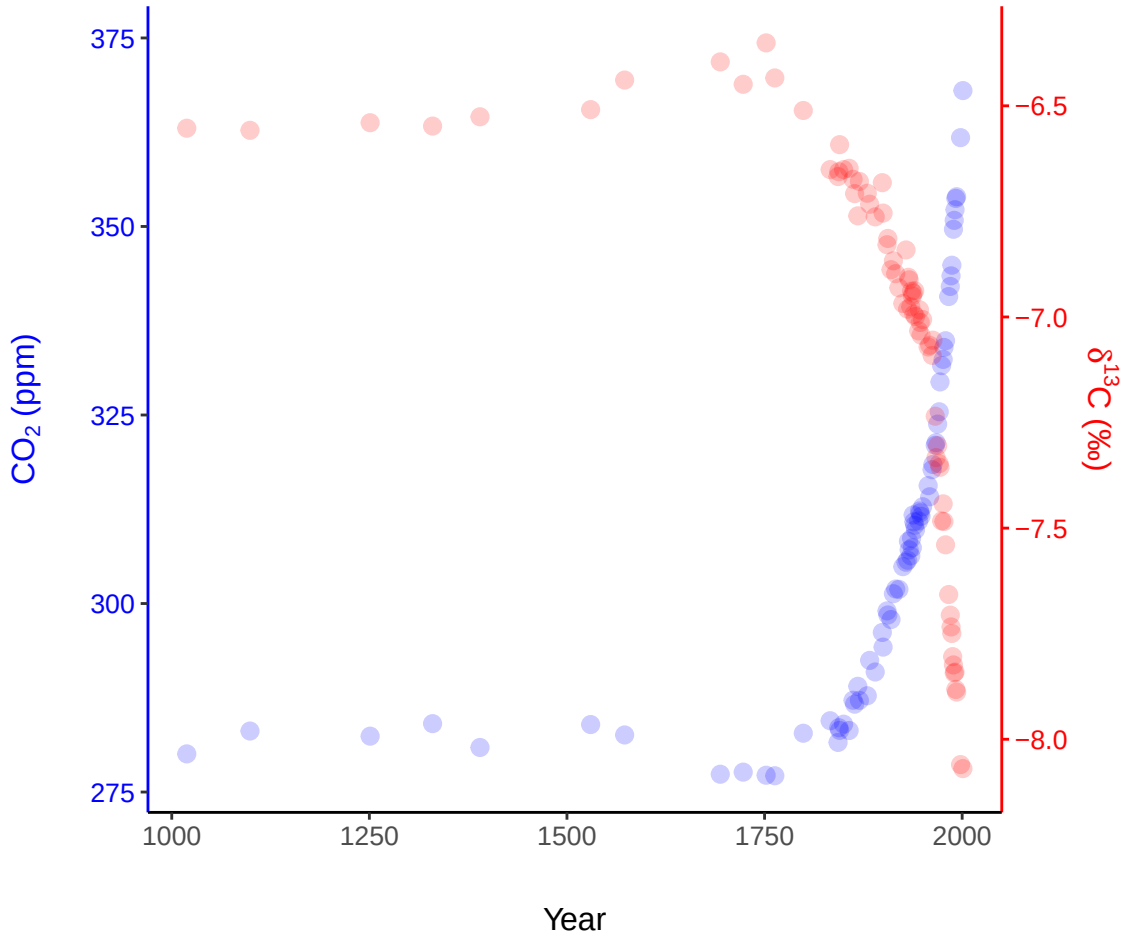


Figure 7.11: A graphical representation of data published by Rubino et al. (2013), using two vertical axes.

There were two vital steps for creating the secondary axis.

- First, as a preliminary step, we transformed the raw δ^{13}C data to allow plotting δ^{13}C points in the range of CO_2 observations (Line 1). The result is the object `data.C$td13C`.
- Second, in Lines 8-11 we scale the secondary axis based on a back-transformation of the transformed data (Eq (7.2)). That is, we solve for y in $y' = 56 \cdot y + 730$ and find $y = (y' - 730)/56$. This is what underlies the code on Line 9: `sec.axis = sec_axis(~ (. - 730)/56,`. Note that axis components were painstakingly colored using `ggplot2::theme()` (Lines 12-19).

■

7.3.11 Defining Graphical Features using Vectors

As we have already seen, it is straightforward to define figure plotting characteristics (symbols, symbol sizes, colors, line types, etc.) using relevant data.

Example 7.6.
In Fig 7.12 we change symbols and colors for a representation of the `asbio::fly.sex` dataset based on experimental treatments:

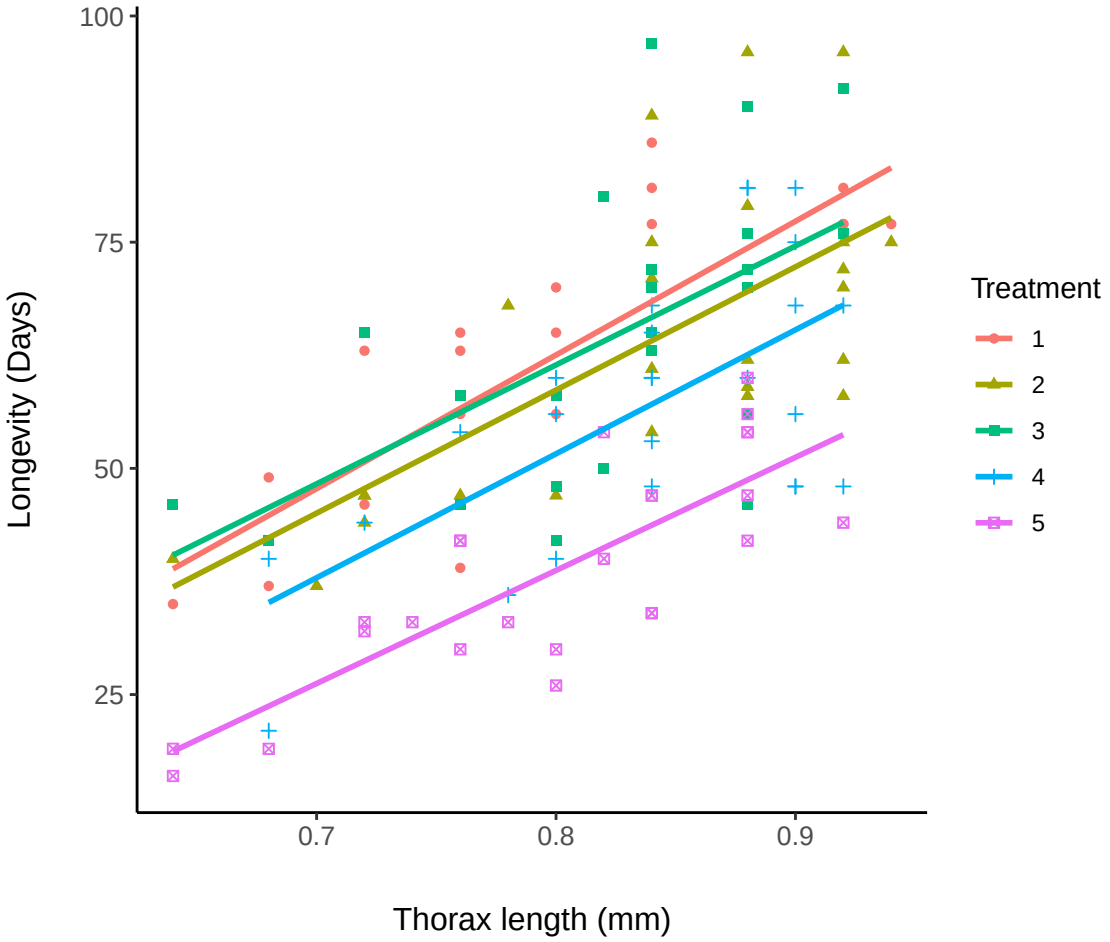


Figure 7.12: A representation of the `fly.sex` dataset.

Note that the linear fits in Fig 7.12 are actually for separate regression models, `longevity ~ thorax`, for each level in `fly.sex$Treatment`. They are *not* from the single ANCOVA model: `lm(longevity ~ thorax * Treatment)`, although this is not clear at all from the `ggplot` graph. It is, however, revealed from:

```
g1 + stat_poly_eq(formula = y ~ x, geom = "debug",
                  output.type = "numeric",
                  summary.fun = function(x) x[["coef.ls"]])
```

```
`geom_smooth()` using formula = 'y ~ x'
```

```
[1] "PANEL 1; group(s) 1, 2, 3, 4, 5; 'draw_function()' input 'data' (head):"
  npcx npcy label
1   NA   NA
2   NA   NA
3   NA   NA
```

```
4   NA   NA
5   NA   NA
                                                                                                  coef.ls
1 -5.5699e+01, 1.4779e+02, 1.6834e+01, 2.0795e+01, -3.3087e+00, 7.1071e+00, 3.0654e-03, 3.0692e-07
2 -5.0242e+01, 1.3613e+02, 2.4519e+01, 2.9186e+01, -2.0491e+00, 4.6640e+00, 5.2023e-02, 1.0753e-04
3    -43.7248157, 131.4496314, 31.3250601, 37.8123482, -1.3958414, 3.4763678, 0.1760921, 0.0020423
4 -5.7992e+01, 1.3700e+02, 2.8260e+01, 3.3625e+01, -2.0521e+00, 4.0744e+00, 5.1714e-02, 4.6763e-04
5 -6.1280e+01, 1.2500e+02, 1.5225e+01, 1.8944e+01, -4.0250e+00, 6.5983e+00, 5.2871e-04, 9.8692e-07
             coefs r.squared rr.confint.level rr.confint.low
1 -55.699, 147.790   0.68712             0.95       0.418224
2 -50.242, 136.127   0.48607             0.95       0.169020
3 -43.725, 131.450   0.34445             0.95       0.058492
4 -57.992, 137.001   0.41920             0.95       0.110089
5   -61.28, 125.00   0.65433             0.95       0.370279
  rr.confint.high adj.r.squared f.value f.df1 f.df2    p.value    AIC
1         0.79674       0.67352  50.511     1    23 3.0692e-07 180.72
2         0.66239       0.46373  21.753     1    23 1.0753e-04 199.31
3         0.56048       0.31595  12.085     1    23 2.0423e-03 202.90
4         0.61539       0.39395  16.600     1    23 4.6763e-04 197.51
5         0.77529       0.63930  43.538     1    23 9.8692e-07 174.04
     BIC  n rr.label b_0.constant     b_0    b_1 fm.method fm.class
1 184.38 25                FALSE -55.699 147.79     lm:qr       lm
2 202.96 25                FALSE -50.242 136.13     lm:qr       lm
3 206.56 25                FALSE -43.725 131.45     lm:qr       lm
4 201.16 25                FALSE -57.992 137.00     lm:qr       lm
5 177.69 25                FALSE -61.280 125.00     lm:qr       lm
  fm.formula fm.formula.chr    x     y group PANEL orientation
1      y ~ x          y ~ x 0.64 97.00     1     1           x
2      y ~ x          y ~ x 0.64 92.95     2     1           x
3      y ~ x          y ~ x 0.64 88.90     3     1           x
4      y ~ x          y ~ x 0.64 84.85     4     1           x
5      y ~ x          y ~ x 0.64 80.80     5     1           x
```

■

7.3.12 Modifying Legends

Note that a legend was created for Fig 7.12 because of designation of groups in the initial aesthetics. Legend characteristics generally need to be modified using `theme()`. For instance, to change the legend location from the right-hand side of the plot to the the left-hand side, I could use:

```
g1 + theme(legend.position = "left")
```

7.3.13 Multiple plots

We can place multiple ggplots into a single graphics device using several approaches. I consider two here: 1) facet functions from the *ggplot2* package, and 2) ggplot extension functions from the package *cowplot*.

7.3.13.1 Faceting

The functions `facet_wrap()` and `facet_grid()` can be used to generate a sequence of plot panels.

Example 7.7.
I will modify Fig 7.12 to demonstrate the use of `facet_wrap()`.

```
1 g1 <- ggplot(fly.sex, aes(y = longevity, x = thorax,
2                           group = Treatment)) +
3   geom_point(aes(colour = Treatment, shape = Treatment)) +
4   facet_wrap(vars(Treatment)) +
5   theme_classic() +
6   margin_theme() +
7   geom_smooth(method = "lm", se = F, aes(colour = Treatment)) +
8   labs(x = "Thorax length (mm)", y = "Longevity (Days)")
9 g1
```

```
`geom_smooth()` using formula = 'y ~ x'
```

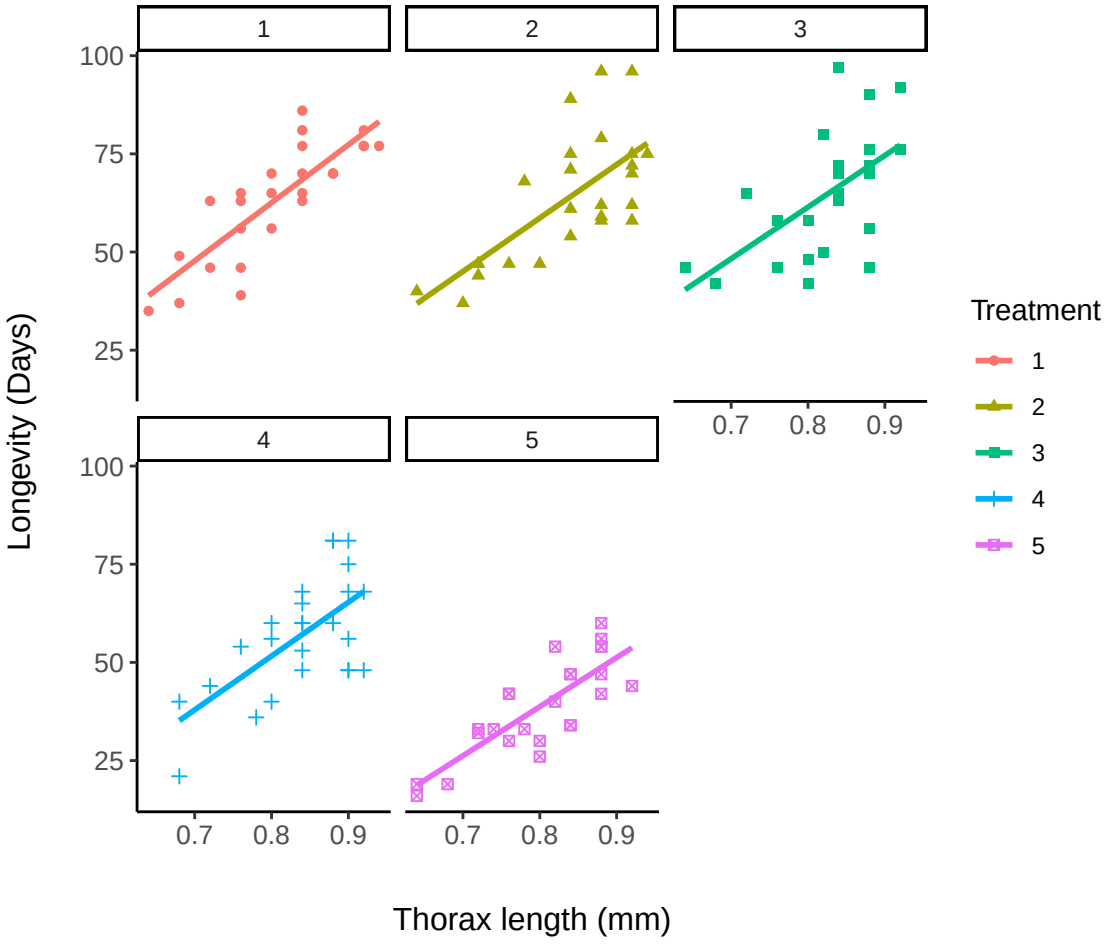


Figure 7.13: Demonstration of facet wrapping using the `fly.sex` dataset.

On Line 4 I specify that different panels should be created for each treatment level using: `facet_wrap(vars(Treatment))`. This could also be accomplished using:

`facet_wrap(~Treatment)`.

■

7.3.13.2 cowplot functions

Multiple plots can also be assembled into a single graphical entity using functions from the *cowplot* package. This requires creating separate plot objects and concatenating them in `cowplot::plot_grid()`.

Example 7.8.
Fig 7.14 shows summaries of US per capita CO_2 emissions and GDP since the start of the industrial revolution with two plots.

```
library(cowplot)
US <- world.emissions |>
  filter(country == "United States")

g2 <- ggplot(US) +
  geom_line(aes(year, co2/population), col = "dark red") +
  theme_classic() + margin_theme() +
  theme(axis.text.x = element_text(angle = 50, hjust = 1, vjust = 0.9)) +
  labs(x = "Year",
       y = expression(paste("Per capita ", CO[2],
                            " emissions (tonnes x ", 10^6, ")")))

g3 <- ggplot(US) +
  geom_line(aes(year, gdp/population), col = "blue") +
  theme_classic() + margin_theme() +
  theme(axis.text.x = element_text(angle = 50, hjust = 1, vjust = 0.9)) +
  labs(x = "Year", y = expression(paste("Per capita GDP")))

plot_grid(g2, g3)
```

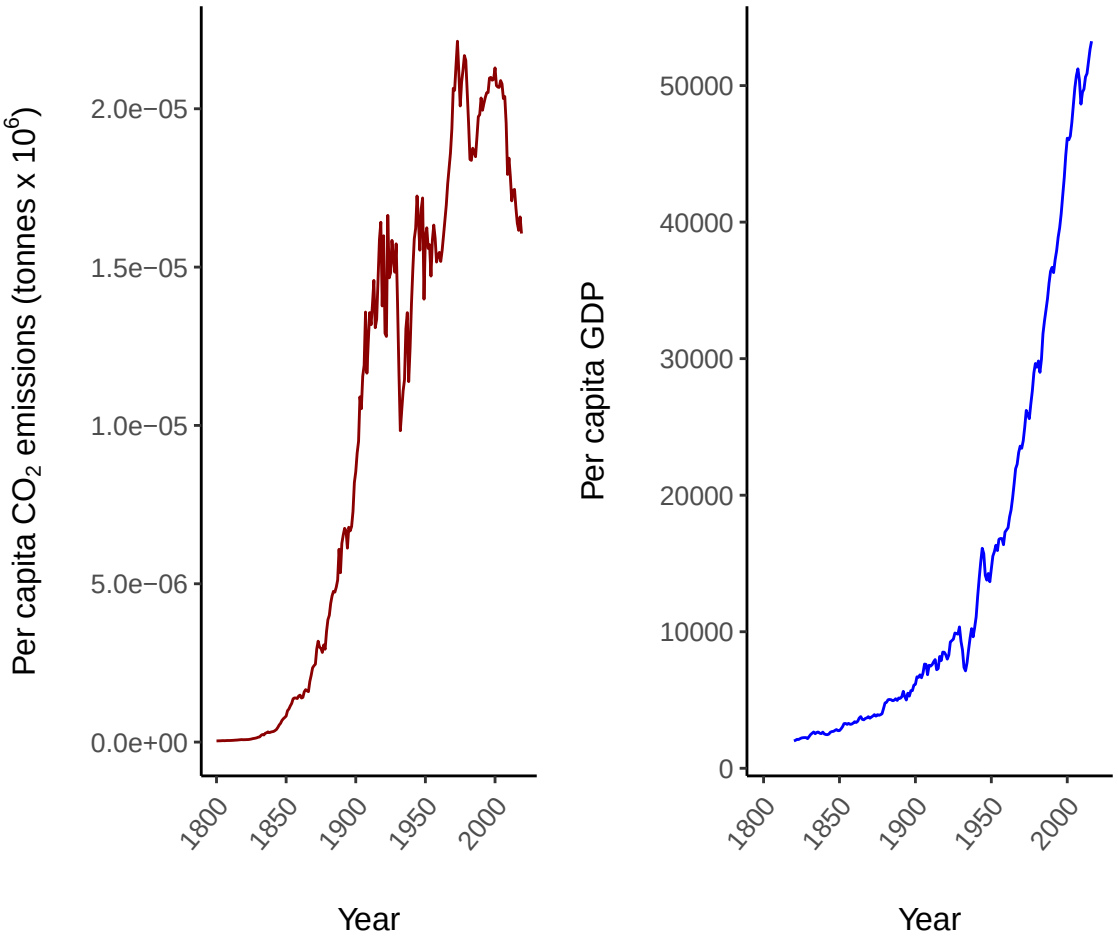


Figure 7.14: Two plots depicting US per capita trends in CO_2 emissions and GDP.

The function `plot_grid` is used on Line 17 to conjoin the ggplot objects `g2` and `g3`.

■

7.3.14 Univariate Distributional Summaries

A number of *ggplot2* functions can be used to graphically summarize distributions of variables. These include `geom_hist()` for histograms, `geom_area()` for area plots, `geom_freq()` for frequency plots, `geom_dotplot()` for dot plots, and `geom_density()` for density plots.

Example 7.9.
Fig 7.15 provides a multi-plot distributional summary of the US CO_2 data using a histogram, an area plot, and a frequency plot. These are created as separate ggplot objects.

```
1 xlab <- expression(paste(CO[2], " Emissions (metric tons x ", 10^6, ")"))
2 Years <- factor(c(rep("1800-1854", 55), rep("1854-1908", 55),
3           rep("1908-1962", 55), rep("1962-2019", 55)))
```

```
4
5  margin_theme2 <- function(){
6    theme(axis.title.y = element_text(hjust=0.6, vjust = 2.8, size = 10),
7    plot.margin = margin(t = 7.5, r = 7.5, b = 7.5, l = 15))
8  }
9
10 histogram <- ggplot(US, aes(co2, fill = Years)) +
11   geom_histogram(binwidth = 500) +
12   theme_classic() +
13   scale_fill_brewer(palette = "Blues") +
14   xlab(xlab) + ylab("Frequency") +
15   margin_theme()
16
17 areaplot <- ggplot(US, aes(co2, fill = Years)) +
18   geom_area(stat="bin") +
19   theme_classic() +
20   scale_fill_brewer(palette = "Spectral") +
21   xlab("") + ylab("Frequency") +
22   margin_theme2()
23
24 freqplot <- ggplot(US, aes(co2, colour = Years)) +
25   geom_freqpoly() +
26   theme_classic() +
27   scale_fill_brewer(palette = "Spectral") +
28   xlab("") + ylab("") +
29   margin_theme2()
```

The histogram, area plot, and frequency plot are created on Lines 10-15, 17-22, 24-29, respectively. Note the use of a second margin theme (Lines 5-8) and the use of `ggplot2::scale_fill_brewer()` to define specific *RColorBrewer* color palettes.

The plots are conjoined, with the area plot and frequency plot splitting the first row, and the histogram occupying the entire second row of the graphical device using `cowplot::plot_grid()`.

```
title <- ggdraw() + draw_label(expression(paste(CO[2] , " in the US")),
                               fontface='bold')
top_row <- plot_grid(areaplot, freqplot, ncol = 2, labels = "AUTO")

plot_grid(title, top_row, histogram, rel_heights = c(0.2, 1, 1.2),
          hjust = c(0,0,-0.6), nrow = 3, labels = c("", "", "C"))
```

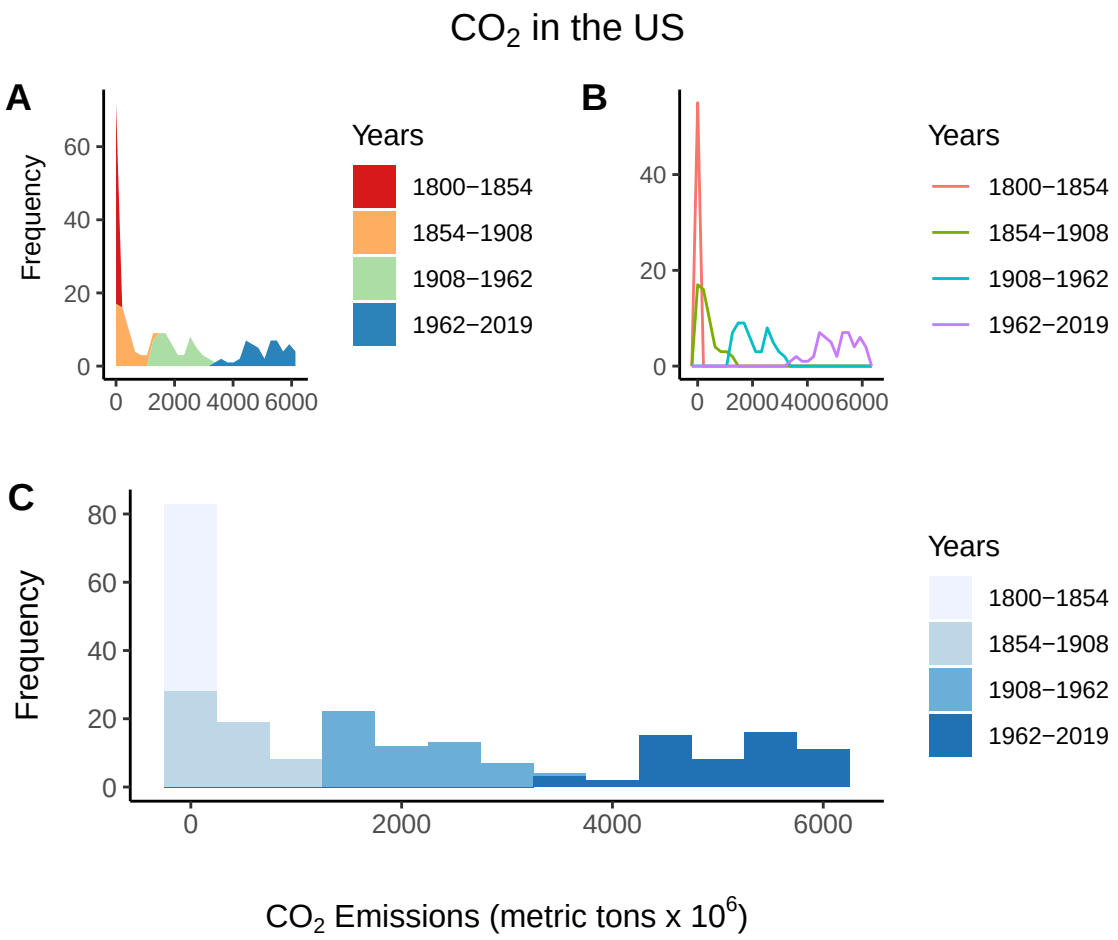


Figure 7.15: Distributional summaries of the US CO_2 data from `asbio::world.emissions`.

■

7.3.15 Barplots

Barplots are straightforward to create in *ggplot2* using the function `geom_bar()`.

Example 7.10.
Consider the `asthma` dataframe from *asbio*. We first convert the time series to a long table format using `reshape2::melt()`, and summarize it using `dplyr::summarise()`.

```
library(reshape2); data(asthma)
asthma.long <- asthma |> melt(id = c("DRUG", "PATIENT"),
                              value.name = "FEV1",
                              variable.name = "TIME")

asthma.long$TIME <- factor(asthma.long$TIME,
                           labels = c("BASE",
```

```
                                        paste("H", 11:18, sep = "")))

summary.FEV <- asthma.long |>
  group_by(TIME, DRUG) |>
  summarise(mean = mean(FEV1),
            se = sd(FEV1)/sqrt(length(FEV1)),
            meanmse = mean - se,
            meanpse = mean + se)
```

In the code for Fig 7.16 below, I group by drug treatments `group = DRUG` (line one) and plot bars using the mean values from `summary.FEV` using `ggplot2::geom_bar()` (Line 6). The argument `stat = "identity"` allows bar heights to be represented by individual numbers, in this case means. Use of `stat = "identity"` is required here. The argument `position = "dodge"` creates side by side bar plots (Line 6).

```
g <- ggplot(summary.FEV, aes(x = TIME, y = mean, group = DRUG)) +
  margin_theme() +
  labs(y = "Forced Expiratory Volume (1 min)",
       x = "Time Period")

g + geom_bar(stat = "identity", position = "dodge", aes(fill = DRUG))
```

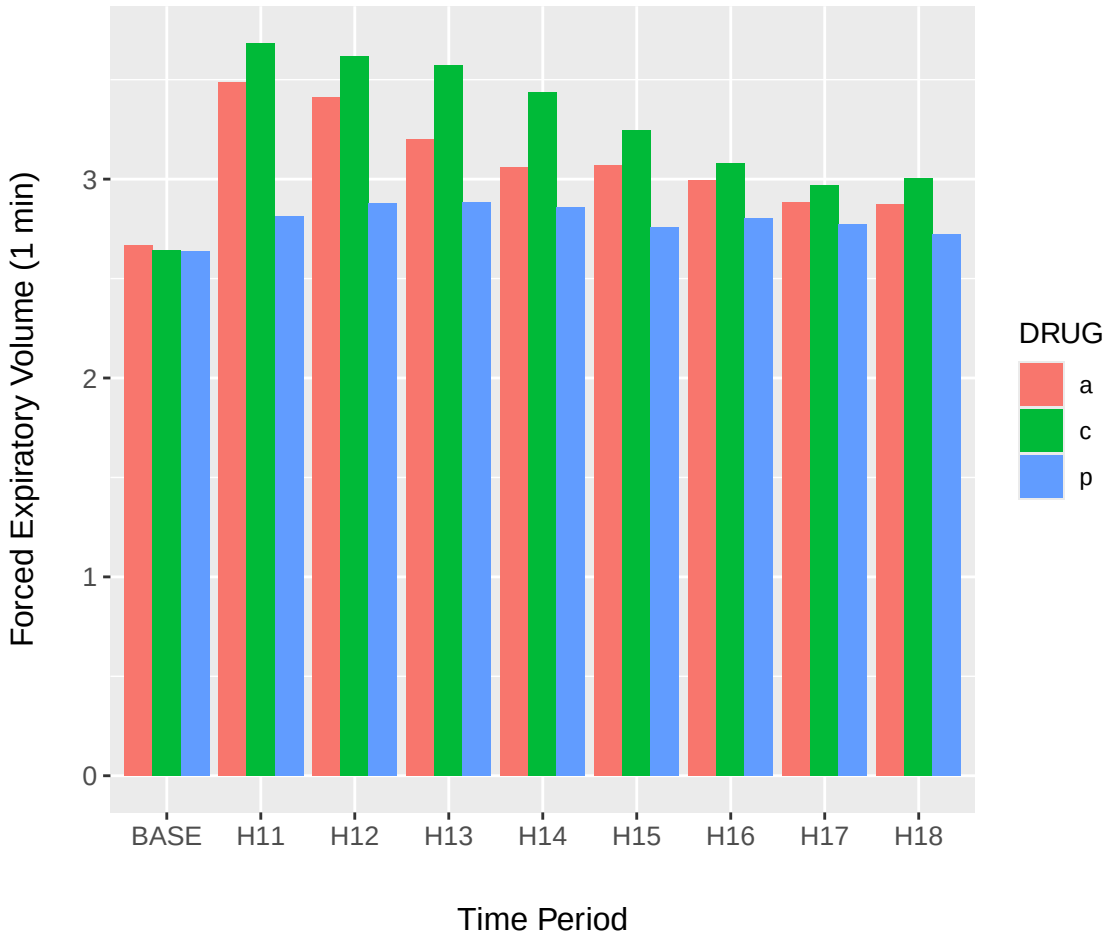


Figure 7.16: Barplot of the asthma data.

■

7.3.16 Interval Plots

The *ggplot2* package allows implementation of interval plots.

Example 7.11.
As an initial demonstration of interval plots, we continue use of barplots from Example 7.10. Overlaying errors on barplots requires the use of `stat_summary()` (Line 1) in the code below. Outcomes from `meanmse` and `meanpse` in the `summary.FEV` dataset represent $\bar{x} - SE$ and $\bar{x} + SE$, respectively. These will define the lower and upper values of the error bars in interval plot. They are called in the arguments `ymin` and `ymax` in the aesthetics of `geom_errorbar()` (Line 5). The background color of the plot is changed on Line 4. The final result is shown in Fig 7.17.

```
1 g + stat_summary(fun = "identity", geom = "bar",
2                  position = position_dodge(width = .9),
```

```
                aes(colour = DRUG), fill = "white") +
  theme(panel.background = element_rect(fill = gray(0.8))) +
  geom_errorbar(aes(ymin = meanmse, ymax = meanpse, colour = DRUG),
                width = 0.2, position = position_dodge(.9))
```

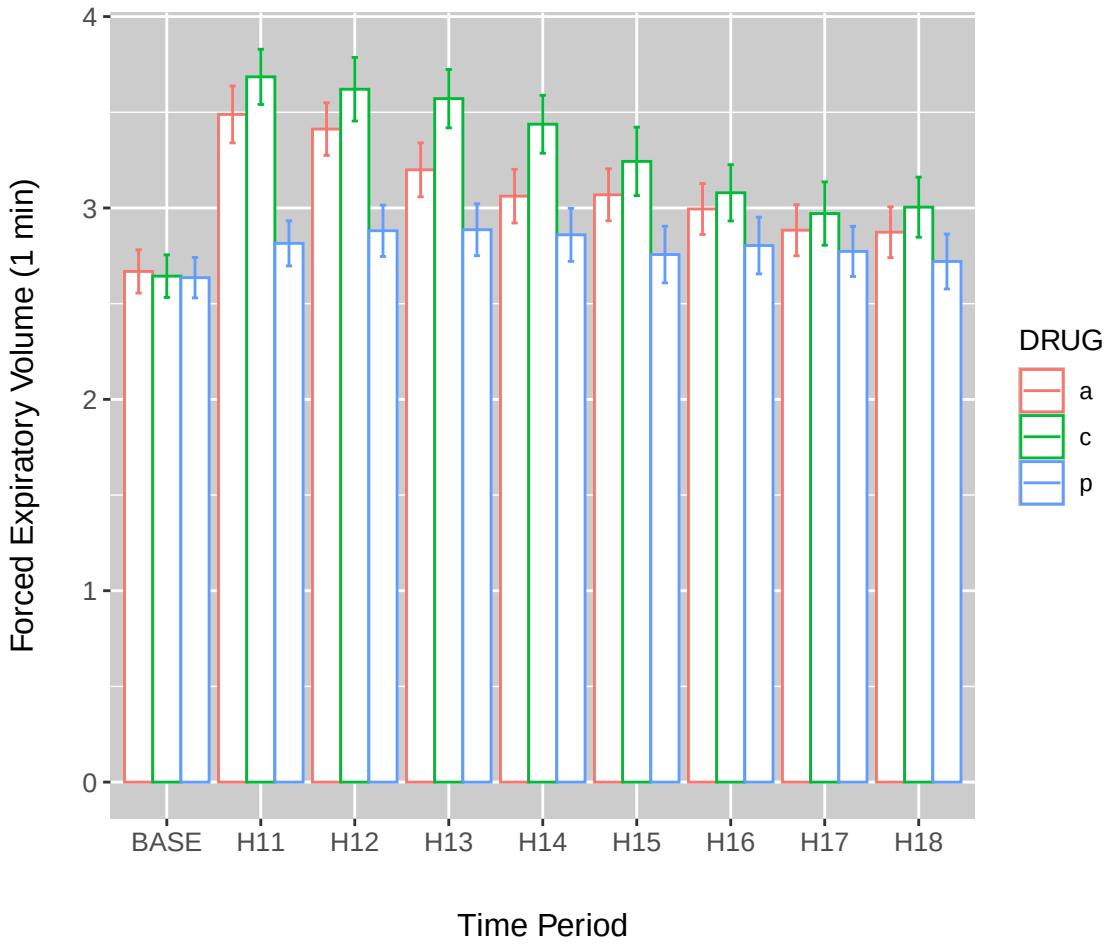


Figure 7.17: Error bars overlaid on a bar plot of the `asthma` data.

■

Example 7.12.
Next we consider overlaying intervals on a line plot Fig (7.18). In the code below, lines connect points at treatment means (Lines 2-3).

```
g + geom_point(size = 2, aes(colour = DRUG)) +
  geom_line(aes(lty = DRUG, colour = DRUG)) +
  theme_classic() +
  margin_theme() +
  geom_errorbar(aes(ymin = meanmse, ymax = meanpse, colour = DRUG),
                width = 0.2)
```

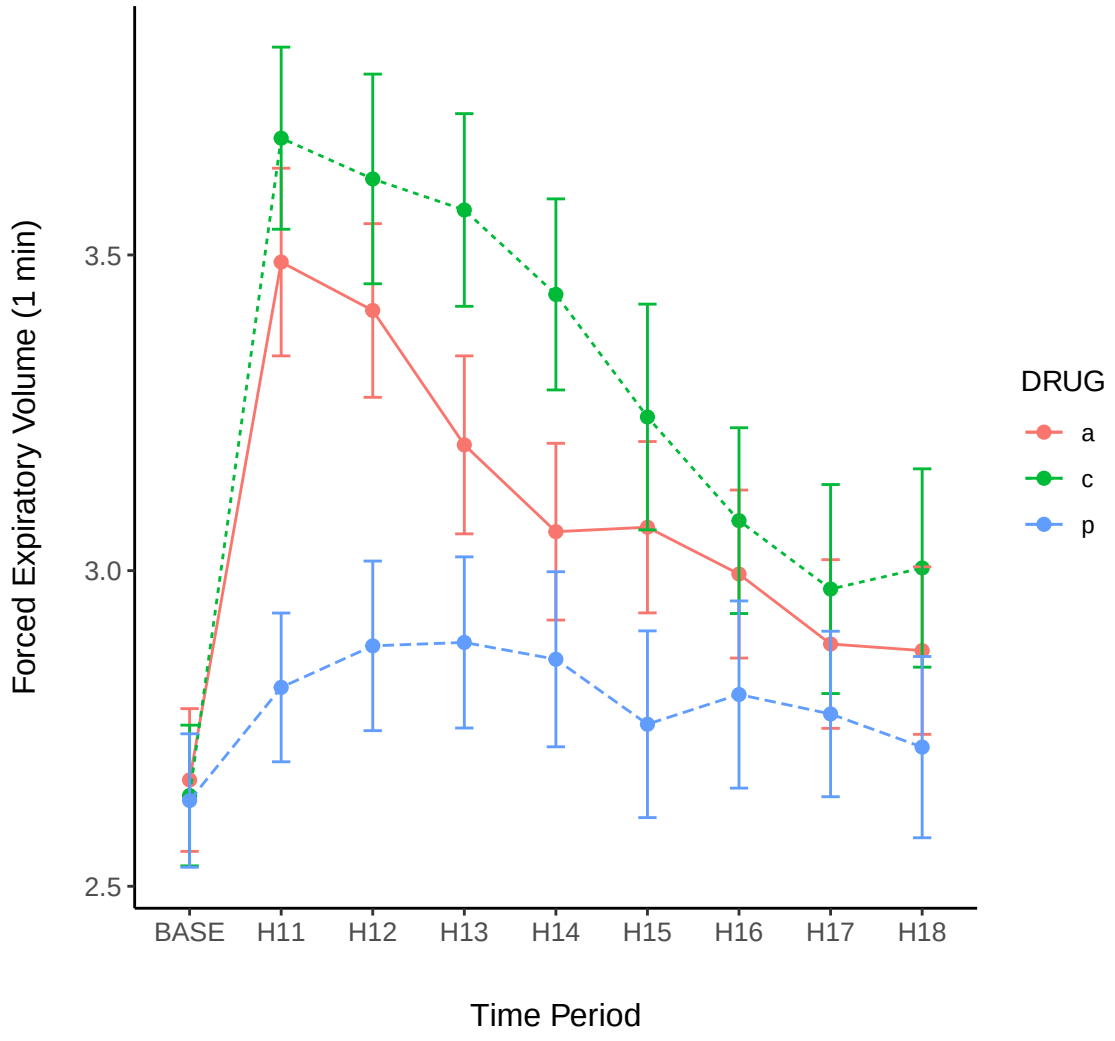


Figure 7.18: Error bars overlaid on a line plot of the `asthma` data.

■

Example 7.13.
Other geoms can be used to create interval plots, including the *ggplot2* function `geom_crossbar()`. In Fig 7.19 we show both raw data and summary standard error crossbars.

```
g + geom_crossbar(aes(ymin = meanmse, ymax = meanpse, colour = DRUG,
                      fill = DRUG), alpha = .2) +
  geom_point(data = asthma.long, aes(y = FEV1, x = TIME, colour = DRUG))
```

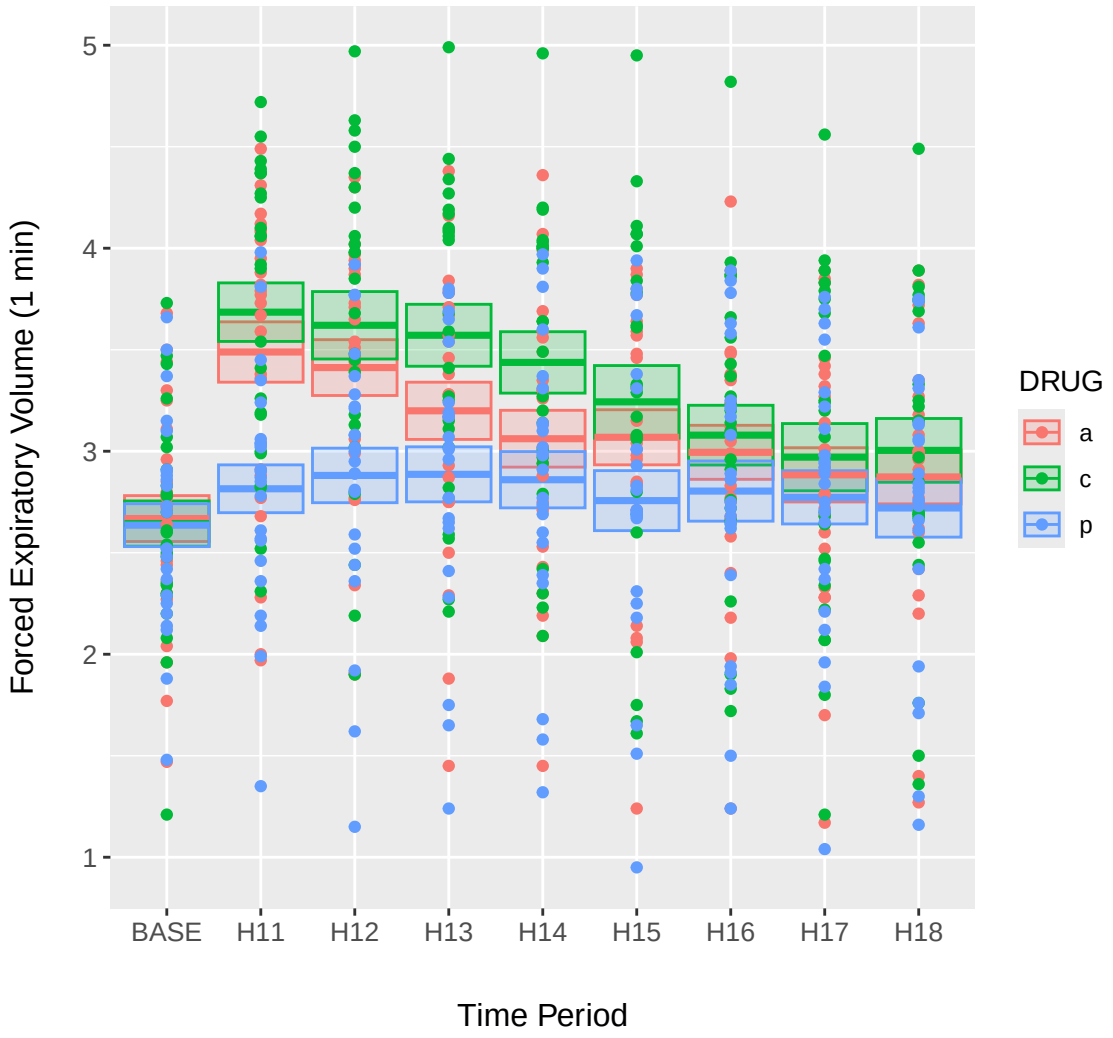


Figure 7.19: Error cross bars overlaid on the `asthma` data.

Note that individual data points are rather difficult to distinguish in Fig 7.19. As a solution we could plot points using using transparent colors, or jitter points with respect to the x-axis (Fig 7.20).

```
g + geom_crossbar(aes(ymin = meanmse, ymax = meanpse, colour = DRUG,
                      fill = DRUG), alpha = .2) +
  geom_jitter(data = asthma.long, aes(y = FEV1, x = TIME, colour = DRUG),
              alpha = .4, width = 0.15) +
  theme_classic()
```

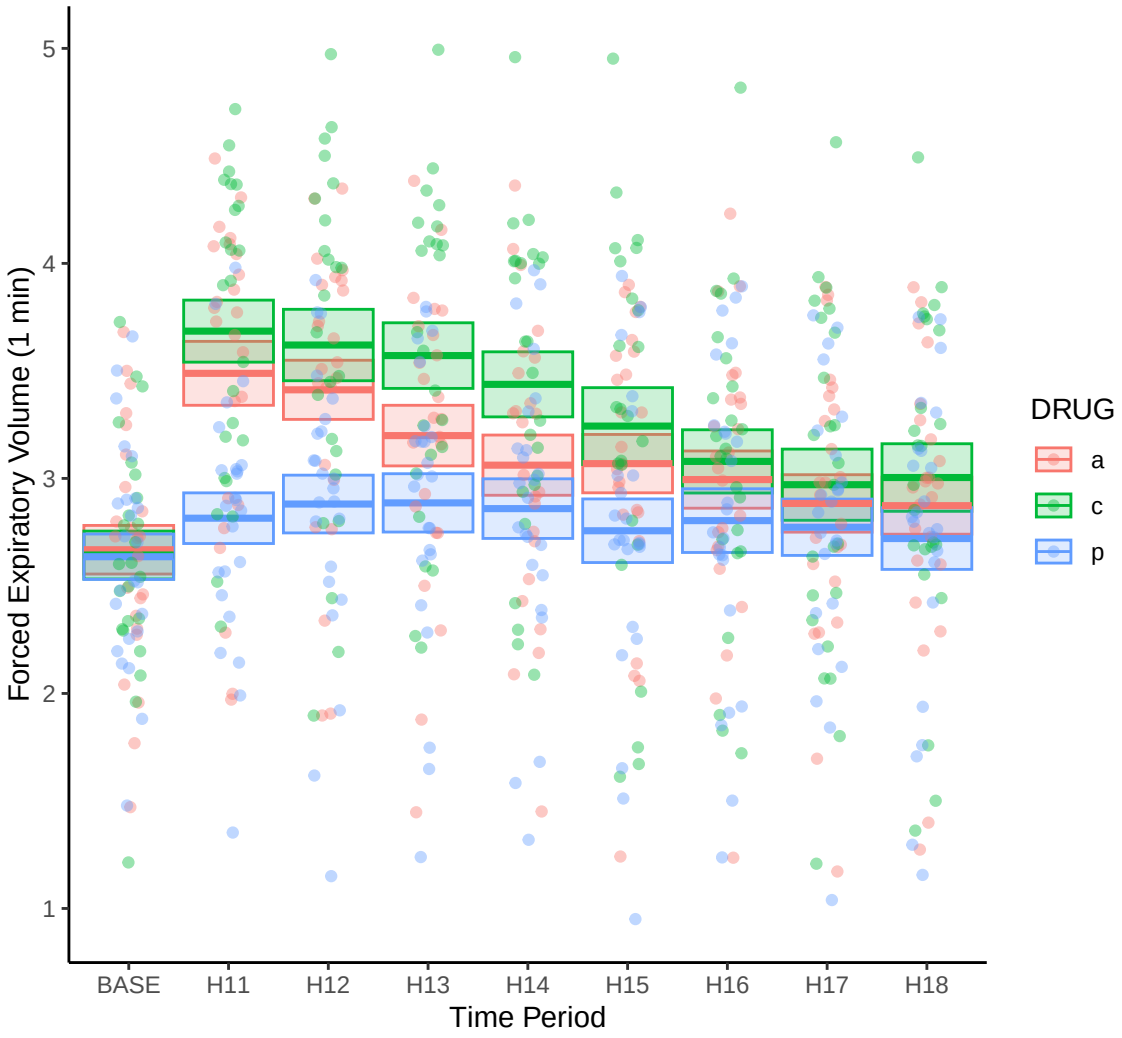


Figure 7.20: Jitter and transparency added to points in Fig 7.19.

■

7.3.16.1 Pairwise Comparisons

Results of statistical pairwise comparisons can be overlain on `ggplot2` rendered boxplots and interval plots using a number of approaches. The package *ggpubr* (Kassambara, 2023) generates *ggplot2*-based "publication ready plots' ', including interval plots showing pairwise comparisons. Examples given here largely follow those in the documentation for *ggpubr*.

Example 7.14.
Crampton et al. (1947) measured the lengths of odontoblasts (cells responsible for tooth growth) in 60 guinea pigs with respect to three dosage levels of vitamin C (0.5, 1, and 2 mg/day), and two delivery methods, orange juice (OJ) or ascorbic acid (VC). The data are in the dataframe `ToothGrowth` from the package *datasets*. In `ToothGrowth` `dose` contains dosages and `supp` contains delivery levels.

```
1  library(ggpubr)
2  df <- ToothGrowth
3  df$dose <- as.factor(df$dose)
4
5  bxp <- ggboxplot(
6    df, x = "dose", y = "len",
7    color = "supp", palette = c("#00AFBB", "#E7B800")) +
8    ylab(expression(paste("Odontoblast length (", mu,"g)"), sep = "")) +
9    xlab("Dosage (mg/day)") +
10   guides(color=guide_legend("Delivery:"))  +
11   scale_y_continuous(expand = expansion(mult = c(0.05, 0.10)))
12
13 bxp + geom_pwc(
14   aes(group = supp), tip.length = 0,
15   method = "t_test", label = "{p.adj.format}{p.adj.signif}",
16   p.adjust.method = "bonferroni", p.adjust.by = "panel",
17   hide.ns = TRUE
18 )
```

In the code above, we have the following important steps:

- On Lines 1-3, I bring in the *ggpubr* package (Line 1), rename the dataframe `ToothGrowth` to be `df` and coerce the `dose` column to be a factor.
- On Lines 5-7, I use the function `ggpubr::ggboxplot()` to define basic plot characteristics.
- On Lines 8-11, nuances are added to the plot including customized axis labels (Lines 8-9), a customized legend title (Line 10), and an alteration to the axis scale (Line 11).
- On Lines 13-18, annotations for pairwise comparisons of delivery methods (`OJ` and `VC`) within dosages are added to the graph using the function `ggpubr::geom_pwc()`.
- On Line 14, I specify that I want delivery methods (in `supp`) compared, and indicate that I don't want lines extending to the compared levels from the label lines (for comparison, see Fig 7.23).
- On Line 15, I indicate the type of test to be used in delivery method comparisons, and the labeling format. `"{p.adj.format}{p.adj.signif}"` indicates that both the adjusted *p*-value and the significance level for the adjusted *p*-value should be printed.
- On Lines 16-17, I specify use of the Bonferroni correction for simultaneous inference for three tests, and to not print results that are non-significant.

The result is shown in Fig 7.21.

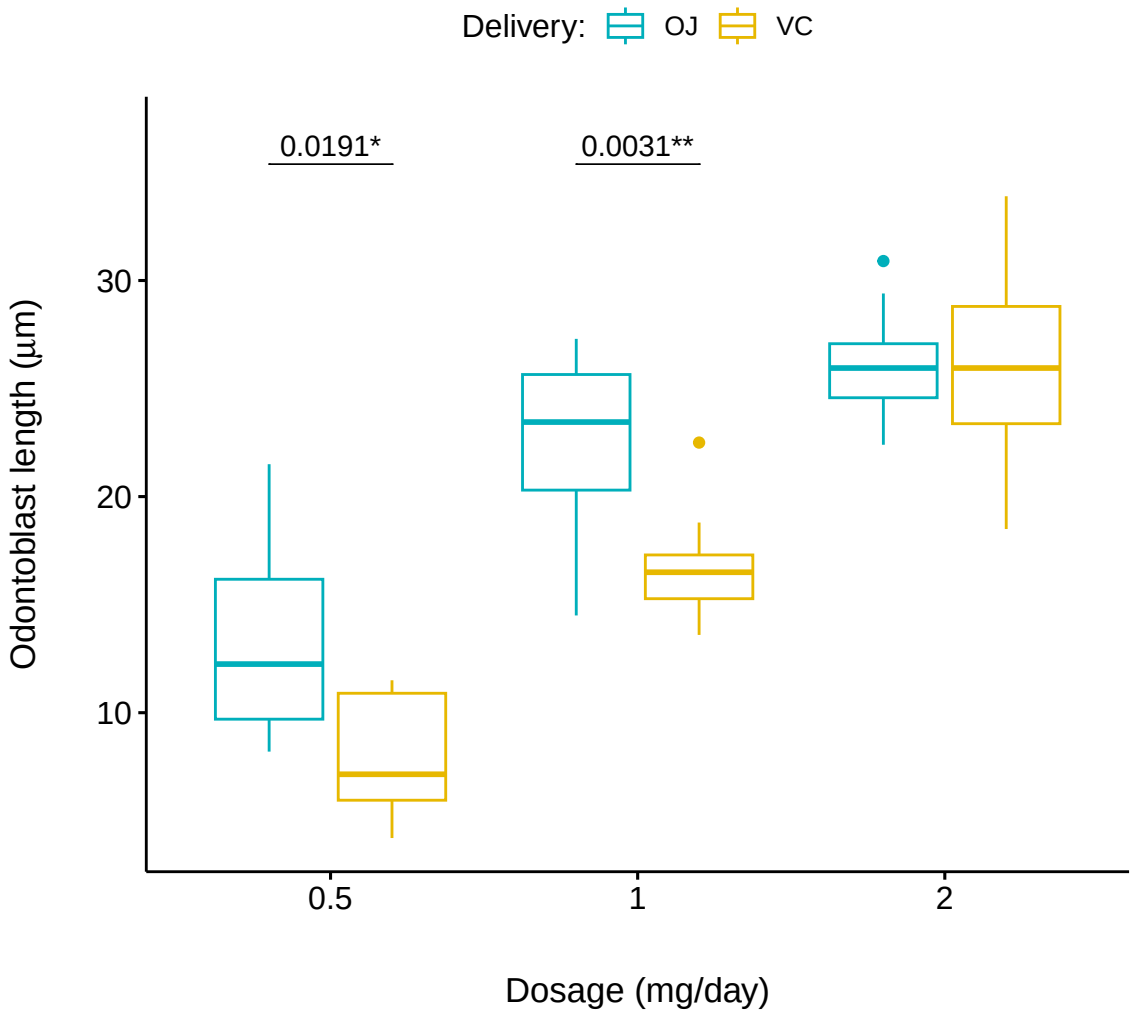


Figure 7.21: Boxplot showing pairwise comparison of delivery levels in dosage for the `Toothgrowth` dataframe.

Below we consider a more complex example that compares both delivery methods (`supp`) and dosage levels (`dose`). This is accomplished by applying `ggpubr::geom_pwc()` twice (Lines 3-7 and Lines 11-15) and printing both results (Fig 7.22).

```
# 1. Add p-values of OJ vs VC in each dose group
bxp.complex <- bxp +
  geom_pwc(
    aes(group = supp), tip.length = 0,
    method = "t_test", label = "p.adj.format",
    p.adjust.method = "bonferroni", p.adjust.by = "panel"
  )
# 2. Add pairwise comparisons between dose levels
# Nudge up the brackets by 20% of the total height
bxp.complex <- bxp.complex +
```

```
  geom_pwc(
    method = "t_test", label = "p.adj.format",
    p.adjust.method = "bonferroni",
    bracket.nudge.y = 0.2
  )
# 3. Display the plot
bxp.complex
```

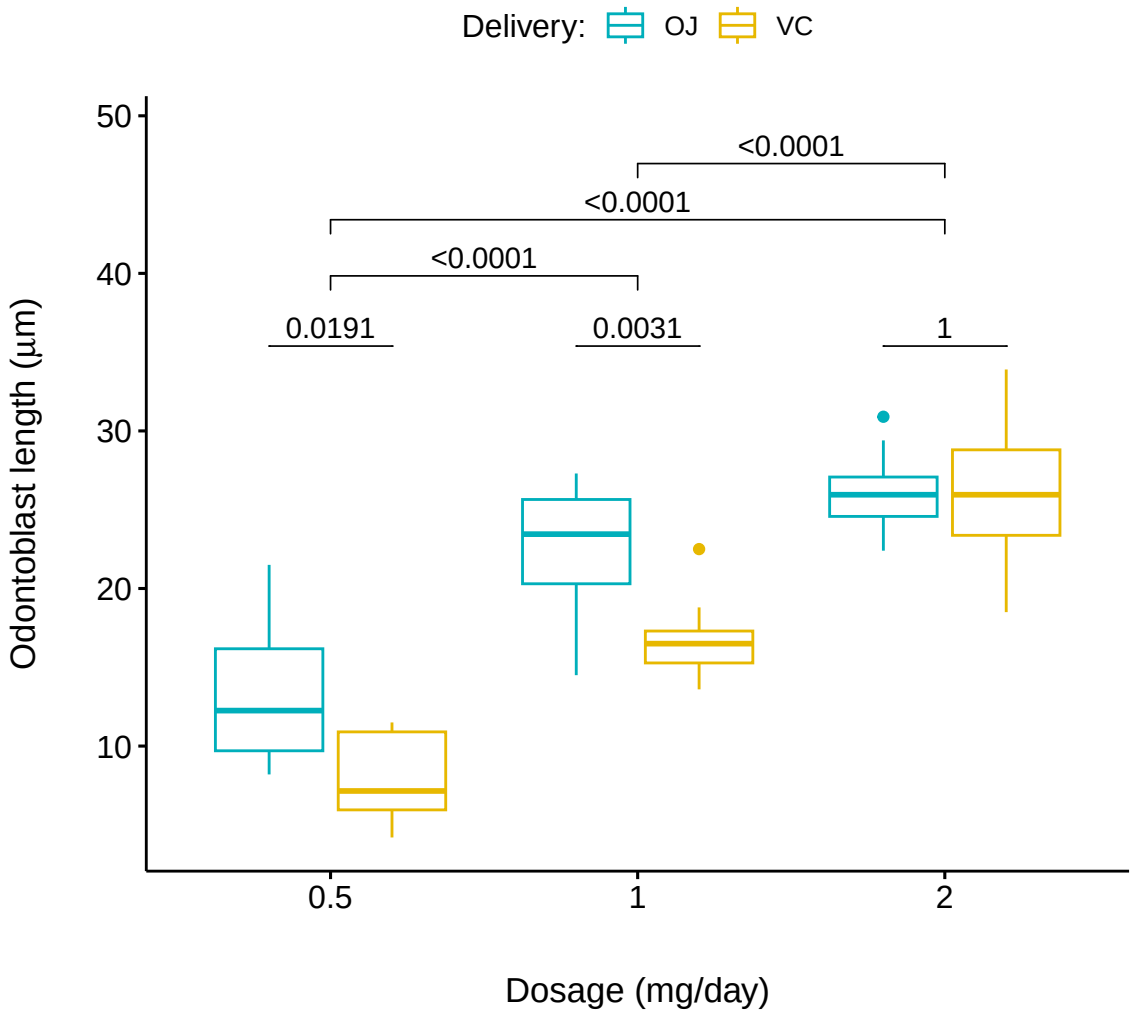


Figure 7.22: Boxplot showing pairwise comparison of delivery levels and delivery levels in dosage for the `Toothgrowth` dataframe.

In the code below, we create an interval plot with a barplot appearance using `ggpubr::ggbarplot()` (Fig 7.23. Note that this requires a different approach for customizing the title of the legend (Line 7).

```r
bp <- ggbarplot(
  df, x = "supp", y = "len", fill = "dose",
  palette = "npg", add = "mean_sd",
  position = position_dodge(0.8)) +
  ylab(expression(paste("Odontoblast length (", mu,"m)"), sep = "")) +
  xlab("Delivery method") +
  scale_fill_discrete(name = "Dosage:")

bp +
  geom_pwc(
    aes(group = dose), tip.length = 0.05,
    method = "t_test", label = "p.signif",
    bracket.nudge.y = -0.08
  ) +
  scale_y_continuous(expand = expansion(mult = c(0, 0.1)))
```

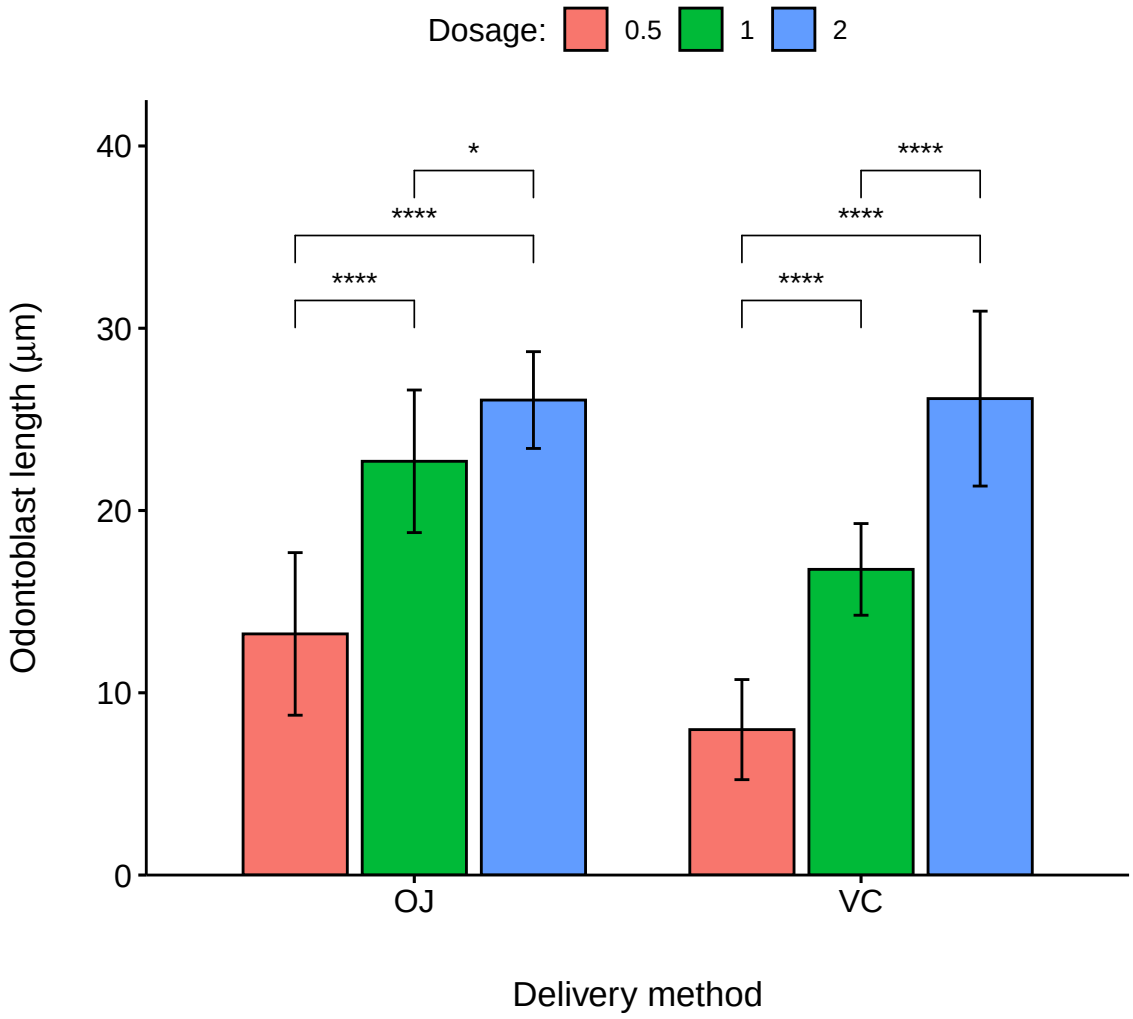


Figure 7.23: Barplot showing pairwise comparison of dosage levels in delivery methods for the `Toothgrowth` dataframe. Bar heights are means, errors are standard deviations.

■

CAUTION!

The function `ggpubr::geom_pwc()` can be potentially misused, illustrating the need for clear explanations (or understanding) when applying statistical algorithms. The default `test` specification in `geom_pwc()` is the Wilcox test, which will seldom be the most powerful method for comparing shifts in location for treatments (although it is strongly resistant to violations of normality). The argument `test = t_test` (specified in Figs 7.21-7.23) runs t-tests in isolation for each pairwise comparison, and thus will not utilize an omnibus ANOVA mean squared error, reducing power. The Bonferroni p-value adjustment method used in Figs 7.21-7.23 is also famous for its low power. Given this situation, it may be most prudent to use `ggpubr::geom_pwc()` as a graphical framework into which summaries, including p-values can be inserted manually. This can be done with the function `ggpubr::stat_pvalue_manual()` whose usage is demonstrated here.

7.3.17 Trellis Plots with Faceting

Like the package *lattice*, *ggplot2* contains functions for making trellis plots. We will use this approach to examine individual patient responses over time in the `asthma` dataset.

```
# subset data to allow readable plots
asthma.long.a <- asthma.long |>
  filter(PATIENT %in% 201:208)
```

Trellising can be enabled by using the *ggplot2* functions `facet_wrap()` and `facet_grid()`. In Fig 7.24 we define faceting within `facet_grid()` using the `PATIENT` column in the data subset `asthma.long.a`. The function `ggplot2::vars()` in `facet_grid()` is analogous to the use of `aes()` in geoms.

```
g <- ggplot(asthma.long.a, aes(y = FEV1, x = TIME, colour = DRUG,
                               group = DRUG)) +
  geom_point() +
  geom_line() +
  theme_light() + margin_theme() +
  theme(axis.text.y = element_text(size=rel(0.7))) +
  facet_grid(rows = vars(PATIENT)) +
  scale_colour_brewer(palette = "Dark2") +
  ylab("Forced Expiratory Volume (1 min)") +
  xlab("Time period")
g
```

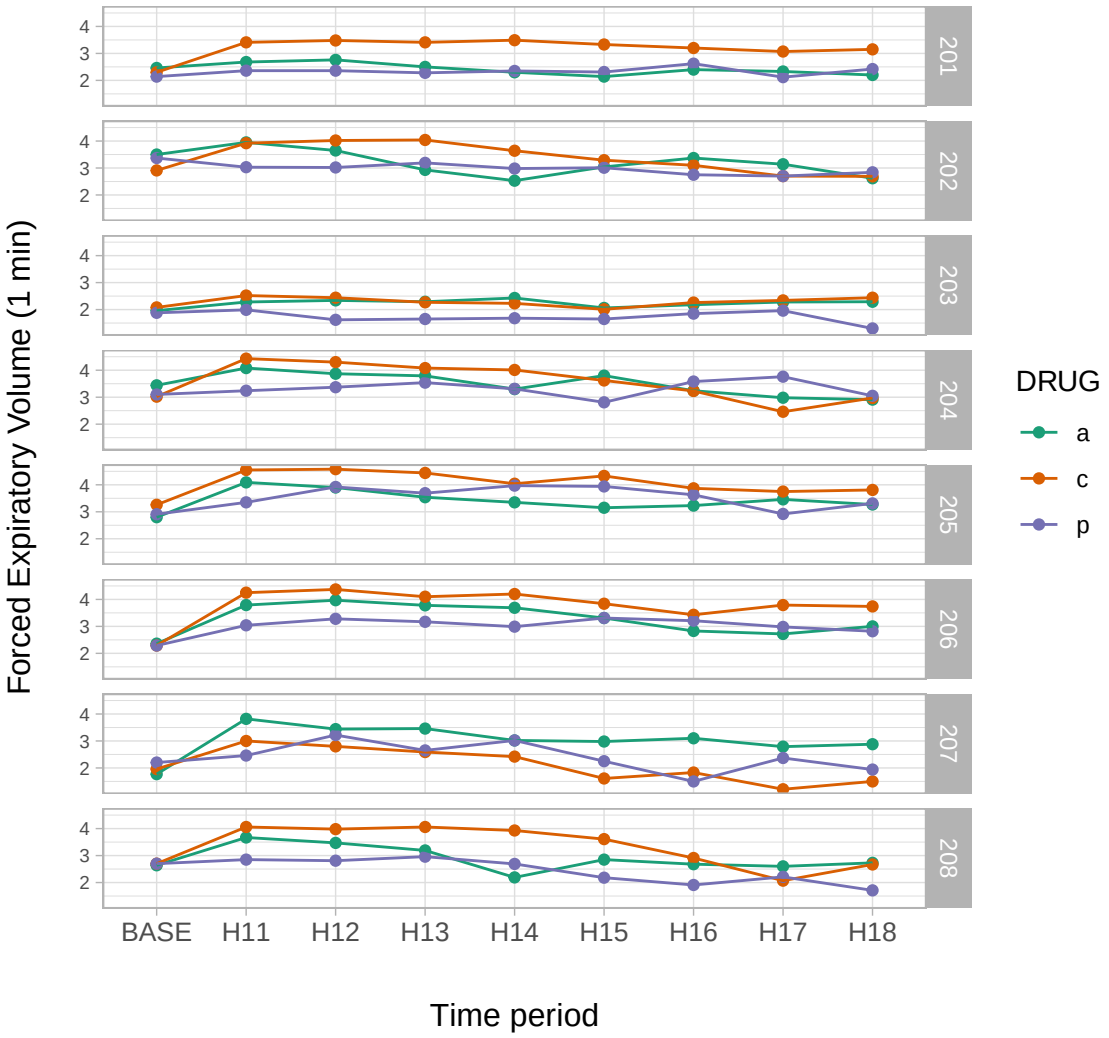


Figure 7.24: A trellis plot showing individual patient responses over time from the `asthma` dataset.

7.3.18 Multivariate Distributional Summaries

Bivariate summaries can be shown in many ways using a *ggplot2* approach.

Example 7.15.
In Fig 7.25, I insert density grobs (graphical objects) on the margins of a scatterplot for five European countries using the *cowplot* functions `axis_canvas()`, `insert_xaxis_grob()`, `insert_xaxis_grob()`, and `gg_draw()`. The right margin shows GDP distributions for each country, whereas the top margin shows CO_2 emission distributions for each country. I also change symbol sizes with year in the main graph. Larger symbols indicate more recent years.

```
europe <- world.emissions |>
  filter(country == c("France", "Italy", "Germany", "United Kingdom")) |>
```

```
  filter(year <= 2019 & year > 1950) # comparable data

pmain <- ggplot(europe, aes(x=co2, y=gdp, color= country)) +
  geom_point(aes(size = year), alpha = .6) +
  xlab(xlab) + ylab("GDP (International dollars)") +
  margin_theme() +
  theme_classic()

xdens <- axis_canvas(pmain, axis = "x") +
  geom_density(data = europe, aes(x = co2, fill = country), alpha=0.6,
               size=.2)

ydens <- axis_canvas(pmain, axis = "y") +
  geom_density(data = europe, aes(y = gdp, fill = country), alpha=0.6,
               size=.2)

p1 <- insert_xaxis_grob(pmain, xdens, grid::unit(.2, "null"),
                        position = "top")
p2 <- insert_yaxis_grob(p1, ydens, grid::unit(.2, "null"),
                        position = "right")
ggdraw(p2)
```

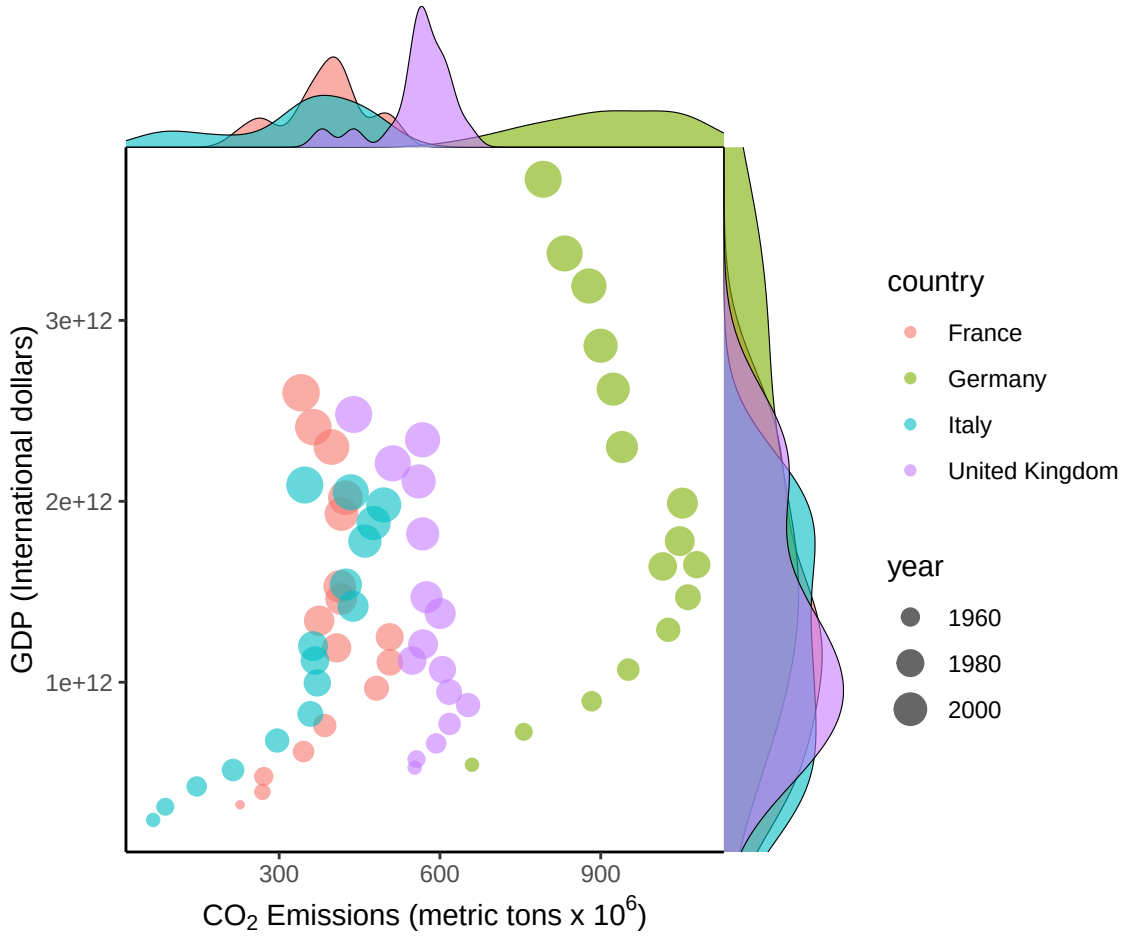


Figure 7.25: Bivariate summaries for European countries from the `asbio::world.emissions` dataset.

■

7.3.19 Maps

The *ggplot2* ecosystem has some support for mapping, including import of ARC-GIS shape files, and creation of map polygons. The function `sf::st_read()` allows loading of simple spatial features, including shapefiles, and the package *ggspatial* provides a number for creating useful maps under a *ggplot2* framework.

Example 7.16.
As an example we will create a map of a small stream network in southwest Idaho named Murphy Creek. Data concerning the creek, including shapefiles, is contained in the package *streamDAG* (Aho et al., 2023b).

```
library(sf); library(ggspatial); library(streamDAG)
mur_sf <- st_read(system.file("shape/Murphy_Creek.shp",
```

```
                                    package="streamDAG"))
data(mur_coords)
coords <- mur_coords[,c(2,3)]
```

```
Reading layer `Murphy_Creek' from data source
  `C:\Users\ahoken\AppData\Local\R\win-library\4.5\streamDAG\shape\Murphy_Creek.shp'
  using driver `ESRI Shapefile'
Simple feature collection with 2 features and 2 fields
Geometry type: LINESTRING
Dimension:     XY
Bounding box:  xmin: 512860 ymin: 4789000 xmax: 514720 ymax: 4789300
Projected CRS: NAD83 / UTM zone 11N
```

The function `ggplot2::geom_sf()` (Line 2 below) can be used to draw different geometric objects depending on features present in the data, e.g., points, lines, or polygons. For the current case a line is generated. The function `ggplot2::expand_limits()` (Line 6) is used to increase the spatial range of the y-axis which otherwise would be extremely narrow (since a single W to E trending line, representing the watershed, is being generated by `geom_sf()`). The *ggspatial* functions `annotation_scale()` and `annotation_north_arrow()` provide spatially explicit scalebars and north-indicating arrows, respectively (Lines 8-9). The final product is shown in Fig 7.26.

```
1  g <- ggplot(mur_sf) +
2    geom_sf(colour = "lightblue", lwd = 2) +
3    theme_classic() +
4    geom_point(data = coords, aes(x = E, y = N), shape = 21,
5               fill = "orange", size = 2.5) +
6    expand_limits(y = c(4788562,4789700)) +
7    ylab("") + xlab("") +
8    annotation_scale() +
9    annotation_north_arrow(pad_x = unit(10.5, "cm"), pad_y = unit(6.6, "cm"))
10 g
```

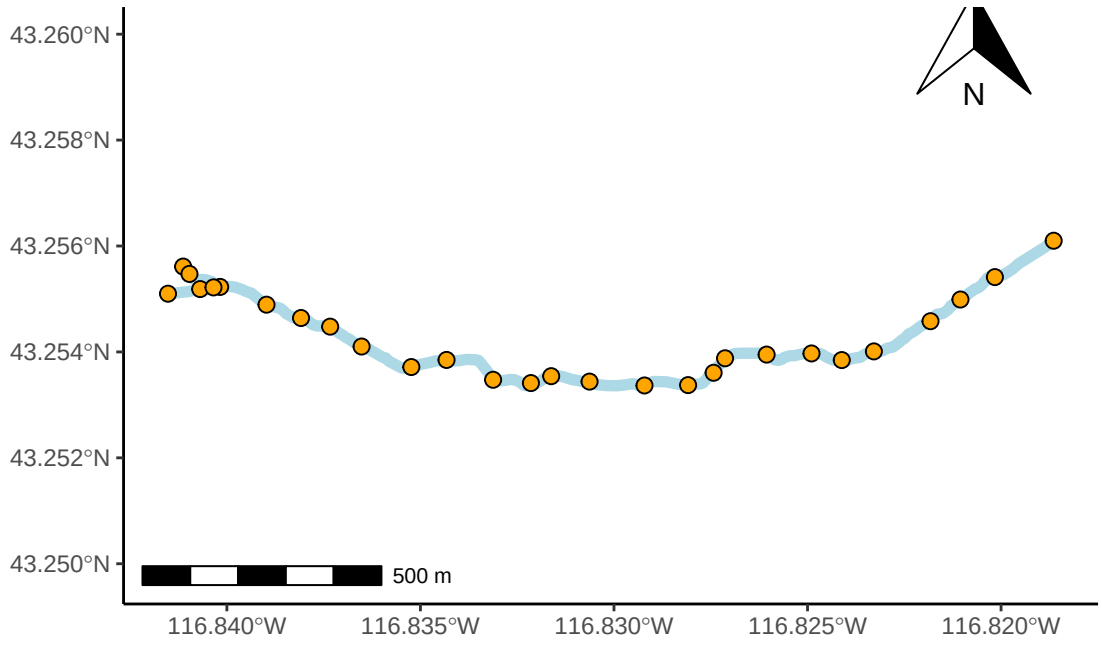


Figure 7.26: Map of the Murphy Creek drainage system in southwest Idaho (outlet coordinates: 43.71839 °N, 116.13747 °W).

■

7.3.20 Animation

Animations in *ggplot2* can be created using looping strategies applied in Ch 6. Looping will be explicitly considered in the context of functions in Chapter 8.

Example 7.17.
As an initial demonstration, we reconsider the asthma data (Fig 7.16). We first construct a function, `asthma.plot()` which will render a ggplot. The lone argument of `asthma.plot()`, `upper`, defines the upper time limit of under consideration in the longitudinal asthma drug study (Line 3). The `upper` argument is called in `geom_line()` (Lines 5-6) to subset, if necessary, the underlying data. Vital to the animation is the `print.ggplot()` function (Line 12). Failure to include this code will create an empty animation.

```
1 summary.FEV$time <- rep(c(0,11:18), each = 3)
2
3 asthma.plot <- function(upper){
4   a <- ggplot() +
5     geom_line(data = summary.FEV[summary.FEV$time > 10 &
6                                  summary.FEV$time <= upper,],
7               aes(x = time, y = mean, colour = DRUG)) +
8     ylim(2.6, 4) +
9     xlim(11, 18) +
```

```
    margin_theme() +
    labs(y = "Forced Expiratory Volume (1 min)",
         x = "Time (Hrs)")
  print(a)
}
```

Next, we create a function that runs `asthma.plot()` for a range of values for `upper`. The function consists of a loop run by `lapply()` (lines 2-3). The final lines of code (lines 8-9) allow the animation to be saved using the function `saveGIF()` from the package *animation*[5].

```
asthma.animate <- function() {
  lapply(12:18, function(i){
    asthma.plot(i)
  })
}

# run animation
asthma.animate()

# save frames into one GIF:
library(animation)
saveGIF(asthma.animate(), interval = 1, movie.name="asthma.gif")
```

The animation result is shown in Fig 7.27.

[5]As noted in Ch 6, use of `animation::saveGIF` requires installation of open source software ImageMagick or GraphicsMagick (see `?saveGIF`).

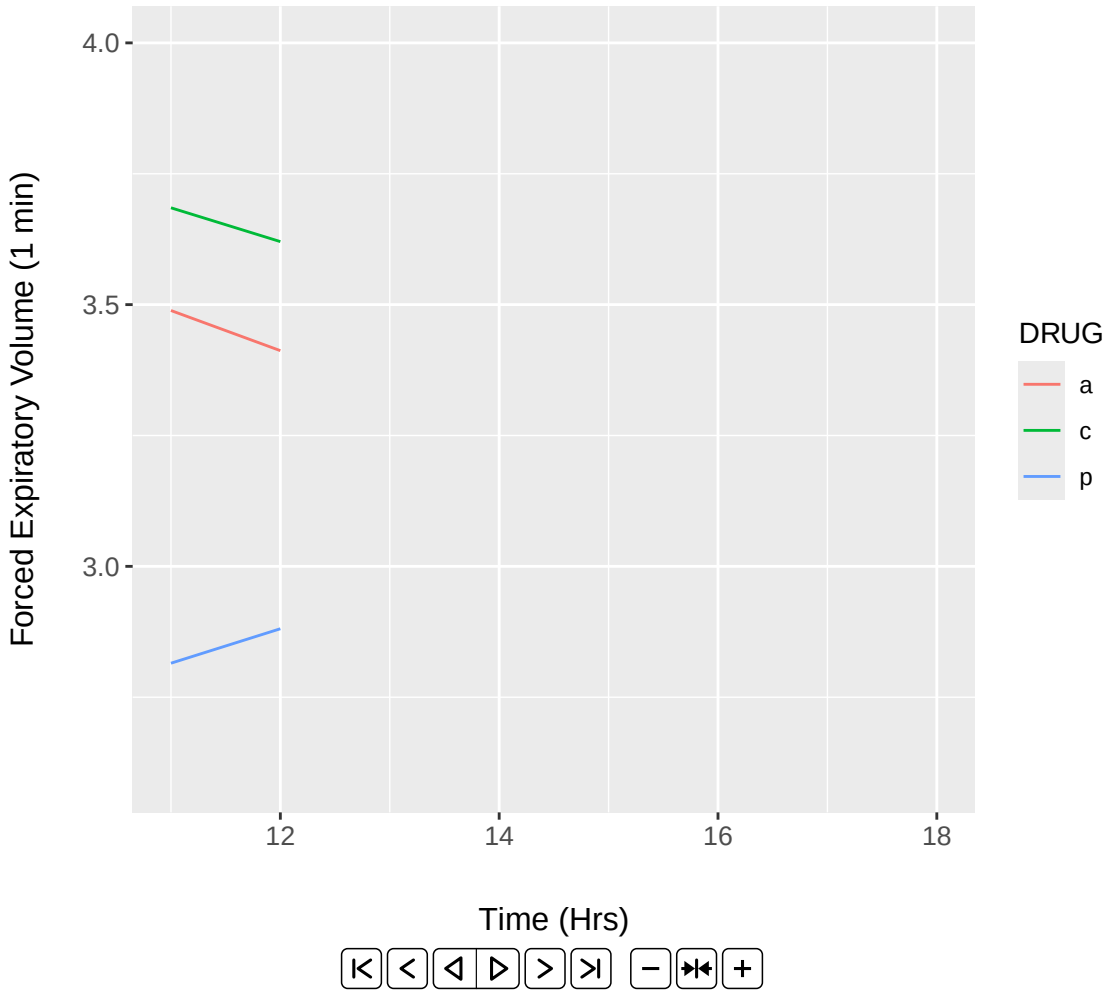


Figure 7.27: Animation demonstration using the `asthma` dataset.

■

Amazing animations can be created with the package *gganimate*. These are demonstrated using several examples.

Example 7.18.
In this example we create a scatterplot animation for the world emissions dataset. In the code below, steps particularly important to the animation occur on Lines 14-17.

- On Line 14 the plot title is modified as the animation progresses, allowing tracking of years. The code `title = 'Year: {frame_time}'` is a *gganimate* convention for extracting corresponding time sequence values (in this case the column `world.emission$year`) for a projection.
- On line 16 The function `gganimate::transition_time()` calls frame transitions between specific point in time in the column `year`. Usefully, the *gganimate* sets the transition time between the states in `transition_time()` to correspond to the actual time difference between them.

- On line 17 `ease_aes()` is used to linearly smooth the animation (in terms of coloration and the geometric positioning of features) between animation frames. The function is based on analogous functions from the package *tweener*.

The package *gapminder* contains rational color designations (i.e., variations on prime colors within continents) for 142 countries. Countries without a color designation are colored gray by `scale_colour_manual()`.

```
1  library(gganimate)
2  library(gapminder)
3  world.data.sub <- world.emissions |>
4    filter(continent != "Redundant") |>
5    filter(year > 1950)
6
7  g <- ggplot(world.data.sub, aes(x = gdp, y = co2, size = population,
8                                  colour = country)) +
9    geom_point(alpha = 0.7, show.legend = FALSE) +
10   scale_colour_manual(values = country_colors) +
11   scale_size(range = c(2, 12)) +
12   scale_x_log10() + scale_y_log10() +
13   facet_wrap(~continent) +
14   labs(title = 'Year: {frame_time}', x = 'GDP') +
15   ylab(bquote(CO[2])) +
16   transition_time(as.integer(year)) +
17   ease_aes('linear')
18   margin_theme()
19 g
```

The final result is shown in Fig 7.28.

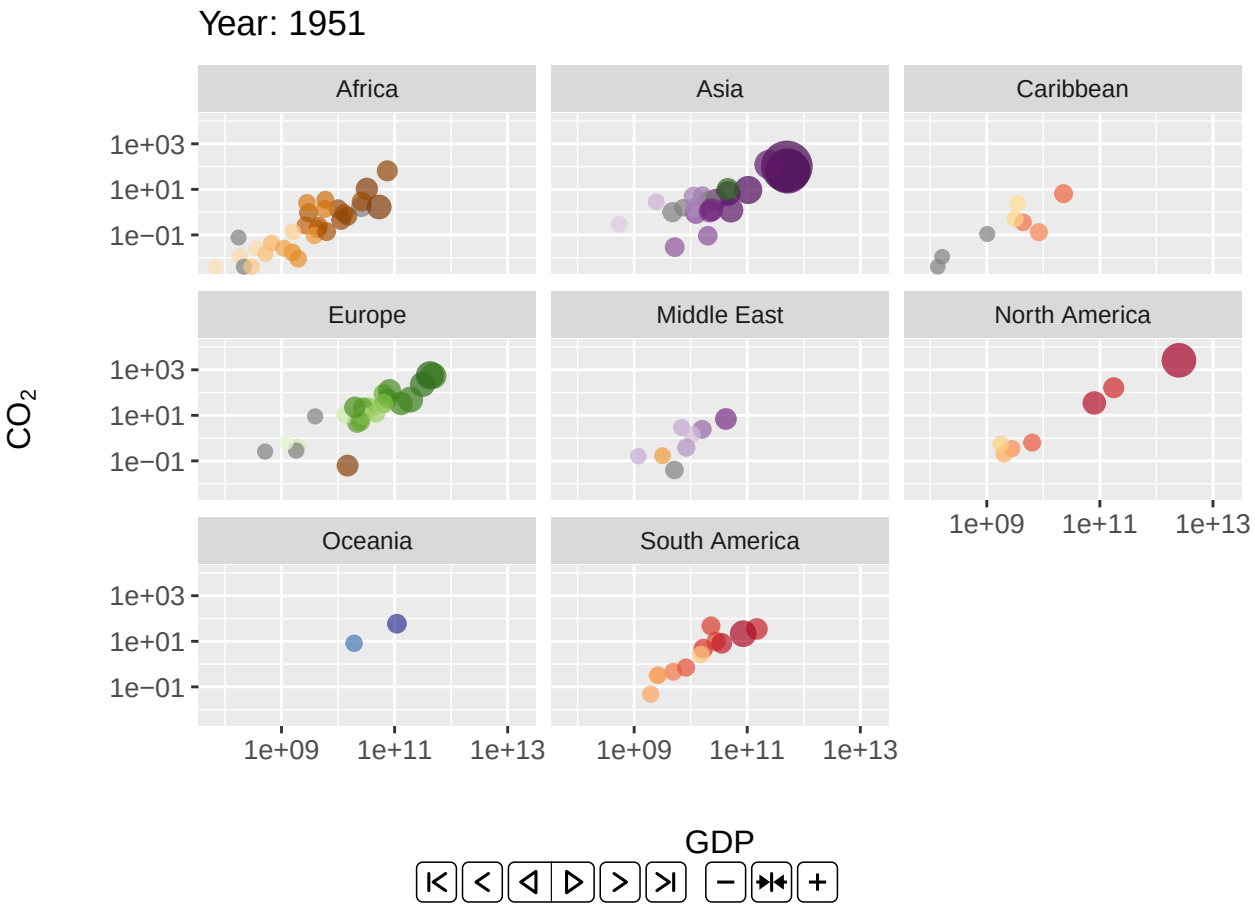


Figure 7.28: Animated scatterplot of CO_2 levels over time for countries within continents. Symbol size scaled by population size.

■

Example 7.19.
The next example uses *gganimate* to animate variation in CO_2 levels over time within continents, using boxplots.

```
g <- ggplot(world.data.sub, aes(x = continent, y = co2,
                                group = continent)) +
  geom_boxplot(aes(fill = continent), show.legend = FALSE) +
  scale_y_log10() +
  labs(title = 'Year: {frame_time}', x = '',
       y = "CO\U2082") +
  theme(axis.text.x = element_text(angle = 50, hjust = 1,
                                   vjust = 0.9, size = 12)) +
  transition_time(as.integer(year))
g
```

The final result is shown in Fig 7.29.

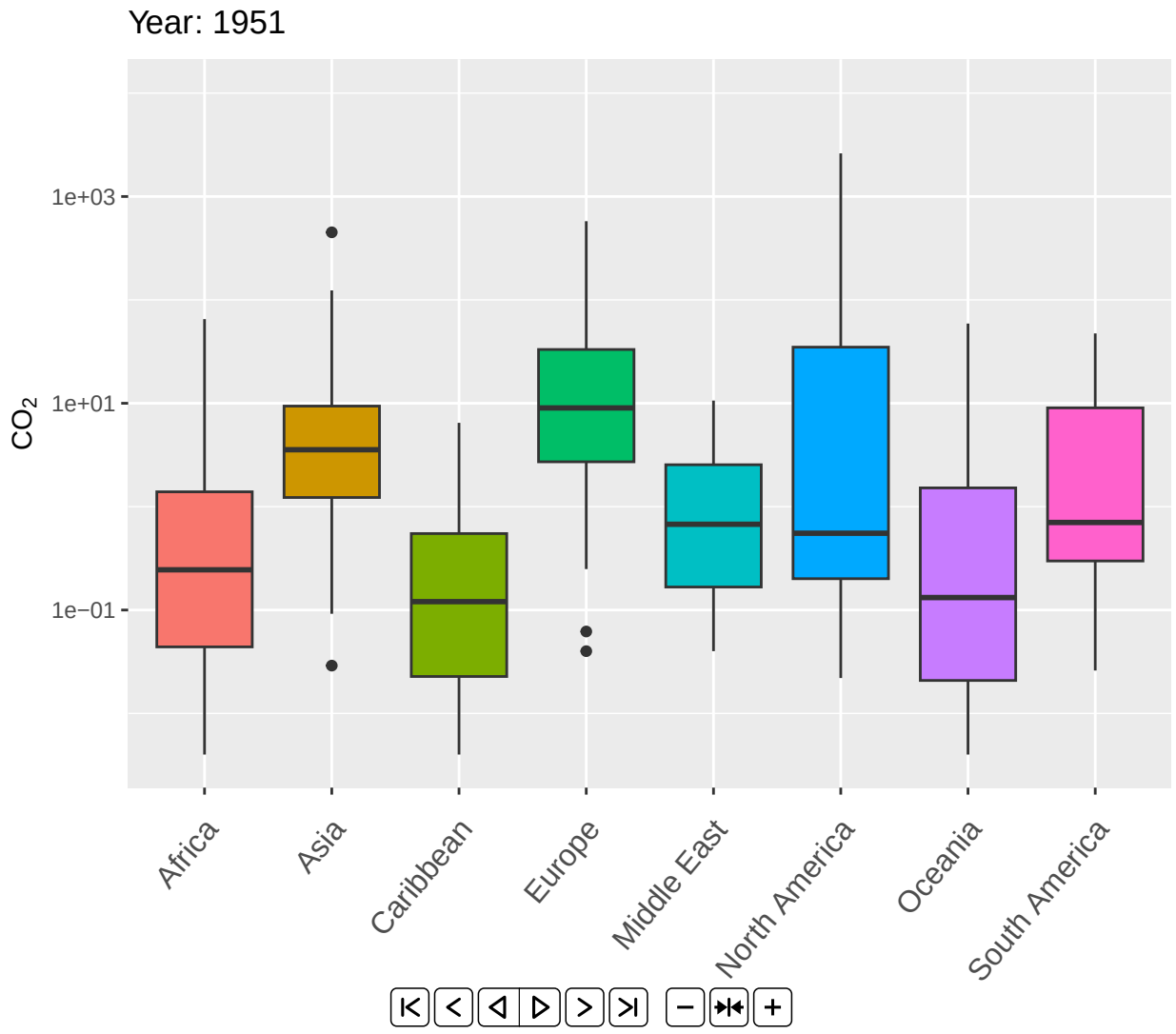


Figure 7.29: Animated boxplot of CO_2 levels over time for countries within continents.

■

Example 7.20.
As a final (rather complex) example, I animate the non-perennial character of Murphy Creek (Fig 7.26) over time.

To prepare for making the map animation, I first bring in a dataset that documents the presence/absence of surface water $= \{0, 1\}$ at 28 locations (i.e., nodes) over 1163 time steps, `mur_node_pres_abs` (Line 1). The 27 stream sections bounded by the the 28 nodes are defined as stream segments. I select from these time designations, at even intervals, to create a data subset of 250 time steps (Lines 2-8).

```
1 data(mur_node_pres_abs)
2 u <- unique(mur_node_pres_abs$Datetime)
```

```
n <- length(u)
frames <- 250
times.sub <- u[round(seq(1, n, length = frames),0)]

w <- which(mur_node_pres_abs$Datetime %in% times.sub)
mnpa.sub <- mur_node_pres_abs[w,]
```

In the code below, the function `sf::st_coordinates()` is used to pull spatial coordinates from the Murphy Creek shapefile underlying the map in Fig 7.26. I also use several functions from the *streamDAG* package, including `streamDAGs()`, which creates a graph-theoretic representation Murphy Creek (see Aho et al. (2023b)), and thus defines how the stream flows from location to location. The function `streamDAG::STIC.RFimpute()` is a wrapper for the random forest algorithm `missForest::missForest()`, and allows imputation of missing stream presence/absence data from the dataset `mur_node_pres_abs`. The function `arc.pa.from.nodes()` from *streamDAG* creates stream segment surface water presence/absence outcomes based on data from the downstream bounding node of each segment (`approach = "dstream"`). The function `vector_segments()` from *streamDAG* is used to create the dataframe `vs` that contains arcs designations for shapefile coordinates in `sf.coords`, based on coordinates in the object `node.coords`, and the function `assign_pa_to_segments()` adds surface water presence/absence designations to `vs` based on outcomes from the object `arc.pa`, whew.

```
mur_graph <- streamDAGs("mur_full")
# impute missing presence/absence data
out <- STIC.RFimpute(mnpa.sub[,-1])
mur.pa.sub <- out$ximp
# arcs from nodes
arc.pa <- arc.pa.from.nodes(mur_graph, mur.pa.sub, approach = "dstream")

node.coords <- data.frame(mur_coords[,(2:3)])
row.names(node.coords) <- mur_coords[,1]
sf.coords <- st_coordinates(mur_sf)[,-3]

vs <- vector_segments(sf.coords, node.coords, realign = TRUE,
                      colnames(arc.pa), arc.symbol = " -> ")
datetime <- mnpa.sub$Datetime
vsn <- assign_pa_to_segments(vs, frames, arc.pa, datetime = datetime)
```

Using the data summaries created from the steps above, I can finally create an animated ggplot map.

```
g <- ggplot(mur_sf) +
  geom_sf(colour = "gray", lwd = 1.8) +
  theme_classic() +
  geom_line(data = vsn, aes(x = x, y = y, group = arc.label,
```

```
                            colour = as.factor(Presence)),
              show.legend = FALSE, lwd = 1.5) +
  scale_colour_manual(values = c("orange","lightblue")) +
  geom_point(data = node.coords, aes(x = E, y = N), shape = 21,
             fill = "white", size = 1.4) +
  expand_limits(y = c(4788562,4789700)) +
  annotation_scale() +
  labs(title = "Date: {frame_time}", x = "", y = "") +
  annotation_north_arrow(pad_x = unit(10.5, "cm"),
                         pad_y = unit(6.6, "cm")) +
  transition_time(as.Date(vsn$Time))
```

The final result, Fig 7.30, shows changing patterns of surface water presence at the Murphy Creek network during the summer of 2019.

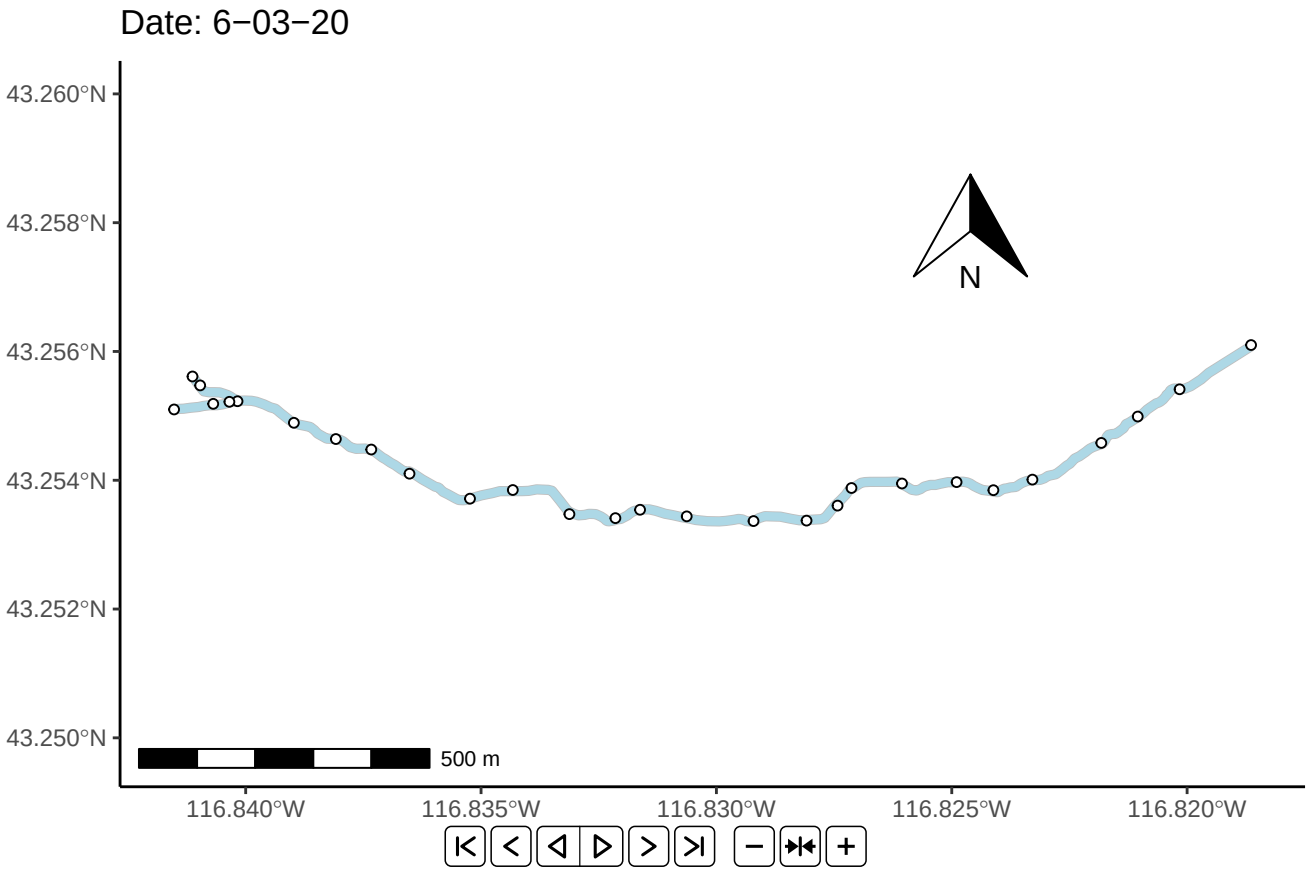


Figure 7.30: Paterns of drying at Murphy Creek, Idaho shown with an animated map. Blue segments indicate the presence of surface water. Gray segments indicate missing data.

■

Exercises

1. Complete the following data management steps on the `asbio::world.emissions` data.
   (a) Eliminate redundant rows using `continent != 'Redundant'` and `dplyr::filter`.
   (b) Filter further to subset the data to the years 1955-2019.
   (c) Filter further to subset the data to 8 countries of interest (your choice).
   (d) Name the dataset `emissions.sub`.
   (e) For the `emissions.sub` dataset, plot CO_2 emissions as a function of year in a scatterplot. Save the ggplot as an object (e.g., g).
2. Continuing Question 1, overlay a linear regression model on g using `geom_smooth(method = "lm")`.
   (a) Extract fitted model components using `g + stat_poly_eq(formula = y ~ x, geom = "debug")` from library *gginnards*. What is the model slope?
   (b) Interpret the meaning of the shaded envelope around the line.
   (c) Annotate the model onto the graph using: `g + stat_poly_eq()` from library *ggpmisc*.
3. Continuing Question 1, (1) color points in g by country (use transparency to allow viewing of points laying atop each other), and (2) vary point size by population size.
4. Continuing Question 1, add a label in g identifying US emissions in 2005.
5. Continuing Question 1, use `geom_hline()` and/or `geom_vline()` to add reference lines to g (your choice as to relevant x or y-axis location).
6. Continuing Question 1, alter the the y-axis limits in g (your choice of limits).
7. Continuing Question 1, create (1) a boxplot, and (2) an interval plot showing CO_2 emissions as a function of country. Interpret the meaning of the hinges, centers, and whiskers of the boxplot and interpret the "errors" in the interval plot.
8. (Advanced) For the dataframe `npk` in the package *datasets*, use functions in the the package *ggpubr* to overlay results of pairwise comparisons of population means on interval plots. Specifically:
   (a) Use `ggbarplot()` to make a barplot showing mean Yields and standard errors as a function of nitrogen `N` and phosphate `P`. Vary bar colors using `P`. Create appropriate axis and legend labels.
   (b) Overlay pairwise comparisons for both `N` and `P` levels on the barplot using the function `geom_pwc()`. Specify `method = t_test` since multiple tests for N will not occur within levels of P.
9. For the `asbio::goats` dataframe, use ggplot approaches to …
   (a) Make plots of the distribution of `NO3` using two of the following functions: `geom_area()`, `geom_freq()`, `geom_dotplot()`, or `geom_density()`.
   (b) Create a scatterplot of `NO3` as a function of feces, Change symbol sizes to reflect the values in `organic.matter`.
   (c) Plot `NO3` and `organic.matter` as simultaneous functions of `feces` by adding a second *y*-axis.

10. Using *gganimate*, and the `asbio::asthma` dataframe, track subject FEV1 levels over time with `geom_point()`. Use faceting to distinguish drug levels.

Chapter 8

Functions

> *"A computer will do what you tell it to do, but that may be much different from what you had in mind."*

- **Joseph Weizenbaum**, *Important early software developer and AI ethisist*

8.1 Introduction to Functions

In computer programming, a *function* is a set of instructions for performing a specific task, or providing specific output. Essentially all processes in **R** are run via functions, prompting the idiom: "Everything that happens in **R** is a function call" (Chambers, 2008). For example, the command: `x <- 2`, assigns the label `x` to the numeric value 2. This is actually accomplished, however, via the function `` `<-` ``. That is, one could rewrite `x <- 2` as:

```
`<-`(x, 2)
x
```

```
[1] 2
```

Similarly, the + operator calls the underlying function `` `+` ``.

```
`+`(2, 2)
```

```
[1] 4
```

Function call translations, for example, from `2 + 2` to `` `+`(2, 2) ``, are made silently through the **R**-interpreter[1], making it unnecessary to compile **R** code into *executable files* (see Ch 9).

[1] Chambers (2008) describes function evaluation as a three step process: *read* → *parse* → *evaluate*, and refers to the programmatic mechanisms underlying this process as the **R**-evaluator.

8.1.1 `function()` and Function Base Types

Generally speaking, an **R** function –at the risk of sounding repetitive– is a function defined by the function `function()` ☺ . Arguments in **R** functions will be contained in a set of parentheses in the call to `function()` itself. The function contents follow, generally delineated by curly brackets. Thus, we have the form:

```
function.name <- function(arg.1, arg.2,...., arg.n){function contents}
```

Recall from Chapter 2 that there are three **R** base types specific to functions: `closure`, `special`, and `builtin`. Functions of base types `special` and `builtin` are constrained to the *base* package, and include *primitive functions* built into the **R** system, and implemented in C. Types `builtin` and `special` can be distinguished as functions that *do* and *do not* evaluate their arguments, respectively (R Core Team, 2024b).

Example 8.1. The code below allows listing of **R** primitive functions in the *base* package (Chambers, 2008).

```
1 base.objs <- objects("package:base", all = TRUE)
2 prim.objs <- base.objs[sapply(base.objs, function(x) is.primitive(get(x)))]
```

On Line 1, a character vector containing *base* functions named `base.objs` is generated using the function `objects()`. Strings from `base.objs` are used to test if the functions are primitive by using `is.primitive()` as the `FUN` argument in `sapply()`. Boolean outcomes from the test are used to subset `base.objs` into the object `prim.objs`.

The summarization is continued in the chunk below.

```
1 base.types <- split(prim.objs,
2                     sapply(prim.objs,
3                            function(x) typeof(get(x))))
4
5 sapply(base.types, length)
```

```
builtin special
    166      42
```

On Line 1, the function `split()` is used to split data in `prim.objs` into base type categories using `typeof(get())` within `sapply()` (Lines 2 and 3). Numbers of items in these groups are tabulated using `sapply()` again (Line 5). There are currently 166 `builtin` and 42 `special` primitive functions in **R**.

Here are the first 20 `special` primitive functions in the *base* package:

```
base.types$special[1:20]
```

```
 [1] "$"         "$<-"       "&&"        ".Internal" "::"
```

```
 [6] ":::"       "@"         "@<-"       "["         "[["
[11] "[[<-"      "[<-"       "{"         "||"        "~"
[16] "<-"        "<<-"       "="         "break"     "call"
```

Here are the first 20 `builtin` primitive functions:

```
base.types$builtin[1:20]
```

```
 [1] "-"                  "!"                  "!="
 [4] "%%"                 "%*%"                "%/%"
 [7] "&"                  "("                  "*"
[10] "...elt"             "...length"          "...names"
[13] ".C"                 ".cache_class"       ".Call"
[16] ".Call.graphics"     ".class2"            ".External"
[19] ".External.graphics" ".External2"
```

Clearly, primitive functions of base type `special`and `builtin` include conventional operators (with bounding accent grave characters). For example, `` `$` `` has base type `special`,

```
typeof(`$`)
```

```
[1] "special"
```

and `` `+` `` has base type `builtin`.

```
typeof(`+`)
```

```
[1] "builtin"
```

■

Primitive functions generally make calls to the function `.Primitive()`, which, in turn, identifies an underlying C routine used for evaluating the outer function. For example, we see that `` `+` ``, as codified in **R**, calls a C routine identified with `"+"`.

```
`+`
```

```
function (e1, e2)  .Primitive("+")
```

8.1.2 Base Type `closure`

Primitive functions of type `special` or `builtin` cannot be created by users outside of the **R** development core team. Thus, base type `closure` represents the only kind of function **R**-users can actually create and easily modify. The name "closure" refers to the programming style underlying these functions, with each assigned to a particular environment with local internal

objects (see Section 5.4 in Chambers (2008)). Approaches for creating a closure function that calls a user-defined primitive function are considered in Section 9.3.2.

Example 8.2.
Consider the simple homemade function `square.me()`[2].

```
square.me <- function(x){
  x^2
}

square.me(4)
```

```
[1] 16
```

```
typeof(square.me)
```

```
[1] "closure"
```

■

Functions of base type `closure` will have three components.

- The **formals** constitute arguments that control the function. These can be accessed via the function `formals()`. For instance,

```
formals(square.me)
```

```
$x
```

The formals of a function will have a `pairlist` base type.

```
typeof(formals(square.me))
```

```
[1] "pairlist"
```

- The **body** constitutes the actual function code. The function `body()` returns the body of a function as an unevaluated expression.

```
body(square.me)
```

```
{
    x^2
}
```

- Recall that an **environment** is a special **R** storage system, and that only objects in the "current" environment (Section 8.8.1.1.1) can be directly accessed using their object

[2] A number of resources exist for debugging **R** functions, including the function `debug()`. Commonly reported errors for functions containing *tidyverse* code are given at the A future for R website.

names (Section 2.3.3). When a function name is defined at the "top level" in an **R** session (e.g., outside of a package or the body of a function), its environment will be the global environment, `.GlobalEnv`. The global environment is maintained throughout a session and can be saved across sessions using, for instance, the function `save.image()` (Section 2.8.2). Environments of functions can be checked, created, or changed using the function `environment()`. **R** environments are thoroughly considered in Section 8.8.1.

The environment of `square.me()` is the global environment.

```
environment(square.me)
```

```
<environment: R_GlobalEnv>
```

Whereas the environment of the function `mean()` is the *base* package.

```
environment(mean)
```

```
<environment: namespace:base>
```

Functions in *base*, including `mean()`, are accessible, because the *base* package namespace is loaded automatically into a session (along with most of the **R** distribution packages) upon opening **R** (see Section 8.8).

```
isNamespaceLoaded("base")
```

```
[1] TRUE
```

■

Notably, `formals()`, `body()` and `environment()` will all return `NULL` for primitive functions, because these call external C code directly.

```
formals(`+`)
```

```
NULL
```

```
body(`+`)
```

```
NULL
```

```
environment(`+`)
```

```
NULL
```

Example 8.3.
As a biological example, we will create a function for calculating sizes of biological populations under *geometric growth*. The geometric growth equation (Eq. (8.1)) is often used to represent population growth for a species with unlimited resources and non-overlapping generations:

$$f(t) = N_0 \lambda^t, \tag{8.1}$$

where: N_0 = initial number of individuals, λ = the geometric rate of increase, and t = the number of time intervals or generations. We have:

```
Geo.growth <- function(N.0, lambda, t){
  Nt <- N.0 * lambda^t
  Nt
}
```

Note that the function has three arguments: `N.0`, `lambda`, and `t`.

A user of `Geo.growth()` must specify each of these arguments. The first line of code in the function body solves $N_0\lambda^t$. Importantly, the second (last) line of body code specifies the object we actually want returned, `Nt`. Without a "return value" nothing will be returned by the function. If one requires multiple return objects, then one can place them in single suitable container like a list.

To increase clarity, one should place the first curly bracket on same line as the arguments, and place last curly bracket on its own line. Readability can also be improved with the use of tabs and spaces. Note that I have indented lines containing related operations. This distinguishes those lines from the first (argument) line and the end (return) line. Note also that spaces are placed after commas, and before and after operators, including the assignment operator. This is also good general practice for function writing[3].

Below we run the function for different values of `N.0`, `lambda`, and `t`.

```
Geo.growth(N.0 = 100, lambda = 1.2, t = 20)
```

```
[1] 3833.8
```

```
Geo.growth(N.0 = 30, lambda = 0.2, t = 3)
```

```
[1] 0.24
```

```
Geo.growth(N.0 = 30, lambda = 1, t = 3)
```

```
[1] 30
```

■

8.2 Environments within Functions: Global vs. Local Variables

Because **R** objects are lexically scoped (Section 1.4.1.2), object names (including function names) are constrained to particular environments (Section 2.3.3). *Global variables* are objects

[3] A good **R** style guide can be found at: (https://google.github.io/styleguide/Rguide.html).

that exist within the global (session) environment and consequently are broadly accessible, whereas *local variables* are only accessible in particular settings. Objects defined within a function, including arguments, are (generally) local to that function, and thus are accessible only within the body of the function.

Example 8.4.
We see that the object `N.t`, which was defined in the last line of `Geo.growth()`, is local to that function, since it cannot be detected in the global environment.

```
Nt
```

```
Error: object 'Nt' not found
```

Objects local to a function (and their values) can be listed by including a call to `ls.str()` within the function.

```
Geo.growth <- function(N.0, lambda,t){
  Nt <- N.0 * lambda^t
  ls.str()
}

Geo.growth(N.0 = 30, lambda = 1, t = 3)
```

```
lambda :  num 1
N.0 :  num 30
Nt :  num 30
t :  num 3
```

Global variables can be assigned in functions using the super-assignment operator, <<–, although I have found the need for this operator to be rare (but see Section 11.2.1). The *function environment* and the *execution environment* within a function as it runs, are reconsidered in Section 8.8.1.1.1.

```
Geo.growth <- function(N.0, lambda,t){
  Nt <<- N.0 * lambda^t
  Nt
}

g <- Geo.growth(N.0 = 30, lambda = 1, t = 3)
Nt
```

```
[1] 30
```

■

Example 8.5.
The `apply` family of functions for data management, including `apply()`, `tapply()`, `sapply()`

and `lapply()` (Section 4.1.1) allow inclusion of user-defined functions (see Example 8.1 from earlier in this chapter). The function `stan()` below centers and scales (standardizes) outcomes. That is, each element in the dataset is subtracted from its mean, and divided by its standard deviation). We can call `stan()` within `apply()`, using the latter function's third (`FUN`) argument.

```
stan <- function(x){
  (x - mean(x))/sd(x)
}

out <- apply(Loblolly[,1:2], 2, stan)
```

As a consequence of the transformation, columns in the object `out` will the same mean (zero), and the same variance (one)

```
apply(out, 2, mean) # zero with rounding error
```

```
    height        age
1.6687e-16 1.8508e-17
```

```
apply(out, 2, var)
```

```
height    age
     1      1
```

■

Example 8.6.
Below is a function called `stats()` that will simultaneously calculate a large number of distinct summary statistics.

```
1  stats <- function(x, digits = 5){
2    require(asbio)
3    ds <- data.frame(statistics = round(c(length(x), min(x), max(x),
4                                          mean(x), median(x), sd(x), var(x),
5                                          IQR(x), sd(x)/sqrt(length(x)),
6                                          kurt(x), skew(x)), digits))
7    rownames(ds) <- c("n", "min", "max", "mean", "median", "sd",
8                      "var", "IQR", "SE", "kurtosis", "skew")
9    ds
10 }
```

Note that the function contains two arguments (Line one): a call to a numeric data vector, `x`, and the number of significant digits to be used in printing the output. Because `digits` has been given a default value (`digits` = 5), only the first argument needs to be specified by the user. The first line of code in the body of the function (Line 2) indicates that package `asbio` is required

by the function (the package contains the functions `asbio::skew()` and `asbio::kurt()` for calculating the data skew and kurtosis, respectively. In Lines 3-8 a dataframe is created called `ds`. This object has one column called `statistics`, that will contain numeric entries for eleven statistical summaries of `x`. The summaries are rounded to the number of digits specified in `digits`. Lines 7-8 define the row names of `ds`. These are the names of the statistics calculated by the function. The last line of code in the body (Line 9) prints `ds`.

We can readily apply `stats()` to a single numeric column.

```
stats(Loblolly[,1])
```

```
          statistics
n          84.00000
min         3.46000
max        64.10000
mean       32.36440
median     34.00000
sd         20.67360
var       427.39793
IQR        40.89500
SE          2.25568
kurtosis   -1.47347
skew       -0.06434
```

Or apply the function to multiple columns, for instance, by calling by calling `stats()` within `apply()`:

```
apply(Loblolly[,c(1:2)], 2, stats)
```

```
$height
          statistics
n          84.00000
min         3.46000
max        64.10000
mean       32.36440
median     34.00000
sd         20.67360
var       427.39793
IQR        40.89500
SE          2.25568
kurtosis   -1.47347
skew       -0.06434

$age
          statistics
n          84.00000
```

```
min         3.00000
max        25.00000
mean       13.00000
median     12.50000
sd          7.89998
var        62.40964
IQR        15.00000
SE          0.86196
kurtosis   -1.37375
skew        0.18925
```

■

8.3 Useful Functions for Writing Functions

8.3.1 `switch()`

A useful tool for function writing is `switch()`. It evaluates and switches among user-designated alternatives which can be defined in a function argument.

Example 8.7.
The function below switches between five different estimators of location (i.e., estimators of a typical or central value from a sample). These are: the sample mean, a trimmed mean (using 10% trimming), the geometric mean, the median, and Huber's M-estimator. See Chapter 4 in Aho (2014) for details concerning these estimators.

```
location <- function(x, estimator){
  require(asbio)
  switch(estimator,
    mean = mean(x), # arithmetic mean
    trim = mean(x, trim = 0.1), # trimmed mean
    geo = exp(mean(log(x))), # geometric mean (use for means of rates)
    med = median(x), # median
    huber = huber.mu(x), # Huber M-estimator
    stop("Estimator not included"))
}
```

```
location(Loblolly[,2], "geo")
```

```
[1] 10.198
```

```
location(Loblolly[,2], "trim")
```

```
[1] 12.765
```

Important to the function above is the pairing of the `estimator` argument in the overall function (Line 1) and a call to `estimator` in the first argument of `switch` (Line 3). As a final component of `switch` we address the contingency that a location estimator is specified that is not codified in the function. This is done using the `stop()` function with an appropriate message (Line 9).

■

8.3.2 `match.arg()`

It is possible to specify partial matching for argument designations using the function `match.arg()`.

Example 8.8.
For instance, what if we only knew (or only wanted to specify) the first couple letters for the specifiable names in the `estimator` argument in the function, `location() from Examole \@ref(exm:exm-exm-f4)`? We could specify a step like the following.

```
indices <- c("mean", "trim", "geo", "median", "huber")
method <- match.arg(estimator, indices)
```

This is incorporated into `location()` on Lines 3-4 below. Note also the change in the first argument of `switch` from `estimator` (Line 5), which may have incomplete spelling of a location estimator name, to `method`, which will contain the complete index names from `index`.

```
1  location <- function(x, estimator){
2    require(asbio)
3    indices <- c("mean", "trim", "geo", "median", "huber")
4    method <- match.arg(estimator, indices)
5    switch(method,
6      mean = mean(x), # arithmetic mean
7      trim = mean(x, trim = 0.1), # trimmed mean
8      geo = exp(mean(log(x))), # geometric mean (use for means of rates)
9      med = median(x), # median
10     huber = huber.mu(x), # Huber M-estimator
11     stop("Estimator not included"))
12 }
```

Now we could do something like:

```
location(Loblolly[,2],"t")
```

```
[1] 12.765
```

```
location(Loblolly[,2],"h")
```

```
[1] 13
```

■

8.3.3 ...

The triple dot (...) operator[4] allows specification of arguments from an existing function from within another function. As a result, the operator is often used for writing *wrapper functions* (functions intended to remotely control secondary, embedded, functions).

Example 8.9.
Imagine you wished to create a wrapper for the function `plot()` that allowed simultaneous computation and customized plotting of a simple linear regression model. We could do something like:

```
1 plot.reg <- function(x, y, ...){
2   reg <- lm(y ~ x)
3   plot(x, y, ...)
4   abline(reg)
5 }
```

The first two formal arguments `x` and `y` on line Line 1, establish plotting coordinates of points, and define the outcomes for the explanatory and response variables, respectively. The third argument is the triple dot operator (Line 1). In the first line in the body of the function (Line 2) we create a general linear regression model using the function `lm()`. Line 3 creates a plot and, importantly, calls the triple dot operator from the arguments in `plot.reg()`. This allows specification of any possible `plot()` arguments, as arguments within `plot.reg()`. For instance, in the usage of `plot.reg()` below, I specify the *x* and *y* axis labels, a plotting character type, and symbols colors (Fig 8.1). The last line of code (Line 4) plots the regression line.

```
with(Loblolly, plot.reg(age, height, pch = 19, col = as.numeric(Seed),
         ylab = "Height (ft)", xlab = "Age (yrs)"))
```

[4] This operator is not the same as the C-internal ... base type (Section 2.3.6).

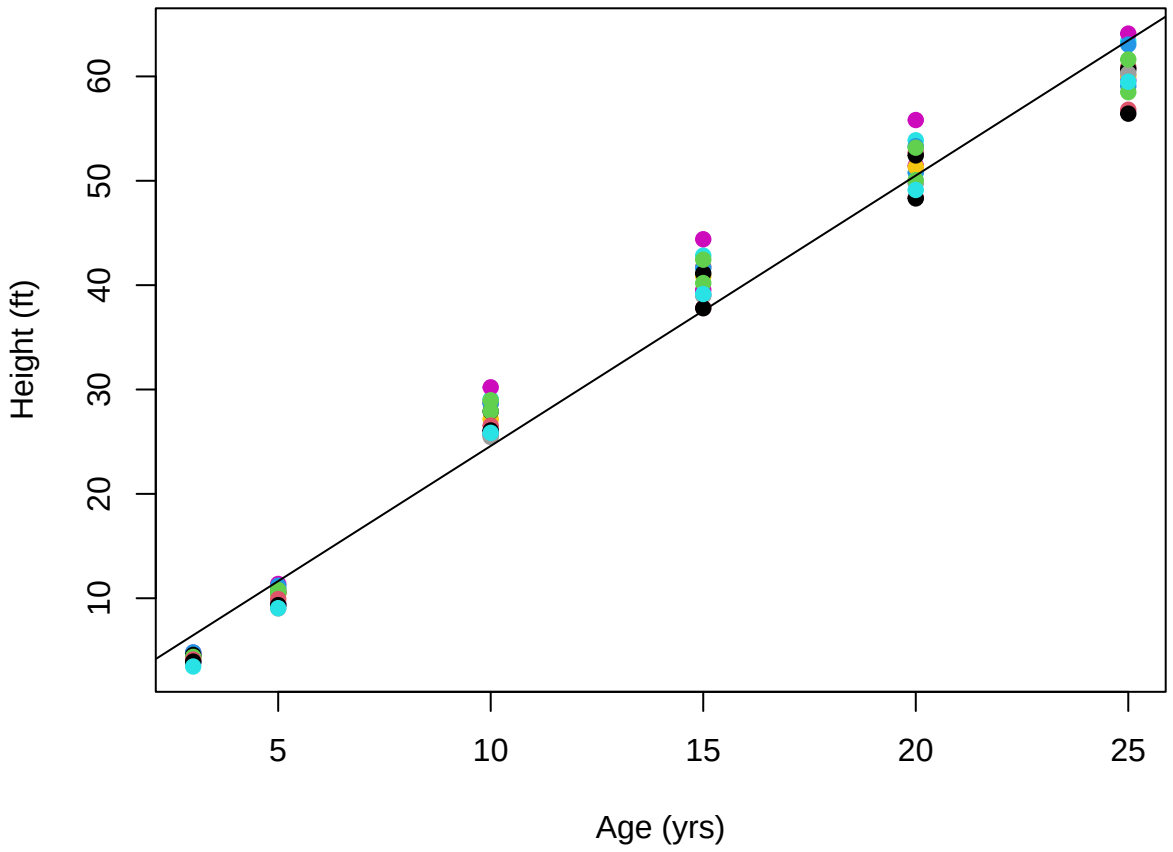


Figure 8.1: Representation of loblolly pine tree height as a function of age. Regression fit overlaid. Seed types are distinguished with colors.

■

8.3.4 `invisible()`

The `invisible()` function can be useful when one wishes to have results computed and saved but not necessarily printed.

Example 8.10.
Assume that we want to retain `plot.reg()` as a plotting function, but wish to have potential access to actual statistical summaries from the regression model. We could rewrite `plot.reg()` as:

```
plot.reg <- function(x, y, plot = TRUE, ...){
  reg <- lm(y ~ x)
  if(plot){ plot(x, y, ...)
  abline(coef(reg))}
  invisible(summary(reg))
}
```

Note that I have added an argument, `plot` (Line 1), to control whether a plot is created via `if(plot)` (Line 3). By suppressing plotting I get no graphics (or text) output:

```
with(Loblolly, plot.reg(age, height, plot = FALSE))
```

However, if I assign a name to the function's output, and print the assigned object, I get summary output for the regression model:

```
lob.model <- with(Loblolly, plot.reg(age, height, plot = FALSE))
lob.model
```

```

Call:
lm(formula = y ~ x)

Residuals:
   Min     1Q Median     3Q    Max
-7.021 -2.167 -0.439  2.054  6.855

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -1.3124     0.6218   -2.11    0.038 *
x             2.5905     0.0409   63.27   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.95 on 82 degrees of freedom
Multiple R-squared:  0.98,  Adjusted R-squared:  0.98
F-statistic: 4e+03 on 1 and 82 DF,  p-value: <2e-16
```

■

8.4 Some Advanced Biometric Examples

R functions can be used to address complex mathematical/statistical problems associated with biological research.

Example 8.11.
Biologists often need to solve systems of dependent differential equations in models describing the propagation of electrochemical signals via action action potentials in neurons (Hodgkin and Huxley, 1952), or models involving species interactions (e.g., competition or predation). For instance, the Lotka-Volterra competition model has the form:

$$\begin{aligned}\frac{dN_1}{dt} &= r_{max1}N_1\frac{K_1 - N_1 - \alpha_{12}N_2}{K_1}\\ \frac{dN_2}{dt} &= r_{max2}N_2\frac{K_2 - N_2 - \alpha_{21}N_1}{K_2}\end{aligned} \tag{8.2}$$

where t denotes time, r_{max1} is the maximum per capita rate of increase (empirical rate) for species 1, and r_{max2} is the empirical rate for species 2, N_1 and N_2 are the number of individuals from species 1 and 2, respectively, K_1 = the carrying capacity for species 1, i.e., the maximum population size for that species, K_2 = the carrying capacity for species 2, α_{12} is the competitive effect of species 2 on the growth of species 1, and α_{21} is the competitive effect of species 1 on the growth rate of species 2.

We first bring in the package *deSolve*, which contains functions for solving ordinary differential equations (ODEs).

```
library(deSolve)
```

We then define starting values for N_1 and N_2 and model parameters.

```
xstart <- c(N1 = 10, N2 = 10)
pars <- c(r1 = 0.5, r2 = 0.4, K1 = 400, K2 = 300, a2.1 = 0.4,
             a1.2 = 1.1)
```

Next, we specify the Lotka-Volterra equations as a function. We will include the argument `time = time` even though it is not explicitly used in the function. This is required by ODE evaluators from *deSolve*.

```
1 LV <- function(time = time, xstart = xstart, pars = pars){
2   N1 <- xstart[1]
3   N2 <- xstart[2]
4     with(as.list(pars),{
5     dn1 <- r1 * N1 * ((K1 - N1 - (a1.2 * N2))/K1)
6     dn2 <- r2 * N2 * ((K2 - N2 - (a2.1 * N1))/K2)
7     res <- list(c(dn1, dn2))
8     })
9 }
```

The most complex part of the function occurs on Lines 4-8 where the system of ODEs in Eq (8.2) is solved. Note the use of `with()` to make the components of the object `pars` accessible between braces on Lines 4 and 8.

The `deSolve::rk4()` function solves simultaneous differential equations using classical Runge-Kutta 4th order integration [Butcher (1987)][^08-ch8-13]. The arguments for `rk4()`

are, in order, the initial population numbers from species 1 and 2, the time frames to be considered, the function to be evaluated, and the parameter values.

```
out <- as.data.frame(rk4(xstart, time = 1:200, LV, pars))
```

The object out contains the number of individuals in species 1 and 2 (N_1 and N_2) for time frames 1-200 (Fig 8.2).

```
plot(out$time, out$N2, xlab = "Time", ylab = "Number of individuals",
     type = "l")
lines(out$time, out$N1, type = "l", col = "red", lty = 2)
legend("bottomright", lty = c(1, 2), legend = c("Species 2", "Species 1"),
col = c(1, 2))
```

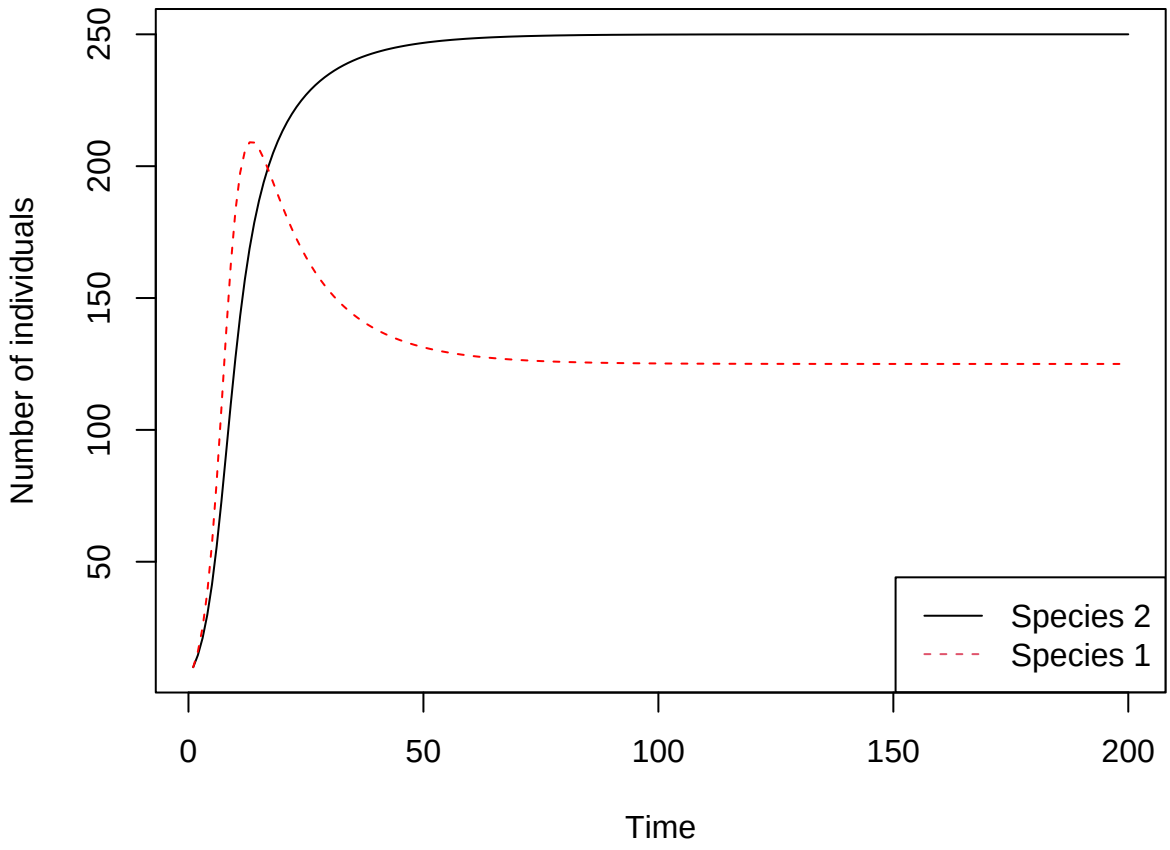


Figure 8.2: Solutions from Lotka Volterra ODEs for $t = \{1, 2, ..., 200\}$. Species 1 and 2 coexist, but at levels appreciably below their carrying capacities as a result of interspecific competition.

■

[^08-ch8-13]: The method of Euler (the simplest method to find approximate solutions to first order equations) can be specified with the function `deSolve::euler()`.

Example 8.12.
Parameter estimation in biostatistics often requires optimization of mathematical functions (finding function minima and maxima). A useful function for this application is `uniroot()`, which searches an interval for the zero root of a function. For instance, many location estimators (those which estimate "central" or "typical" values, e.g., estimators of the true underlying mean) will be the zero root the function:

$$\sum_{i=1}^{n}(x_i - \hat{\mu}) \tag{8.3}$$

where x_i is the ith observation from a dataset, and $\hat{\mu}$ is an estimator of a true location value. We will use `uniroot()` to find a location estimate that provides a zero root for this function.

As data we will use tree heights from the dataframe `loblolly`.

```
data <- Loblolly$height
f <- function(x) sum(data - x)
uniroot(f, c(min(data), max(data)))$root
```

```
[1] 32.364
```

This value is identical to the sample mean of the tree heights.

```
mean(data)
```

```
[1] 32.364
```

Indeed, the sample mean will be always be the zero root of Eq. (8.3). Normally the differences of the data points and the location estimate are squared, preceding summation. Minimizing this function is the process of ordinary least squares.

■

8.5 Loops

Loop functions exist in some form in virtually all programming languages. A "for loop" in **R** is initiated using the function `for()`. The `for` construct requires a specification of three entities

- An index variable, e.g., `i`,
- The statement `in`
- A sequence that the index variable refers to as the loop commences.

Code defining the loop follows, generally delineated by curly brackets. In parallel to function writing it is good style to place the first curly bracket on the same line as the call to for, and to place the last curly bracket on its own line. Thus, we have the basic `for` loop format:

```
for(i in seq){
  loop contents
}
```

In the loop, values of index are used directly or indirectly to specify the ith element of something (e.g., matrix column, vector entry, etc.) as the for loop sequence commences. The replacement/definition process takes place in the "loop contents." For instance,

```
for(i in 1:3){
  print(i)
}
```

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

8.5.1 Extending Scalar Arguments

One application for a loop is to make functions with scalar input arguments amenable to multi-element vector, matrix or dataframe inputs.

Example 8.13.
A library I created, *plant.ecol*, lives only on Github. We can access it with the code:

```
library(devtools)
install_github("moondog1969/plant.ecol")
library(plant.ecol)
```

The package *devtools* facilities building **R** packages from source code, and contains functions, e.g., `install_github()`, for downloading packages from unconventional repositories[5]. The function `plant.ecol::radiation.heatl()` calculates annual incident solar radiation (MJ cm^{-2} yr^{-1}) and heatload, a northern hemisphere thermal index that acknowledges that highest levels of heat loading occur on southwest facing slopes north of the equator. The function requires three arguments for some location of interest: slope, aspect, and north latitude, all measured in degrees.

```
formals(radiation.heatl)
```

```
$slope


$aspect
```

[5] Functions from the *tidyverse* package *usethis* will be useful for setting up passwords and tokens necessary for downloading from some repositories, including Github.

```
$lat
```

The function is designed to accommodate scalar inputs (single values for slope, aspect and lat). We will create a for loop to allow calculation of multiple values from a matrix. For instance, here are potential values for five sites.

```
envdata <- data.frame(slope = c(10, 12, 15, 20, 3),
                      aspect = c(148, 110, 0, 30, 130),
                      latitude = c(40, 50, 20, 25, 45))
```

We first create storage containers for the radiation and heatload results.

```
rad.out <- 1:nrow(envdata) -> hl.out
```

Here is a potential loop for our problem.

```
for(i in 1:length(rad.out)){
  temp <- with(envdata, radiation.heatl(slope[i],
                                        aspect[i],
                                        latitude[i]))
  rad.out[i] <- temp$radiation
  hl.out[i] <- temp$heat.load
}
```

The function was forced to loop around on itself letting `i = 1` during the first loop, `i = 2`, during the second loop, up to `i = 5` on the final loop. Here are the results.

```
cbind(rad.out, hl.out)
```

```
     rad.out  hl.out
[1,] 0.99766 0.94551
[2,] 0.85008 0.77345
[3,] 0.93030 0.96684
[4,] 0.85604 0.83957
[5,] 0.91852 0.90011
```

■

8.5.2 Building on a Previous Result

Another application of a loop is to iteratively build on the results of the previous step(s) in the loop.

Example 8.14.
Consider the following function that counts the number of even entries in a vector of integers.

```
evencount <- function(x){
  res <- 0
  for(i in 1 : length(x)){
    if(x[i] %% 2 == 0) res <- res + 1
    }
  res
}
```

Recall from Ch 2 that %% is the modulus operator in **R**. That is, it finds the remainder in division. By definition the remainder of any even integer divided by two will be zero. At each loop iteration the function adds one to the numeric object `res` if the current integer in the loop is even (if it has remainder zero if divided by two).

```
evencount(1:3)
```

```
[1] 1
```

```
evencount(c(1,2,3,4,10))
```

```
[1] 3
```

■

8.5.3 Summarizing Categorical Variables

A third loop application is the summarization of data with respect to levels in a categorical variable.

Example 8.15.
As an example we will create statistical summaries for `height` and `age` for each `Seed` type in the `Loblolly` dataset, using the `stats` function I created earlier. I first create an empty list to hold my result:

```
result <- list()
```

```
for(i in levels(Loblolly$Seed)){
    temp <- Loblolly[,1:2][Loblolly$Seed ==i,]
    result[[i]] <- as.data.frame(apply(temp, 2, stats))
    names(result[[i]]) <- c("Age","Height")
}
```

`Loblolly$Seed`. This is specified with: `for(i in levels(Loblolly$Seed))`. Note that on Line 2, the first two columns of the `Loblolly` dataset are subset by levels in `Seed`. Here are the results for seed type 305.

```
result$'305' # "name" of one of the 14 Loblolly seed types
```

```
             Age   Height
n          6.0000  6.00000
min        4.7900  3.00000
max       64.1000 25.00000
mean      35.1150 13.00000
median    37.3050 12.50000
sd        23.9271  8.60233
var      572.5056 74.00000
IQR       36.8850 12.50000
SE         9.7682  3.51188
kurtosis  -1.8479 -1.47809
skew      -0.1613  0.25449
```

■

8.5.4 Looping Without `for()`

Looping in **R** is also possible using other general styles of Algol-like[6] languages (e.g., C, C++, Pascal, and Fortran). This is accomplished with the constructs `while()`, `repeat`, and `break`.

Example 8.16.
Consider an example in which 2 is added to a base number until the updated number becomes greater than or equal to 10: We have:

```
i <- 1
while (i < 10) i <- i + 2
i
```

```
[1] 11
```

Or, to explicitly track the loop, we could use:

```
i <- 1; out <- i

while(TRUE){
    j <- i + 2
    out <- paste(out, j, sep = ",")
    i <- j
    if (i > 9) break
}
```

[6] Algol (Algorithmic language) computer languages arose in the late 1950s from the language ALGOL 68. Important examples include Pascal, C, and Fortran.

```
out
```

```
[1] "1,3,5,7,9,11"
```

Or, more simply

```
i <- 1; out <- i

repeat{
    j <- i + 2
    out <- paste(out, j, sep = ",")
    i <- j
    if (i > 9) break
}

out
```

```
[1] "1,3,5,7,9,11"
```

Here `i` took on values 1, 3, 5, 7, 9, and 11 as the loop commenced (this information is accumulated in the object `out`). When `i` equaled 11, the condition for continuation of the loop failed and the loop was halted.

■

Wickham (2019) recommends `for` loops over `while()` loops, and `while()` loops over `repeat` procedures.

CAUTION!

Some care should be exercised with `while()` and `repeat` since infinite loops will result if impossible breaks are specified.

8.5.5 Final Looping Considerations

Despite their potential usefulness, loops can run slowly in **R**, because it is an interpreted language (see Ch 9). Loops can often be avoided altogether. For example, one could rewrite the earlier `evencount()` function as:

```
evencounti <- function(x){
  out <- ifelse(x %% 2 == 0, 1, 0)
  sum(out)
}
# even outcomes from a random Poisson process
evencounti(rpois(1000, 2))
```

```
[1] 503
```

This increases efficiency, as documented by the function `system.time()`:

```
system.time(evencount(1:1000000))
```

```
   user  system elapsed
   0.19    0.03    0.25
```

```
system.time(evencounti(1:1000000))
```

```
   user  system elapsed
   0.07    0.00    0.07
```

Loops can often be run more efficiently using the `apply()` family of functions (see animation examples using `lapply()` in Chs 6 and 7).

If loops are necessary, speed is an issue, and use of alternative approaches (e.g., `lapply()`) is awkward or suboptimal, one can call a compiled C, C++, or Fortran script from within **R** to run the loop. This topic is addressed further in Ch. 9.

8.6 Functional Programming

In functional programming one uses a declarative programming style that applies "pure" (often argument-less) functions[7]. Binary or *infix* operations require exactly two operands, and provide excellent examples of functional programming. The primitive functions `` `+` ``, `` `-` ``, `` `*` `` (Section 8.1) are binary operator functions. When more than two operands are supplied, the functions still work in pairs. Thus,

```
`+`(1,`+`(2,3))
```

```
[1] 6
```

is equivalent to

```
1 + 2 + 3
```

```
[1] 6
```

One can create personalized operator functions using the syntax: `` `% operator name %` `` or `"% operator name %"`.

Example 8.17.
It might be useful to have an operator-style function for computing cumulative sums of individual numbers (although `cumsum()` does this already for numerical or complex objects). We

[7] Famous functional programming languages include *Lisp*, *Scheme*, *F#*, and *Haskell*.

will call our new operator %+%.

```
`%+%` <- function(a,b){c(a, a + cumsum(b))}

2.1 %+% 7.4
```

```
[1] 2.1 9.5
```

To make the operator work for more than two numbers, the second operand must be a multi-element numeric object.

```
2.1 %+% c(2.6, 1.5, 6)
```

```
[1]  2.1  4.7  6.2 12.2
```

■

Useful **R** functions for functional programming include `Reduce()`, `Filter()`, `Find()`, `Map()`, `Negate()`, and `Position()`.

Example 8.18.
As a more applied example, recall from Section 4.2.8 that the %in% operator can be used to indicate if there is a match (or not) for its left operand. This does not clarify, however, how one might specify *not* %in%. The operations !%in% and %!in%, for example, do not work. The code below creates a %!in% operator using the function `Negate()`.

```
`%!in%` <- Negate(`%in%`)
```

We apply `%!in%` below to subset bacterial phylum names.

```
big <- c("Abawacabacteria", "Absconditabacteria", "Acidobacteriota",
         "Actinomycetota", "Aminicenantes", "Atribacterota",
         "Aquificota", "Azambacteria")
small <- c("Acidobacteriota", "Actinomycetota")

w <- which(big %!in% small)
big[w]
```

```
[1] "Abawacabacteria"    "Absconditabacteria" "Aminicenantes"
[4] "Atribacterota"      "Aquificota"         "Azambacteria"
```

■

8.7 Functions with Classes and Methods

R object classes can have particular *methods* for plotting, printing, and summarization. Following a relatively simple series of steps, these methods can be implemented using generic function names, i.e., `plot()`, `print()`, `summary()`. For example, the function `lm()` creates objects of class `lm`.

```
model <- lm(height ~ age, data = Loblolly)
class(model)
```

```
[1] "lm"
```

There are specific summary, print, and plot methods for an object of class `lm`. Code for these methods can be viewed by typing `stats:::summary.lm`, `stats:::print.lm`, and `stats:::plot.lm`, respectively. The `stats:::print.lm` will be called automatically to print an object of class `lm`. For instance,

```
print(model)
```

```
Call:
lm(formula = height ~ age, data = Loblolly)

Coefficients:
(Intercept)          age
      -1.31         2.59
```

or, more simply,

```
model
```

```
Call:
lm(formula = height ~ age, data = Loblolly)

Coefficients:
(Intercept)          age
      -1.31         2.59
```

Here are 20 out of the more than 500 functions on my workstation that can be called with `print()`, depending on the class of the object that is being printed.

```
methods(print)[1:20]
```

```
 [1] "print,ANY-method"             "print,diagonalMatrix-method"
 [3] "print,modelMatrix-method"     "print,sparseMatrix-method"
 [5] "print.aareg"                  "print.abbrev"
 [7] "print.abuocc"                 "print.acf"
 [9] "print.activeConcordance"      "print.addtest"
```

```
[11] "print.AES"                    "print.agnes"
[13] "print.all_vars"               "print.allPerms"
[15] "print.anosim"                 "print.anova"
[17] "print.Anova"                  "print.anova.gam"
[19] "print.anova.lme"              "print.anova.loglm"
```

Importantly, we are not limited to the pre-existing object classes in **R** (e.g., `lm`, `numeric`, `factor`, etc.). Instead, we can create user-defined classes for function output. These classes can also have methods for plotting, printing, and summarization.

8.7.1 S3 and S4

R has two-main approaches for developing OOP classes: *S3*, and *S4* [8] Wickham (2019) notes:

> *"S3 allows your functions to return rich results with user-friendly display and programmer-friendly internals"*

and

> *"S4 is a rigorous system that forces you to think carefully about program design. It's particularly well-suited for building large systems that evolve over time and will receive contributions from many programmers."*

S3 methods tend to be easier to develop than S4 methods, and this approach is recommended for most applications in **R**. The amenability of S4 for interfacing with multiple programmers explains why this approach is required for contribution to the highly collaborative Bioconductor project. S4 OOP classes and their associated methods are implemented via the **R**-distribution package *methods*. I focus on S3 methods here, but briefly consider S4 methods.

8.7.1.1 S3

S3 classes are created using the function `class()`.

```
ISU <- list(name = "Idaho State University", n.students = 12000,
            founded = 1905)
class(ISU) <- "univ"
ISU
```

```
$name
[1] "Idaho State University"

$n.students
[1] 12000

$founded
```

[8] S1 and S2 OOP classes do not exist. S3 and S4 were named according to the versions of S that they accompanied. S versions 1 and 2 didn't have an OOP framework.

```
[1] 1905

attr(,"class")
[1] "univ"
```

Note that the object `ISU` has the class attribute `"univ"`. The *sloop* package contains functions to help distinguish OOP class frameworks. The function `sloop::otype()` can be used to determine if an object is S3, S4, RC, or R6.

```
library(sloop)
otype(ISU)
```

```
[1] "S3"
```

The object `ISU` is S3.

An S3 (or S4 object) is fairly useless without associated methods. Here is a simple print method for an object of class `univ`, i.e., a list with components: `name`, `founded`, and `n.students`.

```
print.univ <- function(x){
  cat(x$name, " was founded in ", x$founded, ",\nand has an enrollment of ",
      x$n.students, " students.", sep = "")
}
```

This dramatically changes the way the object `ISU` is printed.

```
ISU
```

```
Idaho State University was founded in 1905,
and has an enrollment of 12000 students.
```

Functions useful in creating print methods include `cat()` (used above) and `structure()`. The function `cat()` concatenates text into a single character vector, and prints the results. As a simple example, in the code below we bind the string "`iteration = ''`, to a random integer generated from a Poisson distribution `rpois(1, 10)`, and apply a double line break `"\n\n"`.

```
cat("iteration = ", rpois(1, 10), "\n\n", sep = "")
```

```
iteration = 6
```

The function `structure()` allows one to assign an `attribute` set to data.

```
structure(.Data = 1:6, dim = 2:3)
```

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

```
structure(.Data = 1:6, names = LETTERS[1:6])
```

```
A B C D E F
1 2 3 4 5 6
```

We can identify methods specific to some class using the function `sloop::ftype`.

```
ftype(print.univ)
```

```
[1] "S3"     "method"
```

Thus, `print.univ` is a method function. It provides customized printing for objects of class `univ`.

Example 8.19.
Here is a more complex example in which an output object from a *function* has an S3 class. The function `asbio::pairw.anova` is used for adjusting p-values resulting from multiple pairwise comparisons following an omnibus ANOVA (ANalysis Of VAriance). Objects generated by the function have class `pairw` and are S3.

```
eggs <- c(11,17,16,14,15,12,10,15,19,11,23,20,18,17,27,33,22,26,28)
trt <- factor(rep(1:4, c(5,5,4,5)))

library(asbio)
tukey <- pairw.anova(y = eggs, x = trt)
class(tukey)
```

```
[1] "pairw"
```

```
otype(tukey)
```

```
[1] "S3"
```

Objects of class `pairw` have both print and plot methods.

```
ftype(print.pairw) # print method for class pairw
```

```
[1] "S3"     "method"
```

```
ftype(plot.pairw) # plot method for class pairw
```

```
[1] "S3"     "method"
```

Here is the actual `print()` method's output:

```
tukey
```

```
95% Tukey-Kramer confidence intervals
```

```
         Diff     Lower    Upper  Decision Adj. p-value
mu1-mu2   1.2  -4.71976  7.11976    FTR H0     0.935298
mu1-mu3  -4.9 -11.17885  1.37885    FTR H0      0.15489
mu2-mu3  -6.1 -12.37885  0.17885    FTR H0     0.058287
mu1-mu4 -12.6 -18.51976 -6.68024 Reject H0     0.000101
mu2-mu4 -13.8 -19.71976 -7.88024 Reject H0      3.7e-05
mu3-mu4  -7.7 -13.97885 -1.42115 Reject H0     0.014218
```

This same method is used for other objects from *asbio* whose output has class `pairw`. These include objects from functions providing pairwise comparisons of factor levels following an omnibus Friedman's test[9], `pairw.fried()`, and an omnibus Welch's test[10], `pairw.oneway()`.

```
welch <- pairw.oneway(y = eggs, x = trt)
welch
```

```
95% Welch adjusted confidence intervals

         Diff     Lower    Upper  Decision Adj. p-value
mu1-mu2   1.2  -3.39503  5.79503    FTR H0      0.55425
mu1-mu3  -4.9    -8.991   -0.809    FTR H0     0.068803
mu2-mu3  -6.1 -11.06986 -1.13014    FTR H0     0.068803
mu1-mu4 -12.6 -17.53547 -7.66453 Reject H0    0.0032699
mu2-mu4 -13.8 -19.36022 -8.23978 Reject H0    0.0027003
mu3-mu4  -7.7 -12.95069 -2.44931 Reject H0      0.04229
```

■

One can often identify a *method* function by the presence of a period character, `.`, in the function name. For instance, `asbio::plot.pairw()` is a plotting method for objects of class `pairw`. Unfortunately, this is not always true. For example, my function `asbio::parw.anova()` is not a methods function, although the function itself has two methods: `asbio::plot.pairw()`, `asbio::print.pairw()` (Example 8.19).

Viewing underlying code for S3 methods functions may require use of the double colon `::` or triple colon operator `:::`, even if the namespace of the method function is loaded.

Example 8.20.
For instance functions from the *utils* package are loaded automatically upon opening **R**. However, the *utils* function `Bibtex`, used for converting character vectors to BibTeX or LaTeX markup, requires requires `:::` for code depiction.

```
utils::print.Bibtex
```

```
Error: 'print.Bibtex' is not an exported object from 'namespace:utils'
```

[9] Friedman's test is a non-parametric alternative to an ANOVA with a blocking variable.

[10] Welch's test, implemented using `oneway.test`, allows heteroscedasticity among factor levels when comparing factor level means.

```
utils:::print.Bibtex

function (x, prefix = "", ...)
{
    writeLines(paste0(prefix, unclass(x)), ...)
    invisible(x)
}
<bytecode: 0x0000012db9eb4468>
<environment: namespace:utils>
```

■

The reason for this "secrecy" is that internal methods are likely to be specific to classes associated with particular packages, and are unlikely to be useful outside of that context.

Example 8.21.
In this extended exercise we will fashion an advanced function, with an S3 class and create associated methods, using a number of approaches discussed so far in this chapter.

In ecological studies, α-diversity measures the level of species evenness and richness within individual plots in a dataset. The most widely used alpha diversity indices are Simpson's index, D_1, and the Shannon-Wiener index, H'.

$$D_1 = 1 - \sum_{i=1}^{S} p_i^2 \tag{8.4}$$

$$H' = -\sum_{i=1}^{S} p_i \ln p_i \tag{8.5}$$

where S denotes the number of species, and p_i is the proportional abundance of the ith species, $i = 1, 2, \ldots, S$.

Here are features I want my advanced α-diversity function to have:

- Arguments specifying (1) a dataset for analysis, and (2) the type of α-diversity we want calculated. So, two arguments.
- A function capable of handing summaries of communities for a single site, whose data will be a single numeric vector, and dataframes describing abundances of taxa at multiple sites.
- Assignment of correct names of sites (if any) to results.
- Partial matching of diversity method names using `arg.match`.
- An S3 class.
- Invisible components, appropriate for class print and plot methods.

```
alpha.div <- function(x, method = "simpson"){
  if(is.data.frame(x)) rn <- rownames(x) else {
```

```
3     if(ncol(as.matrix(x) == 1)) rn = noquote("") else
4       rn = 1:nrow(as.matrix(x))
5   }
6
7   indices <- c("simpson", "shannon"); method <- match.arg(method, indices)
8
9   x <- as.matrix(x)
10
11  prop <- function(x){
12      if(ncol(x) == 1) out <- x/sum(x)
13      else
14      out <- apply(x, 1, function(x) x/sum(x))
15      out
16    }
17
18    p.i <- prop(x)
19
20  simp <- function(x, p.i){
21      if(ncol(x) == 1) D <- 1 - sum(p.i^2)
22      else
23      D <- 1 - apply(p.i^2, 2, sum)
24      D
25    }
26
27  shan <- function(x, p.i){
28      if(ncol(x) == 1) H <- -sum(p.i[p.i > 0] * log(p.i[p.i > 0]))
29      else
30      H <- apply(p.i, 2, function(x)-sum(x[x != 0] * log(x[x != 0])))
31      H
32    }
33
34  div <- switch(method,
35                simpson = simp(x, p.i),
36                shannon = shan(x, p.i))
37
38    out <- list(p.i = p.i, rn = rn, method = method, div = div)
39  class(out) <- "a_div"; invisible(out)
40  }
```

Below is a breakdown of important components of the function `alpha.div()` above.

- In the arguments (Line 1), x is assumed to be either 1) a dataframe of taxa abundances at sites, with sites in rows (identified by row names) and taxa in columns, or 2) a numeric vector containing abundances of distinct taxa at a single site.
- In the first lines of code in the function *body* (Lines 2-4), the function attempts to obtain

site names from `x`. If `x` is a dataframe, this is done using `rn <- rownames(x)`. If `x` is a vector describing a single site, distinguishing site names is probably not important, hence the code `rn = noquote("")`.

- The `alpha.div()` function contains three sub-functions: `prop()` (Lines 11-16), which allows computation of p_i, and is used to create the object `p.i`, `simp()` (Lines 20-25), which calculates Simpson's diversities, and `shan()` (Lines 27-32), which calculates Shannon-Weiner diversities. The latter function contains exception handling steps for taxa abundances of zero which will be undefined in Eq (8.5). For instance, `p.i[p.i > 0]` on Line 28, and `x[x != 0]` on Line 30.
- Partial matching of diversity method names (i.e., `"simpson"` and `"shannon"`) is facilitated through the function `match.arg()`.
- Switching of diversity methods is done via `switch()` (Lines 34-36).
- The function output is a list named `out`, which contain four objects: the proportional abundances of taxa, the rownames of `x` (i.e, the site names), the diversity method used, and the actual calculated diversities (Line 38).
- In the last line of body code, `out` is assigned to the user-defined class `a_div` and made invisible.

Here we apply the function to the dataset `varespec` from the library *vegan*.

```
library(vegan)
data(varespec)
v.div <- alpha.div(varespec)
class(v.div)
```

```
[1] "a_div"
```

```
otype(v.div)
```

```
[1] "S3"
```

Printing the output object `v.div` results in a rather messy rendering of a list, prompting the creation of an `a_div` print method. Our `print.a_div()` function will succinctly and effectively summarize results from `alpha.div` while allowing access to additional (invisible) information.

Our function for printing can be relatively simple.

```
1 print.a_div <- function(x, digits = 5){
2   method <- ifelse(x$method == "simpson", "Simpson",
3                    "Shannon-Weiner")
4   cat(method, " diversity:", "\n", sep = "")
5   rq <- structure(x$div,  names = x$rn)
6   print(rq, digits = digits)
7   invisible(x)
8 }
```

- The required argument, `x` in `print.a_div` (Line 1), will be an object of class `a_div`,

created by the function `alpha.div()`, e.g., the object `v.div`. Recall that this is a list containing multiple objects.

- The object `x$method` is used create a tidy text summary of the diversity method used (Lines 2-3). This string is combined with a another string and printed with a line break in: `cat(method, " diversity:", "\n", sep = "")` (Line 4).
- The actual diversities are printed with the help of the function `structure()` on Lines 5-6.

```
print(v.div)
```

```
Simpson diversity:
     18      15      24      27      23      19      22      16      28
0.82171 0.76276 0.78101 0.74414 0.84108 0.81819 0.80310 0.82477 0.55996
     13      14      20      25       7       5       6       3       4
0.81828 0.82994 0.84615 0.83991 0.70115 0.56149 0.73888 0.64181 0.78261
      2       9      12      10      11      21
0.55011 0.49614 0.67568 0.50261 0.80463 0.85896
```

Output from `alpha.div()` can also be used for plotting. Here is a plot function for objects of class `a_div`.

```
1  plot.a_div <- function(x, plot.RAC = FALSE){
2    require(ggplot2)
3    margin_theme <- function(){
4    theme(axis.title.x = element_text(vjust=-5),
5          axis.title.y = element_text(vjust=5),
6          plot.margin = margin(t = 7.5, r = 7.5,
7                               b = 20, l = 15))
8    }
9
10   ptype1 <- function(){
11   spi <- apply(x$p.i, 2, function(x)sort(x, decreasing = TRUE))
12   sspi <- data.frame(p.i = stack(as.data.frame(spi))[,1])
13   sspi$Rank <- rep(1:nrow(x$p.i), ncol(x$p.i))
14   sspi$Site <- rep(x$rn,  each = nrow(x$p.i))
15     ggplot(sspi, aes(y = p.i, x = Rank, group = Site)) +
16       geom_line(aes(y = p.i, x = Rank, colour = Site), alpha = .4) +
17       ylab(expression(italic(p[i]))) +
18       theme_classic() + margin_theme()
19   }
20
21   ptype2 <- function(){
22       diversity <- data.frame(div =  x$div, Site = factor(x$rn))
23       method <- ifelse(x$method == "simpson", "Simpson diveristy",
24                        "Shannon-Weiner diveristy")
```

```
25        ggplot(diversity) +
26        geom_bar(aes(y = div, x = Site, fill = div), show.legend = FALSE,
27                 stat = "identity") +
28        theme_classic() +
29        margin_theme() +
30        ylab(method) + xlab("Site")
31    }
32    if(plot.RAC)  ptype1() else ptype2()
33  }
```

- The plot method allows the creation of two distinct types of ggplots by calling distinct sub-functions, `ptype1()` and `ptype2()` via the argument `plot.RAC` (Line one).
- Barplots of site diversities are produced by using the default `plot.RAC = FALSE` which will run the function `ptype2()` on Lines 21-31 (Fig 8.3)
- Rank abundance curves (RACs) are created by specifying `plot.RAC = TRUE` which runs the function `ptype1()` on Lines 10-19 (Fig 8.4). RAC plots allow graphical expressions of both taxa richness and evenness, and may even provide insights regarding resource exploitation in community (Magurran, 1988).

```
plot(v.div)
```

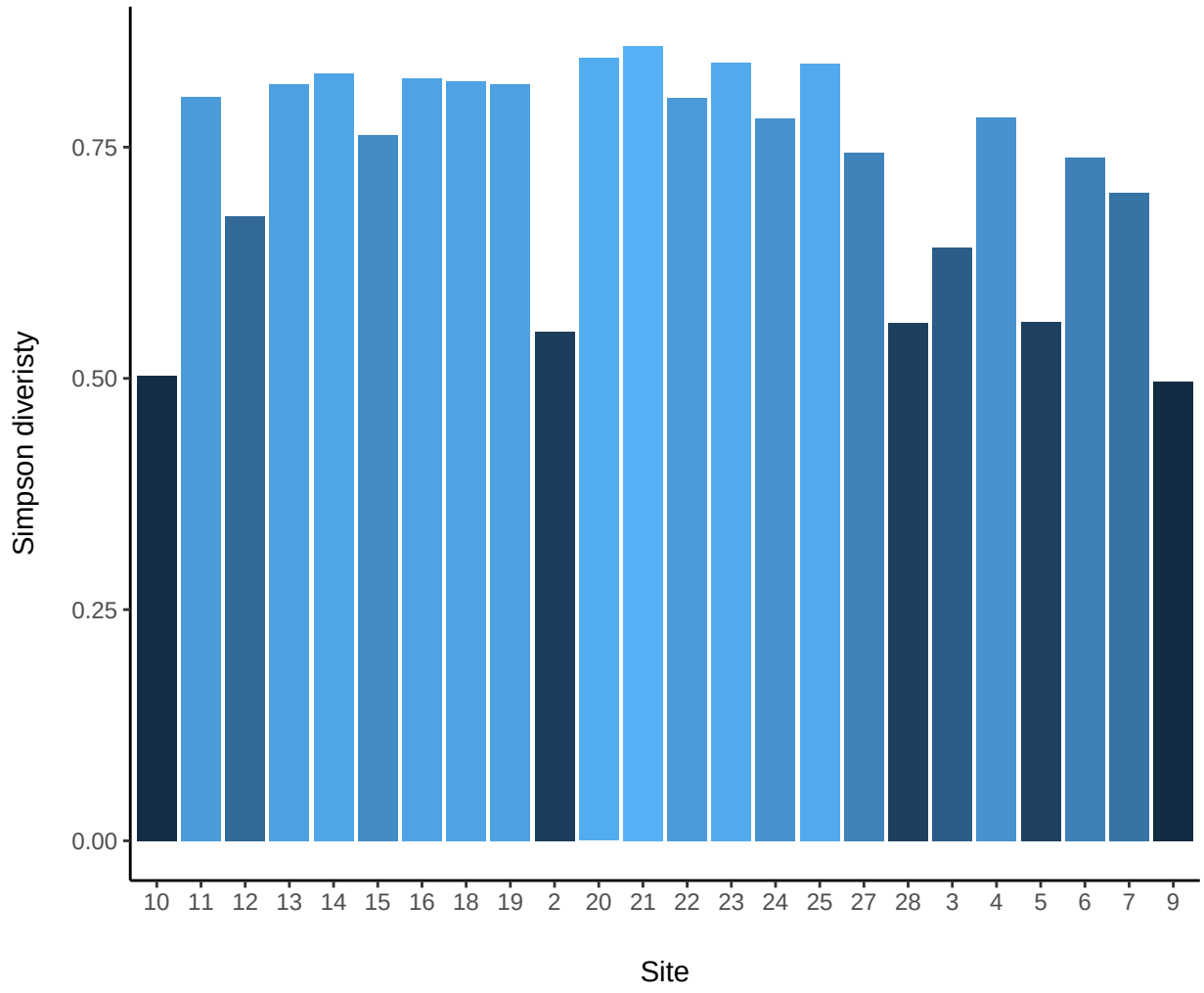


Figure 8.3: Barplot of site diversities from the `vegan::varespec` data. Note that bar colors are varied using the diversities themselves, i.e., `fill = div`.

```
plot(v.div, plot.RAC = TRUE)
```

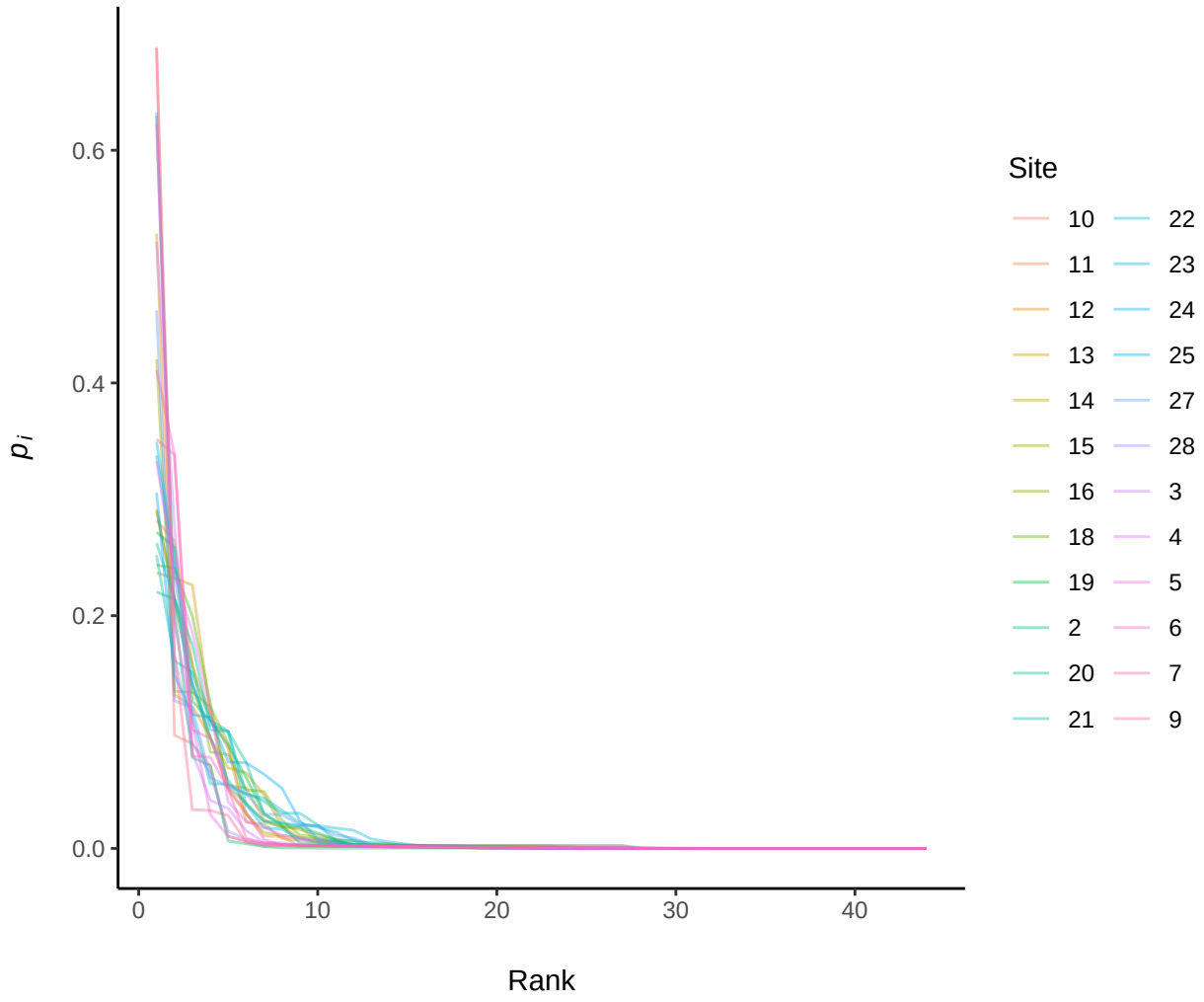


Figure 8.4: Rank abundance curves of the `vegan::varespec` dataset. Line lengths indicate species richness. Larger negative slopes indicate less species evenness.

■

8.7.1.2 S4

An S4 class is defined using the function `setClass()`. Unlike S3 objects and classes, S4 class components, i.e., *slots*, must be defined in `setClass()`, along with the sub-classes of those components. Here I create an S4 class called `univ` for comparison to the S3 class `univ` created in the previous section.

```
setClass("univ", slots = list(name = "character", n.students = "numeric",
                              founded = "numeric"))
```

The S4 class `univ` will have three slots: `name`, `n.students`, and `founded`. S4 objects are created using the `new()` function.

```
ISU <- new(Class = "univ", name = "Idaho State University",
           n.students = 12000, founded = 1905)
```

The object ISU has the S4 class univ.

```
class(ISU)
```

```
[1] "univ"
attr(,"package")
[1] ".GlobalEnv"
```

```
typeof(ISU)
```

```
[1] "S4"
```

```
otype(ISU)
```

```
[1] "S4"
```

Here is the structure of the object:

```
str(ISU)
```

```
Formal class 'univ' [package ".GlobalEnv"] with 3 slots
  ..@ name      : chr "Idaho State University"
  ..@ n.students: num 12000
  ..@ founded   : num 1905
```

Just as components of a list are accessed using $, the slots of an S4 object are accessed using @.

```
ISU@founded
```

```
[1] 1905
```

Or with the function slot().

```
slot(ISU, "founded")
```

```
[1] 1905
```

We set S4 methods using the function setMethod(). Here is an S4 show() method (analogous to S3 print()) for objects of class univ.

```
setMethod("show",
"univ",
function(object) {
 cat(object@name, "was founded in", object@founded,
     "and has an enrollment of",
     object@n.students, "students.", sep = " ")
```

```
}
)

ISU
```

```
Idaho State University was founded in 1905 and has an enrollment of 12000 students.
```

Example 8.22.
As a real-world S4 example, the function `stats4::mle()` estimates parameters for probability density functions using the method of maximum likelihood. Below we estimate the rate parameter for a Poisson distribution, λ, based on a sample of count data.

```
# count data
y <- c(26, 17, 13, 12, 20, 5, 9, 8, 5, 4, 8)

# MLE for lambda in a Poisson distribution given data in y
nLL <- function(lambda) - sum(dpois(y, lambda, log = T))
# mle finds negative log-likelihoods
fit0 <- stats4::mle(nLL, start = list(lambda = 5),
                    nobs = length(y))
```

The MLE for the Poisson rate parameter, from data outcomes in `y`, is approximately 11.545. Notably, this is equal to the sample mean of `y`.

```
fit0
```

```
Call:
stats4::mle(minuslogl = nLL, start = list(lambda = 5), nobs = length(y))

Coefficients:
lambda
11.545
```

```
mean(y)
```

```
[1] 11.545
```

The class of `fit0` is `mle`. The class has an S4 designation.

```
class(fit0)
```

```
[1] "mle"
attr(,"package")
[1] "stats4"
```

```
otype(fit0)
```

```
[1] "S4"
```

The slot structure of an object from class `mle` is complex:

```
str(fit0)
```

```
Formal class 'mle' [package "stats4"] with 10 slots
  ..@ call     : language stats4::mle(minuslogl = nLL, start = list(lambda = 5), nobs = length(y))
  ..@ coef     : Named num 11.5
  .. ..- attr(*, "names")= chr "lambda"
  ..@ fullcoef : Named num 11.5
  .. ..- attr(*, "names")= chr "lambda"
  ..@ fixed    : Named num NA
  .. ..- attr(*, "names")= chr "lambda"
  ..@ vcov     : num [1, 1] 1.05
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : chr "lambda"
  .. .. ..$ : chr "lambda"
  ..@ min      : num 42.7
  ..@ details  :List of 6
  .. ..$ par        : Named num 11.5
  .. .. ..- attr(*, "names")= chr "lambda"
  .. ..$ value      : num 42.7
  .. ..$ counts     : Named int [1:2] 14 8
  .. .. ..- attr(*, "names")= chr [1:2] "function" "gradient"
  .. ..$ convergence: int 0
  .. ..$ message    : NULL
  .. ..$ hessian    : num [1, 1] 0.953
  .. .. ..- attr(*, "dimnames")=List of 2
  .. .. .. ..$ : chr "lambda"
  .. .. .. ..$ : chr "lambda"
  ..@ minuslogl:function (lambda)
  ..@ nobs     : int 11
  ..@ method   : chr "BFGS"
```

■

8.7.2 RC, R6 and others

Aside from S3 and S4, Wickham (2019) discusses several other less widely-used OOP approaches including: R.oo (Bengtsson, 2003), proto (Grothendieck et al., 2016), and particularly, R6 (Chang, 2025) and RC (for Reference Classes). RC is underlain by S4, and thus requires the *methods* package. R6, R.oo, and proto require the *R6*, *R.oo*, and *proto* contributed packages, respectively.

Under RC and R6 syntax, generics (e.g., `plot()` or `print()`) are not used. Instead, methods reflect a *member function* grammar as implemented in Java, C++ (see Section 9.3.1.2), and Python (Section 9.5.6). For an RC or R6 object `foo`, the associated method `bar` would be applied using the syntax: `foo$bar()`. Additionally, RC and R6 methods allow *method chaining*, thus facilitating pipe operations common to the *tidyverse* (Ch 5). RC Reference Classes are generated using `base::setRefClass()`. R6 object classes and methods require the function `R6::R6Class()`.

R6 methods are not S4, and do not use copy-on-modify semantics, making them useful for

handling model objects that exist independently of **R** (Wickham, 2019). Reflecting Java and C++ syntax, R6 allows the creation of *private fields* that can only be accessed from within the class.

8.8 How Functions Work: Environments, Promises, and Function Evaluation

Understanding the process of function evaluation is an important step in acquiring a deeper understanding of **R** because– as noted in the Introduction to this chapter: "Everything that happens in **R** is a function call." To clarify function evaluation, however, a number of other processes require some introduction. These include environmental *search paths* that allow to **R** to correctly identify and utilize functions and other objects, and programmatic *promises* with *lazy evaluation* that reduce memory usage.

8.8.1 Environments

An **R** environment is a specialized storage container consisting of: 1) a collection of named objects, and 2) a pointer to another *enclosing* environment. We have already learned that: 1) All **R** objects, including functions, require an environment (Section 8.1.2), 2) an **R** session (and objects created in that session) are located in the *global environment*, `R_GlobalEnv`, 3) only objects in the *current environment* (Section 8.8.1.1.1) will be accessible by name (Section 2.3.3), and 4) objects created within a function are local to an enclosing function environment, unless defined by the super assignment operator (Section 8.2). An **R** environment is very similar to a list, with three important exceptions (Wickham, 2019):

- First, the name of every object in the environment must be unique. This is *not* true for a list:

```
demolist <- list("a" = 1, "a" = 2)
names(demolist)
```

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

  although it *is* true for a dataframe.

```
demodf <- data.frame("a" = 1, "a" = 2)
names(demodf)
```

```
[1] "a"   "a.1"
```

- Second, despite that fact that numerical referencing is available for all basic **R** data storage types…

```
demolist[[2]]
```

```
[1] 2
```

the names of objects in an environment are *not* ordered, and thus, they are not accessible with numeric identifiers.

- Third (essentially), every environment will have an enclosing "parent" environment.

The package *rlang* contains several useful functions for creating and exploring environments. For example, the function `rlang::env()` can be used to create a new environment (within the *current environment*).

Example 8.23.
Here I create an environment, `e1`, and insert some objects containing snippets of bioinformatics data.

```
library(rlang)
e1 <- env(DNA = c("ACA","CGA"),
          RNA = c("UGU, GAU"),
          a.acid = c("Cys", "Tyr"))
```

The function `rlang::env_print()` shows that `e1` has three character vector bindings.

```
env_print(e1)
```

```
<environment: 0x0000012dcdaab118>
Parent: <environment: global>
Bindings:
* RNA: <chr>
* a.acid: <chr>
* DNA: <chr>
```

The cipher following `<environment:` is the object address for `e1`, a pointer to the location of `e1` in primary memory (Section 3.5).

One can access objects in an environment by name, using the `$` operator or square braces, `[]`.

```
e1$DNA
```

```
[1] "ACA" "CGA"
```

```
e1[["DNA"]]
```

```
[1] "ACA" "CGA"
```

but not by number.

```
e1[[1]]
```

```
Error in e1[[1]]: wrong arguments for subsetting an environment
```

■

8.8.1.1 Special Environments

This section considers several special (and extremely important) **R** environments.

8.8.1.1.1 The Current, Execution, Function, and Empty Environment

The *current environment* defines the environment of the code that is currently being run.

Example 8.24.
The *current environment* can be identified with the function `rlang::current_env()`. Clearly, the *current environment* will often be the *global environment*.

```
current_env()
```

```
<environment: R_GlobalEnv>
```

The *function environment* will be the environment in which the function is created.

```
f <- function(){x <- current_env(); x}
environment(f)
```

```
<environment: R_GlobalEnv>
```

However, the (internal) environment of the function itself will always bind a local environment when called. For example,

```
f()
```

```
<environment: 0x0000012dc8c14270>
```

In this case, the *current environment* is also the *execution environment* of the function `f()`. The *execution environment* of a function will always be a child of the *function environment*, and will generally be ephemeral. That is, once the function has run and created any necessary internal variables, the *execution environment* (and the variables in that environment) will be *garbage collected* and disposed of.

As a demonstration, note that the *execution environment* of `f()` is assigned to a different address every time it is run.

```
f(); f()
```

```
<environment: 0x0000012dc74b5b98>
```

```
<environment: 0x0000012dc3f46d20>
```

■

(Essentially) all environments in **R** have an enclosing *parent environment*, resulting in a hierarchical framework that facilitates lexical scoping and the generation of appropriate search paths.

Example 8.25.
Here is the the parent environment of the function `asbio::pairw.anova()` introduced in Example 8.19.

```
epw <- environment(pairw.anova)
epw
```

```
<environment: namespace:asbio>
```

The immediate parent environment for `pairw.anova()` is the so-called *namespace environment* (see Section 8.8.1.1.2 of the package *asbio*. One can dive deeper into the ancestry of parental environments using `rlang::env_parents()`. Here are the parents for the environment `namespace:asbio`.

```
env_parents(epw)
```

```
[[1]] $ <env: imports:asbio>
[[2]] $ <env: namespace:base>
[[3]] $ <env: global>
```

The output reveals that: 1) the immediate parent of `namespace:asbio` (the parent of `pairw.anova()`) is the *imports environment* (see Section 8.8.1.1.2) from the package *asbio*, 2) the *base* package *namespace environment* is required to identify *asbio* parents, and 3) *asbio* and its functions (and all other loaded packages and their functions) are contained in the *global environment*.

■

The one environment in **R** that *doesn't* have a parent environment is the so-called *empty environment*, `R_EmptyEnv`.

```
empty_env()
```

```
<environment: R_EmptyEnv>
```

If one specifies `last = empty_env()` in `env_parents()`, then *all* parent environments will be shown for a particular environment of interest. This will include the *global environment*, and all loaded package environments. In this process, the last loaded package will be designated as the parent of the global environment, and so on. This process defines a *search path*, because all of

the exported components of all the loaded packages can be called from the *global environment* (Fig 8.5).

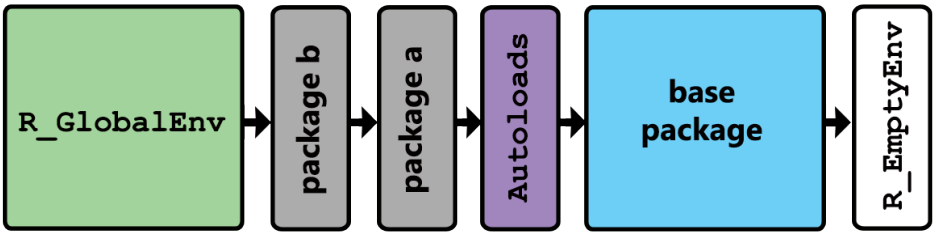


Figure 8.5: Increasing ancestry of environments from left to right, begining with gloabal environment. Package b is the the most recently loaded package.

Example 8.26.
Consider the following example:

```
eps <- env_parents(epw, last = empty_env())
```

Here are the first eight (of 62) enclosing environments for the *asbio namespace environment*, which itself is the parent environment to the function `asbio::pairw.anova()` (Example 8.25).

```
eps[1:8]
```

```
[[1]] $ <env: imports:asbio>
[[2]] $ <env: namespace:base>
[[3]] $ <env: global>
[[4]] $ <env: package:sloop>
[[5]] $ <env: package:plant.ecol>
[[6]] $ <env: package:devtools>
[[7]] $ <env: package:usethis>
[[8]] $ <env: package:deSolve>
```

Identification of the **R** object search path using `base::search()`, is essentially identical to the enclosing parent environment list.

Notably, in any list of environmental ancestors, the last three environments will always be the same. They will be: `Autoloads`→ the *base* package → `R_EmptyEnv` (cf. Fig 8.5).

```
tail(eps, 3)
```

```
[[1]] $ <env: Autoloads>
[[2]] $ <env: package:base>
[[3]] $ <env: empty>
```

`Autoloads` creates a *promise* (Section 8.8.2) to potentially load packages and their functions without actually using memory to complete this task. The *base* package contains the basic

functions that allow **R** to function as a language, and `R_EmptyEnv` underlies all environments.

■

8.8.1.1.2 The Package, Namespace, and Imports Environments

Every package has three fundamental environments. The *package* environment, the *namespace* environment and the *imports* environment. The *package environment* controls how functions and procedures are found by a user (for instance, using the `::` operator). The *namespace environment* serves as an internal interface for a package. It controls how a packaged function finds its required variables (Wickham, 2019). The *imports environment* will be the immediate parent to the *namespace environment*, and defines functions or other objects from other packages that are required for the importing package to work properly (Gupta, 2012). The requirements of an *imports environment* can be listed with the *utils* function `packageDescription()`.

Example 8.27.
Here are the current packages imported by the package *asbio*:

```
packageDescription("asbio", fields = "Imports")
```

```
scatterplot3d, pixmap, plotrix, mvtnorm, deSolve, lattice,
multcompView, grDevices, graphics, stats, utils, gWidgets2,
gWidgets2tcltk
```

■

Package developers can also define a *Depends* field in their packages. These dependencies are not limited to packages. For example, specific versions of **R** may be required for a package. Packages in a Depends field differ from those defined in an imports environment primarily by where they will be located in a search path. If a package is specified in Depends then the package is loaded using an analogous approach to `library()` or `require()`. This makes packages in Depends, an enclosing environment for the global environment (Example 8.26). As a result, Depends can make a package exposed and vulnerable to other loaded packages (Gupta, 2012). A package Depends field can also be listed with `packageDescription()`.

Example 8.28.
Here is the Depends field for *asbio*:

```
packageDescription("asbio", fields = "Depends")
```

```
[1] "R (>= 3.5.0), tcltk"
```

We see that *asbio* currently requires the GUI-generating package *tcltk* (Ch 11), and versions of **R** as or more recent than version 3.5.0.

■

Clearly, defining an *imports environment* and the Depends field are important components of **R** package development (see Ch 10).

Every binding for every function in the *package environment* also occurs in the *namespace environment*. This ensures that every function in a package will be able to call every other function in that package. The *namespace environment*, however, is potentially more inclusive. This is because it can contain entities that do not occur in the *package environment*. These *internal objects* or *non-exported objects* allow package maintainers to "hide" some processes from users. Frequently hidden objects include methods for package-specific classes that are unlikely to be useful outside of the scope of that package. This format is analogous to a *private* or *internal* class or method used by many languages, including Java, C++, and Python.

Parent environments may change if a new package is loaded (Example 8.26). In this case, functions with the same name may be "masked" in more ancestral environments, prompting a warning from **R**. The potential serious consequences of this outcome, however, are generally overcome by the **R** search path framework which relies heavily on the *package*, *namespace* and *imports* environments.

Example 8.29.
As a simple example, the primary purpose of `asbio::pairw.anova()` is to call other *asbio* functions for controlling familywise type I error in a family of pairwise tests, following (potentially) application of an omnibus ANOVA test.

- These functions include: `asbio::lsdCI()`, `asbio::scheffeCI()`, `asbio::dunnettCI()`, and `asbio::tukeyCI()`.
- Further, those *asbio* functions rely on *other* external algorithms, including: 1) functions from the *base* package (which *does not* require explicit importing), including `sum()`, `tapply()`, `seq()`, `outer()`, `paste()`, and `sqrt()`, and 2) the function `anova()` from the *stats* package (which does require importing (Example 8.27)).

The hierarchy of these dependencies prompt potential concerns about `asbio::pairw.anova()` failing if another package is loaded, whose functions have one or more of the *asbio* function names, or the names of required underlying functions from external packages.

Fig 8.6 shows the environmental framework of `asbio::pairw.anova()`, based on Examples 8.25 and 8.26. Note that we have the general search path: `namespace: asbio` → `imports: asbio` → `namespace: base` → `R_GlobalEnv` →. The function's immediate parent environment is `namespace: asbio`. Further, the parent environment of several *asbio* functions implicit to `pairw.anova()`, including `lsdCI()` have the `namespace: asbio` enclosing environment. The function `anova()`, required by `lsdCI()`, is in the *stats* package, which is accessed from the `imports: asbio` environment. Several other functions required by `lsdCI()` are in the *base* package including `sum()`, `tapply()`, `paste()`. These are readily accessible because the `namespace: base` environment will be located just after the *imports environment* of all packages in all search paths.

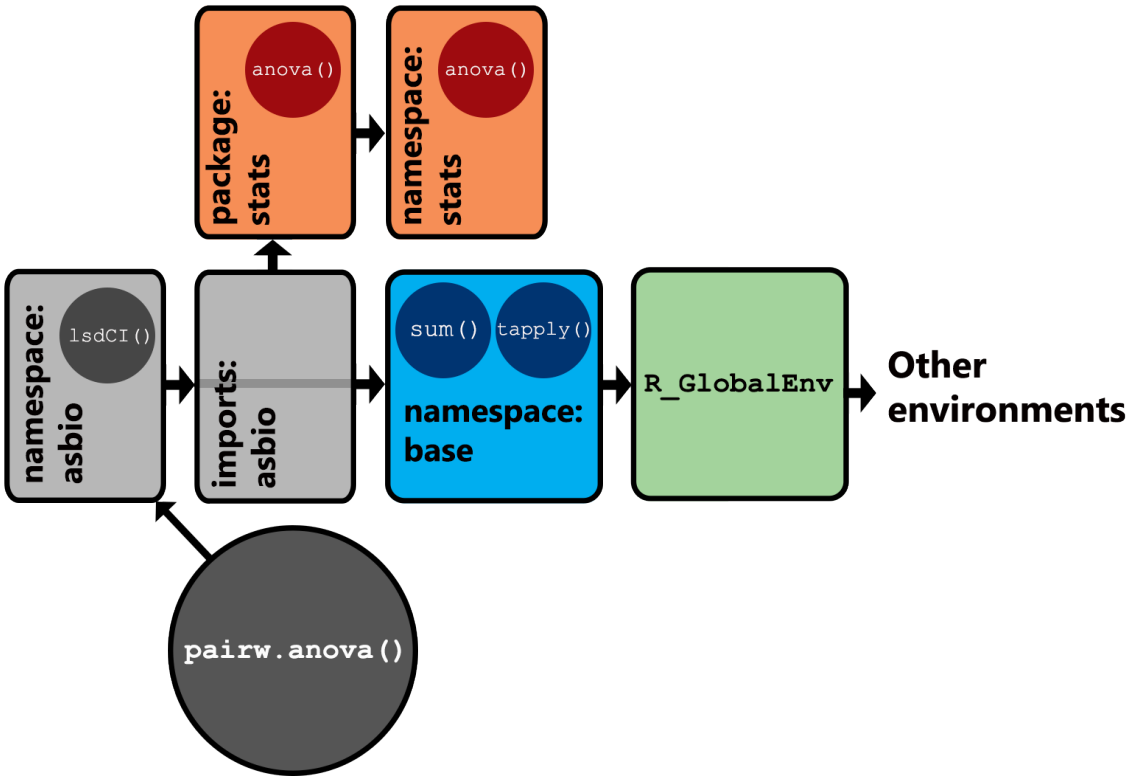


Figure 8.6: The environmental framework of `asbio::pairw()`.

■

8.8.1.1.3 The Caller Environment

A final special environment is the *caller environment*. It refers the environment from which a function is called. This concept is important when following *call stack*. That is, the sequence of operations in a function or process that potentially calls multiple functions. A flawed call stack may result in **R** shut down errors, with potential memory faults.

Example 8.30.
Recall that the *function environment* is the environment where the function was created. For instance, in Example 8.24 we had:

```
f <- function(){current_env()}
environment(f)
```

```
<environment: R_GlobalEnv>
```

However, the *execution environment* of a function is ephemeral, and will be replaced with each function call.

```
f()
```

```
<environment: 0x0000012dbc72f0e0>
```

```
f()
```

```
<environment: 0x0000012dbc676498>
```

Here is a series of nested functions.

```
e <- function(show.local = TRUE){
  L <- list(caller_env(), current_env(), f())
  names(L) <- c("Function Env for e", "Execution Env for e", "Envs in e")
  if(show.local) L
}

f <- function(){
  L <- list(caller_env(), current_env(), g())
  names(L) <- c("Function Env for f", "Execution Env for f",
                "Env for g (in f)")
  L
}

g <- function(){
  caller_env()
}

e()
```

```
$`Function Env for e`
<environment: R_GlobalEnv>

$`Execution Env for e`
<environment: 0x0000012db9c45660>

$`Envs in e`
$`Envs in e`$`Function Env for f`
<environment: 0x0000012db9c45660>

$`Envs in e`$`Execution Env for f`
<environment: 0x0000012db9c45af8>

$`Envs in e`$`Env for g (in f)`
<environment: 0x0000012db9c45af8>
```

Three distinct environments are listed in the output. We begin by calling `e()`, whose *caller environment* is `GlobalEnv_R`. Because `f()` is called by `e()`, the *caller environment* for `f()` is

also the *execution environment* for `e()`, and, because `g()` is called by `f()`, the *caller environment* for `g()` is also the *execution environment* for `f()`.

The function `lobstr::cst()` depicts a call stack sequence as a tree.

```
e <- function() f()
f <- function() g()
g <- function() lobstr::cst()
e()
```

```
   x
 1. \-global e()
 2.   \-global f()
 3.     \-global g()
 4.       \-lobstr::cst()
```

For the example above, `e()` calls `f()`, and `f()` calls `g()`. The function environment of `e()`, `f()`, and `g()` is `GlobalEnv_R`, although `g()` calls `cst()` in *lobstr*.

■

8.8.2 Lazy Evaluation and Promises

Under *lazy evaluation*, objects are evaluated only when those objects are actually needed. Arguments in most **R** functions allow lazy evaluation. For instance, the function below works, even though a value for the argument `z` is not supplied. This is because the arguments are merely *promises*, pending their actual call in the body of the function.

Example 8.31.
For example

```
xy <- function(x, y, z){
  x + y
}

xy(3, 4)
```

```
[1] 7
```

One could expressly include values for `z` (or not) in a function that calls `z`, by adding a conditional statement dependent on its promise status. For instance, using the call `pryr::is_promise(z)`.

```
xyz <- function(x, y, z){
  out <- x + y
  if(pryr::is_promise(z)) out <- out + z
```

```
  out
}
```

```
xyz(4, 3)
```

```
[1] 7
```

```
xyz(4, 3, 2)
```

```
[1] 9
```

A more common solution is to supply argument defaults like `NA` or `NULL`, and use the conventional Boolean functions `is.na()`, and `is.null()` to determine whether those promises should be evaluated.

■

The function `substitute()` allows one to substitute promises (function arguments) bound to an environment. The function `quote()`, on the other hand, returns the evaluable function argument(s). Wickham (2019) refers to this programmatic approach as *non-standard evaluation*.

Example 8.32.
For example, assume that we wish to do pairwise comparison of true means, controlling FWER using Tukey's method. Assume further that we have several independent experiments that use the same factor level framework.

```
x <- factor(rep(c(1,2,3), each = 5))
exp1 <- rnorm(15) # random sample from N(0, 1)
exp2 <- rpois(15, 1) # random sample from POI(1)
exp3 <- rexp(15, 1) # random sample from EXP(1)

# here is quoted version of the call
my_call <- substitute(pairw.anova(y = exp1, x))
# here are names, including (modifiable) promises
eval(my_call)
```

```
95% Tukey-Kramer confidence intervals

            Diff    Lower   Upper Decision Adj. p-value
mu1-mu2 -0.60346 -1.89898 0.69205   FTR H0     0.452139
mu1-mu3  0.37901 -0.91651 1.67453   FTR H0     0.721557
mu2-mu3  0.98247 -0.31304 2.27799   FTR H0     0.148941
```

We can evaluate data from the new experiments by simply updating promises in `my_call`:

```
eval(my_call, list(y = exp2))
```

```
95% Tukey-Kramer confidence intervals

            Diff    Lower   Upper Decision Adj. p-value
mu1-mu2 -0.60346 -1.89898 0.69205   FTR H0     0.452139
mu1-mu3  0.37901 -0.91651 1.67453   FTR H0     0.721557
mu2-mu3  0.98247 -0.31304 2.27799   FTR H0     0.148941
```

```
eval(my_call, list(y = exp3))
```

```
95% Tukey-Kramer confidence intervals

            Diff    Lower   Upper Decision Adj. p-value
mu1-mu2 -0.60346 -1.89898 0.69205   FTR H0     0.452139
mu1-mu3  0.37901 -0.91651 1.67453   FTR H0     0.721557
mu2-mu3  0.98247 -0.31304 2.27799   FTR H0     0.148941
```

■

Lazy evaluation will generally not be possible for arguments in generic functions. For instance, a call to a `print` method for some S3 object (Section 8.7.1.1). In this case, unless a user specifies that an argument can use lazy evaluation, all arguments must be assigned values. This characteristic facilitates the identification of the correct method when making a generic function call (Section 8.8.3).

8.8.3 Function Evaluation

In this section we have learned that function evaluation in **R** often requires the use of environmental search paths (Section 8.8.1.1.2), call stacks (8.8.1.1.3), and promises with lazy evaluation (Section 8.8.2).

Assume we are calling some function, e.g., `pairw.anova()` (Fig 8.6). Important components of the function's search path can be broken into five sequential steps (Gupta, 2012).

1. If the function exists in the *current environment* – for example, it was created in `R_GlobalEnv` or resides within another function that is currently being executed– then **R** will find the function immediately, because **R** looks in the *current environment* first.

2. If the function is not in the *current environment*, then **R** looks to the parent environment of the function. For `pairw.anova()` this is `namespace: asbio` (Fig 8.6). Other *asbio* functions called by `pairw.anova()`, e.g., `lsdCI()` also have `namespace: asbio` as their immediate parent, and can be located at this step.

3. If the function is not in the *namespace environment*, then **R** looks to the *imports environment* of the namespace. For instance, recall that `lsdCI()` which is called by `pairw.anova()` requires `stats::anova()`. This function can be readily located because a promise to load the *stats* package is in `imports: asbio`.

4. If the function is not in the *imports environment*, then **R** looks to the enclosing environment of the *imports environment*, which is always `namespace: base`. For instance, `lsdCI()` requires `sum()`, `tapply()` and several other *base* function. These functions would be obtained at this step.

5. If the function is not found in `namespace: base`, then a search is initiated, beginning with `R_GlobalEnv` and ending with `R_EmptyEnv` (see Fig. 8.5). If a dependency is specified in Depends rather than the *imports environment*, then this is where the function would be found, if it exists at all.

Note that these steps will largely prevent over-writing/masking of important underlying functions.

- First, the developer-defined *imports environment* of the namespace of a function `foo` will force `foo` to use the names/content of packages in the *imports environment*, first. Thus, if an object in the *global environment* uses a name that is also used by imported packages, the imported package object will override the *global environment* object.
- Second, *base* package objects are protected because they are always considered directly after the imported names in the search list for a packaged function.
- Finally, an unsuccessful search will be safely terminated by arriving at `R_EmptyEnv`.

Once a function is "found", evaluation of a function call proceeds, following a three step process (Chambers, 2008).

1. User-defined arguments are matched with required formal arguments (promises).

2. A new *execution environment* is created and objects for each each required formal are assigned to that environment.

3. The body of the function is then evaluated.

Objects called in the *execution environment* of a function are located using essentially the same steps described above for finding functions. That is, the *execution environment* of the function is searched first. If no matches are found an external search path is followed, starting with the *function environment*, then `R_GlobalEnv` (if it is different from the *function environment*), and ending with `R_EmptyEnv`.

Example 8.33.
Consider the following examples:

```
y <- 10
f <- function(x){y}
f(3)
```

```
[1] 10
```

```
f <- function(x){x + y}
f(3)
```

```
[1] 13
```

```
f <- function(x){
  z <- x - 1
  x + y + z}
f(3)
```

```
[1] 15
```

■

Function evaluation may also require calls or direct use of generic functions. The correct method for a generic function call, e.g., `print()`, is identified using a process called *method dispatch*. Method dispatch steps in **R** can be identified using the function `sloop::s3_dispatch`.

Example 8.34.
Here is the process of identifying the correct printing method for an object of class `asbio::pairw` (see Example 8.19).

```
tukey <- pairw.anova(y = eggs, x = trt)
class(tukey)
```

```
[1] "pairw"
```

```
sloop::s3_dispatch(print(tukey))
```

```
=> print.pairw
 * print.default
```

The output: `=> print.pairw` indicates that the *base* function `UseMethod()` has found a print method for the object `tukey` (which has class `pairw`), and that it is called `print.pairw()`. The output `* print.default` indicates that a generic print method exists, and that the function `print.pairw()` will be used in this capacity for `tukey`.

■

Exercises

1. Divide the plant height and soil N values from the dataset from Q. 3 in the Exercises for Chapter 3 (the first two columns of the dataset) by their respective column sums by specifying an appropriate function as the 3rd argument for `apply()`.
2. Below is McIntosh's index of site biodiversity (McIntosh, 1967):

$$U = \sqrt{\sum_{i=1}^{s} n_i^2}$$

where s is the total number of species from a particular site, and n_i is the number of individuals from the ith species, $i = 1, 2, 3, \ldots s$, from that site.

(a) Write a function to calculate the index.
(b) Check it by running it on the following collection of $n_i s$ obtained from a single site: `ni <- c(5,4,5,3,2)`.

3. Below is the Satterthwaite formula for approximating degrees of freedom for the t distribution, under heteroscedasticity:

$$\frac{\left(\frac{S_x^2}{n_x}+\frac{S_y^2}{n_y}\right)^2}{\frac{\left(S_x^2/n_x\right)^2}{n_x-1}+\frac{\left(S_y^2/n_y\right)^2}{n_y-1}}$$

where S_x^2 is the sample variance for variable x, S_y^2 is the sample variance for variable y, n_x is the sample size for x, and n_y is the sample size for y.
(a) Write a function for this equation that has the variables x and y as arguments.
(b) Test the function for `x <- c(1,2,3,2,4,5)` and `y <- c(2,3,7,8,9,10,11)`.

4. Create a function, implementing `switch()`, that can calculate the first or second derivative of a mathematical expression with respect to `"x"`. Test it on `x^3`.

5. Create a function that calculates trimmed means for columns in a quantitative matrix or dataframe. Within the function, use the triple dot operator as an argument in `mean()`, to allow: 1) user-defined trimming (`trim` is an argument in the function `mean()`), and 2) user-defined handling of `NA` outcomes (`na.rm` is also an argument in `mean()`). Your function should have two arguments: one for input data, and one for the triple dot operator, `...`. Test the function on the first two columns of `asbio::cliff.sp`. In your test specify both 10% trimming, and the removal of missing values.

6. The *Fibonacci sequence* describes characteristics of many biological systems, including growth patterns in a wide variety of plant species (Brousseau (1969), Douady and Couder (1996), Fig 8.7), and the ancestral pedigrees of bees (Basin, 1963). The sequence is based on the function:

$$f(n) = f(n-1) + f(n-2) \text{ for } n > 2$$
$$f(1) = f(2) = 1$$

where n represents the nth step in the sequence. Using a loop, obtain the first 100 numbers in the sequence, i.e., find $f(1)$ to $f(100)$. As a check, the first five numbers in the sequence should be: 1, 1, 2, 3, 5.

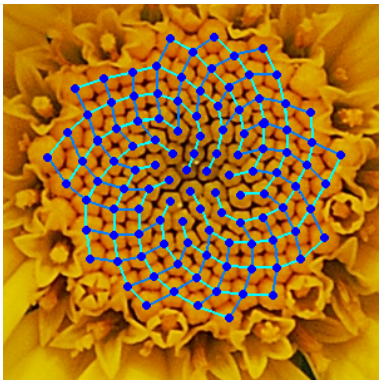


Figure 8.7: The numbers of spirals of disk flowers in yelow chamomile (Asteraceae: *Cota tinctoria*), correspond to the seventh and eighth consecutive Fibonacci numbers: 13 (shown with cyan lines) and 21 (shown with blue lines). https://commons.wikimedia.org/w/index.php?title=File:FibonacciChamomile.PNG&oldid=826903124

7. An interesting chaotic recursive sequence has the function:

$$f(n) = f(n - f(n-1)) + f(n - f(n-2)) \text{ for } n > 2$$
$$f(1) = f(2) = 1$$

   Using a loop, obtain the first 100 numbers in the sequence, i.e., find $f(1)$ to $f(100)$. As a check, the first five numbers in the sequence should be: 1, 1, 2, 3, 3.

8. Create an **R** animation from a for loop 360 steps long that changes the font-size, color, and string rotation of your name (as a character string) in an otherwise empty plot. Assuming the index `i` is used, your loop should include something resembling the code:

```
plot(1:10, type = "n")
text(5.5, "your name", cex = i/36, srt = i, col = i)
Sys.sleep(0.1)
```

9. The Stirling number of the second kind (or the Stirling partition number) counts the number of ways a set of n objects can be partitioned into k groups. This is generally denoted $S(n,k)$ or $\left\{ {n \atop k} \right\}$, and is calculated as:

$$\left\{ {n \atop k} \right\} = \sum_{i=0}^{k} \frac{(-1)^{k-i} i^n}{(k-i)!i!}.$$

   Write a function that calculates $\left\{ {n \atop k} \right\}$ without using a for loop. Use the form: `stirling2 <- function(n, k){function contents}`.

10. The Bell number, B_n, counts the number of ways a set with n elements can be partitioned (Bell, 1938). That is, B_n will be the sum of Stirling numbers for a particular set, for

$k = \{1, 2, \dots, n\}$:

$$B_n = \sum_{k=1}^{n} \left\{ {n \atop k} \right\}.$$

Write a function for calculating B_n that uses `stirling2`, from Q 9, in a for loop. Use the form: `belln <- function(n){function contents}`.

11. The exercise below concerns the speed of loops in **R**. Find the mean estimates of `Loblolly$height` for each level in `Loblolly$seed`, with and without a loop, and find the run times of those operations using the script below:

```
#---- with loop ----#
out <- 1 : nlevels(Loblolly$Seed)

system.time(for(i in levels(Loblolly$Seed)){
temp <- Loblolly[Loblolly$Seed == i,]
out[i] <- mean(temp$height)
})

#---- without loop ----#
system.time(tapply(Loblolly$height, Loblolly$Seed, mean))
```

Describe and discuss your results.

12. Write a function to solve the systems of ODEs below

$$\frac{dx}{dt} = ax + by$$
$$\frac{dy}{dt} = cx + dy$$

To test the function, let $a = 3$, $b = 4$, $c = 5$, $d = 6$, and solve for for $t = 1, 2, \dots, 20$, using classical Runge-Kutta 4^{th} order integration, as implemented by the function `deSolve::rk4()`. Initial values for x and y can be anything but $\{0, 0\}$.

13. Make output from the function in the previous question have an S3 class, and create a plotting method for objects of this class.

14. Provide distinguishing characteristics of an **R** *environment* and an **R** *list*.

15. With respect to special **R** environments:

    (a) Distinguish the current, execution, and empty environments.
    (b) Distinguish the package, namespace, and execution environments.
    (c) Describe the "meaning" of the current environment.

16. Obtain all ancestor environments for the function `asbio::G.mean()`.

Chapter 9

R Interfaces

"You should try things; R *won't break."*

- **Duncan Murdoch**, *from* R*-help (May 2016)*

9.1 Introduction

R can be interfaced with non-native software packages or languages using a software *binding* procedure called an *application programming interface* (API)[1]. The binding provides glue code that allows **R** to work directly with foreign systems that extend its capacities. This can be done in two basic ways.

First, **R**-bindings for external, self-contained software programs can be used. This allows **R**-users to: 1) parameterize and initiate an external program using wrapper functions, and, 2) access the output from that program for further analysis and distillation. If one is using existing APIs, then these operations will generally not require knowledge of non-**R** languages (as the heavy lifting is being done with utility functions within particular **R** packages). One will, however, have to install the **R** package containing the API(s), and the software that one wishes to interface.

Second, one can harness useful characteristics of non-**R** languages by: 1) writing or utilizing source code for procedures in those languages, and 2) using APIs to run those processes in **R**, possibly following their compilation into entities called executable files (Section 9.1.4).

Although bindings for external software are considered briefly (Section 9.1.1), this chapter focuses primarily on interfaces of the second type, particularly bindings to the programming languages Fortran, C, C++, SQL, and Python. Brief backgrounds to those languages are provided here. These, however, should not be considered thorough introductions, given that: 1) I am not a computer language polyglot, and 2) my focus is to demonstrate how other languages can be interfaced with **R**, and not the languages themselves. Appropriate references to language

[1] For more information see Wikipedia (2024g).

resources are provided throughout the chapter[2].

9.1.1 R Bindings for Existing External Software

Many applications exist for interfacing **R** with extant, biologically-relevant software. For example, the **R** package *arcgisbinding*[3] allows **R**-based geoprocessing within ArcGIS Pro and ArcGIS Enterprise.

Example 9.1.
Here I establish a connection to the ArcGIS software package on my computer from within **R**.

```
library(arcgisbinding)
arc.check_product()
```

```
product: ArcGIS Pro (13.5.0.57366)
license: Advanced
version: 1.0.1.311
```

■

The **R** package *igraph* (Csárdi et al., 2025) provides C-bindings for an extensive collection of graph-theoretic tools that can be applied in biological settings, e.g., Aho et al. (2023a). Wrappers for open-source bioinformatics software include the **R** package *RCytoscape*, from the Bioconductor repository, which allows cross-communication between the popular Java-driven software for molecular networks Cytoscape; the **R** package *dartR.popgen* which interfaces with C-based STRUCTURE software for investigating population genetic structure; and the **R** package *strataG* (currently only available on GitHub) which can interface with STRUCTURE, along with the bioinformatics apps: CLUMPP, MAFFT, GENEPOP, fastsimcoal, and PHASE.

R can also be accessed *from* popular commercial software. This capacity is particularly evident in commercial statistical software, including SAS, SPSS, and MINITAB.

9.1.2 Interfacing With Non-R Languages

Source code from other languages can often be interfaced to **R** at the command line prompt, and within **R** functions. For instance, we have already considered the use of Perl regex calls for managing character strings in Ch 4 (Section 4.3), and the **R** Markdown document processing workflow is largely a chain of Markup language conversions (Section 2.10.2.1). Other examples include code interfaces from C, Fortran, C++, SQL, and Python (all formally considered in this chapter), MATLAB (via package *R.matlab*, Bengtsson (2022)), and Java (via package *rJava*,

[2] **R** bindings for languages used primarily for GUI generation and web-based applications, for example, Tcl/Tk, JavaScript, JavaScript Object Notation (JSON), HTML, and Cascading Style Sheets (CSS), are briefly considered in Ch 11.

[3] This package can be installed from r.esri.com using: `install.packages("arcgisbinding", repos="https://r.esri.com", type="win.binary")`.

Urbanek (2021))[4].

9.1.2.1 Costs/Benefits of Interfacing Non-R Scripts

There are costs and benefits to creating/using interface scripts. Costs include:

- Scripts written in non-interpreted languages (e.g., C, Fortran, C++, see Section 9.1.4) will require compilation. Therefore it may be wise to limit such code to package-development applications (Ch 10) because **R** built-in procedures can facilitate this process during package building.
- Interfacing with older, low level languages (e.g., Fortran and C (Section 9.2)) increases the possibility for programming errors, often with serious consequences, including memory faults. That is, *bugs bite* (Chambers, 2008)!
- Interfacing with some languages may increase the possibility for programs being limited to specific platforms.
- **R** programs can often be written more succinctly. For instance, Morandat et al. (2012) found that **R** programs are about 40% smaller than analogous programs written in C.

Despite these issues, there are a number of strong potential benefits. These include:

- A huge number of useful, well-tested applications have been written in other languages, and it is often straightforward to interface those procedures with **R**.
- The system speed of other languages may be much better than **R** for many tasks. For instance, looping algorithms written in non-interpreted languages, are often much faster than corresponding procedures written in **R**.
- Non-OOP languages may be more efficient than **R** with respect to memory usage.

9.1.3 Interfacing with R Markdown/RStudio

Language and program interfacing with **R** can be greatly facilitated with **R** Markdown chunks. This is because many languages other than **R** are supported by **R** Markdown, via *knitr*. The language definition for a particular **R** Markdown chunk is given by the first term in that chunk. For instance, ```` ```{r } ``` ```` initiates a conventional **R** code chunk, whereas ```` ```{python }``` ```` initiates a Python code chunk. Here are the current **R** Markdown language engines (note that items 52-64 are not explicit computer languages).

```
names(knitr::knit_engines$get())
```

```
 [1] "awk"         "bash"        "coffee"      "gawk"        "groovy"
 [6] "haskell"     "lein"        "mysql"       "node"        "octave"
[11] "perl"        "php"         "psql"        "Rscript"     "ruby"
[16] "sas"         "scala"       "sed"         "sh"          "stata"
[21] "zsh"         "asis"        "asy"         "block"       "block2"
```

[4] **R** can also be called *from* a number of different language frameworks including C and C++ (see package *RInside*, Eddelbuettel et al. (2023b), and Section 11.6 for specific examples), Python (via the Python package, *rpy2*), and Java (via the Java package *RCaller*, Satman (2014)).

```
[26] "bslib"      "c"          "cat"        "cc"         "comment"
[31] "css"        "ditaa"      "dot"        "embed"      "eviews"
[36] "exec"       "fortran"    "fortran95"  "go"         "highlight"
[41] "js"         "julia"      "python"     "R"          "Rcpp"
[46] "sass"       "scss"       "sql"        "stan"       "targets"
[51] "tikz"       "verbatim"   "glue"       "glue_sql"   "gluesql"
[56] "theorem"    "lemma"      "corollary"  "proposition" "conjecture"
[61] "definition" "example"    "exercise"   "hypothesis"  "proof"
[66] "remark"     "solution"
```

As evident in the output above, **R** Markdown engines extend to compiled languages including Fortran (engine = `fortran`), C (engine = `c`) and C++, via the *Rcpp* package (engine = `Rcpp`).

9.1.4 Interpreted and Compiled Languages

Source code refers to human-readable instructions under the grammatical framework of some programming language. For instance, the script

```
x <- c(1,3,6)
mean(x)
```

```
[1] 3.3333
```

is an example of **R** source code, with its evaluation result shown.

A computer, however, only fundamentally understands *machine code* (also called *object code*)[5]. Conventionally, machine code is a binary {0, 1} representation of a source code procedure (Section 12.2). The machine code for the **R** script `x <- c(1, 3, 7); mean(x)` is not show here. However, the binary (see Ch 12) translation of $3.33\bar{3}$, is:

```
asbio::dec2bin(mean(x))
```

```
[1] 11.0101010101
```

Source code must be translated into machine code before a computer can execute it.

Non-interpreted (compiled) languages (for instance, Fortran, C, C++, C#, and Java) use a *compiler* (a conversion program) to transform source code into machine code (Ch 11). The result of this process is often called an *executable file*, or simply an *executable* (Figure 9.1). Executables can be called from within **R** (or elsewhere) to run independently, or to enhance other functions and procedures.

[5] Machine code is the lowest level interface to a computer. Assembly language, mentioned briefly in Section 1.4.1 is a higher level process that is not strictly numerical, although it provides direct map between machine code and human-readable mnemonics (Wikipedia, 2025a).

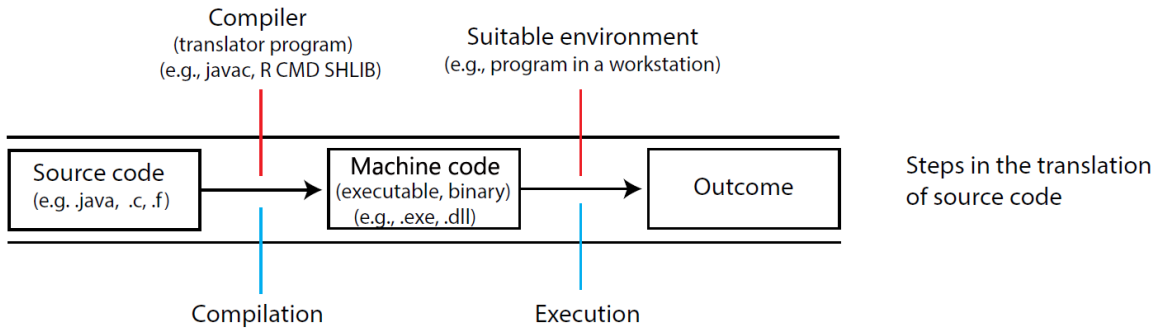


Figure 9.1: Creating an executable file in a compiled language.

Compilers are often specific to underlying compiled languages. For instance, the `ILCPU` compiler is intended only for C# code, and the `clisp` compiler is intended for Lisp. The following compilation frameworks are very important to **R**-users. The last two are particularly important for Windows platforms.

- The GNU Compiler Collection (GCC) contains a large number of open source compilers, including `gcc` (for C), `g++` (for C++), and `gfortran` (for Fortran).

- MinGW ("Minimalist GNU for Windows") is a free open source development environment for creating Windows applications. It includes a GCC port, along with other compilation tools specifically for Windows.
- Rtools is a Windows toolchain, intended primarily for building packages (and **R**) from source code. As of version 4.5, Rtools includes Msys2 –a collection of tools and libraries for building, installing and running Windows software, the GCC 14/MinGW-w64 compiler toolchain for Windows, and QPDF –a command line tool and C++ library that performs content-preserving transformations on PDF files.

R, along with many other useful languages (e.g., Python, JavaScript), is considered an *interpreted language*. In programming, an *interpreter* directly executes source code without the requirement of compilation. **R** uses a Scheme-like interpreter to translate source code into an *intermediate representation* of source code and machine code entities, which is then immediately executed. These operations are generally underlain by the language C. Because translation must precede machine code implementation, interpreted procedures tend to be slower than fully compiled procedures. This is particularly true for iterative processes like loops.

9.1.5 Shells

Compiling object code will require installation/access to an appropriate compiler program. A compiler, in turn, will likely require initiation from a shell command line.

The Windows OS currently has two built-in command line shells[6]. The *Command* shell (also know as cmd.exe or cmd), was introduced in 1993 and maintains strong similarities to the MS-DOS command framework (see Wikipedia (2023b)). *PowerShell*, introduced in 2006, is back-compatible with most cmd commands, but also has advanced programming features, including the ability to generate objects and handle OOP scripts. Other differences between cmd and PowerShell are discussed here. Commands and processes for Windows shells differ in many respects from the POSIX (Portable Operating System Interface) compliant shells generally used by Unix-like systems. The most widely used POSIX shell, *BASH*, allows straightforward execution and modification of Linux/Unix operations that may be difficult to translate to Windows OS[7]. The Windows Subsystem for Linux (WSL) allows one to run Linux, including BASH shells, directly in Windows. I strongly recommend WSL for complex compilation of scripts using makefiles (Section 11.6), Windows management of **R**-driven apps in servers (Section 11.5.7) and high performance computing projects using **R** (Section 12.8.1).

Table 9.1 shows some shell commands, including several that work the same way in both Windows and Linux/BASH. Additional guidance for Windows shells can be found at the learn.microsoft.com website. Additional guidance for BASH can be found here.

Table 9.1: Command shell commands for Windows (i.e., Command shell and PowerShell) and BASH.

Operation	Windows	BASH/Linux
Change directory	`cd`	`cd`
Navigate to address	`cd <path>`	`cd <path>`
Navigate "up" one directory	`cd ..`	`cd ..`
Return to the root dirctory	`cd\\`	`cd` or `cd ~`
Clear command line	`cls`	`clear -x`
List files in directory	`dir`	`ls`
Copy file	`copy <file>`	`cp <file>`
Move file	`move <file> <destination>`	`mv <file> <destination>`
Find a string	`findstr <string>`	`grep <string> <path>`
Get help	`<command> \?` or `help <command>`	`<command> --h`

Example 9.2.
The default home directory for my computer is: C:\Users\ahoken. To navigate to the root (parent) directory of this hierarchy, I could type `cd/` (Table 9.1) in the Windows cmd Command shell.:

[6]The Windows Command shell interface can be accessed by typing: **Windows key** + R , typing `cmd` in the interactive GUI, and pressing Enter , or by typing `cmd` in the Windows Search bar. Similarly, PowerShell can be accessed by typing `powershell` in the Windows Search bar.

[7]BASH is short for Bourne Again SHell. The Bourne shell, developed in the late 1970s, is named for Stephen Bourne who led its development while working at Bell Labs from 1975-1983.

```
C:\Users\ahoken> cd/
C:\>
```

Note that the shell has the same command line prompt as **R**: >.

■

Example 9.3.
What if I wanted to list all the **R**-Markdown files (those with an `.rmd` file extension) in the home directory for this book? I could navigate to the correct directory, and type `dir /b *.rmd` in the Windows Command shell:

```
C:\> cd C:\Users\ahoken\Documents\GitHub\Amalgam
C:\Users\ahoken\Documents\GitHub\Amalgam> dir /b *.rmd
01-Ch1.Rmd
02-Ch2.Rmd
03-Ch3.Rmd
04-Ch4.Rmd
05-Ch5.Rmd
06-Ch6.Rmd
07-Ch7.Rmd
08-Ch8.Rmd
09-Ch9.Rmd
10-Ch10.Rmd
11-Ch11.Rmd
12-Ch12.Rmd
13-references.Rmd
index.Rmd
```

The `\b` option in `dir` means: "use a bare format (no heading information or summary)." The asterisk, `*`, is a wildcard, indicating that only files with a `.rmd` extension should be listed.

■

Example 9.4.
To search for the text: `"An Amalgam of R"` in the **R** Markdown document index.rmd, I could use the `findstr` command:

```
C:\Users\ahoken\Documents\GitHub\Amalgam> findstr "\"An\ Amalgam\ of\ R\"" index.rmd
title: "An Amalgam of R"
```

Note that I escape both quotes and spaces in the string. The entire line of text containing the string is: `title: "An Amalgam of R"` and is part of the YAML header in index.rmd (Section 2.10.2.1). For more information type `findstr \?` in the Windows Command shell (Table 9.1).

It is easier to search for text strings using BASH under a regex approach (Section 4.3.6). Here I

access the Ubuntu version of Linux implemented in WLS from a Windows shell, and navigate to the home directory of this book:

```
> wsl.exe -d Ubuntu

Welcome to Ubuntu 24.04.3 LTS (GNU/Linux 6.6.87.2-microsoft-standard-WSL2 x86_64)

$ cd Documents/GitHub/Amalgam
```

Note that the BASH command line operator is $. Here I get an approximate count of the number of **R** Markdown **R** code chunks in the book by querying the string `'{r'` within underlying .Rmd files, using the `grep` option `-c`.

```
Documents/GitHub/Amalgam$ grep -c '{r' *.Rmd
01-Ch1.Rmd:16
02-Ch2.Rmd:174
03-Ch3.Rmd:256
04-Ch4.Rmd:102
05-Ch5.Rmd:53
06-Ch6.Rmd:138
07-Ch7.Rmd:116
08-Ch8.Rmd:175
09-Ch9.Rmd:158
10-Ch10.Rmd:23
11-Ch11.Rmd:161
12-Ch12.Rmd:42
13-references.Rmd:0
index.Rmd:5
```

There are over 1400 **R** chunks in the book (including some, hidden, formatting scripts)[8], and 256 occur in Chapter 3. Here are the number of C++, SQL and Python chunks used in the current chapter:

```
$ grep -c '{Rcpp' 09-Ch9.Rmd
30
$ grep -c '{sql' 09-Ch9.Rmd
16
$ grep -c '{python' 09-Ch9.Rmd
45
```

■

[8]To sum the occurrences of `'{r'` in the Chapters, I could have used: `grep -c '{r' *.Rmd | awk -F: '{n+=$2} END {print n}'`.

CAUTION!

Shells are powerful tools, and serious damage can be done to your computer through their misapplication. This is particularity true when running destructive commands, e.g., `del`, `rmdir`, `format` as an Administrator.

9.1.6 Compilation for R Interfacing

Windows and Mac OS executables will generally have an .exe or .app extension, respectively, although extensions for Linux/Unix files are not required for a file be recognized and run as an executable. For distribution in **R** packages, however, executables must have a *shared library* format, with .dll, .dylib, and .so, extensions for Windows, Mac OS, and Linux/Unix operating systems, respectively[9]. Shared library objects are different from conventional executables in that they cannot be evaluated directly. In this case, **R** will be required as the executable entry point.

R provides shared library compilers for Fortran and C and several other languages via its `SHLIB` procedure, which is accessed from the `Rcmd` executable. The `Rcmd` program is located in the **R** bin directory, following a conventional download from CRAN, along with several other important **R** executables, including `R.exe` and `Rgui.exe`. `Rcmd` procedures are typically invoked from a shell (e.g., cmd.exe) using the format: `R CMD procedure args`. Here `procedure` is currently one of `INSTALL`, `REMOVE`, `SHLIB`, `BATCH`, `build`, `check`, `Rprof`, `Rdconfig`, `Rdiff`, `Rd2pdf`, `Stangle`, `Sweave`, `config`, `open`, and `texify`, and `args` defines arguments specific to the `Rcmd` command[10]. For example, the shell script:

```
R CMD SHLIB foo
```

would prompt the building of a shared library object from the user-defined script `foo`, which could be comprised, for example, of Fortran, C, or C++ source code[11]. There are actually many ways to compile shared libraries for use in **R**.

- First, as noted above, one could compile a shared library from some script, `foo`, by running `R CMD SHLIB foo` at a shell command line. The shared library could then be loaded and called, using an appropriate foreign function interface. I apply this approach from the Windows Command shell in Example 9.6.
- Second, one could rely only on **R** Markdown engines (see Section 2.7 in Xie et al. (2020)). In particular, one could write a script for a compiled language within a chunk with an approriate language engine. The chunk would be automatically compiled using `SHLIB` when running the chunk. The resulting shared library could then be loaded and called in a subsequent **R** chunk, using an appropriate foreign function interface, e.g., `.Call()` (see Section 9.2). Unfortunately, this process may be hampered by a number of factors,

[9] The Windows extension .dll identifies a Windows *dynamic-link library* (DLL) file, as does the Mac OS .dylib extension. The Linux/Unix .so stands for *shared object* or *shared library* file.

[10] Several of these commands are addressed in Ch 10, including `INSTALL`, `check`, `BATCH`, `build`, AND `Rd2pdf`.

[11] `SHLIB` stands for *shared library*.

including non-administrator permissions and incorrect environmental path definitions, particularly on Windows computers.
- Third, one could use a non-`RCMD SHLIB` compiler, for instance Windows GCC tools in Rtools. Rtools compilers are used throughout Section 9.3. The package *inline* uses the GCC to allow users to create, compile, and run scripts, written in compiled languages, all from the **R** command line (see Section 9.3.2).

9.2 Fortran and C

S, the progenitor of **R**, was created at a time when Fortran[12] routines dominated numerical programming, and **R** arose when C[13] was approaching its peak in popularity. As a result, strong connections to those languages, particularly C, remain in **R**[14]. **R** contains specific base functions for interfacing with both C and Fortran executables: `.C()` and `.Fortran()`. A more recently developed function, `.Call()`, which allows straightforward exchanges of `SEXP` objects to and from C, is formally introduced in Section 9.3.

Recall that an **R** object of class `numeric` will be automatically assigned to base type `double`, although it can be coerced to base type `integer` (with information loss through the elimination of its "decimal" component).

```
as.integer(2.5)
```

```
[1] 2
```

Many other languages, however, do not automatically assign base types. Instead, explicit user-assignments for underlying base types are required.

If one is interfacing **R** with Fortran or C, only a limited number of base types are possible (Table 9.2), and one will need to use appropriate coercion functions for **R** objects if one wishes to use those objects in Fortran or C scripts[15]. When using `.C()` and `.Fortran()`, Interfaced C script arguments must be *pointers*, and arguments in Fortran scripts must be arrays for the types given in Table 9.2.

Raw Fortran source code is generally saved as an .f, or (.f90 or .f95; modern Fortran) file,

[12] As noted in Ch 1, Fortran is one of the oldest programming languages still in active use. Although Fortran's development followed an IBM proposal for an alternative to assembly language in 1953 (Backus, 1998), and the first correctly compiled version of Fortran occurred in 1958, Fortran remains among the top programming languages in the TIOBE index (Wikipedia, 2024f). Early iterations of **S** were strongly dependent on Fortran procedures (Section 1.4).

[13] Recall from Ch 1 that C is a widely-used general programming language developed during the 1970s (Ritchie, 1993). Further, because **R** is largely written in C, it is not surprising that the most direct language for interfacing with **R** is C (Chambers, 2008).

[14] The complete **R** API for C can be viewed by typing: `` `rinternals <- file.path(R.home("include"), "Rinternals.h"); file.show(rinternals)` `` in the **R** console.

[15] The functions `.Call()` and `External()` expect that **R**-dependent objects will be used (as declared `SEXP` objects). The functions `.C()` and `.Fortran()` assume that **R** objects will not be directly used in algorithms.

Table 9.2: Correpsonding types for **R**, C, and Fortran. Table adapted from Chambers (2008).

R base type	**R** coercion function	C type	Fortran type
`logical`	`as.integer()`	`int *`	`integer`
`integer`	`as.integer()`	`int *`	`integer`
`double`	`as.double()`	`double *`	`double precision`
`complex`	`as.complex()`	`Rcomplex *`	`double complex`
`charater`	`as.character()`	`char **`	`character*255`
`raw`	`as.character()`	`char *`	none

whereas C source code is saved as an .c file. One can create a file with the correct file type extension by using `file.create()`.

Example 9.5.
For example, below I create a file called `foo.f90` that I can open (from my working directory) in a text editor (e.g., Notepad) or IDE (e.g., RStudio) to build a Fortran script.

```
file.create("foo.f90")
```

■

RStudio provides an IDE for C, allowing straightforward generation of `.c` files.

9.2.1 Compiling and Executing C and Fortran Programs

Notably, the `SHLIB` compilers will only work for Fortran code written as a subroutine[16] and C code written in void formats[17]. As a result, neither code type will return a value directly.

Example 9.6.
Here is a simple example for calling Fortran and C compiled executables from **R** to speed up looping. The content follows class notes created by Charles Geyer at the University of Minnesota. Clearly, the example could also be run without looping. Equation (9.1) shows the simple formula for converting temperature measurements in degrees Fahrenheit to degrees Celsius.

$$C = 5/9(F - 32) \tag{9.1}$$

where C and F denote temperatures in Celsius and Fahrenheit, respectively.

Here is a Fortan subroutine for calculating Celsius temperatures from a dataset of Fahrenheit measures, using a loop.

[16] A Fortran subroutine is invoked with a `CALL` statement. Unlike a Fortran function, which returns a single value, a subroutine can return many (or no) values.

[17] Void functions in C are used for their side effects, such as performing a task or writing to output.

```
1 subroutine FtoC(n, x)
2 integer n
3 double precision x(n)
4 integer i
5 do 100 i = 1, n
6 x(i) = (x(i)-32)*(5./9.)
7 100 continue
8 end
```

The Fortran code above consists of the following steps:

- On Line 1 a subroutine is invoked using the Fortran function `subroutine`. The subroutine is named `FtoC`, and has arguments `x` (the Fahrenheit temperatures) and `n` (the number of temperatures)
- On Line 2 the entry given for `n` is defined to be an integer (Table 9.2).
- On Line 3 we define `x` to be a double precision numeric vector of length `n`.
- On Line 4 we define that the looping index to be used, `i`, will be an integer.
- On Lines 5-7 we proceed with a Fortran `do` loop. The code `do 100 i = 1, n` means that the loop will 1) run initially up to 100 times, 2) has a lower limit of 1, and 3) has an upper limit of `n`. The code: `x(i) = (x(i)-32)*(5./9.)` calculates Eq. (9.1). The code `5./9.` is used because the result of the division can be a non-integer. The code `100 continue` allows the loop to continue to `n`.
- On Line 8 the subroutine ends. All Fortran scripts must end with `end`.

I save the code under the filename `FtoC.f90`, and transfer it to an appropriate directory (I use `C:/Users/ahoken/Documents/Amalgam/Amalgam_Bookdown/scripts/`). I then open a Windows shell editor.

I compile `FtoC.f90` using the script `R CMD SHLIB FtoC.f90`. Thus, at the Windows Command shell I enter:

```
> cd C:\Program Files\R\R-4.4.2\R\bin\x64
> R CMD SHLIB C:/Users/ahoken/Documents/GitHub/Amalgam/scripts/FtoC.f90
```

Note the change from back slashes to (Unix-style) forward slashes when specifying addresses for `SHLIB`. The command above creates the compiled Fortran executable `FtoC.dll`. Specifically, the Fortran compiler, `gfrotran`, from within the GCC, is used to create an intermediate object file, `FtoC.o`. The object file is then used to create a .dll file with the `gcc` program. By default, the .dll is saved in the directory that contained the source code. Finalization of the compilation requires linkage to the the RTools MinGW toolchain.

Steps in the compilation process can be followed (with some difficulty) in the Windows shell output below. Some lines are broken to increase clarity.

```
using Fortran compiler: 'GNU Fortran (GCC) 14.2.0'
gfortran      -O2  -mfpmath=sse -msse2 -mstackrealign
-c C:/Users/ahoken/Documents/GitHub/Amalgam/scripts/FtoC.f90
-o C:/Users/ahoken/Documents/GitHub/Amalgam/scripts/FtoC.o
```

```
gcc -shared -s -static-libgcc
-o C:/Users/ahoken/Documents/GitHub/Amalgam/scripts/FtoC.dll
tmp.def C:/Users/ahoken/Documents/GitHub/Amalgam/scripts/FtoC.o
-LC:/rtools45/x86_64-w64-mingw32.static.posix/lib/x64
-LC:/rtools45/x86_64-w64-mingw32.static.posix/lib -lgfortran -lquadmath
-LC:/PROGRA~1/R/R-45~1.1/bin/x64 -lR
```

In the output above, snippets beginning with `-`, define `gfortran` and `gcc` program options from within the GCC. For instance, `-c` means "compile and assemble, but do not link," `-o <file>` means "place output in a defined `<file>`", and `-L<directory>` links `<directory>` to the program search path. The `-O` family of flags (including `-O0`, `-O1`, and `-O2`) concern compilation optimization. The option `-O2` indicates "high optimization" at the cost of longer compilation times. Importantly, the option `-shared` indicates that a shared library should be assembled instead of a standard executable. Details on many `gcc` (and `gfortran`) options can obtained by calling `gcc --help` from the BASH command line. The options `-mfpmath`, `-msse2`, `-mstackrealign` are so-called "target-specific options." Details concerning those options are provided in `gcc --help=target`.

Here is analogous C loop script for converting Fahrenheit to Celsius.

```
1
2 void ftocc(int *nin, double *x)
3 {
4   int n = nin[0];
5   int i;
6   for (i=0; i<n; i++)
7     x[i] = (x[i] - 32) * 5. / 9.;
8 }
```

The C code above consists of the following steps.

- Line 1 is a line break.
- On Line 2 a `void` function is initialized with two arguments. The code `int *nin` means "access the value that `nin` points to and define it as an integer." The code `double *x` means: "access the value that `x` points to and define it as double precision."
- Lines 8-9 define the C `for` loop. These loops have the general format: `for ( init; condition; increment ) {statement(s); }`. The `init` step is executed first and only once. Next the `condition` is evaluated. If true, the loop is executed. The syntax `i++` literally means: `i = i + 1`. Note that code lines are ended with a semicolon, `;` and that indices (e.g., `i`) start at `0`. Consideration of the language is greatly expanded in Section 9.3, which considers the language C++.

Once again, I save the source code, `FtoCc.c`, within an appropriate directory. I compile the code using the command `R CMD SHLIB FtoCc.c`. Thus, at the at the Windows Command shell I enter:

```
> cd C:\Program Files\R\R-4.5.1\bin\x64
> R CMD SHLIB C:/Users/ahoken/Documents/GitHub/Amalgam/scripts/FtoCc.c
```

This creates the shared library executable `FtoCc.dll`.

```
using C compiler: 'gcc.exe (GCC) 14.2.0'
gcc  -I "C:/PROGRA~1/R/R-45~1.1/include"
-DNDEBUG -I "C:/rtools45/x86_64-w64-mingw32.static.posix/include"
-O2 -Wall -std=gnu2x  -mfpmath=sse -msse2 -mstackrealign
-c C:/Users/ahoken/Documents/GitHub/Amalgam/scripts/FtoCc.c
-o C:/Users/ahoken/Documents/GitHub/Amalgam/scripts/FtoCc.o
gcc -shared -s -static-libgcc
-o C:/Users/ahoken/Documents/GitHub/Amalgam/scripts/FtoCc.dll
tmp.def C:/Users/ahoken/Documents/GitHub/Amalgam/scripts/FtoCc.o
-LC:/rtools45/x86_64-w64-mingw32.static.posix/lib/x64
-LC:/rtools45/x86_64-w64-mingw32.static.posix/lib
-LC:/PROGRA~1/R/R-45~1.1/bin/x64 -lR
```

Below is an **R**-wrapper that can call the Fortran executable, `call = "Fortran"`, the C executable, `call = "C"`, or use **R** looping, `call = "R"`. Several new functions are used. On Line 10 the function `dyn.load()` is used to load the shared Fortran library file `FtoC.dll`, while on Lines 14-15 `dyn.load()` loads the shared C library file `FtoCc.dll`. Note that the variable `nin` is pointed toward `n`, and `x` is included as an argument in `dyn.load()` on Line 15. On Line 11 the function `.Fortran()` is used to execute `FtoC.dll`, and on Line 16 `.C()` is used to execute `FtoCc.dll`.

```
1  F2C <- function(x, call = "R"){
2    n <- length(x)
3    if(call == "R"){
4      out <- 1:n
5      for(i in 1:n){
6      out[i] <- (x[i] - 32) * (5/9)
7      }
8    }
9    if(call == "Fortran"){
10     dyn.load("C:/Users/ahoken/Documents/Amalgam/Amalgam_Bookdown/scripts/FtoC.dll")
11     out <- .Fortran("ftoc", n = as.integer(n), x = as.double(x))
12   }
13   if(call == "C"){
14     dyn.load("C:/Users/ahoken/Documents/Amalgam/Amalgam_Bookdown/scripts/FtoCc.dll",
15              nin = n, x)
16     out <- .C("ftocc", n = as.integer(n), x = as.double(x))
17   }
18   out
19 }
```

Here I create 10^8 potential Fahrenheit temperatures that will be converted to Celsius using (unnecessary) looping.

```
x <- runif(100000000, 0, 100)
head(x)
```

```
[1] 91.660 15.116 87.877 10.158 19.373 59.775
```

Note first that the Fortran, C, and **R** loops provide identical temperature transformations. Here are first 6 transformations:

```
head(F2C(x[1:10], "Fortran")$x)
```

```
[1]  33.1444  -9.3801  31.0429 -12.1343  -7.0151  15.4305
```

```
head(F2C(x[1:10], "C")$x)
```

```
[1]  33.1444  -9.3801  31.0429 -12.1343  -7.0151  15.4305
```

```
head(F2C(x[1:10], "R"))
```

```
[1]  33.1444  -9.3801  31.0429 -12.1343  -7.0151  15.4305
```

However, the run times are dramatically different[18]. The C executable is much faster than **R**, and the venerable Fortran executable is even faster than C!

```
system.time(F2C(x, "Fortran"))
```

```
   user  system elapsed
   0.67    0.10    0.77
```

```
system.time(F2C(x, "C"))
```

```
   user  system elapsed
   0.61    0.13    0.78
```

```
system.time(F2C(x, "R"))
```

```
   user  system elapsed
   5.62    0.30    6.07
```

■

9.3 C++

C++ (pronounced *see plus plus*) is a high-level, general-purpose, programming language that is well known for its simplicity, efficiency, and flexibility[19]. C++ was originally intended to be a mere extension of C. Although its scope now greatly exceeds this goal, C++ syntax remains similar to C. For instance, like C:

- C++ is a compiled language (it requires a compiler to convert its source code to an executable).
- Lines of C++ code end with semicolons, ;.
- C++ comment annotations begin with \\.

[18] Run on an Intel Core processor with a clock speed of 3 GHz, and 32 GB of RAM.

[19] C++ was first introduced in 1985 by Danish computer scientist Bjarne Stroustrup –who was then a technical staff member at Bell Labs– to add OOP features to the C language (Wikipedia, 2024b).

- The `for` loop syntax for C++ is: `for (init; condition; increment)`.
- C++ index values start at `0`, meaning that the last index value will be `n - 1`.
- Square braces, `[]`, can be used for subsetting. Although, see content regarding *Rcpp* C++ types below.
- C++ logical operators are similar to those used in **R**. For example, `==` is the Boolean *equals* operator, `!` is the unary operator for *not*, and the operators for *and* and *or* are `&&` and `||`, respectively.
- Like C, C++ Boolean designations,`true` and `false` are used (instead of `TRUE` and `FALSE`).

The major difference between C and C++ is that C++ supports objects and object classes, whereas C does not. Helpful online C++ tutorials and references can be found at https://www.learncpp.com/ and https://en.cppreference.com/w/cpp, respectively. As advanced resources, Wickham (2019) recommends the books *Effective C++* (Meyers, 2005) and *Effective STL* (Meyers, 2001)[20].

9.3.1 Rcpp

The **R** package *Rcpp* (Eddelbuettel, 2013; Eddelbuettel and Balamuta, 2018; Eddelbuettel et al., 2023a) provides an extension of the **R** API, with a consistent set of C++ classes (Eddelbuettel and François, 2023). As a result, the package allows users to employ the many useful characteristics of C++ –including fast loops, efficient calls to functions, and access to advanced data container classes including maps[21] and double-ended queues[22], while enjoying the benefits of **R** –including terse scripting and straightforward manipulation of vectors and matrices. As Wickham (2019) notes:

> "I do not recommend using C for writing new high-performance code. Instead write C++ with *Rcpp*. The *Rcpp* API protects you from many of the historical idiosyncracies of the **R** API, takes care of memory management for you, and provides many useful helper methods."

Useful resources for *Rcpp* include extensive vignettes from the package itself, Chapter 25 from Wickham (2019), and the online document Rcpp for everyone (Tsuda, 2020).

In order to use *Rcpp*, users will require additional toolchains, including a dedicated C++ compiler.

- Windows users will need Rtools. Use of Rtools will require that its installation be along an defined environmental path.
- Mac-OS users will require the Xcode command line tools.
- Linux users can use: `` `sudo apt-get install r-base-dev` ``.

[20] *STL* refers to the C++ Standard Template Library, which provides useful data structures and algorithms for C++ programming.

[21] A map is a C++ standard library, `std`, class that stores data in the form of key value pairs.

[22] A double-ended queue (*deque*) is a data container to which elements can be added to or removed from either the front (head) or back (tail). *Rcpp* contains double-ended queue (*deque*) functions for many of its object classes. The **R** function `c()`, and the Python function `append()` (Section 9.5) can also be used for this purpose, within their respective programming settings.

Example 9.7.
As a first step, Eddelbuettel and Balamuta (2018) recommend running a minimal example to ensure that the *Rcpp* toolchain is working. For instance:

```
library(Rcpp)

evalCpp("2 + 2")
```

```
[1] 4
```

Here the function `Rcpp::evalCpp()` creates a compiled C++ shared library, specified in `evalCpp()`, from the text string `"2 + 2"`. This step is accomplished via the function `Rcpp::cppFunction()` (see Example 9.10 below). The `evalCpp()` function then calls shared library, using `.Call()`, to obtain a result in **R**.

■

9.3.1.1 Data Types

Recall (Section 2.3.6) that **R** base types correspond to a C *typedef* alias called an `SEXP` (**S**-expression). *Rcpp* provides dedicated C++ classes for most of the 24 `SEXP` types. Some of these are shown– for scalar, vector, and matrix frameworks– in Table 9.3. Scalars can be aptly handled with C++ standard library, `std`, procedures. The `Rcpp::Vector` types are similar to `std::vector`[23], although the former are designed to facilitate interactivity with **R**.

Table 9.3: Correpsonding types for **R**, C++, and *Rcpp*. Table adapted from Tsuda (2020).

R type	C++ (scalar)	*Rcpp* (scalar)	`Rcpp::Vector`	`Rcpp::Matrix`
`logical`	`bool`	-	`LogicalVector`	`LogicalMatrix`
`integer`	`int`	-	`IntegerVector`	`IntegerMatrix`
`numeric`	`double`	-	`NumericVector`	`NumericMatrix`
`complex`	`complex`	`Rcomplex`	`ComplexVector`	`ComplexMatrix`
`character`	`char`	`String`	`CharacterVector`	`CharacterMatrix`
`Date`	-	`Date`	`DateVector`	-
`POSIXct`	`time_t`	`Datetime`	`DatetimeVector`	-

Rcpp also has types for **R** base types `list` and `S4`, and **R** class `dataframe`. These are called using `Rcpp::List`, `Rcpp::S4`, and `Rcpp::Dataframe`, respectively. *Rcpp* types are designated with their class names.

Example 9.8.
The code (not run) below creates `Rcpp::Vector` objects called `v`. Corresponding **R** code is commented *above* C++ code.

[23] See class documentation here

```
1  // v <- rep(0, 3)
2  NumericVector v (3);
3
4  // v <- rep(1, 3)
5  NumericVector v (3,1);
6
7  // v <- c(1,2,3)
8  // [[Rcpp::plugins("cpp11")]]
9  NumericVector v = {1,2,3};
10
11 // v <- 1:3
12 IntegerVector v = {1,3};
13
14 // v <- as.logical(c(1,1,0,0))
15 LogicalVector v = {1,1,0,0};
16
17 // v <- c("a","b")
18 CharacterVector v = {"a","b"};
```

Note that curly braces `{}` are used to initialize the `NumericVector` object on Line 9, and the `IntegerVector`, `LogicalVector`, and `CharacterVector` objects on Lines 12, 15, 18, respectively. This reflects C++ 11 grammar[24]. C++11 can be enabled with the comment: `// [[Rcpp::plugins("cpp11")]]` (Line 8).

Here I create `Rcpp::Matrix` objects named `m`:

```
1  // m <- matrix(0, nrow=2, ncol=2)
2  NumericMatrix m(2);
3
4  // m <- matrix(v, nrow=2, ncol=3)
5  NumericMatrix m( 2, 3, v.begin());
```

The matrix object on Line 4 above is filled using a `Vector` object named `v`. This is facilitated with the *Rcpp* `Vector` member function `begin()` (Section 9.3.1.2).

Below is a `Rcpp::Dataframe` with columns comprised of `Vectors` named `v1` and `v2`.

```
// df <- data.frame(v1, v2)
DataFrame df = DataFrame::create(v1, v2);
```

Here is a `Rcpp::List` containing `Vectors` `v1` and `v2`.

[24] C++11, released in 2011, replaced the prior C++ standard, C++03, and was itself replaced by C++ standard C++14 in 2014. The current standard is C++23.

```
// L <- list(v1, v2)
List L = List::create(v1, v2);
```

■

9.3.1.2 Member Functions

Rcpp has useful C++ *member functions* (functions that can be used to interact with data of specific user-defined types) for its `Vector`, `Matrix`, `List` and `Dataframe` types. Specifically, for a member function `foo` that corresponds to a type defined for an object `bar`, I would run `foo` on `bar` by typing `bar.foo()`. Note that *Rcpp* member functions in Table 9.4 with generic names, e.g., `length()` are analogous to **R** methods for particular S3 and S4 classes (Section 8.7).

Table 9.4: Some C++ member functions for *Rcpp* types. To run a member function `mfunc()` on an appropriate object `x`, I would type: `x.mfunc()`.

Function	Vector	Matrix	Dataframe	List	Operation
`length(), size()`	X		X	X	Returns length, or no. `Dataframe` columns
`names()`	X		X	X	Names attribute
`sort()`	X			X	Sorts object into ascending order
`get_NA()`	X			X	Returns `NA` values
`is_NA(x)`	X			X	Returns `true` if element `x` is `NA`
`nrows()`		X	X		Returns number of rows
`ncols()`		X	X		Returns number of columns
`begin()`	X		X	X	Returns iterator pointing to first element
`end()`	X		X	X	Returns iterator pointing to end of object
`fill_diag(x)`		X			Fill `Matrix` diagonal with scalar `x`

9.3.1.3 Math with R-like Functions

Rcpp contains **R**-like functions that extend C++ `std` mathematical procedures evaluated under the C `<math.h>` header file, or the C++ `<cmath>` header. The *Rcpp* functions allow users to capitalize on the vectorized efficiencies of **R**, within C++ scripts, while using **R**-like grammar. Table 9.5 shows simple mathematical operators and functions that are generally applicable to both scalar and `Rcpp::Vector` objects. Conversely, Table 9.6 shows vectorized *R*-like functions from *Rcpp*, without analogues in `<math.h>`.

It is important to note that C++, like many other languages including C, and Fortran Python will often generate integer results from mathematical operations, even though they should be double precision. This can be readily demonstrated using `Rcpp::evalCpp()`.

Table 9.5: C++ `math.h` functions for scalars: `s`, `s1`, and `s2`, and **R**-like *Rcpp* functions for a `Vector`: `v`, `v1`, `v2`.

Operation	C++ scalar	*Rcpp* `Vector`	Description
addition	`s1 + s2`	`v + s or v1 + v2`	scalar or vector (elementwise) addition
subtraction	`s1 - s2`	`v - s or v1 - v2`	scalar or vector (elementwise) division
multiplication	`s1 * s2`	`v * s or v1 * v2`	scalar or vector (elementwise) division
division	`s1 / s2`	`v / s or v1 / v2`	scalar or vector (elementwise) division
modulo	`s1 % s2`		remainder of division of `s1` by `s1`
$\mid x \mid$	`abs(s)`	`abs(v)`	absolute value(s) of `s` or elements in `v`.
round	`round(s,d)`	`round(v,d)`	rounds `s` or elements in `v` to `d` digits.
$\sqrt{x}$	`sqrt(s)`		square root of `s`
$\log_2$	`log2(s)`		$\log_2$ of `s`.
$\log_e$	`log(s)`	`log(v)`	$\log_e$ of `s` or elements in `v`.
$\log_{10}$	`log10(s)`	`log10(v)`	$\log_{10}$ of `s` or elements in `v`.
$\log_e$	`log(s)`	`log(v)`	$\log_e$ of `s` or elements in `v`.
e^x	`exp(s)`	`exp(v)`	$\exp()$ of `s` or elements in `v`.
x^n	`pow(s, n)`	`pow(v,n)`	raises `s` or elements in `v` to `n`th power.
$\sin(x)$	`sin(s)`	`sin(v)`	sine of `s` or elements in `v`.
$\cos(x)$	`cos(s)`	`cos(v)`	cosine of `s` or elements in `v`.
$\tan(x)$	`tan(s)`	`tan(v)`	tangent of `s` or elements in `v`.
$\text{asin}(x)$	`asin(s)`	`asin(v)`	arcsine of `s` or elements in `v`.
$\text{acos}(x)$	`acos(s)`	`acos(v)`	arccosine of `s` or elements in `v`.
$\text{atan}(x)$	`atan(s)`	`atan(v)`	arctangent of `s` or elements in `v`.

Example 9.9.
Clearly the answer to $\frac{5}{2}$ is 2.5. However, running this operation in C++ produces:

```
evalCpp("5/2")
```

```
[1] 2
```

One way around this is to add a decimal to the end of the 5 and 2, to indicate that they are not integers. Revisit Example 9.6 for Fortran and C examples of this approach.

```
evalCpp("5./2.")
```

```
[1] 2.5
```

■

9.3.1.4 Inline C++ Code

The function `Rcpp::cppFunction()` allows users to specify C++ code for a single function as a character string at the **R** command line (see minimal Example 9.7 above). The function compiles C++ code, and creates a link to the resulting shared library. It then defines an **R** function that uses `.Call()` to invoke the library.

Table 9.6: **R**-like *Rcpp* functions specific to `Vector`, `v`, objects.

Operation	*Rcpp* `Vector`, `v`	Description
$\min(x)$	`min(v)`	minimum value of `v`
$\max(x)$	`max(v)`	maximum value of `v`
$\sum_{i=1}^{n} x_i$	`sum(v)`	sum of `v`
cumulative sum	`cumsum(v)`	cumulative sum of `v`
cumulative product	`cumprod(v)`	cumulative product of `v`
range	`range(v)`	min and max of `v`
$\bar{x}$	`mean(v)`	mean of `v`
$\tilde{x}$	`median(v)`	median of `v`
s	`sd(v)`	standard deviation of `v`
s^2	`var(v)`	variance of `v`
C++ version of **R** function	`sapply(v,fun)`	applies C++ function `fun()` to `v`
C++ version of **R** function	`lapply(v,fun)`	applies C++ function `fun()` to `v`; returns `List`
C++ version of **R** function	`cbind(x1, x2,...)`	combines `Vector` or `Matrix` in `x1`, `x2`
C++ version of **R** function	`na_omit(v)`	returns `Vector` with `NA` elements in `v` deleted
C++ version of **R** function	`is_na(v)`	labels `NA` elements in `v` `TRUE`

Example 9.10.
Here is a simple function for generating numbers from a Fibonacci sequence. See Question 6 in the Exercises from Ch 8.

```
1 cppFunction(
2     'int fibonacci(const int x) {
3         if (x == 0) return(0);
4         if (x == 1) return(1);
5         return (fibonacci(x - 1)) + fibonacci(x - 2);
6     }')
```

- On Line 2, the C++ function name `finbonacci` is defined. The function output and the class of the argument `x` are both defined to be `int` (integers).
- On Lines 3-4 the first two numbers in the sequence are defined based on Boolean operators.
- On Line 5, later numbers in the sequence ($n > 2$) are defined.

The result from the script is an **R** function that loads the compiled shared library, based on the C++ function `fibonacci`, using `.Call()`.

```
fibonacci
```

```
function (x)
.Call(<pointer: 0x00007ffc41a01860>, x)
```

Here we use the **R** function to generate the 10th Fibonacci number.

```
fibonacci(10)
```

```
[1] 55
```

■

The **R** function `Rcpp::sourceCpp()` allows general compilation of C++ scripts that may contain *multiple* functions.

9.3.1.5 Formal C++ Scripts

We can use *Rcpp* to facilitate the creation of more conventional C++ scripts (not just character strings of C++ code). These will have the general form (Tsuda, 2020):

```
1 #include <Rcpp.h>
2 using namespace Rcpp;
3
4 // [[Rcpp::export]]
5 RETURN_TYPE FUNCTION_NAME(ARGUMENT_TYPE ARGUMENT){
6   //function contents
7   return RETURN VALUE;
8 }
```

- On Line 1, the code `#include <Rcpp.h>` loads the *Rcpp* header file `Rcpp.h`. In several C-alike languages (C, C++, C-obj), header files can be use to provide definitions for functions, variables, and (in the case of C++) new class definitions (Table 9.3). See Chapter 6 in R Core Team (2024c).
- The (optional) code `using namespace Rcpp` (Line 2) allows direct access to *Rcpp* classes and functions. Without this designation, an *Rcpp* function or class `foo` would require the call `Rcpp::foo`, instead of simply, `foo`.
- The comment: `// [[Rcpp::export]]` (Line 4) serves as a compiler *attribute*, and demarks the beginning of C++ code that will be accessible from **R**. The `Rcpp::export` attribute is required (by *Rcpp*) for any C++ script to be run from **R**. The attribute currently requires specification as a comment, because it will be unrecognized within most compilers.
- For `RETURN_TYPE FUNCTION_NAME(ARGUMENT_TYPE ARGUMENT){` (Line 5) users must specify data types of functions, a function name, argument types, and arguments.
- `return RETURN VALUE;` is required if function output is desired.

As before, this process compiles the C++ code into shared library, and creates an **R** function (with the same name as the C++ function) that calls the shared library (Example 9.10). In **R** Markdown, one can check and debug this process by calling **R** to run this function from within the chunk containing the associated C++ script, by using the subsequent code form:

```
1 /*** R
2 FUNCTION_NAME
3 */
```

where `FUNCTION_NAME` is the name of the resultant **R** function.

Example 9.11.
RStudio provides an IDE for C++ scripts. Further, a C++ file obtained using **File>New File>C++** contains an example *Rcpp*-formatted C++ example function, named `timesTwo`, that multiplies some number by two:

```
1 #include <Rcpp.h>
2 using namespace Rcpp;
3
4 // [[Rcpp::export]]
5 NumericVector timesTwo(NumericVector x) {
6   return x * 2;
7 }
```

Note use of the *Rcpp* type `NumericVector` to define function output and values for the argument, `x` (Line 5).

Running the code above compiles `timesTwo` into a shared library, and creates an **R** function (with the same name) in the global environment. This function loads the shared library for use in **R**.

```
timesTwo(5)
```

```
[1] 10
```

■

Example 9.12.
As a series of biological examples, we will create C++ functions (using *Rcpp* tools) for measuring the diversity of ecological communities. Below is a function for calculating relative abundances of species in a community (individual species abundance divided by the sum of species abundances).

```
1 #include <Rcpp.h>
2 using namespace Rcpp;
3
4 // [[Rcpp::export]]
5 NumericVector relAbund(NumericVector x) {
6   int n = x.length();
7   double total = 0;
8   for(int i = 0; i < n; ++i) {
9     total += x[i];
10  }
11  NumericVector rel = x/total;
12  return rel;
13 }
```

The function `relAbund` is a mixture of standard C++ code and calls to C++ classes and procedures from *Rcpp*. In particular,

- On Lines 1 and 2, I bring in the `Rcpp.h` header file, and load the *Rcpp* namespace.
- On Line 4, I include the comment, `// [[Rcpp::export]]` to prompt **R** to recognize code below the line.
- On Line 6, I specify the data types of the function output, `NumericVector`, the function name, the data type for the argument `NumericVector`, and the argument itself, `x`.
- Lines 7-8 are preliminary steps for the loop codified on Lines 9-11. On Line 7, an integer object `n` is created by find the number of observation in `x`. This is done with the *Rcpp* `Vector` member function `length()` (Table 9.4) with the call `x.length()`.
- Lines 9-11 comprise a standard C/C++ looping approach for calculating total abundance (the sum of `x`). The useful operator `+=` adds the right operand to the left operand and assign the result to the left operand.
- On Lines 12-13 relative abundance are calculated and the resulting `NumericVector` is returned.

Recall (Example 6.17) that the dataset `vegan::varespec` describes the abundance of vascular plants, mosses, and lichen species for sites in a Scandinavian taiga/tundra ecosystem. Here I run the function for the site represented in row 1 (site 18).

```
library(vegan)
data(varespec)

relAbund(as.vector(varespec[1,], "double"))
```

```
 [1] 0.00616592 0.12477578 0.00000000 0.00000000 0.19955157 0.00078475
 [7] 0.00000000 0.00000000 0.01793722 0.02320628 0.00000000 0.01816143
[13] 0.00000000 0.00000000 0.05235426 0.00022422 0.00145740 0.00000000
[19] 0.00145740 0.00134529 0.00000000 0.24360987 0.24069507 0.03923767
[25] 0.00336323 0.00201794 0.00257848 0.00280269 0.00280269 0.00257848
[31] 0.00000000 0.00000000 0.00089686 0.00022422 0.00022422 0.00000000
[37] 0.00134529 0.00022422 0.00695067 0.00022422 0.00000000 0.00000000
[43] 0.00280269 0.00000000
```

I ensure that the C++ shared library `relAbund` views `varespec[,1]` as double precision by specifying `mode = "double"` in `as.vector()`.

Recall (Example 8.21) that species relative abundances are used in calculating measures of α-diversity. The code below calculates Simpson diversity (Eq. (8.4)) from a vector of abundance data.

```
1 #include <Rcpp.h>
2 #include <cmath>
3 using namespace Rcpp;
4
5 // [[Rcpp::export]]
6 double simpson(NumericVector x) {
```

```
7   NumericVector y = na_omit(x);
8   double total = sum(y);
9   NumericVector relsq = pow(y/total, 2);
10  return 1 - sum(relsq);
11 }
```

Note that on Line 7, I have dramatically simplified the calculation of relative abundance by replacing the for loop in `relAbund` with the **R**-like `Vector` function `Rcpp::sum()` (Table 9.5). Other **R**-like C++ functions used above include `na_omit()` (Line 6) `Rcpp::pow()` and (Line 8). The former allows handling data with missing values.

```
simpson(as.vector(varespec[1,], mode = "double"))
```

```
[1] 0.82171
```

■

Example 9.13.
The code below shows how one would run some simple mathematical operation in C++ (see Table 9.5) that combine C++ scripting at the **R** command line with formal C++ grammar, including header files.

```
1  src <-
2  '
3  #include <Rcpp.h>
4  #include <math.h>
5
6  using namespace Rcpp;
7  // [[Rcpp::export]]
8
9  List math_demo(){
10   double a = sin(3);
11   double b = log(3);
12   double c = log2(3);
13   NumericVector v = {1,2,3};
14   double d = min(v);
15   NumericVector e = log(v);
16   return List::create(Named("a") = a,
17                       Named("b") = b,
18                       Named("c") = c,
19                       Named("d") = d,
20                       Named("e") = e);
21 }'
22
23 sourceCpp(code = src)
24 math_demo()
```

```
$a
[1] 0.14112

$b
[1] 1.0986

$c
[1] 1.585

$d
[1] 1

$e
[1] 0.00000 0.69315 1.09861
```

■

- The entire C++ script (Lines 2-21) is written into a character string, and assigned the name `src`.
- The first lines of C++ code include calls to both the `Rcpp.h` and `math.h` header files (Lines 3-4), application of the *Rcpp* namespace (Line 6), and designation of `// [[Rcpp::export]]` (Line 7).
- Lines 9-21 codify the C++ function `math_demo`. The function is argumentless (it is meant to demonstrate mathematics using object generated in the function itself) and will return an *Rcpp* `List` (Line 9).
- Lines 10-12 are simple scalar operations using `math.h` functions.
- Lines 13-15 use *Rcpp* `Vector` approaches.
- A `List` containing the generated objects, `a`, `b`, `c`, `d`, and `e` is built and return on Lines 16-20.

9.3.1.6 Accessing/Manipulating Data Types Components

Rcpp data type objects can generally be subset using `()`, `[]`, or with member functions. Both `()` and `[]` can be used with `Rccp::NumericVector`, `Rcpp::IntegerVector` and `CharacterVector` types. `Rcpp::Dataframe` objects require `[]`, whereas `Rcpp::Matrix`, require `()` for subsetting.

Example 9.14.
Here is a long-winded C++ function that demonstrates *Rcpp* subsetting using `Rcpp::NumericVector` objects, an `Rcpp::NumericMatrix` object, and an `Rcpp::Dataframe` object.

```
1 #include <Rcpp.h>
2 using namespace Rcpp;
3 // [[Rcpp::export]]
4 List subsets(){
```

```
  // Create Vectors
  NumericVector nv = {10, 20, 30, 40, 50, 60};
  nv.names() = CharacterVector({"A","B","C","D","E","F"});
  NumericVector nv2 = nv + 1;
  NumericVector nv3 = nv + nv2; // Rcpp allow elementwise Vector operations
  // Create Matrix
  NumericMatrix nm(2, 3, nv.begin());
  // Create Dataframe
  DataFrame df = DataFrame::create(Named("V2") = nv2, Named("V3") = nv3);
  // Indexes
  NumericVector id1 = {1,3};
  CharacterVector id2 = {"A","D","E"};
  LogicalVector id3 = {false, true, true, true, false, true};
  // Vector subsets based on indexes
  int x1 = nv[0];
  int x2 = nv["C"];
  NumericVector x3 = nv[id1];
  NumericVector x4 = nv[id2];
  NumericVector x5 = nv[id3];
  // Matrix subsets
  double x6 = nm(0 , 1); // Row 0 (first row) and column 1 (2nd column)
  NumericVector x7 = nm(1 , _ );  // Row 1 (2nd row)
  NumericVector x8 = nm( _ , 0);  // Column 0 (1st column)
  NumericVector x9 = nm.column(0);  // Column 0 (1st column)
  //Dataframe subsets
  NumericVector x10 = df[0];
  NumericVector x11 = df["V3"];

return List::create(Named("Result1") = x1, Named("Result2") = x2,
                    Named("Result3") = x3, Named("Result4") = x4,
                    Named("Result5") = x5, Named("Result6") = x6,
                    Named("Result7") = x7, Named("Result8") = x8,
                    Named("Result9") = x9, Named("Result10") = x10,
                    Named("Result11") = x11);
}
```

```
subsets()
```

```
$Result1
[1] 10

$Result2
[1] 30

$Result3
 B  D
20 40
```

```
$Result4
 A  D  E
10 40 50

$Result5
 B  C  D  F
20 30 40 60

$Result6
[1] 30

$Result7
[1] 20 40 60

$Result8
[1] 10 20

$Result9
[1] 10 20

$Result10
[1] 11 21 31 41 51 61

$Result11
[1]  21  41  61  81 101 121
```

- As before, I call the `Rcpp.h` header file, apply the *Rcpp* namespace, and designate the attribute `// [[Rcpp::export]]` (Lines 1-3).
- On Line 4, the C++ function `subsets` is defined to have `List` output. No arguments are defined because the goal is to demonstrate *Rcpp* data type subsetting and manipulation, using only objects created within the function.
- On Lines 6-9, I create three `NumericVector` objects. The latter two are on elementwise transformations facilitated by *Rcpp sugar* operators.
- On Line 11, I create a `NumericMatrix` filled with elements from the `Vector` `nv`, using the `Matrix` deque member function `begin()`. Note that *Rcpp* matrices are built by column, given a vector input.
- On Line 13, I create a two column `Dataframe` comprised of the `Vector` objects `nv2` and `nv3`. using the `Matrix` deque member function `begin()`.
- On Line 15-17, three `Vector` objects that will be used for subsequent subsetting are created.
- On Lines 19-23, the objects `x1`, `x2`, `x3`, `x4` and `x5` are created by subsetting the `Vector`, `nv`.
- On Lines 25-28, the objects `x6`, `x7`, `x8`, and `x9` are created by subsetting the `Matrix`, `nm`.
- On Lines 30-31, the objects `x10` and `x11` are created by subsetting the `Dataframe`, `df`.
- On Lines 33-38, the subset objects are assembled into a `List` and are returned by the function.

■

Example 9.15.
We now know enough to extend our scalar function for Simpson's diversity (Example 9.12) to a function that can handle matrices –the conventional format for biological community datasets.

```
1  #include <Rcpp.h>
2  using namespace Rcpp;
3  // [[Rcpp::export]]
4  NumericVector simpson(NumericMatrix x) {
5    CharacterVector rn = rownames(x);
6    NumericVector out = x.nrow();
7    out.names() = rn;
8    int n = out.size();
9
10     for(int i = 0; i < n; ++i) {
11     NumericVector temp = na_omit(x(i , _ ));
12     double total = sum(temp);
13     NumericVector relsq = pow(temp/total, 2);
14     out[i] = 1 - sum(relsq);
15     }
16
17   return out;
18 }
```

- As in previous examples, I first call the `Rcpp.h` header file, apply the *Rcpp* namespace, and define the `// [[Rcpp::export]]` compiler attribute (Lines 1-3).
- The function output will be a `NumericVector` (of Simpson's diversities of sites) and will require a `NumericMatrix` for its argument `x`, with sites in rows and species in columns (Line 4).
- Lines 5-8 generate objects (`out` and `n`) that will be used in a subsequent loop.
- Lines 10-15 define a loop that populates `out` with Simpson's diversities. The code: `NumericVector temp = na_omit(x(i , _ ));` creates a `NumericVector` object, `temp`, consisting of non-missing values in the ith row of `x`.
- On Line 17 `out` is returned.

Here we apply our function to the entire `vegan::varespec` dataset.

```
simpson(as.matrix(varespec))
```

```
     18      15      24      27      23      19      22      16      28
0.82171 0.76276 0.78101 0.74414 0.84108 0.81819 0.80310 0.82477 0.55996
     13      14      20      25       7       5       6       3       4
0.81828 0.82994 0.84615 0.83991 0.70115 0.56149 0.73888 0.64181 0.78261
      2       9      12      10      11      21
```

```
0.55011 0.49614 0.67568 0.50261 0.80463 0.85896
```

We see that our C++ function is much faster than the widely-used function `vegan::diversity()`, which relies on an **R** for loop.

```
m <- matrix(nrow = 10^6, ncol = 10, data = rnorm(10^7) + 10)
system.time(simpson(m))
```

```
   user  system elapsed
   0.32    0.05    0.36
```

```
system.time(vegan::diversity(m, "simpson"))
```

```
   user  system elapsed
   2.23    0.12    2.37
```

■

9.3.2 The inline package

The *inline* **R** package (Sklyar et al., 2025) extends the capacities of `Rcpp::evalCpp()`, `Rcpp::cppFunction()` and `Rcpp::sourceCpp()` by allowing users to create, compile, and run functions written in any language supported by `R CMD SHLIB`, including C, Fortran, C++, and C-obj, from the **R** command line.

Example 9.16.
Consider the following example –based on `? inline::cfunction()`– of a simple C function that raises every value in a numeric vector to the third power.

```
library(inline)

code.cube <- "
      int i;
      for (i = 0; i < *n; i++)
        x[i] = x[i]*x[i]*x[i];
"
cube.fn <- cfunction(signature(n="integer", x="numeric"), code.cube,
                     language = "C", convention = ".C")

cube.fn(20, 1:20)
```

```
$n
[1] 20

$x
 [1]    1    8   27   64  125  216  343  512  729 1000 1331 1728 2197 2744
[15] 3375 4096 4913 5832 6859 8000
```

Note that `code.cube`, is a character string containing C-script (Lines 3-7). The script on Lines 9 and 10 calls `inline::cfunction()` to compile the string into a C shared library executable using `SHILB`. The shared library will be called automatically using `.C()`, allowing `cube.fn` used as an **R** function on Line 9. The object `cube.fn` has an unusual combination of characteristics. It is a function of base type `closure`:

```
typeof(cube.fn)
```

```
[1] "closure"
```

However, it is also S4,

```
isS4(cube.fn)
```

```
[1] TRUE
```

with the following slots:

```
slotNames(cube.fn)
```

```
[1] ".Data" "code"
```

The `code` slot can be obtained using the function `inline::code()`

```
code(cube.fn)
```

```
 1: #include <R.h>
 2:
 3:
 4: void fileb02843d186a ( int * n, double * x ) {
 5:
 6:       int i;
 7:       for (i = 0; i < *n; i++)
 8:         x[i] = x[i]*x[i]*x[i];
 9:
10: }
```

Note that the text string has been converted to a C `void` function, as required by `SHLIB`. The header call `#include <R.h>` provides a built-in **R** API for C code.

The `.Data` slot contains **R** code that is run by interpreter when `cube.fn()` is called. Note that the function uses `.Primitive()` to call the appropriate shared library by way of its object address/pointer.

```
cube.fn@.Data
```

```
function (n, x)
.Primitive(".C")(<pointer: 0x00007ffc9aa61380>, n = as.integer(n),
```

```
    x = as.double(x))
<environment: 0x0000012da423be38>
```

■

9.4 SQL and Databases

Biological databases have grown exponentially in size and number (Sima et al., 2019). Because of this trend, biological databases are often housed in web-accessible warehouses including the National Center for Biotechnology Information (NCBI), data*Base* for Gene Expression Evolution (Bgee), and the European life-sciences infrastructure for biological information (ELIXIR). The Posit website provides a nice resource for working with databases in **R**.

Databases are often assembled in a *Database Management System* (DBMS) format. A DBMS will contain one or more rectangular row/column storage units called *tables*. Rows in tables are called *records* and columns are called *fields* or *attributes*.

Many DBMS formats have evolved based on the *Structured Query Language* (SQL). Although SQL is an American National Standards Institute (ANSI) and International Organization for Standardization (ISO) standard, there are many variants of SQL, and software for managing these languages is often proprietary (e.g., Oracle, Microsoft SQL Server) and potentially expensive. Despite this variety, SQL dialects generally use the same basic SQL commands (Table 9.7), and processes. For example, as a general rule, SQL table fields can be accessed with a period operator. That is, a column, `bar`, in table `foo` is specified as `foo.bar`. SQL guidance can be found at a large number of websites, including the developer site W[3].

Table 9.7: Important SQL commands. Out of convention, SQL commands here are shown in upper-case. SQL keywords, however, are not case sensitive. That is, `select` is the same as `SELECT`.

Command	Meaning
`SELECT`	Extracts data from a database
`FROM`	Used with `SELECT`. A clause identifying a database
`UPDATE`	Updates data in a database
`DELETE`	Deletes data from a database
`CREATE TABLE`	Creates a new table
`WHERE`	Filters records from a table
`AND`	Filters records based on more than one condition
`OR`	Filters records based on more than one condition
`BETWEEN`	Selects values within a given range

9.4.1 DBI

The **R** package *DBI* (R Special Interest Group on Databases (R-SIG-DB) et al., 2024; James, 2009) currently allows communication with 30 SQL-driven DBMS formats. Each supported *DBI*

DBMS uses its own **R** package. For instance, the SQLite DBMS is interfaced with the package *RSQLite* (which will be installed with *DBI*), and the MySQL DBMS can be interfaced using the package *RMySQL*. The *RMariaDB* package can be used to interface either MySQL or the DBMS MariaDB. Opening a DBMS connection will constrain users to the SQL nuances of the selected DBMS. We will concentrate on the non-proprietary DBMS SQLite here.

```
library(DBI)
```

```
library(RSQLite)
```

Example 9.17.
As a first example, we will create a database using "internal" **R** dataframes. First we establish a SQLite DBMS connection using `dbConnect()`.

```
con <- dbConnect(SQLite(), ":memory:")
con
```

```
<SQLiteConnection>
  Path: :memory:
  Extensions: TRUE
```

Unlike many other DBMS frameworks that may require a username, password, host, port, and other information, SQLite only requires a path to the database[25]. The argument `":memory:"` specifies a special path that results in an "in-memory" database.

Notably, the `con` database is an S4 object:

```
isS4(con)
```

```
[1] TRUE
```

Here we append the `asbio::world.emission` dataframe to the database using `dbWriteTable()`.

```
library(asbio)
data(world.emissions)
dbWriteTable(con, "emissions", world.emissions)
```

We see that the table (renamed `emissions`) now exists in the database.

```
dbListTables(con)
```

```
[1] "emissions"
```

Below we use SQL script (within the **R** function `DBI::dbSendQuery()`) to access information

[25] Use of an MySQL DBMS might require something like: `` `Mycon <- dbConnect(RMySQL::MySQL(), host = "your_host", user = "your_user", password = "your_password", dbname = "your_database")` ``.

from the database table `emissions`. In particular –using the commands `SELECT`, `FROM`, `WHERE`, `AND`, and `BETWEEN`– I query the columns `coal_co2` and `gas_co2`, with respect to the `United States`, for the years 2016 to 2019.

```
us <- dbSendQuery(con, "SELECT coal_co2, gas_co2
                  FROM emissions
                  WHERE country = 'United States'
                  AND year BETWEEN 2016 AND 2019")
```

To access *all* columns from `emissions`, I could have used the SQL command: `"SELECT * FROM emissions`.

Here I fetch the query result using `dbFetch()`:

```
us.fetch <- dbFetch(us)
us.fetch
```

```
  coal_co2 gas_co2
1   1378.2  1509.0
2   1337.5  1491.8
3   1282.1  1653.0
4   1094.7  1706.9
```

The fetched result is a dataframe.

```
class(us.fetch)
```

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

One should clear queries using `DBI::dbClearResult()`. This will free all computational resources (local and remote) associated with a query result.

```
dbClearResult(us)
```

Databases can contain multiple tables. Here I append the `asbio::C.isotope` dataframe to the database:

```
data(C.isotope)
dbWriteTable(con, "isotopes", C.isotope)
```

There are now two tables in the database, although they are not relational.

```
dbListTables(con)
```

```
[1] "emissions" "isotopes"
```

When finished accessing a DBMS, one should **always** close the DBMS connection.

```
dbDisconnect(con)
```

■

In **R** Markdown one can use the SQL variant of the chosen DBMS directly, by specifying ```` ```{sql, connection = con}``` ```` when initiating code chunks, where `con` is the name of the database connection (see Section 9.1.3). This approach is often required for complex operations.

Example 9.18.
Reconsidering Example 9.17 we have:

```
con <- dbConnect(SQLite(), ":memory:")
dbWriteTable(con, "emissions", world.emissions)
```

Here I directly specify an SQL query (in SQL).

```
SELECT coal_co2, gas_co2
FROM emissions
WHERE country = 'United States'
AND year BETWEEN 2016 AND 2019;
```

coal_co2	gas_co2
1378.2	1509.0
1337.5	1491.8
1282.1	1653.0
1094.7	1706.9

Reflecting the requirements of several DBMS variants, I end SQL the statements above with a semicolon, `;`.

```
dbDisconnect(con)
```

■

9.4.2 Relational DBMS

Thus far, the justification for an interfaced DBMS may seem vague, since similar data management results could be obtained from **R** lists.

The advantages of creating a DBMS become clearer when considering a *relational* DBMS (RDBMS). An RDBMS allows the straightforward linking of multiple database tables via a common value identifier stored in the tables (Fig 9.2).

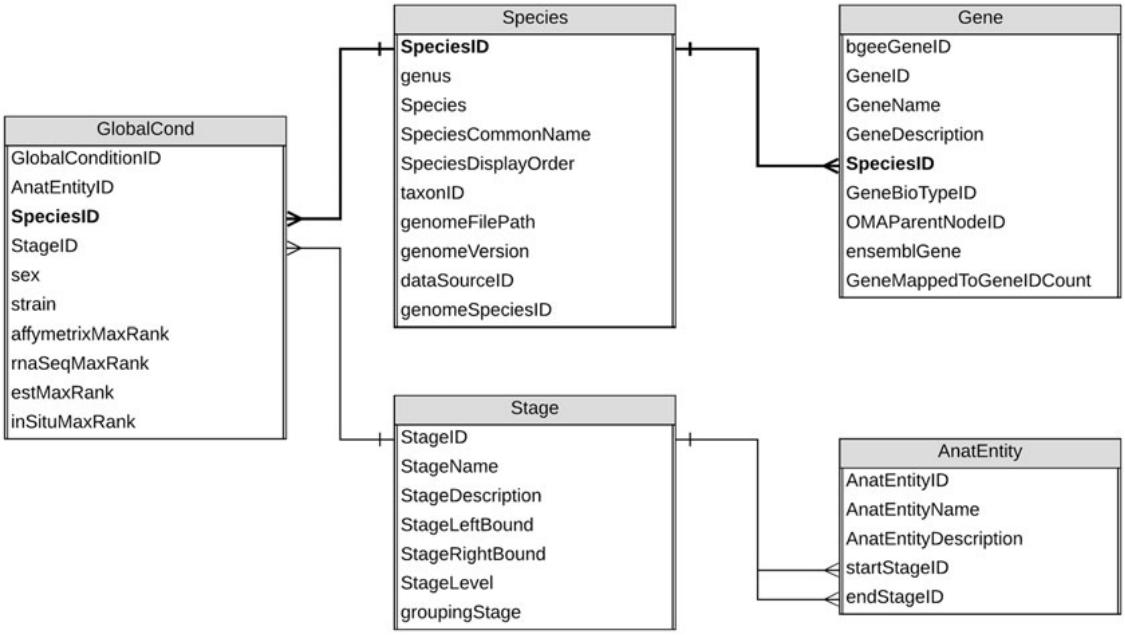


Figure 9.2: A relational database from the gene expression database Bgee. Several tables are linked via the identifier **SpeciesID**. Figure taken from Sima et al. (2019).

Example 9.19.
In this example we will impart relational characteristics to a database based on two **R** dataframes, `asbio::Rabino_CO2` and `asbio::Rabino_del13C`, obtained from (Rubino et al., 2013). The datasets record CO_2 and $\delta^{13}C$ levels from Law Dome and South Pole, Antarctica for a 1000 year timespan. Exact effective date records, precision, and measurement depths all vary for the entries (see Example 7.5), prompting the creation of two separate datasets.

First, I create mean effective date records to eventually provide a single-entry label field for each dataset, based on the `effective.age` of samples.

```
data(Rabino_CO2)
data(Rabino_del13C)

library(tidyverse)
AvgCO2df <- Rabino_CO2 |>
  group_by(effective.age) |>
  summarise(AvgDdepth = mean(depth),
            AvgCO2 = mean(CO2),
            AvgUncertainty = mean(uncertainty))

Avg13Cdf <- Rabino_del13C |>
  group_by(effective.age) |>
  summarise(AvgDepth = mean(depth),
            Avgd13C = mean(d13C.CO2),
```

```
            AvgUncertainty = mean(uncertainty))

names(Avg13Cdf)[1] <- names(AvgCO2df)[1] <- "EffectiveAge"
AvgCO2df$EffectiveAge <- as.integer(unlist(AvgCO2df[,1]))
Avg13Cdf$EffectiveAge <- as.integer(unlist(Avg13Cdf[,1]))
```

The resulting summary dataframes, `AvgCO2df` and `AvgC13df`, do not contain measures from the same effective dates. Specifically, 114 (out of 189) `AvgCO2df` effective age records do not occur in `AvgC13df`.

```
length(AvgCO2df$EffectiveAge) -
  length(which(AvgCO2df$EffectiveAge %in% Avg13Cdf$EffectiveAge))
```

```
[1] 114
```

And 10 (out of 85) `AvgC13df` effective age records do not occur in `AvgCO2df`.

```
length(Avg13Cdf$EffectiveAge) -
  length(which(Avg13Cdf$EffectiveAge %in% AvgCO2df$EffectiveAge))
```

```
[1] 10
```

Nonetheless, we can easily join the datasets in a DBMS, and use their effective ages, to simultaneously query them.

We first request a SQLite database connection.

```
con <- dbConnect(SQLite(), ":memory:")
```

We then add `AvgCO2df` and `AvgC13df` to the database as tables.

```
dbWriteTable(con, "CO2", AvgCO2df)
dbWriteTable(con, "d13C", Avg13Cdf)
```

There are several database joins we can specify using SQL, including `LEFT JOIN` and `RIGHT_JOIN`. Assume that we have two tables in a database , `A` and `B`.

If I request `A LEFT JOIN B`, then the result set will include:

- Records in `A` and `B` *with* corresponding labels.
- Records (if any) in `A` *without* corresponding labels in `B`. In this case, `B` entries are given `NULL` values.

Conversely, if I request `A RIGHT JOIN B`, then the result set will include:

- Records in `B` and `A` *with* corresponding labels.
- Records (if any) in `B` *without* corresponding labels in `A`. In this case, `A` entries are given `NULL` values.

```
1 SELECT AvgCO2, d13C.Avgd13C, CO2.EffectiveAge
2 FROM CO2 LEFT JOIN d13C
3 ON d13C.EffectiveAge = CO2.EffectiveAge
4 WHERE CO2.EffectiveAge > 1990;
```

AvgCO2	Avgd13C	EffectiveAge
352.22	-7.8410	1991
353.73	-7.8820	1992
353.94	-7.8883	1993
357.11	NA	1994
359.65	NA	1996
361.78	-8.0600	1998
368.02	-8.0695	2001

In the SQL code above, I specify a `LEFT JOIN`.

- On Line 1, I specify the fields whose data I want to consider jointly, `AvgCO2`, `d13C.Avgd13C`, and the reference field I wish to use, `CO2.EffectiveAge`, i.e., the `EffectiveAge` field in the `CO2` table.
- On Line 2, I specify the join: `CO2 LEFT JOIN d13C`.
- On Line 3, I identify the fields used to join the tables.
- On Line 4, I limit the printed results to `CO2.EffectiveAge` values greater than 1990.

Note that in the output above there are two effective ages, 1994 and 1996, with CO_2 records but no $\delta^{13}C$ records.

```
1 SELECT AvgCO2, d13C.Avgd13C, CO2.EffectiveAge
2 FROM CO2 RIGHT JOIN d13C
3 ON d13C.EffectiveAge = CO2.EffectiveAge
4 WHERE CO2.EffectiveAge > 1990;
```

AvgCO2	Avgd13C	EffectiveAge
352.22	-7.8410	1991
353.73	-7.8820	1992
353.94	-7.8883	1993
361.78	-8.0600	1998
368.02	-8.0695	2001

The `RIGHT JOIN` SQL statement above is identical to the previous statement except for the Line 2 command: `CO2 RIGHT JOIN d13C`. In the output, complete $\delta^{13}C$ records for the requested effective age range are returned (note that ages 1994 and 1996 are omitted). While not required by the query, corresponding records for CO_2 also exist, and are reported.

```
dbDisconnect(con)
```

■

9.4.3 Creating an SQLite database

Thus far we have used package dataframes to populate an SQLite database connection. A more realistic application would be assembling an SQLite database from related but intentionally separated data files.

Example 9.20.

■

9.5 Python

Python, whose image logo is shown in Fig 9.3, is similar to **R** in several respects. Python was formally introduced in the early 90s, is an open source OOP language that is rapidly gaining popularity, and its source code is usually evaluated in an on-the-fly manner. That is Python, like **R**, is generally used as an interpreted language. Like **R**, comments in Python are made using the pound metacharacter, #[26], and many function calls have similar syntax.

Figure 9.3: The symbol for Python, a high-level, general-purpose, programming language.

There are, however, several fundamental differences between Python and **R**. These include the fact that while white spaces in **R** code (including tabs) simply reflect coding style preferences –for example, to increase code clarity– Python indentations denote code blocks[27]. That is, Python indentations serve the same purpose as **R** curly braces. Another important difference is that **R** object names can contain a . (dot), whereas in Python . means: "attribute in

[26] A Python comment spanning multiple lines can be implemented by enclosing the comment in triple quotes (""" or ''').

[27] In computer science this is called *significant indentation*, or the *off-side* rule.

a namespace." Thus, in Python the period operator . serves the same role as $ in **R** lists, dataframes and environments (Sections 3.1.4, 8.8.1.1). Recall that the . operator is used in a similar way in SQL language queries of database tables (Section 9.4). Python also uses member functions (Section 9.3.1.2) for class methods instead of generic function calls like **R**. IN Python, a method for an object of a specific class is specified using the period operator. Useful guidance for converting **R** code to analogous Python code can be found here.

Python can be downloaded for free from (https://www.python.org/downloads/), and can be activated from the Windows shells using the commands `py` or `python`, and activated from Mac and Unix/Linux shells using the command `python`. General guidance for the Python language can be found at (https://docs.python.org/) and many other sources including these books. Below I call Python from the Windows PowerShell command line.

```
PS C:\>py
Python 3.13.7 (tags/v3.13.7:bcee1c3, Aug 14 2025, 14:15:11) [MSC v.1944 64 bit (AMD64)] on win
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Note that the standard command line prompt for the Python shell is >>>. We can exit Python from the command line by typing `quit()`.

9.5.1 reticulate

```
Module(numpy)
```

Because our primary interest is interfacing Python and **R**, and not Python itself, we will use **R** as our base of operations. This will require the **R** package *reticulate* (Ushey et al., 2023).

```
# install.packages("reticulate")
library(reticulate)
```

RStudio (via *reticulate*) can be used as an IDE for Python[28]. In this capacity RStudio will:

- Generate a Python-specific environment (to provide separate settings for Python and **R** objects).
- Call separate **R** and Python environments, depending on which language is currently used in a chunk. Python code can be run directly in **R** Markdown by defining `python` (instead of `r`) as the first option in an **R** Markdown chunk (Section 9.1.3).

[28] Many IDEs have been developed specifically for Python, although quite a few are proprietary. Free IDEs include a primitive Python-bundled interface called IDLE (IDLE can be opened from the Windows command line using: `Path to python.exe\python.exe -m idlelib`), Jupyter Notebook, a web-based IDE, with many useful features, including support for **R** and Markdown-driven workflow documentation, Spyder, a widely used IDE, (e.g., Pine (2019)), Python Toolkit, which hasn't been updated for a while, and pycharm (which also has a commercial version).

9.5.2 Important Considerations for IDEs and APIs

Python packages are currently installed from one of two package repositories: the *Python Package Index* (*PyPI*) or *Anaconda*. The *Python Installer Package* (*pip*) is designed to install packages from *PyPI* (see Section 9.5.3). A repository manager named *conda* is used in conjunction with *Anaconda*, and its stripped-down repository *Miniconda*.

Using Python can be a headache if: 1) different versions of Python exist on your machine, and it is unclear which versions (if any) have access to necessary repositories[29], and/or 2) Python installations and packages are accessible under one manager (e.g., *pip*), but not another (e.g., *conda*). This is further complicated by the fact that Python APIs (like *reticulate*) or IDEs (like Spyder) may come with their own repository frameworks and default versions of Python.

With this mind, I can specify a path to a specific Python executable to be used in an **R**/*retuculate* session with `reticulate::use_python()`, and specify a repository path (and a path a particular Python executable) with `reticulate::use_condaenv()` and `reticulate::use_miniconda()`. The code below specifies use of my (external to `reticulate`) Miniconda environment as a package repository and Python executable path.

```
use_condaenv("C:/Users/ahoken/miniconda3/")
```

The version of Python used by `reticulate` can be accessed with `Sys.which()`, which finds full paths to program executables.

```
Sys.which("python")
```

```
                                    python
"C:\\Users\\ahoken\\MINICO~1\\python.exe"
```

More details concerning my Python configuration for *reticulate* are revealed with `reticulate::py_config()`:

```
reticulate::py_config()
```

```
python:         C:/Users/ahoken/miniconda3/python.exe
libpython:      C:/Users/ahoken/miniconda3/python313.dll
pythonhome:     C:/Users/ahoken/miniconda3
version:        3.13.5 | packaged by Anaconda, Inc. | (main, Jun 12 2025, 16:37:03) [MSC v.1929 64 bit (AMD64)]
Architecture:   64bit
numpy:          C:/Users/ahoken/miniconda3/Lib/site-packages/numpy
numpy_version:  2.3.1
numpy:          C:\Users\ahoken\MINICO~1\Lib\site-packages\numpy

NOTE: Python version was forced by use_python() function
```

A Python command line interface can be called directly in **R** using:

```
reticulate::repl_python()
```

[29] This is also important because specific versions of Python may dramatically affect the usability of basic Python functions.

Python can be closed from the resulting interface (returning one to **R**) by typing:

```
exit
```

Example 9.21.

The following are Python operations, run directly from RStudio.

```
2 + 2
```

```
4
```

The Python assignment operator is =.

```
x = 2
x + x
```

```
4
```

Here we see the aforementioned importance of indentation.

```
if x < 0:
  print("negative")
else:
  print("positive")
```

```
positive
```

Lack of an indented "block" following `if` will produce an error. Indentations in code can be made flexibly (e.g., one space, two space, tab, etc.) but they should be used consistently.

■

9.5.3 Packages

Like **R**, Python consists of a core language, a set of built-in functions, modules, and libraries (i.e., the *Python standard library*), and a vast collection ($> 200,000$) of supplemental libraries. Imported libraries are extremely important in Python because its distributed version has limited functional capabilities (compared to **R**). A number of important Python supplemental libraries, each of which contain multiple packages, are shown in Table 9.11.

We can install Python packages and libraries using the *pip* package manager for Python or *conda* (Section 9.5.2). Installation only needs to occur once on a workstation (similar to `install.packages()` in **R**). Following installation, one can load a package for a particular work session using the Python function `import` (analogous to `library()` in **R**)[30].

[30] Loading Python libraries (aside from *numpy*) in `reticulate` will produce an error if one specifies a Python location in `use_python()` that does not contain the installed libraries.

Table 9.11: Important supplemental Python libraries. For more information use hyperlinks.

Library	Purpose
sumpy	Fundamental package for scientific computing
scipy	Mathematical functions and routines
matplotlib	2- and 3-dimensional plots
pandas	Data manipulation and analysis
sympy	Symbolic mathematics
bokeh	Interactive data visualizations

Installation of a Python package, *foo*, with *reticulate*, can be accomplished using the function `reticulate::py_install` (in **R**)[31].

```
py_install("foo")
```

Example 9.22.
I wish to install the *ecologits* library, and its dependencies in the *openAI* library. This requires use of *pip*, via *conda*. Hence, I use the command:

```
py_install("ecologits[openai]", method = "conda", pip = TRUE) # Run in R
```

To load the *ecologits* library I use the Python function `import()`:

```
import ecologits
```

■

9.5.4 Functions in Packages

Functions within Python packages are obtained using a `package.function` syntax. Here I import *numpy* and run the function `pi` (which is contained in *numpy*).

```
import numpy
numpy.pi
```

```
3.141592653589793
```

If we are writing a lot of *numpy* functions, Python will allow you to define a simplified library prefix. For instance, here I created a shortcut for *numpy* called `np` and use this shortcut to access the *numpy* functions `pi()` and `sin()`.

[31]This approach generally works well. If problems occur loading libraries with `reticulate::py_install` one can download libraries from the command line using using *pip* or *conda*.

```
import numpy as np
np.sin(20 * np.pi/180) # sin(20 degrees)
```

```
np.float64(0.3420201433256687)
```

Use of the command `from numpy import *` would cause names of functions from *NumPy* to overwrite functions with the same name from other packages. That is, we could run `numpy.pi` simply using `pi`.

Example 9.23.
Here we import the package *pyplot* from the library *matplotlib*, rename the package `plt`, and create a plot (Fig 9.4) using the function `pyplot.plot()` (as `plt.plot()`) by calling:

```
1 import matplotlib.pyplot as plt
2 plt.plot(range(10), 'bo')
```

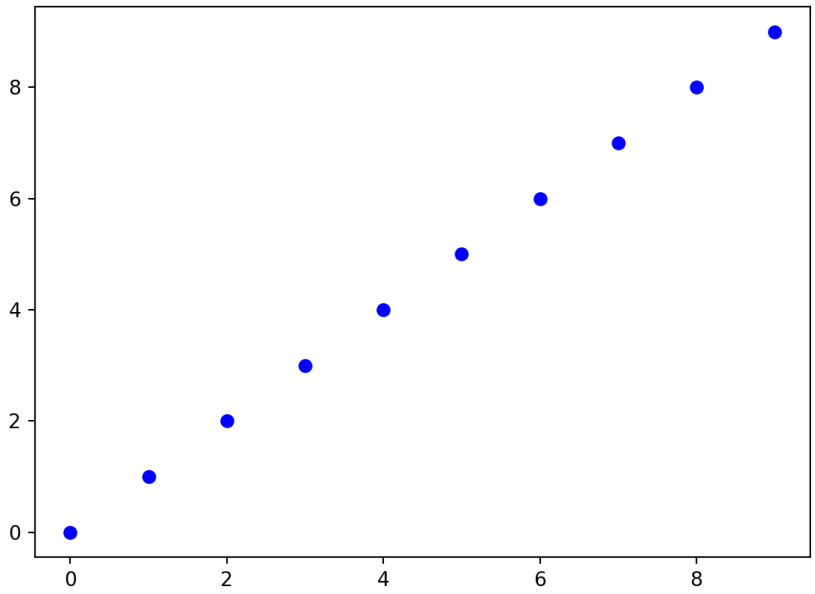


Figure 9.4: Creating a Python plot using **R**.

In Line 2, the command `range(10)` creates a sequence of integers from zero to ten. This is used as the first argument of `plt.plot()`, which specifies the plot x-coordinates. If y coordinates are not specified in the second argument, x-coordinates will be reused as y coordinates. The command `'bo'` places blue filled circles at x,y coordinates. Documentation for `matplotlib.pyplot.plot()` can be found at the matplotlib.org website.

■

9.5.5 Data Types

There are four major built-in dataset storage classes in Python: lists, tuples, sets, and dictionaries (Table 9.12).

Table 9.12: The four basic Python dataset storage classes.

Storage type	Example	Entry characteristics
List	`["hot","cold"]`	Changeable, Duplicates OK
Tuple	`("hot","cold")`	Unchangeable, Duplicates OK
Set	`{"hot","cold"}`	Unchangeable, Duplicates not OK
Dictionary	`{"temp":["hot", cold"]}`	Changeable, Duplicates not OK

All four classes track element order and can be used to simultaneously store different types of data, e.g., character string *and* numbers.

We can make a *Python* `list`, which can contain both text and numeric data, using square brackets or the function `list()`.

```
a = [20, 7, "Hi", 7, "end"]
```

Classes of Python objects can be identified with the Python function `type()`.

```
type(a)
```

```
<class 'list'>
```

An empty list can be specified as `[]`

```
empty = []
empty
```

```
[]
```

Like **R**, we can index `list` elements using square brackets. Importantly, like C-alike languages, indices start at 0. That is, `a[0]` refers to the first element of the list `a`.

```
a[0]
```

```
20
```

And the third element would be:

```
a[2]
```

```
'Hi'
```

As with **R**, square brackets can also be used to reassign `list` values

```
a[3] = 10
a
```

```
[20, 7, 'Hi', 10, 'end']
```

We can use the function `append()` to append entries to the end of a `list`. For instance, to append the number 9 to the object `a` in the previous example, I would type:

```
a.append(9)
a
```

```
[20, 7, 'Hi', 10, 'end', 9]
```

The function `appendleft()` can be used to efficiently append entries to the beginning of an object of class `deque` (from the Python *collections* package). The function `deque()` can be used to convert a `list` into a `deque` (double ended queue).

```
from collections import deque

a = deque(a)
type(a)
```

```
<class 'collections.deque'>
```

```
a.appendleft(0)
a
```

```
deque([0, 20, 7, 'Hi', 10, 'end', 9])
```

Unlike a Python `list`, a data object called a `tuple`, which is delineated using parentheses, contains elements that cannot be changed:

```
b = (1,2,3,4,5)
b[0]
```

```
1
```

```
b[0] = 10 # produces error
```

Multidimensional numerical arrays, including matrices, can be created using functions from *numpy*.

Example 9.24.
Here we define:

$$\boldsymbol{B} = \begin{bmatrix} 1 & 4 & -5 \\ 9 & -7.2 & 4 \end{bmatrix}$$

```
B = np.array([[1, 4, -5], [9, -7.2, 4]])
B
```

```
array([[ 1. ,  4. , -5. ],
       [ 9. , -7.2,  4. ]])
```

We see that `B` is an object of class `numpy.ndarray` (meaning *numpy n*-dimensional array).

```
type(B)
```

```
<class 'numpy.ndarray'>
```

Mathematical matrix operations can be easily applied to `numpy.ndarray` objects. Here I find $\boldsymbol{B} - 5$

```
B - 5
```

```
array([[ -4. ,  -1. , -10. ],
       [  4. , -12.2,  -1. ]])
```

Extensive linear algebra tools are contained in the libraries *numpy* and *scipy*.

■

Unlike a `list`, a **numpy** `array` will allow Boolean indexing and vectorized operations.

9.5.6 Member Functions and Other Attributes

Like C++, Python uses a member function approach to create and call methods for particular classes. As with C++, a member function `foo`, for a class underlying an object `bar`, would be run as: `bar.foo()`. Python classes often have special member functions called *magic methods* or *dunder* (short for double underline) *methods* . These would be called using the syntax: `bar.__foo__()`. Python also uses *instance variables* which are automatically stored as a data attribute of a particular class, but do not require a methods call. An instance variable `foo` for an object `bar` would be called using: `bar.foo`. Available methods and instance variables for an object `bar` can be listed using `dir(bar)`, or ‘`bar.__dir__()`, assuming the object has a `.__dir__()` dunder method.

Example 9.25.
Consider the `numpy.ndarray` object `B` from Example 9.24. There are a large number of object attributes.

```
dir(B)
```

```
['T', '__abs__', '__add__', '__and__', '__array__', '__array_finalize__', '__array_function__', '__array_interface__', '__array_n
```

Note that dunder methods are listed first, and conventional member functions and instance variables are grouped togethor at the end of the `dir()` output.

The dunder method `.__abs__()` prints the elementwise absolute values of the array and `.__len__()` gives the number of rows.

```
B.__abs__()
```

```
array([[1. , 4. , 5. ],
       [9. , 7.2, 4. ]])
```

```
B.__len__()
```

```
2
```

The instance variables for an `array` include `.shape` (which reports the number of rows, columns, etc.) and `.size` which returns the number of `array` elements.

```
B.shape
```

```
(2, 3)
```

```
B.size
```

```
6
```

One can easily easily obtain the mean and standard deviation of any `array` using the `array` member functions `.mean()` and `.std()`.

```
B.mean()
```

```
np.float64(0.9666666666666667)
```

```
B.std()
```

```
np.float64(5.556277730839435)
```

■

9.5.7 Boolean Operations

Python Boolean operators are very similar to those in **R** (Table 9.13). As exceptions, to designate "and" and "or" in Python, one would use the commands `and` and `or`, respectively. Additionally, Python (like C and C++) uses `True` and `False`, instead of `TRUE` and `FALSE`.

Unlike **R** and C, `&&` and `||` are not valid Boolean operators in Python.

Example 9.26.
Consider the following simple examples:

```
a = 2
b = 5

a == b
```

Table 9.13: Python Boolean operators.

Operator	Operation	To ask:	We type:
`>`	$>$	Is x greater than y?	`x > y`
`>=`	$\geq$	Is x greater than or equal to y?	`x >= y`
`<`	$<$	Is x less than y?	`x < y`
`<=`	$\leq$	Is x less than or equal to y	`x <= y`
`==`	$=$	Is x equal to y?	`x == y`
`!=`	$\neq$	Is x not equal to y?	`x != y`
`and`	*and*	Do x and y equal z?	`x == z and y == z`
`&`	*and* (bitwise)	Bitwise comparison of x and y	`x & y`
`|`	*or*	Do x or y equal z?	`x == z or y == z`
`||`	*or* (bitwise)	Bitwise comparison of x or y	`x | y`

```
False
```

```
a != b
```

```
True
```

```
b > a
```

```
True
```

```
a < 4 and b < 4
```

```
False
```

```
a < 4 or b < 4
```

```
True
```

■

Unlike **R**, `&` and `|` are Python *bitwise* Boolean operators for "and" and "or", respectively (Table 9.13). That is, they compare objects by paired bits (see Section 12.3) and, for each bit, return `1` for `True` and `0` for `False`.

Example 9.27.
This example will use the functions `asbio::dec2bin()` and `asbio::bin2dec()` to translate between binary {`0`, `1`} and conventional (decimal) representations of numbers. For additional background see Sections 12.5 and 12.4.

The number 11 can be expressed in binary with four bits: `1011`.

```
asbio::dec2bin(11)
```

```
[1] 1011
```

Whereas the number 14 can be expressed as: `1110`.

```
asbio::dec2bin(14)
```

```
[1] 1110
```

In a bitwise comparison of the number 11 and 14, the first and third bits are equal (both equal 1) while the second and fourth bits are not equal. Thus, the bitwise Boolean result is 1010. This turns out to be the binary version of the number 10.

```
asbio::bin2dec(1010)
```

```
[1] 10
```

This result corresponds to the bitwise comparison of the numbers 11 and 14 in Python, using &.

```
11 & 14
```

```
10
```

■

9.5.8 Mathematical Operations

Basic Python mathematical operators are generally (but not always) identical to **R**. For instance, note that for exponentiation ** is used instead of ^ (Table 9.14). This convention is also used by several other programming languages, including Fortran. Recallthat * can also +-be used by Python in non-mathematical contexts, for instance to load all function names from a package (Section 9.5.4).

Symbolic derivative solutions to functions can be obtained using functions from the library *sympy*. Results from the package functions can be printed in LaTeX for pretty mathematics.

```
py_install("sympy", pip = TRUE) # run in R if sympy hasn't been installed
```

Example 9.28.
Here we solve:

$$\frac{d}{dx} 3e^{-x^2}$$

```
from sympy import *
x = symbols ('x')
fx = 3 * exp(-x ** 2)
print(diff(fx))
```

$$-6xe^{-x^2}$$

Table 9.14: Basic Python mathematical functions and operators.

Operation	Operator/Function	To find	We type
addition	`+`	$2+2$	`2 + 2`
subtraction	`-`	$2-2$	`2 - 2`
multiplication	`*`	2×2	`2 * 2`
division	`/`	$\frac{2}{3}$	`2/3`
modulo	`%`	remainder of $\frac{5}{2}$	`5%2`
integer division	`//`	$\frac{5}{2}$ without remainder	`5//2`
exponentiation	`**`	2^3	`2**3`
$\sqrt{x}$	`sqrt(x)`	$\sqrt{2}$	`numpy.sqrt(2)`
$x!$	`factorial(x)`	$5!$	`numpy.math.factorial(5)`
$\log_e$	`log(x)`	$\log_e(3)$	`numpy.log(3)`
e^x	`exp(x)`	$e^1 = 2.718282\ldots$	`numpy.exp(1)`
$\pi = 3.141593\ldots$	`pi`	π	`numpy.pi`
∞	`inf`	∞	`float('inf')`
$-\infty$	`-inf`	$-\infty$	`float('-inf')`

In Line 2, `x` is defined symbolically using the `sympy.symbols()` function. The variable `x` is used as a term in the expression `fx` in Line 3. The function `fx` is differentiated in Line 4 using the function `sympy.diff()`.

■

Integration in Python can be handled with the function `quad()` in *scipy*.

Example 9.29.
Here we find:

$$\int_0^1 3e^{-x^2}\,dx$$

To perform integration we must install the *scipy.integrate* library using *pip* and bring in the function `quad()`.

```
from scipy.integrate import quad
```

We then define the integrand as a Python function using the function `def()`. That is, `def()` is analogous to `function()` in **R**.

```
def f(x):
  return 3 * np.exp(-x**2)
```

We now run `quad()` on the user function `f` with the defined bounds of integration.

```
quad(f, 0, 1)
```

```
(2.240472398437281, 2.487424042782217e-14)
```

The first number is the value of the definite integral (in this case, the area under the function `f` from 0 to 1). The second is a measure of the absolute error in the numerical approximation.

■

9.5.9 Reading in Data

Data in delimited files, including .csv files, can be read into Python using the *numpy* function `loadtxt()`.

Example 9.30.
Assume that we have a comma separated dataset, named `ffall.csv`, located in the Python working directory, describing the free fall properties of some object over six seconds, with columns for observation number, time (in seconds), altitude (in mm) and uncertainty (in mm). The Python working directory (which need not be the same as the **R** working directory in RStudio) can be identified using the function `getcwd()` from the library *os*.

```
import os
os.getcwd()
```

```
'C:\\Users\\ahoken\\Documents\\GitHub\\Amalgam'
```

We can load `freefall.csv` using:

```
obs, time, height, error = np.loadtxt("ffall.csv",
delimiter = ",", skiprows = 1, unpack = True)
```

The first row was skipped (using `skiprows = 1`) because it contained column names and those were re-assigned when I brought in the data. Note that, unlike **R**, columns in the dataset are automatically attached to the global environment upon loading, and will overwrite objects with the same name.

```
height/1000 # height in meters
```

```
array([0.18 , 0.182, 0.178, 0.165, 0.16 , 0.148, 0.136, 0.12 , 0.099,
       0.083, 0.055, 0.035, 0.005])
```

File readers in *pandas* are less clunky (and more similar to **R**). We can bring in `freefall.csv` using the function `pandas.read_csv()`:

```
py_install("pandas")  # Run if pandas is not installed
```

```
import pandas as pd # run in a Python chunk
ffall = pd.read_csv('ffall.csv')
ffall
```

```
    obs  time  height  error
0     1   0.0     180   3.50
1     2   0.5     182   4.50
2     3   1.0     178   4.00
3     4   1.5     165   5.50
4     5   2.0     160   2.50
5     6   2.5     148   3.00
6     7   3.0     136   2.50
7     8   3.5     120   3.00
8     9   4.0      99   4.00
9    10   4.5      83   2.50
10   11   5.0      55   3.60
11   12   5.5      35   1.75
12   13   6.0       5   0.75
```

The object `ffall` is a *Pandas DataFrame*, which is different in several respects, from an **R** dataframe.

```
type(ffall)
```

```
<class 'pandas.core.frame.DataFrame'>
```

Column arrays in `ffall` can be called using the syntax: `ffall.`, or by using braces, `[]`. For instance:

```
ffall.height
```

```
0     180
1     182
2     178
3     165
4     160
5     148
6     136
7     120
8      99
9      83
10     55
11     35
12      5
Name: height, dtype: int64
```

```
ffall["height"]
```

```
0     180
1     182
2     178
3     165
4     160
5     148
6     136
7     120
8      99
9      83
10     55
11     35
12      5
Name: height, dtype: int64
```

■

9.5.10 Data Analysis in both Python and R

In RStudio, **R** and Python (`reticulate`) sessions are considered separately. When accessing Python from **R**, **R** data types are automatically converted to their equivalent Python types. Conversely, when values are returned from Python to **R** they are converted back to **R** types. It is possible, however, to access each from the others' session.

The *reticulate* command `py` allows one to interact with a Python session directly from the **R** console. Here I convert the *pandas* DataFrame `ffall` into a recognizable **R** dataframe, within **R**.

```
ffallR <- py$ffall
```

Which allows me to examine it with **R** functions.

```
colMeans(ffallR)
```

```
     obs     time   height    error
  7.0000   3.0000 118.9231   3.1615
```

On Lines 1 and 2 in the chunk below, I bring in the Python library *pandas* (from **R**) with the function `reticulate:import()`. The code `pd <- import("pandas", convert = FALSE)` is the Python equivalent of: `import pandas as pd`.

```
pd <- import("pandas", convert = FALSE)
```

As expected, the column names constitute the names attribute of the dataframe `ffallR`.

```
names(ffallR)
```

```
[1] "obs"    "time"   "height" "error"
```

The `ffall` `dataframe`, however, has different characteristics when it is loaded as a *pandas* `DataFrame`. Note that in the code below the *pandas* function `read_csv()` is accessed using `pd$read_csv()` instead of `pd.read_csv()` because an **R** chunk is being used.

```
ffallP <- pd$read_csv("ffall.csv")
```

The names attribute of the *pandas* `DataFrame` `ffallP`, as perceived by **R**, contains over 200 entities due the presence of `DataFrame` attributes (including member functions and instance variables) see Section 9.5.6. Many of these are provided by the built-in Python module *statistics*. Here are the first 20.

```
head(names(ffallP), 20)
```

```
 [1] "abs"        "add"        "add_prefix" "add_suffix" "agg"
 [6] "aggregate"  "align"      "all"        "any"        "apply"
[11] "applymap"   "asfreq"     "asof"       "assign"     "astype"
[16] "at"         "at_time"    "attrs"      "axes"       "backfill"
```

I can call these attributes using the `$` operator, in the style of RC and R6 methods (Section 8.7.2). These procedures clearly demonstrate the straightforwardness of **R**/Python syntheses under *reticulate*.

```
ffallP$mean()
```

```
obs         7.000000
time        3.000000
height    118.923077
error       3.161538
dtype: float64
```

```
ffallP$var()
```

```
obs         15.166667
time         3.791667
height    3495.243590
error        1.512147
dtype: float64
```

```
ffallP$kurt()
```

```
obs      -1.200000
time     -1.200000
```

```
height   -0.692166
error     0.445443
dtype: float64
```

Note that the final result is clearly being provided by Python because kurtosis functions are not native to the **R** *stats* package.

For further analysis in **R** these attributes will need to be explicitly converted to **R** objects using the function `py_to_r()`.

```
trans <- ffallP$transpose() # transpose matrix
transR <- py_to_r(trans)

apply(transR, 1, mean)
```

```
     obs     time   height    error
  7.0000   3.0000 118.9231   3.1615
```

9.5.11 Python versus R

R generally allows much greater flexibility than Python for explicit statistical analyses and graphical summaries. For example, the Python statistics library *Pymer4* actually uses generalized linear mixed effect model (see Aho (2014)) functions from the **R** package *lme4* to complete computations. Additionally, Python tends to be *less* efficient than **R** for pseudo-random number generation[32], since it requires looping to generate multiple pseudo-random outcomes (see Van Rossum and Drake (2009)).

Example 9.31.
Here I generate 10^8 pseudo-random outcomes from a continuous uniform distribution (processor details footnoted in Example 9.6).
R:

```
system.time(ranR <- runif(1e8))
```

```
   user  system elapsed
   2.36    0.10    2.43
```

Python:

[32] A *pseudo-random number generator* (PRNG) is a deterministic algorithm for generating numbers whose properties approximate those of random numbers (Wikipedia, 2024h). A PRNG sequence is dependent on an initial *seed* value provided to the generator. By default, **R** PRNG seeds are generated from the current session time. Details are provided in `?RNG`. Numbers from distinct probability distributions can be generated from an underlying pseudo-random continuous uniform sequence by obtaining the corresponding inverse CDF outcomes for the same lower-tailed probability. **R** can employ a large number approaches for generating pseudo-random numbers, including, by default, the "Mersenne-Twister" (Matsumoto and Nishimura, 1998).

```
1 import time
2 import random
3 ranP = []
4
5 start_time = time.time()
6 for i in range(0,9999999):
7   n = random.random()
8   ranP.append(n)
9 time.time() - start_time
```

```
2.976600408554077
```

The operation takes much longer for Python than **R**.

The Python code above requires some explanation. On Lines 1 and 2, the Python modules *time* and *random* are loaded from the Python standard library, and on Line 3 an empty list `ranP` is created that will be filled as the loop commences. On Line 5, the start time for the operation is recorded using the function `time()` from the module *time*. On Line 6 a sequence of length 10^8 is defined as a reference for the index variable `i` as the for loop commences. On Lines 7 and 8 a random number is generated using the function `random()` from the module *random* and this number is appended to `ranP`. Note that Lines 7 and 8 are indented to indicate that they reside in the loop. Finally, on Line 9 the start time is subtracted from the end time to get the system time for the operation.

■

On the other hand, the system time efficiency of Python may be better than **R** for many applications, including the management of large datasets (Morandat et al., 2012).

Example 9.32.
Here I add the randomly generated dataset to itself in **R**:

```
system.time(ranR + ranR)
```

```
   user  system elapsed
   0.17    0.00    0.19
```

and Python:

```
start_time = time.time()
diff = ranP + ranP
time.time() - start_time
```

```
0.08802533149719238
```

For this operation, Python is faster.

■

Of course, IDEs like RStudio allow, through the package *reticulate*, simultaneous use of both **R** and Python systems, allowing one to draw on the strengths of each language.

Exercises

1. The Fortran script below calculates the circumference of the earth (in km) for a given latitude (measured in radians). For additional information, see Question 6 from the Exercises in Ch 2. Explain what is happening in each line of code below.

```
subroutine circumf(x, n)
double precision x(n)
integer n
x = cos(x)*40075.017
end
```

2. Create a file `circumf.f90` containing the code and save it to an appropriate directory. Take a screen shot of the directory.

3. Compile `circumf.f90` to create `circumf.dll`. In Windows this will require the shell script shown below. You will have to supply your own `Root part of address`, and `Approriate directory` will be the directory containing `circumf.f90`. Take a screen-shot to show you have created `circumf.dll`. Running the shell code may require that you use the shell as an Administrator.

```
cd Root part of address\bin\x64
R CMD SHLIB Appropriate directory/circumf.f90
```

4. Here is a wrapper[33] for `circumf.dll`. Again, you will have to supply `Approriate directory`. Explain what is happening on Lines 2, 4, and 5. And, finally, run: `cearthf(0:90)`.

```
cearthf <- function(latdeg){
  x <- latdeg * pi/180
  n <- length(x)
  dyn.load("Appropriate directory/circumf.dll")
  out <- .Fortran("circumf", x = as.double(x), n = as.integer(n))
  out
}
```

5. Here is a C script that is identical in functionality to the Fortran script in Q. 1. The header `#include <math.h>` (see Section 9.3.2) allows access to C mathematical functions, including `cos()`. Describe what is happening on Lines 7-10.

[33] Unix-alikes should replace `circumf.dll` with `circumf.o`.

```
1
2  #include <math.h>
3
4
5  void circumc(int *nin, double *x)
6  {
7    int n = nin[0];
8    int i;
9    for (i=0; i<n; i++)
10     x[i] = (cos(x[i]) * 40075.017);
11 }
```

6. Repeat Qs, 2 and 3 for the C subroutine `circumc`.

7. Here is an **R** wrapper for `circumc.dll`. Explain what is happening on Lines 4-6 and run: `cearthc(0:90)`.

```
1  cearthc <- function(latdeg){
2    x <- latdeg * pi/180
3    n <- length(x)
4    dyn.load("Appropriate directory/circumc.dll",
5             nin = n, x)
6    out <- .C("circumc", n = as.integer(n), x = as.double(x))
7    out
8  }
```

8. Complete Problem 5 (a-f) from the Exercises in Ch 2 using C++ via *Rcpp*. The code below completes part (a) and (b). Note the use of decimals to enforce double precision.

```
library(Rcpp)

src <-
'
#include <Rcpp.h>
#include <cmath>

using namespace Rcpp;
// [[Rcpp::export]]

List Q8(){
  double a = 1 + 3./10. + 2;
  double b = (1. + 3.)/10. + 2;
  return List::create(Named("a") = a,
                      Named("b") = b);
}'
```

```
sourceCpp(code = src)
Q8()
```

```
$a
[1] 3.3

$b
[1] 2.4
```

9. Using *Rcpp*, Create a C++ function for calculating the Satterthwaite degrees of freedom (see Q 2 from the Exercises in Ch 8). Test using the data: `x <- c(1,2,3,2,4,5)` and `y <- c(2,3,7,8,9,10,11)`.

10. Make a Python list with elements `"pear"`, `"banana"`, and `"cherry"`.
    (a) Extract the second item in the list.
    (b) Replace the first item in the list with `"melon"`.
    (c) Append the number 3 to the list. .

11. Make a Python tuple with elements `"pear"`, `"banana"`, and `"cherry"`.
    (a) Extract the second item in the tuple.
    (b) Replace the first item in the tuple with `"melon"`. Was there an issue?
    (c) Append the number 3 to the tuple. Was there an issue?

12. Using `def()`, write a Python function that will square any value `x`, and adds a constant `c` to the squared value of `x`.

13. Call Python from **R** to complete Problem 5 (a-h) from the Exercises in Ch 2. Document your work in **R** Markdown.

Chapter 10

Building R Packages

> *"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."*

- **Tony Hoare**, *Pioneering British computer scientist*

10.1 Introduction

One of strengths of **R** is its capacity to format and share user-designed software as packages. Clearly it is possible to apply **R** for one's entire scientific career without creating an **R** package. However, development of a package, even if it is not distributed to a formal repository, ensures that your software is trustworthy and portable. Importantly, this chapter only provides a overview of basic topics in package development. The most thorough and up-to-date guide to package creation is the document Writing R Extensions, which is maintained by the the **R** development core team.

10.2 Package Components

An **R** package is a directory of files, generally with nested subdirectories. Specifically,

- `DESCRIPTION` and `NAMESPACE` files define fundamental characteristics of the package, e.g., the author(s), the maintainer, the package version, the dependency on other packages, etc.
- Subdirectories, and their nested files, contain the package contents. The following subdirectories are possible, although not all need to exist within a package.
- The `R` subdirectory contains the package **R** code, stored as .r files, and will almost always exist.
- The `data` subdirectory contains package datasets, usually stored as .rda files, which can be created using `save()`.

- The `man` subdirectory contains the package documentation, stored as .rd files, for functions (in the `R` directory) and data (in `data`), and almost always exists.
- The (optional) `src` subdirectory contains raw source code requiring compilation (C, C++, Fortran). When building a package **R** will call `R CMD SHLIB` (see Section 9.1.6) to create appropriate binary shared library files.

- Other potential subdirectories include: `demo`, `exec`, `inst`, `po`, `tests`, `tools`, and `vignettes`.

Fig 10.1 shows the contents of the *streamDAG* package. These directories, and their files, are contained within a parent directory called `streamDAG`.

Figure 10.1: Subdirectory level components of the *streamDAG* package.

Example 10.1.

Creation of package components can be facilitated with the function `package.skeleton()`. From the `package.skeleton()` documentation Examples (see `?package.skeleton`), assume that we want to build a package that contains two silly functions: (`f` and `g`) and two silly datasets: (`d` and `e`).

```
f <- function(x, y) x + y
g <- function(x, y) x - y
d <- data.frame(a = 1, b = 2)
e <- rnorm(1000)
```

We specify these as the `list` argument in `package.skeleton()` and give the package the name *mypkg*.

```
package.skeleton(list = c("f","g","d","e"), name = "mypkg")
```

Running this code will cause a package skeleton for *mypkg* to be sent to the working directory. Note that the skeleton contains the subdirectories: `data`, `r`, and `man` (Fig 10.2). The datasets `d` and `e` were converted to .rda files by `package.skeleton()` and were placed in the `data` subdirectory. The functions `f` and `g` were converted to .r files and placed in the `r` subdirectory. Documentation skeletons for both functions and both datasets, as .rd files, were placed in the `man` subdirectory. Package `DESCRIPTION`, `NAMESPACE` files, and a throw-away (`Read-and-delete-me`) file were also created (Fig 10.2).

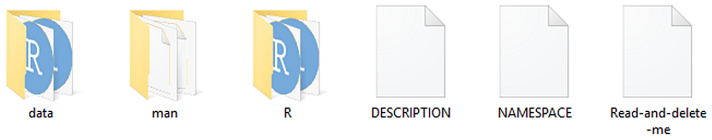


Figure 10.2: Subdirectory level components of the toy *mypkg* package.

■

10.3 Datasets (the `data` Subdirectory)

Datasets in **R** are stored in the `data` subdirectory. Three data formats are possible:

- Raw .r code
- Tabular data (e.g., .txt, .csv files)
- Data "images" created using the function `save()`, e.g., .rda or .Rdata files. This approach is generally recommended, particularly for large datasets. Here we create a simple .rda dataset, and send it to the working directory.

```
x <- rnorm(5)
save(x, file = "x.rda")
```

Data from packages will either be accessible via lazy loading (which allows increased accessibility) or with the `data()` function. Under the former approach, package data objects will not be loaded upon loading of their package environment, however promises are created, requiring the object to be loaded when its name is entered in a session. Lazy loading always occurs for package **R** code but is optional for package data. Lazy loading of data can be specified in a `'LazyData'` field from a package's `DESCRIPTION` file (see below). Examples of lazy loaded data include objects from the package *datasets*. Note that these do not require `data()` for loading:

```
datasets::BOD # data describing Biochemical Oxygen Demand
```

```
  Time demand
1    1    8.3
2    2   10.3
3    3   19.0
4    4   16.0
5    5   15.6
6    7   19.8
```

Under the latter, more common approach, `data(*foo*)` must be called to allow availability of the dataset *foo*.

```
library(asbio)
data(bighorn.sel) # bighorn sheep resource use and availability
bighorn.sel
```

```
          resources avail  y1  n1
1          Riparian  0.06   0 445
2           Conifer  0.13   6 445
3       Mt. Shrub 1  0.16   9 445
4             Aspen  0.15  18 445
5      Rock outcrop  0.06  14 445
6  Sage/Bitterbrush  0.17  63 445
7  Windblown ridges  0.12  46 445
8        Mt shrub 2  0.04  62 445
9  Prescribed burns  0.09 178 445
10         Clearcut  0.02  49 445
```

10.4 R Code (the `r` Subdirectory)

Code for functions is generally stored in the `r` directory, as .r files. IDEs like RStudio, which contain options for the generation of .r scripts, e.g., **File** > New File > R script, can greatly aid in this process. Single .r files can contain multiple functions, although a one function per file approach may be easier to manage.

10.5 Documentation (the `man` Subdirectory)

As functions become complex, it may become difficult to keep track of the meaning of function arguments, and the characteristics of function output, using a simple notes-to-self approach, e.g., `#`. **R** documentation (.rd) files provide a framework for documenting, **R** functions, methods, and datasets. The `prompt()` family of functions can greatly facilitate the creation of .rd files. In Example 10.1, the function `package.skelton()` used the functions `prompt()` and `promptData` to build documentation skeletons for functions and datasets, respectively. For instance, the code below was applied to create documentation for the function `f()`.

```
f <- function(x, y) x + y
prompt(f, filename = "f")
```

```
Created file named 'f'.
Edit the file and move it to the appropriate directory.
```

This code causes the file `f.rd` to be generated, and sent to the working directory for further editing (Fig 10.3).

```
\name{f}
\alias{f}
\title{
}
\description{
}
\usage{
f(x, y)
}
\arguments{
  \item{x}{
}
  \item{y}{
}
}
\details{
}
\value{
}
\references{
}
\author{
}
\note{
}
\seealso{
}
\examples{
## The function is currently defined as
function (x, y)
x + y
}
```

Figure 10.3: Documentation file skeleton for the toy function `f()`

Some guidance for completing .rd files is provided by notes in the skeleton generated by `prompt()`. I have removed these notes in Fig 10.3 to save space. As before, the authoritative resource for documentation building is Writing R Extensions.

Package documentation files can be placed into a `man` directory and compiled into a single documentation entity as the package is compiled[1], or compiled singly for **R** objects that a user deems worthy of documentation. The latter approach is facilitated with the **Preview** widget in RStudio, which is available upon opening an .rd file. Running **Preview** on the file `f.rd` resulting in the .html preview shown in Fig 10.4.

[1]Following package compilation, installation, and loading, this allows access to documentation via `help(documented topic)` or `?documented topic` (Section 2.4)

Figure 10.4: Preview of the .html generated from the code shown Fig 10.3.

An .rd file can be converted to legible documentation in .html, .pdf or other formats by depositing the file in the **R** directory containing `R CMD` routines (e.g., bin/x64), and running the appropriate `R CMD` algorithms from the command line. In Windows this requires first navigating to the directory containing the `R CMD` routines using the Windows shell command line editor (see Ch 9). Important `R CMD` documentation rendering algorithms include:

- `R CMD Rd2pdf` *foo.rd*', can be used to compile the documentation file *foo.rd* into a .pdf document.
- `R CMD Rd2txt` *foo.rd*', can be used to compile the documentation file *foo.rd* into a pretty text format.
- `R CMD Rdconv` *foo.rd*', can be used to compile the documentation file *foo.rd* into a variety formats including plain text, HTML, or LaTeX.

10.6 The DESCRIPTION File

The `DESCRIPTION` file contains basic information about a package. The `DESCRIPTION` file skeleton for the *mypkg* package, created by `package.skeleton()` in Example 10.1, is shown in Fig 10.5.

```
Package: mypkg
Type: Package
Title: What the package does (short line)
Version: 1.0
Date: 2023-10-26
Author: Who wrote it
Maintainer: Who to complain to <yourfault@somewhere.net>
Description: More about what it does (maybe more than one line)
License: What license is it under?
```

Figure 10.5: `DESCRIPTION` file of the toy *mypkg* package.

The `DESCRIPTION` file will have a Debian control file format (see `?read.dcf`. Specifically, fields in `DESCRIPTION` must start with the field name, comprised of ASCII (Ch 12) printable characters, followed by a colon. The value for the field is given after the colon and an additional space (Fig 10.5). If allowed, field values longer than one line must use a space or a tab to start a new line. Specification of `'Package'`, `'Version'`, `'License'`, `'Description'`, `'Title'`, `'Author'`, and `'Maintainer'` fields, shown in Fig 10.5, are mandatory.

- The `'Package'` field gives the name of the package.
- The `'Version'` field gives a user-specified package version. It should be a sequence of at least two non-negative integers separated by single usages `'.'` and/or `'-'` characters.
- The `'Title'` field should provide a descriptive title for the package. It should use title case (capitals for principal words), and not have any continuation lines.
- The `'Author'` field describes who wrote the package. Note that if your package contains wrappers of the work of others, which are included in the `src` directory, then you are not the sole author.
- The `'Maintainer'` field provides a single name followed by a valid email address in angle brackets (Fig 10.5).
- The `'Description'` field should provide a comprehensive description of what the package does. Several (complete) sentences, complete, although these should limited to one paragraph. The field value should not to start with the package name, or `'This package...'`.
- The `'License'` field provides standard open source license information for the package. Failure to specify license information may prevent others from legally using, or distributing your package. Standard licenses available from (https://www.R-project.org/Licenses/) include GPL-2, GPL-3, LGPL-2, LGPL-2.1, LGPL-3, AGPL-3, Artistic-2.0, BSD_2_clause, and BSD_3_clause MIT. See Writing R Extensions for more information.
- Other optional fields include: `'Copyright'`, `'Date'`, `'Depends'`, `'Imports'`, `'Suggests'`, `'Enhances'`, `'LinkingTo'`, `'Additional_repositories'`, `'SystemRequirements'`, `'URL'`, `'BugReports'`, `'Collate'`, `'LazyData'`, `'KeepSource'`, `'ByteCompile'`, `'UseLTO'`, `'StagedInstall'`, `'Biarch'`, `'BuildVignettes'`, `'VignetteBuilder'`, `'NeedsCompilation'`, `'OS_type'`, and `'Type'`. See Writing R Extensions for more information on these fields.

10.7 The NAMESPACE File

The **R** namespace management system allows package authors to specify which variables in the package can be exported to package users, and which variables should be imported from other packages. The mandatory `NAMESPACE` file for the toy *mypkg* package is extremely simple (Fig 10.6). It indicates that all four objects contained in the package, and their associated names, can be exported. If one wishes to export all objects and names for a large package, it is simpler to specify: `exportPattern(.)`.

```
export("f", "g", "d", "e")
```

Figure 10.6: `NAMESPACE` file of the toy *mypkg* package.

Import of exported variables from other packages requires specification of `import` and `importFrom`. The `import` directive imports all exported variables from specified package(s). Thus, `import(foo)` imports all exported variables in the package *foo*. If a package requires some of the exported variables from a package, then `importFrom` can be used. The `NAMESPACE` directive `importFrom(foo, f, g)` indicates that `f` and `g` from package *foo* should be imported.

To ensure that S3 methods for package classes are available, one must register the methods in the `NAMESPACE` file. For instance, if a package has a function `print.foo()` that serves as a print method for class `foo`, then one should include `S3method(print, foo)` as a line in `NAMESPACE`.

Package Compilation As with compilation of C and Fortran files (Ch 9), and the conversion of individual .rd files, the building and installation of a user-designed package requires depositing the package contents in the **R** directory containing the `R CMD` routines.[Or providing a navigation address to the package for `R CMD`.] [Probably the only `R CMD` routine isn't clearly tied to the development of **R** packages is `Rcmd BATCH`, which is used for running **R** scripts from the command line.] As before, one must run `R CMD` routines from the command line, requiring (in Windows) that a user navigate to the directory containing the `R CMD` routines at the Windows shell command line. This is unnecessary in Unix-like operating system (including MacOS), as these algorithms can be called directly from the computer's command line. `R CMD` routines for package building include:

- `R CMD build` *foo*, which would build the package *foo*.
- `R CMD check` *foo.tar.gz*, which would check the tarballed package *foo.tar.gz*, created by `R CMD build`.
- `R CMD INSTALL` *foo.tar.gz* can be used to install the package *foo*.

Example 10.2.
Continuing from Example 10.1, I complete the following steps for package building/compression, checking, and installation.

- Here I Build a tarballed version of the *mypkg* package using: `R CMD build mypkg`.

```
C:\Program Files\R\R-4.3.1\bin\x64>R CMD build mypkg
* checking for file 'mypkg/DESCRIPTION' ... OK
* preparing 'mypkg':
* checking DESCRIPTION meta-information ... OK
* checking for LF line-endings in source and make files and shell scripts
* checking for empty or unneeded directories
* building 'mypkg_1.0.tar.gz'
```

- Here I check the tarballed version of the package using: `R CMD check mypkg_0.1.tar.gz`.

```
C:\Program Files\R\R-4.3.1\bin\x64>R CMD check mypkg_1.0.tar.gz
* using log directory 'C:/Program Files/R/R-4.3.1/bin/x64/mypkg.Rcheck'
* using R version 4.3.1 (2023-06-16 ucrt)
* using platform: x86_64-w64-mingw32 (64-bit)
* R was compiled by
    gcc.exe (GCC) 12.2.0
    GNU Fortran (GCC) 12.2.0
* running under: Windows 10 x64 (build 17134)
* using session charset: ISO8859-1
* checking for file 'mypkg/DESCRIPTION' ... OK
* checking extension type ... Package
* this is package 'mypkg' version '1.0'
* checking package namespace information ... OK
* checking package dependencies ... OK
* checking if this is a source package ... OK
* checking if there is a namespace ... OK
* checking for executable files ... OK
* checking for hidden files and directories ... OK
* checking for portable file names ... OK
* checking whether package 'mypkg' can be installed ... OK
```

Note that the checks from `R CMD check` can be extensive (the output above is just an excerpt). Checks are even more taxing if one uses the option `--as-cran` which performs assessments one must pass for submission to CRAN.

- Finally, I Install the *mypkg* package into my workstation using: `R CMD INSTALL mypkg_0.1.tar.gz`.

```
C:\Program Files\R\R-4.3.1\bin\x64>R CMD INSTALL mypkg_1.0.tar.gz
* installing to library 'C:/Users/ahoken/AppData/Local/R/win-library/4.3'
* installing *source* package 'mypkg' ...
** using staged installation
** R
** byte-compile and prepare package for lazy loading
** help
*** installing help indices
** building package indices
** testing if installed package can be loaded from temporary location
** testing if installed package can be loaded from final location
** testing if installed package keeps a record of temporary installation path
* DONE (mypkg)
```

■

Exercises

1. Create an .rd documentation file for the function for McIntosh's index of site biodiversity from Exercise 2 in 8. Make a .pdf or .html from the .rd file using the appropriate `R CMD` routines.

2. Create an **R** package consisting of at least one function. Specifically,

(a) Create a skeleton of the package using `package.skeleton()`.
(b) Finish the .rd file(s) in `man`.
(c) Complete the `DESCRIPTION` file.
(d) Complete the `NAMESPACE` file.
(e) Build the package using `R CMD build`.
(f) Check the package using `R CMD check`. Modify the package (if necessary) until no more `ERRORS` or `WARNINGS` occur.

Chapter 11

Interactive and Web Applications

"A user interface is like a joke. If you have to explain it, it's not that good."

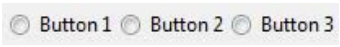
- **Martin LeBlanc**, *Iconfinder cofounder*

11.1 Introduction to GUIs

It is possible build *GUIs* (Graphical User Interfaces) that allow users to interact with **R** using graphical icons and visual indicators. These can be created using a number of methods and language frameworks. The interactive control components of a GUI are called *widgets*. The following are some commonly used widgets:

- *button*: typically a GUI controller for binary (e.g., on/off or run/don't run) operations:

- *radio button*: A button allowing selection from a group of mutually-exclusive options that are linked to specific operations:



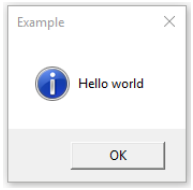
- *check button* or *check box*: A controller that allows selection of one or several mutually-exclusive binary options via a check mark tick, ☑ , or a cross, ☒ . Like other buttons, the widget can be linked to a second variable, allowing flexible rendering of a secondary set of widgets:

- *spin box*: Provides a "spin-able" set of mutually exclusive (often numeric) options that can be selected and linked to operations:

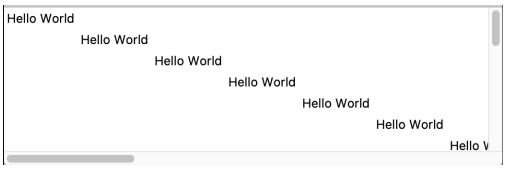
- *combo box*: A text field with a popdown selection list:

- *slider*: A sliding left-to-right (horizontal) or up-down (vertical) handle controller that defines a numeric value (or range of values, if two slider handles are present) for a linked variable that changes uniformly over some range:

- *message box*: A message window that typically prompts a user response and a corresponding linked operation:



- *scroll bar*: A modifiable viewport for a scrollable object (e.g., text that can be examined line by line):



- *pulldown menus*: Interactive menus with pulldown tabs and *menu buttons* that specify operations, potentially including links to other GUIs.

R GUIs are are a mixed bag. On the plus side, **R** GUIs: 1) increase user-friendliness by allowing point and click operations, 2) allow rapid visual assessment of alteration to function arguments via widgets, 3) are often very amenable to graphics manipulations, and 4) are often very useful for data exploration or heuristic demonstrations. On the other hand, **R** GUIs: 1) often result in a loss of flexibility in controlling functions, 2) may contain a visually confusing mish-mash of widgets, and 3) constitute mysterious black boxes, which is contrary to the "mission statement" of **R** (Chambers, 2008). Further, command line (non-GUI) code entry allows an exact record of characteristics given to objects, and specifications provided to functions. This allows straightforward tracking, dissemination, and repeatability of computational analyses.

Despite potential drawbacks, I will explore four methods for creating GUIs that are underlain by **R** or with explicit **R** interactivity. These are: 1) TclTk GUIs generated with the *tcltk* package, 3) Plotly interactive plots via the *plotly* package, 3) applications created through the *shiny* package, and 4) Qt GUI executables driven by C++ bindings to **R** via the *Rinside* package. There are a large number of other packages/approaches for building GUIs with **R** that generally use Java-alike bindings for rendering in HTML. Some of these methods are briefly considered in

Section 11.3.

11.2 tcltk

The **R** distribution package *tcltk* (pronounced: *tickle tee kay*) allows the building of GUIs by providing a binding wrapper for Tcl/Tk, which denotes the Tcl language via its Tk toolkit (Ousterhout, 1991)[1]. In Python, bindings for Tcl/Tk are provided by the Python library *tkinter* which is included in the Python standard library of packages. Unfortunately, better support exists for *tkinter* than *tcltk*[2].

Lack of guidance for the *tcltk* package is likely due to the absence of a large user group. Assistance for the creation of *tcltk* GUIs can be found in several older articles from **R** News (which has since been replaced by the **R** Journal) (Dalgaard, 2001, 2002; Fox, 2007), the book "Programming GUIs in **R**" (Lawrence and Verzani, 2018), and in the GUI code for a number of newer **R** packages, including *Rcmdr* (Fox, 2005; Fox et al., 2023) and *asbio* (Aho, 2023). Despite these resources, however, it is expected that users refer to the Tcl/Tk package manual for argument lists and descriptions of *tcltk* functions. Arguments in *tcltk* functions (generally) have the same names and functionality as their Tcl/tk equivalents, although some experimentation may be necessary.

Tcl/Tk itself is cross platform, and uses facilities particular to the underlying OS. These are *Xlib* (X11) (a windowing system, written in C, for bitmap displays) for Unix/Linux, *Cocoa* for Mac OS, and the *Graphics Device Interface* (GDI) for Windows.

So-called *Themed Tk* (Ttk) GUIs often have advantages over older Tk GUIs, including anti-aliased font rendering, and have been a part of the Tk distribution since Tcl version 8.5. Naming conventions in *tcltk* indicate whether functions are binding for Tk or Ttk operations. The former function names start with `tk`, while the latter start with `ttk`.

Notably, *tcltk* GUIs that use or manipulate **R** graphics devices, particularly those with slider widgets, may work poorly with the native RStudio graphics device: `RStudioGD`. Thus, to run these sorts of GUIs in RStudio, one should open a non-`RStudioGD` device using:

```
dev.new(noRStudioGD = TRUE)
```

The binding mechanism of *tcltk* is apparent by examining underlying code from some of its core functions. The *tcltk* function `tcl()` provides a generic interface for any Tk or Tcl command. For example, the code:

```
tcl("label", tt, text = "Hello, world!", bg = "red")
```

[1] Tcl, an acronym for "tool command language," is an interpreted programming language that is often embedded into C applications. Tk is a Tcl package for GUI building. Tk, when implemented in Tcl, is termed Tcl/Tk.

[2] Many non-Tcl/Tk approaches exist for GUI-building in Python, although they are not included in the Python standard library.

is equivalent to the *tcltk* call:

```
tklabel(tt, text = "Hello, world!", bg = "red")
```

where `tt` is the top-level GUI object. Both of these scripts will call an API to generate the Tcl/tk code:

```
label tp -text "Hello, world!"  -bg red
```

where `tp` is the path name of the Tcl/Tk label object.

Many *tcltk* commands are simply calls to `tcl()`.

Example 11.1.
Here we see that `tcl()` calls `.Tcl.objv()` whose arguments, in turn, are formatted by `.Tcl.args.objv()`.

```
require(tcltk)
tcl
```

```
function (...)
.Tcl.objv(.Tcl.args.objv(...))
<bytecode: 0x0000012d8dba97e0>
<environment: namespace:tcltk>
```

The function `.Tcl.objv()` calls an underlying C algorithm, `.C_dotTclObjv()`, using `.External()` that binds `tcl()` to Tcl/Tk.

```
.Tcl.objv
```

```
function (objv)
structure(.External(.C_dotTclObjv, objv), class = "tclObj")
<bytecode: 0x0000012d8dba8eb0>
<environment: namespace:tcltk>
```

The *tcltk* C executable (Section 9.1.4), `tcltk.dll` constitutes the **R** API for Tcl/tk. It is housed in the *tlctk* package `libs/x64` directory (Ch 10).

```
tcltk:::.C_dotTclObjv$dll
```

```
DLL name: tcltk
Filename: C:/Program
       Files/R/R-4.5.1/library/tcltk/libs/x64/tcltk.dll
Dynamic lookup: FALSE
```

■

Example 11.2. As an initial foray into *tcltk* GUI-building we will create a button interface whose only purpose is to provide a message, and a means for destroying itself.

```
1 tt <- tktoplevel()
2 hello <- tkmessage(tt, text = "Hello world!")
3 spacer = tklabel(tt, padx = 20)
4 DM.but <- tkbutton(tt, text = "Exit", foreground = "red",
5                    background = "lightgreen", padx = 10,
6                    command = function() tkdestroy(tt))
7 tkpack(hello, spacer, DM.but)
```

- On Line 1, I load the *tcltk* package.
- On Line 2, I use use `tktoplevel()` to hierarchically define the "top level" widget as the object `tt`.
- On Lines 3-4, I create a text message object, `hello`, and a spacer object, `spacer`. That latter is used to make room between the message and a button created in the next two lines of code.
- On Lines 5-6 the button widget object `DM.but` is created, using the function `tkbutton()`. The first argument is name of parent widget, `tt`. The `text` argument provides a text label for the button. The arguments `foreground`, `background`, and `padx` are used to define the foreground color (the color of the button text label), the background color of the button, and to make the button wider, respectively. The `command` argument defines the function that the button initiates. In this case, the function `tkdestroy()`, which destroys the GUI.
- On Line 8, `tkpack()` is used to place the button on the parent widget.

The GUI itself is shown in Fig 11.1.

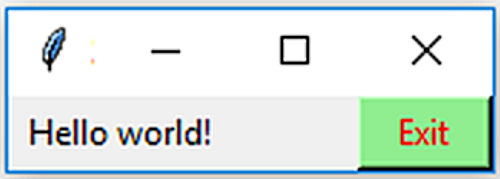


Figure 11.1: A simple *tcltk* GUI.

■

11.2.1 Assigning and Manipulating Widget Values

It is often useful to "remember" object characteristics and assignment values over the course of a GUI's usage. For example, it may be necessary to count the number of times a button is pressed, or display a particular message based on a previous response. Because GUI actions will be carried out by local variables in functions, modifications to those variables will be lost when the function exits. While not usually good practice, one can use the super assignment operator `<<-` inside a function to create global variables. These will retain their values after the function exits.

The `tclVar()` function can be used to create an empty or specific values, which can then be used in call to other functions in the *tcltk* package. For example, to specify an empty `tclVar()` value, one could use:

```
myvar <- tclVar('')
```

To access `myvar` information in widgets with **R** one could then use:

```
rmyvar <- tclvalue(myvar)
```

Conjoined use of the super-assignment operator with `tclVar()` and `tcl()` is often very important when altering object parameters within *tcltk* functions.

11.2.2 User functions and tcltk GUIs

Callbacks are functions that are linked to GUI events. In *tclk* these functions can be user-defined, although they should not have arguments. Callbacks, user defined or otherwise, are generally executed using the `command` argument in a widget function. Recall, for example, use of `tkbutton(tt, command = functiton() tkdestroy)` in Example 11.2. In general, a callback function `foo()` is called using `command = foo()` or `command = substitute(foo())`. Use of `substitute(foo())` allows substitution of variable values in `foo()`. Calling a function `bar()` from within the callback `foo()` may require coding similar to `foo <- function(){substitute(bar())}`. See, for instance, `asbio::anm.ci.tck()`.

Example 11.3.
Consider the following silly example for finding the sum of two numbers.

```
tt <- tktoplevel()
tw1 <- tclVar(''); tw2 <- tclVar('')
tke1 <- tkentry(tt, width = 6, textvariable = tw1, justify = "center")
tke2 <- tkentry(tt, width = 6, textvariable = tw2, justify = "center")

sumf <- function(){
  temp <- as.numeric(tclvalue(tw1)) + as.numeric(tclvalue(tw2))
  tkconfigure(ans, text = paste(temp))
}
```

```
10
11 ans <- tklabel(tt, text = '', background="white", relief = "sunken", padx = 20)
12 tkgrid(tke1, tklabel(tt, text = '+'), tke2, tklabel(tt, text = '='), ans)
13 tkgrid(tklabel(tt, text = ''), columnspan = 5)
14 tkgrid(tkbutton(tt, text = 'Get Sum!', foreground = "red",
15                  background = "lightgreen", command = sumf),
16       columnspan = 5, sticky = "e")
```

- On Line 1, I use `tktoplevel()` to define the "top level" widget.
- On Line 2, I specify empty initial values for the variables `tw1` and `tw2` using `tclVar()`. These values will be editable by users via `tkentry()` widgets.
- On Lines 3-4, I use the function `tkentry()` to provide widgets for users to enter numbers to be summed.
- On Lines 6-9, I create the function `sumf`. The function `tclvalue()`, used to compute the summation object `temp`, allows Tcl variables from `tclVar()` to be evaluated in **R**. These variables, however, will have class `character`, and will require `as.numeric()`, as shown, for mathematical evaluation. One Line 8 (the final line of code in `sumf`, `tkconfigure()` is used to potentially change `ans`, a `tkentry()` object defined on Line 11.
- On Line 11, the `tkentry()` object `ans` is created and an initial empty value is assigned.
- On Lines 12-16, widgets are placed in the GUI using `tkgrid()`. The use of grid geometry approaches including `tkgrid()` is elaborated next. The `tkbutton` widget in the final (bottom) grid of the GUI calls the `sumf` using either `command = sumf`, as shown, or `command = substitute(sumf())`.

The resulting GUI is shown in Fig 11.2.

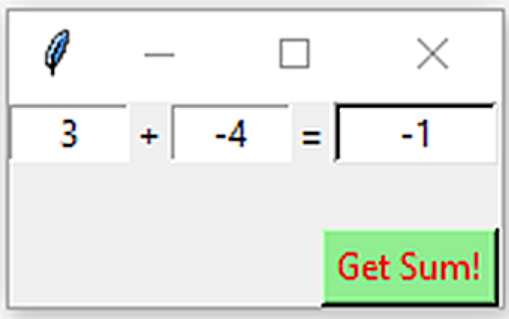


Figure 11.2: A simple *tcltk* GUI, demonstrating the use of `tclVar()` and `tclvalue()`.

■

11.2.3 GUI Geometry

An important consideration in GUI development is *geometry management*, e.g., the dimensions of the GUI and the organization of widgets. By default, Tcl/Tk GUI windows are autosized to hold widgets as they are added. Widgets may be reorganized as the sizes of windows are altered. If a Window becomes too small to contain widgets, the last widget added will be the first removed.

The initial size of GUIs can be specified using the function `tkcanvas()`. The result of the code below is shown in Fig 11.3

```
tt <- tktoplevel()
tktitle(tt) = "Wide GUI"
dim <- tkcanvas(tt, height = 30, width = 500)
tkgrid(dim)
```

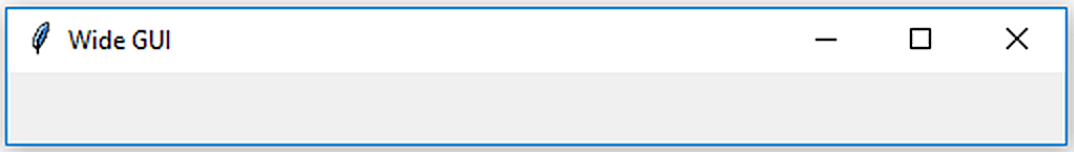


Figure 11.3: A *tcltk* GUI whose initial width was specified using `tkcanvas()`.

Three different geometry managers are available in Tcl/Tk for inserting widgets in GUIs. These are called: *placer*, *packer*, and *grid manager*. The placer tool is seldom used in GUI creation (Dalgaard, 2001). Thus, we will concentrate on GUI construction using packer and grid manager approaches. Only one of these approaches is generally used in the creation of a GUI. The initial Examples 11.2 and 11.3 use simple applications of packing and grid management, respectively.

11.2.3.1 Packing

The function `tkpack()` packs widgets around the edges of a conceptual cavity. Control of this process is provided by the `side`, which has options: `"left"`, `"right"`, `"top"` or `"bottom"`.

Example 11.4.
Note the result of the code below (Fig 11.4).

```
tt <- tktoplevel()
edge <- c("top","right","bottom","left")
buttons <- lapply(1:4,
                  function(i) tkbutton(tt, text = edge[i],
                     background = "lightgreen", foreground = "red"))
```

```
for (i in 1:4)
tkpack(buttons[[i]], side=edge[i], fill = "both")
```

- On Line 1, the top level widget is designated.
- On Line 2, a character vector is created, containing all the possible `side` options for the function `tkpack()`.
- On Lines 3-5, a four item list is generated containing four `tkbutton` widgets.
- In Lines 6-7, buttons are accessed from the `buttons` list and packed, in order, at the specified locations `"top"`, `"right"`, `"bottom"`, and `"left"`. The argument `fill = "both"` ensures that the buttons will occupy all of their allocated *parcels* with respect to the `tkpack()` conceptual central cavity. Because the `top` button was specified first, it takes up the entire top of the GUI. The `right` button, codified next, occupies the entire right-side of the GUI, except for the area now occupied by `top`, and so on. If an object does not fill its parcel it can be anchored to a GUI location using the `tkpack()` argument `anchor`. This is accomplished by specifying compass-style values like `"n"` or `"sw"` which place a widget the middle top, and bottom left of the parcel, respectively. The default option is `anchor = "center"`.

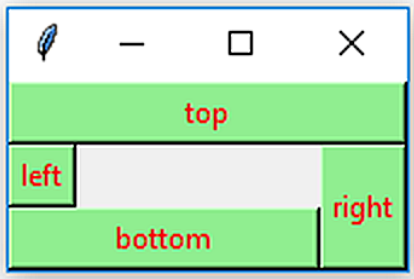


Figure 11.4: A demonstration of the result of packing using `tkpack()`. Code follows (Dalgaard, 2001).

■

Example 11.5.
Calculator construction is often used as a pedagogic exercise in computer programming. As an extended example of packing using `tkframe()`, we will build a *tcltk* calculator GUI. For this example I am indebted to lecture notes for a 2011 statistical programming course at UC Berkeley.

The most important coding concept used here is the pairing of the base **R** functions `parse()` (which converts a string to an expression) and `eval()` (which evaluates an expression). This combination allows the mathematical evaluation of a character string. Consider the string `"9 * 3"`. The mathematical solution can be obtained using:

```
txt = "9 * 3"
eval(parse(text = txt))
```

```
[1] 27
```

Our calculator GUI will require three functions: `mkput()`, `clearit()`, and `docalc()`. Each of these functions creates a global variable, `calcinp`, using the super assignment operator, `<<-`, that provides input to the calculator. Further, in all three functions, `tkconfigure()` is used to change the calculator's display, based on input from the GUI calculator keys.

```
1  calcinp <- ''
2
3  mkput <- function(sym){
4         function(){
5            calcinp <<- paste(calcinp, sym, sep='')
6              tkconfigure(display, text = calcinp)
7            }
8  }
```

- On Line 1 in the chunk above, `calcinp` is initially set to be an empty character string, i.e., `calcinp <- ''`.
- On Lines 3-8, the callback function `mkput` is defined. Note that `mkput` itself contains an argument-less function. This allows `mkput` to have an argument, `sym`, while satisfying the *tcltk* requirement for argument-less callbacks. The code on Line 5, `calcinp <<- paste(calcinp, sym, sep='')`, generates a global, updated form of `calcinp`, that combines an older `calcinp` value with a new calculator key specification, `sym`. The resulting string is placed in the display using `tkconfigure()`.
- The callback function `clearit` below, clears the display (Line 10), and redefines `calcinp` as an empty string (Line 11).

```
9  clearit <- function(){
10       tkconfigure(display, text = '')
11       calcinp <<- ''
12       }
```

- The callback function `docalc` below, evaluates the general `eval(parse(text = calcinp))` framework created by key entry, and provides exception handling in the case of key stroke errors by using `if(class(result) == 'try-error'); calcinp <<- 'Error'` on Lines 15-16. Importantly, the function `try()` (Line 14) will assign the class `try-error` to an expression that fails.

```
13 docalc <- function(){
14       result = try(eval(parse(text = calcinp)))
15       if(class(result) == 'try-error')
16         calcinp <<- 'Error'
```

```
17        else calcinp <<- result
18        tkconfigure(display, text = calcinp)
19        calcinp <<- ''
20        }
```

We call these three functions in the GUI itself, which is generated in the code below (Lines 21-60).

- The largest calculator code component defines the form and arrangement of calculator key (Lines 27-60). Note that buttons are packed, by row, using `tkpack()` within `tkframe()` objects. All button widgets use `command = mkput` except for the clear key, which uses `command = clearit`, and the equals key, which uses `command = docalc`.

```
21 base <- tktoplevel()
22 tkwm.title(base,'Calculator')
23
24 display <- tklabel(base,justify='right',background="white",
25                    relief="sunken", padx = 50)
26 tkpack(display,side='top')
27 row1 <- tkframe(base)
28 tkpack(tkbutton(row1,text='7',command=mkput('7'),width=3),side='left')
29 tkpack(tkbutton(row1,text='8',command=mkput('8'),width=3),side='left')
30 tkpack(tkbutton(row1,text='9',command=mkput('9'),width=3),side='left')
31 tkpack(tkbutton(row1,text='+',command=mkput('+'),width=3),side='left')
32 tkpack(row1,side='top')
33
34 row2 <- tkframe(base)
35 tkpack(tkbutton(row2,text='4',command=mkput('4'),width=3),side='left')
36 tkpack(tkbutton(row2,text='5',command=mkput('5'),width=3),side='left')
37 tkpack(tkbutton(row2,text='6',command=mkput('6'),width=3),side='left')
38 tkpack(tkbutton(row2,text='-',command=mkput('-'),width=3),side='left')
39 tkpack(row2,side='top')
40
41 row3 <- tkframe(base)
42 tkpack(tkbutton(row3,text='1',command=mkput('1'),width=3),side='left')
43 tkpack(tkbutton(row3,text='2',command=mkput('2'),width=3),side='left')
44 tkpack(tkbutton(row3,text='3',command=mkput('3'),width=3),side='left')
45 tkpack(tkbutton(row3,text='*',command=mkput('*'),width=3),side='left')
46 tkpack(row3,side='top')
47
48 row4 <- tkframe(base)
49 tkpack(tkbutton(row4,text='0',command=mkput('0'),width=3),side='left')
50 tkpack(tkbutton(row4,text='(',command=mkput('('),width=3),side='left')
51 tkpack(tkbutton(row4,text=')',command=mkput(')'),width=3),side='left')
52 tkpack(tkbutton(row4,text='/',command=mkput('/'),width=3),side='left')
```

```
tkpack(row4,side='top')

row5 <- tkframe(base)
tkpack(tkbutton(row5,text='.',command=mkput('.'),width=3),side='left')
tkpack(tkbutton(row5,text='^',command=mkput('^'),width=3),side='left')
tkpack(tkbutton(row5,text='C',command=clearit,width=3),side='left')
tkpack(tkbutton(row5,text='=',command=docalc,width=3),side='left')
tkpack(row5,side='top')
```

A slightly modified form of the GUI (with colored buttons)[3] is shown in Fig 11.5.

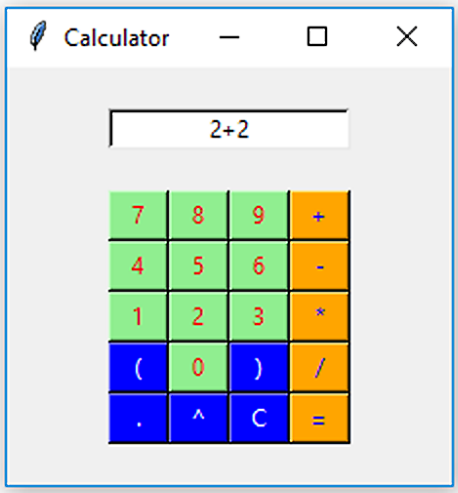


Figure 11.5: A Tcl/Tk calculator GUI generated using the **R** package *tcltk*.

The Python library *tkinter* provides a well supported binding resource for Tcl/Tk. As we know (Ch 9) Python code can be run in **R**, using the package *reticulate*. Fig 11.6 shows an analogous calculator the one shown in Fig 11.5, generated in **R** via the Python script `calc.py`, which is contained at the book website. It is important to note that while the resulting GUI is generated below in an RStudio **R** chunk, via *reticulate*, the code and engines for running the GUI are Python, and thus, do not actually require **R**.

```
library(reticulate)
py_run_file(source_python("https://amalgamofr.org/Ch11functions/calc.py"))
```

The function `reticulate::source_python()` allows one to access Python source code.

[3] Code for Fig 11.5 can be obtained using `source(url("https://amalgamofr.org/Ch11functions/calctcltk.R"))`.

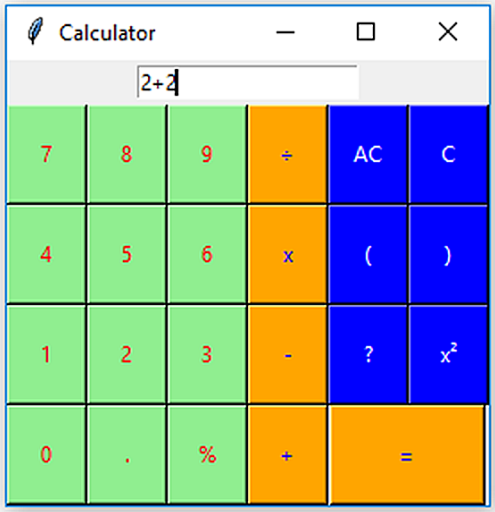


Figure 11.6: A Tcl/Tk calculator GUI generated using Python code via the Python binding library *tkinter*. Code follows a Python demo at the the geeksforgeeks website.

■

11.2.3.2 Grid Manager

Use of `tkpack()` and `tkframe()` provides a great deal of flexibility for creating GUI layouts. They are, however, insufficient for handling a number of issues including lining widgets up vertically and horizontally. The grid manager function `tkgrid()` can be used to lay out widgets in rows and columns using the arguments `column` and `row`. Importantly, `column = 0` and `row = 0` equate to the first column and first row, respectively, in a GUI or container widget. Additional important `tkgrid()` arguments include `columnspan`, `rowspan`, and `sticky`. The latter argument is analogous to `side` in `tkpack()`.

Example 11.6.
The callback function below creates a single large blue dot in an **R** graphics device whose vertical position can be altered with a slider widget.

```
1 plot.me <- function(){
2   y <- evalq(tclvalue(SliderValue)) # Evaluate the expression
3   plot(1, as.numeric(y),xlab = "", ylab = "%", xaxt = "n", ylim = c(0,100),
4        cex = 4, col = 4, pch = 19)
5 }
```

The operation `evalq(foo)`, (Line 2) above, is equivalent to `eval(quote(foo))`. The operation `quote(foo)` simply returns the argument `foo` as an object of class "call".

The GUI code below uses grid geometry to place widgets in specified GUI rows and columns.

```
6  if(names(dev.cur()) == "RStudioGD") dev.new(noRStudioGD = TRUE)
7
8  slider.env <<- new.env()
9  tt <- tktoplevel(); tkwm.title(tt, "Slider demo")
10 SliderValue <- tclVar("50")
11 SliderValueLabel <- tklabel(tt, text = as.character(tclvalue(SliderValue)))
12 tkgrid(tklabel(tt, text = "Slider Value: "),
13        SliderValueLabel, tklabel(tt, text = "%"))
14 tkconfigure(SliderValueLabel, textvariable = SliderValue)
15
16 slider <- tkscale(tt, from = 100, to = 0, showvalue = F,
17                   variable = SliderValue, resolution = 1,
18                   orient = "vertical", command = substitute(plot.me()))
19
20 tkgrid(slider, column = 0, row = 1, columnspan = 2)
21 message = tkmessage(tt, text = "Move the slider to see changes in the plot")
22 tkgrid(message, column = 3, row = 1)
```

- On Line 6, I insure that the interactive will work in the RStudio system by creating a non-RStudio graphics device if the default device is `"RStudioGD"`. The code should work inside and outside of RStudio.
- On Line 8, I create an environment for the slider widget using `new.env()`.
- On Line 9, I use use `tktoplevel()` to hierarchically define the "top level" widget as the object `tt`, and make a title.
- On Line 10, I define 50 as the initial value for the slider widget that will be created.
- On Line 11, a Tk label is created based on the slider output. Note the pairing of `tclVar()` input and `tclvalue()` output. In this process, the `SliderValueLabel` label object is configured to make its value equal to the `SliderValue` object.
- On Lines 12-13, the `SliderValueLabel` is inserted between two text strings in a grid geometry.
- On Lines 16-18, the slider is parameterized using the function `tkscale()`. The use of `substitute()` allows substitution of values for variables bound in the `plot.me()` function.
- On Line 20, the slider is placed into the GUI at column 0 and row 1 (the first column and second row).
- On Lines 21-22, a message is created and placed on the GUI at column 3 and row 1 (the fourth column and second row).

The form of the final GUI is shown in Figure 11.7.

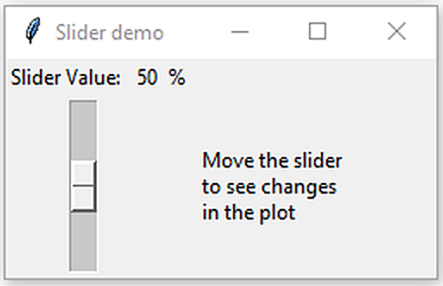


Figure 11.7: A *tcltk* GUI for manipulating an **R** plot.

■

Example 11.7.
Here is another grid manager example with a radio button GUI that allows selection of a bacterial phylum and printing of `"Correct"`, `"Incorrect"` text in the console, based on the button selection. It also embeds a photo.

```
tt <- tktoplevel()
tkwm.title(tt, "Bacterial phyla")
tkgrid(tklabel(tt, text = "Which phylum is shown?", padx = 5,
               pady = 5, font = "bold"), column = 1, row = 1)

values <- c("Acidobacteriota", "Armatimonadota",
            "Caldisericota", "Cyanobacteriota",
            "Elusimicrobiota", "Spirochaetota",
            "Thermomicrobia")

var <- tclVar(values[0]) # initially, no phyla selected

tkimage.create("photo", "cyano", file = "figs11/cyano.gif")

callback <- function() ifelse(tclvalue(var) == "Cyanobacteriota",
                              print("Correct"),
                              print("Incorrect"))

lf <- ttkframe(tt)
sapply(values, function(x) {
  radio_button <- ttkradiobutton(tt, variable = var,
                                 text = x, value = x,
                                 command = callback)

 tkgrid(radio_button, pady = 0, padx = 5, sticky = "nw",
```

```
26          column = 1, rowspan = 1)
27  })
28
29 tkgrid(tklabel(tt, text = ""), sticky = "n", column = 1,
30        row = 9, columnspan = 1)
31 l <- ttklabel(tt, image = "cyano", relief = "ridge")
32 tkgrid(l, sticky = "nw", rowspan = 10, column = 2,
33        row = 0, pady = 25, padx =  10)
```

- On Line 1, the top level widget, `tt`, is designated.
- On Line 2, a GUI title is created.
- On Lines 3-4, the text "Which phylum is shown?" is placed in the column 1, row 1 position, using the grid manager function `tkgrid()`.
- On Lines 5-9, a character vector of bacterial phylum names is created for use in the GUI.
- On Line 11 the initial phylum selection is specified. The use of `tclVar(values[0])` means that *no* selection will be initially designated.
- On Line 13, the function `tkimage.create()` is used to import a photo with .gif formatting (currently the only accepted format). The first argument `"photo"` indicates that an image will be created from a photo. The second argument creates an object name for the import, `"cyano"` that will called in later code.
- On Lines 15-17, a callback function, `callback()` is created to print the text `"correct"` if Cyanobacteriota is selected, and print `"incorrect"` if some other selection is made.
- On Line 19, an embedded frame is created to hold the radio buttons.
- On Lines 20-27, `sapply()` is used to run a user-defined function that embeds radio buttons for each level in the character vector `values`.
  - On Line 21-23, the function creates an object `radio_button` using `ttkradiobutton()`. Note that the top-level path name, `tt` is given as the first argument, the initial radio button designation is given in the second argument, the arguments `text` and `value` will change levels in `vaues` change. the function `callback()` is called using the `ttkradiobutton()` `command` argument to respond to the selected radio button.
  - On Lines 25-26, radio buttons are stacked, one row at a time, using `tkgrid()`, as `sapply()` cycles through levles in `values`.
- On Lines 29-30, an aesthetic empty row is created at the bottom of column one create some additional space.
- On Lines 31-32, the object `l` is created from the function `ttklabel()` to hold the image object `cyno`, created on Line 13.

- On Lines 33-34, the image is embedded into the entirety of column two.

The final form of the GUI is shown in Fig 11.8.

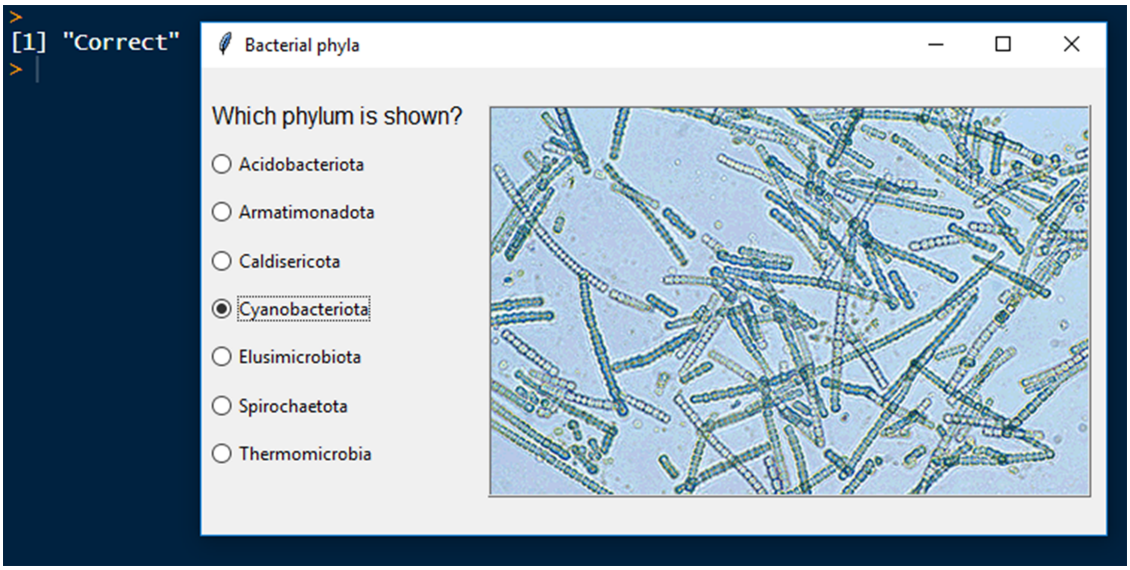


Figure 11.8: A radio button *cltk* GUI. Photo from CSIRO, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=3548094

■

11.2.4 Widget Modifications

Tcl/Tk provides shared modification settings for many of its widgets. These include the standard arguments `foreground` (the widget foreground color, see choices here), `background` (the widget background color), `image` (an image to display in the widget)[4], `relief` (the 3D appearance of the widget), `font`, and `text` (a text string to be placed in the widget), among many others. Be aware, however, that standard Tk widget modifiers may not always align with Ttk modifiers.

Example 11.8.

Here is an example of a GUI with working (but non-functional) widgets that allows demonstration of relief, background, and foreground color options for different types of widgets.

```
types = c("flat", "groove", "raised", "ridge",
          "solid", "sunken")
bg = c("beige", "AntiqueWhite1", "aquamarine4", "burlywood3")
fg = c("BlueViolet","aquamarine3", "white", "black")

base = tktoplevel()
tkwm.title(base,"Widget Styles")
```

[4]This requires creation with `tkimage.create()` (see Example 11.7).

```
8
9  cnames <- tkframe(base)
10 tkpack(tklabel(cnames, text = "Labels", font = "bold"),
11        side = "left", padx = 3)
12 tkpack(tklabel(cnames, text = "Buttons", font = "bold"),
13        side = "left", padx = 17)
14 tkpack(tklabel(cnames, text = "Radio buttons", font = "bold"),
15        side = "left", padx = 0)
16 tkpack(tklabel(cnames, text = "Sliders", font = "bold"),
17        side = "left", padx = 22)
18 tkpack(cnames, side = "top", fill = "both")
19
20 mkframe <- function(type){
21   fr <- tkframe(base)
22   tkpack(tklabel(fr, text = type, relief = type,
23                  fg = fg[1], bg = bg[1]),
24          side = "left", padx = 5)
25   tkpack(tkbutton(fr, text = type, relief = type,
26                  fg = fg[2], bg = bg[2]),
27          side = "left", padx = 30)
28   tkpack(tkradiobutton(fr, text = type, relief = type,
29                  fg = fg[3], bg = bg[3]),
30          side = "left", padx = 10)
31   tkpack(tkscale(fr, from = 0, to = 10, showvalue = T,
32                  variable = 1, resolution = 1,
33                  orient = "horizontal",
34                  relief = type, fg = fg[4],
35                  bg = bg[4]), side = "left", padx = 14)
36
37   tkpack(fr, side = "top", pady = 5)
38 }
39
40 sapply(types, mkframe)
```

See Fig 11.9.

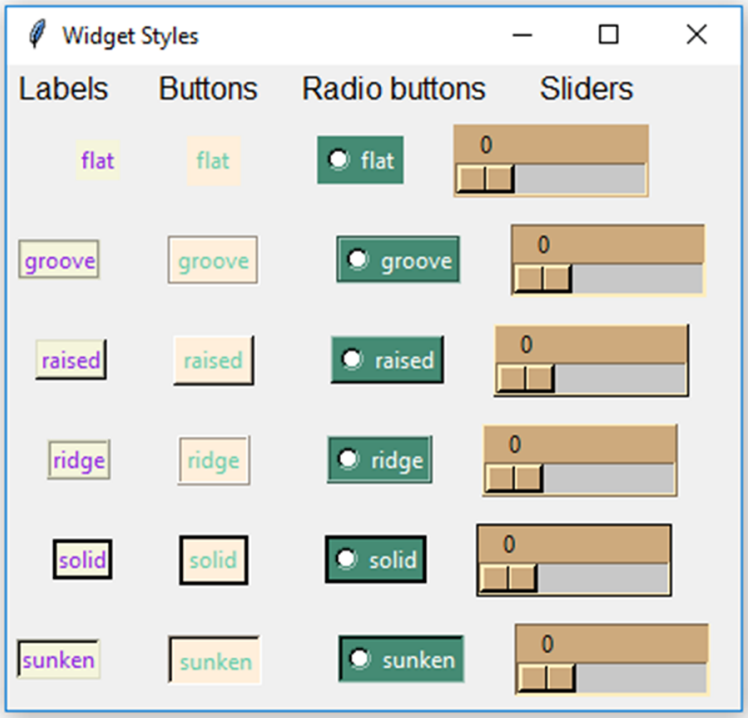


Figure 11.9: Standard widget modifications. Relief styles are varied within each widget type, and background and foreground colors are varied among widget types.

■

11.2.5 Additional tcltk Packages and Toolkits

Several **R** packages have been developed to streamline and extend the capacities of the *tcltk* package. These include *fgui* (Hoffmann and Laird, 2009) and *PBSmodelling* (Schnute et al., 2023, 2013) which provides wrappers for some Tcl/Tk routines to simplify and facilitate GUI creation. The `gWidgets2` package has ambitiously sought to create **R** simplifying binding frameworks for several GUI toolkits including GTK, Qt, and Tcl/Tk[5]. Currently, however, only the `gWidgets2` port for `tcltk`, called *gWidgets2tcltk*, is working. The *gWidgets2tcltk* package is currently used to build interactive self test questions for the pedagogic statistics package *asbio* (Fig 11.10).

```
asbio::selftest.typeIISS.tck1()
```

[5]GTK (formerly GIMP ToolKit), GTK+, and Qt (Section 11.6) are open-source, cross-platform, toolkits for creating GUIs.

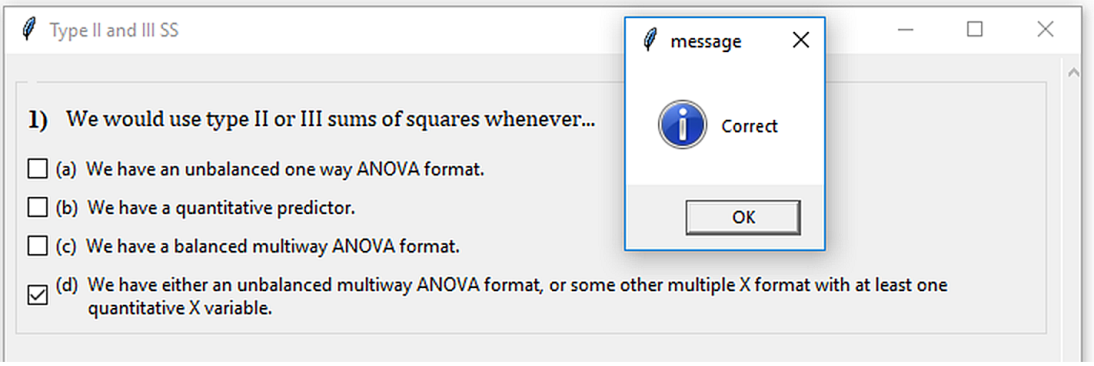


Figure 11.10: A self-test GUI using the *gWidgets2tcltk* function `gcheckboxgroup`. For GUI code type: `fix(selftest.typeIISS.tck1)`.

A number of Tcl/Tk extensions for **R** are available from the *SciViews* family of packages (Grosjean, 2024), including *svDialogs* (an attempt at creating standard cross-platform dialog boxes), *svGUI*, and *tcltk2*. These packages, however, have not been updated (at least on CRAN) for several years.

11.3 JS and JSON Interactive Apps

Many newer interactive **R** applications are implemented using bindings for the JavaScript language[6] (JS) or JavaScript Object Notation[7] (JSON). GUIs generated from these approaches are often embedded in an HTML[8] format, and thus can be viewed from web browsers.

There are many **R** packages for generating JS and JSON apps that are rendered in HTML. The two most popular are *plotly* and *shiny*. These packages, and several others are briefly summarized below. The website https://gallery.htmlwidgets.org/ lists many other amazing **R** packages and examples.

- *plotly* can be used to create a wide variety of interactive HTML/web graphics via the *plotly.js* JS library for interactive charts.
- *shiny* can generate high-quality widget controls that communicate with **R** in real time to produce HTML-embedded analytical results and graphics. The package uses other packages, chiefly *htmltools* and *htmlwidgets*, to provide **R** bindings to JS libraries and HTML code.

[6]Java is an OOP language designed to have few dependencies. Once compiled, Java code can run on all platforms that support Java. Additional details for Java web design are given here. JavaScript, while linguistically similar to Java, has many many important differences. For instance, JavaScript is an interpreted language, whereas Java code is generally compiled.

[7]JavaScript Object Notation (JSON) was derived from JavaScript largely to facilitate server-to-browser session communication.

[8]As noted in Section 2.10.2, HTML (Hypertext Markup Language) is the standard language for structuring web pages and web content. Basic HTML programming details are available from a number of sources, including this Mozilla developer site.

- *dygraphs* is an **R** interface to the *dygraphs* JS charting library. It creates high quality interactive graphs, similar in character to *plotly*, with an emphasis on time series data.

- *leaflet* is an **R** interface to the *leaflet* JS library for creating interactive maps that allow zooming and panning.

- *highcharter* is an **R** API for *Highcharts* JS library and its modules. Creates high-quality interactive graphs similar to those from *plotly*.

- *rbokek* is an **R** interface to the *Bokeh* Python visualization library, which converts Python source code and output into a JSON format. The *rbokek* package can be currently used to produce interactive graphics similar to those from *plotly*.

- *visNetwork* is an **R** package that binds the *vis.js* JS library. The provides high-quality interactives for network graphs and dendritic representations, including classification trees.

- *networkD3* provides **R** bindings for *D3* JS library. Similar to *visNetowrk*, the package provides high-quality interactives for network graphs, including tree diagrams.

- *rglwidget* creates **R** bindings for the JS library WebGL. The package renders hand-rotatable three-dimensional rgl graphics (see Section 6.22) in HTML.

I recommend that readers take the time to explore all these packages, many of which require minimal coding for the generation of GUIs. Because of my own time and space constraints, however, I will focus only on the *plotly* and *shiny* packages over the next two sections of this Chapter.

11.4 plotly

The package *plotly* (Sievert, 2020), uses the **R** package *jsonlite*, which provides an **R** binder for JSON. JSON code is read by the JS library *plotly.js* to create interactive HTML embedded graphics (Fig 11.11). Charts resulting from this process are interactive under a standardized *plotly* format, although they don't represent GUIs in a conventional sense.

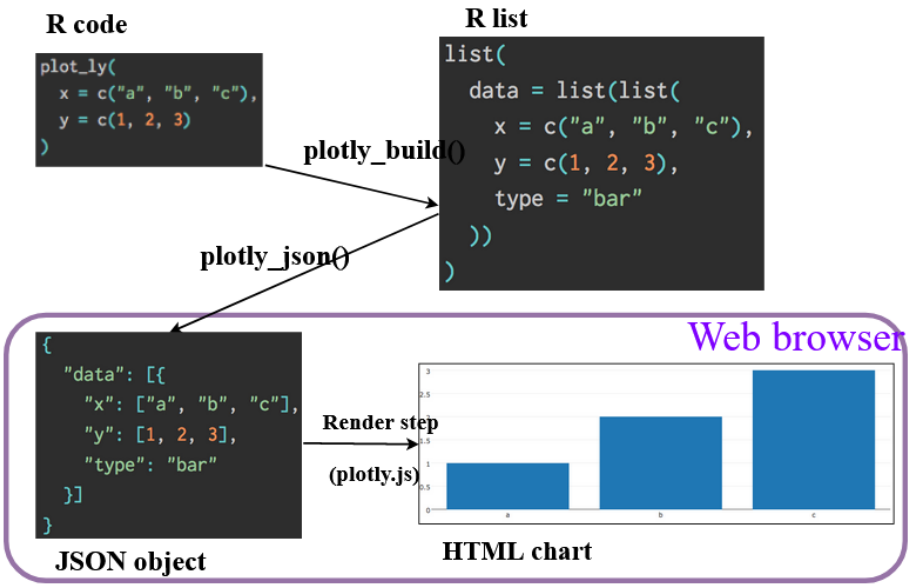


Figure 11.11: A graph from Sievert (2020) that shows the process of creating an HTML-embedded plotly chart from a graph generated in **R**.

Example 11.9.

To provide a simple demonstration I bring in some packages, including *plotly*, and the `world.emissions` data from package *asbio*.

```
library(tidyverse); library(plotly)
library(asbio); data(world.emissions)

subset <- world.emissions |>
  filter(country %in% c("United States", "Mexico", "China",
                        "Germany", "Russia", "Canada")) |>
  filter(year > 1950 & year <= 2019)
```

Plotly graphs are rendered using the function `plot_ly()`.

```
plot_ly(subset, x = ~year, y = ~co2) |>
  add_lines(color = ~country) |>
  layout(
        yaxis = list(tickfont = list(size = 20),
                     title ='CO\U2082 (million tonnes)',
                     titlefont = list(size = 23)),
        xaxis = list(tickfont = list(size = 20),
                     title='Year',
                     titlefont = list(size = 23)),
        legend = list(font = list(size = 20))
```

```
18          )
```

- On Line 8, I call `plot_ly()`. Note the use of the tilde operator to call *x* and *y* axis variables, i.e., `x = ~year, y = ~co2`.
- On Lines 9-18, I call additional plot characteristics using *tidyverse* pipe operators.
  - On Line 9 I use `add_lines()` to add a line *trace* to the plot.
  - Plot characteristics can be modified in a large number of ways using lists within the function `layout` (Lines 10-18).

The result is shown in Fig 11.12. If you are viewing an HTML-rendered version of this book, the lines in the plot will be interactive, and the graphical device will contain a menu that allows summarization of single or multiple data points, panning and zooming.

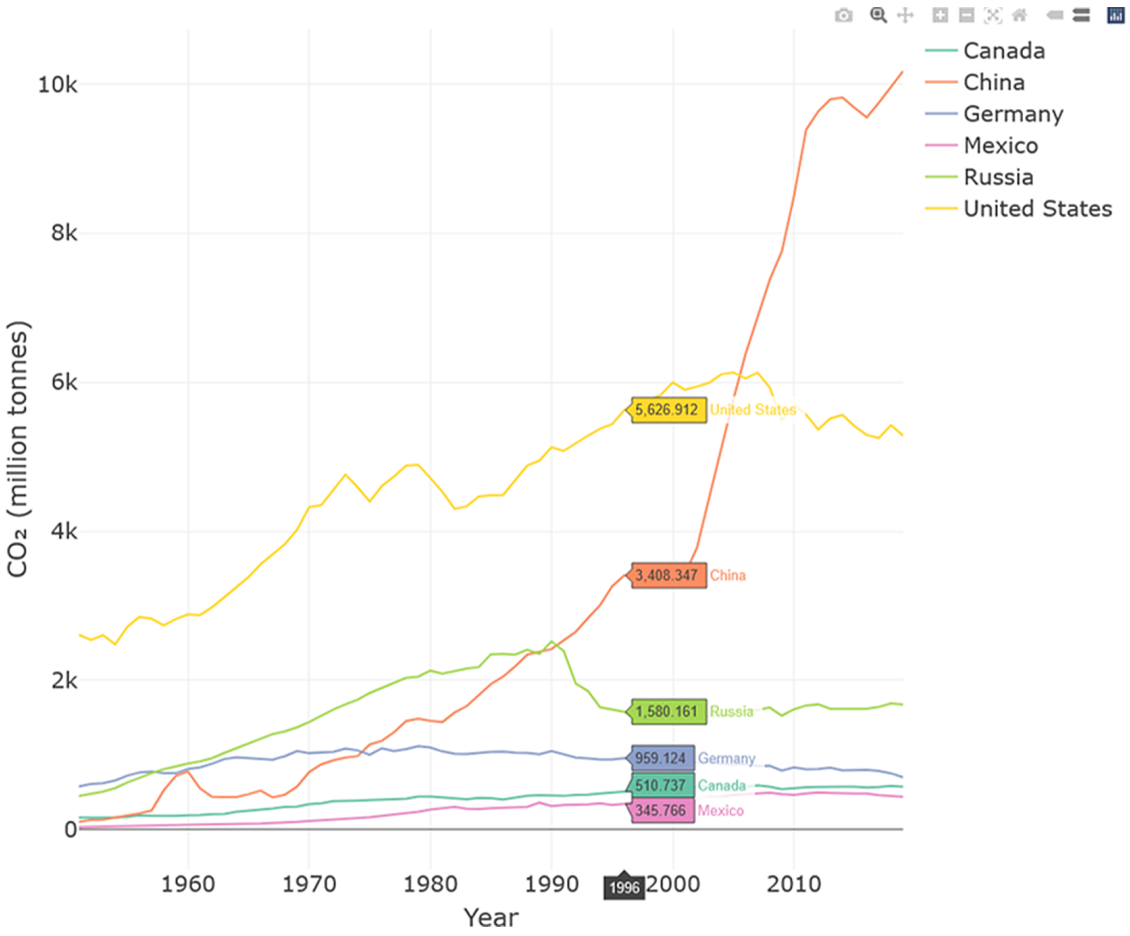


Figure 11.12: Simple *plolty* chart with an interactive trace and standard *plotly* menu shown (topright).

■

11.4.1 ggplot **and** plotly

Ggplot2 objects can be converted (imperfectly) to *plotly* objects using the function `plolty::ggplotly()`. Unfortunately, a large number `ggplot` layout characteristics including figure margins and locations of x and y axis labels will not translate to `ggplotly()`. Instead, we must call on potentially exhaustive hierarchically nested list components. This can be a pain, and it may be expedient to build separate list or function objects to facilitate the process.

Example 11.10.

To illustrate I extend the previous example. First, I create a nested list object, `k`, that specifies desired margin and axis characteristics.

```
k <- list(
  yaxis = list(title = list(font = list(size = 18)),
               tickfont = list(size = 14)),
  xaxis = list(title = list(font = list(size = 18)),
               tickfont = list(size = 15)),
    margin = list(t = 20, r = 20, b = 80, l = 80))
```

Here I create a simple ggplot boxplot, `g`. To get the desired characteristics in the mapped plotly graph I call on list components in `k` within `plotly::layout()`. Fig 11.13 shows the result.

```
g <- ggplot(subset, aes(x = country, y = co2)) +
  geom_boxplot(aes(fill = country)) +
  theme_classic() +
  ylab("CO\U2082 (million tonnes)") +
  xlab("Country")

ggplotly(g) %>%
  layout(showlegend = FALSE, xaxis = k$xaxis, yaxis = k$yaxis,
         margin = k$margin)
```

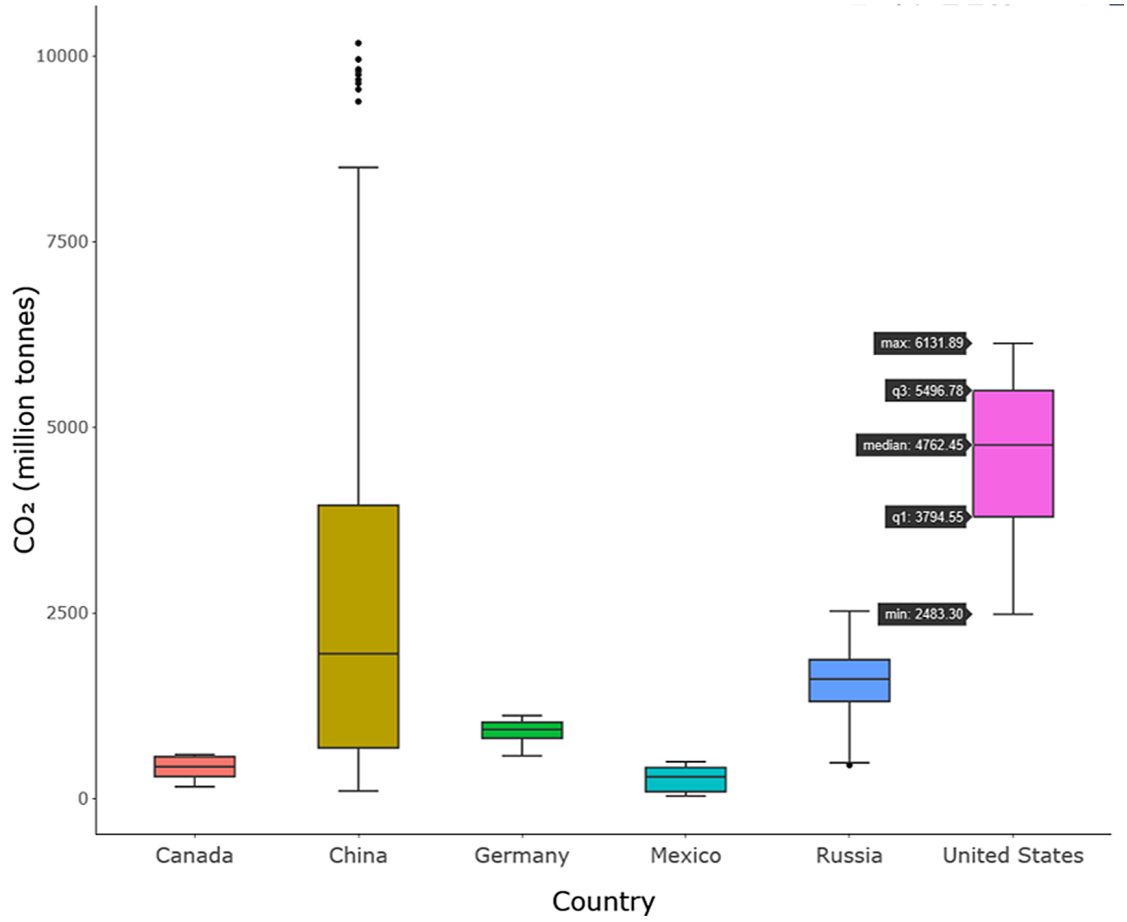


Figure 11.13: Simple *plolty* barplot, based on a ggplot. Interactive trace for the US shown.

■

Example 11.11.

Here is another application using the function `GGally::ggcoef()` to create a coefficient plot. A coefficient plot displays statistical model parameter estimates and confidence intervals. We can make the plot interactive (in HTML) using `ggplotly()` (Fig 11.14).

```
library(GGally)
model <- lm(log(co2) ~ country + year,
            data = subset)

gg <- ggcoef(model,
             exclude_intercept = TRUE,
             errorbar_height = .1,
             color = "blue") +
  theme_bw()
```

```
ggplotly(gg) %>%
  layout(yaxis = list(tickfont = list(size = 15),
                      title ='',
                      titlefont = list(size = 18)),
         xaxis = list(tickfont = list(size = 15),
                      title='Parameter estimates',
                      titlefont = list(size = 18)),
          legend = list(font = list(size = 15))
          )
```

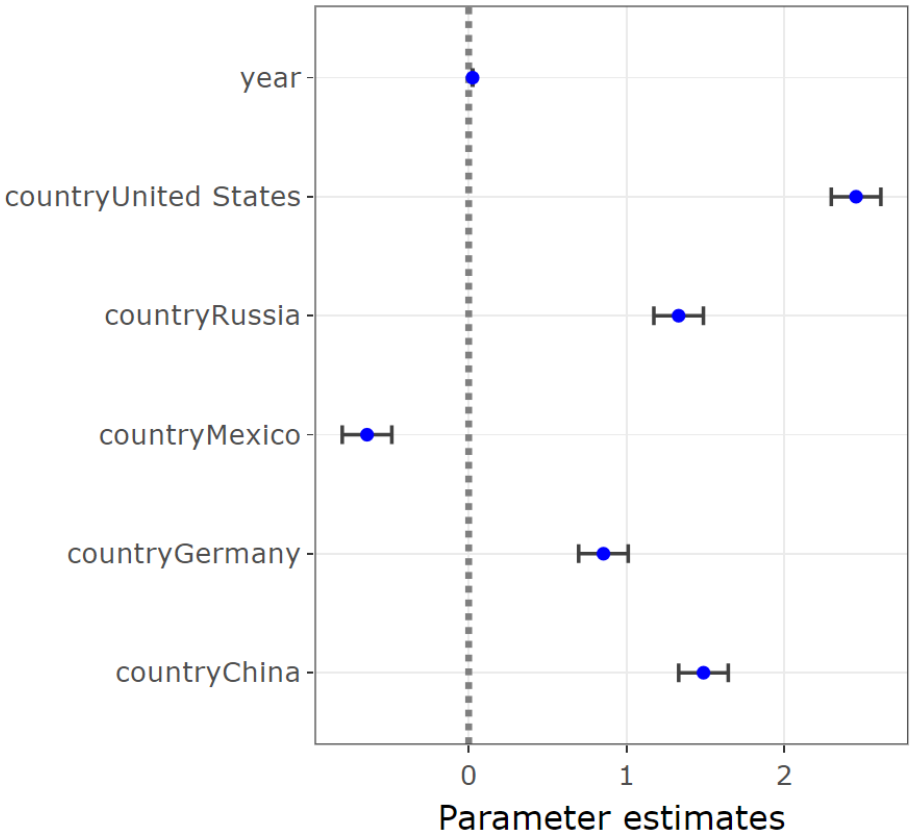


Figure 11.14: Coefficient plot from the function `GGally::ggcoef()`, rendered using `ggplotly()`.

■

11.5 shiny

Probably the easiest and most flexible way to create interactive HTML apps is through tools in the package *shiny*. Unlike *ploty* apps, *shiny* apps will allow real-time interfacing with **R**

for computations. RStudio has internals to facilitate *shiny* app creation for embedding on webpages. Examples given here are often based on apps described in Hadley Wickham's book *Mastering Shiny* (Wickham, 2021).

A *shiny* app will have three components:

- A user interface (`ui`) specification that defines how your app *looks*.
- A server function that defines how your app *works*. The function will (generally) have three arguments `input`, `output`, and `session`.
- An app execution call that conjoins the user interface and server functions. This is done with `shinyApp()`

Example 11.12. As a first example, here is **R** code for rendering text in an HTML app (Fig 11.15).

```
1 library(shiny)
2 ui <- fluidPage(
3   "Hello, world!"
4 )
5 server <- function(input, output, session) {
6 }
7 shinyApp(ui, server)
```

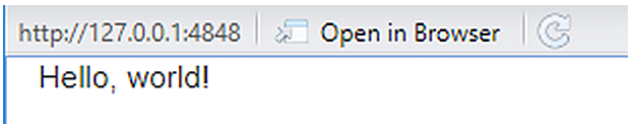


Figure 11.15: A very simple *shiny* app. Code follows Wickham (2021).

- **`ui`:** `shiny::fluidPage()` is a layout function that defines the basic visual structure of the app (Lines 1-4). Among other things the function allows definition of app rows using `shiny::fluidRow()`, and columns (within rows) using `shiny::column()`. Fluid pages can rescale their components in real-time to fill the available GUI window width.
- **`server`:** For this simple example the `server` function contains no commands (Lines 5-6). Although, as a matter of convention, the `server` arguments: `input, output, session` are still included.
- **`shinyApp`:** The app function pairs the `ui`/`server` objects (Line 7).

■

One can open an **R** script with a *shiny* app skeleton in RStudio by going to **File**>New File>Shiny Web Application. This will allow RStudio to recognize the script as app code. This, in turn, allows running the app by either sending its code to the console (e.g., using Ctrl + Enter), or by using the **Run App** button in the **Shiny Web Application** toolbar.

Example 11.13. Here is a simple app that lists and provides details about datasets in the package *asbio* (Fig 11.16).

```
ui <- fluidPage(
  selectInput("dataset", label = "Dataset",
              choices = data(package = "asbio")$results[,3])
)
server <- function(input, output, session) {
}
shinyApp(ui, server)
```

- **ui:** `fluidPage()` includes `shiny::selectInput()`, an input control function that provides the user with a select box widget. An appropriate label `"Dataset"` is defined. Selection box `choices` are the third column in `data(package = "asbio")$results`, which contains the names of the dataframe object names in *asbio*.
- **server:** Once again, the `server` function contains no commands (Lines 5-6).
- **shinyApp:** The app function again pairs the `ui`/`server` objects (Line 7).

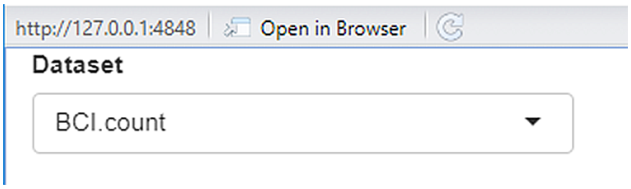


Figure 11.16: A *shiny* app to allow scrolling through *asbio* datasets.

The app in Fig 11.16 has limited usefulness because it provides only the `asbio` dataframe object names. Indeed, we could get more information by simply running `data(package = "asbio")`. Here we insert additional features into the user interface and server to increase functionality (Fig 11.17).

```
adata <- data(package = "asbio")$results[,3]
data(list = adata[1:length(adata)]) # loads all asbio datasets

ui <- fluidPage(
  selectInput("dataset", label = "Dataset", choices = adata),
  verbatimTextOutput("summary"),
  tableOutput("table")
)

server <- function(input, output, session){
    output$summary <- renderPrint({
    dataset <- get(input$dataset)
    summary(dataset)
  })

  output$table <- renderTable({
```

```
17        dataset <- get(input$dataset)
18      dataset
19    })
20  }
21
22  shinyApp(ui, server)
```

- The code `data(list = adata[1:length(adata)])` loads all the *asbio* datasets into the global environment (Line 2).

- **`ui`:** `fluidPage()` now specifies three features, which will occur from top to bottom app, as they are listed (Lines 4-8).

  - The functions`shiny::verbatimTextOutput()` (Line 6) and `shiny::tableOutput()` (Line 7) are controls that define how and where output (depending on the order they are specified in `fluidPage()`) are displayed. Specifically, `verbatimTextOutput()` displays code, and `tableOutput()` displays tables.

- **`server`:** The `server` function has been modified to allow interaction with the user interface (Lines 10-20). It allows generations of two objects: `output$summary` (Line 11) and `output$table` (Line 12), based on `input$dataset` from the `ui`.

  - `output$summary` (lines 10-14) is a rendered expression from `shiny::renderPrint()`. In particular, this will be output from `summary()` (Line 13) for columns in `input$dataset` which is made available in the object `dataset` on Line 12. The output operation is coupled with `verbatimTextOutput("summary")` in the `ui`.

  - `output$table` (Lines 16-19) is a rendered expression from `shiny::renderTable()`. It will show the raw data in a scrollable table for the dataframe specified in the `ui`. This output operation is coupled with `tableOutput("table")` in the `ui`.

- **`shinyApp`:** As before, we generate the app using: `shinyApp(ui, server)` (Line 22).

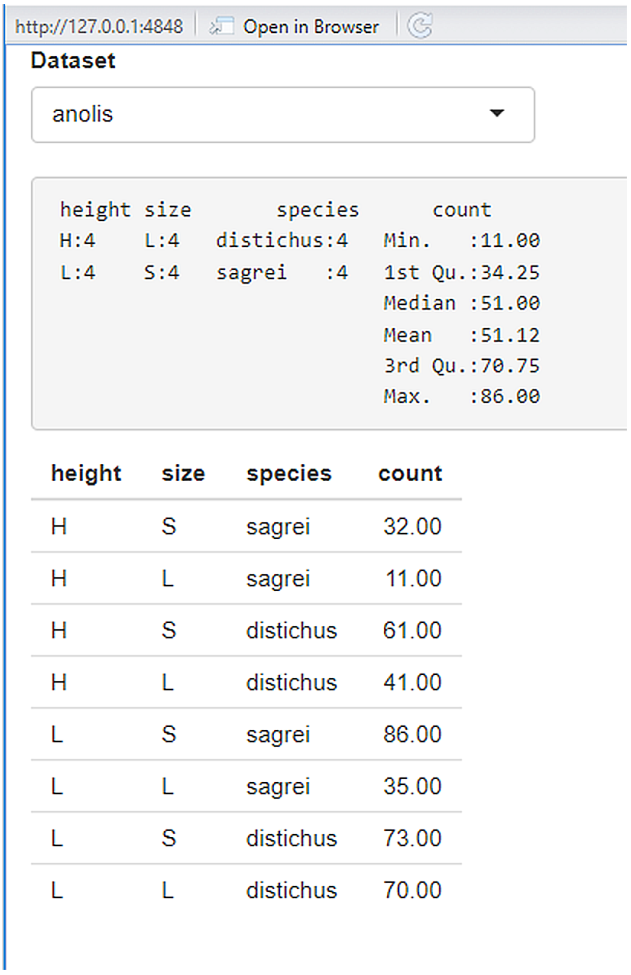


Figure 11.17: A modified *shiny* app to provide summaries of scrollable *asbio* datasets.

■

11.5.1 `ui` Details

Input widget functions are specified in the `ui`. A `ui` input function, e.g., `selectInput()` with first argument `"foo"`, or with `inputId = "foo"` can be called by `server` operations using the script `input$foo`. Most input functions have a 2nd argument called `label` that creates a user-readable label for the control widget on the app. A The 3rd input argument is typically `value` which creates a starting value for the widget control. Fig 11.18 shows many of the standard *shiny* `ui` input functions (without output). Important `ui` import functions are also listed Table 11.1.

```
source("shiny_widgets.R")
shiny_widgets()
```

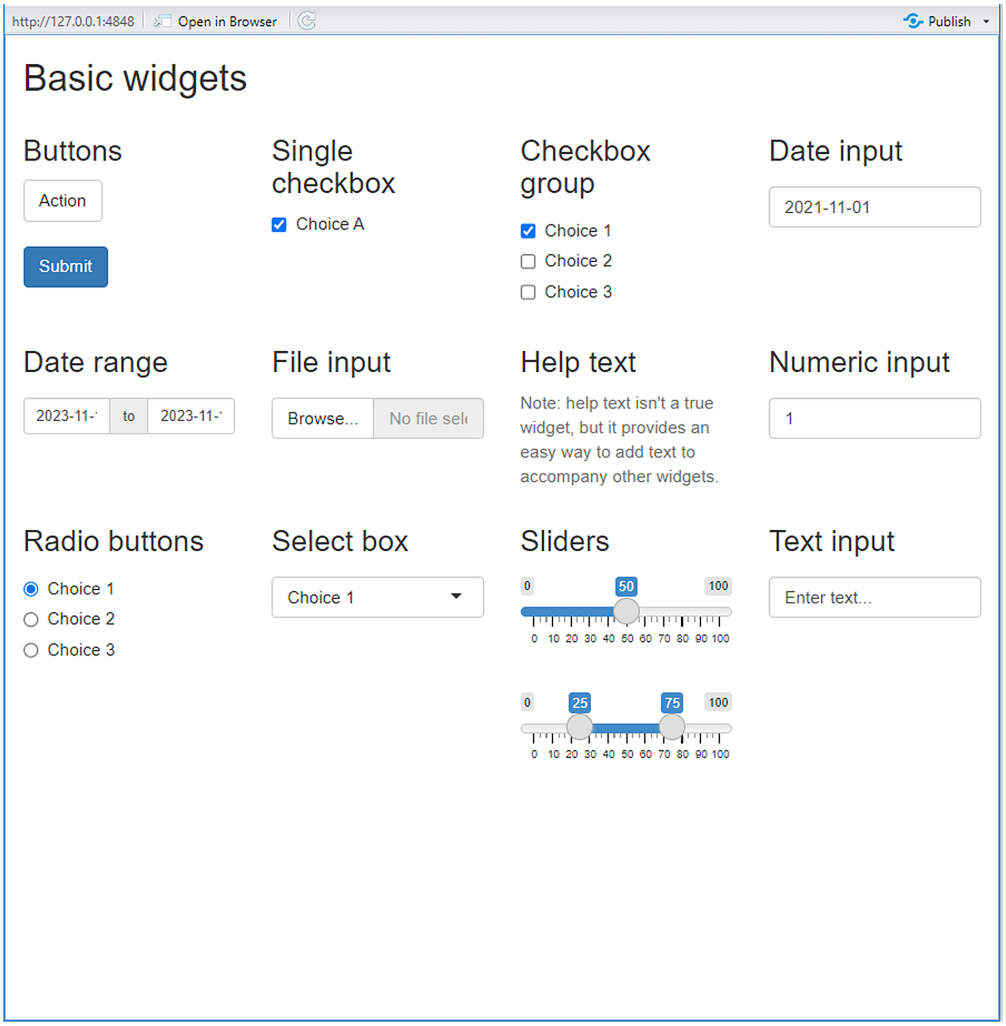


Figure 11.18: A variety of *shiny* input widgets and operations that can be specified in the `ui`. By row, the figure depicts widgets generated by the functions: `actionButton()`, `submitButton()`, `checkboxInput`, `checkboxGroupInput()`, `dateInput()`, `dateRangeInput()`, `fileInput()`, `helpText()`, `numericInput()`, `radioButtons()`, `selectInput()`, `sliderInput()`, and `textInput()`. Also see Table 11.1.

Table 11.1: Some important *shiny* `ui` input functions.

Function	Purpose
`actionButton()` `actionLink()`	Creates an action button or link.
`submitButton()`	Create a submit button.
`checkboxInput()`	Create a checkbox to specify logical values.
`dateInput()`	Create a selectable calendar.
`dateRangeInput()`	Create a pair of selectable calendars.
`fileInput()`	Create a file upload control to upload one or more files.
`helpText()`	Create help text which can be added to input to provide additional information.
`numericInput()`	Create an input control for entry of numeric values
`radioButtons()`	Create a set of radio buttons to select item from a list.
`selectInput()`	Create a selectable list, from which single or multiple items can be selected.
`sliderInput()`	Constructs a slider widget to elect a number, date, or date-time.
`passwordInput`	Create an control for entry for passwords.
`textInput()`	Create an input control for unstructured text

11.5.1.1 Output

Output functions in the `ui` create placeholders that can filled by the `server` function. As with inputs, outputs require a unique ID. For instance, in Fig 11.19, which provides a simple summary of the `asbio::world.emissions` dataset, `output$code` and `output$text` generated in the `server` are placed in the fluid page using `textOutput("text")` and `verbatimTextOutput("code")`.

```
library(asbio)
data(world.emissions)

US <- world.emissions |>
  filter(country == "United States")

ui <- fluidPage(
  textOutput("text"),
  verbatimTextOutput("code")
)
server <- function(input, output, session) {
  output$text <- renderText({
    "Summary of the US CO\u2082 data \n"
  })
  output$code <- renderPrint({
```

```
    summary(US$co2)
  })
}

shinyApp(ui, server)
```

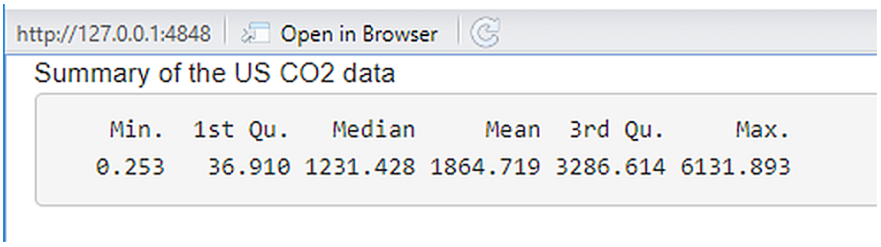


Figure 11.19: A simple app providing a single descriptive statistics summary.

Table 11.2 lists some potential `ui` output functions that can be used in *shiny* apps.

Table 11.2: Some *shiny* `ui` output functions.

Function	Purpose
`downloadButton()`	Create a download button or link.
`downloadLink()`	To be paired with `downloadHandler()` in `server`.
`htmlOutput()` `uiOutput()`	Create an HTML output element.
`imageOutput()` `plotOutput()`	Create a plot or image output element. To be paired with `renderPlot()` and `renderImage()`, respectively, in `server`.
`outputOptions()`	Set options for an output object.
`modalDialog()` `modalButton()`	Create a modal dialog interface.
`showNotification()` `removeNotification()`	Show or remove a notification.
`textOutput()` `verbatimTextOutput()`	Create a text output element. To be paired with `renderText()` and `renderPrint()`, respectively, in `server`.
`urlModal()`	Generate a modal dialog that displays a URL.

11.5.2 `server` Details

As noted earlier, the `server` function requires three arguments: `input`, `output`, and `session`. The `input` argument allows assembly of items from the `ui` front-end to create a list-like

object. The `output` argument in `server` provides output for `ui` inputs, often via rendering and handling functions (Table 11.3).

Table 11.3: Some *shiny* `server` rendering and handling functions.

Function	Purpose
`downloadHandler()`	Create a download button or link. To be paired with `downloadButton()` and `downloadLink()` in `ui`.
`renderPlot()` `renderImage()`	Create a plot or image output element. To be paired with `imageOutput()` and `plotOutput()`, respectively, in `ui`.
`renderText()` `renderPrint()`	Create a text output element. To be paired with `textOutput()` and `verbatimTextOutput()`, respectively, in `ui`.

Fig 11.20 shows an app with sliders inputs, generated using the function `sliderInput()` in the `ui`, and text (product) output which is provided to the `ui` from the `server`, via `renderText()`.

```
ui <- fluidPage(
  sliderInput("x", label = "If x is", min = 1, max = 50, value = 30),
  sliderInput("y", label = "And y is", min = 1, max = 50, value = 30),
  "then x times y is",
  textOutput("product")
)

server <- function(input, output, session) {
  output$product <- renderText({
    input$x * input$y
  })
}

shinyApp(ui, server)
```

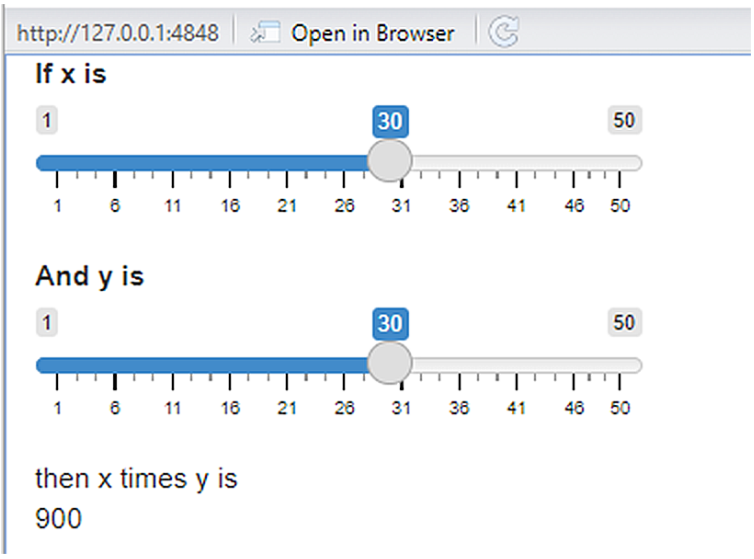


Figure 11.20: A simple slider app.

11.5.3 Running shiny Apps in R Markdown

Unlike *plotly* graphics, *shiny* apps are not automatically interactive in an **R** markdown rendered document. However, it is easy to a make shiny app interactive in this setting (provided that **R** is open to run the app). One simply adds `runtime:shiny` to the **R** Markdown YAML header. Thus, the YAML header in **R** markdown should have the format of Fig 11.21. Note that this approach will now be possible under Bookdown.

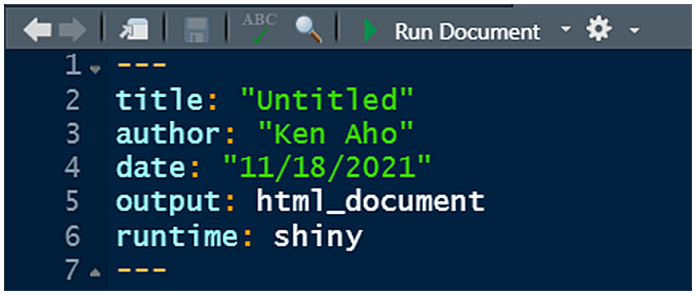


Figure 11.21: YAML header to allow inclusion of *shiny* apps in an **R** Markdown generated HTML.

Some adjustments to **R** and Markdown code may be needed (e.g., figure margins in `renderPlot()` may need to be changed) to make apps fit nicely on a page. A *shiny* app will only work remotely (outside of an **R** session) if a server implementing **R** is used to implement the app's code. This process will be detailed at the end of the chapter.

11.5.4 Reactive Programming

We can increase efficiency in Shiny apps using reactive programming wherein outputs automatically update as inputs change. Under reactive programming we specify interactive dependencies so that when an input changes, all related outputs are automatically updated. The code below results in the app shown in Fig 11.22. Note that the output updates "reactively'' as I type individual characters of my name.

```
ui <- fluidPage(
  textInput("name", "What's your name?"),
  textOutput("greeting")
)

server <- function(input, output, session) {
  output$greeting <- renderText({
   paste0("Hello ", input$name, "!")
  })
}
shinyApp(ui, server)
```

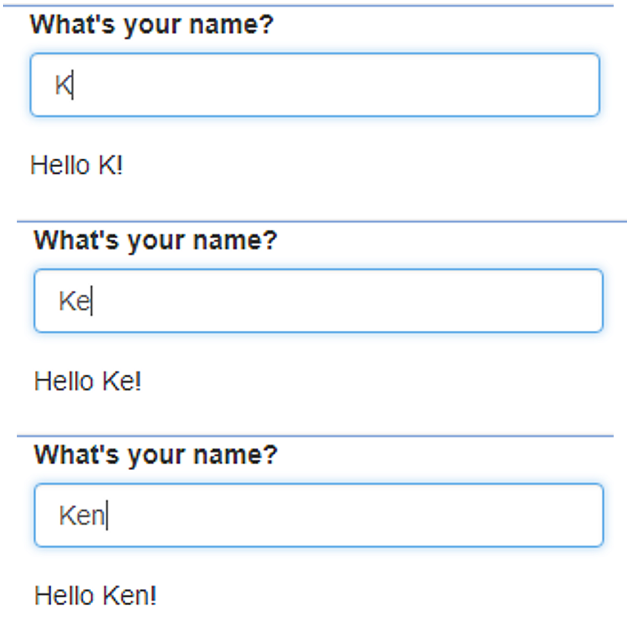


Figure 11.22: Reactive behavior of a simple *shiny* app.

Reactive programming usually occurs in more complex settings than the previous example and requires the function `reactive`.

Example 11.14.
As an extended example, imagine we wish to rapidly examine green house emissions data for the fifty countries in the `asbio::world.emissions` dataset with the highest current populations.

As a first step, we do some data tidying and create a stats summary callback function we will use later.

```
library(tidyverse)
not.redundant <- world.emissions |> filter(continent != "Redundant")

pop.max <- not.redundant |>
  group_by(country) |>
  summarise(max.pop = max(population)) |>
  arrange(desc(max.pop))

# names of 50 largest countries
country_names <- setNames(nm = pop.max$country[1:50])

# 50 largest countries data
top50 <- not.redundant |>
  filter(country %in% pop.max$country[1:50])

# summary stats
summarize <- function(x, rn = c("CO2", "CH4", "NOx", "total GHG")){
    mean <- apply(x, 2, function(x) mean(x, na.rm = T))
    max <- apply(x, 2, function(x) max(x, na.rm = T))
    sum <- apply(x, 2, function(x) sum(x, na.rm = T))
    n <- apply(x, 2, function(x) length(which(!is.na(x))))
    df <- data.frame(mean = mean, max = max, cumulative = sum, n = n)
    row.names(df) <- rn
    df
}
```

- `ui`: We define a relatively complex `ui` that will provide sufficient inputs and outputs for the `server` function.

```
ui <- fluidPage(
  titlePanel(h1("Greenhouse gasses", align = "center")), #h1 = HTML heading
  fluidRow(
    column(6,
      selectInput("country", choices = country_names, label = "Country")
    )
  ),
  fluidRow(
    column(4, tableOutput("diag")), # table
  ),
  br(), # line break
  br(),
  fluidRow(
```

```
    column(12, plotOutput("plot")) # plot
  )
)
```

- Note that we use the `fluidRow()` layout function. Shiny app rows, created with `fluidRow()`, contain twelve columns. These can be divided up in various ways using `shiny::column()`(Fig 11.23).
- The functions `htmltools::br()` and `htmltools::h1()` are HTML tags from the package *htmltools*, which is imported by *shiny*. The `h1()` function creates a first level heading. Thus, it is equivalent to the HTML operator `<h1>`. The `br()` functions equates to an HTML line break tag, i.e., `<br>`. A list of HTML equivalent tags is provided in `?htmltools::builder`.
- The function `plotOutput()` allows input of interactive **R** graphs, including ggplots (see below).

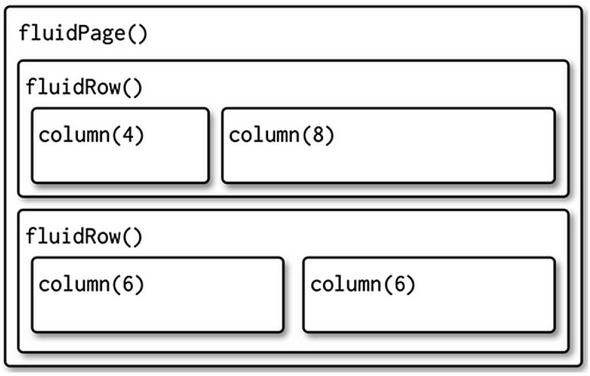


Figure 11.23: Behavior of *shiny* app rows and columns, in the `ui`. Figure taken from Wickham (2021).

- `server`:: The `server` function contains several new features, including use of the function `reactive()`.

```
server <- function(input, output, session) {
  selected <- reactive(top50 %>% filter(country == input$country))

  output$diag <- renderTable(
  summarize(select(selected(), co2, methane, nitrous_oxide, total_ghg)),
  colnames = TRUE, rownames = TRUE
  )

  output$plot <- renderPlot({
  selected() %>%
       ggplot(aes(year, co2)) +
       geom_line() +
```

```
        labs(x = "Year",
             y = expression(paste(CO[2], " emissions (", 10^6, " tonnes)")))
  })
}
```

- The code:

```
selected <- reactive(top50 %>% filter(country == input$country))
```

provides a data subset for a particular country that only needs to be calculated once, and then re-used. This also allows spontaneous (as possible) interaction with the `ui` with respect to this choice. The reactive object `selected` is called several times, as a function, in the `server` function. - The object `output$plot <- renderPlot()` will be a ggplot generated from `selected()` which will be called by `plotOutput("plot")` in the `ui`.

- **`shinyApp`:** As before, we generate the app using:

```
shinyApp(ui, server)
```

The final form of the app is shown in Fig 11.24.

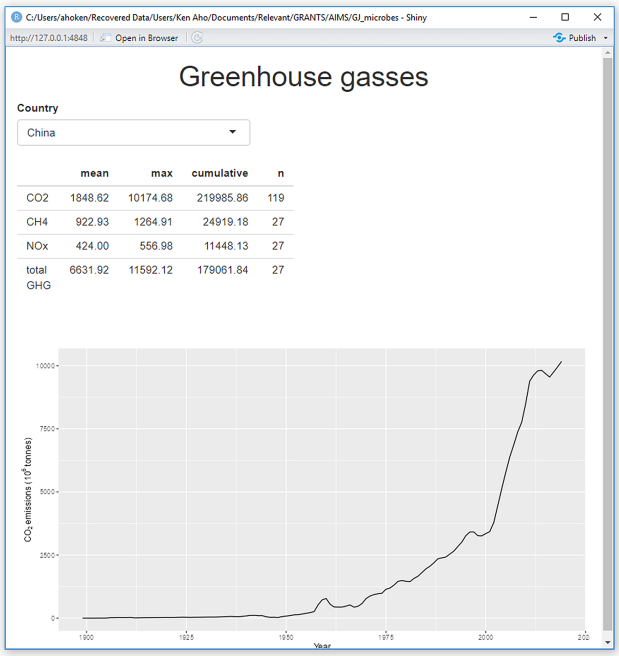


Figure 11.24: A *shiny* app to graphically depict changing CO_2 levels over time for a user-selected country.

■

Example 11.15.
Wickham (2021) used t-test computations to demonstrate reactive programming as shown

(with some modifications) below. We first delineate two callback functions we wish to use in the app.

```
freqpoly <- function(x1, x2, binwidth = 0.1, xlim = c(-3, 3)) {
  require(ggplot2)
  df <- data.frame(
    x = c(x1, x2),
    group = c(rep("x1", length(x1)), rep("x2", length(x2)))
  )
  ggplot(df, aes(x, colour = group)) +
    geom_freqpoly(binwidth = binwidth, linewidth = 1) +
    coord_cartesian(xlim = xlim)
}

t_test <- function(x1, x2) {
  test <- t.test(x1, x2)
  sprintf(
    "p-value: %0.3f \nCI for μ1 - μ2: [%0.2f, %0.2f]",
     test$p.value, test$conf.int[1], test$conf.int[2]
  )
}
```

The function `t.test()` runs t-tests for true normal population means. In particular, assuming $X_1 \sim N(\mu_1, \sigma_1^2), X_2 \sim N(\mu_2, \sigma_2^2)$ we generally consider the hypotheses:

$$\begin{aligned} \mathrm{H}_0 &: \mu_1 = \mu_2 \\ \mathrm{H}_\mathrm{A} &: \mu_1 \neq \mu_2 \end{aligned}$$

By default, `t.test()` does not assume homoscedasticty (that is, it allows $\sigma_1^2 \neq \sigma_2^2$). Thus, it uses the Satterthwaite method to estimate degrees of freedom for the null t-distribution of the test statistic (Aho, 2014). The GUI we will create will run t-tests on randomly generated data from two user-specified normal distributions `x1` and `x2`.

The function `sprintf()` in `t_test()` uses C code to return a formatted combination of text and variable outcomes. The code below combines text and inputs for double precision values (indicated with `f`) for p-values, and bounds for a 95% confidence interval for a true mean difference. The code `%0.3f` indicates rounding to three significant digits. As before, the code `\n` creates a text line break.

- `ui`: we use the function `numericInput()` to specify characteristics of the normal distributions under consideration. The `sliderInput()` function is used to specify x-limits in app-rendered ggplot frequency plot.

```
ui <- fluidPage(
  fluidRow(
    column(4,
      "Distribution 1",
      numericInput("n1", label = "n", value = 200, min = 1),
      numericInput("mean1", label = "μ", value = 0, step = 0.1),
      numericInput("sd1", label = "\u03c3", value = 0.5, min = 0.1, step = 0.1)
    ),
    column(4,
      "Distribution 2",
      numericInput("n2", label = "n", value = 200, min = 1),
      numericInput("mean2", label = "μ", value = 0, step = 0.1),
      numericInput("sd2", label = "\u03c3", value = 0.5, min = 0.1, step = 0.1)
    ),
    column(4,
      "Frequency polygon",
      numericInput("binwidth", label = "Bin width", value = 0.1, step = 0.1),
      sliderInput("range", label = "range", value = c(-3, 3), min = -5, max = 5)
    )
  ),
  fluidRow(
    column(12, plotOutput("hist"))
  ),
  fluidRow(
    column(1),
    column(5, verbatimTextOutput("ttest")),
    column(2),
    column(3, actionButton("simulate", "Simulate!")),
    column(1)
  )
)
```

- **server:** In the server we use reactive programming to generate random samples from a normal distribution. Specifically, for the object `x1` we obtain a random sample of size `input$n1` from a normal distribution with a mean of `input$mean1` and a standard deviation of `input$sd1`. These parameter values are specified in the `ui`.

```
server <- function(input, output, session) {
  x1 <- reactive({input$simulate
        rnorm(input$n1, input$mean1, input$sd1)})
  x2 <- reactive({input$simulate
        rnorm(input$n2, input$mean2, input$sd2)})

  output$hist <- renderPlot({
    freqpoly(x1(), x2(), binwidth = input$binwidth, xlim = input$range)
  }, res = 96)
```

```
  output$ttest <- renderText({
    t_test(x1(), x2())
  })
}
```

- **shinyApp:** As before, we generate the app using:

```
shinyApp(ui, server)
```

The final form of the app is shown in Fig 11.25.

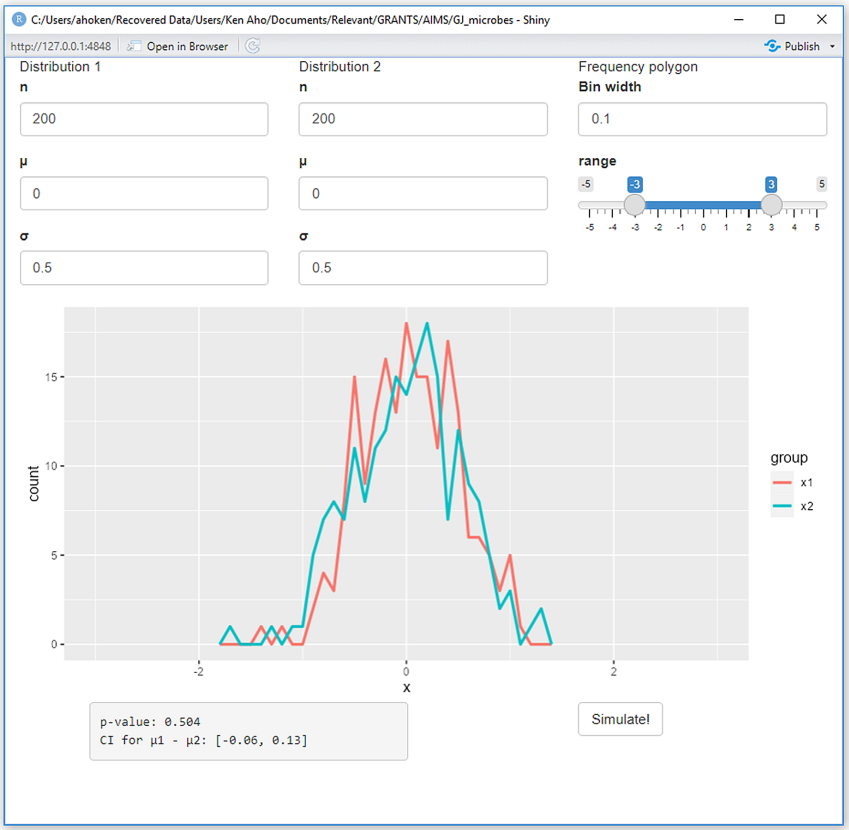


Figure 11.25: A *shiny* app to demonstrate the mechanism of t-tests.

■

11.5.5 Additional Layout Control

We have already learned about techniques like `fluidRow()` to control single page layouts in `fluidPage()`. Another popular HTML layout uses side panels. These can be implemented in the *shiny* `ui` using the functions `sidebarLayout()` and `sidebarPanel()`. Sidebar formatting is summarized in Fig 11.26.

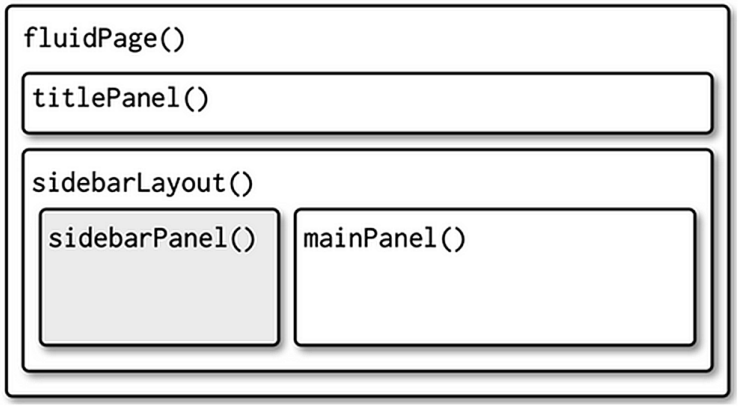


Figure 11.26: Sidebar formatting for *shiny*. Figure taken from Wickham (2021).

Example 11.16.
Here is an example for displaying a normal distribution using sliders in sidebars.

- **ui:** The user interface specifies a sidebar layout, using `sidebarLayout()`, that contains a sidebar panel designated with `sidebarPanel`, and a main panel, designated with `mainPanel()`.

```
ui <- fluidPage(
   titlePanel(h1("Normal Distribution", align = "center")),
   sidebarLayout(
      sidebarPanel(
         sliderInput("mu", "\u03BC", step = 0.2, min = -3,
                     max = 3, value = 0),
         sliderInput("sigma", "\u03C3", min = 0.5, max = 3,
                     value = 1), width = 4
      ),
    mainPanel(plotOutput("plot"))
))
```

- **server:** The only output from the `server` is a base **R** plot of the normal PDF.

```
server <- function(input, output, session) ({
   xmin <- -4; xmax <- 4; ymin <- 0; ymax <- 0.8
   xx <- seq(xmin, xmax, length = 100)

   output$plot <- renderPlot({
       yy <- dnorm(xx, input$mu, input$sigma)
       plot(xx, yy, type = "l", xlim = c(xmin, xmax), ylim = c(ymin, ymax),
            xlab = expression(italic(x)),
            ylab = expression(paste(italic(f), "(", italic(x), ")", sep = "")),
            cex.axis = 1.2, cex.lab = 1.2, lwd = 1.4)
```

```
    })
})
```

} - **shinyApp:** As before, we use `shinyApp()` to generate the app.

```
shinyApp(ui, server)
```

The final form of the app is shown in Fig 11.27.

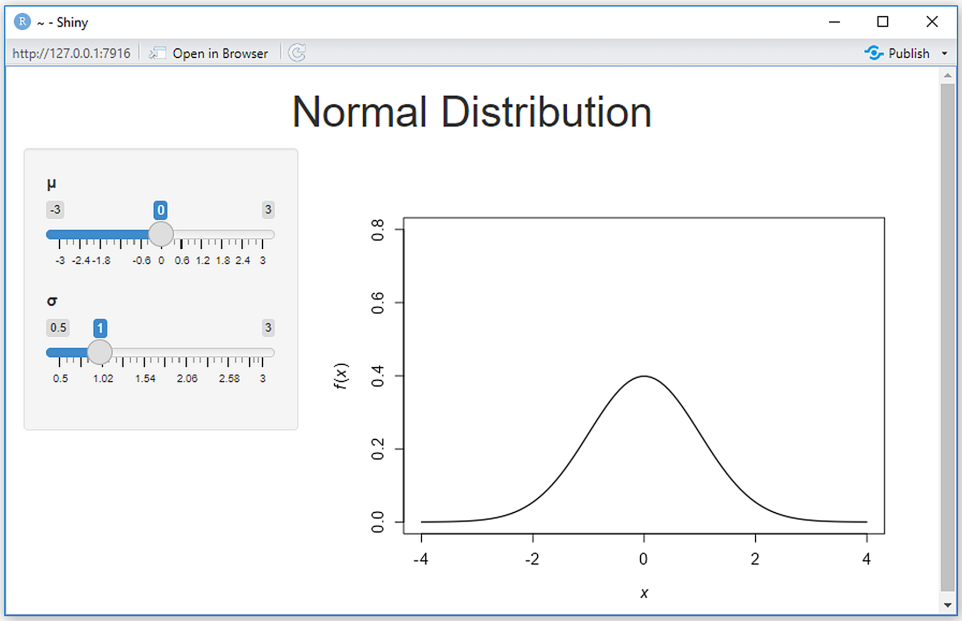


Figure 11.27: A *shiny* app for demonstrating the normal distribution.

Notably, the app is less reactive than an analogous normal distribution GUI generated with *tcltk*, in the package *asbio* due to the fact that *tcltk* GUIs communicate more directly with **R**. Compare the app from Fig 11.27 to `asbio::see.norm.tck()`.

■

11.5.5.1 Multi-page Apps

Complex apps may be impossible to fit onto a single page. In *shiny*, the simplest way to break a app page into multiple pages is to use `tabsetPanel()` and `tabPanel()`. Wickham (2021) that does not provide widget output because of its empty `server` (Fig 11.28). In the `ui` a tabset panel is generated using `tabsetPanel()`. This entity has three panels, each is generated using `tabPanel()`. Only the first panel `"Import data"` currently contains content.

```
ui <- fluidPage(
  tabsetPanel(
    tabPanel("Import data",
      fileInput("file", "Data", buttonLabel = "Upload..."),
      textInput("delim", "Delimiter", ""),
      numericInput("skip", "Rows to skip", 0, min = 0),
      numericInput("rows", "Rows to preview", 10, min = 1)
    ),
    tabPanel("Set parameters"),
    tabPanel("Visualise results")
  )
)

server <- function(input, output, session) {
}
shinyApp(ui, server)
```

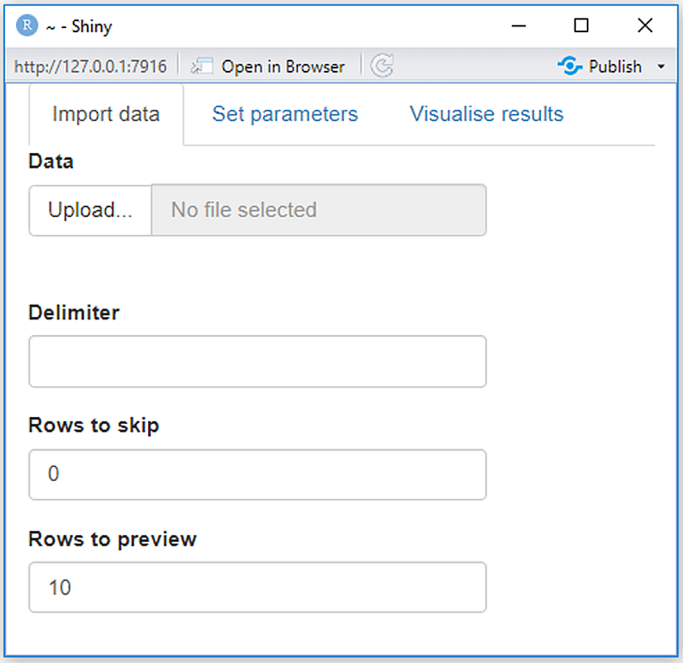


Figure 11.28: A *shiny* multipanel example.

A tabset can be an input when its `id` argument is used. This allows an app to behave differently depending on which tab is currently visible (Fig 11.29).

```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      textOutput("panel")
    ),
    mainPanel(
      tabsetPanel(
        id = "tabset",
        tabPanel("panel 1"),
        tabPanel("panel 2"),
        tabPanel("panel 3")
      )
    )
  )
)
server <- function(input, output, session) {
  output$panel <- renderText({
    paste("Current panel: ", input$tabset)
  })
}
shinyApp(ui, server)
```

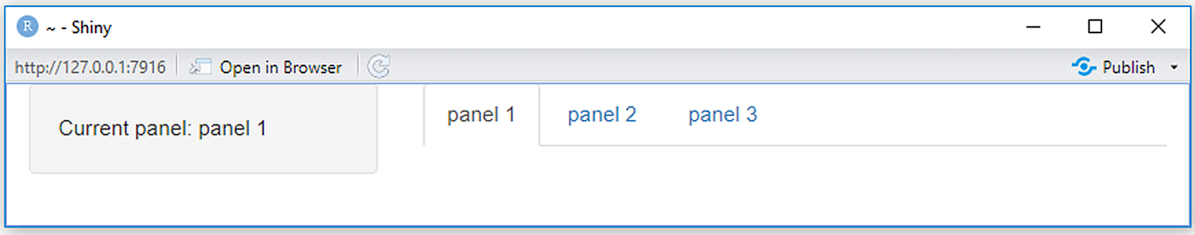


Figure 11.29: Tabs for a multi-page app.

Because tabs are displayed horizontally, there is a limit to their number. The functions `navlistPanel()`, `navbarPage()`, and `navbarMenu()` provide vertical layouts that allow more tabs with longer titles (Fig 11.30).

```
ui <- fluidPage(
  navlistPanel(
    id = "tabset",
    "Heading 1",
    tabPanel("panel 1", "Panel one contents"),
    "Heading 2",
    tabPanel("panel 2", "Panel two contents"),
```

```
    tabPanel("panel 3", "Panel three contents")
  )
)

server <- function(input, output, session) {
}
shinyApp(ui, server)
```

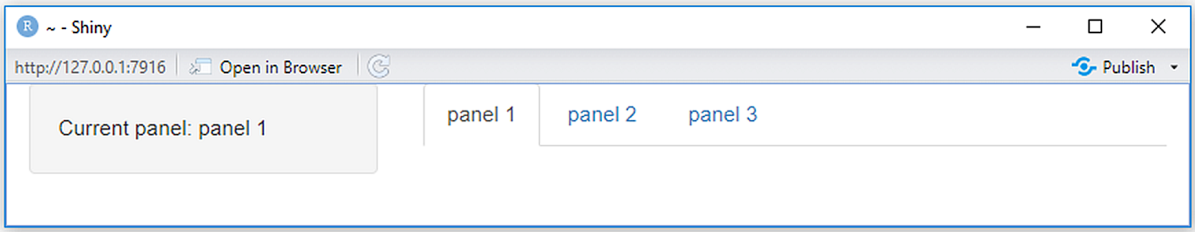


Figure 11.30: A multi-page app with vertical tabs.

11.5.5.2 Layout Themes

Customization of the general *shiny* layout can be obtained by utilizing or modifying **Bootstrap**[9] themes and classes. These can include layouts specific to mobile apps (see package *RInterface*) and Google's material design frame (see package *shinymaterial*).

Here we use the `"darkly"` bootswatch (Fig 11.31). Other choices include `"sandstone"`, `"flatly"`, and `"united"`.

```
ui <- fluidPage(
  theme = bslib::bs_theme(bootswatch = "darkly"),
  sidebarLayout(
    sidebarPanel(
      textInput("txt", "Text input:", "text here"),
      sliderInput("slider", "Slider input:", 1, 100, 30)
    ),
    mainPanel(
      h1(paste0("Theme: darkly")),
      h2("Header 2"),
      p("Some text")
    )
  )
)

```

[9]Bootstrap is a collection of HTML conventions, Cascading Style Sheets (CSS) styles (CSS is a language used to style HTML documents, including colors and fonts), and java script snippets bundled into a convenient form.

```
server <- function(input, output, session) {
}
shinyApp(ui, server)
```

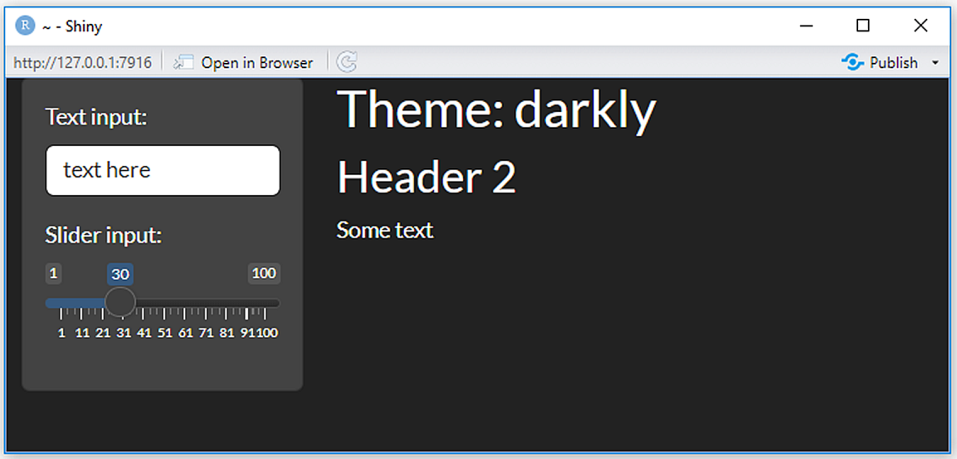


Figure 11.31: A *shiny* app using the `darkly` bootswatch from Bootstrap.

Further control of *shiny* apps can be achieved by programming directly in HTML, CSS, and Java[10]. In fact, HTML code in `uis` is revealed by running `ui` functions directly in the **R** console (Fig 11.32).

```
> fluidPage(
+     textInput("name", "What's your name?")
+ )
<div class="container-fluid">
  <div class="form-group shiny-input-container">
    <label class="control-label" id="name-label" for="name">What's your name?</label>
    <input id="name" type="text" class="form-control" value=""/>
  </div>
</div>
>
>
```

Figure 11.32: Representation of *shiny* code as HTML code.

11.5.6 `plotOutput` Interactives

One of the perks of `plotOutput()` is that it can be an input that responds to mouse pointer events. Such controls are also possible with *tcltk* GUIs. A *shiny* plot can respond to four different mouse events: `click`, `dblclick` (double click), `hover` (i.e., the mouse stays in the same place), and `brush` (a rectangular selection tool).

[10] For more information, check this **R** studio help link, and this book by David Granjon

Example 11.17.
Consider the following simple example:

```
US <- world.emissions %>% filter(country == "United States")

ui <- fluidPage(
  plotOutput("plot", click = "plot_click"),
  verbatimTextOutput("info")
)

server <- function(input, output) {
  output$plot <- renderPlot({
    par(mar = c(5,5,2,2))
    plot(US$year, US$co2, xlab = "Year",
         ylab = expression(paste(CO[2], " (",10^6, " tonnes)")), type = "l")
  }, res = 96)

  output$info <- renderPrint({
    req(input$plot_click)
    x <- round(input$plot_click$x, 2)
    y <- round(input$plot_click$y, 2)
    cat("[year = ", x, ", CO2 = ", y, " million tonnes]", sep = "")
  })
}
shinyApp(ui, server)
```

Note the use of `req()`, to ensure the app doesn't do anything before the first click. The resulting app is shown in Fig 11.33.

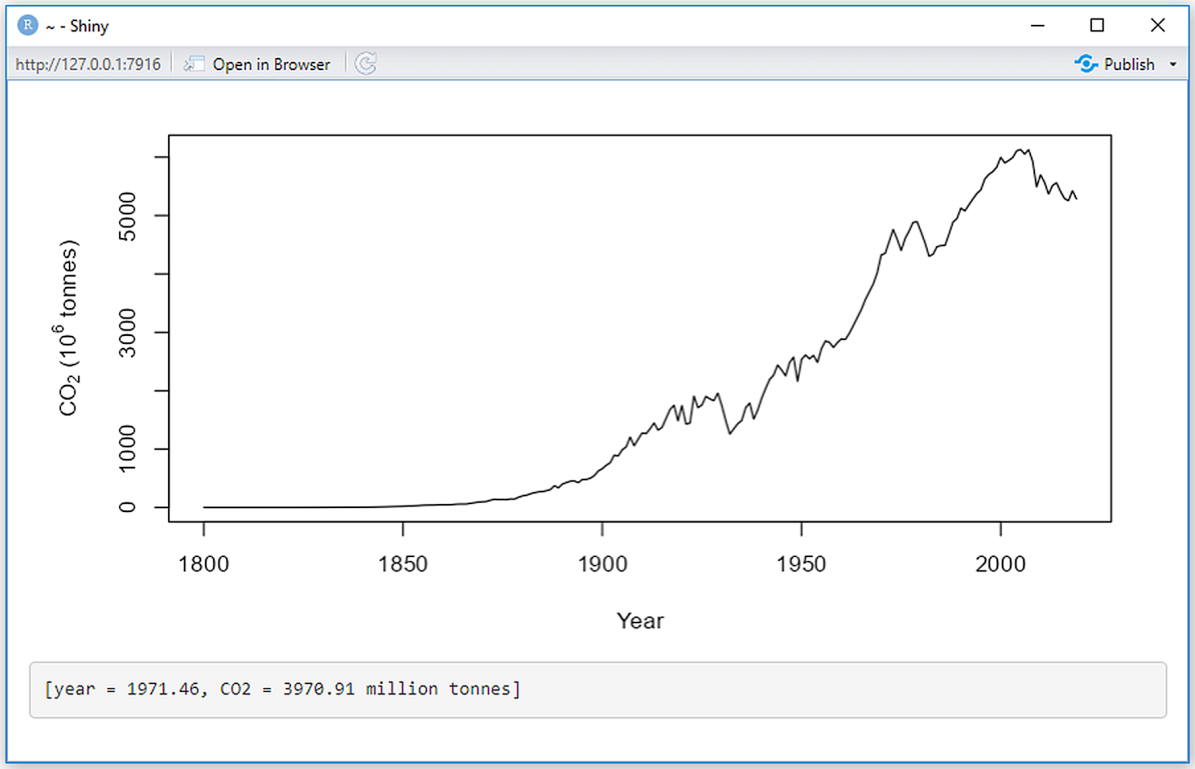


Figure 11.33: A mouse interactive *shiny* app.

Here we use `nearPoints()` to return a dataframe for a point *near* a mouse click.

```
US.ghg <- data.frame(US[,c(3,4,10,11,12,14,15)])

ui <- fluidPage(
  plotOutput("plot", click = "plot_click"),
  tableOutput("data")
)
server <- function(input, output, session) {
  output$plot <- renderPlot({
    par(mar = c(5,5,2,2))
    plot(US.ghg$year, US.ghg$co2, xlab = "Year",
         ylab = expression(paste(CO[2], " (",10^6, " tonnes)")),
         type = "l")
  }, res = 96)

  output$data <- renderTable({
    nearPoints(US.ghg, input$plot_click,
               xvar = "year", yvar = "co2")
```

```
  })
}
shinyApp(ui, server)
```

The resulting app is shown in Fig 11.34.

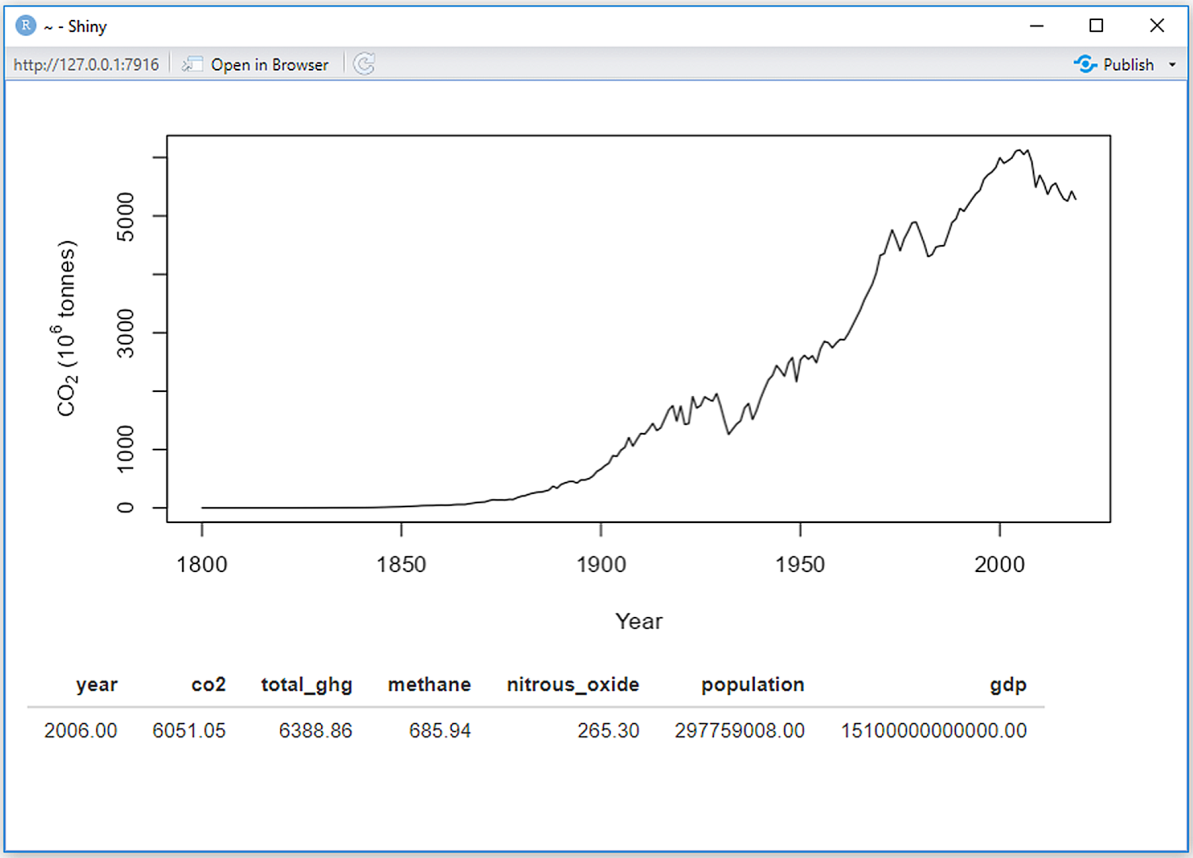


Figure 11.34: Another mouse interactive *shiny* app.

■

11.5.7 Putting Your App Online

11.5.7.1 Using shinyapps.io

A *shiny* app will only work remotely (outside of an **R** session) if a server implementing **R** is used to call the app's code. RStudio helps with this by housing a shiny server site shinyapps.io (Fig 11.35). The site is currently free of charge for a relatively small number of personal applications.

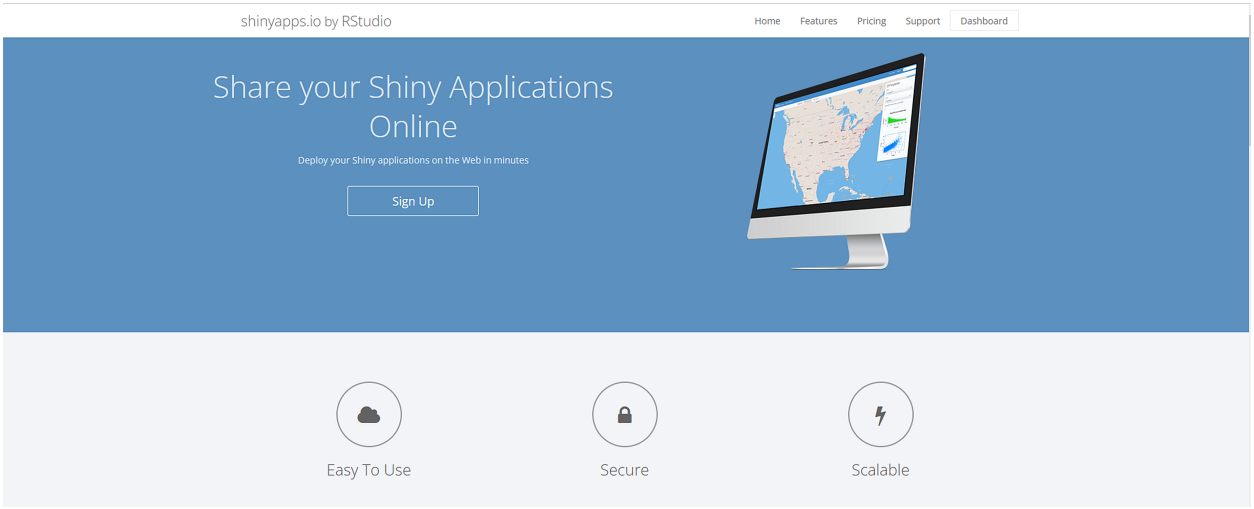


Figure 11.35: The *shiny* apps website https://www.shinyapps.io/.

My personal shinyapps.io account is shown in Fig 11.36.

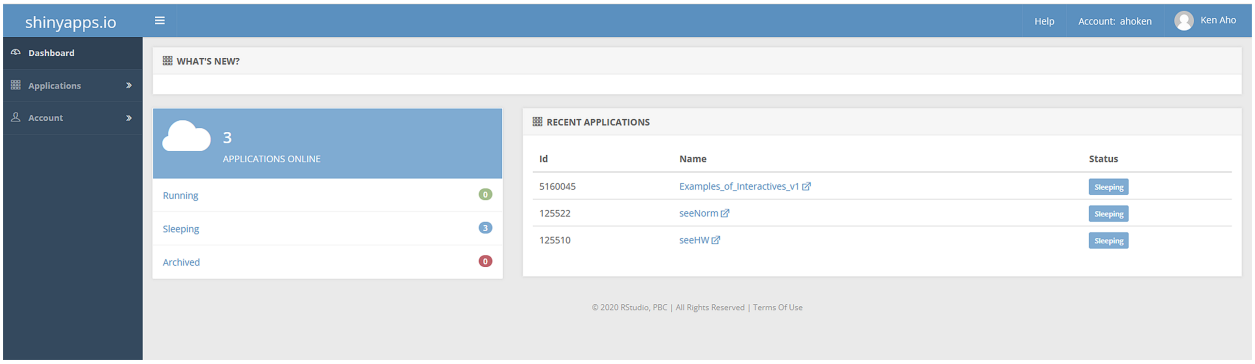


Figure 11.36: My personal *shiny* apps website, with three apps.

The account houses links for some apps summarizing the green house gas data, and the Hardy Weinberg equilibrium.

11.5.7.2 Using Your own Server

It is also possible to run *shiny* apps from a server you control[11] This will generally be driven at the Linux command line using website hosting software like nginx[12], although file transfers using *Secure SHell* (SSH) or *File Transfer Protocol* (FTP) can be managed using freeware like https://filezilla-project.org/. Ultimately you will want to use the Shiny Server server program to run your apps.

[11] For instance using Amazon Web Services account.
[12] The website has a nice tutorial about using nginx with Shiny Server.

Example 11.18.
Consider a recent consulting project involving models for population dynamics for greater sage grouse, *Centrocercus urophasianus*, in south central Idaho (Fig 11.37). I felt that **R** analyses, which would be assessed by a large number of individuals with a mixture of backgrounds, would be best presented as an interactive webpage. I established a *Domain Name System* (DNS) with AWS, and installed **R**, Shiny Server, and necessary **R** packages into the server.

Figure 11.37: A male greater sage grouse (extitCentrocercus urophasianus).

Below I gain access to the site directory framework:

```
1 > ssh ahoken@kenaho.aws.cose.isu.edu
2 ahoken@kenaho.aws.cose.isu.edu's password:
3
4 Welcome to Ubuntu 24.04.2 LTS (GNU/Linux 6.14.0-1010-aws x86_64)
5
6 ahoken@ip-172-31-35-117:/$
```

- On Line 1, I SSH into the AWS site from PowerShell, and am prompted for a password (Line 2).
- On Line 4, I am told that the server is run in Linux/Ubuntu and that I am now in Linux.
- On Line 6, I am delivered to the BASH command line prompt, $

Here is the root directory of the site.

```
ahoken@ip-172-31-35-117:/$ ls
```

```
bin   boot  etc   lib   lib64 media  opt   sbin  snap  sys  usr
dev   home  mnt   proc  run   srv    tmp   var
```

R was downloaded to `etc`. The `etc` directory also contains executables for the server programs nginx and shiny-server. My personal component of the root directory is `home/ahoken/`. It is straightforward to transfer files to and from here. The `srv` directory only contains the

`shiny-server` directory. Looking within `shiny-server` I have placed an `apps` folder, and the file `index.html`, which provides links to other sites I control:

```
ahoken@ip-172-31-35-117:/srv/shiny-server$ ls
```

```
apps  index.html
```

The `apps` directory contains files and subdirectories relevant to the grouse project, including an .Rmd file, `Grouse_app.Rmd`, that codifies the grouse *shiny* app analyses.

```
ahoken@ip-172-31-35-117:/srv/shiny-server/apps$ ls
```

```
Grouse_app.Rmd  book.bib  figs  grouse_shapefiles  table.csv
```

The *shiny* app itself is realized here.

■

11.6 Qt

11.6.1 C++ Calls to R (RInside)

The package *RInside* (Eddelbuettel et al., 2023b) has been created specifically for the purpose of allowing C++ executables to call **R**. For implementing *RInside*, I strongly recommend use of BASH shells, which allow straightforward implementation of make and makefile frameworks (see below). Windows users can install the Windows Subsystem for Linux (WSL) to run Linux-BASH commands (Section 9.1.5).

```
library(Rcpp)
library(RInside)
```

RInside examples make use of make/makefile frameworks. *Makefiles* control the build process of a `make` program, which, in turn, can be used to manage compilation of source code files. Makefiles use their own declarative programming language to define paths to required header files (Section 9.3.1.5) and directories, along with other tasks. The `make` utility can be used to locate makefiles, specify dependencies among multiple source files, streamline source file handling (by only recompiling files that need to be updated), and manage programmatic errors. Some additional details are provided here. *RInside* makefiles are generally used to locate required header files, the **R** home directory, and the *Rcpp* and *RInside* packages.

Example 11.19.
Here is a simple C++ script located in **RInside/examples/standard/** that calls **R** to print `"Hello, world!"`.

```
1 #include <RInside.h>       // for the embedded R via RInside
2
3 int main(int argc, char *argv[]) {
4     RInside R(argc, argv);         // create an embedded R instance
5     R["txt"] = "Hello, world!\n";   // assign a char* (string) to 'txt'
6     R.parseEvalQ("cat(txt)");     // eval init string, ignoring any returns
7     exit(0);
8 }
```

- On Line 1 the *RInside* header file `RInside.h` is called.
- On Line 3 the `main` function, which serves as the entry point for C++ program execution, is initiated with integer output.
- On Line 4, an C++ object called `R` of class `RInside` is instantiated that has two optional arguments.
- On Line 5, a variable `"txt"` is defined in `R` and is assigned the character string `"Hello, world!"`. This step occurs inside an **R** session.
- On Line 6, the **R** function `cat()` is applied to the character string. Output from this operation is "quietly" evaluated with the *RInside* C++ member function `parseEvalQ`. Specifically, `parseEvalQ` executes **R** code formatted as a string, and returns the result to C++.
- On Line 7, an error code of `0` is returned to indicate successful completion of the function.

To compile and run this C++ function (with embedded **R** code), I open the Ubuntu version of Linux implemented in WLS, and navigate to the **examples/standard** directory in *RInside* at the BASH command line.

```
> wsl.exe -d Ubuntu
$ cd /mnt/c/Users/ahoken/AppData/Local/R/win-library/4.3/RInside/examples/standard
```

Running `ls` reveals that the current directory contains 19 C++ source files (the script for the `"Hello, world!"` app is called `rinside_sample0.cpp`), a directory named **cmake**, and two makefiles (one for Windows). The makefiles query **R**, *Rcpp*, and *RInside* for necessary header and library resources, and use this information to complete compilation. This is accomplished using:

```
$ make rinside_sample0
```

I get several lines of compiler output that indicate, among other things, that the executable `rinside_sample0` has been created within the current directory. I run the executable by typing:

```
$ ./rinside_sample0
```

And get

```
Hello, world!
```

■

11.6.2 Qt apps

Qt (pronounced 'cute') is a cross-platform, toolkit for creating GUIs. Among numerous notable examples, GUIs for Google Earth, Mathematica, and the RStudio Desktop GUI all rely on Qt. Qt is currently accessible under both commercial and open-source GPL and LGPL licenses. The proprietary program Qt Creator® is free to students and teachers, and greatly facilitates the creation, debugging, and management of Qt apps. Qt scripts generally rely on a C++ framework, although Qt bindings exist for Python, JavaScript, and C#, among other languages. Qt apps for **R** can look great, but can also involve a lot of coding and the management of multiple files (although this process can be aided by Qt Creator® and other Qt IDEs). As with Tcl/Tk, better support (in the form of applications and online help) exists for building Qt GUIs for Python, than for building Qt GUIs that call **R**.

Example 11.20.
The *RInside* package contains an example for the creation of a Qt/C++ GUI executable that can call **R**. The GUI is based on a seminal *tcltk* GUI for demonstrating kernel density estimation under the **R** function `density()` (see Aho (2014)).

Navigating to **...RInside/examples/Qt** in BASH or Windows shells reveals the presence of two C++ source files (`main.cpp` and `qtdensity.cpp`), a C++ header file (`qtdensity.h`), a Qt project file (`qtdensity.pro`), along with two other other extraneous components: a `README` file, and directory called `cmake`.

```
$ ls
README  cmake  main.cpp  qtdensity.cpp  qtdensity.h  qtdensity.pro
```

Opening `qtdensity.pro` in Qt Creator® triggers a series of screens for defining the configuration of a new Qt project. This includes specification for compilers and debuggers (by default MinGW_64 applications in Windows). This configuration process leads to a project interface containing the necessary GUI-generating files 11.38 in a tidy format.

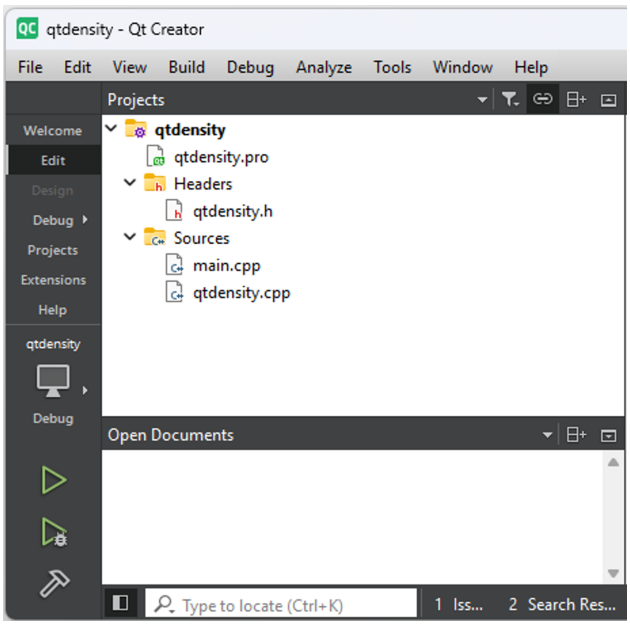


Figure 11.38: Qt Creator® project screen for the *RInside* app `qtdensity`.

Pressing the Hammer widget 🔨 (lower right corner of Fig 11.38) builds the GUI executable (Fig 11.39) inside of a new **Build** subdirectory directory within the **...RInside/examples/Qt** project directory.

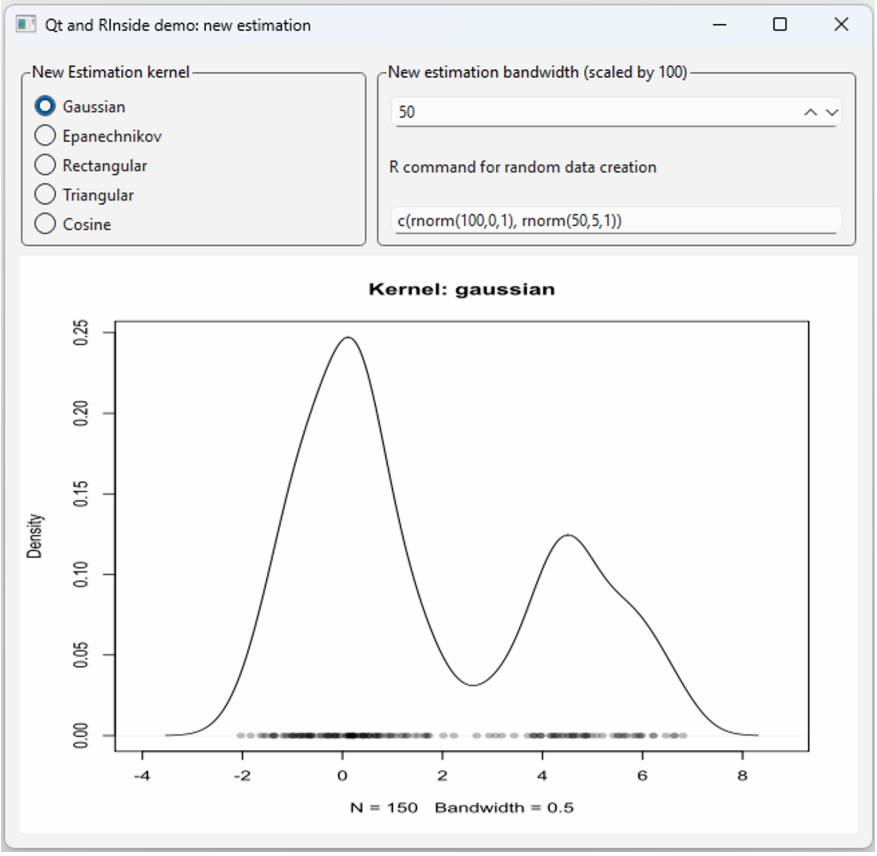


Figure 11.39: A slight modified (to decrease fragility) version of the extitRInside extttqtdensity app.

■

Example 11.21.
The Qt example in *RInside* is relatively complex. Thus, this example creates a much simpler, hand-made "Hello, world!", GUI executable that calls **R**.

To facilitate this process I use the IDEs and other tools in Qt Creator®. 1) I first create a

■

11.7 Comparison of GUI-generating Approaches

This chapter detailed four GUI approaches underlain by **R** or with explicit **R** interactivity. The package *tcltk* uses the Tcl/Tk GUI building tools alongside the native windowing capacities of Windows, Unix-like, and Mac operating systems. The *plotly* and *shiny* libraries render GUIs and interactive plots under an HTML framework. One can also build *Qt* GUIs, underlain by C++-driven executatables that call **R**. A comparative summary of the four approaches introduced in this chapter, is given in Table 11.4.

Table 11.4: Comparison of the three approaches for GUI generation in **R** introduced in this chapter.

	Mechanics	Strengths	Weaknesses
tcltk	Package provides binding for Tcl/Tk GUI building tools.	1) Direct interfacing with **R** 2) Excellent GUI reactivity 3) Wide range of widgets	1) Limited to **R** environment 2) GUIs may have poor aesthetics 3) Awkward coding frameworks 4) Poor support online or otherwise
plotly	Package provides language interfacing from **R** to JSON to HTML.	1) After generation, does not require **R** 2) Some built-in ggplot compatibility	1) GUI capabilities limited to plot interactives
shiny	Language interfacing from **R** to JSON to HTML. Maintains connection to **R** environment	1) Good support online and otherwise. 2) High level of RStudio compatibility. 3) Potentially aesthetic GUIs. 4) Straightforward coding 5) Wide range of widgets	1) Requires direct connection to an **R** session, or server connection to an **R** environment 2) Potentially poor reactivity
Qt	Qt GUIs call **R** via C++ executable.	1) Qt Creator faclitates GUI builds. 2) Potentially aesthetic GUIs. 3) Wide range of widgets	1) Requires direct connection to **R**. 2) Potential GUI fragility 3) Complex coding framework.

Exercises

1. Make a *tcltk* GUI that solves and reports the solutions to differential equations.
2. Make a `plotly` graph of any `gglot2` graph using `ggplotly`.
3. Make a `shiny` app to greet someone. Hint: place the two code chunks below in the `ui` and the `server` function, respectively.

```
textInput("name", "What's your name?")
```

```
output$greeting <- renderText({paste0("Hello ", input$name)})
```

   Make the app interactive inside an **R** Markdown rendered document.
   Along with the code, include a snapshot of the app in action.

4. Your friend has designed an app that solves the exponential growth function for a population with an initial population size of 10, and an intrinsic growth rate of 2, for times, t, from 1 to 50:

$$f(t) = 10 \times \exp(2 \times t).$$

```
ui <- fluidPage(
  sliderInput("t", label = "If t is", min = 1, max = 50, value = 30),
  "then the population size is",
  textOutput("exp.growth")
)
server <- function(input, output, session) {
  output$exp.growth <- renderText({10 * exp(2 * t)})
}
shinyApp(ui, server)
```

   Does the function generate an error? Why?
   Fix the code and provide a snapshot of the app in action.

Chapter 12

R and Your Computer

"Those who can imagine anything, can create the impossible."

- **Alan Turing**, *(1912–1954)*

12.1 How Do Computers Work?

To better understand **R**, we need to understand the underlying constraints of computer systems we use to run **R**. Computers accept data, process data, produce output, and store processed results. This is generally accomplished through through the generation, integration and storage of electrical signals at microscopic scales. A list of (current but often changing) computer hardware terms are given below.

- *Power supply*: Converts alternating current (AC) electric power to low-voltage direct current (DC) power.
- *Motherboard*: A circuit board connecting computer components including the CPU, RAM and memory disk drives.
- *Central Processing Unit* (*CPU*): A microprocessor that performs most of the calculations that allow a computer to function. Specifically, the CPU processes program instructions and sends the results on for further processing and execution by other computer components.
- *Graphics Processing Unit* (*GPU*): An electronic circuit originally designed to accelerate computer graphics, but now widely applied for non-graphic, but highly parallel, calculations.
- *Chipset*: Mediates communication between the CPU and the other computer components.
- *Random Access Memory* (*RAM*): Stores code and data in *primary memory* to allow it to be directly accessed by the CPU. RAM is volatile memory which requires power to retain stored information. Thus, when power is interrupted, RAM data can be lost. RAM types include *dynamic random access memory* (*DRAM*) and *static random-access memory*

(*SRAM*). DRAM constitutes modern computer *main memory* and *graphics cards*. DRAM typically takes the form of an *integrated circuit* chip that can consist of up to billions of memory cells, with each cell consisting of a pairing of a tiny capacitor[1] and transistor[2], allowing each cell to store or read or write one bit of information (Fig 12.1). SRAM uses latching circuitry that holds data permanently in the presence of power, whereas DRAM decays in seconds and must be periodically refreshed. Memory access via SRAM is much faster than DRAM, although DRAM circuits are much less expensive to construct.

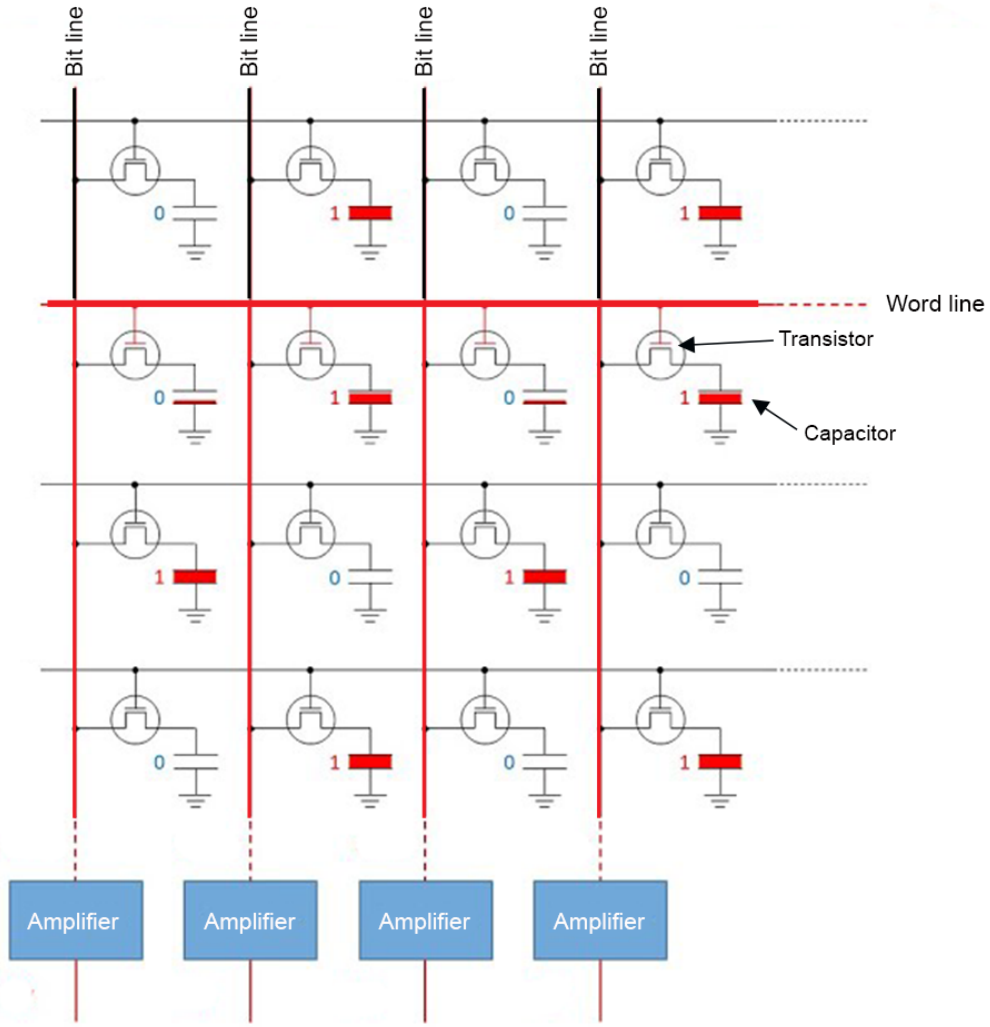


Figure 12.1: Sixteen DRAM memory cells each representing a bit of information for computational storage, reading, or writing. To read the binary word line `0101...` in row two of the circuit, binary signals are sent down the bit lines to sense amplifiers.

[1] A capacitor stores electrical energy by "accumulating electric charges on two closely spaced surfaces that are insulated from each other" (Wikipedia, 2024c).

[2] A transistor is a semiconductor device (a material with an intermediate electrical conductivity, e.g., silicon) that is used to amplify or switch electrical signals and power" (Wikipedia, 2024l).

- *Disk drives*: including CD, DVD, *hard disk* (*HDD*), and *solid state disk* (*SSD*) are used for secondary memory. That is, memory that is not directly accessible from the CPU. Secondary memory can be accessed or retrieved even if the computer is off. Secondary memory is also non-volatile and thus can be used to store data and programs for extended periods. User files and software (like **R**) are generally stored on HDDs or SSDs. *Flash memory*, which uses modified *metal–oxide–semiconductor field-effect transistors* (*MOSFETs*), is typically used on USB and SSD devices to provide secondary memory that can be erased and reprogrammed. Flash memory can also be used in RAM applications.

- *Read-Only Memory* (*ROM*): Stores the BIOS (see below) that runs when the computer is powered on (cold boot) or restarted (warm boot or *reboot*). ROM constitutes primary memory.

- *Basic Input Output System* (*BIOS*): Basic boot (startup) and power management firmware (software that provides low level control for computer hardware). Newer motherboards use the so-called *Unified Extensible Firmware Interface* (*UEFI*) to address BIOS limitations, including restrictive 16 bit addresses.

- *Video card*: Processes computer graphics.

12.2 Base-2 and Base-10

To understand computer processes, it is important to distinguish base-2 (*binary*) and base-10 (*decimal*) numerical systems. In both cases, the *base* refers to the number of unique digits. Thus, base-2 systems can have two unique digits, commonly 0 and 1, and the base-10 system has 10 unique digits: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$. The latter –more widely used system– probably arose because we have ten fingers for counting[3]. A *radix* (commonly a decimal symbol) is used to distinguish the *integer part* of a number from its *fractional part* (Fig 12.2). The radix convention is used by both base-2 and base-10 systems. For example, the decimal number $4\frac{3}{4}$, has integer component 4 and fractional component $\frac{3}{4}$, can be expressed as 4.75. The binary equivalent of $4\frac{3}{4}$ is 100.110.

Traditionally, a base-10 number could only be expressed as a rational fraction whose denominator was a power of ten (Fig 12.2). However, the decimal system can be extended to any real number, by allowing a conceptual infinite sequence of digits following the radix (Wikipedia, 2024e).

[3] A base-20 system used by Pre-Columbian Mesoamerican cultures probably arose because we have twenty fingers and toes (Wikipedia, 2024e).

Millions	Hundred thousands	Ten thousands	Thousands	Hundreds	Tens	Ones	Tenths	Hundredths	Thousands	Ten Thousandths	Hundred Thousandths	Millionths
1,000,000	100,000	10,000	1,000	100	10	1	$\frac{1}{10}$	$\frac{1}{100}$	$\frac{1}{1,000}$	$\frac{1}{10,000}$	$\frac{1}{100,000}$	$\frac{1}{1,000,000}$
10^6	10^5	10^4	10^3	10^2	10^1	10^0	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}

Figure 12.2: A decimal place value chart. A radix (decimal) is placed between the ones and thenths columns to distinguish decimal number components greater than one (to the left), and components less than one but greater than zero (to the right).

12.3 Bits and Bytes

Computers are designed around bits and bytes. A *bit* is a binary (base-2) unit of digital information. Specifically, a bit will represent a `0` or a `1`. This convention occurs because computer systems typically use electronic circuits that exist in only one of two states, on or off. For instance, DRAM memory cells (Fig 12.1) convert electrical low and high voltages into binary `0` and `1` responses, respectively. These signals allow the reading, writing, and storage of data. Although bits are used by all software in all conventional computer operating systems, these mechanisms are easily revealed in **R**[4].

For historical reasons, bits are generally counted in units of bytes. A *byte* equals eight bits. Two major systems exist for counting bytes. The *decimal method*, the most common system, uses powers of 10, allowing implementation of SI prefixes (i.e., kilo = $10^3 = 1000$, mega = $10^6 = 1000^2$, giga = $10^9 = 1000^3$, etc.) (Table 12.1). A computer hard drive with 1 gigabyte (1 billion bytes) of memory will have 1×10^9 bytes = 8×10^9 bits of memory. The *binary system*, used frequently by Windows to describe RAM, defines byte units in multiples of $10^{12} = 1024$.

With a single bit we can describe only $2^1 = 2$ distinct digital objects. These will be an entity represented by a 0, and an entity represented by a 1. It follows that $2^2 = 4$ distinct objects can be described with two bits, $2^3 = 8$ entities can be described with three bits, and so on[5].

[4] Non-binary operating systems are rarely implemented because: 1) they are less efficient, and 2) currently no IEEE standards have been specified. In order of increasing precision and decreasing efficiency, alternative systems include: Limited-Precision Decimal, Arbitrary-Precision Decimal, and Symbolic Calculation systems.

[5] For instance, images often contain eight bit (one byte) variables describing the colors red, green, and blue. Thus, the color red would be a number between 0 and 255 (i.e., red could have $2^8 = 256$ distinct values). Given that the colors blue and green were also eight bit, there would be $256^3 = 16,777,216$ color possibilities (combinations) for any pixel in an image.

Table 12.1: Frequently used byte units.

Decimal		Binary	
Bytes	Name	Bytes	Name (IEC)
1000	kB (kilobyte)	1024	KiB (kibibyte)
1000^2	MB (megabyte)	1024^2	MiB (mebibyte)
1000^3	GB (gigabyte)	1024^3	GiB (gibibyte)
1000^4	TB (terabyte)	1024^4	TiB (tebibyte)
1000^5	PB (petabyte)	1024^5	PeB (pebibyte)

12.4 Decimal to Binary

We count to ten in binary using: `0` = 0, `1` = 1, `10` = 2, `11` = 3, `100` = 4, `101` = 5, `110` = 6, `111` = 7, `1000` = 8, `1001` = 9, and `1010` = 10. Thus, we require four bits to count to ten. Note that the binary sequences for all positive integers greater than or equal to one, start with one.

12.4.1 Positive Integers

We can obtain the binary expression of the integer part of any decimal number by iteratively performing *integer division* by two, and cataloging each *modulus*. The iterations are stopped when a quotient of one is reached. The modulus sequence is read from right to left (backwards). If the whole number of interest is greater than one (i.e., the whole number is not 0 or 1) we place a one in front of the reversed sequence, because all binary sequences for numbers greater than or equal to one must start with one.

Example 12.1.
Consider the number 23:

Modulus (remainder)		1	1	1	0
Integer Quotient	$23/2 = 11$	$11/2 = 5$	$5/2 = 2$	$2/2 = 1$

The reversed sequence is `0111`. We place a one in front to get the binary representation for 23: `10111`. The function `dec2bin()` from *asbio* does the work for us:

```
library(asbio)
dec2bin(23)
```

```
[1] 10111
```

■

12.4.2 Positive Fractions

The fractional part of a decimal number can be converted to binary in a similar fashion.

- To identify the fractional expression as a non integer, start the binary sequence with `0.` (a zero followed by a decimal symbol).
- Double the fraction to be converted, and record a `1` if the product is ≥ 1, and `0` otherwise.
- For subsequent binary digits, multiply two by the fractional part of the previous multiplication. If the product is ≥ 1, record a `1`. If not, record a `0`.

Example 12.2.
Consider the fraction $\frac{1}{4}$. We have:

Binary outcome	0	1
Product	$1/4 \times 2 = 1/2 < 1$	$1/2 \times 2 = 1 \geq 1$
Binary outcome	0	0
Product	$0 \times 2 = 0 < 1$	$0 \times 2 = 0 < 1$

■

We have a clear repeating sequence of zeroes, due to a product of two in the second step. This allows us to stop the growth of the binary expression. For fractions, the binary sequence is read conventionally, from left to right. Thus, the binary expression for $\frac{1}{4}$ is `0.01`.

```
dec2bin(0.25)
```

```
[1] 0.01
```

12.5 Binary to Decimal

The addition of a binary digit (i.e., a bit) represents a doubling of information storage. For instance, as we increase from two bits to three bits, the number of describable integers increases from four (integers 0 to 3) to eight (integers 0 to 7). As a result we say that the rightmost digit in a set of binary digits represents 2^0, the next represents 2^1, then 2^2, and so on. This can be defined with an equation based on Horner's method (Horner, 1815) that allows conversion of binary to decimal numbers:

$$\sum_{\kappa=\min(\kappa)}^{\max(\kappa)} \alpha\beta^{\kappa} \tag{12.1}$$

where α is a quantity known as the *significand*, that contains bit (0, 1) outcomes. For the purpose of binary expressions, the modifying base, β, is 2. The term κ is called (appropriately) *the exponent*.

The maximum and minimum values of κ are determined by counting the number of placeholder digits in the binary expression represented by the significand, with respect to a binary radix point (Fig 12.3). Note that counting starts with respect to 0 (the first digit to the left of the radix) for both positive (bits to the left of the radix) and negative (bits to the right of the radix) values of the exponent, κ. The radix reference has prompted this method to be called *floating point arithmetic*.

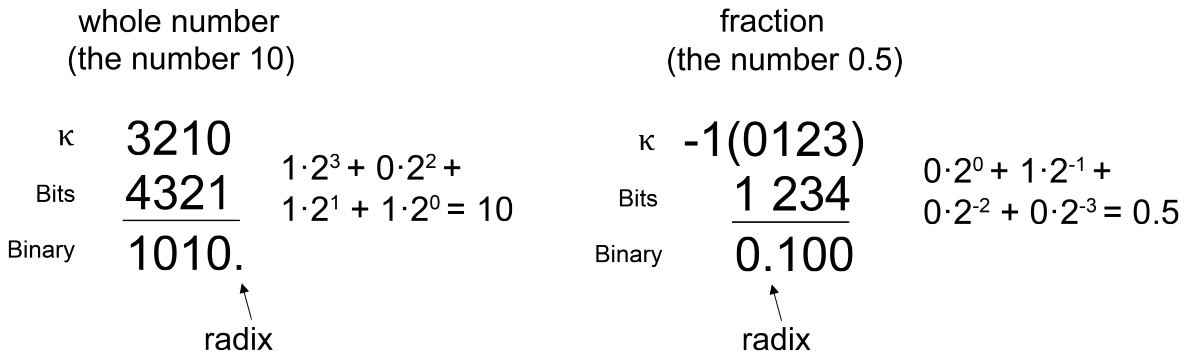


Figure 12.3: Conceptualization of binary to decimal conversion, as given in Eq 12.1.

12.5.1 Positive Integers

For positive integers the entirety of the corresponding binomial expression will be to the left of the radix point (Fig 12.3). Thus, the minimum value of κ will be zero and the maximum value of κ will be the number of digits (bits) in the binary expression, minus one.

Equation (12.1) represents a *dot product*. That is, the equation is a sum of the element-wise multiplication of two vectors. For instance, to find the integers represented by a single binary bit, we multiply the binary digit value, `0` or `1`, by the power of two it represents. Because the single bit signature would occur at the right-most address to the left of the radix, the value of exponent would be 0 (Fig 12.3). That is, $\min(\kappa) = \max(\kappa) = 0$ in Eq. (12.1).

If the single bit equals `0` we have:

$$0 \times 2^0 = 0,$$

and if the single bit equals `1` we have:

$$1 \times 2^0 = 1.$$

Accordingly, to find the decimal version of a set of binary values, we take the sum of the products of the binary digits and their corresponding (decreasing) powers of base 2.

Example 12.3.
For example, the binary number `010101` equals:

$$\begin{aligned}&(0 \times 2^5) + (1 \times 2^4) + \\ &(0 \times 2^3) + (1 \times 2^2) + \\ &(0 \times 2^1) + (1 \times 2^0) = \\ &0 + 16 + 0 + 4 + 0 + 1 = 21.\end{aligned}$$

The function `bin2dec` in *asbio* does the calculation for us.

```
bin2dec(010101)
```

```
[1] 21
```

■

12.5.2 Positive Fractions

For positive fractions, values of the κ exponent will decrease by minus one as bits increase by one (Fig 12.3). Thus, to obtain decimal fractions from binary fractions we multiply a bit's binary value by decreasing negative powers of base two, starting at 0, and find the sum, as shown in Eq (12.1).

Example 12.4.
For example, the binary value `0.01` equals:

$$(0 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) = 0.25$$

```
bin2dec(0.01)
```

```
[1] 0.25
```

■

12.5.2.1 Terminality

Most decimal fractions will not have a clear *terminal* binary sequence. That is, a binary representation of a decimal fraction with a finite number of digits will not exist. This results in mere binary approximations of decimal numbers (Goldberg, 1991). For instance, the 10 bit binary expression of $\frac{1}{10}$ is

```
dec2bin(0.1)
```

```
[1] 0.0001100110
```

But translating this back to decimal we find:

```
bin2dec(0.0001100110)
```

```
[1] 0.0996
```

We can increase the number of bits in the binary expression,

```
dec2bin(0.1, max.bits = 14)
```

```
[1] 0.00011001100110
```

This increases precision, but the decimal approximation remains imperfect.

```
options(digits = 20)
bin2dec(0.00011001100110)
```

```
[1] 0.10000000000000000555
```

Note that these imperfect conversions are the actual results of the division $\frac{1}{10}$ for all software on all current conventional computers (not just **R**)!

```
options(digits = 20)
1/10
```

```
[1] 0.10000000000000000555
```

It may seem surprising that rational fractions like $\frac{1}{10}$ may have non-terminating binary expressions. *Terminality*, however, will only occur for a decimal fraction if a product of 2 results from the successive multiplication steps described in Section 12.4.2. This product does not occur for $\frac{1}{10}$.

Lack of terminality for binary expressions prompts the need for quantifying imprecision in computers systems. This can be obtained from Eq (12.1). In particular, the exponent in Eq (12.1) determines minimum and maximum possible encoded numeric values, and the number of digits in the significand determines numeric precision. Indeed, by changing the base from 2 to 10, Eq (12.1) can be used to quantify the precision of binary and decimal numbers.

Example 12.5.
The decimal number $1,245.42$ has the *scientific notation*: 1.24542×10^3. The expression has a the precision of six digits, because under Eq (12.1) the significand has six digits. Note that applying these digits in Eq. (12.1) we have:

$$\begin{aligned} &1 \times 10^3 + 2 \times 10^2 + 4 \times 10^1 + 5 \times 10^0 + 4 \times 10^{-1} + 2 \times 10^{-2} = \\ &1000 \quad + 200 \quad + 40 \quad + 5 \quad + 0.4 \quad + 0.02 \quad = 1245.42 \end{aligned}$$

■

12.6 Double Precision

In most programs, on most workstations, the results of computations are stored as 32 bits (i.e., 4 bytes) or as 64 bits (8 bytes) of information. The 64 bit double precision format allows high precision representations of both positive and negative integers and their fractional components. Under this framework, one bit is allocated to the sign of the stored item, 53 bits are assigned to the significand, and 11 bits are given to the exponent (Fig 12.4).

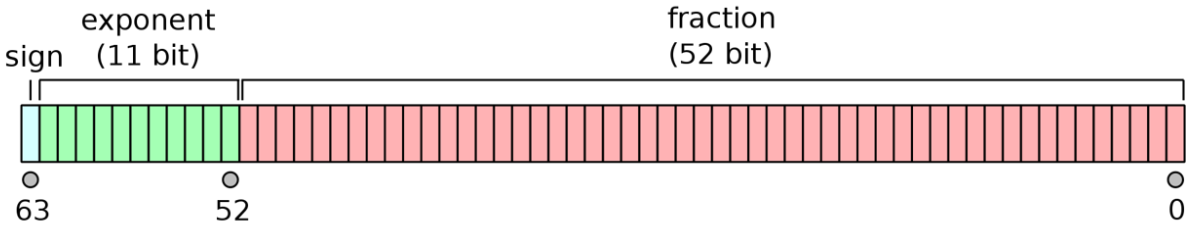


Figure 12.4: The IEEE 754 double-precision binary floating-point format Figure taken from https://commons.wikimedia.org/w/index.php?curid=3595583.

This can be represented mathematically as a more complex form of Eq (12.1):

$$(-1)^{\text{sign}} \left(1 + \sum_{i=1}^{52} b_{52-i} 2^{-i}\right) \times 2^{e-1023} \tag{12.2}$$

which gives the assumed numeric value for a 64-bit double-precision datum with *exponent bias*.

Example 12.6.
The function `bit64()` below is taken from the Examples of the documentation for the *base* function `numToBits()`, which converts digital numbers to 64 bits. The function distinguishes:

- The single bit giving the sign of the number (`0` = positive, `1` = negative).
- The 11 bit exponent.
- A 52 bit significand (without the implicit leading `1`).

```
bit64 <- function(x)
  noquote(vapply(as.double(x),
                 function(x) {
                   b <- substr(as.character(rev(numToBits(x))), 2L, 2L)
                   paste0(c(b[1L], " ", b[2:12], " | ", b[13:64]),
                          collapse = "")}, "")
  )
```

Here is the double precision representation of $\frac{1}{3}$

```
bit64(1/3)
```

```
[1] 0 01111111101 | 0101010101010101010101010101010101010101010101010101
```

We see this follows the form of Eq (12.2). The exponent 01111111101 represents the decimal number 1021:

```
bin2dec(01111111101)
```

```
[1] 1021
```

And one plus the dot product of the significand and base-2 raised to the sequence -1 to -52, multiplied by $2^{1021-1023}$, is:

```
sigd <- strsplit("0101010101010101010101010101010101010101010101010101", NULL)
sigd <- as.numeric(unlist(sigd))
base2 <- 2^(-1:-52)
(1 + sum(sigd * base2)) * 2^-2
```

```
[1] 0.3333
```

That is, we have:

$$\begin{aligned} value &= (-1)^{\text{sign}} \left(1 + \sum_{i=1}^{52} b_{52-i} 2^{-i}\right) \times 2^{e-1023} \\ &= -1^0 \times (1 + 2^{-2} + 2^{-4} + \cdots + 2^{-52}) \times 2^{1021-1023} \\ &\approx 1.33\bar{3} \times 2^{-2} \\ &\approx \frac{1}{3} \end{aligned}$$

■

The 11 bit width of the double precision exponent allows the expression of numbers between 10^{-308} and 10^{308}, with full 15–17 decimal digits precision. This is clearly demonstrated in **R**. Specifically, imprecision problems with non-terminal fractions become evident for decimal numbers with greater than 16 displayed digital digits.

```
options(digits = 18)
1/3
```

```
[1] 0.333333333333333315
```

Additionally, the current upper numerical limit in **R** (ver 4.3.2) is somewhere between:

```
1.8 * 10^307
```

```
[1] 1.8e+307
```

and

```
1.8 * 10^308
```

```
[1] Inf
```

The so-called *subnormal representation*[6] compromises precision, but allows allows fractional representations approaching 5×10^{-324}. This approach is used by **R**, whose smallest represented fraction is between:

```
5.0 * 10^-323
```

```
[1] 4.9406564584124654e-323
```

and

```
5.0 * 10^-324
```

```
[1] 0
```

Binary fractional numbers are expressed with respect to a decimal, and the number of digits will (often) be dictated by the significand. Given 13 bits we have the following binary translations to decimal numbers: `1` = 1/1, `0.1` = 1/2, `0.01010101...` = 1/3, `0.01` = 1/4, `0.00110011` = 1/5, `0.0010101...` = 1/6, `0.001001...` = 1/7, `0.001` = 1/8, `0.000111000111...` = 1/9, `0.000110011...` = 1/10.

12.7 Binary Characters

Characters can also be expressed in binary. The American Standard Code for Information Interchange (ASCII) consists of 128 characters, and requires one byte = eight bits[7]. The newer eight bit Unicode Transformation Format (UTF-8) system –the one used by **R**– can represent 1,112,064 valid code points, using between 1 to 4 bytes = 8 to 32 bits (Wikipedia, 2024n). Specifically, from the perspective of the UTF-16 system, the UTF-8 system uses portions of seventeen *planes* [8], each consisting of sixteen bits (and, thus, $2^{16} = 65,536$ code variants). This results in the quantity:

$$(17 \times 2^{16}) - 2^{11} = 1,112,064$$

The $2^{11} = 2048$ subtraction acknowledges that there are 2048 technically-invalid Unicode

[6] *Subnormal numbers* fill the underflow gap around zero in floating-point arithmetic (Wikipedia, 2024k). For subnormal numbers, e in Eq:(12.2) is taken to be zero. *Underflow* occurs when the result of a calculation is a number with greater precision than the computer can actually represent in its CPU memory (Wikipedia, 2024a).

[7] Originally developed from telegraph code, ASCII has only 128 code points, of which only 95 are printable characters (Wikipedia, 2023a).

[8] In Unicode, a plane is a group of $2^{16} = 65,536$ code points. There are 17 planes because UTF-16, can encode 2^{20} code points (16 planes) as pairs of words, plus the so-called Basic Multilingual Plane (UTF-16 plane 0) as a single word.

surrogates (Wikipedia, 2024m). The first 128 UTF-8 characters are the ASCII characters, allowing back-comparability with ASCII.

Example 12.7.
We can observe the process of binary character assignment in **R** using the functions `as.raw()`, `rawToChar()`, and `rawToBits()`. The base type `raw` (Section 2.3.6) is intended to hold raw byte information. Here is a list of the 128 ASCII characters.

```
rawToChar(as.raw(1:128), multiple = TRUE)
```

```
  [1] "\001" "\002" "\003" "\004" "\005" "\006" "\a"   "\b"   "\t"   "\n"
 [11] "\v"   "\f"   "\r"   "\016" "\017" "\020" "\021" "\022" "\023" "\024"
 [21] "\025" "\026" "\027" "\030" "\031" "\032" "\033" "\034" "\035" "\036"
 [31] "\037" " "    "!"    "\""   "#"    "$"    "%"    "&"    "'"    "("
 [41] ")"    "*"    "+"    ","    "-"    "."    "/"    "0"    "1"    "2"
 [51] "3"    "4"    "5"    "6"    "7"    "8"    "9"    ":"    ";"    "<"
 [61] "="    ">"    "?"    "@"    "A"    "B"    "C"    "D"    "E"    "F"
 [71] "G"    "H"    "I"    "J"    "K"    "L"    "M"    "N"    "O"    "P"
 [81] "Q"    "R"    "S"    "T"    "U"    "V"    "W"    "X"    "Y"    "Z"
 [91] "["    "\\"   "]"    "^"    "_"    "`"    "a"    "b"    "c"    "d"
[101] "e"    "f"    "g"    "h"    "i"    "j"    "k"    "l"    "m"    "n"
[111] "o"    "p"    "q"    "r"    "s"    "t"    "u"    "v"    "w"    "x"
[121] "y"    "z"    "{"    "|"    "}"    "~"    "\177" "\x80"
```

Note that the exclamation point is character number 33. Its 16 bit binary code is:

```
rawToBits(as.raw(33))
```

```
[1] 01 00 00 00 00 01 00 00
```

From the output above, codes 1-31 and 127-128 are not printable characters. Thus, there are only 128 - 33 = 95 printable ASCII characters. Note that codes 7-13 are command characters. For instance, character 10, `"\n"` indicates new line.

■

12.8 Optimizing R

Because attention was given to computational efficiency in several earlier sections in this chapter, here I briefly consider several methods for optimizing **R**. In particular, I consider the use of **R**-interfaces, including scripting from command line OS shells to implement high performance computers (HPCs) and parallel computing.

12.8.1 Calling Unix/Linux HPCs

Under construction

12.8.1.1 Bioinformatics Pipelines

12.8.2 Parallel Computing

Under construction

Exercises

1. Define the following terms:
   (a) Motherboard
   (b) Central processing unit (CPU)
   (c) Random access memory
   (d) Primary memory
   (e) Secondary memory
   (f) Volatile memory
   (g) Non-volatile memory

2. How many bits are in 5 gigabytes and 6 gibibytes?

3. What is the level of trustworthy precision (in number of digits) for decimal fractional components in **R** (and all software that use 64 bit double precision)?

4. Obtain the five bit binary sequence for the number 21 by hand. Check your answer using `dec2bin()`.

5. Find the decimal number corresponding to the five bit binary sequence `11111`. Check your answer using `bin2dec()`.

6. Find the 64 bit expression for the decimal number -2 (minus 2) using the function `bit64()`, as shown in this chapter. Back-transform this binary representation to the decimal number by hand using Eq. (12.2). Use **R** functions like `strsplit()` `unlist()`, etc., to help.

Bibliography

Adler, J. (2010). *R in a nutshell: A desktop quick reference*. " O'Reilly Media, Inc.".

Aho, K. (2014). *Foundational and Applied Statistics for Biologists Using R*. CRC Press.

Aho, K. (2023). *asbio: A Collection of Statistical Tools for Biologists*. R package version 1.9-6.

Aho, K., Derryberry, D., Godsey, S. E., Ramos, R., Warix, S. R., and Zipper, S. (2023a). Communication distance and bayesian inference in non-perennial streams. *Water Resources Research*, 59(11):e2023WR034513.

Aho, K., Kriloff, C., Godsey, S. E., Ramos, R., Wheeler, C., You, Y., Warix, S., Derryberry, D., Zipper, S., Hale, R. L., et al. (2023b). Non-perennial stream networks as directed acyclic graphs: The R-package streamdag. *Environmental Modelling & Software*, 167:105775.

Allaire, J., Xie, Y., Dervieux, C., McPherson, J., Luraschi, J., Ushey, K., Atkins, A., Wickham, H., Cheng, J., Chang, W., and Iannone, R. (2024). *rmarkdown: Dynamic Documents for R*. R package version 2.28.

Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., et al. (1999). *LAPACK users' guide*. SIAM.

Backus, J. (1998). The history of Fortran I, ii, and iii. *EEE Annals of the History of Computing*, 20:68–78.

Basin, S. (1963). The fibonacci sequence as it appears in nature. *The Fibonacci Quarterly*, 1(1):53–56.

Bates, D., Mächler, M., Bolker, B., and Walker, S. (2015). Fitting linear mixed-effects models using lme4. *Journal of Statistical Software*, 67(1):1–48.

Bates, D., Maechler, M., and Jagan, M. (2023). *Matrix: Sparse and Dense Matrix Classes and Methods*. R package version 1.5-4.1.

Becker, R. and Chambers, J. (1978). Design and implementation of the 'S' system for interactive data analysis. In *The IEEE Computer Society's Second International Computer Software and Applications Conference, 1978. COMPSAC'78.*, pages 626–629. IEEE.

Becker, R. and Chambers, J. (1981). S: a language and system for data analysis, Bell Laboratories computer information service. *Murray Hill, New Jersey*.

Becker, R., Chambers, J., and Wilks, A. (1988). *The New S language*. CRC Press.

Becker, R. A., Cleveland, W. S., and Shyu, M.-J. (1996). The visual design and control of trellis display. *Journal of Computational and Graphical Statistics*, pages 123–155.

Bell, E. T. (1938). The iterated exponential integers. *Annals of Mathematics*, 39(3):539–557.

Bell Labs (2004). The creation of the UNIX operating system. [Online; accessed 8-July-2024].

Bengtsson, H. (2003). The R.oo package - object-oriented programming with references using standard R code. In Hornik, K., Leisch, F., and Zeileis, A., editors, *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*, Vienna, Austria.

Bengtsson, H. (2022). *R.matlab: Read and Write MAT Files and Call MATLAB from Within R*. R package version 3.7.0.

Bivand, R. S., Pebesma, E., and Gómez-Rubio, V. (2013). *Applied Spatial Data Analysis with R, Second edition*. Springer, NY.

Bonnin, S. (2021). *Intermediate R: introduction to data wrangling with the Tidyverse (2021)*. GitHub bookdown document.

Boutin, P., Hailpern, B., Proebsting, T., and Wiederhold, G. (2002). Mother tongues - tracing the roots of computer languages through the ages. *Wired*.

Breslow, N. E. and Day, N. (1980). *Statistical methods in cancer research. Vol. 1. The analysis of case-control studies.*, volume 1. IARC Publications.

Brinkmann, R. (2009). *Dire Predictions: Understanding Global Warming. The Illustrated Guide to the Findings of the Intergovernment Panel on Climate Change*. JSTOR.

Brousseau, A. (1969). Fibonacci statistics in conifers. *The Fibonacci Quarterly*, 7(5):523–532.

Butcher, J. C. (1987). *The Numerical Analysis of Ordinary Differential Equations: Runge-Kutta and General Linear Methods*. Wiley-Interscience.

Canty, A. and Ripley, B. D. (2022). *boot: Bootstrap R (S-Plus) Functions*. R package version 1.3-28.1.

Chambers, J. M. (1999). Computing with data: Concepts and challenges. *The American Statistician*, 53(1):73–84.

Chambers, J. M. (2008). *Software for data analysis: programming with R*, volume 2. Springer.

Chambers, J. M. (2020). S, R, and Data Science. *The R Journal*, 12(1):462–476.

Chambers, J. M. and Hastie, T. J. (1992). Statistical models. In *Statistical models in S*, pages 13–44. Routledge.

Chang, W. (2025). *R6: Encapsulated Classes with Reference Semantics*. R package version 2.6.1, https://github.com/r-lib/R6.

Cleveland, W. S. (1993). *Visualizing data*. Hobart press.

Cooley, J. W. and Tukey, J. W. (1965). An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301.

Corbató, F. J. and Vyssotsky, V. A. (1965). Introduction and overview of the multics system. In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, pages 185–196.

Crampton, E. et al. (1947). The growth of the odontoblasts of the incisor tooth as a criterion of the vitamin C intake of the guinea pig. *Journal of Nutrition*, 33:491–504.

Crawley, M. J. (2012). *The R Book*. John Wiley & Sons.

Csárdi, G., Nepusz, T., Traag, V., Horvát, S., Zanini, F., Noom, D., and Müller, K. (2025). *igraph: Network Analysis and Visualization in R*. R package version 2.1.4.

Dalgaard, P. (2001). A primer on the R-tcl/tk package. *R News*, 1(3):27–31.

Dalgaard, P. (2002). Changes to the R-tcl/tk package. *R News*, 2(3):25–27.

Douady, S. and Couder, Y. (1996). Phyllotaxis as a dynamical self organizing process part i: the spiral modes resulting from time-periodic iterations. *Journal of theoretical biology*, 178(3):255–273.

Eddelbuettel, D. (2013). *Seamless R and C++ Integration with Rcpp*. Springer, New York. ISBN 978-1-4614-6867-7.

Eddelbuettel, D. and Balamuta, J. J. (2018). Extending R with C++: A Brief Introduction to Rcpp. *The American Statistician*, 72(1):28–36.

Eddelbuettel, D. and François, R. (2023). Exposing C++ functions and classes with Rcpp modules. *Vignette included in R package Rcpp*.

Eddelbuettel, D., Francois, R., Allaire, J., Ushey, K., Kou, Q., Russell, N., Ucar, I., Bates, D., and Chambers, J. (2023a). *Rcpp: Seamless R and C++ Integration*. R package version 1.0.11.

Eddelbuettel, D., Francois, R., and Bachmeier, L. (2023b). *RInside: C++ Classes to Embed R in C++ (and C) Applications*. R package version 0.2.18.

Faraway, J. J. (2004). *Linear Models with R*. Chapman and Hall/CRC.

Faraway, J. J. (2016). *Extending the Linear Model with R: Generalized Linear, Mixed Effects and Nonparametric Regression Models*. CRC press.

Fisher, R. A. and Mackenzie, W. A. (1923). Studies in crop variation. ii. the manurial response of different potato varieties. *The Journal of Agricultural Science*, 13(3):311–320.

Fox, J. (2005). The R Commander: A basic statistics graphical user interface to R. *Journal of Statistical Software*, 14(9):1–42.

Fox, J. (2007). Extending the R commander by "plug-in" packages. *R news*, 7(3):46–52.

Fox, J. (2009). Aspects of the social organization and trajectory of the R project. *R J.*, 1(2):5.

Fox, J., Marquez, M. M., and Bouchet-Valat, M. (2023). *Rcmdr: R Commander*. R package version 2.9-1.

Fox, J. and Weisberg, S. (2019). *An R Companion to Applied Regression*. Sage, Thousand Oaks CA, third edition.

Gagolewski, M. (2022). stringi: Fast and portable character string processing in R. *Journal of Statistical Software*, 103(2):1–59.

Garsiel, L. (2018). Behind the scenes of modern web browsers. [Online; accessed 28-April-2025].

Geyer, C. J. (1991). Constrained maximum likelihood exemplified by isotonic convex logistic regression. *Journal of the American Statistical Association*, pages 717–724.

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM computing surveys (CSUR)*, 23(1):5–48.

Grosjean, P. (2024). *SciViews-R*. UMONS, MONS, Belgium.

Grothendieck, G., Kates, L., and Petzoldt, T. (2016). *proto: Prototype Object-Based Programming*. R package version 1.0.0.

Gupta, S. (2012). How R searches and finds stuff. [Online; accessed 30-May-2025].

Haddock, S. H. D. and Dunn, C. W. (2011). *Practical computing for biologists*. Sinauer Associates Sunderland, MA.

Henry, L. and Wickham, H. (2025). *rlang: Functions for Base Types and Core R and 'Tidyverse' Features*. R package version 1.1.5.

Hershey, A. V. (1967). *Calligraphy for computers*, volume 2101. US Naval Weapons Laboratory.

Hodgkin, A. L. and Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500.

Hoffmann, T. J. and Laird, N. M. (2009). fgui: A method for automatically creating graphical user interfaces for command-line R packages. *Journal of Statistical Software*, 30(2):1–14.

Horner, W. (1815). A new method of solving numerical equations of all orders, by continuous approximation. In *Abstracts of the Papers Printed in the Philosophical Transactions of the Royal Society of London*, volume 2, pages 117–117. JSTOR.

Hornik, K. and the R Core Team (2023). R FAQ.

Hothorn, T., Hornik, K., van de Wiel, M. A., and Zeileis, A. (2006). A Lego system for conditional inference. *The American Statistician*, 60(3):257–263.

Hothorn, T., Hornik, K., van de Wiel, M. A., and Zeileis, A. (2008). Implementing a class of permutation tests: The coin package. *Journal of Statistical Software*, 28(8):1–23.

Ihaka, R. (1998). R: Past and future history. *Computing Science and Statistics*, 392396.

Ihaka, R. and Gentleman, R. (1996). R: a language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314.

James, M. D. A. (2009). Package 'dbi'.

Kassambara, A. (2023). *ggpubr: 'ggplot2' Based Publication Ready Plots*. R package version 0.6.0.

Kernighan, B. W. and Ritchie, D. M. (2002). *The C programming language*. Pearson Education Asia.

Lawrence, M. and Verzani, J. (2018). *Programming graphical user interfaces in R*. Chapman and Hall/CRC.

Leisch, F. (2002). Sweave: Dynamic generation of statistical reports using literate data analysis. In *COMPSTAT: Proceedings in computational statistics*, pages 575–580. Springer.

Lemon, J. (2006). Plotrix: a package in the red light district of R. *R-News*, 6(4):8–12.

Lin, G. (2023). *reactable: Interactive Data Tables for R*. R package version 0.4.4.

Maechler, M., Rousseeuw, P., Struyf, A., Hubert, M., and Hornik, K. (2022). *cluster: Cluster Analysis Basics and Extensions*. R package version 2.1.4 — For new features, see the 'Changelog' file (in the package source).

Magurran, A. E. (1988). *Ecological Diversity and its Measurement*. Princeton University Press.

Matsumoto, M. and Nishimura, T. (1998). Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30.

McCarthy, J. (1978). History of lisp. In *History of Programming Languages*, pages 173–185. Stanford University.

McIntosh, R. P. (1967). An index of diversity and the relation of certain concepts to diversity. *Ecology*, 48(3):392–404.

Meyers, S. (2001). *Effective STL: 50 specific ways to improve your use of the standard template library*. Pearson Education.

Meyers, S. (2005). *Effective C++: 55 specific ways to improve your programs and designs*. Pearson Education.

Morandat, F., Hill, B., Osvald, L., and Vitek, J. (2012). Evaluating the design of the R language: Objects and functions for data analysis. In *ECOOP 2012–Object-Oriented Programming: 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings 26*, pages 104–131. Springer.

Murdoch, D. and Adler, D. (2025). *rgl: 3D Visualization Using OpenGL*. R package version 1.3.17.

Murrell, P. (2019). *R graphics, 3rd edition*. Chapman and Hall/CRC.

Oksanen, J., Simpson, G. L., Blanchet, F. G., Kindt, R., Legendre, P., Minchin, P. R., O'Hara, R., Solymos, P., Stevens, M. H. H., Szoecs, E., Wagner, H., Barbour, M., Bedward, M., Bolker, B., Borcard, D., Carvalho, G., Chirico, M., De Caceres, M., Durand, S., Evangelista, H. B. A., FitzJohn, R., Friendly, M., Furneaux, B., Hannigan, G., Hill, M. O., Lahti, L., McGlinn, D., Ouellette, M.-H., Ribeiro Cunha, E., Smith, T., Stier, A., Ter Braak, C. J., and Weedon, J. (2022). *vegan: Community Ecology Package*. R package version 2.6-4.

Ousterhout, J. K. (1991). An x11 toolkit based on the tcl language. In *USENIX Winter*, pages 105–116. Citeseer.

Pebesma, E. and Bivand, R. S. (2023). *Spatial Data Science With Applications in R*. Chapman & Hall.

Pine, D. J. (2019). *Introduction to Python for science and engineering*. CRC press.

Pinheiro, J. C. and Bates, D. M. (2000). *Mixed-Effects Models in S and S-PLUS*. Springer, New York.

R Core Team (2023). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

R Core Team (2024a). *R internals*.

R Core Team (2024b). *R language definition*.

R Core Team (2024c). *Writing R Extensions*.

R Special Interest Group on Databases (R-SIG-DB), Wickham, H., and Müller, K. (2024). *DBI: R Database Interface*. R package version 1.2.3.

Reilly, E. D. (2003). *Milestones in computer science and information technology*. Greenwood Publishing Group Inc.

Ritchie, D. M. (1984). The unix system: The evolution of the unix time-sharing system. *AT&T Bell Laboratories Technical Journal*, 63(8):1577–1593.

Ritchie, D. M. (1993). The development of the c language. *ACM Sigplan Notices*, 28(3):201–208.

Rubino, M., Etheridge, D., Trudinger, C., Allison, C., Battle, M., Langenfelds, R., Steele, L., Curran, M., Bender, M., White, J., et al. (2013). A revised 1000 year atmospheric δ13c-co2 record from law dome and south pole, antarctica. *Journal of Geophysical Research: Atmospheres*, 118(15):8482–8499.

Ryan, M. S. and Nudd, G. R. (1993). The viterbi algorithm. *Department of Computer Science Research Report*.

Sarkar, D. (2008). *Lattice: Multivariate Data Visualization with R*. Springer, New York.

Satman, M. H. (2014). RCaller: A software library for calling R from Java. *British Journal of Mathematics & Computer Science*, 4(15):2188.

Schanda, J. (2007). Cie colorimetry. *Colorimetry: Understanding the CIE system*, 3:25–78.

Schnute, J. T., Couture-Beil, A., and Haigh, R. (2023). *PBSmodelling: GUI Tools Made Easy: Interact with Models and Explore Data*. R package version 2.69.3.

Schnute, J. T., Couture-Beil, A., Haigh, R., and Kronlund, A. (2013). Pbsmodelling 2.65: user's guide. *Canadian Technical Report of Fisheries and Aquatic Sciences*, 2674:viii–194.

Schwartz, M., Harrell Jr, F., Rossini, A., and Francis, I. (2008). R: Regulatory compliance and validation issues a guidance document for the use of R in regulated clinical trial environments.

The R Foundation for Statistical Computing, c/o Department of Statistics and Mathematics, Wirtschaftsuniversität Wien, Augasse, pages 2–6.

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423.

Sievert, C. (2020). *Interactive Web-based Data Visualization with R, plotly, and shiny*. CRC Press.

Signorell, A. (2023). *DescTools: Tools for Descriptive Statistics*. R package version 0.99.52.

Sima, A. C., Stockinger, K., de Farias, T. M., and Gil, M. (2019). Semantic integration and enrichment of heterogeneous biological databases. *Evolutionary genomics: statistical and computational methods*, pages 655–690.

Sklyar, O., Eddelbuettel, D., and Francois, R. (2025). *inline: Functions to Inline C, C++, Fortran Function Calls from R*. R package version 0.3.21.

Steele, Guy Lewis, J. (1978). Rabbit: A compiler for scheme. Master's thesis, Massachusetts Institute of Technology.

Sussman, G. J. and Steele Jr, G. L. (1998). Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439.

Thieme, N. (2018). R generation. *Significance*, 15(4):14–19.

Thompson, K. (1972). *Users' Reference to B*. [Online; accessed 7-Sep-2023].

Tierney, L. (2023). *codetools: Code Analysis Tools for R*. R package version 0.2-19.

Tsuda, M. E. (2020). Rcpp for everyone. [Online; accessed 14-April-2025].

Tukey, J. W. et al. (1977). *Exploratory data analysis*, volume 2. Reading, MA.

Urbanek, S. (2021). *rJava: Low-Level R to Java Interface*. R package version 1.0-6.

Ushey, K., Allaire, J., and Tang, Y. (2023). *reticulate: Interface to 'Python'*. R package version 1.31.

Van Rossum, G. and Drake, F. L. (2009). *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA.

Venables, W. N. and Ripley, B. D. (2002). *Modern Applied Statistics with S*. Springer, New York, fourth edition edition. ISBN 0-387-95457-0.

Wand, M. (2023). *KernSmooth: Functions for Kernel Smoothing Supporting Wand and Jones (1995)*. R package version 2.23-21.

Wickham, H. (2010). A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28.

Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.

Wickham, H. (2019). *Advanced R*. CRC press.

Wickham, H. (2021). *Mastering Shiny*. O'Reilly Media, Inc.

Wickham, H. (2022). *lobstr: Visualize R Data Structures with Trees*. R package version 1.1.2.

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Müller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., Takahashi, K., Vaughan, D., Wilke, C., Woo, K., and Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, 4(43):1686.

Wickham, H., Çetinkaya-Rundel, M., and Grolemund, G. (2023). *R for data science*. O'Reilly Media, Inc.

Wikipedia (2023a). Ascii. [Online; accessed 9-November-2023].

Wikipedia (2023b). Ms-dos. [Online; accessed 5-September-2025].

Wikipedia (2023c). Null object. [Online; accessed 14-December-2023].

Wikipedia (2023d). Perl. [Online; accessed 21-December-2023].

Wikipedia (2024a). Arithmetic underflow. [Online; accessed 7-February-2024].

Wikipedia (2024b). C++. [Online; accessed 11-April-2025].

Wikipedia (2024c). Capacitor. [Online; accessed 14-February-2024].

Wikipedia (2024d). Cielab color space. [Online; accessed 20-August-2024].

Wikipedia (2024e). Decimal number. [Online; accessed 5-February-2024].

Wikipedia (2024f). Fortran. [Online; accessed 21-October-2024].

Wikipedia (2024g). Language binding. [Online; accessed 19-November-2024].

Wikipedia (2024h). Pseudorandom number generator. [Online; accessed 4-November-2024].

Wikipedia (2024i). S (programming language). [Online; accessed 23-October-2024].

Wikipedia (2024j). String (computer science). [Online; accessed 22-January-2024].

Wikipedia (2024k). Subnormal number. [Online; accessed 7-February-2024].

Wikipedia (2024l). Transistor. [Online; accessed 14-February-2024].

Wikipedia (2024m). Utf-16. [Online; accessed 26-March-2024].

Wikipedia (2024n). Utf-8. [Online; accessed 26-March-2024].

Wikipedia (2025a). Machine code. [Online; accessed 6-May-2025].

Wikipedia (2025b). Markup language. [Online; accessed 28-January-2025].

Wikipedia (2025c). String interning. [Online; accessed 16-May-2025].

Wikipedia (2025d). Usage share of operating systems. [Online; accessed 5-March-2025].

Wilkinson, L. (2012). *The grammar of graphics*. Springer.

Wood, S. (2017). *Generalized Additive Models: An Introduction with R*. Chapman and Hall/CRC, 2 edition.

Wood, S. N. (2011). Fast stable restricted maximum likelihood and marginal likelihood estimation of semiparametric generalized linear models. *Journal of the Royal Statistical Society (B)*, 73(1):3–36.

Xie, Y. (2013). animation: An R package for creating animations and demonstrating statistical methods. *Journal of Statistical Software*, 53:1–27.

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2016). *bookdown: Authoring Books and Technical Documents with R Markdown*. Chapman and Hall/CRC, Boca Raton, Florida.

Xie, Y. (2023). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.34.

Xie, Y. (2024). *tinytex: Helper Functions to Install and Maintain TeX Live, and Compile LaTeX Documents*. R package version 0.51.

Xie, Y., Allaire, J. J., and Grolemund, G. (2018a). *R markdown: The definitive guide*. Chapman and Hall/CRC.

Xie, Y., Cheng, J., and Tan, X. (2024). *DT: A Wrapper of the JavaScript Library 'DataTables'*. R package version 0.33.

Xie, Y., Dervieux, C., and Riederer, E. (2020). *R markdown cookbook*. CRC Press.

Xie, Y., Mueller, C., Yu, L., and Zhu, W. (2018b). *animation: A Gallery of Animations in Statistics and Utilities to Create Animations*. R package version 2.6.

Zelazo, P. R., Zelazo, N. A., and Kolb, S. (1972). "Walking" in the newborn. *Science*, 176(4032):314–315.

Zhu, H., Travison, T., Tsai, T., Beasley, W., Xie, Y., Yu, G., Laurent, S., Shepherd, R., Sidi, Y., et al. (2022). kableextra: Construct complex table with "kable" and pipe syntax. 2021. *URL https://CRAN. R-project. org/package= kableExtra. R package version*, 1(1):579.

Index of Terms

Index of R Operators and Functions