James E. Gentle

R for Data Science and Applications in Finance

Preface

This book began as a set of notes for myself to facilitate my use of R for various applications, mostly in the analysis of financial data. I wanted a ready reference for such simple tasks as initiating or manipulating `xts` objects or adding graphics elements to a plot that was not produced by the base `plot` function. Initially this was just a collection of R scripts, but I decided that embedding them in a LaTeX 2_ε document with an index would make them more useful. I tried to make the index complete and useful to me. It contains a mix of task-oriented terms relating to financial data analysis and names of computer functions or terms relating to computer usage.

The result is this book that is now intended for other persons with interest in analyzing financial data using R.

The book uses real financial data in the examples. The book identifies internet repositories for getting *real* data and *interesting* data.

In addition to real data, the book discusses methods of simulating artificial data following various models, and how to use simulated data in understanding and comparing statistical methods.

Data preparation and cleansing are also discussed.

Chapter 1 is on the basics of R, but it is not a tutorial on R. It emphasizes features of R that make it a particularly useful software system for financial analysis. Section 1.6 discusses open sources of financial data and access of the data using R. Chapter 2 focuses on nature of financial data, in particular on equity returns and their statistical properties.

Chapter 3 describes simulation methods in R. Monte Carlo simulation plays a major role in analysis of financial data. Chapter 4 covers graphics in R. Plots for time series objects with date indexes require specialized graphics functions, and these functions are discussed.

Chapter 5 covers use of R in financial time series analysis. There are various ways of representing time series data in the computer. Simple time series data are equally spaced, so the indexing can be sequential integers. The simple time series data objects in R inherit from numeric vector or matrix objects. These objects add only metadata to specify the beginning and ending dates, and the

frequency within simple integer indexes. Chapter 5 describes the simple time series objects and how they are used.

Unequally-spaced time series require a date object as an index. Chapter 5 describes date data and how to manipulate it, and then discusses R objects for unequally-spaced time series and how to manipulate those objects.

Chapters 7 and 8 describe use of R in two important areas of numerical analysis, linear algebra and optimization.

Chapter 9 covers use of R when high performance computing is important, such as when very large datasets are to be analyzed.

A reader with prior exposure to R may be able to skip or to skim various sections of the book, but even a reader with no experience with R should be able quickly to pick up enough R to produce simple plots and perform simple analyses. The best way to do this is to look at a few code fragments, execute the code, and then make small changes to it and observe the effects of those changes.

Acknowledgments

I thank the developers of the R software system and the R Core Team who maintain and enhance the system. I thank the many package developers and those who maintain the packages for continuing to make R more useful.

I used TEX via LATEX 2$_\varepsilon$ to write the book. I did all of the typing, programming, etc., myself, so all mistakes are mine. I would appreciate receiving notification of errors or suggestions for improvement.

Fairfax County, Virginia — James E. Gentle
June 11, 2020

Contents

1

R: The System and the Language

R is an interactive computer language and system that is particularly well-suited for statistical applications. It is an object-oriented functional programming system.

The standard R GUI has a subwindow for code (the "Editor" window), and a subwindow to interact directly with the R interpreter (`stdin`, the "Console" window). Most R users write R statements in the console and then submit them directly to the system for execution. RStudio, which is a free and open-source integrated development environment for R, provides an even more useful interface.

R is available for download for various computer platforms from the Comprehensive R Archive Network (CRAN) at

`cran.r-project.org/`

Executables for various Apple Mac, Linux, and Microsoft Windows platforms can be downloaded from that site. The R system is maintained by the R Core Team (2020).

Although this chapter is written at an introductory level and describes basic properties of R, it is not a general tutorial. The choice of topics is somewhat eclectic, but generally the emphasis is placed on particular features of R that are more likely to arise in applications to financial data analysis, such as how dates are specified and used.

A reader with some prior knowledge of R can skim most of the chapter, but should read more carefully less familiar material, such as perhaps date data in Section 1.2.5. Also, some more esoteric topics are mentioned, such as the reserved words and symbols in R (page 5).

R is a very well-defined system, and most things work just as a logical or intuitive interpretation of an R expression would lead the user to believe. There are some functions, operators, or objects, however, whose properties may seem counterintuitive, or else may not conform to forms that allow simple manipulation. I will often mention such aspect that I find difficult to work with (such as working with levels of factors, for example).

Packages and Functions

R has a modular design, and the basic system consists of a number of "libraries" or "packages" that provide specific functionalities. The basic R distribution comes with a small number of standard packages, with names like `base`, `utils`, `stats`, and so on. A package for a specific project or set of tasks can be developed by an R user, and if the package meets certain standards, the R Core Team may incorporate it into the CRAN distribution and documentation system. There are over 17,000 packages currently in CRAN that have been contributed by R users from around the world. Two other repositories of R packages are Bioconductor, at

```
www.bioconductor.org
```

and R-Forge, at

```
r-forge.r-project.org
```

Packages may duplicate one another in functionality, and they vary somewhat in their quality. Each package may bring its own baggage into the user's working environment, and sometimes this causes unexpected results for other common operations.

There are also a number of R functions and R packages available at GitHub in the various users' directories.

R is a programming language with operators, functions, constants, and datasets. The language provides the ability for the user to add new operators, functions, constants, and datasets.

Computational tasks are performed in R by functions and operators, so after understanding the basic syntax of the language, "learning R" consists mainly of learning the functions that perform the tasks to be accomplished, and learning what packages contain those functions.

Functions in different packages could have the same name, but perform different tasks or perform tasks differently. The packages in CRAN are controlled so as to prevent duplicate names, but in any event, it is important to know which package contains a given function. A specific function can be specified by naming the package and the function, in the form `stats::smooth()` for the `smooth()` function in the `stats` package, or `forecast::ma()` for the `ma()` function in the `forecast()`, for examples.

Versions of R and Other Distributions

The design and development of R has been very orderly. Frequent updates are released, but the updates are well-tested prior to being released, and in most cases, backward compatibility is maintained.

The version numbering scheme consists of three fields, separated by periods: x.y.z. Many users of R maintain their own computing environments, and these users should be diligent in updating their R versions. (I do it once or

twice a year.) Installation of a new version often means that packages must be reinstalled. Packages themselves must also be updated occasionally.

In addition to the standard distribution versions of R available from CRAN, there are various add-ons and enhancements, some also available from CRAN. One of the most notable of these is RStudio, which is an integrated R development system including a sophisticated program editor. (I now use RStudio most of the time that I use R.)

Microsoft R Open (MRO), formerly known as Revolution R Open (RRO), is an enhanced distribution of R from Microsoft Corporation. An important feature of Microsoft R Open is the use of the Intel Math Kernel Library (MKL) to allow multiple threads if the CPU follows the Intel multi-core architecture. This can result in significant speedup of numerical computations. MRO also provides an ability to maintain snapshots of the system, including packages, so that R scripts will yield reproducible results even if components of the the system change over time.

Statistical analysis of financial data requires access to the data, and software to manipulate the data and to perform statistical analyses. There are many sources of financial data. We will discuss some sources and how to access them through R in Section 1.6.

1.1 Documentation

There are a number of books and tutorials on R, and a vast array of resources on the internet. Some of the more useful advanced books on R are listed in the references beginning on page 221. There are also many online resources, such as those listed at

```
cran.r-project.org/other-docs.html
```

Two additional useful sites are

```
www.rdocumentation.org/
```

and

```
rdrr.io/
```

The latter site has a page that allows the user to type a snippet of R code, execute it, and receive the output in a subwindow of the browser. This allows access to the R system on a connected device where R is not installed, such as on a tablet or a smart phone.

Many useful functions are in packages that are not included in the standard R distribution. To find packages and other software that provide functions for a specific task, the Task View webpage at CRAN may also be useful:

```
cran.r-project.org/web/views/
```

The Task View webpage lists relevant packages for the various types of computations or areas of application.

To use a function, the package containing that function must be loaded in the current R session.

Example Code in this Book

I will show many short fragments of R code. Most can be executed as shown, although in a few cases, I may omit some necessary initializations.

Many snippets of R code are shown. Whenever the results shown are not immediately obvious, the reader should go through them in detail, possibly in an R session. In some cases, the R code shown is a straightforward translation of basic mathematical formulas for clarity, but it may not be the best way to perform the computations.

In the example code, I will often show exactly what the user types in the R console in a form such as this:

```
x <- 5
```

(This statement means that the value 5 is to be assigned to a variable that the user has named "x".) Sometimes I show an R prompt before the user's input together with the output that R prints to the console. Tokens that are part of the R system, other that simple operators such as "<-" or "+", will be written in a different font, such as `sqrt()`.

```
> y <- sqrt(9)
> y
[1] 3
```

If the code fragment requires some setup or other statement that are not shown, I will use an ellipsis to indicate that statements are missing:

```
   ...
ysum <- ysum + sqrt(yi)
```

Also if the R code in the example requires a package other than the standard ones (`base`, `utils`, `stats`, and so on), I will indicate which packages must be loaded, using a different font such as `forecast`. Packages are loaded by the R function `require()` or `library()`:

```
   ...
require(forecast)
Acf(z)
```

Names and Special Characters

Entities in R are identified by names, which either are chosen by the user to represent user-defined objects or are set by the system. In the examples above, the names `sqrt`, `require`, `forecast`, and `Acf` were set by the system. The names `x`, `y`, `ysum`, `yi`, and `z` were chosen by the user. Names are case sensitive.

User-chosen names must conform to certain rules. Names cannot include special characters such as the arithmetic or logical operators (see page 11), or other special symbols such as "[" or "(". The first character in a name cannot be a numeral.

Names cannot be the same as R program control words such as "`for()`" (see page 16). These are *reserved words*. Also, all of the missing value logical constants such as "`NA`" as described in Section 1.2.6 are reserved. Names of ordinary R functions are not reserved. In general, however, it is better not to use such names so as to avoid confusion.

There is also some special characters in R. For example, "`#`" is a special character generally indicating that what follows is a comment not to be interpreted or executed. (The "`#`" character may also be used within a character string to specify a hexadecimal value defining a color; see page 156.) The "`#`" character cannot appear in the name of an R object.

Another special character is the "backtick" or backslash "\". This character is used as a special type of quote (see Figure 1.9), but its more common usage is as a control character, similar to its use in the C programming language. For example, in a character string, "`\n`" means a new line. The string "`\\`" means the single character "\". Within a character string, "`\x`" followed by two hexadecimal characters designates a single character in the ASCII coding. For example, "`\x4B`" is upper-case "K" and "`\x6B`" is lower-case "k". The ASCII hexadecimal coding for a blank character is "`\x20`".

```
> cat("\x4B\x20\x6B\n")
K k
>
```

The "\" character cannot appear in the name of an R object.

1.1.1 Help

R has an integrated help system for functions and other objects included in the system. Documentation is stored in PDF or HTML format and stored locally. In the standard GUI setup, the documentation can be accessed from a "Help" menu.

Documentation for R objects is generally stored in the form of `man` pages and can also be accessed through the R `help()` function, for example, for the R function `rnorm()`,

```
help(rnorm)
```

The R "?" operator also performs the same operation:

```
?rnorm
```

Depending on the platform and internet connectivity, the `help()` function opens a browser that displays the `man` page at an R site.

The help functions also work for R operators, but the operator symbols must be quoted, for example,

```
?"%*%"
```

The help functions also provide information about other objects included with the R system, such as constants and datasets.

```
?pi
?cars
```

(`pi()` is the constant π, and `cars()` is a built-in dataset.)

If we are already familiar with an R function, but just need a reminder of the arguments for the function and any defaults they may have, we can use the `args()` function:

```
> args(rnorm)
function (n, mean = 0, sd = 1)
NULL
```

(`NULL` has no relevant meaning here.)

1.1.2 Demos and Vignettes

Examples of the use and output of some R functions can be obtained by use of the `demo()` function. If a function has a set of demos at the CRAN site, the `demo()` function will source the demo scripts into the console. For example,

```
demo(smooth)
```

This sources several snippets of code to illustrate the `smooth()` function on different dataset with various options. There are no demos for most R functions, and there is considerable variation in the demos provided by `demo()`.

More information and examples for functions and other objects included in packages are often available through the `vignette()` function. This function is (or may be) particularly useful for obtaining information on a package or a class of objects used in a package.

Note that the argument to `vignette()` must be quoted.

```
vignette("xts")
```

There are no vignettes for most R packages, and there is considerable variation in the information provided by `vignette()`. Invocation of this function may initiate a window with just a few comments, or it may bring into view an extensive PDF document.

1.1.3 Finding Functions and Other Objects

As mentioned above, "learning R" consists mainly of learning the functions that perform the tasks to be completed, and learning what packages contain those functions. Learning R also involves becoming familiar with the other object provided by the system.

Using the `help()` function requires knowledge of the name of the function or operator. The `help()` function also requires a package containing the function be loaded in the current session.

Often, the easiest way to find the name of a function to do a specific task is just to search the internet for that task; for example, use a search engine to search for "smoothing in R", and you will likely find the name of one or more R functions useful in smoothing. The Task View webpage, as noted above, may be useful for finding relevant packages.

The `help.search()` function can be used to search for functions in any installed R package to do a specific task, based on a character string. For example, to find functions in any installed packages that do smoothing, we may use

```
help.search("smooth")
```

Note that the character string must be quoted. (In the `help.search()` function, the character string is not the name of an R function; although in this case, it turns out that the R function is indeed named `smooth()`.)

The `help.search()` function opens a browser that displays a page listing all functions in all of the installed packages whose `man` pages contain the specified string. The items in the list give the package name, the function name, and a brief statement of what the function does:

```
stats::smooth   Tukey's (Running Median) Smoothing
```

If any of the list functions have demos, they are listed, and vignettes associated with relevant packages that have vignettes are listed. Both demos and vignettes are linked to the corresponding items among installed packages. The names of functions are linked to the `man` pages.

The `help.search()` function does not require that the relevant packages are loaded, only that they be installed. The package containing any function to be used, or to be directly accessed by the `help()` function, must be installed and loaded in the current session. (See page 42 for information on how to do this.)

To determine what packages are currently loaded, the `search()` function can be used (see page 39), and to determine what functions are in a given package that is loaded, the `ls()` function can be used; see Figure 1.17 or the other example on page 41. (The `ls()` function is essentially the same as the Unix `ls`; it means "list".)

The help system also leads the user to an index of all functions in a given package. (Using the `help()` function for any function in a package brings up a link to the index for that package.)

Once the name of the function is known and the package containing the function is loaded, the `help()` function can be used to get a description of the function.

Exercises: Documentation

The exercises generally require you to use R. In many cases, the exercises require computations on data. If the emphasis is on the method rather than the actual data, we will often use artificially generated "random" data. Standard normal random numbers serve this purpose well, and when an exercise asks you to "generate 100 standard normal random numbers", the simple R statements

```
set.seed(12345)
x <- rnorm(100)
```

will yield the sample data. The `set.seed()` function ensures that the same numbers can be generated.

1.1.1. a) Find two functions in the basic `stats` package that perform the computations for principal components analysis.
   b) What are the differences in these functions?

1.1.2. What does the binary operator `%in%` do, and when/how would you use it?

1.1.3. On a phone or a tablet, navigate in a browser to `rdrr.io`. Select the "Run in browser" option.
   a) Execute the `help()` function for `rnorm()`.
   b) Generate 100 standard normal random numbers, compute their mean, and plot them. Code to do this is

```
x <- rnorm(100)
mean(x)
plot(x)
```

1.2 Program Structure and Syntax

Entities in R are identified by names that are case sensitive and must conform to certain rules as mentioned above. Operations are performed by functions.

One of the most common functions is `assign()`, which assigns one entity to another.

```
assign("r", 5)
assign("s", 3)
assign("rp", exp(r) + cos(2*pi*s))
```

R functions and operators for manipulating character strings, such as `paste()` and `as.character()`, can be useful in forming the name of the object in the `assign()` function.

There is also an R assignment operator, "`<-`", which we have seen in the examples above. One of the most common types of R statements is an *assignment statement*, which assigns values to an object by use of the `<-` operator.

```
r <- 5
s <- 3
rp <- exp(r) + cos(2*pi*s)
```

Other statements control the flow of a program or a group of statements, as perhaps in a loop, request that a function or group of statements be executed, or merely request that the values of an object be displayed. A statement that consists only of the name of an object results in the display of that object:

```
> r <- 5
> r
[1] 5
```

This last illustration is intended to replicate the appearance of the R user's console. The has "`>`" symbols are meant to represent the prompts on the user's console. I will often use this form when I want to show both what the user types and what the R system displays.

An R statement may also just be a comment for the reader or the programmer. A comment is any sequence of characters not enclosed in quote symbols that follow the symbol "#". When a sequence of R statements is put together,

comments may help to understand what the statements do. The readability of a group of R statements may also be improved by indentation. Indentation or the placement of a statement on a line has no meaning in R.

```
y <- r+3
       #  This is a comment
```

R statements are line-oriented but continue from one line to another until a syntactically complete statement is terminated by an end-of-line:

```
x <- r   #  this statement is syntactically complete; x gets r
y <- r+  #  this statement is not yet syntactically complete
    3    #  it is now complete; y gets r plus 3
```

Multiple statements on the same line are separated by semicolons:

```
x <- 5; y <- x+3
```

1.2.1 Functions and Operators

Actions in R are performed by functions or operators. There are a number of functions and operators provided in the R system, and we will give a brief overview here. In Section 1.4 we will discuss functions and operators in more detail, and in Section 1.4.2 we will list important R functions for common mathematical computations, including computations involving common probability distributions in. The user can also add functions and operators to an R session, as we will illustrate in Section 1.2.3.

R Functions

The R system provides a number of functions, including most of the standard mathematical functions, such as sin, cos, $\sqrt{\ }$, exp, and so on. These functions have the same name as used in mathematics, or else a simple mnemonic name: `sin()`, `cos()`, `sqrt()`, and `exp()`.

A function in R is invoked by using the name of the function followed by matching parentheses, with or without arguments between them. The function can be invoked within any syntactically correct expression, including a function. (There are some technical issues that involve the actual execution of a function, which we will not get into here.) A function may return an R object or it can alter the R environment.

Many standard Unix commands, such as "`ls`", "`grep`", "`cat`", and so on, have corresponding R functions that have essentially the same meaning in R as in the Unix kernel operating system.

There are functions that provide other types of action, such as the `help()` function mentioned above. Some functions perform rather complicated statistical analyses, such as `lm()`, which fits linear models and computes relevant statistics for analysis of the model (see Section 7.4), or `arima()`, which fits an ARIMA model to a time series (see Section 5.3).

R also includes some binary functions for set operations, including `union()`, `intersect()`, and `setdiff()`. The `setequal()` and `is.element()` functions return `TRUE` or `FALSE` depending on the equality of two set or the inclusion of an element in a set. (See Figure 1.41 on page 104.)

R Operators

An "operator" is a function expressed in a special, usually more convenient syntax. The common arithmetic operator "`+`" uses a simple form to add two numbers, for example.

There are other R operators for common tasks, even when there is a simple R function to perform the task. For examples, the R "?" operator performs the same operation as the `help()` function, and the R assignment operator, "`<-`" performs the same operation as the `assign()` function.

Arithmetic Operators

R has the usual arithmetic operators "`+`", "`-`","`*`", "`/`", and "`^`". Most of these are binary infix operators, used for example as in `x+y` or `x-y`. The operator "`-`" can also be used as a unary prefix operator as in `-y`. R also has a mod operator "`%%`" where `x %% y` yields $x \mod y$.

There are also two useful arithmetic operators for use with vectors and matrices, the Cayley product "`%*%`" and the outer product "`%o%`". We will discuss these in Section 1.3.2, and in Chapter 7 we will discuss functions for operations on vectors and matrices in general.

Logical Operators

Logical conditions in R, which have a value of `TRUE` or `FALSE`, can be defined by logical operators. The negation logical operator is `!`, which is a unary prefix operator.

The common binary infix logical relational operators are `==`, `!=`, `<`, `<=`, `!<`, `>`, `>=`, and `!>`, with obvious meanings. The conjunctions are `&&` and `|`. The binary function `xor` performs the exclusive operation. R also has a logical operator for set inclusion, `%in%`. Examples:

```
3<5 is TRUE
3<2 && 2+2==4 is FALSE
3<2 | 2+2==4 is TRUE
xor(3<2, 2+2==4) is TRUE
```

xor(2<3, 2+2==4) is FALSE
2%in%1:4 is TRUE

(The is.element() function performs the same operation as the operator %in%.)

A logical condition can be an operand in an arithmetic expression. A TRUE condition has a value of 1 and a FALSE condition has a value of 0; for example,

```
> 1+(3<2)*2+(3<5)*3
[1] 4
```

Logical conditions are often used to determine whether or not a group of R statements are to be executed using if(), else(), and elseif() statements. Logical conditions are also used in the looping control statements, for() and while().

Other Operators

R also has a number of operators of various types, such as the unary prefix "?" illustrated above. A useful operator that can be used to generate a sequence of values is the ":" (colon) operator. It can be used when a given beginning value and ending value have an obvious sequence of values between them, as for example 8 and 12. The obvious values between them are the integers 9, 10, and 11. With a starting value of 12 and ending value of 8, the values would be the same, except in reverse order. Negative integers can also imply an obvious sequence of intermediate values.

```
> 8:12
[1]  8  9 10 11 12
> 12:8
[1] 12 11 10  9  8
> -2:3
[1] -2 -1  0  1  2  3
> 2:-3
[1]  2  1  0 -1 -2 -3
```

The colon operator can also be used with non-integral values, so long as the implied sequence is "obvious". Sometimes an "obvious" approximation is made:

```
> 4.5:3.5
[1] 4.5 5.5 6.5
> 4.5:6.6
[1] 4.5 5.5 6.5
```

The sequence operator has various other forms (see Figure 1.4 on page 25, for example). The `seq()` function provides more options for generating a sequence.

Another operator that we will encounter often is the binary infix operator "`$`", which is used to extract components of objects, as we will discuss in Section 1.3.4.

An important operator in statistical data analysis is the tilde operator, "`~`", to separate the left-hand and right-hand sides of a statistical model. There are many details in the syntax for specification of a statistical model, and we will not go into them them. As an example, consider the multiple linear regression model

$$y_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \beta_3 x_{3i} + \epsilon_i. \tag{1.1}$$

If the regression variables in R are `y`, `x1`, `x2`, and `x3`, the linear regression model with the intercept and additive error term as in equation (1.1) would be specified in R as

```
y ~ x1+x2+x3
```

There is extensive syntax in R to allow for many variations on this; see Table 1.9 on page 127.

While the R language does not support pipes, a piping operation can be implemented by a binary operator. The operator `%>%` in the `magrittr` package performs a pipe:

```
> library(magrittr)
> x <- 9
> x %>% sqrt()  # x  is piped to sqrt
[1] 3
```

Piping is of very little utility in ordinary numeric computations, but it may be useful in set operations.

Precedence of Operators

The order in which operators in an expression are evaluated depends on how the operators are grouped syntactically and within a single group, on their *precedence*. For example, the value of the expression

```
12+4*2
```

depends on whether "+" is evaluated before "*" or "*" is evaluated first. If the precedence of two operators are different, the one with "higher precedence" is evaluated first.

There are fixed rules of precedence for R operators, some of which I state below. First, however, let us observe the effect of good programming practice. When grouping symbols are used, we do not have to know the rules.

```
(12+4)*2  # +  is performed first
12+(4*2)  # *  is performed first
```

We know, and anyone looking at the program segment knows, what the expression means in each case.

In general, the binary arithmetic operators have precedence order from high to low: first "^", next "*" and "/", then "+" and "-". The unary prefix operator "-" has higher precedence that any of these:

```
> 12^2+4*2
[1] 152
> 12^2+4*-2
[1] 136
> 12^-2+4*2
[1] 8.006944
```

Most operators with equal precedence are evaluated left to right:

```
> 8/4/2
[1] 1
> 8/(4/2)
[1] 4
```

but some are evaluated right to left:

```
> 2^4^2
[1] 65536
> (2^4)^2
[1] 256
```

The colon sequence operator ":" has precedence between the exponentiation operator and the multiplicative operators, as we can see in the following examples.

```
> 4.5:6.5
[1] 4.5 5.5 6.5
> 4:6+.5
```

```
[1] 4.5 5.5 6.5
> 1:(3+2)
[1] 1 2 3 4 5
> 1:3*2
[1] 2 4 6
> 1:(3*2)
[1] 1 2 3 4 5 6
> 1:3^2
[1] 1 2 3 4 5 6 7 8 9
> 1:(3^2)
[1] 1 2 3 4 5 6 7 8 9
```

This precedence of the colon operator is often a source of errors for R programmers. The precedence ranking is counterintuitive for many users, so I highly recommend that grouping symbols be used anytime the sequence operator is used together with any other operator.

The piping operator `%>%` in the `magrittr` package is of higher precedence than the arithmetic operators.

```
> library(magrittr)
> x <- 9
> x+16 %>% sqrt()
[1] 13
> (x+16) %>% sqrt()
[1] 5
```

A complete precedence list can be obtained by

```
?Syntax
```

1.2.2 Program Control Structures

R statements can be grouped into control structures by "{" and "}". With the beginning and ending grouping symbols, there is no need for a "begin" or an "end" statement.

Grouped statements can be executed together as a subprogram (formed by a `function()` statement). They can be executed conditionally using `if()`, `else()`, or `elseif()` statements:

```
   ...
if (xi>=0) {
  nnonneg <- nnonneg+1
  sum <- sum + sqrt(xi)
} else {
  nneg <- nneq +1
}
```

Grouped statements can also be executed in a loop using `for()` or `while()` statements:

```
n <- 10
sum <- 0
sumsq <- 0
for (i in 1:n) {
  sum <- sum + i
  sumsq <- sumsq + i^2
}
```

Conditional and repetitive tasks can often be performed by use of an R function, and in general, it is best to avoid use of the conditional and looping constructs above if an alternative is available.

1.2.3 User-Written Functions and Operators

A simple way a user can extend R is to write a function using the R function constructor called `function()` and the grouping symbols "{" and "}". New functions can be added to an R session with ease. Multiple functions may have the same name, but only one is active at a time. A function can be referenced uniquely by specifying the package it is in together with its name (see discussion on environments beginning on page 38).

A function consists of arguments (called "formals"), a body, and the current environment from with the function is invoked. The `return()` function within the body of the user function causes the execution of the user function to terminate and the argument of the `return()` function to be returned as the value of the user function. If the argument of the `return()` function is missing, the value returned by the user function is `NULL`. If the end of a function is reached without calling `return()`, the value of the last evaluated expression is returned.

An example of an R function called `myFun` is shown in Figure 1.1. This function determines the slope and intercept of the line that is determined by two given points. Notice how it handles the special case of a vertical line. (A more careful implementation would determine if the value `abs(x2-x1)` is too small so as to prevent overflow of the result.) The `myFun` function returns the intercept and slope in a numeric vector. If the the `return()` function is not invoked explicitly, the user-written function will return the expression that is evaluated just before the function is exited (in this case, just `intercept`).

Once the function has been entered into the R session, the function can be invoked by assigning values to the arguments `x1`, `y1`, `x2`, and `y2`, called "formal arguments", and then issuing the R statement

```
ab <- myFun(x1,y1, x2,y2)
```

```
myFun <- function(x1,y1, x2,y2){
#   Given points (x1,y1) and (x2,y2), determine the intercept and slope of
#   the line that goes through them.
#   Returns a vector of length 2.
  if (x2==x1) {
    slope <- 1
    intercept <- 0
  } else {
    slope <- (y2-y1)/(x2-x1)
    intercept <- y1-slope*x1
  }
  return(c(intercept,slope))
}
```

Figure 1.1. A Simple Function in R

The names `x1`, `y1`, `x2`, and `y2` are *formal arguments* to the function. They are assigned a value when the function is invoked, and then are used within the function. Any changes made to the formal arguments within the function are not passed back to the environment in which the function was invoked.

There are many tools for debugging an R program. A very useful tool to use when writing a function is the R function `browser()`. This function, without arguments, causes the execution of the user's function to pause and return program control to the user, who can then have access to the variables within the function. The R function `ls()` is also useful within a user-written function to identify local variables (see Figure 1.17).

A common error when a large program is being developed is that variables are referenced without being initialized. Sometimes this is because the names are miss typed; other times it is just an oversight. If there is a variable in the R workspace whose name is the same as the uninitialized variable, then that value will be used, and neither the R system nor the user will realize the mistake, unless there is a type mismatch. In the process of developing and testing the program, the R statement

```
rm(ls())
```

is sometimes useful. This statement removes all of the variables in the workspace, so it has use only in the program development process.

Two other useful debugging tools are the packages `testthat` and `assertive`. Cotton (2017) provides a discussion of debugging methods in general and a tutorial on the use of these two packages in particular.

In a simple case as in Figure 1.1, the computations of the function are expressed in ordinary R statements. In other cases, the computations can be

expressed in a compiler language such as Fortran or C, compiled, and then linked into R. In either case, the user's function is invoked by its name, just as if it were part of the original R system.

In addition to defining new functions, whether in R or in some other code that is linked into R, there are also other ways in which R can be extended. New classes and new environments (see Section 1.2.7) can be defined and incorporated into an R session.

An alternative to a user-written R function is an R GUI app, which may be useful in a task in which various values may be easily entered and evaluated.

R GUI Apps

Using an R GUI app may be very different from using R itself. Once an R GUI app is created, probably by someone conversant in R, the app can be shared with other users who may not even know R. Most R GUI apps allow interaction with the R system, however. An R GUI app, for example, can have the appearance of an Excel spreadsheet. (Microsoft Excel® is the most widely used program in the world for statistical analyses.)

There are several packages that create GUI apps. The most widely used package is `Shiny`, developed by RStudio.

User-Defined Operators

The user can also define operators in R using the `function()` function. It is standard practice to name a newly-defined function with some meaningful character or character string surrounded by "`%`" on either side, such as the built-in operators `%*%`, `%o%`, and `%in%` that we have already mentioned.

To define an operator, we enclose it in quotes, either `"` or `'`.

As an example, let us consider the special operator "\" that is provided by MATLAB® for solving a system of linear equations. (MATLAB is a general-purpose mathematical software system that I will refer to occasionally. I will generally use the name in mostly lower case, Matlab. Octave is an open-source free clone of Matlab and provides much of the same functionality as Matlab. I will sometimes refer to the two as "Matlab/Octave".)

For the system of linear equations $Ax = b$ with `A` and `b` properly initialized as Matlab objects, the Matlab statement

```
>> x = A\b
```

causes the solution to be computed. For example

```
>> A = [1,2;6,3];
>> b = [8;21];
>> x = A\b
x =
   2
   3
```

We will now define an R operator that does the same as the Matlab operator "\".

This example uses some functions in R for working with matrices that we will encounter in Section 1.3.2. The example also emphasizes the existence of special characters in R (see page 5).

The R expression that would most closely mimic the Matlab operator is "%\%". As it turns out, however, "\" is a special character in R, and cannot be used in any user-defined name or expression. Therefore, we choose the symbol "%bs%" (for "backslash") to represent the operator. We define it as

```
"%bs%" <- function(A,b) return(solve(A,b))
```

using a standard R function, `solve()` (that does the same thing).

To use the operator, we write it without the quotes:

```
A <- matrix(c(1,2,6,3), nrow=2, byrow=TRUE)
b <- c(8,21)
x <- A%bs%b
```

1.2.4 Characteristics of Data: Classes, Modes, and Types

R is an *object-oriented* system. (Computer science purists may make various distinctions regarding an "object orientation", but we will ignore them here.) A basic idea in object-oriented systems is that "objects" (all "things" are objects) are characterized by properties that determine what computational methods are applied to the object. The properties go with the object itself, so that details of the computational method do not need to be specified each time the method is applied.

In this section we will discuss characteristics of individual atomic data items and simple sequences of items of the same kind. In Section 1.3, we will discuss more complicated objects that may consist of various arrangements of elements of different kinds.

Objects in R are characterized in various ways, such as *class*, *mode*, and *type*. The class of an object, which is its most essential characteristic, can be determined by the function `class()`. The mode of an object can be determined by the function `mode()` and the type can be determined by the function `typeof()`. An R function, whether built-in, such as `sqrt()`, or user-defined, such as `myFun` defined in the R statements shown in Figure 1.1 on page 17, is of class `function` and mode `function`.

```
> class(sqrt)
[1] "function"
> mode(sqrt)
```

```
[1] "function"
> class(myFun)
[1] "function"
> mode(myFun)
[1] "function"
```

The type of functions may vary.

The mode and the type correspond to the most basic nature of an object. Although there are slight differences in mode and type, we will ignore them here. The characteristics of an object are determined when it is first initialized, and those characteristics determine what kinds of operations can be performed on the object.

In this section we will be concerned with various classes of data: *factor*; *numeric* and the subclass *integer*; *complex*; *character*; *logical*; and *date*. In statistical datasets, we often describe some variables and "categorical", meaning that they do not represent ordinary measurements or counts. In R, the `factor` and `character` classes generally correspond to categorical variables. We will discuss these classes and give examples of them in this section.

First, however, we will discuss a simple structure formed by a sequence of atomic objects all of the same class.

Atomic Vectors

A basic R structure is a single sequence of individual elements of the same class. We call the object an "atomic vector". The qualifier "atomic" is to distinguish the vector fram a *list*, which is a vector that may contain elements of different classes. We will use the term "vector" to refer to an atomic vector. The elements of a vector in this sense are not necessarily numbers. For vectors containing numeric values, we can perform the usual vector operations in the mathematical sense, as we will discuss in Section 1.3.2.

A single datum is considered to be a vector with one element.

The class of a vector is generally not *vector*, but rather it is the same as the class of its elements.

A vector is constructed by the `c()` function.

```
> x <- c(9,8,7,6,5,4,3,2,1)
> x
[1] 9 8 7 6 5 4 3 2 1
> class(x)
[1] "numeric"
> a <- c("a", "bc", "d")
[1] "a"  "bc" "d"
> class(a)
[1] "character"
```

An element within a vector object is addressed by an index. Indexes of arrays are indicated by "`[ ]`". Indexing of arrays starts at 1; for example, `x[1]` refers to the first element of the one-dimensional array `x`, just as is common in mathematical notation for the first element in the vector x, that is, x_1.

Any set of valid indexes can be specified; for example, `x[c(3,1,3)]` refers to the third, the first, and again the third elements of the one-dimensional array `x`.

```
> x[1]
[1] 9
> x[c(3,1,3)]
[1] 7 9 7
```

Negative values can be used to indicate removal of specified elements from an array; for example, `x[c(-1,-3)]` refers to the same one-dimensional array `x` with the first and third elements removed. The order of negative indexes or the repetition of negative indexes has no effect; for example, `x[c(-3,-1,-3)]` is the same as `x[c(-1,-3)]`. Positive and negative values cannot be mixed as indexes.

```
> x[c(-1,-3)]
[1] 8 6 5 4 3 2 1
```

A useful function for working with vectors is `length()`, which returns the number of elements in an atomic vector.

```
> length(x)
[1] 9
> length(x[c(3,1,3)])
[1] 3
> length(x[c(-1,-3)])
[1] 7
```

The function `length()` operates on the entire vector at once. Other functions operate on the individual elements of the vector.

When a function that ordinarily has an atomic element as its argument is applied to a vector, the function is applied to each element in turn, yielding an output that is a vector, as in Figure 1.2.

This property of R functions is one of the most important aspects of the R system. It not only is a convenience for the programmer, it also results in efficient execution of the code.

```
> sqrt(x)
[1] 3.00000 2.82843 2.64575 2.44949 2.23607 2.00000 1.73205 1.41421 1.00000
```

Figure 1.2. Vectorized Function

Factors

In statistical data analysis, some variables in the dataset are used to identify groups or classes of the observations. These variables are called categorical variables. Such variables are called "factors" in R, and the different categories identified by the factors are called "levels". R treats all factor vectors in a consistent and efficient manner. (Recall that in a statistical dataset, variables generally correspond to vectors.) The levels of a factor are assigned to the positive integers, which then can be used to index the groups.

The reason factors are important is that they can be used to perform mathematical operations or statistical analyses on another variable separately at each level of a factor; see page 77 for an example.

A factor is constructed from an atomic vector of arbitrary mode by the `factor()` function. The elements in the atomic vector are converted to their character representations; for example, 2 becomes `"2"` and $4/2$ becomes `"2"`. The character representations are then sorted and the integers $1, 2, \ldots$ are assigned to them in order. The factor vector then contains integers $1, 2, \ldots$ corresponding to unique values of the underlying vector. The R function `levels()` shows the levels of a factor and the `str()` function shows both the levels and the individual elements. The `levels()` shows the length of the factor vector (not the number of levels).

To retrieve the original value of an element in the factor object, it is necessary to convert the factor object back to the character object, and then if necessary, convert the character object back to the class of the original object.

Figure 1.3 illustrates a factor variable. Notice that `levels(sexfac)[1]` is "f", which could be known only if the sorted values of the levels were known.

A factor is not a numeric vector, even though it is of mode `numeric` and type `integer`. An arithmetic function such as `sum()` cannot be applied to a factor.

Once a factor has been defined, its possible values are limited to the original levels. Individual values can be reassigned only if they are among the valid levels.

```
#  reassignment of factor values
> sexfac[1] <- "f"
```

```
> sexfac <- factor(
+       c("m", "m", "f", "m", "f", "f", "m", "f", "m", "f", "f", "f", "m", "f"))
> sexfac
 [1] m m f m f f m f m f f f m f
Levels: f m
> levels(sexfac)
[1] "f" "m"
> str(sexfac)
  Factor w/ 2 levels "f","m": 2 2 1 2 1 1 2 1 2 1 ...
> length(sexfac)
[1] 14
> class(sexfac)
[1] "factor"
> mode(sexfac)
[1] "numeric"
> typeof(sexfac)
[1] "integer"
> sexfac[1]
[1] m
Levels: f m
> as.character(sexfac[1])
[1] "m"
```

Figure 1.3. An R Factor Variable

```
> sexfac[2] <- "t"
Warning message:
In `[<-.factor`(`*tmp*`, 2, value = "t") :
  invalid factor level, NA generated
```

The reason factors are important is that they can be used to perform mathematical operations or statistical analyses on another variable separately at each level of a factor; see page 77 for an example.

Numeric Data

Numeric data basically corresponds to a finite subset of the real numbers. Technically, numeric data in R is of mode `numeric()` but has a type that depends on how the data are stored in the computer, such as `double()` or `int()`. All numeric data in R is of type `double()` by default. This corresponds to the standard definition of "double precision", which according to standards defined by IEEE, has at least 53 bits of precision. This precision corresponds to about 16 decimal places. (Numbers whose first digit is 1 in a decimal representation can be represented to higher precision than numbers whose first digit is 9.)

Numerical computations in R are generally performed in double precision. The results are usually displayed with six or seven digits. The user can control this throughout an R session by use of the `options()` function.

The R function `round()` returns the value of a numeric object rounded to the precision specified. Two related functions are `floor()`, which is the greatest integer function, and `ceiling()`, which returns the integer that is greater than or equal to the value of a numeric object.

The R function `sprintf()`, which is essentially the same as the C function or the same name, can be used to control the number of digits printed without changing the value of the object. For example, `sprintf("%.16f",x)` will display the value of the numeric object `x` showing 16 decimal places (to the right of the decimal point; not "significant digits"). For numbers very large or very small in absolute value, the exponential format would be preferred, and `sprintf("%.16e",x)` will display the value of the numeric object `x` in exponential notation with one digit to the left of the decimal point and 16 digits to the right of the decimal point.

Vectors of Numeric Data

If the elements of the vector are numeric and have a simple sequence, the "`:`" (colon) operator can be used to generate the elements. In that case, if the elements in the sequence are integers, the class of the vector is `integer()`; otherwise it is `numeric()`. Examples are shown in Figure 1.4.

There are many useful R functions, such as `min()`, `max()`, `sum()`, and `mean()`, that operate on numeric vectors (or more generally on numeric arrays, as we will see later). Two additional functions for working with numeric vectors are `which.min()` and `which.max()` that return the index of the minimum and maximum value respectively. These are illustrated in Figure 1.5.

Computations and other manipulations with a numeric vector in R are very similar to the computations and manipulations of a mathematical vector over a real field. A list of scalar numeric values can be treated as a vector without modification. Simple mathematical operations on vectors include scalar multiplication of a vector, and addition of two vectors form the same vector space; that is, vectors with the same number of elements from the same field. Numerical analysts call the combination of these two operations an *axpy* operation, from the common mathematical expression $z = ax + y$, where a is a scalar and x and y are vectors.

The ordinary arithmetic operators can be used with operands that are of different types, specifically scalars and vectors. (Technically, the operators are *overloaded*.)

In Figure 1.6 the "`+`" in `a+x` implies addition of two different types of objects, whereas the "`+`" in `a*x+y` implies addition of the same type of objects. The "`*`" in `a*x` implies multiplication of two different types of objects, a scalar and a vector.

```
> yi <- 1:9
> yi
[1] 1 2 3 4 5 6 7 8 9
> class(yi)
[1] "integer"
> y <- 1:9+0.5
> y
[1] 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5
> class(y)
[1] "numeric"
> xi <- 9:1
> xi
[1] 9 8 7 6 5 4 3 2 1
> class(xi)
[1] "integer"
> x <- c(9,8,7,6,5,4,3,2,1)
> x
[1] 9 8 7 6 5 4 3 2 1
> class(x)
[1] "numeric"
> x[2:6]
[1] 8 7 6 5 4
> x[-(2:6)]
[1] 9 3 2 1
```

Figure 1.4. Numeric Vectors and the Colon Operator

We will discuss additional operations on numeric vectors in Section1.3.2.

Complex Data and Arithmetic

Another kind of numeric data corresponds to a finite subset of the complex numbers, and may contain an imaginary component. This kind of numeric data is of mode `complex()` and also of type `complex()`. The real and imaginary components are both of mode `numeric()` and of type `double()`. A complex number can represented in a manner similar to an ordinary mathematical representation, such as $3 + 2\mathrm{i}$, but it better to construct the value using the R function `complex()`. The usual numerical functions and operators work with complex numbers in the usual way.

There is an R function, `complex()`, that can be used to instantiate a complex value. There is also a postfix operator, `i`, that represents $\sqrt{-1}$, which can be used to initialize a complex variable. The "`i`" in the formal argument list of `complex()` is not an operator. (For clarity, I recommend use of `complex()` instead of `i`.)

```
> x <- c(9,8,7,0,10,4,3,2,1)
> x[1]
[1] 9
> x[c(3,1,3)]
[1] 7 9 7
> x[c(-1,-3)]
[1]  8  0 10  4  3  2  1
> min(x)
[1] 0
> which.min(x)
[1] 4
> max(x)
[1] 10
> x[which.max(x)]
[1] 10
> min(x[2:5])
[1] 0
> which.min(x[2:5])
[1] 3
> sum(x)
[1] 44
> mean(x)
[1] 4.888889
```

Figure 1.5. Subvectors and Functions for Numeric Vectors

```
> a <- 2
> x <- c(1,2,3)
> y <- c(4,5,6)
> a+x
[1] 3 4 5
> a*x
[1] 2 4 6
> x+y
[1] 5 7 9
> a*x+y
[1]  6  9 12
```

Figure 1.6. Operations on Numeric Vectors

The associated functions `Re()` and `Im()` allow for simple manipulations and computations. Notice that some of the functions for operations on complex numbers begin with an upper-case letter.

```
> x1 <- 3
> class(x1)
[1] "numeric"
> z1 <- x1+2i
> class(z1)
[1] "complex"
> z2 <- complex(real=x1, imaginary=2)  #  Does the same thing; better.
> z1==z2
[1] TRUE
> x1*z1
[1] 9+6i
> z1+z2
[1] 6+4i
> x2 <- Re(z1)
> x1==x2
[1] TRUE
> mode(x1)
[1] "numeric"
> typeof(x1)
[1] "double"
> mode(z1)
[1] "complex"
> typeof(z1)
[1] "complex"
> mode(Im(z1))
[1] "numeric"
> typeof(Im(z1))
[1] "double"
```

Figure 1.7. Complex Data in R

R can do complex arithmetic. Complex objects are of class, mode, and type `complex()`. There are the two standard "`is.`" and "`as.`" functions for dealing with a complex object. In mathematics, the field of real numbers is a subfield of the complex numbers, but in R the numeric class is not a subclass of the complex class, and a numeric object will not be promoted to a complex object.

There are two built-in R functions for simple manipulations with complex numbers, `Mod()` for computing the modulus of a complex quantity and `Conj()` for computing the complex conjugate of a quantity. Figure 1.8 illustrates the use of complex objects in R.

Complex arithmetic arises in certain types of statistical analyses of time series. For example, analysis in the "frequency domain" of a time series focuses on the periodic behavior of the time series, and the standard models for the frequency are based on Fourier transforms, which involve complex arithmetic. The computational workhorse for Fourier transforms is the fast Fourier transform, or FFT. The R function `fft()` computes an FFT.

```
> x <- -4
> z <- sqrt(x)    # x is numeric, not complex.
Warning message:
In sqrt(-4) : NaNs produced
> z <- sqrt(as.complex(x))
> z
[1] 0+2i
> is.complex(z)
[1] TRUE
> is.numeric(z)
[1] FALSE
> y <- z*(2+2i) - 1i # Note the 1 before the i; otherwise, it is the variable i.
> y
[1] -4+3i
> Mod(y)
[1] 5
> sqrt(Re(y)^2+Im(y)^2)
[1] 5
> Conj(y)
[1] -4-3i
```

Figure 1.8. Complex Arithmetic in R

Most of the computations for analysis of financial data, indeed for statistical analysis generally, involve only real numbers. In most computations involving real numbers, all computational results remain in the real domain. The exception of course is when a root of a negative number is to be taken. This type of computation arises in the solution of a polynomial equation. In statistical analyses, the need to solve a polynomial equation arises occasionally in time series analysis, so we will encounter it briefly in Chapter 5. Solving a polynomial equation is also required in evaluating eigenvalues. A real matrix that is not symmetric may have non-real eigenvalues.

Character Data

Character data represent symbols rather than numbers. Character data are of class, mode, and type `character()`. Character data in R is composed of *character strings*, each consisting of a variable number of *characters*, each of which is encoded in a standard form in a fixed number of bits (usually 8).

Character data are initialized by enclosing the symbols within a pair of either single quotes or double quotes. Anything within the pair of quotes is considered a character. (A blank space is a character, and nothing is a character.) R displays character data within double quotes. A double quote itself is displayed with an escape character, which is a backward slash. Also, similar to the `as.complex()` function to convert numeric data to an object of

complex class as shown in Figure 1.8, there is an `as.character()` function to convert numeric data to a character object. Figure 1.9 illustrates some initializations of character data.

```
> a1 <- 'ab c'
> a2 <- "ab c"
> a1
[1] "ab c"
> a2==a1
[1] TRUE
> mode(a1)
[1] "character"
> typeof(a1)
[1] "character"
> b3 <- '"ab" "c"'
> b3
[1] "\"ab\" \"c\""
> b4 <- "'ab' 'c'"
> b4
[1] "'ab' 'c'"
> b3==b4
[1] FALSE
> j <- 3
> as.character(j)
[1] "3"
> as.character(-3)
[1] "-3"
> x <- 3.5
> as.character(x)
[1] "3.5"
```

Figure 1.9. Character Data in R

Because a character string is a single entity, the individual characters in the string are not accessible as they would be if they were part of a vector or some other array. The `nchar()` function determines the number of individual characters in a character string. The length of a single character string is 1 no matter how many individual characters are in the string.

R provides a number of functions for working with the individual characters in a character string.

The `paste()` function is useful for combining character strings into longer strings. The `paste()` function will insert a blank between each of the constituent strings, unless another separator is specified by `sep=`.

The R function `substring()` provides the opposite kind of operation from `paste()`. This function extracts the characters from a given starting position to a given ending position.

Arithmetic operations cannot be performed on character data. R provides a number of functions to manipulate character data, including string operations to work with substrings within a character string.

Figure 1.10 illustrates various operations with character data.

```
> c <- "c"
> nchar(c)
[1] 1
> b <- 'abcde'
> nchar(b)
[1] 5
> length(c)
[1] 1
> length(b)
[1] 1
> e <- paste(c,b)
> e
[1] "c abcde"
> nchar(e)
[1] 7
> f <- paste(c,b,sep="")
> f
[1] "cabcde"
> nchar(f)
[1] 6
> substring(e,1,2)
[1] "c "
> substring(f,2,4)
[1] "abc"
```

Figure 1.10. Operations on Character Data in R

If the ending position in `substring()` is beyond the last character, the function just gives the result up to the last character. If the beginning position is beyond the last character, the function just gives a character string with no characters. (As mentioned above, *nothing*, where a character is expected, is a character.)

```
> substring(c,1,2)
[1] "c"
> n <- substring(c,2,2)
> n
[1] ""
> nchar(n)
[1] 0
> class(n)
[1] "character"
```

The substring() function can also be invoked with two vectors of equal length. In this case, the function generates an output vector of character strings.

```
> b
[1] "abcde"
> d <- substring(b, c(1,1,3), c(2,nchar(b),5))
> d
[1] "ab"    "abcde" "cde"
> length(d)
[1] 3
```

Character strings can be combined into a character vector in the usual way. The length of a vector of character strings is the number of strings in the vector, and the number of characters in each string can be determined by the nchar() function in the usual way. As mentioned above, when a function that ordinarily has an atomic element as its argument is applied to a vector, the function is applied to each element in turn, yielding an output that is a vector; hence the nchar() function and the substring() function can to vectors of character strings. (See the comments on page 58 concerning vectorized functions.)

```
> c
[1] "c"
> b
[1] "abcde"
> v <- c(c,b)
> length(v)
[1] 2
> nchar(v)
[1] 1 5
> substring(v,1,2)
[1] "c"  "ab"
```

Figure 1.11. Operations on Character Data in R

Logical Data

Logical data represent *true* or *false*. Logical data in R are of class, mode, and type logical(). In arithmetic expressions in R, logical data also have the properties of numeric data, with a logical value of TRUE corresponding to a numeric value of 1 and a logical value of FALSE corresponding to a numeric value of 0, as illustrated on page 11.

1.2.5 Date Data

Dates can be represented in many ways, "January 2, 2020", "2 Jan 2020", "1/2/20", "2-1-2020", and so on. Most of these representations are immediately interpretable by humans, but some are not, such as "1/2/2020" and "2/1/2020", without a statement of the rule governing the format. In computer data processing of date data, some level of standardization is desirable, and unambiguous rules are necessary. Date data may also include the time of day. In that case, more rules governing all of the fields are necessary.

There are various functions in R for determining the current date and time and for basic operations on the date data. Most of these functions are based on underlying Unix functions for working with times and dates.

Often, only the date is required, and the `as.Date()` function in the `base` R package is sufficient. This function creates an object of class `Date`. Dates, of course, can be represented in a number of ways, such as "10/19/1987", "19/10/1987", or "October 19, 1987" for the same date, so long as we know the format.

In some applications more expressive methods of representing dates and time are required, and more options for manipulating date date are needed. The `date()` function in the `lubridate` package is a convenient way of assigning date data. It also creates an object of class `Date`.

The International Standard ISO 8601 specifies representation of a date in the form "1987-10-19" (but also allows the form "19871019"), and the Portable Operating System Interface (POSIX) standards adopted by the IEEE Computer Society uses dates in this general format. The ISO 8601 format also allows for date plus time in hours, minutes, and seconds. The full format is *yyyy-mm-dd hh:mm:ss*. All fields have a fixed number of integer characters except "ss", which can have an appended decimal fractional part. The fields must be filled from the left, and the last field on the right indicates the full time period specified; for example, "1987-10" refers to the full month of October, 1987. I will generally refer to this form of representation, or most often, just the date portion of this, as the "ISO 8601 format" or as a "POSIX format".

External data files to be read into R often have dates in non-POSIX formats. One way of dealing with such dates is to treat them as character data and then use `as.Date()` or some other R function to convert them to POSIX format; see page 131, for example.

The `as.Date()` function accepts dates in this format as default, but also allows dates in other formats, as shown in Figure 1.12, for example. Some of the formats for `as.Date()` are shown in Table 1.1. The formats can also be used in the `format()` function for printing dates in different forms (again, see Figure 1.12), and can be used in other functions that operate on dates, such as `strftime()`.

Date data created by `as.Date()` or by `date()` in `lubridate()` is of class `Date`, of mode `numeric` even though it appears as character data, and of type `double` (in most environments). Internally, a date is represented as the number

Table 1.1. Some Date Formats for R Functions

Code	Meaning
%d	Day of the month (decimal number)
%m	Month (decimal number)
%b	Month (3-letter abbreviation)
%B	Month (full English name)
%y	Year (2 digits; breaks between 68 and 69)
%Y	Year (4 digits)
%j	Day of the year

of days before or after January 1, 1970. If the date is before January 1, 1970, the number is negative; if after, it is positive. January 1, 1970, is represented as 0. The `format()` argument in various functions that operate on date data specifies the format in which the date is represented for interaction with the user, but it does not change the internal representation. The `base` R package also provides utility functions, such as `weekdays()` for working with objects of class `Date`. The R function `strftime()` is useful for getting dates into different formats. A simple example of `strftime()` operating on a given input date is to determine the number of the day within the year; January 1, is day 1; December 31 is 365 unless the year is a leap year, in which case it is 366.

Figure 1.12 illustrates some of the properties and operations, using only the `base` package.

```
>  x <- as.Date("Oct 24, 1929",format="%b %d, %Y")
>  class(x)
[1] "Date"
>  mode(x)
[1] "numeric"
>  format(z, "%b %d, %Y")
[1] "Oct 19, 1987"
> weekdays(x)
[1] "Thursday"
> x
[1] "1929-10-24"
> strftime(x, format="%j")
[1] "297"
> strftime("2017-12-31", format="%j")
[1] "365"
> strftime("2020-12-31", format="%j")
[1] "366"
```

Figure 1.12. Examples of Date Data and Functions to Work with It

The base package of R also includes two functions, as.POSIXct() and as.POSIXlt(), that handle time as well as dates. The POSIXct and POSIXlt classes also carry information about time zones and daylight savings time.

There are also a number of other packages for working with dates and times. Three commonly used packages are timeDate, chron, and lubridate. I use lubridate most often. The date() function in lubridate requires an input in a POSIX format.

```
> library(lubridate)
> xd <- date("1929-10-24")
> xd
[1] "1929-10-24"
```

Many of the packages for working with time series, such as those discussed in Chapters 2 and 5, also provide functions for dealing with dates and times.

More useful functions and operators for handling date data are provided in various R packages. The date objects must conform to specific rules for representation of the date.

Operations on Date Data

Date data of class Date are of mode numeric (recall, internally, they are the number of days before or after January 1, 1970), and numerical operations can be performed on date data. Numeric data can be added to Date data (but not multiplied with it). Addition of numeric quantities to a Date object results in a Date object of the appropriate value. The operators are aware of leap years and other aspects of the Gregorian calendar.

```
> as.Date("2016-02-28")+1
[1] "2016-02-29"
> as.Date("2016-02-28")+0:3
[1] "2016-02-28" "2016-02-29" "2016-03-01" "2016-03-02"
> as.Date("2016-02-28")-0:3
[1] "2016-02-28" "2016-02-27" "2016-02-26" "2016-02-25"
```

Objects of class Date can be "subtracted" from each other, but not added to each other. The binary "-" infix operator is used to determine the length of time from one date to another. The result is number of days from the first date to the second, which is negative if the second date precedes the first. When an object of class Date is subtracted from another, an object of class difftime is created.

The seq() function works with Date data. The by() parameter in seq() can be "days", "weeks", "months", or "years". The colon sequence operator ":" produces a sequence in days using the internal representation, that is, the

numbers of days before or after January 1, 1970. For dates used as indexes ranges within the set of indexes can be indicated by "/".

Figure 1.13 illustrates some operations, using only the `base` package.

```
#    "Black Thursday"
>  x <- as.Date("Oct 24, 1929",format="%b %d, %Y")
#    "Black Tuesday"
>  y <- as.Date("29 October 1929",format="%d %B %Y")
#    "Black Monday"
>  z <- as.Date("10/19/87",format="%m/%d/%y")
> weekdays(c(x,y,z))
[1] "Thursday" "Tuesday"  "Monday"
> x
[1] "1929-10-24"
> x+5
[1] "1929-10-29"
>  x-y
Time difference of -5 days
> y
[1] "1929-10-29"
>  class(x-y)
[1] "difftime"
>  z-x
Time difference of 21179 days
> seq(x, length=6, by="weeks")
[1] "1929-10-24" "1929-10-31" "1929-11-07" "1929-11-14"
[5] "1929-11-21" "1929-11-28"
> seq(from=x, to=y, by="days")
[1] "1929-10-24" "1929-10-25" "1929-10-26" "1929-10-27"
[5] "1929-10-28" "1929-10-29"
> x:y
[1] -14679 -14678 -14677 -14676 -14675 -14674
```

Figure 1.13. Examples of Date Data and Functions to Work with It

Date Data as Indexes to Arrays

An important application of date data in working with financial data is its use as an index to arrays of data. Much of financial data are time series. Each data item or observation corresponds to a specific date.

In the examples of atomic vectors we have considered above, the indexes are just positive integers. In many time series, integers are used as indexes just as in simple vectors, and this is generally acceptable it the dates are evenly spaced and we have a mapping from actual dates to the integers.

If the elements in a time series are not equally spaced in time, use of simple integers as indexes may not be satisfactory. In addition, it may be awkward to

determine the integer index that corresponds to a given date in the ordinary year, month, and day format.

There are some common R data structures that use dates in POSIX form as indexes. This allows for easy manipulation of the data at specific dates or within specific ranges of dates. Date ranges in indexes can be indicated by "/"; for example, "198710/19871120" represents October 1 through November 20, 1987. We will describe and illustrate use of dates as indexes in R arrays in Section 1.3.6.

1.2.6 Missing Values

Occasionally, in the course of performing numerical computations, a value results that does not correspond to an ordinary number and cannot be represented in the ordinary way. A computation such as $1/0$ yields a result we might designate as ∞ and characterize by certain properties such as $\infty + 1 = \infty$, $2\infty = \infty$, $-2\infty = -\infty$, and so on. The result of a computation such as $0/0$ has no meaningful properties. The standard way that computer systems deal with these special values is to designate them as special objects that follow the appropriate rules of arithmetic. The object similar to ∞ is usually called "`Inf`" and the meaningless object is usually called "`NaN`" ("not a number"). The `NaN` object is not a value, so if an object has a missing value and `NaN` has been assigned to it, it is not equal to `NaN`. R provides a special function `is.nan()` to test whether a value is a `NaN`.

```
> x <- 1/0
> x
[1] Inf
> x+1
[1] Inf
> -2*x
[1] -Inf
> y <- 0/0
> y
[1] NaN
> y==NaN
[1] NA
> is.nan(y)
[1] TRUE
> y+1
[1] NaN
> -2*y
[1] NaN
> x+y
[1] NaN
```

Often in financial analyses a data item cannot be given a value consistent with its ordinary meaning. The PE ratio of the stock of a company whose earnings are exactly 0 would be ∞, and one with a loss would be some finite

negative value. In financial datasets these unusual PE values are often reported just as 0. Other times these PE values are just left blank, that is, they are just missing.

Missing values can arise in financial datasets for a variety of reasons. In financial applications, data may be missing for various reasons, including simple failures to report data or changes in definitions of certain quantities. In company reports or in survey data, values may be missing because the respondent did not provide a response for a particular field.

These missing values are not the result of extreme or meaningless mathematical computations such as 0/0.

R provides a very simple way to handle values that are not available. A special logical constant called `NA` is assigned to a missing value. Statistical computations on datasets with some `NA` items can be handled in various ways. Sometimes it is appropriate to declare that the computations cannot be performed and so the results are missing; in other cases, such as computing the mean of a set of numbers, it may be appropriate to report the mean of all of the non-missing numbers.

The single constant `NA` is generally sufficient for all instances of missing or undefined values, but additional constants such as `NA_integer_`, `NA_characters_`, and so on are also available if it is desirable to distinguish the data types. The numerical constants `Inf`, `NaN`, and so on are also sometimes appropriate. A `NaN` is `NA`, but an `NA` is not necessarily `NaN`.

Computer manipulations with objects that are missing obviously should be performed with some care. The facilities provided by R for dealing with missing values should be utilized. The `NA` object is not a value, so if an object has a missing value and `NA` has been assigned to it, it is not equal to `NA`. R provides a special function `is.na()` to test whether a value is missing.

```
> z <- NA
> z==NA
[1] NA
> z <- NA
> z==NA
[1] NA
> z==z
[1] NA
> is.na(z)
[1] TRUE
> is.nan(z)
[1] FALSE
> y <- 0/0
> is.na(y)
[1] TRUE
> is.nan(y)
[1] TRUE
```

Often when data contain missing values, we wish to ignore those values and process all of the valid data. Many functions in R provide a standard way

of requesting that this be done, by means of the logical argument `na.rm()`. If `na.rm()` is true, then the function ignores the fact that some data may be missing, and performs its operations only on the valid data.

```
> xx <- c(1,2,NA,3)
> mean(xx)
[1] NA
> mean(xx, na.rm=TRUE)
[1] 2
```

In some cases where data are missing, we may wish to impute values by using the available data to estimate or predict the values that are missing.

1.2.7 Working Environment

Within an R session various packages may be loaded and various objects may be initialized within the working environment of the program. When the session is ended, the environment in which these items exist no longer exists. The items themselves,however, may be saved in computer files so that they can be reinstated in a subsequent R session.

Environments

A set of R statements may consist of separate modules, such as the statements forming functions and the other statements that may invoke those functions. A group of statements that make up an R function may introduce variables that are local to the function; that is, these variables with their assigned values exist only within the scope of the function definition. For example, in the function `myFun` defined in the R statements shown in Figure 1.1 on page 17, there are two variables, `slope` and `intercept` that are introduced and assigned values within the function. In other R statements that may be executed in the same session as when this function is initiated or invoked, the existence or the values assigned to these two variables are unchanged.

An R *environment* consists of a collection of objects, and it also may refer to its parent environment. A name of an object refers to the object in the environment that has that name, or else if no object with that exists within the environment, it refers to the object in the parent environment with that name.

At any point in an R session there is a hierarchy of environments, and names of objects have meanings that are resolved by going back through the hierarchy of environments. The name of an object instantiated within a given environment always refers to the object in that environment.

There is always a global environment. Other environments may depend on what packages have been loaded. The environments can be determined using

```
> search()    # the output depends on the current R session
 [1] ".GlobalEnv"        "package:stats"     "package:graphics"
 [4] "package:grDevices" "package:utils"     "package:datasets"
 [7] "package:methods"   "Autoloads"         "package:base"
```

Figure 1.14. A Search List

the `search()` function. This function displays the search list that is used in resolving object names. An example is shown in Figure 1.14.

The order in this list is the order in which objects (primarily functions) will be identified. For example, if there is a function in the `graphics` package with the same name as a function in the `stats` package, the one in the `stats` package is the one that will be used. The search list is the reverse of the order in which the packages were loaded.

A package may be unloaded by the `detach()` function.

Names within an environment, or "namespace", must be unique. The same name can be used for different objects in different environments, however. An object can be referenced by the name of its environment, followed by "::" followed by the name of the object within that environment, `env::obj`. Figure 1.15 shows an example with an object called `t`, which is also the name of a function in the R `base` package that computes the transpose of a matrix. In this example, `t` is defined as a function to add 5 to its argument. (This would not be a good idea! Nevertheless, look at the example.) Standing alone, "`t`" refers to this new function. The `t` object in the `base` package can be referenced as "`base::t`". *It is recommended that user-written functions have more descriptive and generally longer names.*

```
> t <- function(x) return(x+5)
> x <- matrix(c(1,2,3,4),nrow=2)
> t(x)
     [,1] [,2]
[1,]    6    8
[2,]    7    9
> base::t(x)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

Figure 1.15. t as a User Function and as a Built-in Function

The example function `myFun` shown in Figure 1.1 establishes an *environment* in which the local variable names `slope` and `intercept` have meaning. The names `x1`, `y1`, `x2`, `y2`, and `x` are *formal arguments* to the function. They are assigned a value when the function is invoked, and then are used within the function. The names of the variables in the environment in which the function is invoked do not have to be the same. Any changes made to them within the function are not passed back to the environment in which the function was invoked.

The function, however, inherits variables from the environment in which it was invoked. The function `myFun1` shown in Figure 1.16 does the same thing as the `myFun` function if the variables `x1`, `y1`, `x2`, and `y2` exist in the environment in which the function is invoked. *Omitting the formal arguments and using object names assumed to exist in the environment in which a function is invoked is not recommended.*

```
myFun1 <- function(x) {
#  Given the points (x1,y1) and (x2,y2) and a set of abscissas x,
#  determine the ordinates at x for the line that goes through all of them.
    slope <- (y2-y1)/(x2-x1)
    intercept <- y1 - slope*x1
    y <- slope*x + intercept
    return(y)
}
```

Figure 1.16. A Simple Function in R; Compare Figure 1.1()

The objects within a specific environment can be listed by the `ls()` function. The function without any arguments returns a list of the objects created in the current environment; hence if it is invoked within a function, it returns a list of the objects created within the function. It can also give a list of the objects in a specified environment, so it is useful for determining the names of functions within a given package. The package must be loaded, of course. (Note in the example below that `datasets` is generally included with the basic R system.)

The `list.files()` function is the same as the common operating system `ls()` command, and lists the files in the specified directory, or in the working directory by default (see page 42). Notice that `ls()` refers to an R environment, while `list.files()` refers to an environment of the computer operating system. Figure 1.17 illustrates the use of `ls()` and `list.files()`.

Occasionally, some part of the name of an R function is known. For example, we may know that there is an R function that rounds to a specific number of significant figures. We vaguely remember the name of the function to be

```
> ls()    #  the output depends on the current R session
  [1] "a"                       "A"
  [3] "x1"                      "x2"
  [5] "y"
> ls("package:datasets")
  [1] "ability.cov"           "airmiles"
  [3] "AirPassengers"         "airquality"
  [5] "anscombe"              "attenu"
       ... etc. ...
> list.files()
 [1] "grSF10108.ps"   "grSF10109.ps"   "grSF10110.ps"   "grSF10111.ps"
 [5] "grSF10112.ps"
> list.files("../Web/Data")
 [1] "DJId.csv"        "GSPC1990.csv"    "GSPCd.csv"       "INTCd.csv"
 [5] "INTCd20173Q.csv"
```

Figure 1.17. The `ls()` and `list.files()` Functions

"`sig...`". We can search for the function by listing all of the functions in a given package with a specific character string in the name. Rounding is a basic activity, so we might just search within the `base` package for all functions that have "`sig...`" in their names:

```
> ls("package:base", pattern = "sig")
[1] "assign"          "delayedAssign"   "psigamma"        "sign"
[5] "signalCondition" "signif"
```

From this list of objects that contain the pattern "`sig`" in their names, we rightly conclude that the function of interest is `signif()`. Hence, we can use `?signif` to see exactly how this function works.

There are also some functions in R for using objects from different environments. The one most commonly used perhaps is `get()`, which returns an object from a different environment to the current environment.

Workspace

The R workspace is the current environment in a session. It contains all current objects and settings.

The current environment can be saved for use in a subsequent R session by use of the `save.image()` function. This creates a "workspace image" of the R session that contains the current values of all defined objects at that point in time. An R user will often do this at the end of the session. The R system even prompts the user to do this when the program is exited. The image of the environment will be saved in a designated file with default filename extension

.RData. If no file is specified, it is stored in the file
.RData in the current working directory (see below). Later, either in the same session or in a different session even on a different compatible computer, the workspace can be restored by the `load()` function. This is useful when working on a given project over multiple sessions. Different project can have different saved workspace images.

The `save()` function in R saves specified objects that can be retrieved in a later session of R, even on a different computer, using the `load()` function.

Working Directory

An R session has a *working directory*. The path to any external file begins in this directory, so if any external files are to be read or written, it is convenient to set the working directory to the directory where most of the external files will reside. Different working directories can be used for different projects.

The working directory is set by the function `setwd()`. The function `getwd()` without an argument displays the working directory.

A directory path in R is specified using "/" as the separator of directory names, as in Unix/Linix. (In Microsoft Windows, the usual separator is "\".) In Microsoft Windows, for example,

```
> setwd("c:/Books/Data/")
> getwd()
[1] "c:/Books/Data"
```

Notice that names of external objects, such as files, are character objects. (This means that the literals are enclosed in quotation marks.) Also, note that the last "/" in the path is not necessary. Use of a character variable may allow more flexibility. An alternative to the statement `setwd("c:/Books/Data/")` are the statements

```
maindir <- "c:/Books/"
wkdir <- paste(maindir,"Data",sep="")
setwd(wkdir)
```

Packages

A group of functions written either in R or in a compiler language can be packaged together in a "library" or "package", and then all of the functions, together with associated documentation, can be loaded into R just by loading the package.

To use a package, first *install* it on the R system being used, and then *load* it in the current R session.

Previously developed packages can be installed with the local system, and then can be loaded in an R session. There are many R packages stored in the Comprehensive R Archive Network (CRAN). An R system program, such as `install.packages()`, can install them on a local system, and they can be loaded in an R session by the `library()` function. The `install.packages()` function requires that the package name be a character string, but the `library()` function does not require that, although it allows it.

```
install.packages("vars")
library(vars)
```

The main window in the standard R GUI has a menu item for "Packages", which allows both for installing and loading packages. The "Load package" selection gives a list of all installed packages on the system.

Packages from other repositories can also be installed with the `install.packages()` function using the `repos` argument. For example, to install a package from

```
install.packages("XYZ", repos="http://r-forge.r-project.org"))
```

(XYZ is not a package, at least not at this time.)

Installing packages from sources other than CRAN may not be straightforward for a number of reasons. Although there are some useful and well-developed packages at the other sites, some of the packages at the other sites may not be as robust as those at CRAN.

There is an R function `install_github()` in the `devtools` package to install packages from GitHub. This function requires the username, that is, the name of the GitHub user account, as well as the name of the package. Alternatively, there is a `githubinstall` package available from CRAN that contains functions to search GitHub accounts for projects with names similar to a specified character string. Once the name of a package (project or directory) is known, the `gh_install_packages()` function can be used to install it, given only the package name; that is, the has a similar interface to that of `install.packages()`. The `githubinstall()` function in the `githubinstall` package is an alias for `gh_install_packages()`.

The R system generally consists of a set of packages, a "`base`" package, a "`stats`" package, and so on. When the R program is invoked, a standard set of packages are loaded. Beyond those, the user can load any number of other packages that have been installed on the local system.

Within an R session, the available functions include all of those in the loaded packages. Sometimes, there may be name conflicts (two functions with the same name). The conflicts are resolved in the order of the loading of the packages; last-in gets preference.

As mentioned above, the `ls()` function can be used to determine the functions included in a package, so the functions in the `vars` package, after it is loaded could be determined by

```
ls("package:vars")
```

External Data Connections and Filenames

Storing or manipulating data or performing computations on the data on different computer systems ("platforms") can present problems because of the differing formats of storage or differing programming paradigms. We call these protocols an "application programming interface" or "API". There are many standard APIs that facilitate manipulating data on different systems.

The Open DataBase Connectivity, or ODBC, is a standard application programming interface for accessing database management systems. The R package `RODBC` provides the ODBC facility.

One of the most widely-used database management systems is MySQL, which is an open-source relational database management system (DBMS) built on SQL, the Structured Query Language. Most professionals in the information processing field are familiar with some SQL system, and so for them, MySQL is simple to use, although the functionality may not be exactly the same. In any event, most of the functionality of an SQL DBMS can be accessed through the R package `RMySQL` that provides direct connections to MySQL.

It is common for programs, especially operating systems, to associate file types with filename extensions, often consisting of three characters; for example, a file named "book.tex" is probably a TEX file and a file named "book.pdf" is probably a PDF file. R generally does not distinguish the file type by an extension. Though a file named "Mydata.csv" is probably a CSV file, the R functions dealing with it cannot omit the extension.

Most functions in R data are designed to ignore any meaning that may be conveyed by a filename extension, so the user must specify full filenames.

Exercises: Program Structure and Syntax

1.2.1. For values that naturally form a sequence or else have some simple structural relationship to each other, the use of an R array to store and retrieve those values is generally appropriate, for example,

```
> x <- c("d","c","b","a")
> x[2]
[1] "c"
```

Occasionally, however, we may wish to form names of R objects that express their order in a sequence or else their value.
Assume that we have the R object `x` as above.

a) Write an R expression that creates R variables of the form `xd`, `xc`, and so on, using `x`, where `xd="d"`, `xc="c"`, and so on.
b) Write an R expression that creates R variables of the form `x1`, `x2`, and so on, using `x`, where `x1=x[1]`, `x2=x[2]`, and so on.

1.2.2. In the following, write simple R expressions. Although extraneous parentheses often are useful for improved clarity, do not use any extraneous parentheses in your solutions.

a) Write an R expression that yields the sequence of even integers from 2 to 20.
b) Write an R expression that yields the sequence of odd integers from 1 to 19.
c) Write an R expression that yields the sequence of the squares of odd integers from 1 to 19; that is, 1, 9,....
d) Write an R expression that yields the sequence of the square roots of odd integers from 1 to 19; that is, 1, 1.732051,....
e) Write an R expression that yields the sequence of halves from 0.5 to 20.0; that is, 0.5, 1.0, 1.5,....

1.2.3. Write an R function, `evens`, that yields the sequence of even integers between and including `x1` and `x2`.
For example, if `x1=-2.5` and `x2=4.5`, the function returns -2, 0, 2, 4.
If `x1` is greater than `x2`, the function returns NULL.

1.2.4. Factors.

a) Suppose we have a factor variable that is initialized by the statement
```
xf <- factor(c("d","c","b","a"))
```
Write an R expression that yields the third value in `xf`. Note that the value is a character.
b) Suppose we have a factor variable that is initialized by the statement
```
yf <- factor(1:4)
```
Write an R expression that yields 7 times the third value in `yf`. (The value is 21.)

1.2.5. Often in producing graphs or other output in R, we wish to include some text that specifies the value of a variable. If the variable is numeric, 500 for example, we need the character representation of the numeral, `"500"` for example. If the variable is a factor, we need the character representation of the factor level.
For example, we may want to print the phrase

The mean of 500 random numbers is 5.33.

where 500 is a quantity used in the program, and may change on another run, and 5.33 is the computed mean rounded to two decimal places.
Write an R character string expression that corresponds to the text
"The mean of 500 random numbers is 5.33."
but allows the number and the computed mean to be variable, depending on the R variables `n` and `xm` in the program.
Let `n=500` and `xm=5.333333`.
Use the `print()` function to display the character string.

1.2.6. Shiny apps.
Write a Shiny app that yields the sequence of even integers between and including `x1` and `x2`, as in Exercise 1.2.3.

1.2.7. Missing values.
In some applications, especially in survey sampling, a missing value is coded as a "0".
Write an R statement that codes the value of the variable `x` as 0 if `x` is missing, either an NaN or an NA.

1.3 R Objects and Classes

Each entity, or *object*, in R belongs to a specific *class* that determines the general characteristics of the entity and the *methods* or computations that can be performed on the object. An object in R also has a *mode* and a *type*, which determines more fundamental properties of the object.

In Section 1.2.4, we encountered various classes of data, such as the `numeric` class and the related classes of `integer` and `complex`, the `character` class, and the `Date` class. R has a number of other built-in object classes, such as `function`, which we have also encountered.

In this section, we will discuss additional built-in object classes that provide structure to data, such as `matrix`, `table`, `list`, and `data.frame`. Many of the packages from CRAN define additional classes, and the user also can define new classes.

Classes

We have used the `class()` function to determine the class of an object. For the main built-in classes such as `numeric`, `matrix`, and so on, there are three associated functions, a constructor function, a conversion function (an "`as.`" function), and a test function (an "`is.`" function). There are also "`as.`" and "`is.`" functions for some types of data, such as `double`. We saw examples of the `as.complex()` function in Figure 1.8 and the `as.character()` function in Figure 1.9.

```
> y1 <- 4
> y2 <- as.integer(y1)
> y3 <- as.character(y1)
> y4 <- as.complex(y1)
> y5 <- sqrt(y4)
> is.integer(y1)
[1] FALSE
> is.integer(y2)
[1] TRUE
> is.integer(y3)
[1] FALSE
> is.integer(y4)
[1] FALSE
> is.integer(y5)
[1] FALSE
> is.complex(y5)
[1] TRUE
```

Figure 1.18. Converting and Testing Classes and Types

The "`as.`" function does not change the class of its argument. The "`as.`" and "`is.`" functions are illustrated in Figure 1.18.

There are three different kinds of classes and methods that apply to objects in R. The two most commonly used are called "S3" or "S4". We will not consider the differences in the types of classes in this book, but rather refer the interested reader to Chambers (2008, 2016) or Wickham (2019).

1.3.1 Arrays

Proper organization of data allows for efficient access and processing. Organization and access of distributed external databases can present significant challenges, but organization of data within a single computer system is fairly straightforward. A very simple, yet general basic organization is as a rectangular (or "hyperrectangular") array. A one-dimensional array is just an ordered set, a two-dimensional array corresponds to a two-dimensional rectangular table, and higher-dimensional arrays correspond to multi-way tables.

The elements in an array may be of any class, mode, or type. In the basic array structure, only one type is allowed. The simplest example of a one-dimensional array is an atomic vector, which we discussed above. An atomic vector has the same class as the elements in the array, but an array is of class `array`. An array is constructed by the R function `array()`.

The R statements below produce the same results as those for atomic vectors on page 20, except that the objects produced here are of class `array` instead of class `numeric` and `character` as in the former case. The operations

on the arrays, however, depend on the class, mode, and type of the individual elements.

```
> x1 <- array(c(9,8,7,6,5,4,3,2,1))
> x1
[1] 9 8 7 6 5 4 3 2 1
> class(x1)
[1] "array"
> a1 <- array(c("a", "bc", "d"))
[1] "a"  "bc" "d"
> class(a1D)
[1] "array"
> xsq1 <- c(9,4,1)
> sqrt(xsq1)
[1] 3 2 1
```

An array can have any number of dimensions. The numbers of elements are specified in the `dim` argument in the `array()` function. Data are cycled through the first dimension first, the second dimension next, and so on. An array of data can be reshaped by the `dim` argument.

```
> xxx <- 1:12
> x34 <- array(xxx, dim=c(3,4))
> x34
    [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> x43 <- array(x34, dim=c(4,3))
> x43
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

Reshaping a data array is a fundamental operation in data management and analysis, and we will encounter other, more complicated instances of reshaping data later.

The `dim()` function returns a vector whose elements are the lengths of the dimensions. *The* `dim()` *function requires that its argument be of class* `array`, or at least be an object that inherits from the `array` class. The function returns `NULL` for an atomic array. The `length()` function returns the total length; that is, the total number of values in all dimensions.

```
> x2D <- array(1:12, dim=c(3,4))
> x2D
    [,1] [,2] [,3] [,4]
```

```
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> dim(x2D)
[1] 3 4
> length(x2D)
[1] 12
> x3D <- array(1:12, dim=c(2,3,2))
> x3D
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
> dim(x3D)
[1] 2 3 2
> length(x3D)
[1] 12
```

The `head()` and `tail()` functions are useful for obtaining the first few elements of a large array. The functions are particularly useful for one- or two-dimensional arrays. The number of items to return can be specified with the `n` argument. In the case of a two-dimensional array, for example, `n` determines number of lines (the first dimension) to return.

```
> y1D <- array(1:10000)
> head(y1D)
[1] 1 2 3 4 5 6
> tail(y1D)
[1]  9995  9996  9997  9998  9999 10000
> y2D <- array(1:10000, dim=c(2500,4))
> head(y2D, n=4)
     [,1] [,2] [,3] [,4]
[1,]    1 2501 5001 7501
[2,]    2 2502 5002 7502
[3,]    3 2503 5003 7503
[4,]    4 2504 5004 7504
> tail(y2D, n=4)
        [,1] [,2] [,3]  [,4]
[2497,] 2497 4997 7497  9997
[2498,] 2498 4998 7498  9998
[2499,] 2499 4999 7499  9999
[2500,] 2500 5000 7500 10000
> y3D <- array(1:10000, dim=c(500,5,4))
> head(y3D)
[1] 1 2 3 4 5 6
> tail(y3D)
```

```
[1]  9995  9996  9997  9998  9999 10000
```

While arrays generally are useful, two special cases stand out, because they correspond to the mathematical objects, vectors and matrices, which we will discuss from that standpoint in Section 1.3.2.

Indexing and Subarrays

An array has an index for each dimension. As with vectors, the indexes are enclosed in "`[ ]`", and separated by commas. The indexing in each dimension uses the positive integers from 1 to the range of the dimension. Again, as with vectors, negative values can be used to indicate omission of values. Positive and negative values cannot be combined in the same dimension, but they can be used in the same expression in different dimensions.

The R statements below for the `x1` array yield the same values in objects as illustrated beginning on page 20, except that the objects produced below are of class `array` instead of class `numeric` as in the former case.

```
> x1 <- array(c(9,8,7,6,5,4,3,2,1))
> x1[1]
[1] 9
> x1[c(3,1,3)]
[1] 7 9 7
> x1[c(-1,-3)]
[1] 8 6 5 4 3 2 1
> length(x1[c(3,1,3)])
[1] 3
> length(x1[c(-1,-3)])
[1] 7
```

For higher dimensional arrays, the indexing consists of a vector of the positions in the respective dimensions. The index for each dimension may be a vector.

```
> x2D <- array(1:12, dim=c(3,4))
> x2D[2,1]
[1] 2
> x3D <- array(1:12, dim=c(2,3,2))
> x3D[2,1,2]
[1] 8
> x3D[2, c(2,3), 2]
[1] 10 12
> x3D[2, c(2,3), -2]
[1] 4 6
> x3D[1:2, c(2,3), 1:2]
, , 1
```

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

, , 2

     [,1] [,2]
[1,]    9   11
[2,]   10   12
```

If the subarray consists of all elements in a given dimension, the index itself can be omitted; the comma indicating its position in the expression must be given, however.

```
> x3D[, c(2,3), ]
, , 1

     [,1] [,2]
[1,]    3    5
[2,]    4    6

, , 2

     [,1] [,2]
[1,]    9   11
[2,]   10   12

> x3D[2, c(2,3), ]
     [,1] [,2]
[1,]    4   10
[2,]    6   12
```

If the number of elements in a given dimension is reduced to 1, that dimension is effectively eliminated. The R operators on arrays remove such a dimension; that is, the number of dimensions in the object is reduced. This operation is called "downcasting". The class of the subarray object may also not be `array`. The class of a three-dimensional array reduced to two dimensions becomes of class `matrix` (which is effectively still of class `array`. A two-dimensional array reduced to one dimension is no longer of class `matrix` or of class `array`. The `dim()` function does not work as we might expect on a one-dimensional array.

```
> x3D[, c(2,3), ]        #  Generates a 3-D object
, , 1

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

```
, , 2

     [,1] [,2]
[1,]    9   11
[2,]   10   12
> class(x3D[, c(2,3), ])
[1] "array"
> dim(x3D[, c(2,3), ])
[1] 2 2 2

> x3D[2, c(2,3), ]       #  Generates a 2-D object
     [,1] [,2]
[1,]    4   10
[2,]    6   12
> class(x3D[2, c(2,3), ])
[1] "matrix"
> dim(x3D[2, c(2,3), ])
[1] 2 2

> x3D[2,c(2,3),2]      #  Generates a 1-D object
[1] 10 12
> class(x3D[2, c(2,3), 2])
[1] "integer"
> dim(x3D[2, c(2,3), 2]) #  The vector is not an array
NULL
```

This behavior of the `dim()` *function is often a source of errors for R programmers.*

The original structure can be retained in the new object obtained by this kind of subsetting by use of the `drop` in the subsetting operation.

```
> x3D[2, c(2,3), 2, drop=FALSE]  #  Generates a 3-D object
, , 1
     [,1] [,2]
[1,]   10   12
> class(x3D[2, c(2,3), 2, drop=FALSE])
[1] "array"
> dim(x3D[2, c(2,3), 2, drop=FALSE])
[1] 1 2 1
```

Both the rows and the columns of a two-dimensional array can be given names, and then elements of the array can be addressed using the names as indexes. The names of rows are assigned by the `rownames()` function and the names of rows are assigned by the `colnames()` function. These two functions can also be used to determine what names have been assigned to the rows and columns. The ordinary integral indexes can also be used to address the elements, in the usual way.

```
x2D <- array(1:12, dim=c(3,4))
> rownames(x2D) <- c("A", "B", "C")
```

```
> colnames(x2D) <- c("1st", "2nd", "3rd", "4th")
> x2D
  1st 2nd 3rd 4th
A   1   4   7  10
B   2   5   8  11
C   3   6   9  12
> x2D["B", ]
1st 2nd 3rd 4th
  2   5   8  11
> x2D[2, ]
1st 2nd 3rd 4th
  2   5   8  11
> x2D["B", "3rd"]
[1] 8
```

Matrices

An array with two dimensions is called a "matrix" in R, and in most cases, it is of class `matrix`. This term and the class are used even if the object does not correspond to a mathematical matrix, for example if the elements are character strings. (In mathematics, *matrix* is an object whose elements are members of a field, such as $\mathbb{R}$.)

Because the two-dimensional array is so commonly used, there is an R function `matrix()` to construct a matrix. In the `matrix()` function only the number of rows or number of columns need to be specified. The data elements are specified by column unless the `byrow` argument is TRUE.

```
> A <- matrix(c(1,5,9,2,6,10,3,7,11,4,8,12), nrow=3)
> A
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
> B <- matrix(c(1,2,3,4,5,6,7,8,9,10,11,12), ncol=3, byrow=TRUE)
> B
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

A matrix can also be constructed by binding vectors or matrices as the columns of the matrix (the `cbind()` function) or by binding vectors or matrices as the rows of the matrix (the `rbind()` function).

There are slight differences in the matrices produced by `cbind()` or `rbind()` using vectors and those produced by use of `matrix()`, as `A` and `B` above. The matrices produced by `cbind()` have names of the constituent vectors for its columns and those produced by `rbind()` have names of the

constituent vectors for its columns. If matrices are bound to form a new matrix, that matrix gets the names of rows and columns of the corresponding matrices whose rows and columns have names. The names, which are characters, can be used to access elements in the matrix. (Neither the row names nor the column names must be unique. If there are duplicate row names, the first one is assumed; likewise, for column names.)

```
> r1 <- c(1,2,3,4); r2 <- c(5,6,7,8); r3 <- c(9,10,11,12)
> c1 <- c(1,5,9); c2 <- c(2,6,10); c3 <- c(3,7,11); c3 <- c(4,8,12)
> AA <- rbind(r1, r2, r3)
> BB <- cbind(c1, c2, c3, c4)
> AA
   [,1] [,2] [,3] [,4]
r1    1    2    3    4
r2    5    6    7    8
r3    9   10   11   12
> rownames(AA)
[1] "r1" "r2" "r3"
> colnames(AA)
NULL
> BB
     c1 c2 c3 c4
[1,]  1  2  3  4
[2,]  5  6  7  8
[3,]  9 10 11 12
> AA["r2", ]
[1] 5 6 7 8
> AA[2, ]
[1] 5 6 7 8
> AA["r1", 2]
r1
 2
> AA[, r1]
   [,1] [,2] [,3] [,4]   # {it Note r1}
r1    1    2    3    4
r2    5    6    7    8
r3    9   10   11   12
> cbind(AA, BB[1:3, ])
              c1 c2 c3 c4
r1 1  2  3  4  1  2  3  4
r2 5  6  7  8  5  6  7  8
r3 9 10 11 12  9 10 11 12
```

Figure 1.19. Simple Manipulations of Matrices

Matrices can be subsetted in the same way as other arrays, using the indexes. The `dim` keyword is not directly applicable to matrices (although in can be used when a matrix is an argument of the `array()` function), but

of course the `nrow` or `ncol` serve a similar purpose. A common reshaping of matrices is to make the rows to be columns and the columns to be rows. This called transforming the matrix. The transpose of a matrix is obtained by the R function `t()`:

```
> t(AA)
     r1 r2 r3
[1,]  1  5  9
[2,]  2  6 10
[3,]  3  7 11
[4,]  4  8 12
```

The R function `t()` also operates on a vector, treating the vector as a "column" vector:

```
> x <- 1:3
> t(x)
     [,1] [,2] [,3]
[1,]    1    2    3
```

1.3.2 Numerical Vectors and Matrices

For mathematical applications on the computer, we need objects that simulate certain mathematical objects, such as integers, real numbers, sets, vectors, and matrices.

Mathematical vectors and matrices are defined in terms of elements from a field (essentially, "numbers"). In R, the terms "vectors" and "matrices" refer to more general arrays (that is, their elements may not be numbers).

If the elements of R vectors and matrices are numbers, however, they can be manipulated in ways similar to mathematical operations on vectors and matrices. As we have mentioned above, atomic vectors of class `numeric` can be operated on just as vectors in a vector space.

In this section we will describe some simple operations on vectors and matrices. In Chapter 7 we will discuss other operations of linear algebra and applications in statistical analysis of financial data.

Basic Operations with Numerical Vectors and Matrices

Most operators such as "`+`", "`-`", "`*`", and "`/`" are applied elementwise when the operands are arrays. The symbol "`*`", for example, indicates the Hadamard product of two matrices, that is, the elementwise product. This is often denoted in mathematics by "$\odot$". In the expression

```
AA * BB
```

the number of rows of `AA` must be the same as the number of rows of `BB`, and the number of columns of `AA` must be the same as the number of columns of `BB`. We have

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \odot \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 1 & 4 & 9 \\ 16 & 25 & 36 \\ 49 & 64 & 81 \end{bmatrix}.$$

Cayley multiplication of matrices, on the other hand, in mathematics is just indicated by juxtaposition, that is, there is no explicit symbol; but in R it is denoted by the symbol "`%*%`". Cayley multiplication is the "usual" multiplication of matrices. The expression

```
AA %*% BB
```

indicates the Cayley product of the matrices, where the number of columns of `AA` must be the same as the number of rows of `BB`:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 30 & 36 & 42 \\ 66 & 81 & 96 \\ 102 & 126 & 150 \end{bmatrix}.$$

Subarrays can be used directly in expressions. For example, the expression

```
AA[c(1,2), ] %*% BB[, c(3,1)]
```

yields the product

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 3 & 1 \\ 6 & 4 \\ 9 & 7 \end{bmatrix} = \begin{bmatrix} 42 & 30 \\ 96 & 66 \end{bmatrix}.$$

Operations Mixing Numerical Scalars, Vectors, and Matrices

Like many other software systems for array manipulation, R usually does not distinguish between scalars and arrays of size 1. For example, if

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

the expression

```
x %*% y %*% z
```

yields the same value as `5*z` because the expression `x %*% y %*% z` is interpreted as `(x %*% y) %*% z` and `(x %*% y)` is a scalar.

The expression `x %*% (y %*% z)` is invalid because `y` and `z` are not conformable for multiplication.

In the case of one-dimensional and two-dimensional arrays, R sometimes allows the user to treat them in a general fashion, and sometimes makes a hard distinction. For example, the function `length()` returns the number of elements in an array, whether it is one-dimensional or two-dimensional. The function `dim()`, however, returns the numbers of rows and columns in a two-dimensional array, but returns `NULL` for a one-dimensional array.

The basic atomic numeric type in R is a vector, and a $1 \times n$ matrix or an $n \times 1$ matrix may be cast as vectors (that is, one-dimensional arrays). This is often exactly what we would want. There are, however, cases where this casting is disastrous, for example if we use the `dim()` function, as shown in Figure 1.20. To prevent this casting, we can use the `drop` keyword in the subsetting operator.

```
> A <- matrix(c(1,2,3,4,5,6), nrow=3)
> dim(A)
[1] 3 2
> b <- A[, 2]
> dim(b)        # b  is not two-dimensional
NULL
> length(b)
[1] 3
> C <- A[, 2, drop=FALSE]
> dim(C)        # C  is two-dimensional
[1] 3 1
```

Figure 1.20. Downcasting and Preserving the Class in R

Matrices as Statistical Datasets

One of the most common and widely-useful structures for observational data is a rectangular array in which the rows correspond to observational units, and the columns correspond to observable features, as in Table 1.2. This structure is essentially a matrix.

Table 1.2. Statistical Dataset in a Flat File Structure

	Var_1	Var_2	...
Obs_1	X	X	...
Obs_2	X	X	...
⋮	...	...	...

Statistical analyses can often be formulated in terms of operations on vectors and matrices. Observations on a single univariate feature are stored in a vector, and observations on several features are stored in a matrix in which each column corresponds to a feature or variable and each row corresponds to an observational unit. When a matrix is used in this way, it is convenient to give names to the columns that correspond to names of the features. This can be done by means of the `colnames()` function, which can also be used to retrieve the names of the columns of a matrix, if they have been given names.

When a matrix is used to contain a statistical dataset, in addition to the names of the variables, it may be convenient to have names for the observational units, that is, for the rows. This can be done by means of the `rownames()` function, which can also be used to retrieve the names of the rows of a matrix, if they have been given names. The row names and the column names can be used as indexes to access the elements. (As noted previously, neither the row names nor the column names must be unique.)

Figure 1.21 shows an example.

Vectorization

We often want to apply some operation to all of the elements in an array or to all of the elements in a given dimension of an array. As we noted in Figure 1.2 on page 22, most R functions that operate on scalar objects do this automatically when they are given an array object. User-written functions do this also (if they are properly written).

```
> sqr <- function(x) return(x^2)
> xm <- matrix(c(1,2,3,4), nrow=2)
> xm
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> sqr(xm)
     [,1] [,2]
[1,]    1    9
[2,]    4   16
```

```
> Stks <- matrix(c(157.74, 200,
+                   24.64, 400,
+                   75.59, 200,
+                   46.93, 400,
+                  101.57, 300), byrow=TRUE, ncol=2)
> colnames(Stks) <- c("Price","Quantity")
> rownames(Stks) <- c("AAPL","BAC","COF","INTC","MSFT")
> colnamesStks)
[1] "Price"    "Quantity"
> Stks
      Price Quantity
AAPL 157.74      200
BAC   24.64      400
COF   75.59      200
INTC  46.93      400
MSFT 101.57      300
> Stks["BAC","Price"]
[1] 24.64
> Stks["BAC", ]
   Price Quantity
   24.64   400.00
> Stks[, "Price"]
  AAPL    BAC   COF    INTC   MSFT
157.74  24.64 75.59   46.93 101.57
```

Figure 1.21. A Matrix as a Statistical Dataset

Thus, there is no need to loop explicitly over the elements of the array. The user should not think of the individual elements, but rather think of the object as a matrix.

There are many instances in which we may wish to perform operations on the individual elements in an array. That does not mean, however, that we must loop through those elements. At the user level, the appropriate R function may operate on a large, complicated structure, but at the computational level, the function operates on the elements separately.

For example, suppose we have an array that consists of numeric elements and we want to form an analogous array that whose elements depend on a binary condition involving the elements of the given array, maybe whether those elements are equal to some specified value. The R function `ifelse()` performs this task easily.

```
> xv <- c(1,2,4,2,5,6,2)
> xv2 <- ifelse(xv==2, 2, 0)
> xv2
[1] 0 2 0 2 0 0 2
```

The result of `ifelse()` will be an array of the same dimensions as the given array, but it does not have to be of the same data type.

```
> xm <- matrix(c(1,2,3,4), nrow=2)
> xm2 <- ifelse(xm>2, TRUE, FALSE)
> xm
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> xm2
      [,1] [,2]
[1,] FALSE TRUE
[2,] FALSE TRUE
```

Often when working with a multidimensional array, we want to perform an operation on a one-dimensional projection of the array. For example, in a two-dimensional array, we may want to compute the sums or the means of the individual columns. The `apply()` R function provides the ability to perform the operation of another R function that accepts a vector argument over either the rows or the columns. The `apply()` function has an argument for the "margin". In a two-dimensional array, the rows are margin 1 and the columns are margin 2. For the case of sums or means, there are four special functions, `rowMeans()`, `rowSums()`, `colMeans()`, and `colSums()` (each of which allows the user to interchange how "column" and "row" are interpreted), that perform the same operation as `apply()`. These functions are illustrated in Figure 1.22.

The `apply()` function "vectorizes" the operations and it should be used instead of explicitly looping over the rows or columns.

The `apply()` R function provides the ability to perform the operation of a function that accepts a vector argument over any of the lower-dimensional projections of the multidimensional array.

The function being applied operates in its usual way. Arguments to the function being applied can be passed as arguments to the `apply()` function. For example, if missing values are present, some functions give optional ways of handling them with the `na.rm` argument, and the value of `na.rm`. Using the 3-dimensional array `x3D` used above, but with one element set to NA, we illustrate the various sums in Figure 1.23.

The `apply()` operates on structures of class `array` (or related, such as class `matrix`), as illustrated in Figures 1.22 and 1.23. There are other "apply" functions appropriate for other structures, such as lists, and we will mention some of them when discussing other structures.

1.3.3 Time Series Objects; The `ts` Class

Since most financial data are time series, an R class for time series is useful.

```
> A <- matrix(c(1,2,3,6,5,4), nrow=2)
> A
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    6    4
> apply(A, 1, sum)   # Compute marginal sums over the rows
[1]  9 12
> rowSums(A)            # same
[1]  9 12
> apply(A, 1, var)   # Compute marginal variances over the rows
[1] 4 4
> apply(A, 2, sum)   # Compute marginal sums over the columns
[1] 3 9 9
> colSums(A)            # same
[1] 3 9 9
> apply(A, 2, var)   # Compute marginal variances  over the columns
[1] 0.5 4.5 0.5
```

Figure 1.22. Vectorized Operations on Matrices

The first consideration in forming a computer structure for time series data is whether or not the data are equally spaced in time, or approximately so.

A large number of the time series of interest can be considered to be equally spaced. For example, while "daily" stock prices are not equally spaced in time, for most practical purposes, we can consider them to be.

If the data are equally spaced, all we need is the beginning or ending date or time and the time interval between successive points in time. For equally-spaced data, we may use natural divisions of the time intervals. For example, if the data are collected monthly, we may consider a year as the basic time unit and define a "frequency" of 12. Quarterly data could be considered to have a frequency of 4 within a basic time unit of one year.

For unequally-spaced data, a more complicated structure would be necessary. A data frame with a variable representing the time, which may be of class `Date`, would be one possibility, but a better solution would be a class of object that handles the time in a more natural way. The `zoo` or `xts` objects serve this purpose. We will defer discussion of these objects to Section 1.3.6.

In this section we will concentrate on equally-spaced data, and describe two object classes provided in the `base` package, `ts` and `mts`.

`ts` and `mts` Objects

The `ts` class is a numeric vector with attributes that specify the beginning and ending and the frequency of the time series. The `mts` class is similar to

```
> x3D <- array(1:12, dim=c(2,3,2))
> x3D[1,2,1] <- NA
> x3D
, , 1

     [,1] [,2] [,3]
[1,]    1   NA    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

> applyx3D, 1, sum)
[1] NA 42
> apply(x3D, c(1,2), sum)
     [,1] [,2] [,3]
[1,]    8   NA   16
[2,]   10   14   18
> apply(x3D, c(1,2), sum, na.rm=TRUE)
     [,1] [,2] [,3]
[1,]    8    9   16
[2,]   10   14   18
```

Figure 1.23. Vectorized Operations on a Three-Dimensional Array

`ts` but allows for multiple synchronized time series as numeric columns in a matrix; hence, we restrict most discussion in the following to the `ts` class.

We will illustrate the characteristics of the class by a time series with the 21 values $20, 21, \ldots, 40$ scrambled randomly. The `ts()` function is the constructor for the class, and the attributes are assigned by the keyword arguments `start`, `end`, and `frequency`. These arguments are of class `numeric`; a `Date` object is invalid.

Unless some meaning is attached to the beginning, ending, and frequency attributes, they just default to the beginning and ending indexes of the object and to a frequency of 1:

```
> set.seed(12345)
> data <- sample(20:40)
> ts(data)
Time Series:
Start = 1
End = 21
Frequency = 1
```

```
 [1] 35 37 34 40 27 22 24 36 29 31 20 21 26 30 39 28 32 38 33 23 25
```

If a starting time is specified and a frequency is not specified (that is, the frequency=1), the times are just interpreted as a sequence of the starting time in increments of 1:

```
> ts(data, start=2016)
Time Series:
Start = 2016
End = 2036
Frequency = 1
 [1] 35 37 34 40 27 22 24 36 29 31 20 21 26 30 39 28 32 38 33 23 25
```

set.seed(12345) data ¡- sample(20:40) ts(data)

Frequency in ts Objects

Much economic data are observed monthly, quarterly, or annually. The `frequency` attribute of `ts` objects allows for divisions of the basic time unit into subunits.

While the concept of frequency is straightforward, its use in `ts` objects is meaningful for only three special values. A frequency of 1, which is the default in most functions involving `ts` objects, indicates that no subdivisions of the basic time unit are made; and that basic time unit is more-or-less irrelevant. A frequency of 4 or 12 indicates that basic time unit ia a year and that months and quarters are defined with respect to the Western calendar. The starting time is to be a year number in that calendar. A frequency of 12 indicates that the time divisions are months, and a frequency of 4 indicates quarters. Other values of `frequency` are ignored, although they are retained as attributes of the object.

```
> ts(data, start=2016, frequency=12)
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
2016  35  37  34  40  27  22  24  36  29  31  20  21
2017  26  30  39  28  32  38  33  23  25

> ts(data, start=2016, frequency=4)
     Qtr1 Qtr2 Qtr3 Qtr4
2016   35   37   34   40
2017   27   22   24   36
2018   29   31   20   21
2019   26   30   39   28
2020   32   38   33   23
2021   25
```

```
> ts(data, start=2016, frequency=3)
Time Series:
Start = c(2016, 1)
End = c(2022, 3)
Frequency = 3
 [1] 35 37 34 40 27 22 24 36 29 31 20 21 26 30 39 28 32 38 33 23 25
```

The starting date can be specified as a decimal fraction if the fractional part to 7 digits corresponds to a subunit corresponding to the frequency. Other fractional values are processed, but are not interpreted.

```
> ts(data, start=c(2016.0833333333), frequency=12)
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
2016      35  37  34  40  27  22  24  36  29  31  20
2017  21  26  30  39  28  32  38  33  23  25

> ts(data, start=2016.25, frequency=4)
     Qtr1 Qtr2 Qtr3 Qtr4
2016        35   37   34
2017   40   27   22   24
2018   36   29   31   20
2019   21   26   30   39
2020   28   32   38   33
2021   23   25

> ts(data, start=c(2016.08), frequency=12)
Time Series:
Start = 2016.08
End = 2017.74666666667
Frequency = 12
 [1] 35 37 34 40 27 22 24 36 29 31 20 21 26 30 39 28 32 38 33 23 25
```

The starting and ending periods are specified as a two-element vector with the first element giving the year and the second element giving the month or the quarter, depending on the frequency.

```
> datats <- ts(data, start=c(2016,6), frequency=12)
> datats
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
2016                     35  37  34  40  27  22  24
2017  36  29  31  20  21  26  30  39  28  32  38  33
2018  23  25

> ts(data, start=c(2016,6), frequency=4)
     Qtr1 Qtr2 Qtr3 Qtr4
```

```
2017        35   37   34
2018   40   27   22   24
2019   36   29   31   20
2020   21   26   30   39
2021   28   32   38   33
2022   23   25
```

These examples indicate most of the relevant properties of `ts` objects. The `ts()` function and most of the R functions that operate on `ts` objects are robust to the values of the attributes. If `start`, `end`, and `frequency` correspond to simply interpretable dates in years and months or quarters, those interpretations will be applied; otherwise, the object will just be interpreted as a numeric vector and the attribute apply without interpretation in common terms.

There are some useful functions that summarize or manipulate `ts` objects. The functions `start()`, `end()`, and `frequency()` functions retrieve the indicated attributes of a `ts` objects, and the `deltat()` function returns the period as a fraction of the basic time unit, that is, the reciprocal of the frequency. The `time()` function extracts the time index and converts it to fractional amounts in the basic time unit. It returns a `ts` object. (The `datats` time series below is one in the examples above. We display it again for easy reference.)

```
> datats
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
2016                     35  37  34  40  27  22  24
2017  36  29  31  20  21  26  30  39  28  32  38  33
2018  23  25
> start(datats)
[1] 2016    6
> end(datats)
[1] 2018    2
> frequency(datats)
[1] 12
> deltat(datats)
[1] 0.08333333
> head(time(datats))
[1] 2016.417 2016.500 2016.583 2016.667 2016.750 2016.833
```

Subsetting `ts` Objects

A `ts` object is a numeric vector with some associated attributes. The attributes can be used in certain operations, but they are quite limited. For example, they cannot be used directly as indexes to the `ts` object.

The `window()` function can be used to subset a `ts` object by specifying beginning and ending points in the time series. (Again, we use the `datats` time series from above.)

```
> subdatats = window(datats, start=c(2016,9), end=c(2017,2))
> subdatats
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
2016                                  40  27  22  24
2017  36  29
```

Subsetting a `ts` object using indexes, such as `datats[4:8]`, preserves the correct data, but produces a numeric (in this case, integer) object.

```
> subdata = datats[4:9]
> subdata
[1] 40 27 22 24 36 29
> class(subdata)
[1] "integer"
```

The meanings of the data in `ts` objects must be interpreted correctly when working with objects of different frequencies. Monthly and quarterly stock price data, for example, are measured at the ends of the respective periods; hence, quarterly data can be produced from monthly data just by selecting ever third month.

In order to convert a monthly series to a quarterly series by selecting the months of Mar, Jun, Sep, and Dec, the subset can be chosen by use of the indexes, and then the object converted back to a `ts` object with a frequency of 4. To do this, we first determine the starting month of the series, and then find the end of the first quarter that includes that month.

```
> strtyr <- start(datats)[1]
> strtmo <- start(datats)[2]
> endyr <- end(datats)[1]
> endmo <- end(datats)[2]
> ind1 <- strtmo:(12*(endyr-strtyr)+endmo)
> ind <- ind1[ind1%%3==0]
> qrtdatats <-
+      ts(datats[ind+1-strtmo], start=c(strtyr,ceiling(strtmo/3)), frequency=4)
> qrtdatats
     Qtr1 Qtr2 Qtr3 Qtr4
2016        35   40   24
2017   31   26   28   33
```

Aggregating Data in `ts` Objects

Aggregating data in a time series generally means formation of equivalent data at a lower frequency, converting daily data to weekly data, for example.

The way data are aggregated depends on the kind of data. Stock price data for a given quarter, for example, it just the price data at the end of the

last month in that quarter. This can be determined as in the simple example above. On the other hand, for data such as revenue or earnings, the amount at the end of a quarter is equal to the sum of that in the three months comprising the quarter; it is not just the same as monthly data for particular months. For other types of data, such as extremes, the values for a quarter correspond to values in some month in the quarter. The highest price of a stock in a quarter is the highest price in one of the months comprising the quarter, but which month is not known until the data in all three months are examined.

The R function `aggregate()` computes functions of data at a given frequency into aggregated data at a lower frequency. The most common types of aggregation, as indicated above, are summation of the values within the higher frequency time units, and determination of the maximum of minimum value within those time units. The operation is specified by the keyword argument `FUN` in `aggregate()`.

As illustrated in the example above, if the series at the higher frequency does not begin at a natural time period for the lower frequency, the aggregation must be adjusted.

If the aggregation involves all monthly data within the quarter, we must find the end of the first *full* quarter after that month. Note the difference in this and what was done above for data similar to closing prices.

Suppose that the data in our example `datats` are volumes of trades within the months making up the time series. To aggregate these data to quarters involves summing the volumes within the three months comprising a quarter. Notice that the first quarter in our aggregated series is the third quarter.

```
> strtyr <- start(datats)[1]
> strtmo <- start(datats)[2]
> qrtaggreg <- aggregate(window(datats, start= c(strtyr,3*ceiling(strtmo/3)+1)),
                         nfrequency=4, FUN=sum)
> qrtaggreg
     Qtr1 Qtr2 Qtr3 Qtr4
2016            111   73
2017   96   67   97  103
```

Now, suppose that the data in our example `datats` are the highest price of a stock within each month. To aggregate these data to quarters, we must determine the maximum among the three months comprising each quarter. Again, the first quarter in our aggregated series is the third quarter.

```
> qrtmax <- aggregate(window(datats, start= c(strtyr,3*ceiling(strtmo/3)+1)),
                         nfrequency=4, FUN=max)
> qrtmax
     Qtr1 Qtr2 Qtr3 Qtr4
2016             40   27
2017   36   26   39   38
```

Merging ts Objects

ts objects that have have the same frequency can be merged into a matrix by use of the cbind() function. The result is an mts object. If the starting and ending dates do not match, NAs result in the unmatched cells. The ts.intersect() function selects only the cells whose time stamps are the same.

```
> data1ts <- ts(20:23, start=c(2016,6), frequency=12)
> data2ts <- ts(28:32, start=c(2016,7), frequency=12)
> datamts <- cbind(datats, data2ts)
> class(datamts)
[1] "mts"    "ts"     "matrix"
> datamts
         data1ts data2ts
Jun 2016      20      NA
Jul 2016      21      28
Aug 2016      22      29
Sep 2016      23      30
Oct 2016      NA      31
Nov 2016      NA      32
> ts.intersect(data1ts, data2ts)
         data1ts data2ts
Jul 2016      21      28
Aug 2016      22      29
Sep 2016      23      30
```

The ts object class has obvious limitations. Except for the monthly and quarterly rates, the frequency attribute is not very useful. The class has its own methods for plotting (plot.ts(), which we will use in Section 1.4.3), and plotting methods for mts objects allow either multiple plots on one set of axes or plots in multiple panels. There are also some other R functions for operations on equally-spaced time series objects, such as lag() and diff() that will be discussed beginning on page 99, although those functions operate equally well on numeric vectors that are not of class ts.

The standard R functions for modeling time series, such as arima(), and widely-used packages for time series, such as forecast, work with ts/mts objects.

1.3.4 Lists

A *list* in R is a simple, but very useful structure. It differs from an atomic vector in that the elements of a list may be objects of different types, including other lists. The elements of a list can be named and accessed by their names, or else they can be accessed by an index, similar to an atomic vector, but because of the fact that the elements of a list can be other lists, the elements have a hierarchy, and a simple index in a list refers to a top-level item in the list (see portfolios[2] in the example below).

If an element in a list has a name within the list, the name of the element is the name of the list and the name of the element with the symbol $ separating the names. The names of all upper-level elements in a list can be obtained by the `names()` function.

The power of lists lies in their flexibility. They are useful for the results of a statistical analysis. Consider, for example, a linear regression analysis using least squares. In R, this is performed by the `lm()` function.

A regression analysis consists of several results. It always includes the usual coefficient estimates and sums of squares. In some cases, however, the analysis may include computations of various statistics on the residuals, and/or assessment of possible effects of multicollinearity. The `lm()` function produces a list that contains the results of the analysis.

```
> regr <- lm(formula)
> names(regr)
 [1] "coefficients"  "residuals"     "effects"     "rank"
 [5] "fitted.values" "assign"        "qr"          "df.residual"
 [9] "xlevels"       "call"          "terms"       "model"
```

A list is constructed by the `list()` function. The elements of the list may be given names when the list is formed.

```
portfolios <- list("Fund1"=c("INTC","MSFT"),
                   "Fund2"=c("IBM","ORCL","CSCO"))
```

The list `portfolios` contains two elements, each of which is an atomic character vector. An element of a list may be accessed by its name, or by its index:

```
> portfolios$Fund2
[1] "IBM"  "ORCL" "CSCO"
> portfolios[2]
$Fund2
[1] "IBM"  "ORCL" "CSCO"
```

Two R functions that are useful in working with lists, especially list produced by R functions instead of the user, are `str()` (for "structure") and `names()` which was shown above.

```
> str(portfolios)
List of 2
 $ Fund1: chr [1:2] "INTC" "MSFT"
 $ Fund2: chr [1:3] "IBM" "ORCL" "CSCO"
> names(portfolios)
[1] "Fund1" "Fund2"
```

Figure 1.24 illustrates some of the features of lists.

```
> mylist <- list("one_to_three"=1:3,
+                "a",
+                list("four_to_six"=4:6, "b", list(7:9, "c")))
> mylist
$one_to_three
[1] 1 2 3
[[2]]
[1] "a"
[[3]]
[[3]]$four_to_six
[1] 4 5 6
[[3]][[2]]
[1] "b"
[[3]][[3]]
[[3]][[3]][[1]]
[1] 7 8 9
[[3]][[3]][[2]]
[1] "c"
> str(mylist)
List of 3
 $ one_to_three: int [1:3] 1 2 3
 $             : chr "a"
 $             :List of 3
  ..$ four_to_six: int [1:3] 4 5 6
  ..$            : chr "b"
  ..$            :List of 2
  .. ..$ : int [1:3] 7 8 9
  .. ..$ : chr "c"
> names(mylist)
[1] "one_to_three" ""             ""
```

Figure 1.24. Lists and the `str()` Function

The "[" operator is used to specify an index into a list, `portfolios[2]` or `mylist[1]`, for examples. The element indexed in the list, however, is returned as a list.

Figure 1.25 illustrates some more features of lists. Some results are more obvious than others. Make sure that you understand each.

The `lapply()` function and two variations `sapply()` and `vapply()` operate on structures of class `list`.

1.3.5 Data Frames

As we have noted, one of the most common and widely-useful structures for observational data is a rectangular array in which the rows correspond to ob-

```
> mylist$one_to_three
[1] 1 2 3
> class(mylist[1])
[1] "list"
> mylist[1]
$one_to_three
[1] 1 2 3
> class(mylist[[1]])
[1] "integer"
> mylist[[1]]
[1] 1 2 3
> mylist$one_to_three[2]
[1] 2
> mylist[[1]][2]
[1] 2
> mylist[[3]]$four_to_six
[1] 4 5 6
> mylist[[3]][1]
$four_to_six
[1] 4 5 6
> mylist[[3]][3]
[[1]]
[[1]][[1]]
[1] 7 8 9
[[1]][[2]]
[1] "c"
> mylist[[3]][[3]][1]
[[1]]
[1] 7 8 9
```

Figure 1.25. Lists and the $ and [[Operators (Using `Mylist` from Figure 1.24)

servational units, and the columns correspond to observable features, yielding a flat file structure as in Table 1.2.

For numerical data, this structure naturally corresponds to a mathematical matrix and to an R object of class `matrix`. In statistical applications, it may be convenient to give names to the columns and rows of a matrix, as the matrix `Stks` in Figure 1.21.

The `matrix` class in R can also be of mode and type `character`, but a `matrix` object cannot contain elements of both `numeric` and `character` modes. Often in statistical applications, some variables may be numerical and others may be character. Some variables may have other characteristics, such as being dates.

The R class `data.frame` accommodates data in this structure. Each variable or column can be of any mode.

An example of some data is shown in Table 1.3 below. The numerical values are exactly the same as those in the matrix `Stks` of Figure 1.21, but

here the names of the stocks are values of a variable called "Symbol", instead of being names of the rows of a matrix. This is a dataset with three variables,

Table 1.3. Stock Data; Prices and Numbers of Shares

Symbol	Price	Quantity
AAPL	157.74	200
BAC	24.64	400
COF	75.59	200
INTC	46.93	400
MSFT	101.57	300

"Symbol", which is a character variable, "Price", and "Quantity". It is still in the flat file structure of Table 1.2. We will build an R data frame called "Stocks_1" with these data in Figure 1.26 below.

There are other ways of forming a data frame, but this is one of the simplest and most common.

In many ways an R data frame is like a matrix; for example, the elements of a data frame can be accessed by the same kind of indexing as done with a matrix, and the R functions `nrow()` and `ncol()` are the same. In other ways, however, data frames behave more like R lists.

Both the rows and the columns in a `data.frame` have names. The names of the columns are usually the names of the variables. The names of the columns are the names of the vectors used to form the data frame, if that was the way it was formed. The default and most common names for the rows are just the positive integers, which can be interpreted as the observation numbers. The names of the columns in an R data frame are similar to the names of columns in a matrix, and are accessed likewise by `colnames()`. The names of the rows in an R data frame are similar to the names of rows in a matrix, and are accessed likewise by `rownames()`.

The column names in an R data frame are the variable names in the common paradigm of a statistical dataset, so the "names" are those. The same R function `names()` is used for data frames as is used for lists. A variable in a data frame can be accessed by the name of the data frame followed by $ and then by the name of the variable. Note that $ cannot be used to extract columns (or variables) in a matrix.

The row names and the column names can be used as indexes to access the elements.

Figure 1.26 illustrates the creation of a data frame and some simple manipulations with it.

The last statement in Figure 1.26 may give one pause. In this snippet of code, `Stocks_1$Symbol` is merely a character variable, so we not expect it to

```
> Symbol   <- c("AAPL", "BAC", "COF", "INTC", "MSFT")
> Price    <- c(157.74, 24.64, 75.59,  46.93, 101.57)
> Quantity <- c(   200,   400,   200,    400,    300)
> Stocks_1 <- data.frame(Symbol, Price, Quantity)
> Stocks_1
  Symbol  Price Quantity
1   AAPL 157.74      200
2    BAC  24.64      400
3    COF  75.59      200
4   INTC  46.93      400
5   MSFT 101.57      300
> Stocks_1$Price
[1] 157.74 24.64 75.59 46.93 101.57
> Stocks_1[1,2]
[1] 157.74
> Stocks_1[2,3]
[1] 400
> Stocks_1[1, ]
  Symbol  Price Quantity
1   AAPL 157.74      200
> names(Stocks_1)
[1] "Symbol"   "Price"    "Quantity"
> colnames(Stocks_1)
[1] "Symbol"   "Price"    "Quantity"
> rownames(Stocks_1)
[1] "1" "2" "3" "4" "5"
> class(Stocks_1$Price)
[1] "numeric"
> class(Stocks_1$Symbol)
[1] "factor"
```

Figure 1.26. An R Data Frame with Data from Table 1.3()

be of class `factor`. By default, however, when R forms a data frame, character vectors are converted to class `factor`.

Factors in Data Frames

On page 22, we discussed categorical variables and described the `factor` class, which is appropriate for such variables.

As mentioned following Figure 1.26, when R forms a data frame, character vectors are converted to class `factor`. For variables that are indeed categorical variables, this is appropriate; otherwise, however, this may not be what we want. The `stringsAsFactors=FALSE` option in `data.frame()` prevents this from happening. On the other hand, it may be useful to have one or more variables to be factors. This will be the case if the vector is initially a factor. These two points are illustrated in Figure 1.27, where we build a data frame

similar to that in Figure 1.26 and include a variable that really should be a classification variable.

```
> Sector   <- factor(c("Tech", "Fin", "Fin", "Tech", "Tech"))
> Symbol   <- c("AAPL", "BAC", "COF", "INTC", "MSFT")
> Price    <- c(157.74, 24.64, 75.59, 46.93, 101.57)
> Quantity <- c(   200,   400,   200,    400,    300)
> Stocks_2 <- data.frame(Sector, Symbol, Price, Quantity,
+                        stringsAsFactors=FALSE)
> Stocks_2
  Sector Symbol  Price Quantity
1   Tech   AAPL 157.74      200
2    Fin    BAC  24.64      400
3    Fin    COF  75.59      200
4   Tech   INTC  46.93      400
5   Tech   MSFT 101.57      300
> class(Stocks_2$Symbol)
[1] "character"
> class(Stocks_2$Sector)
[1] "factor"
> Stocks_2$Sector[1]
[1] Tech
Levels: Fin Tech
```

Figure 1.27. An R Data Frame with a Factor Variable and a Character Variable

The `stringsAsFactors=FALSE` option can also be set globally for an R session by use of the `options()` function:

```
options(stringsAsFactors=FALSE)
```

It is usually not a good idea to use the global `options()` function, however.

Working with factors in a data frame involves the same considerations we discussed on page 22. For instance, to obtain the sector value of the first observation, we do not use

```
Stocks_2$Sector[1]
```

but rather we must use

```
as.character(Stocks_2$Sector[1])
```

As mentioned previously, the reason factors are so useful is that mathematical operations or statistical analyses can be performed on other data separately at each level of a factor. This can be done on variables in a data frame or just on individual vectors of the same length when one of them is of class `factor`.

Operations on Data Frames

As we have seen, ordinary matrix-type indexing can be used in a data frame. These indexes can be used to form subsets of a data frame, and of course the result is usually a data frame. If the result consists of a single column, however, even if the original data frame only had one column, the result is an atomic vector by default. This is the same kind of downcasting we observed for matrices in Figure 1.20. To prevent this casting, as with matrices, we can use the `drop` keyword in the subsetting operator. Figure 1.28 illustrates these points using the `Symbol` factor vector and the `Stocks_2` data frame from Figure 1.27. (See also Figure 1.20.) The downcasting does not change the variable in the data frame.

```
> class(Stocks_2[,1])
[1] "factor"
> class(Stocks_2[,1,drop=FALSE])
[1] "data.frame"
> class(Stocks_2[,2])
[1] "numeric"
> class(Stocks_2[,2,drop=FALSE])
[1] "data.frame"
> SymbolDF <- data.frame(Symbol)
> class(SymbolDF)
[1] "data.frame"
> class(SymbolDF[-1,,drop=FALSE])
[1] "data.frame"
> class(SymbolDF[-1,])
[1] "factor"
```

Figure 1.28. Subsetting R Data Frames and Downcasting

Data frames can be combined or updated using `cbind()` and `rbind()`, just as with matrices. The data frame `Stocks_2` could have been built using `Stocks_1` from Figure 1.26 along with the vector `Sector` from Figure 1.27 using `cbind()`.

```
Stocks_2 <- cbind(Sector, Stocks_1)
```

(There would be one small difference: in building `Stocks_1`, the `Symbol` variable became a factor; but in building `Stocks_2`, we forced it to be a character variable. In the next few examples, as we build the `Stocks_1` data frame in various ways, it will be different from the original `Stocks_1` data frame because the `Symbol` variable will be a character variable.)

Data frames can also be combined in other meaningful ways using the `merge()` function. The operation performed by `merge()` is called "join" in general database terminology. This function is very flexible, and among other options, allows merging by the variable names or by the observation names or by various combinations. The `merge()` function operates on only two data frames at a time, and joins the data frames by matching a specified variable that is in both data frames.

The simple R operator "[" can be used to select elements or variables (columns) or observations (rows) within an R data frame just as in any array,. For example, the data frame `Stocks_1` above can be formed from the `Stocks_2` data frame as follows.

```
Stocks_1 <- Stocks_2[, c("Symbol","Price","Quantity")]
```

The R function `subset()` can also be used for that purpose.

```
Stocks_1 <- subset(Stocks_2, select=c("Symbol","Price","Quantity"))
```

The `select()` function in the `dplyr` package provides more flexibility in subsetting data frames.

Neither the "[" operator nor the `subset()` function can be used with the unary "-" operator to remove variables by specifying their names. The "-" operator can be used with the regular row or column indexes, however. For example, the data frame `Stocks_1` above can be formed from the `Stocks_2` data frame as follows.

```
Stocks_1 <- Stocks_2[, -1]
```

Similar to the `apply()` function discussed earlier, the `tapply()` R function applies another operation to a given vector or set of vectors, but `tapply()` does it separately for each level of a specified factor vector. The `tapply()` function returns an array. For example, using the objects above, we can compute the total value in each sector as follows.

```
> tapply(Price*Quantity, Sector, sum)
   Fin   Tech
 24974  80791
```

The arguments for the `tapply()` function above could have been specified as the variables in the data frame `Stocks_2`, but since those variable names are also the names of the individual vectors, the vectors were used as arguments in this example. The first argument can be a vector of any class, but it cannot be a data frame.

The `split()` function splits either a vector or a data frame into separate subsets corresponding to the different level in a factor. Again, using the objects above, we separate the whole data frame into parts corresponding to the different levels of the factor variable `sector`.

```
> split(Stocks_2, Sector)
$Fin
  Sector Symbol Price Quantity
2    Fin    BAC 24.64      400
3    Fin    COF 75.59      200

$Tech
  Sector Symbol  Price Quantity
1   Tech   AAPL 157.74      200
4   Tech   INTC  46.93      400
5   Tech   MSFT 101.57      300
```

The `split()` function yields a list whose primary elements are the resulting data frames. The `unsplit()` function restores the separate pieces into a single frame.

The `by()` function is similar in operation to the `tapply()` function, except the first argument can be a data frame and the function can be more general, usually an R function for a statistical analysis such as regression. The data in the toy example above does not allow a meaningful illustration of `by()`, but we will see examples of it in later chapters.

Date Information in Data Frames

Many sets of financial data, such as daily or weekly stock prices or interest rates, are time series. If the time steps are roughly equal, it may be sufficient to label the times as 1, 2, 3, and so on, with some accompanying metadata specifying the starting time and the length of the time intervals. The `ts` class, discussed in Section 1.3.3, is an adequate structure to handle this kind of time series, but if the the time intervals are not equal or if it is necessary to know the actual dates, the time of each observation needs to be an explicit part of the dataset. (Another deficiency of the `ts` class, of course, is that it can only handle numeric data.)

Date information can easily be incorporated in a data frame. One variable can be of class `Date`, for example, as in the data frame for INTC daily stock prices and volumes traded for the last few trading days of January and the first few of February, 2020, shown in Figure 1.29. The `date()` function in the `lubridate` package could be used instead of the `as.Date()` function.

```
> Price <- c(60.84, 60.10, 59.93, 58.93, 58.97, 59.30)
> Volume <- c(18056000, 15293900, 17755200, 21876100, 23133500, 18813300)
> Date <- as.Date(c("2020-01-02", "2020-01-03", "2020-01-06", "2020-01-07",
+                   "2020-01-08", "2020-01-09"))
> INTCdf <- data.frame(Date, Price, Volume)
> INTCdf
        Date Price   Volume
1 2020-01-30 66.47 18522400
2 2020-01-31 63.93 25268400
3 2020-02-03 64.42 16654600
4 2020-02-04 65.46 20970800
5 2020-02-05 67.34 23401400
6 2020-02-06 67.09 17408000
7 2020-02-07 66.02 18134600
8 2020-02-10 66.39 22299300
>
> INTCdf[3, ]
        Date Price   Volume
3 2020-02-03 64.42 16654600
```

Figure 1.29. Data Frame with a Date Variable; Indexing February 3, 2020

Picking a date or a range of dates in the data frame `INTCdf` in Figure 1.29 can be somewhat awkward. To get the data for February 3, 2020, if we know that it corresponds to the third row, we could address it as in the code in the figure. If we did not know which row, we could get the data for the date by matching the value of the `Date` variable. We could also determine the row number using `which()`.

```
> INTCdf[INTCdf$Date=="2020-02-03", ]
        Date Price   Volume
3 2020-02-03 64.42 16654600
> which(INTCdf$Date=="2020-02-03")
[1] 3
```

Another way of handling dates in data frames is to use the dates as the row names of the data frame. This is illustrated in Figure 1.30 using the same data as in the data frame `INTCdf` created in Figure 1.29.

```
> Price <- c(60.84, 60.10, 59.93, 58.93, 58.97, 59.30)
> Volume <- c(18056000, 15293900, 17755200, 21876100, 23133500, 18813300)
> Date <- as.Date(c("2020-01-02", "2020-01-03", "2020-01-06", "2020-01-07",
+                   "2020-01-08", "2020-01-09"))
> INTCdf2 <- data.frame(Price, Volume)
> rownames(INTCdf2) <- Date
> INTCdf2
           Price   Volume
2020-01-30 66.47 18522400
2020-01-31 63.93 25268400
2020-02-03 64.42 16654600
2020-02-04 65.46 20970800
2020-02-05 67.34 23401400
2020-02-06 67.09 17408000
2020-02-07 66.02 18134600
2020-02-10 66.39 22299300
>
> INTCdf2["2020-02-03", ]
           Price   Volume
2020-02-03 64.42 16654600
```

Figure 1.30. Data Frame with Dates as Row Names; Indexing February 3, 2020

Either of the methods illustrated above for obtaining a specific date is adequate, but neither structure allows for more general operations with the dates or with ranges of dates. We will discuss a more flexible structure in Section 1.3.6.

Reshaping Data Objects

Much of financial data fits into the general format of a flat file, as in Table 1.2 on page 58, in which the rows correspond to observations and the columns correspond to variables. The small example of stocks in Table 1.3 fit this structure nicely, even after we add another variable "Sector", as in Figure 1.27. The structure also accommodates additional observations, which in this case, would be additional stocks.

If we were interested in the differences between the two sectors, we might want to organize the data to separate the sectors. The organization of the data as in left-hand side of Table 1.4 shows stocks in the two sectors separately. This arrangement of the data does not fit the basic structure of Table 1.2, however.

We could easily add more variables and maintain the structure of Table 1.2. We could, for example, add prices at different times. The "Price" variable in Table 1.3 happens to be the stock price as of the close on the last trading day of 2018. Suppose we add to the dataset the corresponding prices as of the last

trading day of 2019. This would be a dataset with two variables corresponding to price. The dataset shown in right-hand side of Table 1.4 includes both prices, and the arrangement of the data corresponds to the basic structure of Table 1.2. The layout allows us easily to make comparisons between the two prices for each of the stocks. We can form an R data frame with a variable for the 2018 prices and another variable for the 2019 prices.

Table 1.4. "Wide" Datasets

Separate Sectors

Sector			
Tech		Fin	
Symbol	Price	Symbol	Price
AAPL	157.74	BAC	24.64
INTC	46.93	COF	75.59
MSFT	101.57		

Prices at Two Times

Symbol	Price	2019 Price
AAPL	157.74	293.68
BAC	24.64	35.26
COF	75.59	103.06
INTC	46.93	59.93
MSFT	101.57	157.77

It is not always obvious, however, what are the variables; that is, what are Var_1, Var_2, and so on, in Table 1.2. This is often the case when one of the variables is measured at different times. For example, The price data in the dataset on the right-hand side of Table 1.4 could be combined into a single variable if we introduce another variable for year. The additional variable may be a factor if only two or three years are involved. We may, however, anticipate building a dataset with prices corresponding to multiple dates, not just a few year-end dates. In that case, we may form a date variable.

The idea of combining the two price variables in the right-hand side of Table 1.4 and introducing another variable for date, is in a sense the opposite of what was done in the formation of the dataset on the left-hand side of Table 1.4 in which the factor variable corresponding to the sector was used to form two groups of data. In any event, the two sets of data shown in "wide" formats in Table 1.4 can be structured into "tall" formats as shown in Table 1.5.

How a dataset is displayed in a table depends on the purpose of displaying the data. The question is what is the best way to suit the purpose. Whatever the purpose, the clutter of the table must be considered. The format of the dataset on the right-hand side of Table 1.5 seems unnecessarily cluttered.

In computer structures, repetition, visual clutter, and so on are not relevant. Data frames should be constructed in a manner that allows for easy manipulation and analysis using the facilities provided in R.

For the two datasets we have shown in Tables 1.4 and 1.5 R data frames corresponding closely to the layouts of Table 1.5 would be preferred.

In financial applications using the data in Tables 1.4 and 1.5, it is actually more likely that a different wide data structure would be more appropriate. It

Table 1.5. "Tall" Datasets

Sector	Symbol	Price
Tech	AAPL	157.74
Tech	INTC	46.93
Tech	MSFT	101.57
Fin	BAC	24.64
Fin	COF	75.59

Date	Symbol	Price
2018-12-31	AAPL	157.74
2018-12-31	BAC	24.64
2018-12-31	COF	75.59
2018-12-31	INTC	46.93
2018-12-31	MSFT	101.57
2019-12-31	AAPL	293.68
2019-12-31	BAC	35.26
2019-12-31	COF	103.06
2019-12-31	INTC	59.93
2019-12-31	MSFT	157.77

is one with six variables, date, and the names of the five stocks. There would be only two observations, corresponding to the two dates. In Exercise 1.3.6,

Table 1.6. Time Series of Prices

Date	AAPL	BAC	COF	INTC	MSFT
2018-12-31	157.74	24.64	75.59	46.93	101.57
2019-12-31	293.68	35.26	103.06	59.93	157.77

you are asked to form a data frame of this structure by manipulating a data frame with a structure as in the table on the right side of Table 1.5.

In data analysis it is often the case that the raw data may be stored in various formats at various repositories. Manipulation of data into a common format often requires major effort before any analysis can begin. Serious consideration of these mechanical aspects is the main way in which "data science" differs from "statistics", where the emphasis is on the modeling, analysis, and inference.

The `merge()` and `split()` functions are useful in reshaping R data frames.

Two other basic R functions that aid in manipulating a data frame into an appropriate form are the `stack()` (and the related `unstack()`) and the `reshape()` functions. In addition, there are two packages, `reshape2` and `tdyr`, that provide additional functionality for restructuring data frames. The names of the functions `melt()`, `colsplit()`, and `dcast()` in the `reshape2` package, and `gather()`, `separate()`, and `spread()` in the `tdyr` package indicate the nature of the manipulations that they perform. See Wickham (2014) for discussions of "tidy data" and the `tdyr` package. Wickham also developed the `reshape2` package.

1.3.6 Time Series Objects

Many sets of financial data, such as daily or weekly stock prices or interest rates, are time series. How a computer dataset should be constructed to hold time series data depends on the nature of the time series. The `ts` class, discussed in Section 1.3.3, is adequate for many purposes. The `data.frame` class with a date variable or with dates used as row names as discussed beginning on page 77, is also adequate for many purposes.

An object that accepts dates as indexes and also allows subsetting operations based on the dates would be useful. The `zoo` class of objects does just that (see Zeileis and Grothendieck, 2005). There are other classes, such as `xts`, that inherit from `zoo`. We will consider properties of these classes below, generally referring to them as `xts` objects.

There are other aspects of a time series that can be incorporated in a data structure. For example, in some financial time series, especially ones relating to equity prices, within the fundamental time unit of the time series, four values may be of interest: the opening value, the closing value, the maximum value during that time period, and the minimum value. Time series datasets that include these values are called "OHLC" datasets ("open, high, low, close"). Data structures capturing this information are particularly useful for graphing ("candlestick" charts).

`xts` Objects

For an example, if the Price and Volume variables in the data frame `INTCdf` above are stored in an `xts` object, the dates are indexes, and the specific date February 3, 2020 can be indexed more naturally. In the following, we will refer to the data frames `INTCdf` and `INTCdf2`, created in Figures 1.29 and 1.30.

In Figure 1.31, we show an `xts` object, `INTCxts`, that contains the same data as in the data frame, `INTCdf`, but the date variable is no longer included; rather, the date information is used as the index.

An object of class `xts` has a row index that is an object of class `date`, and date operators can be used to manipulate the individual rows.

The values in an object of class `xts` must all be numeric. The actual data portion of the object is essentially a `matrix`.

An object of class `xts` provides not only for date stamping, but it allows for operations on the object via date functions and operators. On the other hand, sometimes computations are easier to perform by first converting `xts` objects to numeric objects using `as.numeric`. Of course, they can also be converted to R data frames, and sometimes this makes processing the data simpler. Conversion of an `xts` object to a data frame is straightforward using `data.frame`. A "date" variable in the data frame is created from the index in the `xts` object by the `index()` function.

There are several R packages that use the `xts` class, including the `xts` package and the `quantmod` package, which we will use extensively in this book.

```
> INTCxts
           Price   Volume
2020-01-30 66.47 18522400
2020-01-31 63.93 25268400
2020-02-03 64.42 16654600
2020-02-04 65.46 20970800
2020-02-05 67.34 23401400
2020-02-06 67.09 17408000
2020-02-07 66.02 18134600
2020-02-10 66.39 22299300
>
> INTCxts["2020-02-03"]
           Price   Volume
2020-02-03 64.42 16654600
```

Figure 1.31. An `xts` Object; Indexing February 3, 2020. Compare Data Frames in Figures 1.29 and 1.30

An object of class `xts` can be created by the `xts()` function. The `xts()` function has an interface similar to that of the `data.frame()` function, except that if there are more than one variable, the data in the variables must be formed into a matrix. Also, and more importantly, `xts()` has an argument, `order.by`, which specifies how the data are to be ordered nad which becomes part of the `xts` object. The index can be integers or dates in any of the acceptable R formats. The entries in the `xts` object (that is, the rows) are ordered by the index.

The `xts` object `INTCxts` in Figure 1.31 corresponds to the data frame `INTCdf` formed in Figure 1.29 or the data frame `INTCdf2` formed in Figure 1.30. The `xts` object can be formed by the `xts()` function. The `xts()` function can either use vectors as were used in forming the data frames, or it can use the data frames themselves. When an `xts` object is formed from a data frame, if the data frame has a date variable, that variable is dropped and its values used as the index in the `xts` object, assigned with the `order.by` argument. If, instead, the data frame has row names corresponding to dates, the row names would be captured by the `rownames()` function and used as the `order.by` argument.

A `xts` object cannot include any variable from a data frame that is not numeric.

Three ways to form an `xts` object are illustrated in Figure 1.32. They all produce the same object.

The `xts` object inherits the names of the variables in the data frame or else the names of the matrix columns (if any) or the name of the vector. A variable in an `xts` object can be accessed by the name of the followed by $

```
INTCxts <- xts(cbind(Price, Volume), order.by=Date)
INTCxts <- xts(subset(INTCdf,select=c("Price","Volume")), order.by=INTCdf$Date)
INTCxts <- xts(INTCdf2, order.by=as.Date(rownames(INTCdf2)))
```

Figure 1.32. Formation of an `xts` Object from Vectors or from Two Types of Data Frames

and then by the name of the variable, just as accessing a variable in a data frame.

Although it is not done as often, a data frame can be created from an `xts` object. The procedure essentially is to form vectors from which to form a data frame, as is done in Figure 1.29. Given the `xts` object `INTCxts`, the data frame `INTCdf` from above could be created by the statements below

```
Date <- index(INTCxts)
Price <- as.numeric(INTCxts$Price)
Volume <- as.numeric(INTCxts$Volume)
INTCdf <- data.frame(Date, Price, Volume)
```

One of the most useful features of `xts` objects is the ability to access elements or subsets by use of the index.

Indexing and Subsetting `xts` Objects

The elements of an `xts` object can be addressed either by the time index alone or by two indexes, the time and the column. Instead of a time index, the row number can be used.

The date index of the i^{th} row of an `xts` object can be obtained by use of the `index()` function in the `xts`; for example, the date index of the third row in the `xts` object `INTCxts` is obtained by

```
index(INTCxts)[3]
```

If a time index is used, it must be specified in the ISO 8601 date format order, but the "-" separators may be omitted; "20170202" is the same as "2017-02-02".

The time index of an `xts` object can be manipulated in the ways described in Section 1.2.5.

If a specified time index does not exist in an `xts` object, only the column names are returned (see Figure 1.33).

A range of times in an `xts` object can be specified using the "/" separator in the form "from/to", where both "from" and "to" are optional. If either is missing, the range is interpreted as the beginning or the end of the data object, as appropriate. Exact starting and ending times need not match the underlying data; the nearest available observation is chosen.

Figure 1.33 shows examples of indexing. It also shows a way to determine if a specified date does not exist in an `xts` object. The `xts` object is displayed in Figure 1.31. It was created in Figure 1.32.

Figure 1.34 shows another example of indexing `xts` objects and identifying missing values.

OHLC Datasets

In many financial time series the time of a datum corresponds specifically to value at the end of a time period, but the time itself is associated with an interval of time. A common example is a time series of the daily closing prices of a stock. There are three associated prices that may be of interest, The opening price, the high of the day, and the low of the day. Time series datasets that include these values are called "OHLC" datasets ("open, high, low, close"). An additional item of interest may be the number of shares traded on that day, the volume.

Plots of daily prices often show the opening, high, low, and closing prices. A convenient way of doing this is with a "candlestick" at each day. A candlestick has the appearance of the graphic in Figure 1.35, with the meanings shown.

An example of a daily candlestick plot, together with a panel showing the trading volume, is in Figure 1.51 on page 122.

Many traders believe that the relationships among the open, high, low, and close, which can be seen very easily in a candlestick, indicate something about the future price moves.

A standard data structure in financial analysis is an `xts` object containing an OHLC dataset with variable names of a standard form. Figure 1.36 shows an `xts` object, `INTC`, in this form. The closing prices and the volumes are the same as in the `INTCxts` object shown in Figure 1.31. (Note the additional variable, which is the closing price adjusted over time to account for stock splits and dividends.)

The standard forms of the variable names allow for changing of the frequency, say from daily to weekly or monthly.

Changing the Frequency in `xts` Objects

The `to.period()` function in the `xts` package operates on an `xts` object to produce a new `xts` with a coarser frequency; that is, for example, it may take an `xts` object that contains daily data and produce an `xts` object containing monthly data.

```
> INTCxts["2020-02-03"]
           Price   Volume
2020-02-03 64.42 16654600
> INTCxts["2020-02-03","Price"]
           Price
2020-02-03 64.42
> INTCxts["20200203"]
           Price   Volume
2020-02-03 64.42 16654600
> INTCxts["20200131/20200205"]
           Price   Volume
2020-01-31 63.93 25268400
2020-02-03 64.42 16654600
2020-02-04 65.46 20970800
2020-02-05 67.34 23401400
> INTCxts["20200131/"]
           Price   Volume
2020-01-31 63.93 25268400
2020-02-03 64.42 16654600
2020-02-04 65.46 20970800
2020-02-05 67.34 23401400
2020-02-06 67.09 17408000
2020-02-07 66.02 18134600
2020-02-10 66.39 22299300
> INTCxts["/20200131"]
           Price   Volume
2020-01-30 66.47 18522400
2020-01-31 63.93 25268400
> INTCxts["202001"]
           Price   Volume
2020-01-30 66.47 18522400
2020-01-31 63.93 25268400
> INTCxts["202002"]
           Price   Volume
2020-02-03 64.42 16654600
2020-02-04 65.46 20970800
2020-02-05 67.34 23401400
2020-02-06 67.09 17408000
2020-02-07 66.02 18134600
2020-02-10 66.39 22299300
> INTCxts["20200201"]
     Price Volume
```

Figure 1.33. Indexing and Subsetting in `xts` Objects (Using Object Created in Figure 1.32)

Different kinds of data must be aggregated differently, as we discussed and illustrated beginning on page 66. The closing prices are merely the closing prices at the end of the higher-frequency time period; the high and low prices are the highest and lowest within the higher-frequency time periods

```
> d1 <- index(INTCxts)[1]
> d2 <- index(INTCxts)[dim(INTCxts)[1]]
>      #  Determine if a date missing (weekend or holiday)
> for (i in d1:d2) if(length(INTCxts[as.Date(i)]>0)==0)
+        print(paste(as.Date(i),"is a weekend day or holiday"))
[1] "2020-02-01 is a weekend day or holiday"
[1] "2020-02-02 is a weekend day or holiday"
[1] "2020-02-08 is a weekend day or holiday"
[1] "2020-02-09 is a weekend day or holiday"
```

Figure 1.34. Missing Items in `xts` Objects (Using Object Created in Figure 1.32)

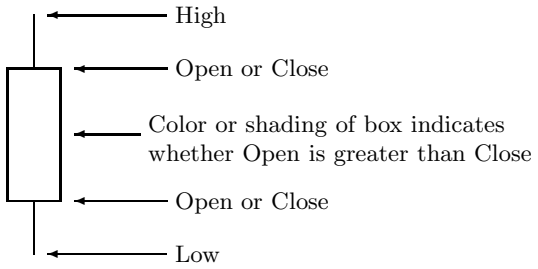


Figure 1.35. A Candlestick

```
> INTC
           INTC.Open INTC.High INTC.Low INTC.Close INTC.Volume INTC.Adjusted
2020-01-30     65.64     66.50    64.93      66.47    18522400      65.77273
2020-01-31     65.80     65.98    63.67      63.93    25268400      63.25937
2020-02-03     64.46     65.04    64.30      64.42    16654600      63.74423
2020-02-04     65.77     66.06    64.92      65.46    20970800      64.77332
2020-02-05     66.69     67.60    66.13      67.34    23401400      66.63360
2020-02-06     67.30     67.40    66.77      67.09    17408000      66.71315
2020-02-07     66.86     67.30    66.01      66.02    18134600      65.64916
```

Figure 1.36. An OHLC Dataset in an `xts` Object

comprising the coarser period; and the volume data are the sums of the volumes within the higher-frequency time periods comprising the coarser period.

If the columns of the `xts` dataset that correspond to open, high, low, close prices, and volume are all appropriately identified by the names of the columns, then the `to.period()` function produces a dataset with a coarser frequency that has the appropriate values of high, low, close prices, and volume for each of the new periods.

Figure 1.37 shows the daily data from Figure 1.36 converted to weekly data. Compare the values in the figures.

```
> to.period(INTC, "weeks")
           INTC.Open INTC.High INTC.Low INTC.Close INTC.Volume INTC.Adjusted
2020-01-31     65.64      66.5    63.67      63.93    43790800      63.25937
2020-02-07     64.46      67.6    64.30      66.02    96569400      65.64916
```

Figure 1.37. An `xts` Object Containing Daily OHLC Data Converted to Weekly

The `to.period()` function assumes that the input is an OHLC dataset. If the input is not an OHLC dataset, the `OHLC=FALSE` can be used; otherwise, some of the output of `to.period()` will be spurious.)

Figure 1.38 shows the use of `to.period()` on the `xts` object displayed in Figure 1.31, which is not an OHLC dataset. Notice that data for 2020-02-10 are produced, even though it was not the end of the week; compare with Figure 1.37. Notice also that the Volume data are incorrect; they are merely the daily data for the day at the end of the week.

```
> to.period(INTCxts, "weeks", OHLC=FALSE)
           Price   Volume
2020-01-31 63.93 25268400
2020-02-07 66.02 18134600
2020-02-10 66.39 22299300
```

Figure 1.38. Changing Time Periods in an `xts` Object. The Volume Variable Is Not Aggregated Correctly

Merging `xts` Objects

One of the most useful methods for `xts` objects is the ability to merge them based on the time index.

Merging of datasets in general is a common activity, and the ability to do so is provided in most database management systems, including those built on SQL.

The `merge()` function in R is quite flexible, and provides for most of the common options. A common application of course is to match observations in one dataset with those in the other, based on common values of one of the variables. In the case of two time series, generally, we want to match based on the time index, and that is what `merge()` does for `xts` objects. (Note that the time index is not one of the variables; rather, is is a special type of row name.)

The R `merge()` function only merges two datasets at a time.

We use two small `xts` objects, `dataxts1` and `dataxts2`, to illustrate the `merge()` function in Figure 1.39.

The examples in Figure 1.39 should be self-explanatory. The merged dataset has missing values in positions for which the value is lacking in one of the datasets. Note the use of the `join` keyword that limits the merged dataset to the observations that have the common values of the time index. The last merge in the examples is processed so that all rows with missing values were omitted. This is often useful in cleaning data.

```
> dataxts1
            INTC  MSFT
2017-01-05 36.35 62.58
2017-02-03 36.52 63.68
> dataxts2
              GSPC
2017-01-05      NA
2017-01-27 2294.69
2017-02-03 2297.42
> merge(dataxts1, dataxts2)
            INTC  MSFT    GSPC
2017-01-05 36.35 62.58      NA
2017-01-27    NA    NA 2294.69
2017-02-03 36.52 63.68 2297.42
> merge(dataxts1, dataxts2,join="inner")
            INTC  MSFT    GSPC
2017-01-05 36.35 62.58      NA
2017-02-03 36.52 63.68 2297.42
> na.omit(merge(dataxts1, dataxts2))
            INTC  MSFT    GSPC
2017-02-03 36.52 63.68 2297.42
```

Figure 1.39. Merging `xts` Objects

R Functions with xts Objects

Some standard R functions do not work in the same way on `xts` objects as they do on other objects. One that we encounter often in financial applications is in the computation of log returns. The simple way of doing this is `diff(log(XYZ))`, which yields a vector of length one less than the length of `XYZ`. If `XYZ` is an `xts` object, however, `diff(log(XYZ))` is an `xts` object with length the same as the length of `XYZ`, with an NA in the first position. Figure 1.40 illustrates the problem and two ways to deal with it. I often convert `xts` objects to `numeric` objects because I always know exactly what R functions will do to `numeric` objects. The disadvantage of the `numeric` object, of course, is that date data are lost. I often remedy this by use of either the `as.xts()` function or the `merge()` function.

There are plot methods for `xts` objects in the generic `plot()` function. There are also additional graphing functions for `xts` objects in the `quantmod` package. We will discuss and illustrate graphics with `xts` objects in Section 4.4 beginning on page 157.

1.3.7 Tables

It is often convenient to arrange data into tables in various ways. This is often appropriate for categorical data, which in R usually are of class `factor`, as we described on page 22. Consider the factor variable `sexfac` in Figure 1.3. The only relevant property of that small dataset are the numbers of the two factor levels. The R function `table()` forms a factor dataset into a table that shows just the levels and the counts of the levels. The table is of class `table`. We can form a simple table from the factor variable `sexfac`.

```
> sex <- c("m", "m", "f", "m", "f", "f", "m", "f", "m", "f", "f", "f", "m", "f")
> sexfac <- factor(sex)
> sextab <- table(sexfac)
> class(sextab)
[1] "table"
> sextab
sexfac
f m
8 6
```

The table only shows the salient property; there are 8 “f” and 6 “m”.

A dataset may contain multiple categorical variables. In a data frame, the categorical variables are generally of class `factor`. (As we noted in regard to Figure 1.26, character variables are converted to class `factor` when forming a data frame from vectors of variables.) Using the vector `sexfac` formed in Figure 1.3, and a character vector `party` shown below, we form a data frame called `people`, consisting of two variables. There are 14 observations in the data frame.

```
> library(xts)
> class(INTCdC)
[1] "xts" "zoo"
> INTCdC[1:2,]
           INTC.Close
2017-01-03      36.60
2017-01-04      36.41
> #  standard computation of returns
> INTCdCReturns <- diff(log(INTCdC))
> #  get NA
> INTCdCReturns[1:2,]
             INTC.Close
2017-01-03           NA
2017-01-04 -0.005204724
> #  problem is diff; not log
> diff(INTCdC)[1:2,]
           INTC.Close
2017-01-03         NA
2017-01-04  -0.189998
> #  clean NAs
> INTCdCReturnsClean <- na.omit(INTCdCReturns)
> INTCdCReturnsClean[1:2,]
             INTC.Close
2017-01-04 -0.005204724
2017-01-05 -0.001649313
> #  fix name
> names(INTCdCReturnsClean)<-"INTC.Return"
> INTCdCReturnsClean[1:2,]
            INTC.Return
2017-01-04 -0.005204724
2017-01-05 -0.001649313
> #  alternatively, get returns in numeric vector
> INTCdCReturnsNum <- diff(log(as.numeric(INTCdC)))
> INTCdCReturnsNum[1:2]
[1] -0.005204724 -0.001649313
```

Figure 1.40. Unexpected Behavior in Computing Log Returns in `xts` Objects

```
> party <- c("R","D","D","R","D","R","R","D","R","R","D","R","D","D")
> people <- data.frame(sexfac, party)
> head(people, n=3)
  sexfac party
1      m     R
2      m     D
3      f     D
> class(people$party)
[1] "factor"
```

If there are no other variables in the data frame `people`, then all of the information contained in the 14×2 data frame is contained in a two-way contingency table shown in Table 1.7.

Table 1.7. Contingency Table

		party	
		D	R
sex	f	5	3
	m	2	4

The contingency table shown in Table 1.7 can be produced by the R function `table()` operating on the two vectors `sexfac` and `party`, or on the data frame `party` containing them.

The kinds of operations performed by the `apply()` function, or similar functions that operate on specific dimensions of an array are common in working with tables, and there are R functions for operations on objects of class `table`. One of the most useful is the `addmargins()` function, which computes the row and column sums of the table. (This function does the computations of both of the general two-dimensional array functions, `rowSums()` and `colSums()` mentioned earlier.)

```
> addmargins(table(people))
     party
sexfac   D  R Sum
  f      5  3   8
  m      2  4   6
  Sum    7  7  14
```

The data input to the `table()` function are either vectors, matrices, or data frames. The function assumes that each vector or column in a matrix or variable in a data frame is a categorical variable, and hence `table()` identifies each level or each different value within each variable and treats them separately. The results of the `table()` function is an array whose number of dimensions is the number of variables, and the range of each dimension is the number of levels of each variable.

The data in contingency tables are convenient visual presentations of the numbers of observations in the various cells. By comparing the relative counts in the cells, we can assess whether or not the factors are related to each other. If the factors are not related, then we can work out the expected relative counts, and compare those to the counts observed. It would appear, for example, from the contingency table in Table 1.7 that there may be a relationship between party and sex; there seems to be more "D" corresponding to "f". (Using the

margin sums, we see 5/8 of "f" are "D", whereas only 1/3 of "m" are "D".) The sample is rather small, and we do not know how the sample was collected, so we should be careful in making inferences based on those data.

There are various statistical tests that allow us to formalize the assessment of relationships among the variables. The most common, but perhaps not the best, statistical test is called *Pearson's chi-squared test*. It is implemented in the `chisq.test()` function.

Multidimensional Tables

Most statistical analyses of tables are performed on two-way contingency tables, indeed, most multivariate analysis is bivariate analysis, that is, pairwise variable analysis. A multi-way table is a simple concept that extends the ideas of a table to multiple factors. The R functions operate on tables of any dimension.

The tables in the examples above may be extended to account for another variable. This yields a three-way table as below.

```
> sexfac <- c("m","m","f","m","f","f","m","f","m","f","f","f","m","f")
> party <- c("R","D","D","R","D","R","R","D","R","R","D","R","D","D")
> voted <- c("N","Y","Y","N","Y","N","N","N","Y","Y","N","Y","Y","N")
> votes <- data.frame(sexfac, party, voted)
> table(votes)
, , voted = N
      party
sexfac D R
     f 3 1
     m 0 3
, , voted = Y
      party
sexfac D R
     f 2 2
     m 2 1
```

The `addmargins()` function operates in the usual way on multi-way tables. Statistical analyses such as chi-squared tests and other nonparametric analyses are performed on pairwise factors in tables, however.

Reshaping of tables as with data frames discussed on page 79 is a common operation, and the `reshape()` function as well as the functions in the `reshape2` and `tdyr` packages are useful for this purpose.

Exercises: R Objects and Classes

As in previous exercises, these exercises may require computations on data. If the emphasis is on the method rather than the actual data, we will often use artificially generated "random" data. Standard normal random numbers serve this purpose well. An exercise may ask you to generate n normal random

numbers with a mean of m and a standard deviation of s. To do this for a sample of size 50, a mean of 100, and a standard deviation of 3, you can use the simple R statements

```
n <- 50
m <- 100
s <- 3
set.seed(12345)
x <- s*rnorm(n) + m
```

1.3.1. Suppose we have a vector of positive integers, `x`, and we want to form two vectors, `y` and `z`, where `y` consists of the ordered elements from `x` that are evenly divisible by 4, and `z` consists of the remaining elements from `x`.
Write R expressions to form `y` and `z`.
Test your code with an example.

1.3.2. Let A and B be matrices and let x, y, and z be vectors as shown below.

$$A = \begin{bmatrix} 3 & 2 \\ 2 & 7 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix},$$

$$\begin{aligned} x &= (2, 4), \\ y &= (-3, 1), \\ z &= (1, 2, 3). \end{aligned}$$

(All vectors are "column" vectors, no matter how they are displayed. I will occasionally refer to "row" vectors to refer to vectors that correspond to rows of a matrix.)
Use R to compute all Hadamard and Cayley matrix/vector products involving no more that three operands and for which the operands are conformable, such as $A \odot A$, AA, $A^{\mathrm{T}}A$, $A^{\mathrm{T}} \odot A$, $x^{\mathrm{T}}Ax$, AB, and so on.

1.3.3. `ts` objects.
In this exercise, we need some data, but the data themselves are not important, so we will just use normal random numbers.
Interpret the monthly and quarterly data in this exercise as observations made at the end of the month or the end of the quarter.

a) Make four random `ts` objects, each beginning in May, 2020 and going through September, 2021, with a frequency of 12.
   - Closing prices at the end of the month: `prmon`. Use as the data a sample of normal random numbers with a mean of 100, rounded to two decimal points.
   - High prices for the month: `prhimon`. Use as the data the same data as in `prmon` plus 5.

- Low prices for the month: `prlomon`. Use as the data the same data as in `prmon` minus 5.
- Volume traded during the month: `volmon`. Use as the data a sample of normal random numbers with a mean of 10000 and a standard deviation of 3, rounded to integers.

Display each of the sets of data.

b) Form a `ts` object `prqrt` that represents the quarter-end data in `prmon`. (The first observation in `prqrt` is the datum for June, 2020, that is, the second quarter.)

c) Aggregate (appropriately!) the monthly high-price data into quarterly data in an `xts` object `prhiqrt`. (The first observation in `prhiqrt` is the third quarter of 2020.)

d) Aggregate (appropriately!) the monthly low-price data into quarterly data in an `xts` object `prloqrt`.

e) Aggregate (appropriately!) the monthly volume data into quarterly data in an `xts` object `volqrt`.

f) Form an `mts` object that contains the data from `prqrt`, `prhiqrt`, `prloqrt`, and `volqrt` for all quarters for which they all have data.

1.3.4. Assume that we have an R data frame `Xdf` with a numeric variable `weight`, and a factor variable `sizes` with levels "small", "medium", and "large". The general characteristics of `Xdf` are shown by the `head()` function:

```
> head(Xdf, n=4)
  weight  sizes
1     45  small
2     32  small
3     90  large
4     76 medium
```

a) We want the mean weight in each category.
Write an R statement to obtain these values and store them in an array called `meanwts`.

b) Write an R statement that will print a phrase stating the mean weight of the items with a specified level of the `sizes` factor. Recall that the factor levels are sorted, so your R statement should just require the index of the level; for example, if the index is specified as `i=1` then the "large" sizes are indicated. The statement is of the form

The mean weight of the large items is 1.23.

where large is the specified level of the factor variable, and 1.23 is the computed mean at that level, rounded to two decimal places.
Note that you statement has an index `i`, and that it uses the `sizes` variable and the `meanwts` object above.
Make a small dataset to test your R code.

1.3.5. Use the data shown in shown in Tables 1.4 and 1.5 to form a single R data frame called `Stocks` that has four variables, `Date`, `Sector`, `Stock`, and `Price`. There are ten observations.
The `Date` variable is of class `Date`; the `Sector` variable is of class `factor`, the `Stock` variable is of class `factor`, and the `Price` variable is of class `numeric`.
Print the data frame.

1.3.6. Using the data frame in Exercise 1.3.5, form a data frame named `Prices` that consists of five time series of the prices of the five stocks. The variable names are `Date`, as before, and the names of the stocks `AAPL` and so on. Except for the dates, all of the data are prices; there is no variable named "`Price`". This dataset is shown in Table 1.6. (Each time series has only two observations; but the point of the exercise is to be able to do it.)
Print the data frame.

1.3.7. Make an `xts` object `Pricesxts` from the data frame `Prices` in Exercise 1.3.6. (That data set is shown in Table 1.6.)
"`Date`" is no longer one of the variables; it is an index in `Pricesxts`.
Print the `xts` object.

1.3.8. `xts` objects.
In this exercise, we will create an `xts` object that contains the same data as the `ts` objects in Exercise 1.3.3.
Instead of the four random `ts` objects representing monthly data beginning in May, 2020 and going through September, 2021, we will form a single `xts` object with variable names that correspond to the standard names in an OHLC dataset, A.Open, A.High, A.Low, A.Close, A.Volume, and we will set dates that correspond to trading days. (We will exclude weekend days, but we may include holidays on which the market is closed; otherwise, doing the exercise would require a list of trading holidays, and that is not relevant to the point of the exercise.)

a) Form an R vector called `Date` that consists of the dates of the last non-weekend day in each month from May, 2020, through September, 2021. (Although it may be more fun to determine the POSIX form of the last day in each month using the rules of the Gregorian calendar, you may assume that the dates are `"2020-05-31"`, `"2020-06-30"`, and so on.)

b) Form simple numeric vectors `A.Open` and `A.Close` with entries that correspond to the data in `prmon` in Exercise 1.3.3. (We just let the opening price for the month be the same as the closing prices. While it is unlikely that real data would ever be this way, it makes no difference for the exercise.)
Now form simple numeric vectors `A.High`, `A.Low`, and `A.Volume` with entries that correspond respectively to the data in `prhimon`, `prlomon`, and `volmon` in Exercise 1.3.3.

c) Form a single OHLC `xts` dataset `Adata` with monthly data corresponding to the five time series created in Exercise 1.3.8b, and ordered by the dates created in Exercise 1.3.8a.
d) Now produce an OHLC `xts` dataset of quarterly data corresponding to the data in `Adata` in Exercise 1.3.8c.
e) Form the subset of `Adata` in Exercise 1.3.8c that consists only of months in 2020.

1.4 R Functions

Actions on objects are performed in R by *functions*. As we have seen, there are many "built-in" functions, such as `sqrt()`, `exp()`, and so on, that come with the basic R systems. There are many packages that provide additional functions, and the user can also write functions.

1.4.1 General Properties of R Functions

A function has a name and may accept arguments that are to be operated on. The arguments are enclosed in parentheses, and may be positional or named. Named arguments may have default values. An R function may operate differently on different types of objects.

In this section, we after reviewing some of the general properties of R functions, we describe some useful functions for mathematical computations and manipulations, some useful statistical functions, and then some R functions for graphics. There are many technical details of functions that we will ignore. We emphasize functions that are in the basic packages, rather than among the thousands of user-contributed packages. We will continue to refer to the functions that are included in the basic R system (the `base` package, the `stats` package, and so on) by the term "built-in", to distinguish them from functions included in other packages that must be loaded explicitly. (Some R functions are of type `builtin`, but others are not, even some of the ones that I call "built-in" because they are included in the basic packages. Some of these built-in functions are of type `closure`, rather than of type `builtin`. User-written functions are also of type `closure`. The R code forming many functions of type `closure` can be displayed by entering the name of the function without parentheses.)

Most functions in R are *generic*, in the sense that what the function does depends on the class or even the mode or type of the object(s) passed as the argument(s) to the function. The function `exp()` is a simple example.

In `exp(x)`, `x` must be of mode `numeric`, `complex`, or `logical`. If `x` is of class or mode `character`, the function will generate an error, because the exponential of a non-numeric quantity does not make sense.

If `x` is of mode `numeric` or `logical`, then the result `exp(x)` is of mode `numeric`; if `x` is of mode `complex`, then `exp(x)` is of mode `complex`. If `x` is

of class `numeric`, then `exp(x)` is of class `numeric`; If `x` is of class `matrix`, as above, then `exp(x)` is also of class `matrix`.

Statistical functions in R such as `var()` for variance may compute different quantities depending on the class of the argument. If `x` is a numeric vector `var(x)` is just the same variance, while if `x` is a matrix, `var(x)` is the variance-covariance matrix of the variables in the columns of `x`. The R functions `cov()` and `cor()` produce single scalar results if their arguments are two vectors. If their arguments are matrices, they produce matrices, respectively, the variance-covariance matrix (the same as `var()`) or the correlation matrix.

As we have mentioned, missing values occur often in statistical datasets. Sometimes the appropriate way to handle missing values is just to remove them from the computations, as we did in the example on page 38 using the `na.rm()` argument. Many other statistical functions offer more options for handle missing values; for example, in computing correlations, sometimes we want to compute the correlations for all pairs without a missing value ("pairwise"), but other times we want to omit the complete observation if one field contains a missing value ("casewise"). We will consider some of the options in Section 7.4.

The fact that most R functions that generally operate on a single quantity immediately accept an array and perform the operations on each element in the array, producing an array of results is also an example of the generic aspects of the function. Figure 1.2 shows a simple example of this when an atomic vector of length greater than 1 is used as an argument for a function that in a simpler form operates on a scalar. In that case, the *vectorized* operation is much more efficient than it would be to use `sqrt()` on each element individually.

Although R provides program control structures for loops (`for()` and `while()`), loops should be used only when a single operation is not available. In problem-solving, we sometimes think about the individual low-level steps. It is almost always better, at least in the initial approach, to think at the highest level relevant to the problem. A simple example here is the multiplication of two matrices, `A` and `B`. This operation is performed by multiplying corresponding elements in a row of `A` and a column of `B`, accumulating the sum those products, then moving on to the next column in `B` and doing the same thing, then moving back to the next row of `A` and repeating this process. In elemental computations, this is done in three nested loops of computations. In R, it is one operation: `A%*%B`.

We think in terms of matrices, not in terms of rows and columns or in terms of elements.

1.4.2 Some Useful R Functions

R provides a number of useful functions for the common computations in finance, applied mathematics, and statistical analyses. We have discussed some

of these functions already. The output of an R function can immediately be included as an argument to another function, or even to the same one.

There are utility functions for most common operations, including reading and writing. We will describe some of the variations on reading in data in Section 1.6.

In this section, we will mention several R functions, categorized by their type of usage.

As we have mentioned, after learning the basic syntax and program structure, "learning R" consists primarily of learning the functions that perform the various tasks.

Constructor Functions

A number of R functions, such as `c()`, `character()`, `array()`, `list()`, `matrix()`, and `data.frame()`, are constructor functions; that is they are used to construct R objects. Some of the constructor functions build objects of a particular class, type, and mode. (We can use the `class()` function to determine the class of the object produced.)

Some R functions may require an object of a particular class, type, and/or mode. Corresponding to many constructor functions, there are conversion functions ("`as.`" functions). The "`as.`" function does not change the class of its argument. We saw examples of the `as.complex()` function in Figure 1.8 and the `as.character()` function in Figure 1.9.

For some built-in classes, there are logical test functions ("`is.`" functions), `is.numeric()`, `is.matrix()`, and so on.

Functions for Operations on Numeric Vectors

R provides several utility functions for processing a numeric vector. Some, with obvious names, are `length()`, `sum()`, `mean()`, and `var()`. These functions return single values that are summary statistics for the several values in a vector.

Another useful R function for operations on elements of vectors is the `ifelse()` function.

Functions for Operations on Numeric Time Series

To summarize or just to manipulate the values in a time series vector, instead of a single value, a vector of values is necessary. We will discuss processing of time series data in more detail in Chapters 2 and 5, but here we will mention a useful function for working with time series.

A time series is a sequence

$$\ldots, x_{-2}, x_{-1}, x_0, x_1, x_2, \ldots$$

A useful operator for time series is the *backward difference* operator, $\triangle(\cdot)$, defined by

$$\triangle(x_t) \equiv x_t - x_{t-1}. \tag{1.2}$$

The corresponding R function is `diff()`, which produces the differences between successive points in a sequence. Differences between consecutive terms are called *differences of lag 1*; ones between a term and the k^{th} subsequent term are called *differences of lag k*. The `diff()` function has an additional argument to specify the lag. The default lag is 1.

Notice that the sum of all first order differences is equal to the difference from the first to the last.

```
> x <- c(10,13,11,15)
> xdiffs <- diff(x)
> xdiffs2 <- diff(x, lag=2)
> x
[1] 10 13 11 15
> xdiffs
[1]  3 -2  4
> xdiffs2
[1] 1 2
> sum(xdiffs)
[1] 5
> x[4]-x[1]
[1] 5
> diff(x, lag=3)
[1] 5
```

The result of `diff()` is one element shorter than the original vector, and with `lag=`k, it is k elements shorter. This is often the case with functions of time series; a difference or a return cannot be computed for the first data point. Smoothing computations, such as moving averages, cannot be computed until enough data to complete the initial window have been accumulated. (We will discuss smoothing of time series in Chapter 5.) In cases like these, there is a choice to be made: whether or not the output vector be shorter than the vector or other object containing the data. Obviously, the computations cannot be performed until enough data is available. In computer operations, however, we may choose to work with arrays of different sizes, but sometimes it is more convenient to insert NA where there is no legitimate value, so as to have arrays of the same size. In many functions in some packages for financial applications, vectors such as differences or moving averages are padded with NA so as to have vectors of the same length.

Notice that for a numeric vector, the `diff()` function merely performs the operation in equation (1.2):

```
> x[-1] - x[-length(x)]
[1]  3 -2  4
```

If `x` is a `ts` object, the result of `diff(x)` is a `ts` object; however, the result of `x[-1]-x[-length(x)]` is just a `numeric` object.

The trivial differences (in difference functions!) have practical consequences for the R user, as we see in computation of the return on a financial asset.

Returns on Financial Assets

The *return* is the relative change in value of the asset plus any income that the asset has generated over some time period. In fixed-income asset such as bonds, the return results primarily from the accumulated interest. In negotiable equity assets such as common stock, the returns result primarily from change in the market value of the stock. In this book our emphasis will be on equity assets.

Consider a sequence of prices, $P_1, P_2, \ldots, P_n$, of a common stock that pays no dividends. The *one-period simple return*, from time $t-1$ to time t, is

$$R_t = \frac{P_t - P_{t-1}}{P_{t-1}}. \tag{1.3}$$

If we have n prices, we can compute $n-1$ one-period returns. (We assume that the periods are of the same length, say one day. This is common in financial applications; we compute "daily" returns, although the length of time from one trading day to the next is not constant. We ignore the "weekend effect", which also may be due to holidays.)

Now, suppose we have a time series of prices of this stock stored in an R vector, `prices`. The numerator in equation (1.3) can be obtained using `diff()`, and since R does element-wise multiplication with numeric vectors, we can just divide the differences by the prices. Hence, we compute the vector of one-period simple returns as

```
sreturn <- diff(prices)/prices[-1]
```

If the difference function produced a vector with an NA in the first position, we would not need to remove the first element in the denominator. The resulting vector of simple returns, however, would also have an NA in the first position. Either way is acceptable; we just need to know which way the software handles the situation.

Simple returns, as in equation (1.3), are often quoted in financial reports. They are easy to understand. They are the same as percentage returns, after adjusting by 100%.

If we know the price P_{t-1} and the rate of return R_{t-1} at time $t-1$, at time t, we have the price at time P_t

$$P_t = P_{t-1}(1 + R_{t-1}), \tag{1.4}$$

by rearranging equation (1.3). Increasing the base price in this way for the next time period is called *compounding*.

The most common issues in working with returns involve compounding. If the price from time $t-1$ to t increases by $R_{t-1}P_{t-1}$, the price at time $t+1$, should increase further by $R_t(1+R_{t-1}P_{t-1})$. This leads to the more general formula for compound growth, which is often stated in annualized values.

Using common notation, let P be the value of an asset, let r be the annualized rate of growth due to interest, dividends, or just change in market value and assume that this amount is compounded c times per year. Assuming constancy of this process, after t years, the value of the asset is

$$P(1+r/c)^{ct}. \tag{1.5}$$

The frequency of compounding changes the rate of growth, and leads to different values for an "annual rate".

It is now common to compound continuously, and much of financial analysis is done under the assumption that the compounding is continuous. In continuous compounding, the c in equation (1.4) goes to ∞, which formally results in

$$P\mathrm{e}^{rt}, \tag{1.6}$$

which starting with P_{t-1} and ending one year later with P_t, analogous to equation (1.3), yields

$$r_t = \log(P_t) - \log(P_{t-1}). \tag{1.7}$$

This is called the *log return*.

The properties of returns are some of the most important subjects in finance. The standard deviation of the returns is called the *volatility*, and it is topic we will encounter often in this book.

A log return can be defined over any interval. An important property of log returns is that they are additive. In the notation above, where t represent time in any fixed-length interval, the log return over two consecutive time points, $\log(P_t) - \log(P_{t-2})$, is just the sum of the log return over the single-period intervals:

$$\log(P_t) - \log(P_{t-2}) = (\log(P_t) - \log(P_{t-1})) + (\log(P_{t-1}) - \log(P_{t-2}))\,.$$

Now suppose we have a time series of prices stored in an R vector, `prices`, as above. The log returns at the frequency of the time series can be obtained easily.

```
logreturn <- diff(log(prices))
```

If `prices`, contains prices $P_1, P_2, \ldots, P_n$, then `logreturn` is a vector of length $n-1$ containing the log returns.

We will encounter snippets of R code for both simple and log returns often in this book.

This illustrates a simple property of R functions. The output of one R function, `log()` in this case, can be immediately input as the argument to an R `diff()` in this case.

Some Common Mathematical Functions

R includes many built-in functions for various mathematical operations with obvious names, such as `sqrt()`, `log()`, `exp()`, `sin()`, and so on. These and similar mathematical functions are called "elementary functions". The elementary functions are of type `builtin`.

There are several other built-in functions for mathematical operations such as `polyroot()` to find all complex roots of a polynomial, and `uniroot()` to find a root of a general function.

The Matlab system mentioned above has many mathematical functions, especially in the area of linear algebra. The `pracma` package, written by Hans Werner Borchers, contains R functions with the same name and function as many of the Matlab functions. Not only do these functions extend the computational capabilities of R, they also serve to make some Matlab or Octave code transportable into R code. If a Matlab name conflicts with a standard R name, the R function in `pracma` is capitalized so as to avoid the conflict. A list of names of all Matlab functions that are implemented in `pracma` can be obtained by the `matlab()` function without arguments.

One of the most important design features of functions in R is that in most cases, the arguments to the functions can be arrays (where it makes sense for them to be). The value of the function is an array of the appropriate shape.

Other Mathematical Objects and Operations

R provides facilities for many operations in addition to numerical computations. One of the most common non-computational operations is sorting. R does sorting very efficiently. Often we may just wish to check if an object is sorted. A useful R function is `is.unsorted()`, which checks if an object is sorted without sorting it.

R does not have an object class of "set" (although we could create one). R, however, does provide functions for the standard set operations such as union and intersection, and the logical operators of set equality and inclusion, as illustrated in Figure 1.41.

The R function `unique()` is useful for removing duplicate elements in an R object.

Some Useful Statistical Functions in R

R has a number of functions for computing simple statistics. These R functions have mnemonic names, such as `mean()`, `var()`, `cov()`, and `cor()`.

```
> S1 <- c(1,2,3,2)
> S2 <- c(3,2,1)
> S3 <- c(4,3,2,1)
> setequal(S1,S2)
[1] TRUE
> union(S1,S3)
[1] 1 2 3 4
> union(S1,S1)
[1] 1 2 3
> intersect(S1,S3)
[1] 1 2 3
> intersect(S1,S1)
[1] 1 2 3
> 1 %in% S1
[1] TRUE
> 5 %in% S1
[1] FALSE
```

Figure 1.41. Set Functions Operators in R: `setequal`, `union`, `intersect`, `%in%`()

R also has a number of statistical functions for other common analyses, which may produce extensive summary statistics. Two of the most useful of these functions are `lm()` for "linear model" and `glm()` for "generalized linear model". These functions produce coefficient estimates, various sums of squares, various t and F statistics along with their p-values, and a variety of other statistics. The R function `summary()` can be used to extract the more commonly-used output statistics from many R statistical functions such as `lm()` and `glm()`.

Some of the more basic statistical functions are not included in the `stats` package; for example, there is no function to compute the skewness or kurtosis statistics. Functions for these statistics are available in a number of packages. The `e1071` package, for example, contains the `skewness()` and `kurtosis()` functions, along with several other functions for basic statistical computations. (The `kurtosis()` function computes the excess kurtosis.) Also, the `pracma` package mentioned above contains functions for simple statistical computations, such as moving averages.

R Functions for Probability Distributions

There are a number of built-in functions for computations involving several common probability distributions. The function names consist of a root name that identifies the family of distributions, such as `norm` or `t` and a prefix to

determine the type of function, density, CDF, or quantile, or a function to simulate random numbers. Functions whose names begin with "d" compute the density or probability function, with "p" compute the CDF (the probability of a value less than or equal to a specified point), and with "q" compute the quantile. For example, `pnorm(x)` computes the probability that a standard normal random value is less than or equal to `x`.

R also provides functions for simulating random samples from these distributions (see Section 3.2). Functions to generate random numbers have names that begin with "`r()`".

The root names of common distributions are shown in Table 1.8.

The arguments to the R function are the point at which the result is to be computed, together with any parameters to specify the particular distribution within the family of distributions. For example, the root name of the R functions for the Poisson distribution is `pois()`. The Poisson family of distributions is characterized by one parameter, often denoted as "λ". Hence, if `lambda` (which may be a numeric array) is initialized appropriately,

$$\begin{aligned} &\texttt{dpois}(\texttt{x}, \texttt{lambda}) = \lambda^x \mathrm{e}^{-\lambda}/x! \\ &\texttt{ppois}(\texttt{q}, \texttt{lambda}) = \textstyle\sum_{x=0}^{q} \lambda^x \mathrm{e}^{-\lambda}/x! \\ &\texttt{qpois}(\texttt{p}, \texttt{lambda}) = q, \quad \text{where } \textstyle\sum_{x=0}^{q} \lambda^x \mathrm{e}^{-\lambda}/x! = p, \end{aligned} \tag{1.8}$$

where all elements are interpreted as numeric arrays of the same shape. The values of the mean and standard deviation are taken from the *positional relations* of the arguments.

Various R functions for computations involving a Poisson random variable with parameter $\lambda = 5$ are shown for example in Figure 1.42.

```
> dpois(3, lambda=5)
[1] 0.1403739      #  probability that a random variable equals 3
> ppois(3, lambda=5)
[1] 0.2650259      #  probability less than or equal to 3
> qpois(0.2650259, lambda=5)
[1] 3              #  quantile corresponding to 0.2650259
```

Figure 1.42. Values in a Poisson Distribution with $\lambda = 5$

For the univariate normal distribution, there are two parameters, the mean and the variance (or standard deviation). The root name of the R functions is `norm()`, and the names of the parameters are `mean()` and `sd()` (for standard deviation *not* the variance). Hence, if the variables `mean()` and `sd()` (which may be numeric arrays of the same shape) are initialized to m and s,

Table 1.8. Root Names and Parameters for R Functions for Distributions

Continuous Univariate Distributions		Discrete Distributions	
`unif`	uniform	`binom`	binomial
	`min=0, max=1`		`size, prob`
`norm`	normal	`nbinom`	negative binomial
	`mean=0, sd=1`		`size, prob`
`lnorm`	lognormal	`multinom`	multinomial
	`meanlog=0, sdlog=1`		`size, prob`
`chisq`	chi-squared		*only the* `r` *and* `d` *versions*
	`df, ncp=0`	`pois`	Poisson
`t`	t		`lambda`
	`df, ncp=0`	`geom`	geometric
`f`	F		`prob`
	`df1, df2, ncp=0`	`hyper`	hypergeometric
`beta`	beta		`n, m, k`
	`shape1, shape2`		
`cauchy`	Cauchy		
	`location=0, scale=1`		
`exp`	exponential		
	`rate=1`		
`gamma`	gamma		
	`shape, scale=1`		
`gumbel`	Gumbel		
{`evd`}	`loc=0, scale=1`		
`laplace`	double exponential		
{`rmutil`}	`m=0, s=1`		
`logis`	logistic		
	`location=0, scale=1`		
`pareto`	Pareto		
{`EnvStats`}	`location, shape=1`		
`stable`	stable		
{`stabledist`}	`alpha, beta,`		
	`gamma=1, delta=0`		
`weibull`	Weibull		
	`shape, scale=1`		
Generalized Distributions		**Continuous Multivariate Distributions**	
`gl`	generalized lambda	`mvnorm`	multivariate normal
{`gld`}	`lambda1=0, lambda2=NULL,`	{`mvtnorm`}	`mean=0, sigma=`I
	`lambda3=NULL, lambda4=NULL,`		*only the* `r` *and* `d` *versions*
	`param="fkml", lambda5=NULL)`	`dirichlet`	Dirichlet
`ged`	generalized error	{`MCMCpack`}	`alpha`
{`fGarch`}	`mean=0, sd=1`		*only the* `r` *and* `d` *versions*
`gpd`	generalized Pareto	`mvt`	multivariate t
{`evir`}	`xi, mu=0, beta=1`	{`mvtnorm`}	`sigma, df, delta=NULL`
`snorm`	skewed normal		*only the* `r` *and* `d` *versions*
{`fGarch`}	`mean=0, sd=1, xi=1.5`		

$$\begin{aligned}
&\texttt{dnorm}(\texttt{x}, \texttt{mean}, \texttt{sd}) = \tfrac{1}{\sqrt{2\pi}s}\mathrm{e}^{-(x-m)^2/2s^2} \\
&\texttt{pnorm}(\texttt{q}, \texttt{mean}, \texttt{sd}) = \textstyle\int_{-\infty}^{q} \tfrac{1}{\sqrt{2\pi}s}\mathrm{e}^{-(x-m)^2/2s^2}\mathrm{d}x \qquad (1.9) \\
&\texttt{qnorm}(\texttt{p}, \texttt{mean}, \texttt{sd}) = q, \quad \text{where } \textstyle\int_{-\infty}^{q} \tfrac{1}{\sqrt{2\pi}s}\mathrm{e}^{-(x-m)^2/2s^2}\mathrm{d}x = p,
\end{aligned}$$

where all elements are interpreted as numeric arrays of the same shape. (Because this is a continuous distribution, the value `dnorm(x, mean, sd)` is not a probability, as `dpois(x, lambda)` is.)

The values of the mean and standard deviation are taken from the *positional relations* of the arguments, but the keyword arguments allow different forms of function calls. If the mean is stored in the variable `m` and the standard deviation is stored in the variable `s`, we could use the function reference

```
dnorm(x, sd=s, mean=m)
```

The parameters often have defaults corresponding to "standard" values. For the univariate normal distribution, for example, the default values for the mean and standard deviation respectively are 0 and 1. This is the "standard normal distribution". If either parameter is omitted, it takes its default value. The function reference

```
dnorm(x, sd=s)
```

is for a normal distribution with a mean of 0 and a standard deviation of `s`.

The standard normal distribution is so important that we define standard symbols for its PDF and CDF. The PDF is denoted by $\phi(\cdot)$ (this is `dnorm()`$(\cdot)$), and the PDF is denoted by $\Phi(\cdot)$ (this is `pnorm()`$(\cdot)$).

The Black-Scholes Formulas for Options Prices

An objective in financial analysis is to determine the "fair price" of an asset. The fair price depends on the nature of the asset and what is to be done with the asset. Even for relatively simple assets, this is not an easy task. The fair price of a share of stock may depend on book value, realized earnings, of expected earnings.

Derivative assets depend on other, underlying assets; hence, the fair price of a derivative asset depends on the fair price of the underlying. A common type of publicly-traded derivative is a *call* or a *put* option on shares of a publicly-traded stock. A call is the right to buy the stock at a specific price a put is the right to sell the stock at a specific price.

The parameters of the option include the number of shares, the price at which they can be bought or sold, and at what times the purchase or sale can be made. An option is *settled* or *exercised* when the purchase or sale is made to close out the option. (Related calls and puts involve proprietary indexes; the option is settled by cash transfer at expiry. An index cannot be bought or sold.) Most publicly-traded stock options can be exercised any time before their expiration and most index options can be settled only at the time of their expiration. Options that can be exercised at any time before their expiration

are called "American" options, and those that at settled only at expiration are called "European" options.

In the following, we will use notation that is standard in analysis of option pricing:

t	current time
S_t	price of a unit of the underlying at at time t
K	strike price
r	risk-free annual interest rate
σ	volatility of the underlying (standard deviation of annual log returns)
T	expiration time
$\Phi(\cdot)$	CDF of standard normal distribution

(1.10)

At expiration, the value of the call option is

$$C_T = \begin{cases} S_T - K & \text{if } S_T \geq K \\ 0 & \text{if } S_T < K, \end{cases} \tag{1.11}$$

and the value of the put option is

$$P_T = \begin{cases} K - S_T & \text{if } S_T \leq K \\ 0 & \text{if } S_T > K. \end{cases} \tag{1.12}$$

We define the *fair price of an option* as its expected value at exercise discounted to the present.

The expected value depends on the expected price of the underlying, and that depends on a stochastic model of the prices of the underlying over the time period leading up to expiration. A simple model developed by Black, Scholes, and Merton is a stochastic differential equation involving a geometric Brownian motion (see Black and Scholes, 1972, and Merton, 1973).

For a European option, determining the expected value involves solving the stochastic differential equation subject to the boundary conditions (1.11) and (1.12). We will not show the development of the solution here, but rather just show the formulas that the solution yields.

If the stock pays no dividends, the solution involves integrating the the differential equation over the range of time from the present, t, to expiry T. Let T_t denote the amount of time measured in years from t to T; that is, $T_t = T - t$.

This integration involves integration of the normal probability density over that time range to two endpoints,

$$d_1 = \frac{\log(S_t/K) + (r + \sigma^2/2)(T_t)}{\sigma\sqrt{T_t})()} \tag{1.13}$$

and

$$d_2 = \frac{\log(S_t/K) + (r - \sigma^2/2)(T_t)}{\sigma\sqrt{T_t})()}, \tag{1.14}$$

which arise respectively from the *delta* of the option, which is the relative change in price of the option to the change in price of the underlying, and the strike price adjusted by the value of a risk-free bond paying a rate of r. The integrals of the normal density to those two points are $\Phi(d_1)$ and $\Phi(d_2)$.

The result are the two Black-Scholes formulas, for the fair value of a call,

$$C_t = \Phi(d_1)S_t - \mathrm{e}^{-r(T_t)}K\Phi(d_2), \tag{1.15}$$

and for the fair value of a put,

$$P_t = -\Phi(-d_1)S_t + \mathrm{e}^{-r(T_t)}K\Phi(-d_2). \tag{1.16}$$

These two formulas were significant theoretical advances when they were first put forth in the 1970s. There are three significant problems for applications, however. Two involve σ, the standard deviation of the distribution of log returns. The value of σ is not observable, and in any event, it is likely not to be constant. The other problem arises from the fact that the normal probability model for the distribution of returns has lighter tails than the empirical frequency distributions.

Estimation of the distribution of returns is difficult. First of all, the distribution is not constant over any significant time period. Secondly, because the elements of any set of observational data are not independent, estimation of any parameter of the distribution, such as σ, that ignores relationships within the data generally leads to poor estimates.

The simplest estimate of σ for any asset is the "historical volatility" of the price of the asset, which is the sample standard deviation of the computed returns of that asset over some recent time period.

Observed market prices of options, together with all of the observable values in equations (1.15) and (1.16) can be used to determine the value of σ that would yield an option value corresponding to the market price. The σ arrived at in this way is called the *implied volatility*. Doing this for various options yields different values of the volatility, while if the model actually were correct, the computed volatility would be the same for all options.

The `uniroot()` can be used to find the value of σ that solves the equation. To apply `uniroot()` in this way, for call options, the user would write a function of the form

```
fun <- function(x, S, K, r, Tt, Ct)
     d1 <- (log(S/K) + (r + x^2/2)*Tt) / (x*sqrt(Tt))
     d2 <- d1 - x*sqrt(Tt)
     value <- S*pnorm(d1) - K*exp(-r*Tt)*pnorm(d2) - Ct
     return(value)
```

where `S`, `K`, `r`, and `Tt` correspond to the values in the Black-Scholes formula, and `Ct` is the observed market price of the call option. The put option formula could be used similarly.

In addition to these practical considerations, the model of prices does not take into account the behavior of traders. The Black-Scholes model, nevertheless, remains one of the most useful approaches to modeling options prices.

Option pricing is one of the most important areas in finance. There are various approaches, generally based either on a discrete binary stochastic process or on a stochastic differential equation, as in the Black-Scholes approach. Careful consideration of the issues is beyond the scope of this book. The purpose here is just to illustrate a formulaic basic approach. The interested reader is referred to the many references on the topic, such as Hull (2017).

1.4.3 R Functions for Graphics

R has a rich set of functions for graphics, and there are several packages that contain more graphics functions. We will discuss the graphics capabilities of R more fully in Chapter 4, but here we will briefly introduce functions for two simple types of graphs, a histogram and a scatterplot. One class of statistical graphs shows the relative frequencies of observational data. The other main class shows the actual values of one or more observed variables. The generic `plot()` function produces a number of types of graphs depending on the class of the object to be plotted and on the options chosen in the arguments in `plot()`.

Graphs to Illustrate Frequency Distributions

Our interest is not always in the individual values of a sample of data, but rather in the general properties of the frequency distribution. These properties include the general values of the data (their "location"), how much the values are spread out (their "scale"), whether they seem to be symmetrically distributed (their "skewness"), how concentrated they are and to what extent a small proportion are far away from the central location (their "kurtosis"). These properties correspond to the sample moments we have discussed previously, but a graph that represents the frequency distribution visually can be more useful.

Histograms

An easy way to get a picture of the frequency distribution of a set of data is to form nonoverlapping subregions of the full set of data, and then to count the numbers of the observations within these bins.

A *histogram* is a simple visual representation of the density of a sample. For a sample of a single variable, a histogram has a horizontal axis that has been partitioned into nonoverlapping bins spanning the range of values in the sample.

Figure 1.43 shows a histogram of the simple daily returns for the S&P 500 Index for the year 2019.

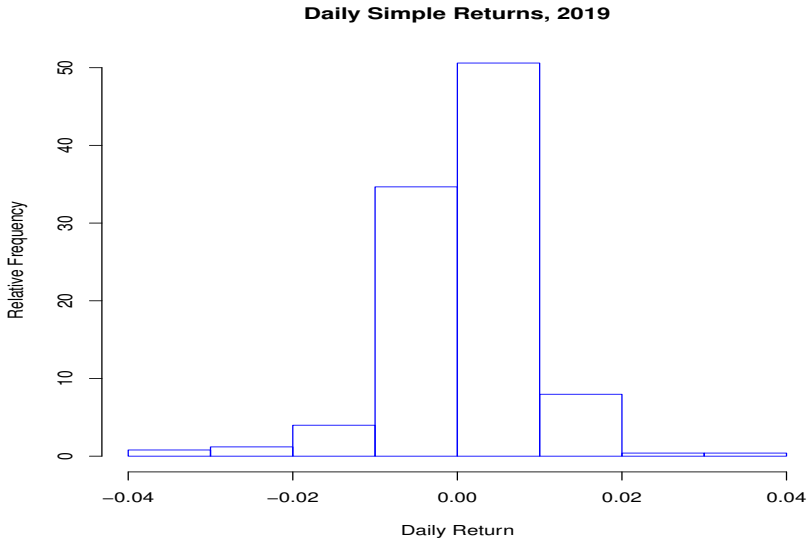


Figure 1.43. Histogram of Daily S&P 500 Simple Returns for 2019

One of the most important types of financial data is returns, either simple or log, on stocks or on stock indexes. The histogram reveals many aspects of the distribution of the daily simple returns of one of the major US stock indexes. From the histogram, we see that the returns are very heavily concentrated around 0, yet there are a few that are very extreme relative to the bulk of the returns. (This is a well-known property of equity returns.) In addition to the properties of the distribution shown in this histogram, graphical objects could be added to this plot to reveal or illustrate other aspects of the distribution. Histograms of other datasets could also be superimposed or juxtaposed to reveal comparative properties of the distributions.

The histogram in Figure 1.43 was produced by the R function `hist()`.

```
hist(SPret, freq=FALSE, main="Daily Simple Returns, 2019",
     xlab="Daily Return", ylab="Relative Frequency", border="blue")
```

In this R statement, we see three options that most R graphics functions provide: `main`, `xlab`, and `ylab`. They all have obvious meanings.

There are many variations on histograms. A histogram can show the *frequencies* of the various ranges of values, that is, the *total numbers*, or it can show the *relative frequencies*. The histogram in Figure 1.43 shows relative frequencies, which are determined so that the total area within the histogram

is 1. In the R function `hist` this choice is controlled by the logical variable `freq`. If `freq` is TRUE, frequencies are plotted; if `freq` is FALSE, as in the example on page 111, relative frequencies are plotted.

The general appearance of the histogram depends on the breakpoints. These can be controlled in the `hist` function by use of the `breaks` keyword argument. The breakpoints are not necessarily evenly spaced. If this keyword is not used in the function reference, the function will makes judicious choices, as in the code above and as shown in Figure 1.43.

In a histogram regions within the range of the data are chosen. These regions can be noncontiguous. They can even be just discrete points, which would be appropriate for categorical data. The R function `barplot()` is generally designed for discrete data. In financial applications, the plot produced is usually called a "barchart".

Kernel Density Estimates

While histograms are based on a division of the data into bins, another approach to making a visual representation of the frequency distribution is based on estimating the density of the population at a given point. This is a fundamental distinction. There is no basis for doing this with categorical data.

To estimate the density of the population at a given point using a histogram, first, the bin containing the point is identified, and then the estimated density at that point is just the height of the histogram bar containing the point. The estimate is the same for all points within a given bin.

If the objective is to estimate the density at a given point, it might be better to form the histogram bins so that one bin is centered on the point of interest. The other bins are not of interest. The estimated density only depends on the width of this bin and the count of observations within the bin.

Now suppose we do this at another point. Then we do it at yet another point.

We assume that we have a set of data, $x_1, \ldots, x_n$, that is a sample from some data-generating process, and we want to estimate the probability density of the data-generating process at a given point, x, not necessarily one of the observations in the dataset.

We form a bin enclosing x. Let h be the width of the bin. Let k be the number of observations within the bin.

Then the estimated density at x is

$$\hat{f}(x) = \frac{k}{nh}. \tag{1.17}$$

This is the same estimate as we would get from a histogram, if one of the bins happens to be centered on x.

If we define an indicator function $I(x_i)$ so that $I(x_i) = 1$ if $|x - x_i| \leq h/2$ and $I(x_i) = 0$ otherwise, then the count in the bin, k, is

$$k = \sum_{i=1}^{n} I(x_i). \tag{1.18}$$

These ideas generalize easily. Suppose that instead of 0 or 1 depending of whether or not $|x - x_i| \leq h/2$, we allow a range of values from 0 to some large value, $n/(2h)$ say, such that if x_i is close to x, a large value is assigned and if x_i is farther from x, a smaller value is assigned. Instead of a simple count of the number of close values, we assign weights to all values in the dataset based on their closeness to the point of interest, and then total up the points. We scale these distances by h. This allows us to preserved the basic form of equation (1.17).

There are many ways we could assign weights to the data point based on their scaled distance to x. Let K be a function that assigns these weights. We use

$$K\left(\frac{x - x_i}{h}\right).$$

We call K a *kernel function*.

Instead of a simple count for k as in equation (1.18), we have

$$k = \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

for use in equation (1.17), yielding the nonparametric kernel density estimate at the point x,

$$\hat{f}(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right). \tag{1.19}$$

There many reasonable possibilities for the kernel function. For example, let

$$K_{\mathrm{R}}(t) = \begin{cases} 1 & \text{if } |t| \leq 1/2 \\ 0 & \text{otherwise.} \end{cases} \tag{1.20}$$

Note that this simple *rectangular kernel* yields the estimate in equation (1.17).

Generally, a kernel function should decrease the farther $x - i$ is from x; or the closer t is to 0 in a formulation such as equation (1.20) Another choice, for example, is the *triangular kernel*,

$$K_{\mathrm{T}}(t) = \begin{cases} 1 - |t| & \text{if } |t| \leq 1/2 \\ 0 & \text{otherwise.} \end{cases} \tag{1.21}$$

The kernel function does not have to be of finite domain. The normal PDF with a mean of 0 and a standard deviation of h for example, is often used as a kernel function. As it turns out, the appearance and the statistical properties of the density estimates do not depend crucially on the kernel function. The smoothing parameter, however, can result in major differences.

The R function `density()` computes the density estimates at a number of points and produces an object of class `density`. The `density()` function

allows the user to choose the kernel and the smoothing parameter, h, called the `bandwidth`.

The generic `plot()` function produces a smooth plot of the estimates. For the S&P 500 simple returns for 2019, discussed above, we can fit the density using default settings and plot it with the R statements.

```
dens <- density(SPret)
plot(dens, main="Daily Simple Returns, 2019",
     xlab="Daily Return", ylab="Probability Density", col="blue")
```

In `plot()` function, we see three common options in R graphics, `main`, `xlab`, and `ylab`. This produces the plot shown in Figure 1.44.

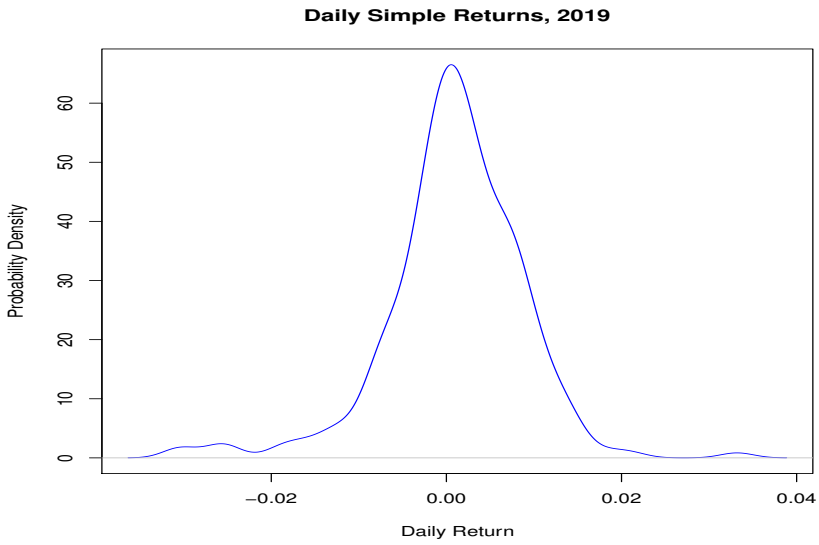


Figure 1.44. Kernel Fit of the Probability Density of Daily S&P 500 Simple Returns for 2019; Compare the Histogram in Figure 1.43

Scatterplots and Lineplots

Plots of two variables are generally produced on a pair of coordinate axes. A scatterplot of two variables displays a point for each observation, and a lineplot displays a broken line connecting various points on the graph. The broken line may be a single line and it may not connect points in the data.

Plots of just one variable are often made the same way as plots of two variables by using the index of the variable as a second variable.

Figure 1.45 shows a scatterplot of the simple daily returns for Tesla stock (TSLA) and the S&P 500 Index for the year 2019.

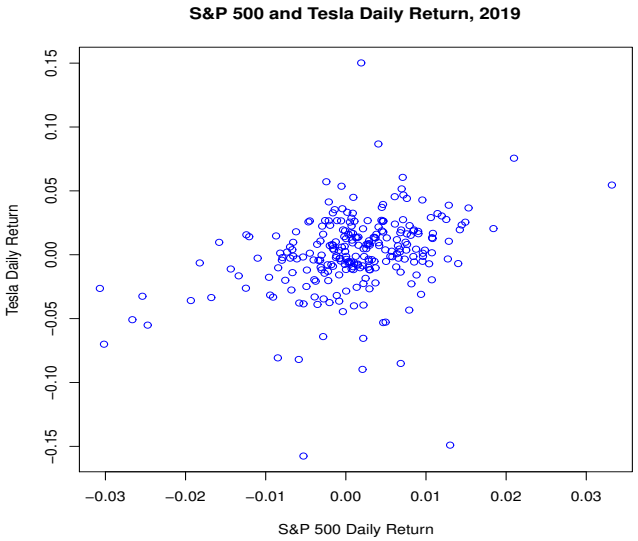


Figure 1.45. Scatterplot of Daily Tesla and S&P 500 Simple Returns for 2019

The scatterplot was produced by the R function `plot()`.

```
plot(SPret, TSret, type="p", main="S&P 500 and Tesla Daily Return, 2019",
     xlab="S&P 500 Daily Return", ylab="Tesla Daily Return", col="blue")
```

Notice that the first argument in `plot()` is plotted on the horizontal axis and the second argument on the vertical axis.

The R function `plot()` has some of the same options as the `hist()` function. Another option in `plot()` is `type`, which produces a scatterplot if `type="p"` and a lineplot if `type="p"` or both if `type="b"`. The `type` also allows for a variety of additional types of graphs. If `type` is not specified, the `plot()` function produces a plot appropriate for the class of objects being plotted. For atomic numerical vectors, the default is a scatterplot; the `type="p"` argument in the example above was not needed.

When `plot()` is used to produce a plot of a single variable, the vector containing values of that variable is used in place of the second argument, and the index of the vector is used as the first argument.

The `col` argument is common to many R graphics functions, and allows specification of color by a variety of codes, including simple English words in some cases. The `col` argument in `hist()` fills in the bars with the specified color, as opposed to the `border` argument that we used in the example above.

Adding Elements to a Graph

While the `plot()` and `hist()` functions initiate a new graph, many of the graphics functions in R allow graphical elements to be added to an existing graph. Some functions have a logical argument `add`, which if TRUE, adds the elements they produce to an existing graph. The `legend()` function can be used to annotate the graph so as to distinguish the graphical elements.

The `add` argument in `hist()` allows the addition of a histogram to an existing histogram. The first histogram must have a range that encloses the range of all subsequent histograms. This can be accomplished with the `xlim` argument in `hist()`, or just by making sure that the the range of the first histogram is at least as great as that in any subsequent histogram. Histograms of both Tesla stock and the S&P 500 Index can be produced on the same set of axes with this code

```
hist(SPret, freq=FALSE, main="Daily Simple Returns, 2019",
     xlab="Daily Return", ylab="Relative Frequency", border="blue")
     xlim=c(min(TSret),max(TSret)))
hist(TSret, freq=FALSE, add=TRUE, border="red")
legend("topleft", legend=c("Tesla","S&P 500"), col=c("red","blue"), lty=c(1,1))
```

yielding the histograms in Figure 1.46.

Four functions that are designed to add elements to an existing graph are `lines()`, `points()`, `abline()`, and `text()`, which in most cases add the obvious elements. While `lines()` adds line segments connecting points, the `abline()` function draws a line across the entire graph with an intercept `a` and a slope `b`. The additional elements merely overplot existing elements.

For example, to add a least squares regression line for Tesla returns regressed on the "market returns", following the `plot()` statement in the code above, we can use

```
lsfit <- lm(TSret~SPret)
abline(lsfit$coeff[1], lsfit$coeff[2], col="red")
```

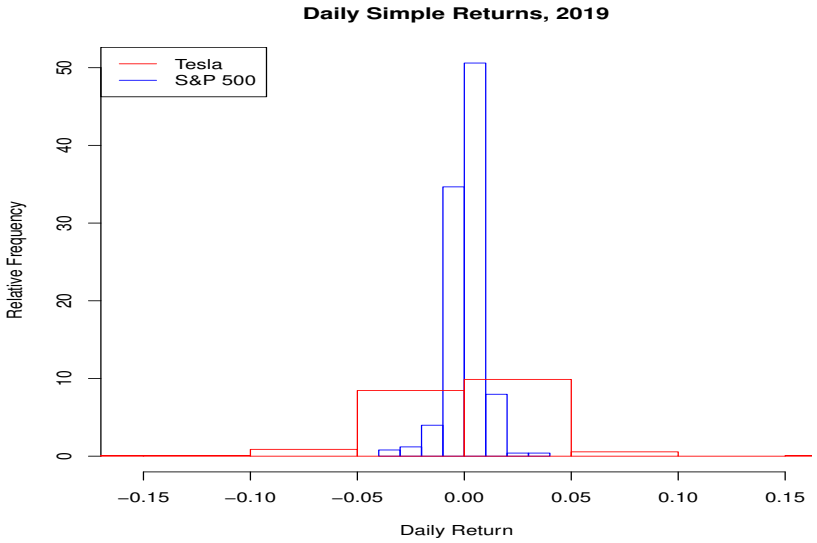


Figure 1.46. Histograms of S&P 500 and Tesla Daily Returns for 2019

The `lm()` R function computes a least squares fit of a linear regression line.

To highlight some outlying points in the graph, we can plot those points in a different color. Since the `points()` function overplots an existing graph, we can use

```
points(SPret[abs(TSret)>=0.1], TSret[abs(TSret)>=0.1], col="red")
```

Figure 1.47 shows a scatterplot from Figure 1.45 with the additional elements.

Plotting Time Series Objects

A time series plot is a lineplot in which the horizontal axis represents time. The generic `plot()` function is used to plot time series. Both `ts` and `xts` objects have their own plot methods in `plot()`. Among other aspects of the time series methods is the use of the standard `type` argument with `type="l"`. The `plot.ts()` function uses the `ts` methods on any numeric vector.

The data on the daily simple returns of the S&P 500 Index plotted in the histogram in Figure 1.43 is a time series, but the histogram does not show any aspects related to time. The returns shown the histogram are plotted as a time series as shown in Figure 1.48. It was produced by the R function `plot.ts()`, which is a method within the generic `plot()` function.

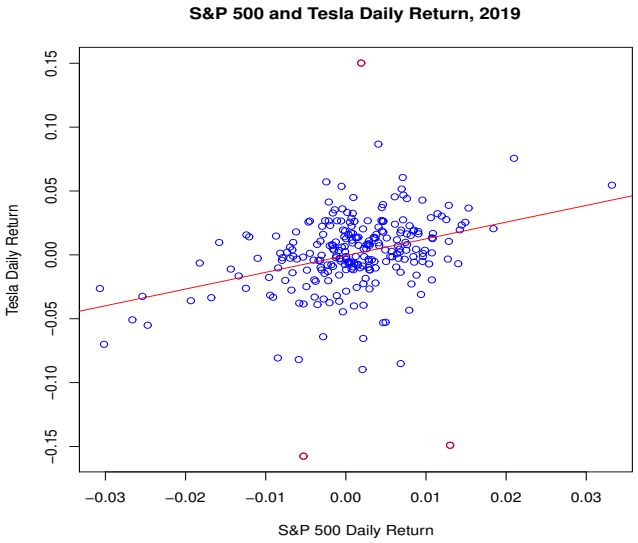


Figure 1.47. Scatterplot of Daily Tesla and S&P 500 Simple Returns for 2019

```
plot.ts(ret, main="Daily Simple Returns, 2019",
        xlab="Time (Days)", ylab="Returns", col="blue")
```

From the time series plot, we can make some general observations. The data do not seem to exhibit any trend in the mean; the data are centered about 0 for the full year. On the other hand, we note that the distribution of the data is not constant. The data are highly variable at the beginning of the year, and then go through periods of lower and higher volatility. This relates to a problem we alluded to earlier in options pricing. The volatility σ is an important component in that model, yet from the plot in Figure 1.48, it does not appear to be a constant quantity.

Some time series data, such as the returns, should be studied both as a time series and as a static sample, as in the histogram. The prices or the index itself can be studied only as time series because the series is not stationary, and so the most common type of graph is a *time series* plot, in which the horizontal axis represents time. Multiple time series can be plotted on the same graph.

Figure 1.49 shows the time series plot of the closing values of the index over the same period as the histogram and time series plot of the S&P 500 Index.

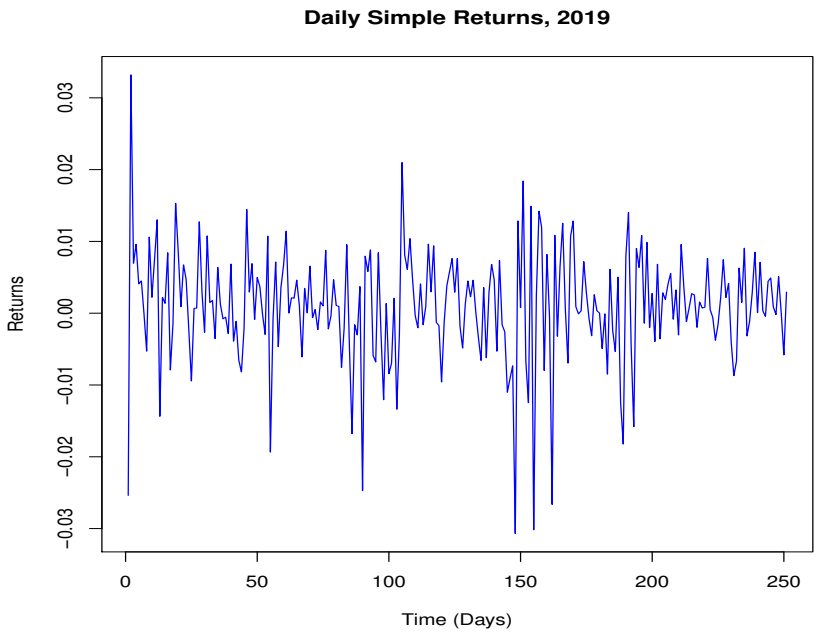


Figure 1.48. Time Series Plot of Daily S&P 500 Simple Returns for 2019

The `plot` method for an `xts` object (`plot.xts()`) accesses the dates and uses them to label tick marks on the time axis. It is much more difficult for software to choose the dates to print than it is to choose which integers to print. The `plot.xts()` function does a good job of choosing these dates, as in Figure 1.50 below.

An `xts` object that contains OHLC data can be plotted in a candlestick chart by the `chartSeries()` function in the `quantmod` package. If the OHLC dataset also contains a volume variable, `chartSeries()` plots the volume in a separate panel, as shown in Figure 1.51.

Colors can be used to enhance the information content in a candlestick plot. In Figure 1.51, green is used on days when the price increased from open to close, and red is used on days when the price decreased. The colors are used both in the candlestick and in the volume barchart.

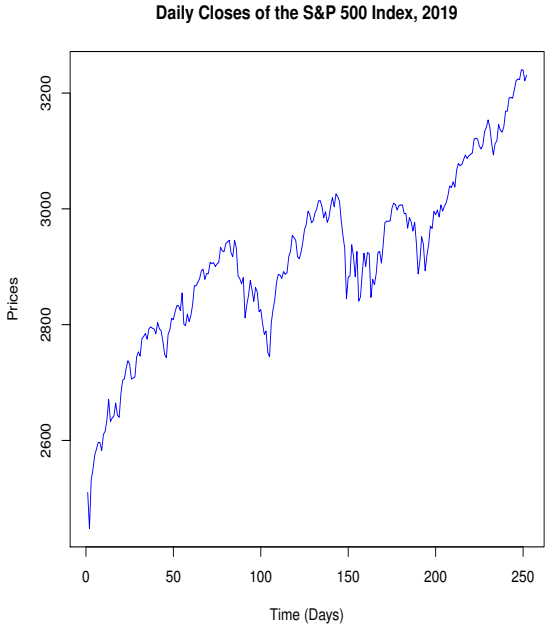


Figure 1.49. Time Series Plot of Daily S&P 500 Closes for 2019

Exercises: R Functions

1.4.1. Install and load the `e1071` package (if not already done). Generate 100 standard normal random numbers. Compute their sample skewness and sample excess kurtosis.
Are the sample values consistent with the model parameters?

1.4.2. This exercise assumes a knowledge of elementary statistics.
A common statistical hypothesis test has the null hypothesis of the form $\mu = 100$, where μ is the mean of some population from which we have a random sample. The size of the sample is important. Let n be the sample size.
The standard alternative hypothesis is $\mu \neq 100$. In this case, we perform a "two-sided" test.
Often, the hypothesis is "one-sided", of the form $\mu \leq 100$, for example.
If we assume that the population is normal, either test is performed by computing the t value and then determining its significance or "p-value"; that is, the probability of a more extreme value given the null hypothesis. The specific t distribution is determined by the degrees of freedom, which in this case is $n - 1$.

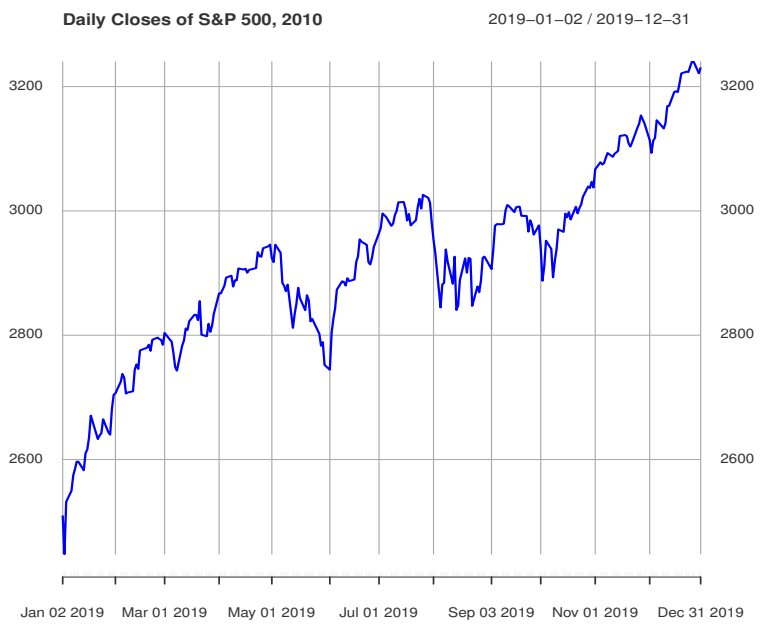


Figure 1.50. Time Series Plot of Daily Closes of S&P 500 as `xts` Object (Compare Figure 1.49)

Suppose we have a sample of size 50, and we use R to do the computations for a statistical hypothesis regarding the mean of an assumed normally-distributed population. Suppose we obtain the computed t value in the R object `tcomp`.
Write the R expression to determine the p-value of `tcomp` for a two-sided test.
Write the R expression to determine the p-value of `tcomp` for a one-sided test in which the null hypothesis is that the population mean is less than or equal to a specified value.

1.4.3. Shiny apps.
Write a Shiny app to determine the two-sided p-value of a computed t value. The user supplies to the app the computed t value and the degrees of freedom.

1.4.4. Black-Scholes formulas.

a) Write an R function, `BlackScholes`, to compute either value of either a call or a put with strike K and time to expiry T for a stock whose price is S and whose return volatility is σ when the risk-free interest rate is r. The defining function is

Figure 1.51. Candlestick Plot of Daily Prices and Volume Traded for Tesla for the Second Half of 2019

```
BlackScholes <- function(type, S, K, r, T, sd) {
```

b) Suppose that the volatility of a stock is 0.33. On a day when the market price of the stock is \$63.49, what is the fair value of a \$65 call 37 days out, if the risk-free interest is 0.02?

c) Use the Black-Scholes formulas to determine the *implied volatility* of the underlying based on the following observed data. (These data were observed on June 10, 2020, for the July 2020 options on the stock of Intel Corporation (INTC). (The expiry for July 2020 options was July 17.)

i. The 65 call 37 days out has a market price of \$2.00. The price of the stock is \$63.49 and the risk-free interest is 0.02. (These values correspond to those in Exercise 1.4.4b.) What is the implied volatility of the underlying?

ii. The 60 call 37 days out has a market price of \$4.90. (The price and the interest is the same as in Exercise 1.4.4(c)i.) What is the implied volatility of the underlying?

iii. The 70 call 37 days out has a market price of \$0.59. (The price and the interest is the same as in the previous exercise.) What is the implied volatility of the underlying?

d)
   i. The 65 call 8 days out has a market price of \$0.78. (The price and the interest is the same as in Exercise 1.4.4c.) What is the implied volatility of the underlying?
   ii. The 65 call 68 days out has a market price of \$3.10. (The price and the interest is the same as in the previous exercise.) What is the implied volatility of the underlying?

e) What do you observe about the implied volatilities at different strikes and at different expiries in Exercise 1.4.4c and 1.4.4d?

1.4.5.
a) Generate a sample of 100 standard normal random numbers. Plot a histogram of the sample.
b) Now generate a sample of 100 normal random numbers with mean 100 and variance 100. Make a scatterplot of the two samples.

1.4.6. Install and load the `pracma` package (if not already done).
a) Primes. Use the appropriate functions form `pracma()`.
   i. Determine the prime factorization of 1,000,000.
   ii. Determine the prime factorization of 999,999.
   iii. Is 999957 a prime?
   iv. Is 999959 a prime?
   v. Is 999961 a prime? (twin prime?)
   vi. Determine all primes up to 1,000,000. Just print the first 6 and the last 6.
   vii. Determine 100! using `fact{pracma}()`. This is a very large number.
   Now determine 100! using `factorial{base}()`. Compare the two computed values to 17 significant digits. The value computed by `fact()` is more accurate.

b) Matlab has many utility functions that are not implemented in the basic R system. The `bits()` function is an example. Use `bits()` to determine the binary representation of π to 56 bits (54 bits after the binary point).

1.5 Models

Data analysis involves the study of a *data-generating process*. A data-generating process is any activity or entity that yields observable data. Trading of stocks, bonds, and futures are the most prominent data-generating processes in finance. A *statistical model* is a description of a data-generating process that focuses on the distribution of the observable data and relationships among the observable variables.

1.5.1 Statistical Models

Two important classes of statistical models are *distributional models* and *models of relationships*.

Distributional Models

Distributional models describe the relative frequencies of occurrence of values of a variable. Standard forms of such a model are a *probability density function* or *PDF* and a *probability function*.

The *normal distribution*, also called the Gaussian distribution, is a common distributional model. It depends on two quantities, the mean μ and the variance σ. The PDF of the normal distribution is

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \mathrm{e}^{-(x-\mu)^2/2\sigma^2}. \tag{1.22}$$

This density has the familiar bell-shaped graph.

The normal PDF serves as a model for the frequency distribution of many familiar phenomena. To denote the distribution of a random variable X as normal with mean μ and variance σ, we may write

$$X \sim \mathrm{N}(\mu, \sigma^2). \tag{1.23}$$

Other distributions are used as models of other observational data, and R provides a set of functions for various computations on these distributions, including their PDFs. The R functions for working with various probability distributions are discussed beginning on page 104.

Relational Models

Another general type of statistical model in one that describes the relationship among variables. There are various forms of relational models. In a common form of this model, all variables are treated more-or-less equally. The objective is to determine some lower-dimensional structure, such as a hyperplane, that expresses the relationships among the variables. Many techniques in multivariate analysis, such as principal component analysis, are bases on models in which all variables are treated symmetrically.

Another common form of a relational model is asymmetric; some variables are considered to be dependent on other variables. In this form of the model, we consider one variable to be the *response* and the other variables to be *"independent" variables*, or *factors*, or *regressors*. (I often use quotation marks in this context, because the word "independent" has another common meaning in statistics.)

Letting y represent the response variable, and $x_1, \ldots, x_m$ represent the other variables, we write a model of the relationship as

$$y \approx f(x_1, \ldots, x_m), \tag{1.24}$$

where f is some function that expresses this relationship. It is not assumed that the xs *determine* y, certainly not determine y exactly.

We often write this relationship in the form

$$y = f(x_1, \ldots, x_m) + \epsilon, \tag{1.25}$$

where ϵ represents some random *error*. ("Error" is not meant to represent a mistake; it just represents an additive deviation of any particular response from the model response.)

A common instance of the model (1.27) is the *linear regression model*

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_m x_m + \epsilon, \tag{1.26}$$

in which $\beta_0, \beta_1, \ldots, \beta_m$ are assumed to be unobservable constants.

Time Series Models

In another form of a relational model, a time value is associated with each observation, and the time variable is the most important "independent" variable. In a time series model, the response variable is a variable at a particular point in time, say x_t. A general time series model is of the same general form as equation (1.27), but the variables independent variables include previous values of the same response variable. The general form is

$$x_t = f(t,\ x_{t_1}, \ldots, x_{t_r},\ y_t, y_{t_1}, \ldots, y_{t_s}) + \epsilon, \tag{1.27}$$

where $t_i < t$ (that is, t_i represents previous times), and $y_t, y_{t_1}, \ldots, y_{t_s}$ represent covariates measured at the same time or previously. Many time series models do not include any covariates.

Estimation and Fitting Models

Use of a statistical model in data analysis generally involves estimation of various components of the model using observed data. For example, if we assume a model of the form in expression (1.23), we may use the data to estimate the mean and variance, perhaps using the sample mean and variance.

There are many specific methods that can be used to estimate parameters or other components of models, such as maximum likelihood, least squares, and the method of moments. Much of the theory of statistics is directed toward development of good methods of statistical estimation, and the evaluation of properties of statistical estimators. We will not consider the details of these

methods in this book, but we may mention a specific method from time to time, such as maximum likelihood methods in time series models and least squares in regression models.

R provides many functions for estimation and fitting of models of various types. We will consider R functions for fitting time series models in Chapter 5 and various linear models in Chapter 7. In Chapter 6, we will consider some R functions for working with Bayesian models.

Prediction with Models

In a relational model, estimation of the components of the model by fitting it to observed data results in *predicted values* of the response.

We often denote estimated or predicted values with a caret or hat:

$$\widehat{y} = \widehat{\beta}_0 + \widehat{\beta}_1 x_1 + \cdots + \widehat{\beta}_m x_m. \tag{1.28}$$

More generally, we may represent a fitted model as

$$\widehat{y} = \widehat{f}(x_1, \ldots, x_m). \tag{1.29}$$

1.5.2 Defining a Model in Computer Software

Specifying a mathematical expression to be evaluated or defining a character string is relatively straightforward in most computer languages. To evaluate $x + e^y$, for example, we define variables `x` and `y` and write `x+exp(y)`. To form "ab" from "a" and "b", for example, if we have character variables `ca` set to `"a"` and `cb` set to `"b"`, we write `paste(ca,cb,sep="")`.

A regression model such as

$$y_i = \beta_0 + \beta_1 x_{1i} + \cdots + \beta_m x_{mi} + \epsilon_i,$$

along with standard distributional assumptions about ϵ may be relatively easy to specify in computer code. Variations and generalizations of the model, however, such as AOV models with covariates, generalized linear models, and nonlinear models, may be quite complicated. Standard forms of time series models, such as ARIMA models, can be represented fairly simply. We will describe those methods in Chapter 5. Bayesian models present additional problems for describing the model in computer code. We will consider some methods of specifying Bayesian models in Chapter 6.

Relational Models in R

An important aspect of any statistical software system is how the user specifies relational models in the system.

Wilkinson and Rogers (1973) described a scheme for representing relational models in an early statistical software package called GENSTAT. That scheme

has been expanded, and a version of the Wilkinson-Rogers notation is used to specify models in other computer software systems, including Matlab and R. We will consider some simple statistical relational models, and the way they are specified in R.

Statistical models of relationships are specified in R by a *formula*, which is a an R object of class `formula`. A formula class object consists of three parts,

$$\texttt{response} \quad \texttt{~} \quad \texttt{terms}, \tag{1.30}$$

where "response" is a numeric response vector and "terms" is a series of terms specifying the predictors and their form in the model. The symbol `~` indicates "equals" "plus an additive error term", with no specification of the nature of that error term.

The form of the model is indicated by operators such as "+", "*", and ":". The model is assumed to contain an additive constant, unless a "no-intercept" model is specified by "0+" or "-1". Most mathematical functions and operators can be included in the formula. The exponentiation operator has a special meaning. It refers to an interaction, unless it is escaped using the `I()` function (see the example). Table 1.9 shows some examples, using the standard statistical notation, and with the obvious interpretation of the R objects as statistical variables. Notice, in particular, that `(x1+x2+x3)^2` yields the interactions (cross-products) but not the squared terms. The operators "*", ":", and "^" are used primarily in classification (AOV) models.

Table 1.9. Some Example Model Formulas in R

`y ~ x1+x2`	$y_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \epsilon_i$
`y ~ 0+x1+x2`	$y_i = \beta_1 x_{1i} + \beta_2 x_{2i} + \epsilon_i$
`y ~ x1+x2-1`	$y_i = \beta_1 x_{1i} + \beta_2 x_{2i} + \epsilon_i$
`y ~ x1+x2+x1:x2`	$y_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \beta_3 x_{1i} x_{2i} + \epsilon_i$
`y ~ x1*x2`	$y_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \beta_3 x_{1i} x_{2i} + \epsilon_i$
`y ~ (x1+x2+x3)^2`	$y_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \beta_3 x_{3i} + \beta_4 x_{1i} x_{2i} + \beta_5 x_{1i} x_{3i} + \beta_6 x_{2i} x_{3i} + \epsilon_i$
`y ~ x1+I(x1^2)+log(x2)`	$y_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{1i}^2 + \beta_3 \log(x_{2i}) + \epsilon_i$

A `formula` object can be constructed by the `formula` function, as shown in Figure 1.52.

Exercises: Models

1.5.1. Write the R formula that corresponds to the polynomial regression model,

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \beta_3 x_i^3 + \epsilon_i.$$

```
> model1<-formula("y~x1+x1^2+log(x2)")
> model1
y ~ x1 + x1^2 + log(x2)
> class(model1)
[1] "formula"
> reg <- lm(model1)
...
```

Figure 1.52. A Formula in R

1.5.2. Write the R formula that corresponds to the full factorial model with no intercept,

$$y_i = \beta_1 x_{1i} + \beta_2 x_{2i} + \beta_3 x_{1i} x_{2i} + \beta_4 x_{1i}^2 + \beta_5 x_{2i}^2 + \epsilon_i.$$

1.6 Inputting and Wrangling Data in R

There are many different ways to input data into R. Data can be read from a file or directly from the computer console. We must tell the system three things: where the data are; how it is stored; and what R object to put it into.

We will consider some of the more common ways of inputting data here, but caution the reader that there are many details we will not go into.

Two primary considerations in bringing data into an R object are how the items in the input file or typed onto the console are to be interpreted, that is, what type of R object is to be created, and how the items are to be separated one from another. Another consideration is the class of the R object to be created to contain the data.

One of the simplest functions for inputting data is `scan()`, which produces an atomic vector if all elements are of the same type or else produces a list. If no file to be read is specified, then R will read data from the console until a line feed is entered with no other characters on the line. The R function `readline()` is also useful for inputting data directly from the console.

Sources of Data

Much of the financial data of interest to a data scientist or even to an ordinary participant in the market is freely available at a number of sites on the internet. There are other sources available for various fees, often substantial. Some major commercial database are Bloomberg, Datastream, FactSet, CRSP, and Global Financial Data. They differ in coverage in various ways: in

the instruments included, in geographical coverage, and in time periods covered. We will consider some of the open repositories and methods of accessing data from them in Section 1.6.2.

Other types of data feeds stream all transactions in a given market. These transactions occur at the rate of millions per minute. Such "high frequency" data exhibits some different statistical properties from daily data. We will not consider high frequency data in this book.

Data Quality and Derived Data

Except for occasional missing values, raw data such as closing prices, daily highs and lows, trading volume, and so on, are generally accurate and are consistent from one source to another. Simple derived data such as PE ratios are also generally consistent from one source to another, except for clear distinctions in definitions, such as in the case of PE, whether it is trailing or future (based on projected earnings).

Derived data that depends on more complicated models, such as beta, may differ because of different definitions of the quantities, and in some cases may differ further because they depend on different methods of fitting the models.

1.6.1 Inputting Data into R from External Files

For data stored in a tabular form in an external file, the R program `read.table()`, which allows various kinds of metadata (headers, field separators, and so on), produces an R data frame from the data in the external file. Because `read.table()` produces a data frame, columns in the table can be of different classes, and character data by default will be of class `factor` by default. The class of each column can be specified by the `colClasses` argument. If the classes are not specified, `read.table()` attempts to determine the class from the elements in the first line of data. Numbers stored in common formats and character strings beginning with a letter are usually interpreted correctly, but it is generally a good idea to specify the classes explicitly using the `colClasses` argument.

The complementary function `write.table()` can be used to create an external file and to store data in an R matrix or data frame in the external file.

Comma Separated (CSV) Files and Spreadsheets

A simple but useful structure for storing data is called "comma separated value" (or just "comma separated") or "CSV". The storage is of plain text, and commas are used to separate the fields. (In locales where commas are used as decimal points, the fields are usually separated by semicolons, but the structure is also called "CSV".) Blanks between fields are generally ignored.

The basic CSV structure is the same as that of a spreadsheet, so spreadsheet data are often stored as CSV files because they are portable from one system to another. Most spreadsheets, of course, contain formatting characteristics and other relationships such as formulas that cannot be saved as plain text.

Although the R function `read.table()` allows the user to specify various things as field separators, the function `read.csv()` assumes that the separators are commas; hence, it may be more useful for inputting data from a CSV file. The `read.csv()` function behaves similarly to the `read.table()` function in other respects, and just as the `read.table()` function, `read.csv()` produces an R data frame.

Consider, for example, a text file named `datRF10110.csv` in the working directory, as shown below:

```
Location, Date, Number of Cases
Alabama, 2020-1-20, 10
Alabama, 2020-2-15, 30
Alaska, 2020-1-20, 5
Alaska, 2020-2-15, 18
```

This can be read into an R data frame as shown. Notice that `read.csv()` assumes by default that the first line is a header, and note how blanks in the header are handled. Note further that the character variable is interpreted to be of factor class.

```
> dat <- read.csv("datRF10110.csv", colClasses=c("character","Date","numeric"))
> dat
  Location       Date Number.of.Cases
1  Alabama 2020-01-20              10
2  Alabama 2020-02-15              30
3   Alaska 2020-01-20               5
4   Alaska 2020-02-15              18
> split(dat, dat$Location)
$Alabama
  Location       Date Number.of.Cases
1  Alabama 2020-01-20              10
2  Alabama 2020-02-15              30

$Alaska
  Location       Date Number.of.Cases
3   Alaska 2020-01-20               5
4   Alaska 2020-02-15              18
```

Although most spreadsheet programs provide facilities for converting date formats, for example from "mm/dd/yyyy" to a POSIX format, often a CSV file contains dates in an non-POSIX format. In this case, a simple expediency is to read the dates as characters and then convert them to a POSIX format using the `as.Date()` function.

Suppose, for example that the data above are stored in a CSV file named `datRF10120.csv`, and the dates are as shown in plain text.

```
Location, Date, Number of Cases
Alabama, 1/20/20, 10
Alabama, 2/15/20, 30
Alaska, 1/20/20, 5
Alaska, 2/15/20, 18
```

We build the same data frame as above using the following R statements.

```
> dat<-read.csv("datRF10120.csv",
                colClasses=c("character","character","numeric"))
> dat$Date <- as.Date(dat$Date, "%m/%d/%y")
> dat
  Location       Date Number.of.Cases
1  Alabama 2020-01-20              10
2  Alabama 2020-02-15              30
3   Alaska 2020-01-20               5
4   Alaska 2020-02-15              18
```

Note that if a CSV file is stored by a spreadsheet program, the exact format of the file may depend on the spreadsheet program and display settings within the program. Also, when a CSV file is opened by a spreadsheet program, the spreadsheet program may make certain assumptions about the formatting. If the spreadsheet program subsequently saves the file as a CSV file, it may not be the same as the original file. For example, if Microsoft Excel reads in the CSV file `datRF10110.csv` above, it will interpret the second column as dates. If Excel than saves it as a CSV file, the actual text of the file depends on the formatting choices made in the Excel program. If, for example, the format for dates corresponds to the standard American format, the CSV file stored will be the same as the text of `datRF10120.csv` above (along with some non-ASCII characters corresponding to a heading indicator and line breaks, which may differ from the line break characters in the `datRF10110.csv` file).

The complementary function `write.csv()` can be used to create an external CSV file and to store data in an R matrix or data frame in the CSV format in the file.

CSV files provide the simplest means for exchanging data between R and a spreadsheet program, such as Microsoft Excel. The spreadsheet program can input data from a CSV file and can save many types of data in its native format as CSV files.

1.6.2 Obtaining Financial Data Directly from the Internet

There are various sources of financial data on the internet, and data can be input to R in a number of ways.

The data in many internet repositories are stored in CSV files. These data can often be read directly into R.

Data Repositories; URLs, HTTP(S), and XML

Sites on the internet are addressed in a standard way, and data at the sites are accessed through specific protocols. A *Uniform Resource Locator* or *URL* is used to specify the access scheme and the address of the data file to be accessed. The most common scheme to access internet data is the *Hypertext Transfer Protocol* or *HTTP*. An encrypted version of the protocol is called HTTPS, and many sites use the encrypted protocol. An example of a URL that we use often is

```
https://finance.yahoo.com/
```

HTTP interacts with data stored in a *Hypertext Markup Language* or *HTML* format, which is a standard ASCII (or text) format. HTML determines the way the data are presented, or how it appears in a web browser. For example, the text string

$$< \texttt{b} > \texttt{First} < /\texttt{b} >$$

causes a web browser to display the text in boldface: **First**. (Of course, exactly how the text would appear depends on other environmental settings of the display device and scheme.)

XML, or the *eXtensible Markup Language* is used to specify the organization and meaning of data at a web site.

A web browser provides a simple method for accessing data on the internet. ("Data" includes text and pictures as well as what we usually think of as data: stock prices, interest rates, and so on.) Data accessed in a web browser can often be downloaded into a file that can then be used by a program like R, but this is often inconvenient. There are various ways that a program like R can read data at a website. In R, many of the functions for doing this depend on the `RCurl` package. `RCurl` provides the capability of reading the XML or HTML directly, and sometimes this is necessary in order to determine the parameters to use in a program to access data. We will not discuss this lower-level package, but instead refer the interested reader to other texts, such as Nolan and Temple Lang (2014).

Yahoo Finance

FRED

R Software for Reading Data from the Internet

As we have seen, there are various sources of financial data on the internet and data can be input to R in a number of ways.

The data in many internet repositories are stored in CSV files. These data can often be read directly into R.

The quantmod Package

One of the best ways of obtaining financial data from the internet is by use of the getSymbols() in the quantmod package written by Jeffrey A. Ryan.

The getSymbols() in the quantmod package is user-friendly, and provides access to a variety of financial data at various repositories, which is specified in the src. The default repository is Yahoo Finance.

Arguments in getSymbols() may be specific to the data source and/or to the type of data. For example, the from and to arguments in getSymbols() are not implemented when the source is FRED. Whenever specific time periods are not selected in getSymbols() data just for those periods can be put into an R object using the subsetting mechanism for xts objects.

There are also other functions in quantmod for retrieval of specific types of financial data, such as getDividends(), getFX(), getMetals(), and getOptionChain().

For a specified stock, the getOptionChain() function returns the option chain, which, for each expiry, is a list with two components, one for calls and and one for puts. For each strike price, each component consists of last, bid, and ask prices, volume (of previous trading day), and open interest for the option.

The quantmod functions are also used in other R packages; for example, the tq_get() function in the tidyquant package and the get.hist.quote() function in the tseries package use the data acquisition functions in quantmod.

Other useful functions to obtain data are the Quandl() function in the Quandl package and the getYieldCurve() function in the ustyc package. The data source in Quandl() is specified as part of the Quandl code. The data source in getYieldCurve() is the US Treasury Department website.

Another useful package for processing financial data is TTR by Joshua Ulrich. This package includes several functions for smoothing financial data.

There are also a number of R packages and other software to process HTTP requests and data acquisition from the internet. Some of these are listed and briefly described, with links, under the "WebTechnologies" section of the task views webpage:

```
https://https://cran.r-project.org/web/views/
```

One of the best ways of obtaining financial data from the internet is by use of the getSymbols() function in the quantmod package written by Jeffrey A. Ryan.

The getSymbols() function is user-friendly, and provides access to a variety of financial data at various repositories, which is specified in the src. The default repository is Yahoo Finance. The quantmod() functions produce xts objects (see Section 1.3.6).

Arguments in getSymbols() may be specific to the data source and/or to the type of data. For example, the from and to arguments in getSymbols() are not implemented when the source is FRED. Whenever specific time peri-

ods are not selected in `getSymbols()` data just for those periods can be put into an R object using the subsetting mechanism for `xts` objects.

The log returns for the S&P 500 Index for 2019 that were shown in the histogram of Figure 1.43 and the time series plot of Figure 1.48 were computed from closing data obtained from the website for Yahoo Finance using `getSymbols()`. The `getSymbols()` function normally produces an environment with many useful properties, and in later discussions I will make use of the `quantmod` environment. Often, however, I want to keep all objects simple so that I know what to expect in using standard R functions. The following R code does not use the `quantmod` environment and it treats the data as in the regular `numeric` class. By using `head()`, we see that the closing prices are in the 4^{th} column.

```
> library(quantmod)
> z <- getSymbols("^GSPC", env=NULL, from="2019-1-1",
+                 to="2020-1-1", periodicity="daily")
> head(z, n=3)
           GSPC.Open GSPC.High GSPC.Low GSPC.Close GSPC.Volume GSPC.Adjusted
2019-01-02   2476.96   2519.49  2467.47    2510.03  3733160000       2510.03
2019-01-03   2491.92   2493.14  2443.96    2447.89  3822860000       2447.89
2019-01-04   2474.33   2538.07  2474.33    2531.94  4213410000       2531.94
> plot.ts(as.numeric(z[,4]),
+         main="Daily Closes of the S&P 500 Index, 2019",
+         xlab="Time (Days)", ylab="Prices", col="blue")
```

***Figure 1.49

For a specified stock, there are also `quantmod` functions to obtain dividend data, `getDividends()`, and and to obtain prices of listed put and call options on the stock, `getOptionChain()`. The `getOptionChain()` function returns the option chain, which, for each expiry, is a list with two components, one for calls and and one for puts. For each strike price, each component consists of last, bid, and ask prices, volume (of previous trading day), and open interest for the option.

There are also other functions in `quantmod` for retrieval of other types of financial data, such as `getFX()` for foreign exchange rates and `getMetals()` for metal commodities.

The `quantmod` functions are also used in other R packages; for example, the `tq_get()` function in the `tidyquant` package and the `get.hist.quote()` function in the `tseries` package use the data acquisition functions in `quantmod`.

Other useful functions to obtain data are the `Quandl()` function in the `Quandl` package and the `getYieldCurve()` function in the `ustyc` package. The data source in `Quandl` is specified as part of the Quandl code. The data source in `getYieldCurve()` is the US Treasury Department website.

Another useful package for processing financial data is `TTR` by Joshua Ulrich. This package includes several functions for smoothing financial data.

There are also a number of R packages and other software to process HTTP requests and data acquisition from the internet. Some of these are listed and briefly described, with links, under the "WebTechnologies" section of the task views webpage:

`https://https://cran.r-project.org/web/views/`

1.6.3 Data Cleansing

Often the first step in getting data ready for analysis is just to get the data into a proper format. The format required may depend on the model that guides the analysis, or it may depend on requirements of the software. Some R functions, for example, may require that the data be stored in a data frame. This initial step may also involve acquisition and assemblage of data from disparate sources. These data preparation activities are often called "data wrangling" or "data munging".

Individual items in a dataset in a repository such as Yahoo Finance or FRED may be missing or invalid. Also, a program such as `getSymbols()` or `Quandl()` that obtains the data may not work properly. This kind of problem may be due to a change in the structure of the data repository. Such a problem occurred in April 2017 when, due to changes made at Yahoo Finance, `getSymbols()` and other programs to access data at Yahoo Finance quit working properly. (The initial problems have been fixed, but these programs still sometimes return incorrect data; see the notes to this appendix.)

Unexpected things should always be expected. Consider, for example, the weekly data series from FRED. Suppose we want to regress weekly data from Moody's corporate bond rates on weekly effective fed funds rates (or the weekly differences. If we want the weekly data for these two series for the period from 2015 through 2017, an obvious way to get them is to use the following code.

```
getSymbols("WBAA", src = "FRED")
WBAA1 <- WBAA["20150101/20171231"]
getSymbols("FF", src = "FRED")
FF1 <- FF["20150101/20171231"]
```

Surprisingly, however, WBAA1 has 157 rows and FF1 has 156. This is because the `WBAA` data are for Fridays and the `FF` data are for Wednesdays, and the number of Fridays in the period from 2015 through 2017 is different from the number of Wednesdays in that period.

Head and Tail

When a dataset is first brought into R, before performing any analyses, it is a good idea to look at the first few and last few values. This can be done using

the head() and tail() functions. The number of rows can be specified by the n argument.

```
head(x, n=2)
tail(x, n=2)
```

Whenever there are known relations that should exist among variables or among observations, it is a good idea to perform some simple consistency checks to ensure that those relations hold in the dataset. This can often be done visually using head().

```
sum(is.na(x))
```

The function complete.cases() can be used to determine which cases in a data.frame() or which rows in a matrix contain no missing values.

The function na.omit() produces a copy of an R object with the missing values omitted. The logical parameter na.rm is available in many R functions to specify that missing values are to be removed before performing any computations, if possible.

```
> x <- c(1, 2, NA, 3)
> mean(x)
[1] NA
> mean(x, na.rm=TRUE)
[1] 2
> mean(na.omit(x))
[1] 2
```

In var() and cov(), if na.rm is true, then any observation (or "case") with a missing value is omitted from the computations. These functions, however, provide more options through the use keyword. If use="pairwise.complete.obs" is specified, for example, the computation of the covariance of two variables will use all complete pairs of the two.

The zoo package (and hence quantmod) has a function na.approx() that will replace missing values with a linearly interpolated value and a function na.spline() that will use a cubic spline to approximate the missing value.

Missing Data

*** refer to previous discussion

The Yahoo Finance data generally go back to January 1962 or to the date of the initial public offering. If the from date specified in getSymbols() is

a long time prior to the IPO, some meaningless numerical values may be supplied along with NAs in some fields prior to the date of the IPO. When using `getSymbols()` for a period back to the IPO, the `from` date must not be "too long" before the IPO date. Most of the fields before the IPO are `NA`, but to find where the valid data begin, we cannot just search for the first `NA` (processing the data in reverse chronological order). As mentioned elsewhere, the data from Yahoo Finance or the data returned by `getSymbols()` may occasionally contain `NA`s, for which I have no explanation.

As emphasized earlier, when processing financial data, we take the attitude of "big data" processing; we do not want to look up the IPO date and hard-code it into the `from` date in `getSymbols()`. If a `from` date prior to the IPO is specified, the output can be inspected automatically by the computer and the `from` date can be adjusted.

Merging Datasets

*** refer back to previous discussion

Problems sometimes arise when merging data sets. Sometimes this is because one of the datasets is messy. Other times it is because some rows are missing in one or the other of the datasets. The `join` keyword in `merge()` can ensure that only the proper rows are matched with each other. Sometimes, the best cleanup is just to use the `na.omit()` function.

Another problem in merging datasets by the dates is that the actual dates associated with different comparable time series may never match. For example, the weekly series of FRED data may be computed on different days of the week. The weekly fed funds rate is as of Wednesday and the weekly Moody's bond rates are as of Friday. Hence, if a data frame containing `FF` is merged with a data frame containing `AAA` over any given period, the resulting data frame would have twice as many observations as in those for `FF` and `AAA`, and each variable would have `NA`s in every other row.

Data cleansing is an important step in any financial analysis. If the data are garbage, the results of the analysis are garbage. In addition to the issues of misinterpreting the data format, unfortunately, there are many errors in the data repositories.

Exercises: Inputting and Wrangling Data in R

1.6.1. Write a CSV file named `Prices.csv` corresponding to the data frame `Prices` in Exercise 1.3.6.

1.6.2. Read the CSV file `Prices.csv` from Exercise 1.6.1 into an R data frame called `Pricesdf`.
Print `Pricesdf`. (It should be exactly the same as the data frame `Prices`.)

1.6.3. Read from the internet the daily closing prices of Intel Corporation (INTC) for the period January 1, 2017, through September 30, 2017.
   a) Plot the closing prices as a time series.
   b) Compute the simple daily returns.
      Plot the simple daily returns as a time series, and make a histogram of the simple daily returns.
   c) Compute the daily log returns.
      Plot the daily log returns as a time series, and make a histogram of the daily log returns.

1.6.4. Obtain the weekly rates on the 3-Month US T-Bill (WTB3MS) and the weekly on the 10-Year US T-Treasuries (WGS10YR) from FRED.
Plot these rates from 2000-01-01 through 2020-04-30 as time series on the same set of axes.
Was "the yield curve" inverted during this period?
Were the rates ever negative?

1.6.5. getOptionChain
Exercises 1.4.4 and 2.1.1 Black-Scholes

1.6.6. Shiny apps.
Write a Shiny app to compute the beta of a specified security with respect to a specified "market", where the historic statistics are computed at a specified frequency for a specified time period.
If the market is "M" and market returns at the specified frequency over the specified period is R_M, and the specified security is "i" with returns R_i, the beta for the security for the given frequency computed over the given time period is

$$\beta_i = \frac{\mathrm{Cov}(R_i, R_M)}{\mathrm{V}(R_M)}$$

Your app should accept the symbols for the security and the market (assuming the correct form), the frequency ("periodicity" in `quantmod` terminology), and the beginning and ending dates (in POSIX form).
Once the values are entered, your app should compute and display the security's beta.

2

Financial Data in R

While in Chapter 1 we discussed general capabilities of the R system and methods of acquiring financial data, in this chapter, we will focus on the data itself.

There are many types of financial and economic data. Some general types of data include measurements of economic activities, such as employment and production data; prices and changes in prices of commodities, equities, and real properties; and earnings accruing from business activities or from ownership of financial or real assets.

Most financial data are parts of time series, because they refer to characteristics measured or observed at particular points in time.

In this book, although we will occasionally discuss various types of financial data, we will focus primarily on prices of equities and on returns on equity assets. The examples will illustrate important characteristics of the data, but our main interest will be in the methods and the software to reveal the characteristics.

The sequence of daily closes of the S&P 500 Index for the year 2019, as shown in Figure 1.49 on page 120, is a time series. It would not be very informative to compute aggregated statistics such as the mean or standard deviation of the prices, or to make a histogram of the prices. Other types of financial data, however, can be analyzed and studied either as a time series in which the relationship of the data to time is important, or the data can be aggregated over a specific time period and the sample can be analyzed without any reference to time. For example, the daily S&P 500 returns can be viewed as a daily time series, as in Figure 1.48 on page 119, or they can be viewed as a static aggregation, as in the histogram in Figure 1.43.

2.1 Aggregated Data

For data on a single variable, if we ignore any aspect of differing times associated with the data, the properties of interest are the simple statistics that

correspond to characteristics of the distribution, such as the sample mean, the sample variance, the sample skewness, and the sample kurtosis. The overall shape of the frequency distribution may be of interest. and graphical displays are useful in assessing the shape of the distribution.

2.1.1 Frequency Distributions; Graphical Displays

The frequency distribution of data is similar to a probability function or a probability density function (PDF). A probability model expresses the probability of a random variable taking on a specific value or being within a specified region of its domain. A frequency distribution shows the relative frequencies of data within specified regions of the range of the data.

Binned Data; Histograms

A histogram, as in Figure 1.43 on page 111, provides a simple picture of the frequency distribution of a variable. It shows the range of the values of the variable and the general shape of the distribution.

Figure 1.43 is a histogram of the daily simple returns of the S&P 500 Index for the year 2019. Since these are simple returns, they can be interpreted directly as percentage changes.

From the histogram, we see that the returns were generally between -1% and 1%. They were more-or-less symmetrically distributed about 0. There were a few daily moves of between 1% and 2%, and even fewer greater than $\pm 2\%$.

The histogram in Figure 1.43 was produced by the R statement shown on page 114. The R function `hist` produces output that allows us to inspect the data more closely. The object produced by `hist` is a list with various components including `breaks`, with an obvious meaning, and `density`, which is the relative frequency of occurrence between each pair of breaks. We illustrate the use of these list components using some R functions that we mentioned in Chapter 1.

```
> SPhist <- hist(ret, freq=FALSE, main="Daily Simple Returns, 2019",
+                xlab="Daily Return", ylab="Relative Frequency", border="blue")
> SPhist$breaks
[1] -0.20 -0.15 -0.10 -0.05  0.00  0.05  0.10  0.15  0.20
> SPhist$density
[1] 0.079681 0.079681 0.876494 8.446215 9.880478 0.557769 0.000000 0.079681
> round(diff(SPhist$breaks)*SPhist$density, 3)
[1] 0.004 0.004 0.044 0.422 0.494 0.028 0.000 0.004
> sum(diff(SPhist$breaks)*SPhist$density)
[1] 1
```

From this, we see that the percentage between -1% and 0% was 42.2%, and between 0% and 1% was 49.4%.

The daily simple returns of the Index have been consistent with the returns shown in the histogram for 2019, and a financial analyst should be generally familiar with these properties of the S&P 500 Index. In some years, 2020, for example, the Index experienced some more extreme daily changes, as we will see below.

First, however, let us consider some variations on histograms. The general appearance of the histogram depends on the breakpoints. These can be controlled in the `hist` function by use of the `breaks` keyword argument. The breakpoints are not necessarily evenly spaced. If this keyword is not used in the function reference, as in the code on page 114, the function will makes judicious choices, as shown in the histogram. We show some histograms with other choices of breakpoints in Figure 2.1.

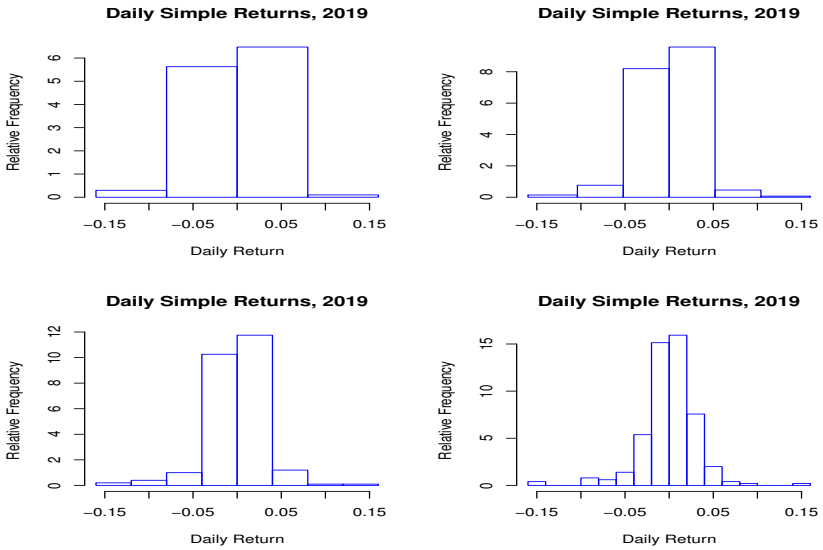


Figure 2.1. Histogram of Daily S&P 500 Simple Returns for 2019

The histogram in the top left in Figure 2.1, for example, was produced by the R statement below.

```
hist(ret, freq=FALSE, main="Daily Simple Returns, 2019",
     xlab="Daily Return", ylab="Relative Frequency", border="blue",
     breaks=c(-0.04,-0.03,-0.01,0.01,0.03,0.04)
```

Kernel Density Estimation

***kernels

2.1.2 Simple Statistics

**

Univariate Statistics

Multivariate (Bivariate) Statistics

If there are more than two variables, properties of interest include bivariate statistics such as covariances and correlations. We might also be interested in fitting relational models involving the variables.

Relational Statistics

PE
- beta
- Sharpe ratio
- Black-Scholes

Exercises: Aggregated Data

2.1.1. Black-Scholes write function
Exercise 1.6.5
Use getOptionChain

put in Solutions

```
BlackScholes <- function(S, K, r, T, sig, type)

  if(type=="C")
  d1 <- (log(S/K) + (r + sig^2/2)*T) / (sig*sqrt(T))
  d2 <- d1 - sig*sqrt(T)

  value <- S*pnorm(d1) - K*exp(-r*T)*pnorm(d2)
  return(value)

  if(type=="P")
  d1 <- (log(S/K) + (r + sig^2/2)*T) / (sig*sqrt(T))
  d2 <- d1 - sig*sqrt(T)

  value <-  (K*exp(-r*T)*pnorm(-d2) - S*pnorm(-d1))
  return(value)
```

2.2 Time Series

A *time series* is a sequence of events or a sequence of data in which the sequence is indexed by time. The order of occurrence of the events is a component of the time series. For some time series, only the order characterizes the sequence; for other time series, the actual time associated with each event or datum is relevant.

2.2.1 Spacing in a Time Series

For many time series, the actual length of time between events or the observed data is not important. Annual data, such as the closing value of the Dow Jones Industrial Index at the end of the year or the total volume of all trades on the New York Stock Exchange (NYSE) during the year, are essentially equally spaced from one year to the next, even though different years do not cove the same length of time, and the closing trading day of any given year can vary by one or two days. Some daily data, such as total rainfall at a given location, are equally-spaced, while other daily data, such as the total volume of all trades on the NYSE, are not equally spaced, because the daily data are trading-day data and are not recorded on weekends or holidays.

In many cases, even if the time series are not equally spaced, we can ignore the differences in time intervals. Considering the data to be equally spaced greatly simplifies analyses or computer processing. A computer object to handle equally-spaced data requires no field for the date; a simple integral index suffices. In other cases, however, the effects of weekends or other gaps in the sequence may be important, and this requires a more complicated computer object to store the data.

2.2.2 Types of Time Series

Another way of distinguishing types of time series depends on how the data are collected. There are three common types of time series based on how the observations are made, aggregated data, sampled data, and tick data.

Aggregated data are summary data made over an interval of time. For example, the daily trading volume of a given stock, fund, or group of stocks is an aggregated datum. The lengths of the period are obviously relevant metadata. Trading volume is routinely measured and reported for days, for weeks, and for months. Note the each of these periods may vary in actual length; a day may be shortened, as before a holiday; a week may have four days or five days, depending on the timing of a holiday; and a month may have various numbers of days, depending on the timing of weekends and holidays and on the month itself.

Sampled data are observations made at chosen points in time, for example, the closing price of a stock, which is generally defined as the price at which the stock was last traded prior to the close. Note that it could also be defined

in other ways, as, for example, the weighted average of the best bid and best ask.

Tick data are time series data recorded at the times of the events that generates them. For example tick-by-tick stock data are the prices and volumes of trades made at the time the trades occurred. In an active market, these data are "high frequency". The underlying statistical models that tick data tend to follow are different from data sampled at longer intervals.

For both aggregated data and sampled data, an important distinction is whether or not the observations are made at equally-spaced (or nearly-equally-spaced) intervals. Tick data are generally unequally spaced, because they arise as a result of actions of individual entities.

If the observations are equally-spaced, then if the date of the first observation is known, and the interval length or the frequency of the observations is known, then a simple index of positive integers is sufficient to identify the date of each observation, or to form subsets of the dataset corresponding to a specified time period.

2.2.3 R Software for Processing and Analyzing Time Series

For equally-spaced time series data, there is no need to have an explicit time-stamp for each observation. A sequential numerical index together with a starting time, a frequency, and some knowledge of a calendar mapping is sufficient to associate a time with each observation.

If the time series is not equally spaced, a column (variable) that contains the date may provide the necessary abilities to deal with the dates. Instead of a variables, the dates could be used as names of the rows (observations). Neither of these methods would allow the kinds of operations we might wish to perform on the time series data, such as subsetting by dates or merging two time series by date, as we discussed in Section 1.3.6.

Most of the common techniques for processing and analyzing time series assume that the data are equally spaced, and hence do not involve an explicit time stamp for each observation. Fundamental operations, such as lagging and differencing, have simple meanings only if the data are equally spaced. Autocorrelations have meaning and can be computed under an assumption that the data are equally spaced. Familiar linear time series models, such as AR, MA, ARMA, and ARIMA models apply only to data that are equally spaced.

The R `stats` package provides extensive capabilities for working with equally-spaced time series, including functions to compute the ACF and to fit various linear time series models. The `stats` package also provides a time series class, `ts`, which causes some R functions to produce results that are more appropriate for a time series (such as the generic `plot` producing a graph with connecting line segments). There are also special functions in the `stats` package for working with `ts` objects, such as the `diff` function, which produces

a time series of differences with a specified lag. In Chapter 5 we will discuss analysis of some of the the basic time series models.

There are a number of other R packages for working with and analyzing time series. Some of these packages allow more flexibility in date data and some provide special classes for time series objects, and they provide functions for various kinds of analyses of time series. Some of the available packages can be seen under the TimeSeries section at

`https://cran.r-project.org/web/views/`

Various packages implement irregular time series based on POSIXct time stamps. These are especially in financial applications, because weekends and holidays make most financial data unequally spaced.

The R package `timeDate` contains functions to provide financial date and time information, including information about weekends and holidays for various stock exchanges.

The package that I will use most often in this book is `xts`, written by Jeffrey A. Ryan, Joshua M. Ulrich, and Ross Bennett. This package is based on the `zoo` ("z- ordered observations") package. Manipulations of `xts` objects were discussed in Section 1.3.6.

Time Series Objects with Equal Spacing

Much of the data used in statistical analyses can be contained in a numeric vector or matrix, and financial data consisting primarily of numbers can be just stored in R atomic vectors or matrices. Data frames provide more metadata and through it allow for a wider range of operations, but data frames do not provide any special abilities for time series data.

There are two classes of R objects provided in base R that provide simple metadata for equally-spaced numeric time series, `ts` and `mts`.

Plotting `ts` Time Series Objects

```
plot(GDPQ, main="US Quarterly GDP", ylab="GDP (billions of USD)")
```

We notice several things about the plot that may need clarification. There are 8 data points, which represent values that are each aggregated data over a period of one quarter. The meaning of the line segments joining the observations is not clear. (The line segments may represent a smoothed fit of the instantaneous monthly rate of the GDP, but that is a rather arcane concept.) The abscissa labels seem to be off by one quarter. The first plotted point corresponds to the first quarter of 2018, which, of course is more appropriately

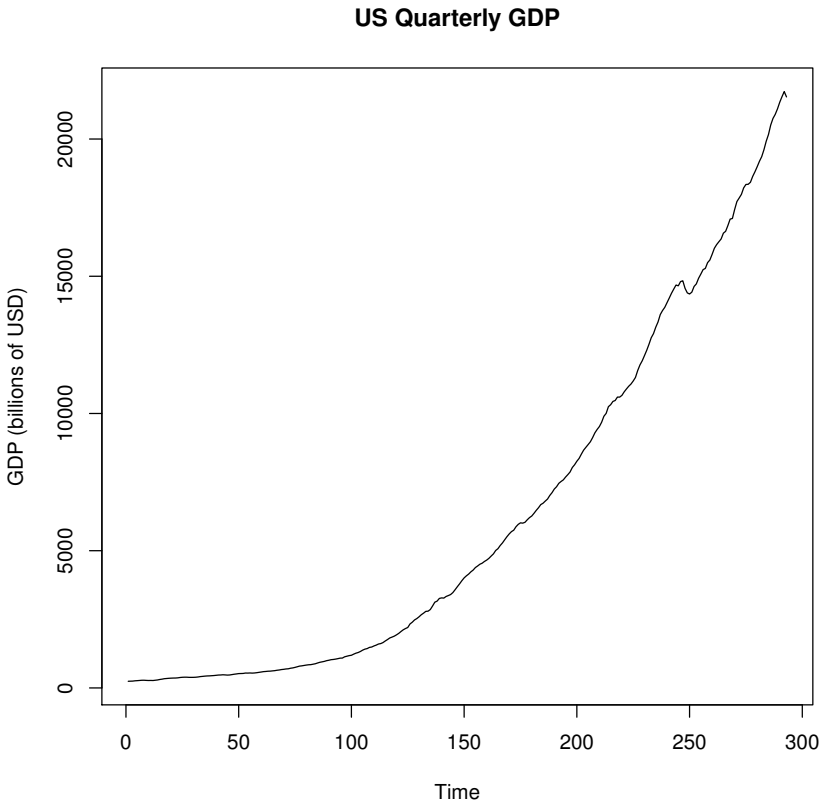


Figure 2.2. Time Series Plot of US Quarterly GDP

associated with the end of the quarter, rather than the beginning. Aside from those issues, the data correspond fairly closely to equally-spaced intervals.

Stock prices are time series that are widely analyzed. Figure 2.3 shows the daily closes of INTC for the month of February, 2020. Weekends are excluded as well as the US holiday of President's Day, so there were 19 trading days during that month.

The graph in Figure 2.3 would seem to suggest that the intervals between the data are equal, when, in fact, there were wide differences, as we see in a more accurate plot in Figure 1.50.

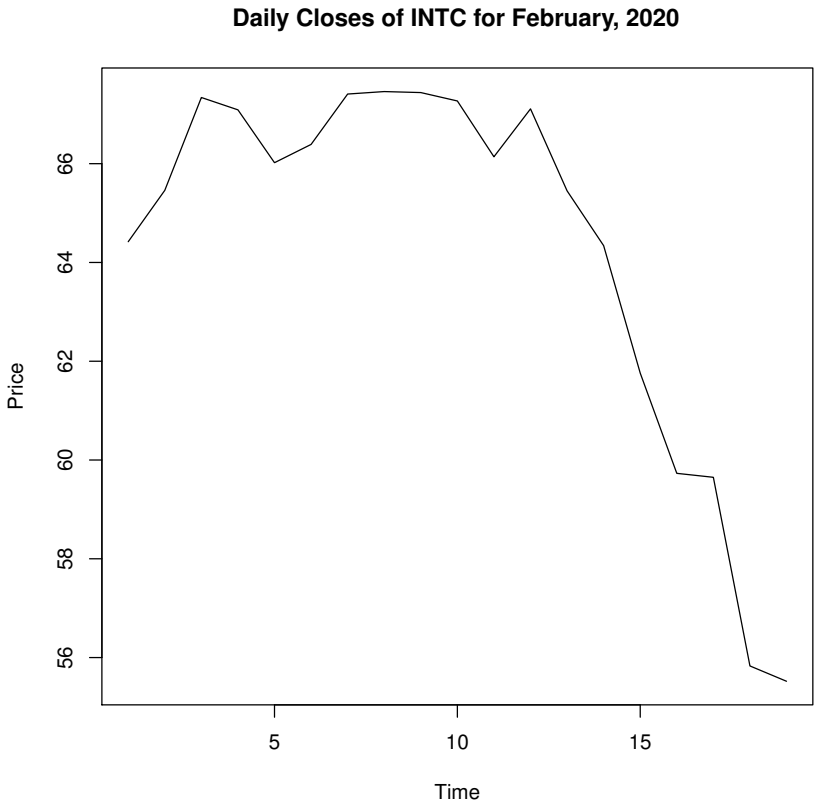


Figure 2.3. Time Series Plot of Daily Closes of INTC, Treated as Equally Spaced

3

Simulating Data in R

We have mentioned the useful collection of built-in functions involving several common probability distributions that compute the value of the density or probability function at a specified point, the value of the CDF at a specified point, or the quantile for a given probability. These functions have names that begin with "`d`", "`p`", or "`q`", and with another part that is a mnemonic referring to the name of the distribution. In addition to these three functions for a given distribution, there is another function that begins with "`r`" that indicates that a "random sample" of data from that distribution is to be simulated. The sample is not actually random, because it it generated according to fixed rules in the software. (Sometimes we use the term "pseudorandom".) The rules generate a rather haphazard sequence from a given starting point, which the user can specify using the R function `set.seed`.

The general starting point is to generate numbers that seem to be distributed uniformly and independently over the interval $(0, 1)$. There are several ways that pseudorandom uniform numbers can be generated, and R provides a function `RNGkind` that allows the user to choose the type of basic generator.

A slightly different approach is to put restrictions on the process so as to make the numbers more uniform, at the expense of being less "random". Such numbers are called "quasirandom". The `randtoolbox` package contains the functions `halton`, `sobol`, and `torus` that generate quasirandom uniform numbers. By default, these functions generate the same sequence each time. The package also has functions to perform statistical tests on samples of random numbers.

For a given probability distribution, the arguments to the "`r`" function are the same as in the other functions for that distribution, except instead of the point at which the function value is to be computed, the first argument is a positive integer or a vector.

For the Poisson family of distributions for which we defined the three functions that evaluate the density, probability, and quantile in equation (**??**), we also have the function

`rpois(n, lambda)`

that generates a vector of n values (where n is some positive integer) that will be similar in their statistical properties as the corresponding properties of a sample of the same size from a Poisson distribution with parameter `lambda`. The mean of a random variable with parameter λ is λ and the standard deviation is $\sqrt{\lambda}$. For example, we have the results shown in Figure 3.1.

```
> set.seed(12345)
> lambda <- 5
> n <- 1000
> x <- rpois(n, lambda=lambda)
> mean(x)
[1] 5.077
> sd(x)
[1] 2.17389
```

Figure 3.1. Simulating a Sample of Poisson Data in R

For the univariate normal distribution for which we defined the three functions that evaluate the density, probability, and quantile in equation (1.9), we also have the function

`rnorm(n, mean, sd)`,

that generates a vector of n (where n is some positive integer) values that will be similar in their statistical properties as the corresponding properties of a sample of the same size from a normal distribution with mean `m` and standard deviation `s`. For example, we have the results shown in Figure 3.2.

```
> set.seed(12345)
> m <- 100
> s <- 10
> n <- 1000
> x <- rnorm(n, mean=m, sd = s)
> mean(x)
[1] 100.462
> sd(x)
[1] 9.987476
```

Figure 3.2. Simulating a Sample of Normal Data in R

3.1 Generating "Random Numbers"

3.1.1 Methods

3.2 Simulating Data in R

3.2.1 Managing the Seed

3.3 Simulating Time Series

Brownian motion |rwe1071::rwiener
Brownian bridge |rwe1071::rbridge

4

Graphics in R

Murrell, Paul (2018)

R provides several functions to produce graphical displays of various types. Most of these functions have a very simple interface with default settings, but they also allow specification of several graphical characteristics, such as line color and thickness, labeling of the axes, size of characters, and so on. All of the graphics in this book were produced using R.

A graphical display has many elements. A well-designed graphics software system (R is such a system!) allows the user to control these elements, but does not impose an onerous burden of requiring specification of every little detail. Most of the graphics functions in R have simple interfaces with intelligent defaults for the various graphical components. The R function `par` allows the user to set many graphical parameters, such as margins, colors, line types, and so on. (Just type `?par` to see all of these options.)

I highly recommend use of the `ggplot2` package for graphics in R (see Wickham 2016). Some plotting functions in `ggplot2` are similar to those in the basic `graphics` package of R, but they make more visually appealing choices in the displays. Graphical displays in `ggplot2` are built by adding successive *layers* to an initial plot. Options can be set in `ggplot2` by use of the `opt` function. General graphics elements are determined by "themes", and there are a number of functions with names of the form `theme_xxx` to control the themes.

4.1 Types of Graphs

Two-dimensional graphs display the relationship between two variables. There are different kinds of graphs depending on the interpretation of one of the variables.

A display surface is essentially two-dimensional, whether it is a sheet of paper or a monitor. When there are more than two variables, we generally display them two at a time, possibly in a square array.

There are some situations, however, in which we wish to display a third dimension. A common instance of this occurs when we have just two variables, but we wish to represent another special variable, such as a frequency density. Likewise, with two variables, we may have a third variable, measuring their joint frequency density. Those graphs were produced by the standard generic `plot` applied to a `kde` object produced by the `kde` function in the `ks` package using the keyword argument `display` to specify `"persp"` or `"image"`.

When the object to be viewed is a surface, as in this case, a *contour plot* is useful. The R function `contour` produces contour plots. The R function `image` also produces special, colored contour plots. The function `contourplot` in the `lattice` package provides more functionality for contour plotting. Another R function useful for viewing a three-dimensional surface is `persp`, which shows a surface from an angle that the user specifies. As mentioned above, the standard generic `plot` function has been instrumented to produce contour and three-dimensional surface plots of objects of certain classes.

There are also some R packages that provide functions for producing three-dimensional scatterplots, for example, `scatterplot3d` in the `scatterplot3d` package.

- **scatterplot:** points in two-dimensions that represent the two variables;
  R: `plot`, `qplot`, `ggplot`, `points`
  - **matrix of scatterplots** (or other bivariate plots)
    R: `pairs`
  - **superimposed scatterplots of the columns of two matrices**
    R: `matplot`
  - **fan chart**
    R:`fanplot`
- **line plot:** continuous functions in two-dimensions
  R: `plot`, `qplot`, `ggplot`, `lines`, `curve`
- **bar plot:** bars whose heights represent values of a single variable
  R: `barplot`
- **grouped data:** one variable is a factor (an indicator of group membership)
  R: `hist`; `boxplot`; or use graphical parameters to distinguish groups
- **time series:** one variable represents time over an interval
  R: `plot`, `plot.ts`
- **frequency distribution:** one variable represents a frequency count or density
  - **histogram**; R: `hist`
  - **q-q plot**; R: `qqplot`
  - **boxplot**; R: `boxplot`
  - **density plot**; R: `density`, `plot.density`
  - **bivariate density plot** R: `kde` in `ke` `plot` with `kde` methods
- **special graphics for financial data** candlesticks, bars, bands, and so on.
  R: `plot`, `plot.xts`, `chartSeries`

*** adding graphic elements
Figure 1.50 page 121

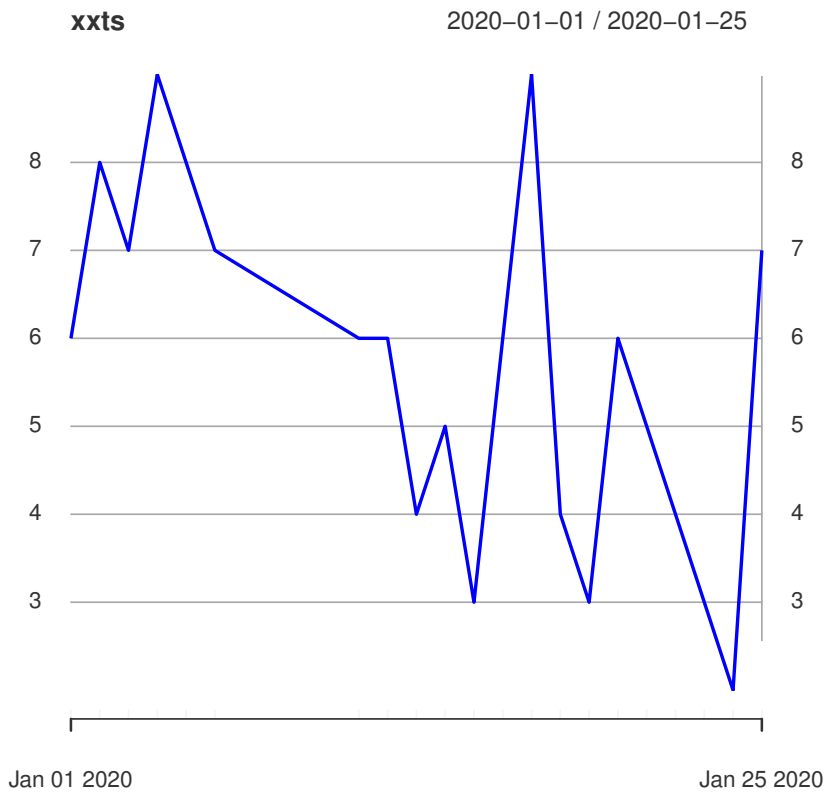


Figure 4.1. Simple Time Series Plot (Figure 1.50) with Added Lines

Plots of different types of objects may have slightly different appearances. For example, a scatterplot of a time series will have line segments connecting the points, whereas a scatterplot of a vector will not have those lines, unless the user expressly requests them.

4.2 Graphics in R

A graph in R may be completed with one function call. For example, the histogram in Figure 1.43 was produced by one simple call to `hist`.

Many graphs are composed of multiple plots or multiple graphic components such as titles or legends. These graphs are built up from calls to various R functions. Some R functions, such as `plot`, `barplot`, `qqplot`, and `hist`, are designed to initiate a plot. Other R functions, such as `lines`, `text`, `title`, and `legend`, are designed to add components to an existing plot. Some R functions, such as `curve`, can initiate a plot or add to a plot. (The keyword `add` determines this.) Each separate component is a layer on the graph.

One R function is called to produce the basic plot and then other functions are called to add other plots or components. For example, Figure **??** was produced by a call to `hist` to make the basic histogram followed by a call to `curve` to add the normal probability density function.

To produce the graph in Figure **??**, first of all, the data were scaled so that each series would begin with the same value. The a call to `plot` was to make the basic line graph using `type="l"`. In this call, the `ylim` parameter was set so that it would accommodate the largest value in any of the series. The labeling of the axes and the tick marks for the horizontal axis were set, either in the call to `plot` or immediately afterwards. Next, two calls to `lines` were made to plot the logs of the other two series. Finally `legend` was called to add the legend.

Color in Graphics

The argument `col` causes a specific color to be used to plot a line or a point. Some colors can be specified by name, for example, `col="red"` specifies some shade of red. Colors can also be specified using an additive RGB scheme. A color in this scheme is specified by six hexadecimal digits preceded by "`#`", in which the first two digits specify the intensity of red, from 0 (none) to 254 (full saturation), the second two digits specify the intensity of green, from 0 to 254, and the last two digits specify the intensity of blue. For example, `col="#00FF00"` represents green, `col="#FF00FF"` represents magenta, and `col="#780078"` represents a light magenta. The default color for a line and for points is black, `col="#000000"`, that is, no reflection. (White is `col="#FFFFFF"`, full reflection of all colors.)

4.3 Layouts of Graphs

Many figures, such as Figures **??**, **??**, and **??**, consist of rectangular arrays of multiple plots. These can be set up in R by use of the multiple figure parameter `mfrow` or `mfcol` in `par`. (The difference is the order in which the individual plots are put into the rectangular array; `mfrow` indicates that they should be produced row by row.) The side-by-side plots in Figure **??** were produced using `par(mfrow=c(1,2))`, and the four plots in Figure **??** were produced using `par(mfrow=c(2,2))`. The R function `layout` can also be used to control the arrangement of multiple figures.

The mar parameter in par can be used to control the margins around a plot. This is particularly useful in multiple figure graphs.

4.4 Graphics with xts Objects

In plots of time series, the labels of the tick marks of the time axis are often just the positive integers, corresponding to the indexes of the observations. The software that chooses these labels usually does so in a "pretty" fashion; the labels are chosen in a reasonable way so that the sequence of printed values follows a regular pattern. In many cases, of course, the time series object does not even contain the actual date information.

For financial data collected over an interval of several days or even of several years, a desirable alternative to using integers is to use labels that actually correspond to the calendar time. One of the major advantages of xts objects is that they do contain the date information, and the methods for processing these objects can actually use this information; that is, the date information in an xts object is more than just names of the rows, as in an ordinary R data frame.

The plot method for an xts object (plot.xts) accesses the dates and uses them to label tick marks on the time axis. It is much more difficult for software to choose the dates to print than it is to choose which integers to print. The plot.xts does a good job of choosing these dates. The time labels on the graphs in Figure 1.50 on page 121 is an example.

4.4.1 Building Plots with xts Objects

Simple plots; add line

4.4.2 Graphics with OHLC Objects

Many xts objects are OHLC datasets. The plotting methods for OHLC xts object provide an option for candlestick graphs.

A candlestick plot

Figure 4.2. Daily Open, High, Low, and Closing (OHLC) Prices and Volume of INTC January 1, 2017 through March 31, 2017; *Source:* Yahoo Finance

5

Time Series Analysis in R

`strucchange` structural changes

The mathematical model for a time series is a *stochastic process*, which is a sequence of random variables,

$$\ldots, X_{t-1}, X_t, X_{t+1}, \ldots, \tag{5.1}$$

often denoted in general as $\{X_t\}$.

The term "time series" also refers to a sequence of *data* corresponding to a sequence of times. Since a sequence of random variables is different from a sequence of data or observations, a careful treatment of time series would make precise distinctions between the model and the realizations. Certain terms, such as "expectation", "stationarity", and so, only apply to the model. Terms such as "mean" and "variance" can apply either to the model or to the observed data. They are related, of course, but they are different, and I will often use the terms "sample mean" and "sample variance" for clarity. (To emphasize the difference, we may note, for example, the the model variance can be infinite, but the sample variance must always be finite.)

We often denote random variables using upper-case letters, as in $\{X_t\}$, and we denote data or observations, which we consider to be *realizations* of random variables, using lower-case letters, as in $\{x_t\}$. Unless it is important for clarity, I do not always make this distinction in my notation.

In Chapter 2 we discussed the nature of time series data, and the important difference between data that we can assume is equally spaced in time and data with more general time stamps. As we have pointed out there, weekends and holidays make most financial data unequally spaced. The R objects that handle unequally-spaced data must be somewhat more complicated than those for equally-spaced data.

Relationships among the Observations

In any statistical analysis, a major concern whether a single model can apply to all of the data in a given sample. In most analyses we just assume that a sin-

gle constant model is appropriate, and this is often a reasonable assumption. In models used in ordinary statistical analysis, we assume certain measures such as means and variances are constant. We often assume that the data come from a constant probability distribution, and that one observation does not affect any other observation. In this case, we say the data are "identically and independently distributed", or iid. A slightly weaker assumption to replace independence is "uncorrelated"; that is, the observations have 0 correlation with each other. (Independence implies 0 correlation, but in general, 0 correlation does not imply independence. In the special case of a normal distribution, 0 correlation does imply independence. A normal distribution is also called a "Gaussian distribution", especially by people working in time series analysis.)

In time series analysis, the question of constancy becomes more pertinent because we do expect changes over time. Time series analysis, in fact, is the study of the changes in time. If the time series were just a sequence of unrelated events, the methods of statistical inference, which involve aggregation of data, could not be used.

In models used in time series analysis, we may also assume that certain measures are constant. For some types of measures, we are more interested in a *conditional* measure; that is, the property of one element in the series conditional on previous elements. The conditional mean of X_t, for example, is

$$\mathrm{E}(X_t \mid X_{t-1}, X_{t-2}, \ldots).$$

It is a function of preceding observations. If *function* is constant, we can use statistical methods to make inferences about the data-generating process.

There are various extents to which a model may be constant, or "stationary". The concept of stationarity is important throughout the analysis of time series. We will define types of stationarity precisely in later sections.

Autocovariance and Autocorrelation

The property of a time series model that distinguishes it from other statistical models is the relationship between elements within the sequence. As with other statistical models, the most useful measure of this relationship is the *covariance* or the related *correlation*, both of which are based on deviations from the means. In a time series model, these measures that are of interest are those between elements of the sequence, so we refer to them as *autocovariance* and *autocorrelation*.

For a time series $\{X_t\}$, the *autocovariance between two elements* X_s and X_t that have finite means μ_s and μ_t is the ordinary covariance,

$$\mathrm{Cov}(X_s, X_t) = \mathrm{E}\left((X_s - \mu_s)(X_t - \mu_t)\right). \tag{5.2}$$

The variance of X_s, $\mathrm{V}(X_s)$, sometimes denoted as σ_s^2, is the covariance $\mathrm{Cov}(X_s, X_s)$.

The *autocorrelation between two elements* X_s and X_t that have finite variances σ_s^2 and σ_t^2 is the ordinary correlation,

$$\mathrm{Cor}(X_s, X_t) = \frac{\mathrm{Cov}(X_s, X_t)}{\sqrt{\sigma_s^2 \sigma_t^2}}. \tag{5.3}$$

These general definitions allow for the six measures, μ_s, μ_t, σ_s^2, σ_t^2, $\mathrm{Cov}(X_s, X_t)$, and $\mathrm{Cor}(X_s, X_t)$ to be different for different values of s and t. If there is no constancy, that is, stationarity, among these quantities or if there is no model of deterministic relationships among them, there would be no basis for statistical inference. In time series analysis, we make various levels of assumptions about what is constant.

Stationarity

One of the most common levels of assumptions is that μ_s and σ_s^2 are constant for all s, and $\mathrm{Cov}(X_s, X_t)$ is constant for any fixed lag $|s - t|$. This is usually what we mean by the unqualified term "stationary".

Under an assumption of stationarity, we can simplify the notations in the expressions of equations (5.2) and (5.3). We can denote the means and variances by μ and σ^2 without subscripts. The autocovariance and autocorrelation are functions only of the lag, $h = s - t$, and not of s and t. Let us denote the autocovariance at lag h as the function $\gamma(h)$,

$$\gamma(h) = \mathrm{E}\left((X_t - \mu)(X_{t-h} - \mu)\right). \tag{5.4}$$

We see that the constant variance σ^2 with this notation is

$$\sigma^2 = \gamma(0). \tag{5.5}$$

Let us denote the autocorrelation at lag h as the function $\rho(h)$. We have

$$\rho(h) = \frac{\gamma(h)}{\gamma(0)}. \tag{5.6}$$

Because we often deal with different time series, say $\{X_t\}$ and $\{Y_t\}$, we often use the notation for the separate time series to distinguish the measures that correspond to the different time series, for example, μ_X and μ_Y, σ_X^2 and σ_Y^2, $\gamma_X(h)$ and $\gamma_Y(h)$, and $\rho_X(h)$ and $\rho_Y(h)$.

The quantities defined in equations (5.2) and (5.3) or (5.4) and (5.6) are properties of the *random variables* in the time series *model*. There are also analogues of these model quantities for observed data. We call them the *sample autocovariance* and *sample autocorrelation*, although often we drop the "sample" when it is clear what we are referring to.

Sometimes to be precise, I will distinguish model quantities, as $\{X_t\}$ in expression (5.1), from corresponding sample quantities by the use of upper

case letters for the random variables and lower case letters for the realizations or observations, $\{x_t\}$,

$$\ldots, x_{t-1}, x_t, x_{t+1}, \ldots. \tag{5.7}$$

We will discuss the sample autocovariance and sample autocorrelation in Section 5.2.3, beginning on page 170.

5.1 White Noise and Related Time Series

In the analysis of time series, we begin with a few basic models that we often use as null cases in the analysis. The simplest of these models is white noise. A data-generating process yielding white noise has no properties of real interest. The time series itself has no interesting features that relate to passage of time. Hence, if we can establish that a given time series is *not* white noise, there must be something of interest in the data-generating process.

This approach, setting up a null strawman, is a fundamental plank in the scientific statistical method.

White noise is the basic strawman in time series analysis.

5.1.1 White Noise

A process $\{w_t\}$ that is a sequence of random variables with mean 0, constant finite variance, and 0 autocorrelations at all lags is called *white noise*. Because there is a similar model in continuous time, we sometimes call this *discrete white noise*. The 0 autocorrelation condition can be stated as

$$\mathrm{Cor}(w_t, w_{t+h}) = 0 \quad \text{for } h = 1, 2, \ldots \tag{5.8}$$

We will indicate that a sequence $\{w_t\}$ is a white noise by the notation

$$w_t \sim \mathrm{WN}(0, \sigma_w^2).$$

(A more appropriate notation would be "$\{W_t\} \sim \mathrm{WN}(0, \sigma_w^2)$", but in this book, I have generally chosen simpler notation unless it is incorrect or confusing.)

A white noise process does not need to have a "beginning", and for any t, $\mathrm{E}(w_t) = 0$ and $\mathrm{V}(w_t) = \sigma_w^2$.

A special type of white noise is *Gaussian white noise*, defined as white noise in which the random variables have a normal (Gaussian) distribution. In this case, the sequence is iid.

A Gaussian white noise of length 100 and with $\sigma_w^2 = 16$ can be simulated with the R statement

```
w <- rnorm(100, sd=4)
```

A white noise process in which the sequence is iid is called a *strict white noise* whether Gaussian or not, and the white noise process without independence is sometimes called a *weak white noise*. (Note that other authors use these terms somewhat differently. Some may impose further conditions on a sequence in order to call it a white noise. Some require that each term in the sequence has the same distribution in addition to equal first two moments, and/or require that the sequence be independent, instead of just zero-correlated, and/or may allow the mean to be nonzero.)

A strict white noise of length 100 following a t distribution with 3 degrees of freedom can be simulated with the R statement

```
w <- rt(100, df=4)
```

In this case, $\sigma_w^2 = 2$.

A simple linear regression with time as the independent variable and a white noise error term,

$$x_t \approx \beta t + w_t,$$

is a *white noise with drift*.

5.1.2 Linear Combinations of White Noise

A simple model based on white noise is a linear combination of white noise variates w_t of the form

$$x_t = \mu + \sum \psi_{tj} w_{tj}, \tag{5.9}$$

where μ and ψ_{tj} are constants.

A general *linear process* may have no beginning or ending; that is, the summation in the model may go from $-\infty$ to ∞. We require that

$$\sum_{j=-\infty}^{\infty} \psi_j < \infty.$$

Linear processes that depend only on the past are of interest. A *one-sided linear process* is a linear process that can be written in the form

$$x_t = \mu + \sum_{j=0}^{\infty} \psi_j w_{t-j}. \tag{5.10}$$

In a *finite* linear process, $\psi_j = 0$ for $j < k_1$ or $j > k_2$ for some finite constants k_1 and k_2.

A *moving average of a white noise* is a finite linear process of normalized sums of the form

$$x_t = \frac{1}{k}\sum_{j=1}^{k} w_{t-j}. \tag{5.11}$$

We call k the "window width" or the "band width".

We will also use the term "moving average" to refer to more general linear combinations of the form $x_t = \sum_j \psi_j w_{t-j}$, with various restrictions on ψ_j, but the word "average" in this case does not imply a weighted mean.

Notice that for a finite moving average of white noise,

$$x_t = \sum_{j=1}^{k} \psi_j w_{t-j},$$

we have

$$\mathrm{E}(x_t) = \sum_{j=1}^{k} \psi_j \mathrm{E}(w_{t-j}) = 0, \tag{5.12}$$

and

$$\mathrm{V}(x_t) = \sum_{j=1}^{k} \psi_j^2 \mathrm{V}(w_{t-j}) = \sum_{j=1}^{k} \psi_j^2 \sigma_w^2. \tag{5.13}$$

The covariance of two terms, $\mathrm{Cov}(x_t, x_{t+h})$, depends on whether there is any overlap in the white noise terms in x_t and x_{t+h}. If there is no overlap, the covariance is 0.

5.1.3 Random Walk

A simple model of a time series is the *random walk*, in which f is just x_{t-1},

$$x_t = x_{t-1} + w_t, \tag{5.14}$$

and $\{w_t\}$ is a white noise.

Although many time series models do not have a "beginning", a random walk process needs to have a starting point, which we usually denote as x_0, and run t over the positive integers.

A random walk is a simple constant diffusion process, but not all diffusion processes are random walks.

With a starting point of x_0, the random walk process in equation (5.14) can be expressed as

$$x_t = x_0 + \sum_{i=1}^{t} w_i. \tag{5.15}$$

The mean of a random variable following a random walk process starting at x_0 is

$$\begin{aligned} \mathrm{E}(x_t) &= \mathrm{E}(x_0) + \mathrm{E}\left(\sum_{i=1}^{t} w_i\right) \\ &= x_0; \end{aligned} \tag{5.16}$$

that is, the unconditional mean of a random walk is constant, depending only on x_0. The variance of a random walk variable, on the other hand, is

$$\begin{aligned} \mathrm{V}(x_t) &= \mathrm{V}(x_0) + \mathrm{V}\left(\sum_{i=1}^{t} w_i\right) \\ &= 0 + \sum_{i=1}^{t} \mathrm{V}(w_i) \\ &= t\sigma_w^2, \end{aligned} \tag{5.17}$$

that is, it depends on t and in fact grows linearly as t grows.

The R function `cumsum` together with a source of white noise can be used to generate a random walk.

```
w <- rnorm(100, sd=4)
x <- cumsum(w) + 10
```

In financial applications, we often model log returns as random walks. We use the standard deviation more often than the variance. The standard deviation grows as $\sqrt{t}$. This is why we annualize daily volatility σ_d by $\sqrt{253}\sigma_\mathrm{d}$.

5.1.4 Aggregated Log Returns

Because log returns are additive, for any time period, the accumulated return of a sequence of daily returns $r_1, \ldots, r_t$ over that period form a walk:

$$r_t = \sum_{i=1}^{t} r_i. \tag{5.18}$$

This is a random walk if, for the individual sub-periods, the log returns correspond to a random variable with 0 mean and constant finite variance, and if they have 0 correlation with each other. One form of the Random Walk Hypothesis assumes log returns follow a white noise process; hence, the log returns are a random walk under that hypothesis.

5.1.5 Random Walk with Drift

A variation is a *random walk with drift*, given by

$$x_t = \delta + x_{t-1} + w_t, \tag{5.19}$$

or

$$x_t = x_0 + t\delta + \sum_{i=1}^{t} w_i. \tag{5.20}$$

In financial applications, fixed payments constitute a drift, for example.

The random walk with drift δ has $\mathrm{E}(x_t) = t\delta + x_0$. This is an important difference; the mean of a random walk is independent of the time (it is constant), but the mean of the random walk with drift is dependent on the time.

5.1.6 Geometric Random Walk

A *geometric random walk* is a process whose logarithm is a random walk:

$$x_t = x_{t-1}\mathrm{e}^{w_t}, \tag{5.21}$$

or, starting at x_0,

$$x_t = x_0 \exp(w_t + \cdots + w_1). \tag{5.22}$$

Under a form of the Random Walk Hypothesis, cumulative log returns are a random walk, and so asset prices follow a geometric random walk.

The additive nature of log returns allows us to relate a principal amount at time t, P_t, in terms of the initial amount P_0, as

$$P_t = P_0 \exp(r_t + \cdots + r_1), \tag{5.23}$$

where $r_1, \ldots, r_t$ are log returns. (This results from equation (5.18).) If the log returns r_i are a white noise process, that is, if their sums form a random walk, the process $\{P_t\}$ forms a geometric random walk. Simple returns do not have this property.

If the log returns are from a Gaussian white noise process, that is, if the r_i in equation (5.23) are random variables distributed iid as $\mathrm{N}(0, \sigma^2)$, then P_t has a lognormal distribution. The process is called a *lognormal geometric random walk* with parameter σ.

5.1.7 General Autoregressive Processes

An extension of the random walk with drift model (5.19) is the autoregressive model,

$$x_t = \phi_0 + \phi_1 x_{t-1} + \cdots + \phi_p x_{t-p} + w_t. \tag{5.24}$$

We will discuss autoregressive models more fully in Section 5.3. In the random walk, $p = 1$, and in the simple random walk $\phi_0 = 0$.

5.1.8 Multivariate Processes

All of the models we have described above are univariate; that is, the individual observations and random variables are scalars. Each model has a multivariate generalization, however.

The multivariate generalization of any of these models retains the model's essential characteristics from one observation to another; that is, from one point in time to another point in time. The elements of the vector variables in the multivariate models at a given point in time, however, may have any kind of relationships to each other.

For example, the elements of a multivariate white noise process may have nonzero correlations among themselves, but the correlations of a given element over time are zero. If

$$\ldots, w_{-1}, w_0, w_1, \ldots$$

are random d-vectors with constant mean 0 and constant $d \times d$ variance-covariance matrix Σ_w, the process $\{w_t\}$ is a *multivariate white noise* process if the covariance matrix of w_s and w_t, $\mathrm{E}(w_s w_t^{\mathrm{T}})$, for $s \neq t$ is zero.

Likewise, we can define multivariate moving averages of white noise, multivariate linear processes, multivariate random walks, and so on. A multivariate random walk, for example, has the same form as the model (5.14),

$$x_t = x_{t-1} + w_t,$$

except that x_t and x_{t-1} are vectors, and $\{w_t\}$ is a multivariate white noise.

There are several simple models of time series that are the be

The R function `cumsum` together with a source of white noise can be used to generate a random walk.

Exercises: White Noise and Related Time Series

5.1.1. xxx.
5.1.2. yyy.

5.2 Basic Operations on Time Series Data

In this section we describe some simple operations on time series. Most of the operations can be performed whether or not the data are equally spaced, but in some cases the interpretation of the results of the operation are not meaningful if the data are not equally spaced.

The R `stats` package provides extensive capabilities for working with time series with equally-spaced times. The `stats` package provides a time series class, `ts`, which causes some R functions to produce results that are more appropriate for a time series (such as the generic `plot` producing a graph with connecting line segments). There are also special functions in the `stats`

package for working with `ts` objects, such as the `diff` function, which produces a time series of differences with a specified lag. Class `ts` only deals with numeric time stamps.

There are a number of R packages for working with and analyzing time series. Some of these packages allow more flexibility in date data and some provide special classes for time series objects, and they provide functions for various kinds of analyses of time series. Some of the available packages can be seen under the TimeSeries section at

`https://cran.r-project.org/web/views/`

Various packages implement irregular time series based on POSIXct time stamps.

The R package `timeDate` contains functions to provide financial date and time information, including information about weekends and holidays for various stock exchanges.

5.2.1 The Shift and Difference Operators

One of the most useful linear filters is the *backshift operator*, $\mathrm{B}(\cdot)$, which for the time series

$$\ldots, x_{i-1}, x_i, x_{i+1}, \ldots$$

is defined by

$$\mathrm{B}(x_t) \equiv x_{t-1}. \tag{5.25}$$

This is also called a *lag operator*, and it is sometimes denoted as $\mathrm{L}(\cdot)$.

A backshift on a time series in R can be performed simply by manipulating the index of a vector (or matrix).

```
> x <- c(1, 2, 3, 3, 2, 1)
> n <- length(x)
> x
[1] 1 2 3 3 2 1
> c(NA,x[-n])                    #  backshift once
[1] 2 3 3 2 1
> c(NA,NA,x[-c(n-1,n)])          #  backshift twice
[1] NA NA  1  2  3  3
> k <- 2
> c(rep(NA,k),x[-c((n+1-k):n)])  #  backshift k times
[1] NA NA  1  2  3  3
> c(x[-c(1,2)],NA,NA)            #  forward shift twice
[1]  3  3  2  1 NA NA
```

The `lag` function in the `dplyr` package of the `tidyverse` suite does backshifts as above, and the associated `lead` does forward shifts. The resulting vectors are the same length as the operand vector, so they contain some NA entries as above. (The `lag` function in the R `stats` package does not do a backshift.)

In R, the `diff` function performs differencing on a time series. On a numeric vector or an object of class `ts`, the `diff` function yields an object of shorter length than the operand; that is, there are no NAs, as in the lag filter illustrated above. By default, the `diff` function performs differencing at lag 1. Differencing at a lag greater than 1 can be specified by a second argument.

```
> x <- c(1, 2, 3, 3, 2, 1)
> x
[1]  1  2  3  3  2  1
> diff(x)                # difference of order 1
[1]  1  1  0 -1 -1
> x[-1] - x[-length(x)]  # difference of order 1
[1]  1  1  0 -1 -1
> diff(diff(x))          # difference of order 2
[1]  0 -1 -1  0
> diff(diff(diff(x)))    # difference of order 3
[1] -1  0  1
> diff(x,2)              # difference at lag 2
[1]  2  1 -1 -2
```

The `diff` function performs differently on an object of class `xts`. It yields NAs in that case, similar to the NAs in the lag filter illustrated on page 168.

5.2.2 Returns

The backshift and difference operators are central to the computation of returns. A one-period simple return (equation (1.3), page 101) is

$$R = \triangle(x_t)/\mathrm{B}(x_t),$$

and the log return is

$$r = \triangle(\log(x_t)).$$

The sequence of returns is a filter of the time series of prices.

In R, these returns are computed using the `diff` function:

```
xsimpret <- diff(x)/x[-length(x)]
xlogret <- diff(log(x))
```

(Again, recall the difference in operations on `xts` objects.)

For any filter on a finite vector that results in a shorter vector, such as computation of returns, differencing, or backshifting, the computer software designer must decide whether to return a shorter vector (as `diff` in the `base` package) or to return a vector of the same length with some meaningless values, usually NAs (as `diff` in the `quantmod` package or `lag` in the `dplyr` package). And, of course, the software user must be aware of the decision.

5.2.3 The Sample Autocorrelation Function

We defined the autocovariance and autocorrelation, $\mathrm{Cov}(X_s, X_t)$ and $\mathrm{Cor}(X_s, X_t)$, for general time series in equations (5.2) and (5.3). If the terms in these expressions are different for different values of s and t, then with a sample of multiple observations, we have only one statistic each for μ_s and μ_t, and we have no statistic for the quantities σ_s^2 and σ_t^2 or $\mathrm{Cov}(X_s, X_t)$ and $\mathrm{Cor}(X_s, X_t)$. Statistical analysis has no basis unless we assume some degree of constancy.

In time series analysis, we generally assume that μ_t and σ_t^2 are constant for all t.

We also generally assume weak stationarity; that is $\mathrm{Cov}(X_s, X_t)$ and $\mathrm{Cor}(X_s, X_t)$ do not depend on s and t individually, but only on the lag $h = s - t$. Hence, for stationary time series, we have the autocovariance and autocorrelation functions, $\gamma(h)$ and $\rho(h)$ in equations (5.4) and (5.6), which depend only on the lag h.

Under this assumption, given observations $x_1, \ldots, x_n$, we have a basis for estimating $\gamma(h)$ and $\rho(h)$, using sample analogues of the autocovariance and autocorrelation functions.

We will denote the sample quantities as $\hat{\gamma}(h)$ and $\hat{\rho}(h)$.

The basic rule is to replace an *expectation* in a model with an *average* of observed sample quantities; for example, the model quantity

$$\mu = \mathrm{E}(X)$$

is replaced by the sample mean

$$\bar{x} = \frac{1}{n}\sum_{t=1}^{n} x_t.$$

Often in statistical data analysis, especially for second order quantities such as variances and covariances, instead of dividing by n, we divide by a modified quantity such as $n - 1$ in the familiar expression for the sample variance,

$$s^2 = \frac{1}{n-1}\sum_{t=1}^{n}(x_t - \bar{x})^2. \tag{5.26}$$

Such adjustments may be done to make the sample quantity have better statistical properties, such as unbiasedness in the case of s^2.

Now, consider the autocovariance at lag h:

$$\mathrm{E}\left((X_t - \mu)(X_{t-h} - \mu)\right).$$

From a sample $x_1, \ldots, x_n$, the obvious analogue is some average of all $(x_t - \bar{x})(x_{t-h} - \bar{x})$, that is the products using all observations the are h steps apart.

There are obviously $n - k$ such pairs; however, by convention, we define the *sample autocovariance function* as

$$\hat{\gamma}(h) = \frac{1}{n} \sum_{t=h+1}^{n} (x_t - \bar{x})(x_{t-h} - \bar{x}). \tag{5.27}$$

The statistic $\hat{\gamma}(h)$ is not unbiased for $\gamma(h)$ at any value of h, but it is consistent at any fixed value of h.

From (5.27), we have

$$\hat{\sigma}^2 = \hat{\gamma}(0) = \frac{1}{n} \sum_{t=1}^{n} (x_t - \bar{x})^2. \tag{5.28}$$

Although this is not unbiased for σ^2 as s^2 is, it is consistent.

This yields the *sample autocorrelation function*:

$$\begin{aligned} \hat{\rho}(h) &= \frac{\hat{\gamma}(h)}{\hat{\gamma}(0)} \\ &= \frac{\sum_{t=h+1}^{n} (x_t - \bar{x})(x_{t-h} - \bar{x})}{\sum_{t=1}^{n} (x_t - \bar{x})^2}. \end{aligned} \tag{5.29}$$

We sometimes denote the sample autocorrelation function as SACF, or just as ACF, the same as the model autocorrelation function $\rho(h)$. The sample autocorrelation function $\hat{\rho}(h)$ is consistent for $\rho(h)$ at each value of h.

5.2.4 The Partial Autocorrelation Function

5.2.5 Autocorrelation Functions in R

`acf` `Acf` `forecast`
`pacf`

Exercises: Basic Operations on Time Series Data

5.2.1. xxx.
5.2.2. yyy.

5.3 Linear Models for Time Series

Hyndman and Khandakar (2008), Automatic time series forecasting: The forecast package for R,

As we have emphasized, statistical analysis begins with a model. The model may just describe a probability distribution or it may describe relationships among observable variables. The model may be very general, it may have unspecified parameters, or it may be very specific.

A model provides a *smoothing* of the data. This means an equation, a line, or just a rule that provides "smoothed" values for components of each observation in the data such that the smoothed values appear less noisy that the original observed values. Examining the smoothed values rather than the raw data is akin to studying a forest without being distracted by the trees.

The main objectives of statistical data analysis are to understand the data-generating process and/or to make predictions of events that have not yet been observed.

We accomplish these objectives using various coordinated approaches.

For a given dataset or data-generating process, we may

- develop and fit a model
- compute and model overall summary properties
- identify and analyze anomalous data
- subregions of the data that correspond to different generating processes

These steps are performed iteratively, more or less in the order shown.

A common form of statistical models for the relationship of the variable y to the variables $x_1, \ldots, x_m$ is as shown in equation (1.27) on page 125,

$$y \approx f(x_1, \ldots, x_m),$$

or

$$y = f(x_1, \ldots, x_m) + \epsilon,$$

where ϵ represents a random error.

A more specific form of this model is the familiar linear regression model

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_m x_m + \epsilon.$$

Models for time series naturally involve observations of the same variable at different times. A model for a univariate time series $\{x_t\}$ is of the form

$$x_t \approx f(x_{t-1}, x_{t-2}, \ldots). \tag{5.30}$$

A linear model for a univariate time series $\{x_t\}$ is of the form

$$x_t \approx \phi_0 + \phi_1 x_{t-1} + \cdots + \phi_p x_{t-p}. \tag{5.31}$$

We may rewrite this expression with an additive error term. In a time series, an appropriate error term may be a white noise, so we have

$$x_t = \phi_0 + \phi_1 x_{t-1} + \cdots + \phi_p x_{t-p} + w_t,$$

where $\{w_t\}$ is a white noise process. This is called an "autoregressive" model.

Another useful linear time series model represents $\{x_t\}$ as a linear combination of the past elements in a white noise process:

$$x_t = w_t + \theta_1 w_{t-1} + \cdots + \theta_q w_{t-q},$$

This is called a "moving average" model.

5.3.1 Fitting Parametric Models

The autoregressive and moving average models mentioned above are parametric linear models. There are other linear parametric time series models that are variations on these

A first step in the analysis of data following a parametric model is to estimate the parameters in the model. This means to fit the model.

As we have mentioned there are several methods that have been developed and studied in the theory of statistics to estimate parameters. In the case of both autoregressive and moving average models, the most common estimators are based on a maximum likelihood approach (assuming the white noise is Gaussian white noise).

Denoting the estimates of the parameters using a caret or hat, we may write a fitted autoregressive model as

$$x_t = \hat{\phi}_0 + \hat{\phi}_1 x_{t-1} + \cdots + \hat{\phi}_p x_{t-p}.$$

A model evolves during statistical analysis. It may begin with a tentative identification of the relevant variables. If there is just one variable of interest, the model may just be a probability distribution. A realistic probability distribution, even for just one variable, is rarely a single simple distribution; it is almost always a mixture of distributions, each of which may be similar to some standard distribution, such as a normal distribution or a Pareto distribution, for example.

After fitting a model, it should be evaluated using various summary statistics computed from the available data. Any observations that do not seem consistent with the model should be identified and studied further.

A model obviously applies only to some specified population or time period. The limits of applicability of a model should be noted. In the case of time series models, even for a single variable, the model may change over time. Identification of *changepoints* is an important aspect of time series analysis.

5.3.2 Autoregressive Models

$$x_t = \phi_0 + \phi_1 x_{t-1} + \cdots + \phi_p x_{t-p} + w_t, \tag{5.32}$$

where $\{w_t\}$ is a white noise process.

5.3.3 Moving Average Models

$$x_t = w_t + \theta_1 w_{t-1} + \cdots + \theta_q w_{t-q}, \tag{5.33}$$

5.3.4 Integrating Models; Differencing

5.3.5 ARIMA Models

5.3.6 R Packages and Functions

5.3.7 Extensions

Exercises: Linear Models for Time Series

5.3.1. xxx.
5.3.2. yyy.

5.4 Nonparametric Analysis of Time Series

A nonparametric model is a general model whose main objective is to smooth the data or to provide predictions. A nonparametric model is often an algorithm or a formula, rather than a simple equation, as is used in most parametric models.

The general form of a nonparametric model for a time series model is the same as a parametric model,

$$x_t \approx f(x_{t-1}, x_{t-2}, \ldots),$$

except that the function f cannot be written as some simple equation involving the $x_{t-1}, x_{t-2}, \ldots$, and we do not attempt to express the functional relationship f completely.

While time series models may not have a beginning point, in applications, all datasets are finite. We assume a given finite set of observations, $x_1, \ldots, x_{t-1}$.

The objective is to use the set of observations to determine a function $\mathcal{S}$ that predicts x_t,

$$\hat{x}_t = \mathcal{S}(x_{t-1}, \ldots, x_1). \tag{5.34}$$

We call $\mathcal{S}$ a smoothing function.

5.4.1 A General Approach to Smoothing Time Series

Smoothing a time series is based on a general assumption that x_t is related in some systematic way to $x_1, \ldots, x_{t-1}$; for example, x_t may be approximately the mean,

$$x_t \approx \frac{1}{t-1} \sum_{i=1}^{t-1} x_{t-i}.$$

This is an intuitively appealing approach.

Moving Averages

At time t, instead of taking $\hat{x}_t$ as the average of all preceding values, it might be appropriate to limit the averaging to values closer in time to t. We introduce a positive integer $h \leq t-1$ and define the smoother as

$$\begin{aligned}\hat{x}_t &= \mathcal{S}_{\mathrm{MA}(h)}(x_{t-1}, \ldots, x_{t-h}) \\ &= \frac{1}{h}\sum_{i=1}^{h} x_{t-i}. \end{aligned} \tag{5.35}$$

The smoother $\mathcal{S}_{\mathrm{MA}(h)}$ is a *moving average with a window of length* h. For a time series in which the unit of time is a day, this is an "h-day moving average". In smoothing daily financial time series such as stock prices, a 50-day moving average or a 200-day moving average. The larger is h, the more slowly the moving average changes from on day to the next; that is, the "smoother" is the fitted series.

Weighted Moving Averages

At time t, instead of averaging all previous h values equally, it may be better to use a weighted average in which values closer in time are weighted more heavily. We have a smoother of the form

$$\begin{aligned}\hat{x}_t &= \mathcal{S}_{\mathrm{K}(h)}(x_{t-1}, \ldots, x_1) \\ &= \frac{1}{h}\sum_{i=1}^{t-1} x_{t-i} K\left(\frac{t-i}{h}\right), \end{aligned} \tag{5.36}$$

where h is a positive number less than t. The function $K(\cdot)$, which must be nonnegative, is called a kernel function and the smoother in equation (5.36) a time series *kernel smoother*.

We discussed kernel functions on page 112 for use in probability density estimation, and gave examples, such as the triangular kernel in equation (1.21),

$$K_{\mathrm{T}}(t) = \begin{cases} 1-|t| & \text{if } |t| \leq 1/2 \\ 0 & \text{otherwise.} \end{cases}$$

Kernel methods generally depend on a *smoothing parameter*, h, used both in the kernel density estimate in equation (1.19) and in the smoother in equation (5.36). The smoothing parameter is sometimes called a "window width" just as in the moving average smoother, but it does not delimit the distance back in time that observations can be used in the smoothing function.

Kernel functions and kernel-based smoothing methods are used in many areas of nonparametric data analysis, but time series smoothing is a special kind of smoothing. A time series smoother at any given point can only

depend on values before that point in time. Note that the summation in equation (5.36) includes all observations up to time t. Of the various kernels used in density estimation, most are symmetric about 0, such as the rectangular, triangular, and Gaussian (normal) that we mentioned. The kernel used in time series smoothing, however, must be one-sided. A folded version of a symmetric kernel can be used. A folded triangular kernel, for example, is

$$K_{\mathrm{FT}}(t) = \begin{cases} 1+t & \text{if } -1 \le t \le 0 \\ 0 & \text{otherwise.} \end{cases}$$

Just as a histogram corresponds to a kernel density estimate with a rectangular kernel, a moving average smoother corresponds to a kernel smoother in which the kernel is a folded rectangular kernel; thst is, instead of equation (1.20), we have

$$K_{\mathrm{FR}}(t) = \begin{cases} 1 & \text{if } -1 \le t \le 0 \\ 0 & \text{otherwise.} \end{cases}$$

Exponentially Weighted Moving Average

A moving average or a kernel-based smoother with a finite window can be updated recursively.

***

A variation of an MA is called an *exponentially weighted moving average* (EWMA). An EWMA is a weighted average of the current price (in financial applications, the previous day's price) and the previous EWMA. If at time t, the current price is p_t, and the proportion assigned to the current price is α, then

$$\mathrm{EWMA}_t = \alpha p_t + (1-\alpha)\mathrm{EWMA}_{t-1}. \tag{5.37}$$

The recursive formula (5.39) does not indicate how to get a value of the EWMA to get started. In practice, this is usually done by taking as a starting point an MA for some chosen number of periods k. Although an EWMA at any point in time includes all previous prices since the inception of the average, it is often expressed in terms of a number of periods, say k, just as in ordinary MAs. The number of periods is defined as $2/\alpha - 1$. Other ways of sequentially weighting the prices are often used. A popular one is called "Wilder smoothing". Some MAs with different names are different only in how the "period" is used in forming the weight.

The `movavg` function in the `pracma` package does not use [e - "exponential"] computes the exponentially weighted moving average. The exponential moving average is a weighted moving average that reduces influences by applying more weight to recent data points () reduction factor 2/(n+1); or

Summary Statistics

Statistics resulting from use of the model to smooth the data ... ***

Outliers

Observations that do not seem consistent with the model ... ***

Changepoints

Time points at which the model changes in some way ... ***

Local Smoothing

Most nonparametric methods of smoothing are called "local". At any observation in the time series, nearby observations are used to compute the smoothed value to associate with the given observation.

***two main types - averaged values or "representative values"

One of the simplest ways of local smoothing time series data is just to replace each value with an average of it and its nearby values. The more nearby values used, the smoother the result will be.

***window width

Figure 5.1 shows a simple time series with 37 observations. The smoothed version shown in red was formed by averaging three values around and including each observation. The smoothed version shown in blue was formed by averaging five values around and including each observation. It is smoother than the version using only three values. The ultimate of this smoothing would be to use all available values. This is what is shown in green in Figure 5.1.

how to average? *** kernel weighting

how to average? ***mean or median – order statistics within the window

*** what to do at beginning and end?

see below

other methods not based on averaging (ATS e.g.)

*** any smoothing method can be iterated: smooth the smooth

Smoothing of Time Series

simple questions: centered or lagging?

Datasets for Examples and Illustrations

To explore various method of nonparametric smoothing of time series, we will use the same few example datasets.

- Small simple example, `x`. This is the example shown in Figure 5.1.

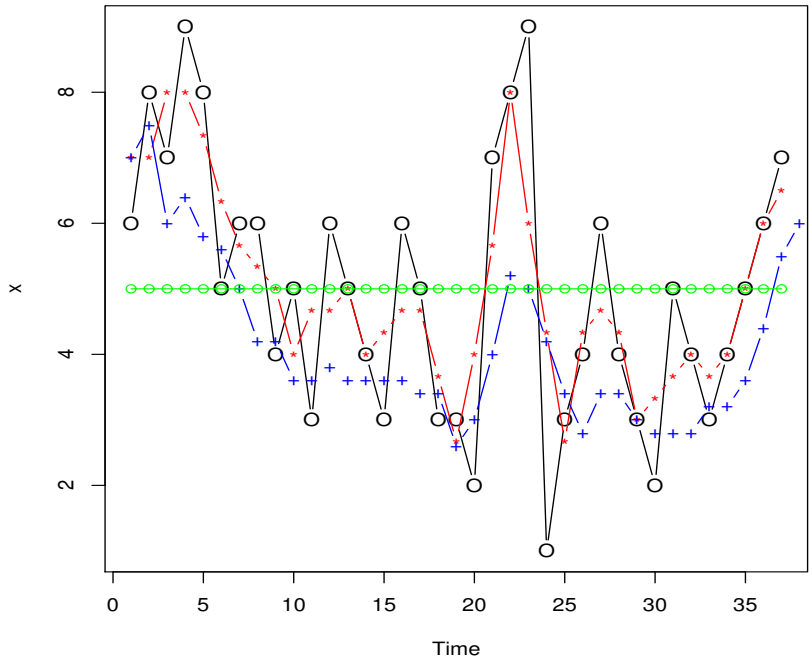


Figure 5.1. Smoothing by Local Averaging

```
x <-
c(6,8,7,9,8,5,6,6,4,5,3,6,5,4,3,6,5,3,3,2,7,8,9,1,3,4,6,4,3,2,5,4,3,4,5,6,7)
```

- INTC daily closes, first three quarters, 2017.

```
library(quantmod)
z <- getSymbols("INTC", env=NULL, from="2017-1-1",
                        to="2017-10-1", periodicity="daily")
INTCd2017Q3 <- as.numeric(z[,6])
```

- IBM daily closes, 1970 through 2014.

```
library(quantmod)
z <- getSymbols("IBM", from="1970-1-1",
                       to="2014-12-31", periodicity="daily", warnings=FALSE)
IBMd <- as.numeric(z[,6])
```

These three time series are shown in Figure 5.2. Note that the time scales are very different.

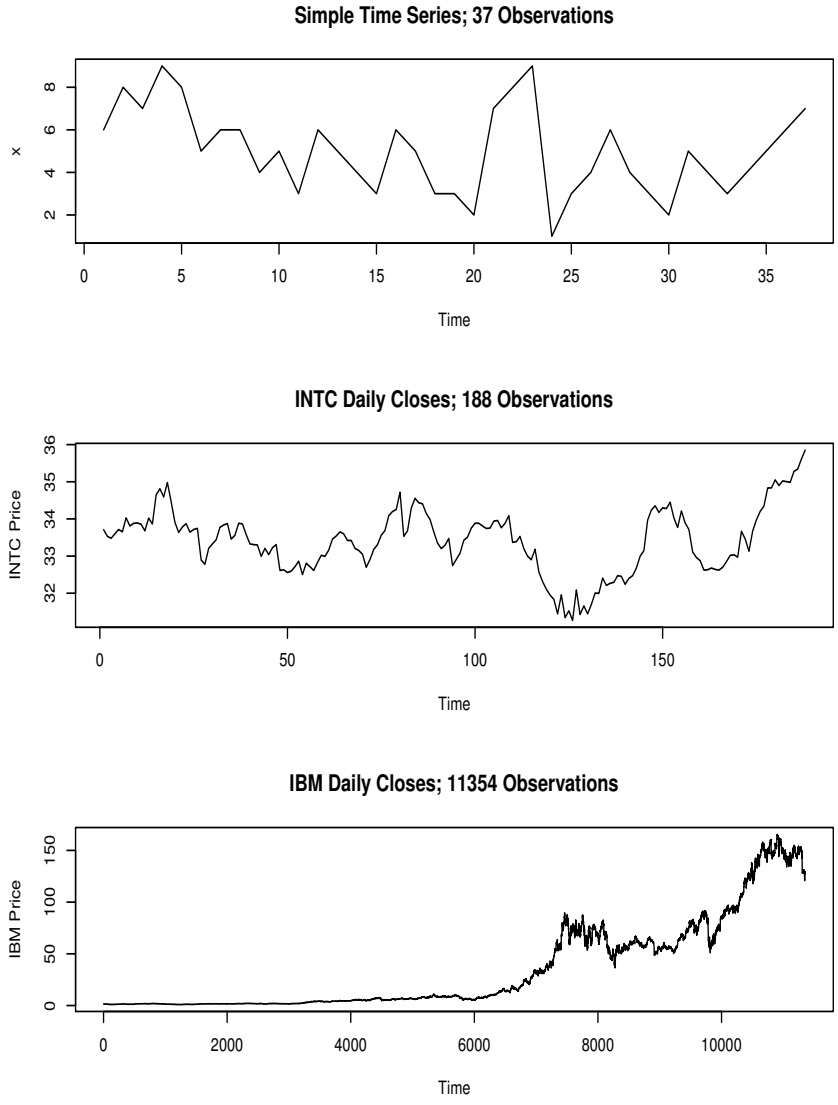


Figure 5.2. Times Series for Examples

5.4.2 Moving Averages

Moving averages are averages of nearby observations within a specified window width, as shown in Figure 5.1. The averages can be computed using the data on either side of the given point, or they can use only data on one side or the other. In financial applications, obviously only use of prior data makes any sense.

There are two simple remaining issues; what to do at the beginning of the time series before there are enough data to fill a window width, and secondly, whether to include the current observation, or only the previous observations.

There is no function in the basic R package that computes a moving average, but several packages, including `pracma` and `quantmod`, provide functions for computing moving averages. Functions from different packages do computations in different ways.

The simplest way that we will generally use for illustration is as follows. Given the time series $x_1, x_2, \ldots$, for the window width T, the moving average values $m_1, m_2, \ldots$ are

$$\begin{aligned} m_1 &= x_1 \\ m_i &= \tfrac{1}{i-1} \textstyle\sum_{j=1}^{i-1} x_j \quad \text{for } 2 \le i \le T \\ m_i &= \tfrac{1}{T} \textstyle\sum_{j=i-T-1}^{i-1} x_j \quad \text{for } T < i \end{aligned} \tag{5.38}$$

Variations on the Averaging

An obvious modification of simple averaging within a moving window is to weight the observations differently, giving higher weight to the observations nearer to the point where the average is computed. Again, for a time series, the moving average should be based only on prior values, although if the current value is included, the effect is negligible, depending on the window width of course.

The `movavg` function in the `pracma` package does not use the observations farther ahead in time, so it is appropriate for practical usage in time series. The `movavg` function allows for both a simple moving average based on the mean and a weighted moving average, in which the which the ***

various types of weighted moving averages. In each case, however, the moving average includes the current point. A "simple" moving average is the mean described above, except that the current point is included. A

c computes the simple moving average. n indicates the number of previous data points used with the current data point when calculating the moving average.

[t - "triangular"] computes the triangular moving average by calculating the first simple moving average with window width of ceil(n+1)/2; then it

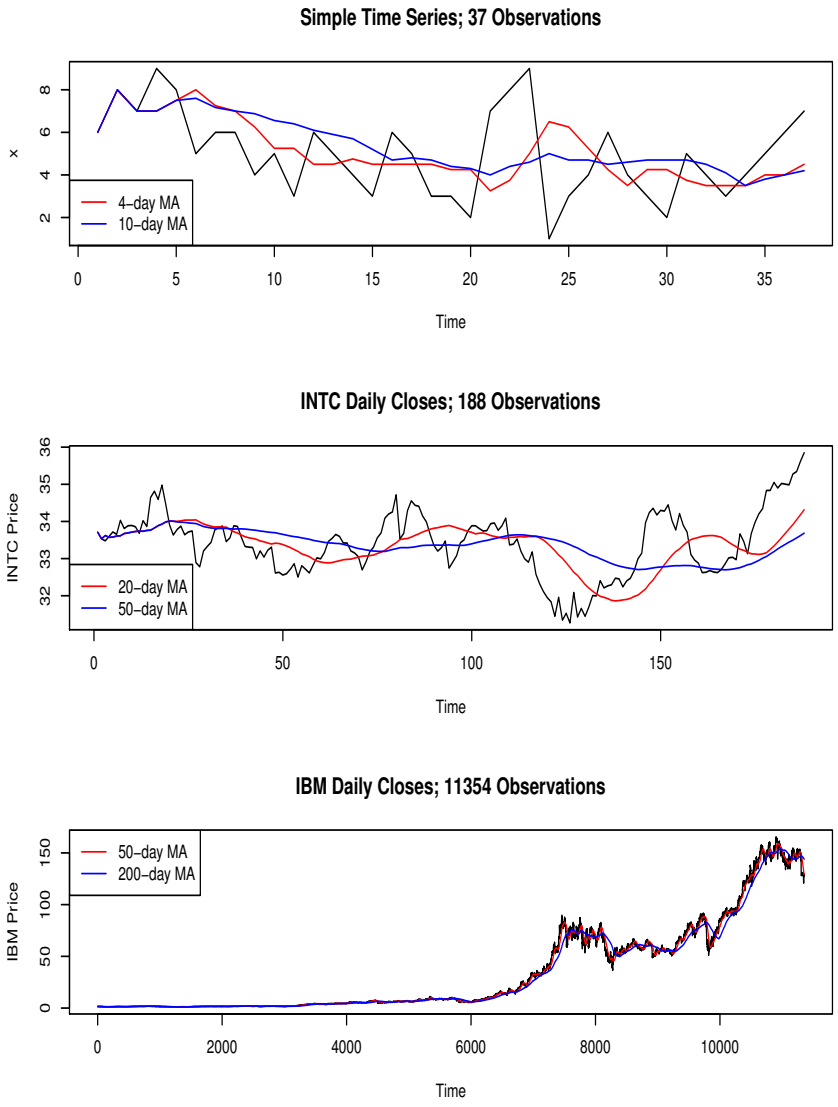


Figure 5.3. Moving Averages of Various Window Widths for the Examples

calculates a second simple moving average on the first moving average with the same window size.

[w - "weighted"] calculates the weighted moving average by supplying weights for each element in the moving window. Here the reduction of weights follows a linear trend.

Exponentially Weighted Moving Average

A variation of an MA is called an *exponentially weighted moving average* (EWMA). An EWMA is a weighted average of the current price (in financial applications, the previous day's price) and the previous EWMA. If at time t, the current price is p_t, and the proportion assigned to the current price is α, then

$$\text{EWMA}_t = \alpha p_t + (1 - \alpha)\text{EWMA}_{t-1}. \tag{5.39}$$

The recursive formula (5.39) does not indicate how to get a value of the EWMA to get started. In practice, this is usually done by taking as a starting point an MA for some chosen number of periods k. Although an EWMA at any point in time includes all previous prices since the inception of the average, it is often expressed in terms of a number of periods, say k, just as in ordinary MAs. The number of periods is defined as $2/\alpha - 1$. Other ways of sequentially weighting the prices are often used. A popular one is called "Wilder smoothing". Some MAs with different names are different only in how the "period" is used in forming the weight.

The `movavg` function in the `pracma` package does not use [e - "exponential"] computes the exponentially weighted moving average. The exponential moving average is a weighted moving average that reduces influences by applying more weight to recent data points () reduction factor 2/(n+1); or

Moving Averages in `quantmod`

`addDEMA` Add Moving Average to Chart
`addEMA` Add Moving Average to Chart
`addEVWMA` Add Moving Average to Chart
`addMA` Add Moving Average to Chart
`addSMA` Add Moving Average to Chart
`addWMA` Add Moving Average to Chart
`addZLEMA` Add Moving Average to Chart

`add_DEMA` Add Moving Average to Chart
`add_EMA` Add Moving Average to Chart
`add_EVWMA` Add Moving Average to Chart
`add_GMMA` Add Moving Average to Chart
`add_SMA` Add Moving Average to Chart
`add_VMA` Add Moving Average to Chart
`add_VWAP` Add Moving Average to Chart
`add_WMA` Add Moving Average to Chart

`addMACD` Add Moving Average Convergence Divergence to Chart

5.4.3 Variations on the Averaging: Kernel Smoothing

The term "kernel" is used in various areas of mathematics to denote various types of mathematical objects. In statistics and data science, it usually is a function that allows local aggregation of data. If the data are counts of observations at or near a given point, a kernel function provides a weighting of the nearby points to compute a local density, as we discussed in Section 2.1. This application is discussed in Silverman (1986), as well as in many other places.

Another application of a kernel function is in local smoothing of the relationship of one variable y to another variable x. Instead of a linear regression of y on x, we can fit a local smooth

$$\hat{y} \approx f(x),$$

in which the function f is not explicitly identified. This type of application is discussed in Wand and Jones (1994), as well as in many other places.

For a given set of observations on y on x, we can use observations on either side of a given point to fit $\hat{y}$. In a time series, it is not possible to use future observations in the fit; hence the kernel used to weight the observations must apply only to the prior observations, or give 0 weight to future observations. Ghosh (2018) discussed this type of application, even with irregularly-spaced observations in the time series.

It would make sense for a smoothing kernel for a time series to be asymmetric, tailing off for observations farther back in time. Asymmetric kernels are just as tractable as the more common symmetric ones. Silverman (1986) discussed their use and possible advantages in density estimation. A simple compact asymmetric kernel is a beta density.

Hirukawa (2018) disucsses asymmetric kernels, primarily one with support on the positive real line, $\mathbb{R}_+$. These kernels could include the densities of common distributions such as gamma, lognormal, and so on.

Folded Kernels

We propose a new class of asymmetric kernels: *folded kernels.*

folded Gaussian
folded Epanechnikov
folded triangular
folded biweight

Nonparametric Smoothing of Time Series Using Folded Kernels

folded triangular is same as "weighted" in `movavg{pracma}` except that `movavg` includes the current point

5.4.4 Alternating Trend Smoothing

Works by fitting line segments with alternating up-down trends between changepoints.

identifies changepoints, which can be used for further smoothing, either with ATS or with other methods; e.g. knots in splines

Simple use: find patterns

Smoothing Parameter

The smoothing parameter ***

Changepoints

*** mentioned use of changepoints for further smoothing.

*** how about for a second ATS smoothing

Figure 5.7 shows an ATS smooth of the changepoints identified in a previous ATS smooth of the raw data.

Evolution in Time

5.4.5 Smoothing Splines

Splines are piecewise polynomials each of which approximate data over some region and connects smoothly with the adjoining polynomials. If the approximations are exact, that is, if the polynomials interpolate the data, the set of polynomials is called an *interpolating spline*.

Figure 5.9 shows a cubic smoothing spline fit changepoints identified by ATS smooth for the knots.

Exercises: Nonparametric Analysis of Time Series

5.4.1. xxx.
5.4.2. yyy.

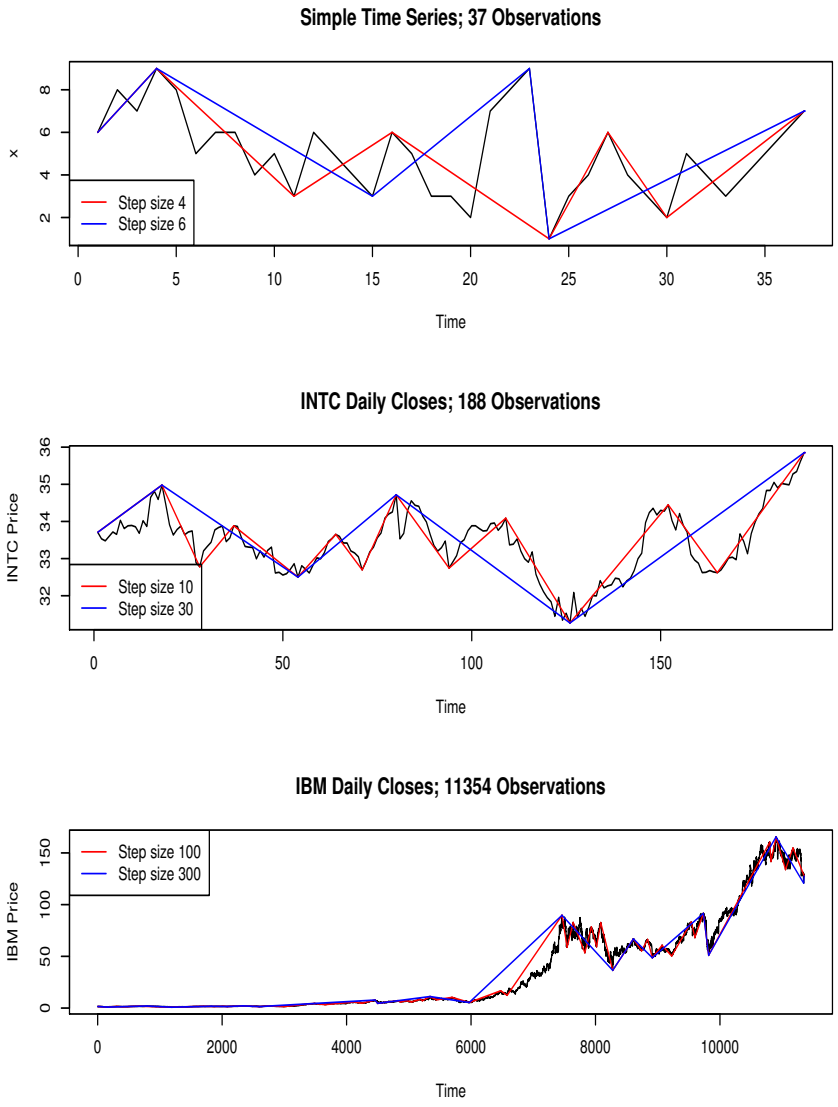


Figure 5.4. ATS of Various Step Sizes for the Examples

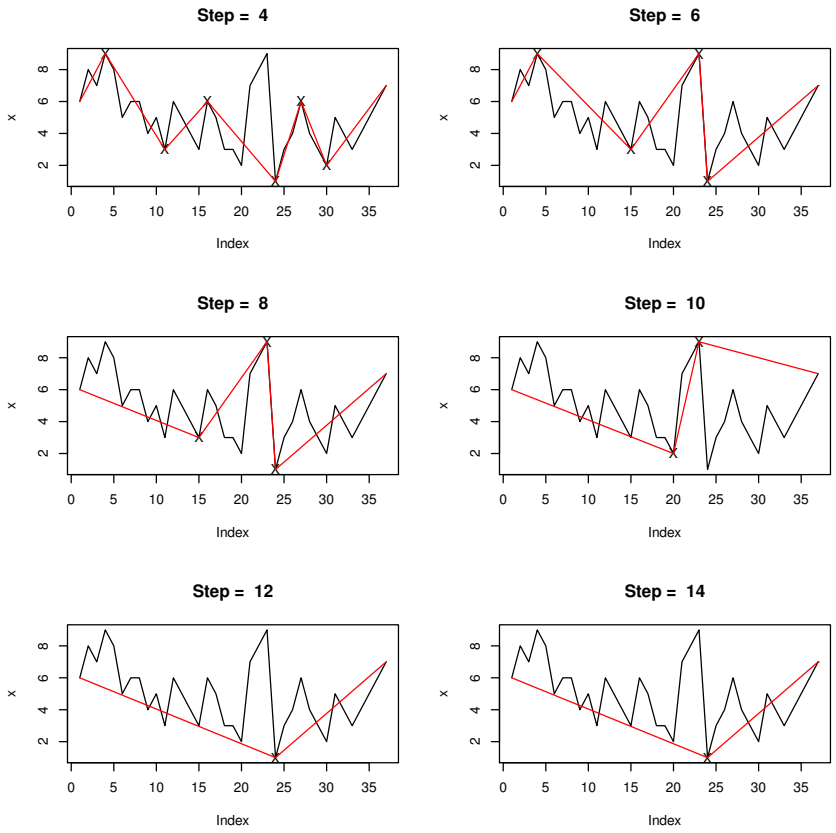


Figure 5.5. Effect of Different Smoothing Parameters

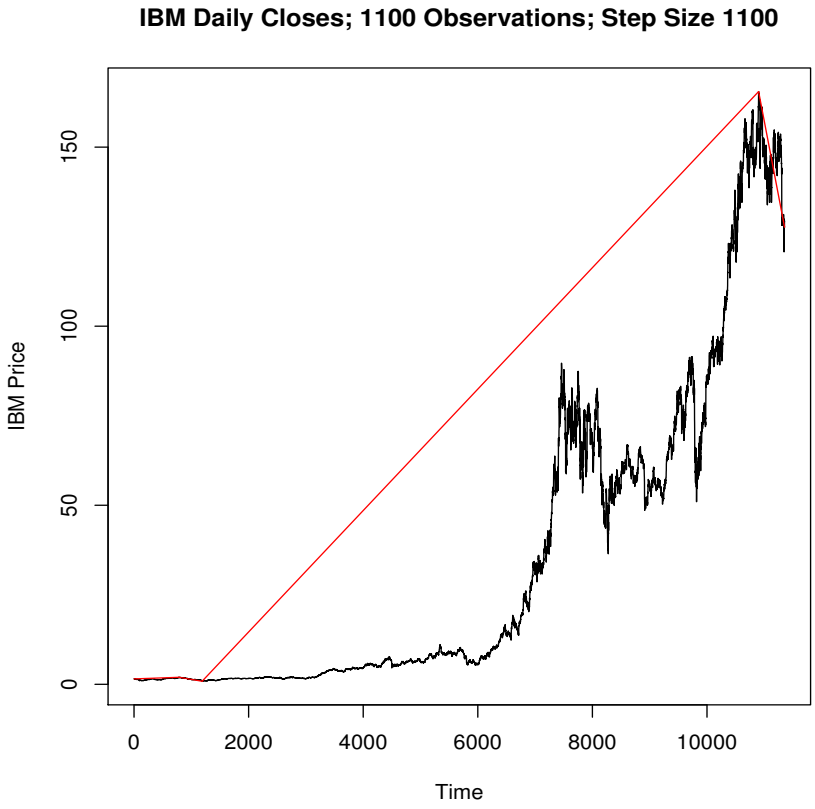


Figure 5.6. Very Wide Window on a Long Time Series

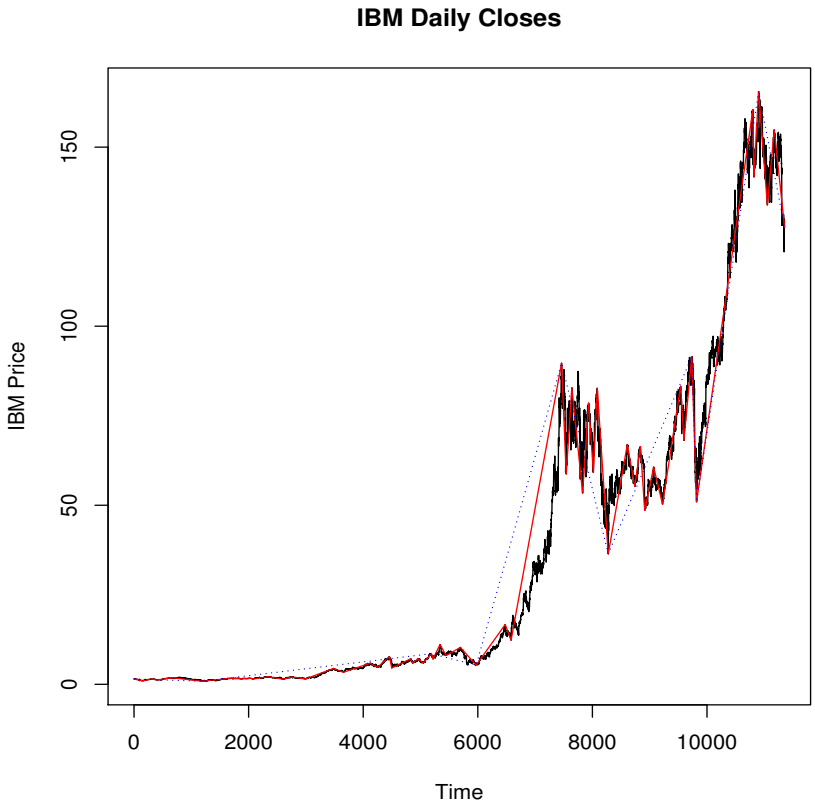


Figure 5.7. ATS Fit to ATS-Identified Changepoints

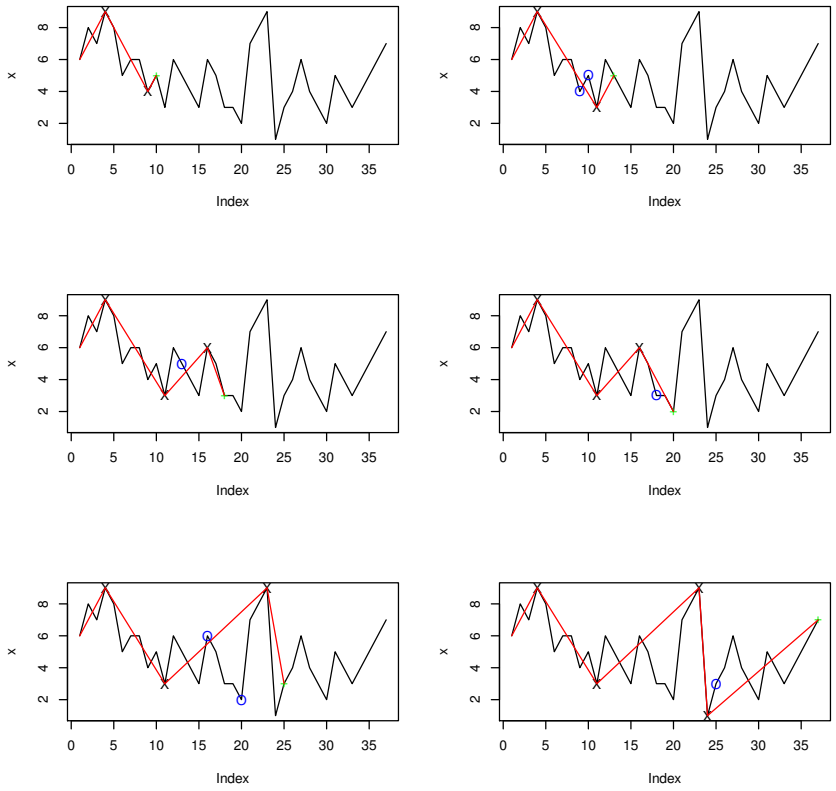


Figure 5.8. The Evolution of an ATS Fit over Time

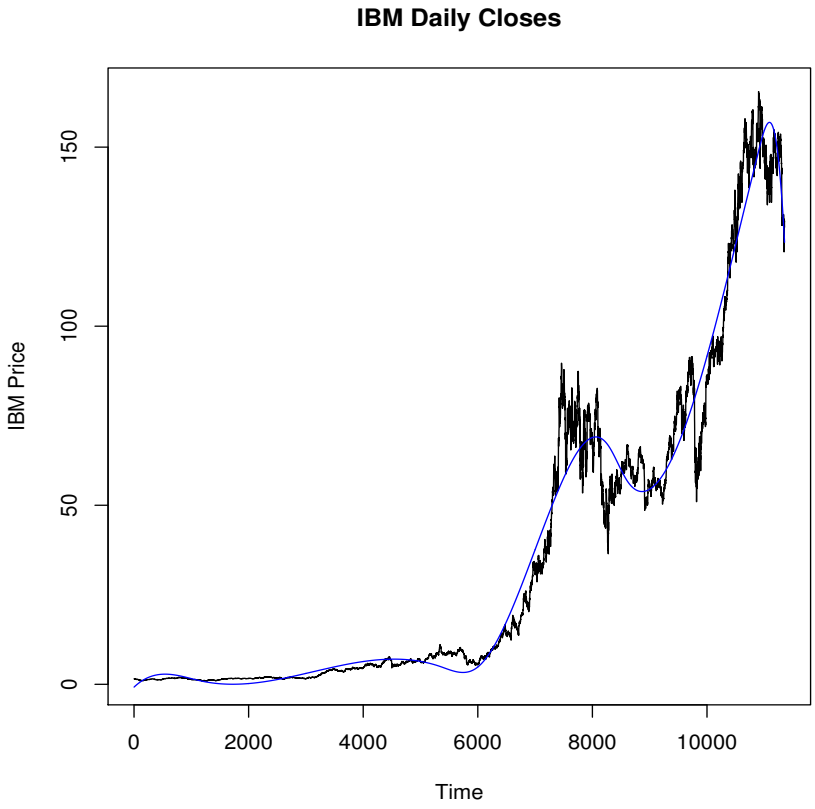


Figure 5.9. Cubic Spline Fit Using ATS-Identified Changepoints as Knots ***

6

Bayesian Analysis in R

intro... Kruschke (2015), *Doing Bayesian Data Analysis. A Tutorial with R, JAGS, and Stan*

6.1 Bayesian Analysis

In the Bayesian paradigm for data analysis, all variables in the model are random variables; some observable, and some latent. Model parameters are unobservable, but random, so statistical inferences concern the probability distributions of the parameters.

The model begins with a probability distribution that expresses beliefs about the properties of the random parameters, and a conditional probability distribution of the observables, given the parameters. The analysis involves forming joint and marginal distribution from those two distributions, so as to form a conditional probability distribution of the parameters, given the observations. (This process is outlined in Section **??** beginning on page **??**.) The observed data are then plugged into this conditional distribution to form the "posterior distribution", which is the basis for any inferences about the parameters.

A problem with performing a Bayesian analysis is the step in forming the marginal distribution of the observables, which may involve a rather complicated integration. Rather than forming this distribution analytically, the standard way of doing it is to use Monte Carlo simulation, in particular, to identify a Markov chain whose stationary distribution is the posterior distribution of interest and then simulate it. There are several ways this simulation can be performed. Techniques of this type are called Markov chain Monte Carlo (MCMC). A major problem with this procedure arises from the difficulty in determining when a stationary distribution has been identified. Much of the computational effort in using MCMC for Bayesian analysis is devoted to assessing the analysis *process*, rather than to the analysis itself. These issues are discussed further in Section **??** beginning on page **??**.

Another practical problem with performing a Bayesian analysis is the complexity of communicating with computer software to specify the model. One of the more successful software systems for Bayesian analysis was developed in the MRC Biostatistics Unit at the University of Cambridge. There are various versions of the software, all having a name that includes "BUGS" (Bayesian inference Using Gibbs Sampling), such as WinBUGS and OpenBUGS (see Lunn et al., 2012). A similar program, called JAGS ("Just Another Gibbs Sampler"), was developed by Martyn Plummer. It uses essentially the same model description language, but was developed independently of the BUGS project and is open source (available from SourceForge).

The BUGS and JAGS programs can be run as standalone systems, but for the power and flexibility for general data manipulations, the programs can be executed from within R. There are various R packages to facilitate this, but none are completely self-contained. A BUGS or JAGS executable must be available in a place where R can invoke it. The appropriate excecutable may be found by searching the internet. The program that I use is JAGS. There are various R packages to incorporate it into R; the one I use is `rjags`.

BUGS and JAGS share a clunky design that requires an external text file that specifies the model, which can be represented as a directed acyclic graph (DAG).

The BUGS/JAGS language is somewhat similar to the R language, but it has some functions with the same or similar names that have different meanings for the arguments. There is also a class of functions that *specify a distribution*. The names of these functions all begin with "`d`", and the other component is a mnemonic similar to the R functions for probability distributions (Table **??**). For example, in the BUGS/JAGS language, `dnorm` just means "the normal distribution", not the CDF of the normal distribution, as in R. BUGS/JAGS provides many common distributions. These distributions are used in the model statement to specify how a particular variable is distributed.

An important difference in the BUGS/JAGS functions that specify a distribution and the R functions that evaluate distributional functions is that arguments of some BUGS/JAGS functions may specify a *precision*, whereas the corresponding R functions specify a *standard deviation*. (The term "precision" is used in various ways in statistics, often in a nontechnical, general sense. It is quantified in two different ways in the literature. In one definition, it is the reciprocal of the standard deviation; in the other, it is the reciprocal of the variance. In most literature on Bayesian methods, including the BUGS/JAGS language, it is the reciprocal of the variance.)

$$\text{BUGS/JAGS} : \texttt{dnorm}(\texttt{mu}, \texttt{tau}) \quad \Leftrightarrow \quad \text{R} : \textit{pre}\texttt{norm}(\textit{arg}, \texttt{mu}, \texttt{1/sqrt}(\texttt{tau}))$$

The text file specifying the model is saved in an accessible directory with a filename extension of "`bug`".

After defining the model using the BUGS/JAGS language and storing the file in the R working directory, we can process it, and use it to analyze

data using R commands. The R function `jags.model` in the `rjags` package performs the usual BUGS/JAGS operations using the appropriate executable file. The function requires starting values, which can be obtained by ordinary R functions. The R function `coda.samples` in the `rjags` package organizes the output into an object of class `mcmc.list`, and the standard R function `summary` displays the output.

CODA ("Convergence Diagnostic and Output Analysis") is a software system that monitors and reports on the MCMC performance in the analysis. There are separate versions of CODA that are incorporated into JAGS and the various versions of BUGS. There are many types of diagnostic output about the *performance of the method of analysis*, which is not the same as the analysis itself.

6.2 MCMC Methods

6.3 R Packages

7

Linear Algebra and Linear Models with R

intro ...

assumed to know basic definitions

inner products, norms, metrics

The `matrixcalc` package contains many useful functions for working with matrices.

7.1 Matrix Transformations and Factorizations

the `pracma` package provides many of the rich set of Matlab matrix computations

7.2 Eigenanalysis

7.3 Linear Systems of Equations

7.4 Linear Regression Models

A simple example is linear regression.

Linear regression models are smoothers.

Given a set of observations on a number of possibly related variables, we may assume an asymmetric model of the relationships in which one variable, y, is linearly "dependent" on the other variables, $x_1, x_2, \ldots, x_m$:

$$y_i = \beta_0 + \beta_1 x_{1i} + \cdots + \beta_m x_{mi} + \epsilon_i, \tag{7.1}$$

where ϵ_i is a random "error" term accounting for the fact that the relation between y and $x_1, x_2, \ldots, x_m$ is not exact. When the model (7.1) is fit to data, that is, specific values of the βs, $\widehat{\beta}_0, \widehat{\beta}_0, \ldots, \widehat{\beta}_m$, are chosen or estimated, then the fitted ys, $\hat{y}_i$, are "smoother". The smoothed ys all on the hyperplane

$$y_i = \widehat{\beta}_0 + \widehat{\beta}_1 x_{1i} + \cdots + \widehat{\beta}_m x_{mi}. \tag{7.2}$$

*** missing value options ****promise

8

Optimization and Applications in R

intro ...

8.1 Finding Roots of Equations

8.2 Unconstrained Descent Methods

8.3 Unconstrained Combinatorial Methods

8.4 Constrained Optimization

9

Big Data and High Performance Computing in R

In this section we will briefly consider some of the issues in handling big data. There are two main considerations, speeding up the computations and computer memory usage. We not cover those methods in any detail; for more information we refer the interested reader to Kane, Emerson, and Weston (2013), or other sources on high-performance and distributed computing.

9.1 Long Vectors

Because the elements of arrays are accessed by integer indexes, the dimensions of arrays are generally limited by the maximum size of integers supported by the computing platform. Computing systems are characterized by their "word size", which is the number of bits used in the basic representation of numeric quantities. In some systems, called "32-bit systems", the number is 32. In "64-bit systems", which are now the most common ones. the number is 64. The word size places a limit on the magnitude and precision of the numbers that can be represented in the computer. In a 32-bit system, a scheme for all representing signed integers with any given magnitude up to some maximum, that maximum is obviously $2^{31} - 1$. Many 64-bit systems also lack support for integers larger than $2^{31} - 1$, although that limit is not due to the hardware.

Software systems such as R generally have to provide special support for arrays with dimensions greater than $2^{31} - 1$. R has limited support for such large arrays, called generically "long vectors".

9.2 Parallel Processing

Advances in the design and construction of the central processing unit steadily increase computational speed. Even greater increases in speed can be achieved by performing operations in parallel. Some tasks, such as data access and numerical computations, can easily be performed in parallel. It is because

the are very different kinds of operations. In a given task, some numerical computations can be performed in parallel because they are independent of each other. Even if a set of numerical computations are dependent on each other, there may be some subsets that can be performed in parallel. Cross validation and use of the bootstrap are two examples of statistical analyses that are computationally intensive, with substantial amounts of repetitive computations.

Most computers have multiple "cores" or processing units. These are generally just used on separate jobs or on different types of tasks in a single job. In order to make use of them for numerical computations in a single job the program itself must be structured to fork off separate sequences of computations. It is also often necessary for the user to provide some ad hoc directives.

An R package that provides a facility for parallel processing is the `foreach` package. The package includes a directive, `foreach`, that allows the user to specify that a function or a group of operations are to be performed in parallel over separate cores. An example of one form of the command is

```
foreach(i in I, j in J) %dopar% parfun(i,j,x)
```

where `parfun` is some function to be executed in parallel. The `foreach` function also provides options for combining the outputs of the function after execution.

The `foreach` directive requires a "parallel backend" to distribute the tasks over the separate processors. The parallel backend must be loaded and must be "registered", or otherwise all operations specified by `foreach` will be executed serially. The `doMC` package provides a parallel backend for the `foreach` package, and the `registerDoMC` function in the `doMC` package registers the backend and specifies the number of cores to be used, if they are available. An example of statements that must precede invocation of the `foreach` directive is shown below.

```
library(foreach)
library(doMC)
registerDoMC(4)
```

The standard processing unit in a computer is called a central processing unit (CPU) and may consist of one or a small number of cores. On the other hand, a *graphical processing unit* or GPU may contain thousands of cores. When GPUs were first introduced by Nvidia Corporation, the cores only performed very simple operations (essentially, just addition). Over time Nvidia has developed GPUs further to perform more general arithmetic operations, and has integrated GPUs with CPUs in a "compute unified device architecture", or CUDA. "CUDA" now refers to a programming interface at both a

low and a high level to allow programs to utilize an architecture that combines GPUs and CPUs. Some computations are very simple to set up and perform using CUDA, and many financial research firms make heavy use of GPUs, especially in simulations of trading scenarios. There are some R packages (for example, `gpuR`) designed to provide partial interfaces to CUDA, but at this time (2019), it is not straightforward to integrate any significant number of R facilities into a GPU environment. This will change, however.

9.3 Distributed Computing

Statistical analyses increasingly rely on data from disparate sources. In this book we refer to only a few standard sources, and the access to them is straightforward. Other financial analyses may involve data that are stored at multiple sites and that are being updated continually.

In the examples in this book, all processing of the data is done on a single machine (my computer), but in many applications, the processing is distributed over the locations where the data are stored. Tools, such as Hadoop and Spark, have been developed that facilitate use of R or other analysis programs on multiple datasets and on multiple processors. We refer the interested reader to a book on this subject such as Parsian (2015).

9.4 Use of Compiled Program Units

When a program executes slowly, it is often because only a few statements are requiring an inordinate amount of time. These statements may be in a loop. To improve the computational efficiency of a program, the first thing is to identify the statements in the program that are taking most of the time. How to do this profiling, of course, depends on the programming system. There is a good profiling system for R called `lineprof`. I will not discuss its use here, but rather refer the interested reader to Wickham (2019), Chapter 23.

R performs a computation by first *interpreting* the user's R statement requesting the computation, and then sending the actual instruction to do the computation to the computer. The interpretation step takes time, which can be eliminated by processing the user's initial request along with the user's other requested computations, and submitting the compiled requests as one task. The mechanism for doing this is the use of a compiled programming language such as Fortran or C.

Exactly how to link a program written in Fortran, C, or some other compiler language into R depends on the operating system and the compiler/linker for the other language. A very useful R package called `Rcpp`, developed by Dirk Eddelbuettel and others facilitates the linkage of C++ programs into R. In fact, using the `cppFunction` function in `Rcpp`, the C++ code can be written inline, and `cppFunction` does the compilation and linkage automatically. Only

a few things are necessary to do prior to using `Rcpp`. On Microsoft Windows systems, for example, the only thing required is to obtain `Rtoolsxx.exe` from CRAN and execute it. (Here, "`xx`" is the version number. Because `Rtools` interacts so intimately with the R system, the version number of `Rtools` must be compatible with the version of R itself.) `Rtools` includes a C++ compiler and other necessary components. Eddelbuettel (2013) provides a complete reference for the use of `Rcpp`.

Figure 9.1 shows an example of the use of `cppFunction` to form a function that does the same thing as the function in Figure 1.1 on page 17. The `Rcpp` package uses a class called `NumericVector` with an obvious meaning, and other classes for vectors with similar names.

```
library(Rcpp)
cppFunction("NumericVector myFunC(double x1,double y1, double x2,double y2,
            NumericVector x) {
// Given the points (x1,y1) and (x2,y2) and a set of abscissas x,
// determine the ordinates at x for the line that goes through all of them.
    int n = x.size();
    NumericVector y(n);
    double slope = (y2-y1)/(x2-x1);
    double intercept = y1 - slope*x1;
    for (int i = 0; i<n; ++i) {
         y[i] = slope*x[i] + intercept;
    }
    return(y);
}")
```

Figure 9.1. A Simple Function in C++; Compare Figure 1.1

Two things to remember when programming in C++: array indexes start at 0; statements are terminated by ";". (This is not rocket science, but it's one reason I use Fortran instead of C++ as my programming language. It may be why C++ programmers don't like Fortran.)

After the code in Figure 9.1 has been processed by R, the function `myFunC` is available for use. This function does the same thing as the function `myFun` written in R and shown in Figure **??**. After that function has been processed, it is also available for use, and Figure 9.2 shows both of these functions being executed. (In this case, these functions take the pair of points (0,1) and (5,11) and determine the straight line, $y = a+bx$, that goes through these two points. Then, at $x = 1, 2, 3$ the function determines the corresponding y values that are on the line. The other points on the line are (1,3), (2,5), and 3,7).)

```
> x1<-0; y1<-1; x2<-5; y2<-11
> x <- c(1, 2, 3)
> myFun(x1, y1, x2, y2, x)
[1] 3 5 7
> myFunC(x1, y1, x2, y2, x)
[1] 3 5 7
```

Figure 9.2. Execution of `myFun` (Figure 1.1) and `myFunC` (Figure 9.1)

Exercises: Incorporating C++ into R

Incorporating a C++ function into R.
Install `Rcpp` and `Rtools` or any other system software necessary on your system (or get somebody else to do it), and then write a C++ function to determine the sum of the elements in a numeric vector in R.

9.5 Memory Management

Data are stored in a computer's random-access memory (RAM) prior to and after performing computations on the data. The size of the RAM of course varies with the computer. If the amount of data in an analysis exceeds the size of the RAM, then the data must be swapped in and out of RAM from and to external memory, such as a "hard drive". The term "big data" is often used to refer to an amount of data that exceeds the size of the RAM on a given computer. In this sense, it depends on the available computing resources. "Big data" is also applied to an amount of data that requires an excessive amount of computational time. Obviously, these are rather vague meanings, but it is not important for me to try to be more precise. We will use the term in this general way.

None of the examples used in this book, such as daily stock prices even going back a century or more, would even be close to being considered big data. Every stock transaction of every publicly-traded stock over a period of a year or so does constitute big data.

To determine how much computer memory is being used by an R object, the R package `pryr` provides two useful functions, `object_size` and `mem_used` with obvious uses. Figure 9.3 illustrates their usage. (Note also the use of `ls` and `mget` from the `base` package.)

We note that the `xts` OHLC data frame `DJId`, which we have used in several examples, takes up only 439 kilobytes of memory, even though it contains data for 31 years. Each datum in an R object takes up a known amount of

```
> library(pryr)
> library(quantmod)
> DJId <- getSymbols("^DJI", env=NULL, from="1987-1-1",
+           to="2017-12-31", periodicity="daily", warnings=FALSE)
> object_size(DJId)
439 kB
> object_size(mget(ls())) #  the output depends on the current R session
3.64 MB
> mem_used()              #  the output depends on the current R session
48.4 MB
```

Figure 9.3. Functions in `pryr` for Memory Information

space, integers, 4 bytes; real numbers, 8 bytes; dates, 4 or 8 bytes; and so on. Knowing this, is is possible to determine approximately how much memory any R object will occupy. (I say "approximately" for two reasons; there is a small amount of slop associated with any object, and secondly, `object_size` does not return an exact amount.) In the case of the daily Dow Jones `xts` OHLC data frame there are 6 columns of real numbers taking up 8 bytes each and 1 column of labels taking up 8 bytes in this case. For 31 years with approximately 253 days per year, we have

$$253 \times 31 \times 8(6+1) = 439,208,$$

which is close to the reported value of 439 KB. (Note that `object_size` calls the units "kB".)

Occasionally, of course, we do encounter datasets that are too big to fit in memory. Two R packages that allow for data to be swapped in and out of memory are `bigmemory` and `ff`. The `ff` package produces an R object of class `ffdf`. A simple function that inputs an `ffdf` object, from a CSV file or other tabular format, is `read.table.ffdf`.

Another approach to handling large dataset is to use the database management system `MySQL`. The R package `RMySQL` provides a convenient interface for this. Database management systems typically do not store the data in memory.

Finally, I will mention a version of the `lm` function that processes the data in chunks. It is the `biglm` function in the `biglm` package. The methods for performing linear least squares computations have been a staple of statistical computing for years.

Solutions to Selected Exercises

Some exercises require use of a browser to navigate the web and perform certain activities within the browser. These are generally straightforward exercises, and no solutions are provided here. The general rule for solving problems of this type is to try things until the solution is found. This is the approach that is used by "tech savvy" children.

Exercises beginning on page 8

1.1.1a Using `help.search("principal components")`, we find `stats::prcomp` and `stats::princomp`, both in the basic `stats` package.

1.1.1b Using `help(prcomp)` and `help(princomp)`, we find that the main difference is that `prcomp` allows either raw data or a covariance/correlation matrix to be input and `prcomp` requires raw data. There are also several relatively minor differences, including for example, whether n or $n-1$ is used in the computation of a covariance matrix from the raw data input. (This does not change the computation of a correlation matrix.

1.1.2 The `%in%` operator is a binary logical operator that returns TRUE if the first operand is included at the primary level in the second operand; otherwise, it returns FALSE. A simple example is with `x` being a numeric variable with the value 3, and `d` being an atomic vector with value `(1,2,3)`. The expression `x%in%d` is TRUE. Likewise if e is the list `(1,"a",3,"b")` The expression `x%in%e` is TRUE.
More examples are seen in the following. The primary objects at the first level of the second operand object are the elements of the set.

```
> f <- list(1,c(2,3),list(1,"a"),"b")
> 1 %in% f
[1] TRUE
> 2 %in% f
[1] FALSE
> 5 %in% f
```

```
[1] FALSE
> "a" %in% f
[1] FALSE
> "b" %in% f
[1] TRUE
```

The components of the first operand object are treated as atomic elements to which the operator is applied sequentially.

```
> c(1,2) %in% f
[1]  TRUE FALSE
> c(2,3) %in% f
[1] FALSE FALSE
> "a" %in% f
[1] FALSE
> list(1,"a") %in% f
[1]  TRUE FALSE
> "b" %in% f
[1] TRUE
```

Exercises beginning on page 44

1.2.1a Let `i` be the index into `x`. The expression is

```
assign(paste("x",x[i],sep=""), x[i])
```

We can test it as follows.

```
> x <- c("d","c","b","a")
> for (i in 1:4) assign(paste("x",x[i],sep=""),x[i])
> xd
[1] "d"
> xc
[1] "c"
```

1.2.1b Let `i` be the index into `x`. The expression is

```
assign(paste("x",as.character(i),sep=""),x[i])
```

We can test it as follows.

```
> x <- c("d","c","b","a")
> for (i in 1:4) assign(paste("x",as.character(i),sep=""),x[i])
> x1
[1] "d"
> x2
[1] "c"
```

1.2.2a `2*(1:10)`
1.2.2b `2*(1:10)-1`
1.2.2c `(2*(1:10)-1)^2`

1.2.2d `sqrt(2*(1:10)-1)`

1.2.2e `1:40/2`

1.2.3

```
evens <- function(x1,x2)
   if (x1>x2) return(NULL)
   return(2*((ceiling(x1/2)):((floor(x2/2)))))
```

```
> x1 <- -2.5
> x2 <- 4.5
> evens(x1,x2)
[1] -2  0  2  4
> evens(x1+10,x2)
NULL
```

1.2.4a

```
> xf <- factor(c("d","c","b","a"))
> as.character(xf[3])
[1] "b"
```

1.2.4b

```
> yf <- factor(1:4)
> 7*as.numeric(yf[3])
[1] 21
```

1.2.5

```
> n <- 500
> xm <- 5.333333
> print(paste("The mean of ", as.character(n), " random numbers is ",
        as.character(round(xm,2)), sep=""))
[1] "The mean of 500 random numbers is 5.33"
```

1.2.6

1.2.7

```
if (is.na(x)) x <- 0
```

`is.na()` is TRUE for both NaNa ans NAs.

Exercises beginning on page 93

1.3.1

```
> x <- c(3,4,5,7,8,9,10)
> y <- x[x%%4==0]
> z <- x[x%%4!=0]
> y
[1] 4 8
> z
[1]  3  5  7  9 10
```

1.3.2

```
A <- matrix(c(3,2, 2,7), nrow=2, byrow=TRUE)
B <- matrix(c(1,2,3, 4,5,6), nrow=2, byrow=TRUE)
x <- c(2,4)
y <- c(-3,1)
z <- c(1,2,3)
```

```
> A*A
     [,1] [,2]
[1,]    9    4
[2,]    4   49
> A%*%A
     [,1] [,2]
[1,]   13   20
[2,]   20   53
> t(A)%*%A
     [,1] [,2]
[1,]   13   20
[2,]   20   53
> t(A)*A
     [,1] [,2]
[1,]    9    4
[2,]    4   49
> x*x
[1]  4 16
> t(x)%*%x
     [,1]
[1,]   20
> x%*%t(x)
     [,1] [,2]
[1,]    4    8
[2,]    8   16
> t(y)%*%x
     [,1]
[1,]   -2
> x%*%t(y)
     [,1] [,2]
[1,]   -6    2
[2,]  -12    4
> t(x)%*%A%*%y
     [,1]
[1,]  -10
> t(y)%*%A%*%x
     [,1]
[1,]  -10
```

```
> A%*%B
     [,1] [,2] [,3]
[1,]   11   16   21
[2,]   30   39   48
> t(B)%*%A
     [,1] [,2]
[1,]   11   30
[2,]   16   39
[3,]   21   48
```

1.3.3a

```
> set.seed(12345)
> prmon <- ts(round(rnorm(17),2)+100, start=c(2020,5), frequency=12)
> prhimon <- prmon+5
> prlomon <- prmon-5
> volmon <- ts(round(3*rnorm(17))+10000, start=c(2020,5), frequency=12)
> prmon
       Jan   Feb   Mar    Apr    May    Jun   Jul    Aug    Sep   Oct    Nov   Dec
2020                          100.59 100.71 99.89  99.55 100.61 98.18 100.63 99.72
2021 99.72 99.08 99.88 101.82 100.37 100.52 99.25 100.82  99.11
> prhimon
        Jan    Feb    Mar    Apr    May    Jun    Jul    Aug    Sep    Oct    Nov    Dec
2020                             105.59 105.71 104.89 104.55 105.61 103.18 105.63 104.72
2021 104.72 104.08 104.88 106.82 105.37 105.52 104.25 105.82 104.11
> prlomon
       Jan   Feb   Mar   Apr   May   Jun   Jul   Aug   Sep   Oct   Nov   Dec
2020                         95.59 95.71 94.89 94.55 95.61 93.18 95.63 94.72
2021 94.72 94.08 94.88 96.82 95.37 95.52 94.25 95.82 94.11
> volmon
       Jan   Feb   Mar   Apr   May   Jun   Jul   Aug   Sep   Oct   Nov   Dec
2020                         9999 10003 10001 10002 10004  9998  9995  9995
2021 10005  9999 10002 10002 10000 10002 10007 10006 10005
```

1.3.3b

```
> strtyr <- start(prmon)[1]
> strtmo <- start(prmon)[2]
> endyr <- end(prmon)[1]
> endmo <- end(prmon)[2]
> ind1 <- strtmo:(12*(endyr-strtyr)+endmo)
> ind <- ind1[ind1%%3==0]
> prqrt <- ts(prmon[ind+1-strtmo], start=c(strtyr,ceiling(strtmo/3)),
+ frequency=4)
> prqrt
       Qtr1   Qtr2   Qtr3   Qtr4
2020        100.71 100.61  99.72
2021  99.88 100.52  99.11
```

1.3.3c

```
> strtyr <- start(prhimon)[1]
> strtmo <- start(prhimon)[2]
> prhiqrt <- aggregate(window(prhimon, start=c(strtyr,3*ceiling(strtmo/3)+1)),
+         nfrequency=4, FUN=max)
```

```
> prhiqrt
       Qtr1   Qtr2   Qtr3   Qtr4
2020               105.61 105.63
2021 104.88 106.82 105.82
```

1.3.3d

```
> strtyr <- start(prlomon)[1]
> strtmo <- start(prlomon)[2]
> prloqrt <- aggregate(window(prlomon, start=c(strtyr,3*ceiling(strtmo/3)+1)),
+         nfrequency=4, FUN=min)
> prloqrt
      Qtr1  Qtr2  Qtr3  Qtr4
2020             94.55 93.18
2021 94.08 95.37 94.11
```

1.3.3e

```
> strtyr <- start(volmon)[1]
> strtmo <- start(volmon)[2]
> volqrt <- aggregate(window(volmon, start=c(strtyr,3*ceiling(strtmo/3)+1)),
+         nfrequency=4, FUN=sum)
> volqrt
      Qtr1  Qtr2  Qtr3  Qtr4
2020             30007 29988
2021 30006 30004 30018
```

1.3.3f

```
> ts.intersect(prqrt, prhiqrt, prloqrt, volqrt)
         prqrt prhiqrt prloqrt volqrt
2020 Q3 100.61  105.61   94.55  30007
2020 Q4  99.72  105.63   93.18  29988
2021 Q1  99.88  104.88   94.08  30006
2021 Q2 100.52  106.82   95.37  30004
2021 Q3  99.11  105.82   94.11  30018
```

1.3.4a

```
meanwts <- tapply(Xdf$weight, sizes, mean)
```

1.3.4b

Here, we assume that the index of the factor levels is specified as `i`.

```
print(paste("The mean weight of the ",
        as.character(levels(sizes)[i]), " items is ",
        as.character(round(as.numeric(meanwts[i]),2)), sep=""))
```

Here is a test.

```
> sizes <-factor(
    c("medium","large","small","medium","small","medium","large"))
> weight <- c(2,4,3,2,3,4,7)
> Xdf <- data.frame(sizes, weight)
> meanwts <- tapply(Xdf$weight, sizes, mean)
```

```
> #  print the mean corresponding to "medium",
> #   that is, the second level
> i <- 2
> print(paste("The mean weight of the ",
+         as.character(levels(sizes)[i]), " items is ",
+         as.character(round(as.numeric(meanwts[i]),2)), sep=""))
[1] "The mean weight of the medium items is 2.67"
```

1.3.5

```
Date <- as.Date(c(rep("2018-12-31",5),rep("2019-12-31",5)))
Sector   <- factor(rep(c("Tech", "Fin", "Fin", "Tech", "Tech"),2))
Stock   <- rep(c("AAPL", "BAC", "COF", "INTC", "MSFT"),2)
Price    <- c(157.74, 24.64, 75.59, 46.93, 101.57,
               293.68, 35.26, 103.06, 59.93, 157.77)
Stocks   <- data.frame(Date, Sector, Stock, Price)
Stocks
```

We get

```
         Date Sector Stock  Price
1  2018-12-31   Tech  AAPL 157.74
2  2018-12-31    Fin   BAC  46.93
3  2018-12-31    Fin   COF 101.57
4  2018-12-31   Tech  INTC  24.64
5  2018-12-31   Tech  MSFT  75.59
6  2019-12-31   Tech  AAPL 157.74
7  2019-12-31    Fin   BAC  46.93
8  2019-12-31    Fin   COF 101.57
9  2019-12-31   Tech  INTC  24.64
10 2019-12-31   Tech  MSFT  75.59
```

1.3.6

```
> splits <- split(Stocks,Stock)
> D1 <- splits[[1]][-c(2,3)]
> names(D1) <- c("Date", as.character(Stocks$Stock[1]))
> D2 <- splits[[2]][-c(2,3)]
> names(D2) <- c("Date", as.character(Stocks$Stock[2]))
> D3 <- splits[[3]][-c(2,3)]
> names(D3) <- c("Date", as.character(Stocks$Stock[3]))
> D4 <- splits[[4]][-c(2,3)]
> names(D4) <- c("Date", as.character(Stocks$Stock[4]))
> D5 <- splits[[5]][-c(2,3)]
> names(D5) <- c("Date", as.character(Stocks$Stock[5]))
>
> Prices   <- merge(merge(merge(merge(D1,D2,by="Date"),
+                  D3,by="Date"),D4,by="Date"),D5,by="Date")
> Prices
        Date   AAPL   BAC    COF  INTC   MSFT
1 2018-12-31 157.74 24.64  75.59 46.93 101.57
2 2019-12-31 293.68 35.26 103.06 59.93 157.77
```

1.3.7 We assume that there is a data frame named `Prices`, as constructed in Exercise 1.3.6.

```
> libray(xts)
> Pricesxts <- xts(Prices[, -1], order.by=Prices$Date)
> Pricesxts
             AAPL   BAC    COF  INTC   MSFT
2018-12-31 157.74 24.64  75.59 46.93 101.57
2019-12-31 293.68 35.26 103.06 59.93 157.77
```

1.3.8a

```
Dates <- as.Date(c("2020-05-31","2020-06-30","2020-07-31","2020-08-31",
                   "2020-09-30","2020-10-31","2020-11-30","2020-12-31",
                   "2021-01-31","2021-02-28","2021-03-31","2021-04-30",
                   "2021-05-31","2021-06-30","2021-07-31","2021-08-31",
                   "2021-09-30"))
for (i in 1:length(Dates))
      if (weekdays(Dates[i])=="Sunday") Dates[i] <- Dates[i] - 2
      if (weekdays(Dates[i])=="Saturday") Dates[i] <- Dates[i] - 1



> Dates
 [1] "2020-05-29" "2020-06-30" "2020-07-31" "2020-08-31" "2020-09-30" "2020-10-30"
 [7] "2020-11-30" "2020-12-31" "2021-01-29" "2021-02-26" "2021-03-31" "2021-04-30"
[13] "2021-05-31" "2021-06-30" "2021-07-30" "2021-08-31" "2021-09-30"
```

1.3.8b

```
set.seed(12345)
A.Open <- round(rnorm(17),2)+100
A.Close <- A.Open
A.High <- A.Close+5
A.Low <- A.Close-5
A.Volume <- round(3*rnorm(17))+10000
```

1.3.8c

```
> library(quantmod)
> Adata <- xts(cbind(A.Open, A.High, A.Low, A.Close, A.Volume),
+              order.by=Dates)
> Adata
           A.Open A.High A.Low A.Close A.Volume
2020-05-29 100.59 105.59 95.59  100.59     9999
2020-06-30 100.71 105.71 95.71  100.71    10003
2020-07-31  99.89 104.89 94.89   99.89    10001
2020-08-31  99.55 104.55 94.55   99.55    10002
2020-09-30 100.61 105.61 95.61  100.61    10004
2020-10-30  98.18 103.18 93.18   98.18     9998
2020-11-30 100.63 105.63 95.63  100.63     9995
2020-12-31  99.72 104.72 94.72   99.72     9995
2021-01-29  99.72 104.72 94.72   99.72    10005
2021-02-26  99.08 104.08 94.08   99.08     9999
2021-03-31  99.88 104.88 94.88   99.88    10002
2021-04-30 101.82 106.82 96.82  101.82    10002
2021-05-31 100.37 105.37 95.37  100.37    10000
```

```
2021-06-30 100.52 105.52 95.52  100.52    10002
2021-07-30  99.25 104.25 94.25   99.25    10007
2021-08-31 100.82 105.82 95.82  100.82    10006
2021-09-30  99.11 104.11 94.11   99.11    10005
```

1.3.8d

```
> to.period(Adata, "quarters")
        Adata.Open Adata.High Adata.Low Adata.Close Adata.Volume
2020-06-30  100.59     105.71     95.59      100.71        20002
2020-09-30   99.89     105.61     94.55      100.61        30007
2020-12-31   98.18     105.63     93.18       99.72        29988
2021-03-31   99.72     104.88     94.08       99.88        30006
2021-06-30  101.82     106.82     95.37      100.52        30004
2021-09-30   99.25     105.82     94.11       99.11        30018
```

1.3.8e

```
> Adata["2020"]
           A.Open A.High A.Low A.Close A.Volume
2020-05-29 100.59 105.59 95.59  100.59     9999
2020-06-30 100.71 105.71 95.71  100.71    10003
2020-07-31  99.89 104.89 94.89   99.89    10001
2020-08-31  99.55 104.55 94.55   99.55    10002
2020-09-30 100.61 105.61 95.61  100.61    10004
2020-10-30  98.18 103.18 93.18   98.18     9998
2020-11-30 100.63 105.63 95.63  100.63     9995
2020-12-31  99.72 104.72 94.72   99.72     9995
```

Exercises beginning on page 120

1.4.1 The actual values generated depend on the seed when the statements are executed.

```
> libray(e1071)
> x <- rnorm(100)
> skewness(x)
 [1] 0.09904714
> kurtosis(x)
 [1] -0.3086477
```

The kurtosis of the normal distribution is 3 and the excess kurtosis is 0. The `e1071::kurtosis` function computes the excess kurtosis, so the computed value is consistent with the population value.

The skewness of the normal distribution is 0, so the computed value is consistent with that value.

1.4.2 For the two-sided test: 2*pt(tcomp,49).

For the one-sided test: 1-pt(tcomp,49).

1.4.4a

```
BlackScholes <- function(type, S, K, r, Tt, sig)

  if(type=="C")
  d1 <- (log(S/K) + (r + sig^2/2)*Tt) / (sig*sqrt(Tt))
  d2 <- d1 - sig*sqrt(Tt)

  value <- S*pnorm(d1) - K*exp(-r*Tt)*pnorm(d2)
  return(value)

  if(type=="P")
  d1 <- (log(S/K) + (r + sig^2/2)*Tt) / (sig*sqrt(Tt))
  d2 <- d1 - sig*sqrt(Tt)

  value <-  (K*exp(-r*Tt)*pnorm(-d2) - S*pnorm(-d1))
  return(value)
```

1.4.4b

```
> S <- 63.49
> K <- 65
> r <- 0.02
> Tt <- 37/365
> sig <- 0.33
> BlackScholes(type="C", S, K, r, Tt, sig)
[1] 2.055642
```

The Black-Scholes price based on the assumed volatility is $2.06.

1.4.4c For call options, we first write a function as in the text for use in the `uniroot()` function.

```
fun <- function(x, S, K, r, Tt, Ct)
     d1 <- (log(S/K) + (r + x^2/2)*Tt) / (x*sqrt(Tt))
     d2 <- d1 - x*sqrt(Tt)
     value <- S*pnorm(d1) - K*exp(-r*Tt)*pnorm(d2) - Ct
     return(value)
```

1.4.4(c)i The stock price, the strike price, the risk-free rate, and the time to expiry are the same as in Exercise 1.4.4b. We initiate theobserved price of the call option, and then invoke `uniroot()`.

```
> Ct <- 2.00
> uniroot(fun,interval=c(0.1, 4.0),S,K,r,Tt,Ct=2)
$root
[1] 0.3230405
```

The implied volatility is 0.32.

1.4.4(c)ii

```
> S <- 63.49
> K <- 65
> r <- 0.02
```

```
> Tt <- 37/365
> Ct <- 2.00
```

1.4.4(c)iii

```
> S <- 63.49
> K <- 65
> r <- 0.02
> Tt <- 37/365
> Ct <- 2.00
```

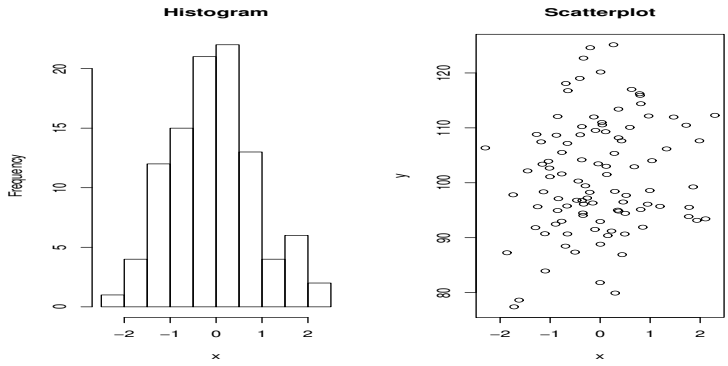
1.4.5 To generate 100 standard normal random numbers and plot a histogram:

```
x <- rnorm(100)
hist(x, main="Histogram")
```

To generate 100 normal random numbers and plot a histogram:

```
y <- rnorm(100, mean=100, sd=10)
plot(x, y, main="Scatterplot")
```

The plots are shown below.



1.4.6(a)i

```
> libray(pracma)
> factors(1000000)
 [1] 2 2 2 2 2 2 5 5 5 5 5 5
```

1.4.6(a)ii

```
> factors(999999)
 [1]  3  3  3  7 11 13 37
```

1.4.6(a)iii

```
> head(primes(1000000), n=6)
[1]  2  3  5  7 11 13
> tail(primes(1000000), n=6)
[1] 999931 999953 999959 999961 999979 999983
```

1.4.6(a)iv

```
> isprime(999957)
 [1]  0
```

999957 is not a prime

1.4.6(a)v

```
> isprime(999959)
[1] 1
```

1.4.6(a)vi

```
> isprime(999961)
[1] 1
```

999959 and 999961 are twin primes. How many twin primes are there?

1.4.6(a)vii

```
> sprintf("%.16e", fact(100))
[1] "9.3326215443944103e+157"
> sprintf("%.16e", factorial(100))
[1] "9.3326215443942249e+157"
```

1.4.6b

```
> bits(pi)
[1] "11.00100100001111110110101010001000100001011010001100000"
```

Exercises beginning on page 127

1.5.1 `y ~ x1+I(x1^2)+I(x1^3)`

1.5.2 `y ~ 0+x1+x2+x1*x2+I(x1^2)+I(x1^2)`

or

`y ~ 0+(x1+x2)^2+I(x1^2)+I(x1^2)`

Exercises beginning on page 137

1.6.1 The data frame created in Exercise 1.3.6 is called `Prices`.

```
setwd("c:/Books/Data/")
write.csv(Prices, "Prices.csv", row.names=FALSE)
```

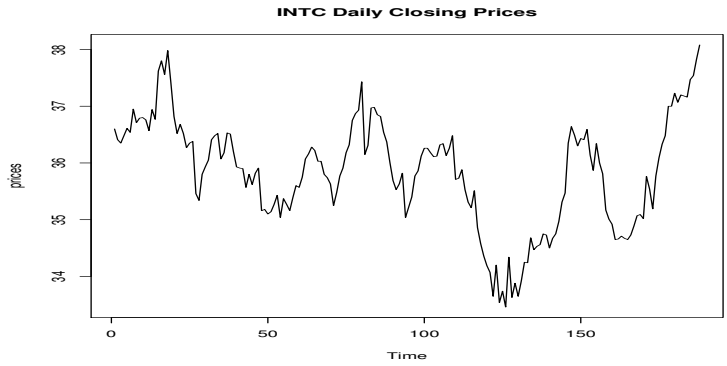
1.6.2 We create a data frame called `Pricesdf`. It is the same as `Prices`

```
> setwd("c:/Books/Data/")
> Pricesdf <- read.csv("Prices.csv")
> Pricesdf
        Date   AAPL   BAC    COF  INTC   MSFT
1 2018-12-31 157.74 24.64  75.59 46.93 101.57
2 2019-12-31 293.68 35.26 103.06 59.93 157.77
```

1.6.3a

```
library(quantmod)
z <- getSymbols("INTC", env=NULL, from="2017-1-1",
         to="2017-10-1", periodicity="daily")
prices <- as.numeric(z[,4])
plot.ts(prices, main="INTC Daily Closing Prices")
```

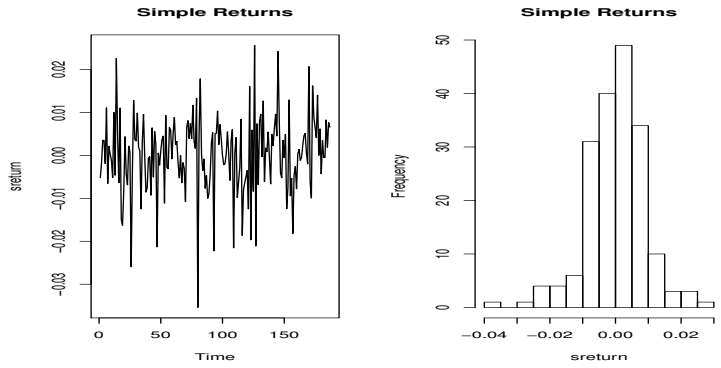
The plot is shown below.



1.6.3b Using the data from the previous exercise, we have

```
sreturn <- diff(prices)/prices[-1]
plot.ts(sreturn, main="Simple Returns")
hist(sreturn, main="Simple Returns")
```

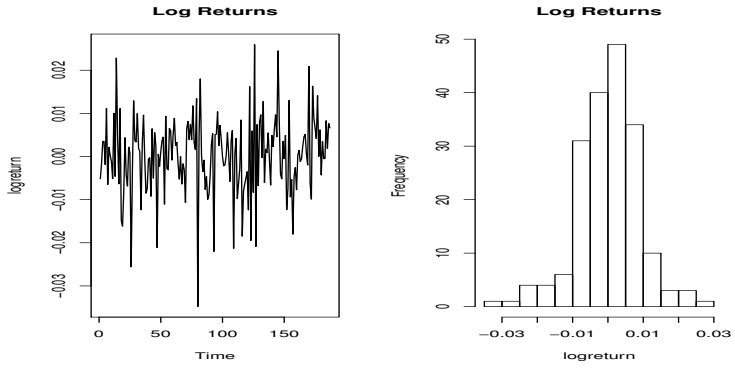
The plots are shown below.



1.6.3c

```
logreturn <- diff(log(prices))
plot.ts(logreturn, main="Log Returns")
hist(logreturn, main="Log Returns")
```

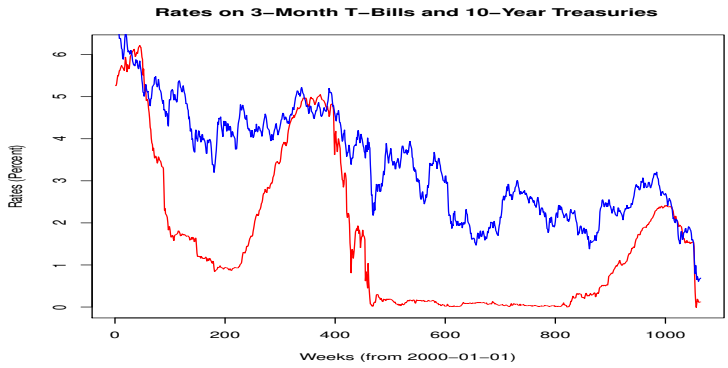
The plots are shown below.



1.6.4

```
xxx<-getSymbols("WTB3MS", src="FRED", env=NULL)
yyy<-getSymbols("WGS10YR", src="FRED", env=NULL)
plot.ts(as.numeric(xxx["2000/",1]), col="red", ylab="Rates (Percent)",
        xlab="Weeks (from 2000-01-01)",
        main="Rates on 3-Month T-Bills and 10-Year Treasuries")
lines(as.numeric(yyy["2000/",1]), col="blue")
xxx[xxx<0,1]
```

The plot is shown below.



The yield curve defined in terms of the 3-month and 10-year was briefly inverted at three times during this period.
The weekly rate on the 3-Month T-Bill was negative on one occasion.

References

Black, Fischer, and Myron Scholes (1972), The valuation of option contracts and a test of market efficiency, *Journal of Finance* **27**, 399–417.

Chambers, John M. (2008), *Software for Data Analysis. Programming with R*, Springer, New York.

Chambers, John M. (2016), *Extending R*, Chapman & Hall / CRC Press, Boca Raton.

Cotton, Richard (2017), *Testing R Code*, Chapman & Hall / CRC Press, Boca Raton.

Eddelbuettel, Dirk (2013), *Seamless R and C++ Integration with Rcpp*, Springer, New York.

Gentle, James E. (2020), *Statistical Analysis of Financial Data with Examples in R*, Chapman & Hall / CRC Press, Boca Raton.

Ghosh, Sucharita (2018), *Kernel Smoothing: Principles, Methods and Applications*, John Wiley & Sons, Hoboken.

Hirukawa, Masayuki (2018), *Asymmetric Kernel Smoothing: Theory and Applications in Economics and Finance*, Springer, Singapore.

Hull, John C. (2017) *Options, Futures, and Other Derivatives*, tenth edition, Pearson, New York.

Hyndman, Rob J., and Yeasmin Khandakar (2008), Automatic time series forecasting: The forecast package for R, *Journal of Statistical Software* **26**(3).

Kruschke, John K. (2015), *Doing Bayesian Data Analysis. A Tutorial with R, JAGS, and Stan*, second edition, Academic Press / Elsevier, Cambridge, Massachusetts.

Merton, Robert C. (1973), Theory of rational option pricing, *The Bell Journal of Economics and Management Science* **4**, 141–183.

Murrell, Paul (2018), *R Graphics*, third edition, Chapman & Hall / CRC Press, Boca Raton.

Nolan, Deborah, and Duncan Temple Lang (2014), *XML and Web Technologies for Data Science with R*, Springer, New York.

R Core Team (2020), *R: A language and environment for statistical computing*, R Foundation for Statistical Computing, Vienna, Austria.
URL `www.R-project.org/`.

Silverman, B. W. (1986), *Density Estimation for Statistics and Data Analysis*, Chapman & Hall / CRC Press, Boca Raton.

Wickham, Hadley (2014), Tidy data, *Journal of Statistical Software* **59**(10).

Wickham, Hadley (2016), *ggplot2. Elegant Graphics for Data Analysis*, second edition, Springer, New York.

Wickham, Hadley (2019), *Advanced R*, second edition, Chapman & Hall / CRC Press, Boca Raton.

Wilkinson, G. N., and C. E. Rogers (1973), Symbolic description of factorial models for analysis of variance, *Journal of the Royal Statistical Society. Series C (Applied Statistics)* **22**, 392–399.

Wand, M. P., and M. C. Jones (1994), *Kernel Smoothing*, Chapman & Hall / CRC Press, Boca Raton.

Zeileis, Achim, and Gabor Grothendieck (2005), zoo: S3 infrastructure for regular and irregular time series, *Journal of Statistical Software* **14**(6).

Index