Andrew O. Finley and Jeffrey W. Doser

Forestry 472: Ecological Monitoring and Data Analysis

Contents

List of Tables

List of Figures

Preface

This text is an introduction to data sciences for Forestry and Environmental students. Understanding and responding to current environmental challenges requires strong quantitative and analytical skills. There is a pressing need for professionals with data science expertise in this data rich era. The McKinsey Global Institute[1] predicts that "by 2018, the United States alone could face a shortage of 140,000 to 190,000 people with deep analytical skills as well as 1.5 million managers and analysts with the know-how to use the analysis of big data to make effective decisions". The Harvard Business Review dubbed *data scientist* "The Sexiest Job of the 21st Century"[2]. This need is not at all confined to the tech sector, as forestry professionals are increasingly asked to assume the role of *data scientists* and *data analysts* given the rapid accumulation and availability of environmental data (see, e.g. Schimel and Keller (2015)). Thomson Nguyen's talk[3] on the difference between a data scientist and a data analyst is very interesting and contains elements relevant to the aim of this text. This aim is to give you the opportunity to acquire the tools needed to become an environmental data analyst. Following Bravo et al. (2016) a *data analyst* has the ability to make appropriate calculations, convert data to graphical representation, interpret the information presented in graphical or mathematical forms, and make judgements or draw conclusions based on the quantitative analysis of data.

[1] http://www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation

[2] https://hbr.org/2012/10/data-scientist-the-sexiest-job-of-the-21st-century

[3] www.import.io/post/data-scientists-vs-data-analysts-why-the-distinction-matters

1

Data

1.1 FEF Tree Biomass Data Set

When thinking about data, we might initially have in mind a modest-sized and uncomplicated data set that serves a fairly specific purpose. For example, in forestry it is convenient to have a mathematical formula that relates a tree's diameter (or some other easily measured attribute) to stem or total biomass (i.e. we cannot directly measure tree biomass without destructive sampling). When coupled with forest inventory data, such formulas provide a means to estimate forest biomass across management units or entire forest landscapes. A data set used to create such formulas includes felled tree biomass by tree component for four hardwood species of the central Appalachians sampled on the Fernow Experimental Forest[1] (FEF), West Virginia Wood et al. (2016). A total of 88 trees were sampled from plots within two different watersheds on the FEF. Hardwood species sampled include *Acer rubrum*, *Betula lenta*, *Liriodendron tulipifera*, and *Prunus serotina*, all of which were measured in the summer of 1991 and 1992. Data include tree height, diameter, as well as green and dry weight of tree stem, top, small branches, large branches, and leaves. Table 1.1 shows a subset of these data

The size of this dataset is relatively small, there are no missing observations, the variables are easily understood, etc.

1.2 FACE Experiment Data Set

We often encounter data gleaned from highly structured and complex experiments. Such data typically present challenges in organization/storage, exploratory data analysis (EDA), statistical analysis, and interpretation of analysis results. An example data set comes from the Aspen Free-Air Carbon Diox-

[1] http://www.nrs.fs.fed.us/ef/locations/wv/fernow

TABLE 1.1: A subset of the tree biomass data from the FEF.

species	dbh_in	height_ft	stem_green_kg	leaves_green_kg
Acer rubrum	6.0	48.0	92.2	16.1
Acer rubrum	6.9	48.0	102.3	12.9
Acer rubrum	6.4	48.0	124.4	16.5
Acer rubrum	6.5	49.0	91.7	12.0
Acer rubrum	7.2	51.0	186.2	22.4
Acer rubrum	3.1	40.0	20.8	0.9
Acer rubrum	2.0	30.5	5.6	1.0
Acer rubrum	4.1	50.0	54.1	6.1
Acer rubrum	2.4	28.0	10.2	2.5
Acer rubrum	2.7	40.4	20.2	1.6

ide Enrichment[2] (FACE) Experiment conducted from 1997-2009 on the Harshaw Experimental Forest[3] near Rhinelander, Wisconsin. The Aspen FACE Experiment was a multidisciplinary study that assessed the effects of increasing tropospheric ozone and carbon dioxide concentrations on the structure and functioning of northern forest ecosystems. The design provided the ability to assess the effects of these gasses alone (and in combination) on many ecosystem attributes, including growth, leaf development, root characteristics, and soil carbon. The data set considered here comprises annual tree height and diameter measurements from 1997 to 2008 for *Populus tremuloides*, *Acer saccharum*, and *Betula papyrifera* grown within twelve 30 meter diameter rings in which the concentrations of tropospheric ozone and carbon dioxide were controlled Kubiske (2013). Because there was no confinement, there was no significant change in the natural, ambient environment other than elevating these trace gas concentrations. Although the basic individual tree measurements are similar to those in the FEF data set we saw in Section 1.1, (i.e., height and diameter), the study design specifies various tree species clones, varying gas treatments, and treatment replicates. Further, because these are longitudinal data, (measurements were recorded over time) the data set presents many missing values as a result of tree mortality. Table 1.2 contains the first five records as well as 5 more randomly selected records in the data set. Here, a row identifies each tree's experimental assignment, genetic description, and growth over time.

Notice that several height measurements in 2008 contain missing data. If all year measurements were shown, we would see much more missing data. Also, notice that this data set is substantially larger than the FEF data set with 912 rows and 39 columns of data in the full data set.

[2] http://www.nrs.fs.fed.us/disturbance/climate_change/face
[3] http://www.nrs.fs.fed.us/ef/locations/wi/rhinelander/

TABLE 1.2: A small portion of the FACE experiment data set

	Rep	Treat	SPP	Clone	ID..	X1997_Height	X2008_Height
1	1	1	B	B	4360	51.0	632
2	1	1	A	216	4359	58.0	742
3	1	1	B	B	4358	24.0	916
4	1	1	A	216	4357	58.0	981
5	1	1	B	B	4356	41.0	914
183	1	3	B	B	5017	40.0	NA
625	3	1	B	B	6853	55.5	936
835	3	3	B	B	7573	48.0	NA
259	1	4	B	B	5327	48.0	659
96	1	2	A	216	4697	27.0	NA

1.3 PEF Inventory and LiDAR Data Set

Coupling forest inventory with remotely sensed Light Detection and Ranging (LiDAR) data sets using regression models offers an attractive approach to mapping forest variables at stand, regional, continental, and global scales. LiDAR data have shown great potential for use in estimating spatially explicit forest variables over a range of geographic scales (Asner et al., 2009), (Babcock et al., 2013), (Finley et al., 2011), (Næsset, 2011), (Neigh et al., 2013). Encouraging results from these and many other studies have spurred massive investment in new LiDAR sensors, sensor platforms, as well as extensive campaigns to collect field-based calibration data.

Much of the interest in LiDAR based forest variable mapping is to support carbon monitoring, reporting, and verification (MRV) systems, such as defined by the United Nations Programme on Reducing Emissions from Deforestation and Forest Degradation[4] (UN-REDD) and NASA's Carbon Monitoring System[5] (CMS) (Le Toan et al., 2011), (Ometto et al., 2014). In these, and similar initiatives, AGB is the forest variable of interest because it provides a nearly direct measure of forest carbon (i.e., carbon comprises $\sim$ 50% of wood biomass, West (2004)). Most efforts to quantify and/or manage forest ecosystem services (e.g., carbon, biodiversity, water) seek high spatial resolution wall-to-wall data products such as gridded maps with associated measures of uncertainty, e.g., point and associated credible intervals (CIs) at the pixel level. In fact several high profile international initiatives include language con-

[4] http://www.un-redd.org
[5] http://carbon.nasa.gov

TABLE 1.3: A small portion of the PEF inventory data set

	MU	plot	easting	northing	biomass.mg.ha	stocking.stems.ha
118	17	24	530304	4965983	112.96	2325
403	31	11	530575	4964959	NA	NA
539	8	23	530004	4967094	71.05	10927
167	19	63	530436	4965217	NA	NA
62	14	21	530218	4966445	NA	NA
410	31	32	530657	4964999	NA	NA
308	27	31	530449	4965815	134.35	7872
471	6	13	529560	4967220	30.33	3016
556	9	14	529601	4966363	140.00	1284
65	14	24	530339	4966652	163.52	433

cerning the level of spatially explicit acceptable error in total forest carbon estimates, see, e.g., UN-REDD (2009) and UNFCCC (2015).

Here, we consider a data set collected on the Penobscot Experimental Forest[6] (PEF) in Bradley and Eddington, Maine. The dataset comprises LiDAR waveforms collected with the Laser Vegetation Imaging Sensor[7] (LVIS) and several forest variables measured on a set of 589 georeferenced forest inventory plots. The LVIS data were acquired during the summer of 2003. The LVIS instrument, an airborne scanning LiDAR with a 1064 nm laser, provided 12,414 LiDAR pseudo-waveform signals within the PEF. For each waveform, elevations were converted to height above the ground surface and interpolated at 0.3 m intervals. Figure 1.1 shows PEF LiDAR energy returns at 12 m above the ground, forest inventory plot locations, and management unit boundaries. The forest inventory data associated with each plot were drawn from the PEF's database of several on-going, long-term silvicultural experiments (see Kenefic et al. (2015)). Below we provide a plot containng the geographic coordinates, biomass (mg/ha), basal area (m^2/ha), stocking (trees/ha), diameter class (cm), and management unit. Table 1.3 shows a subset of data for 10 randomly selected plots (where each row records plot measurements) in the forest inventory data set.

[6] www.fs.fed.us/ne/durham/4155/penobsco.htm
[7] http://lvis.gsfc.nasa.gov

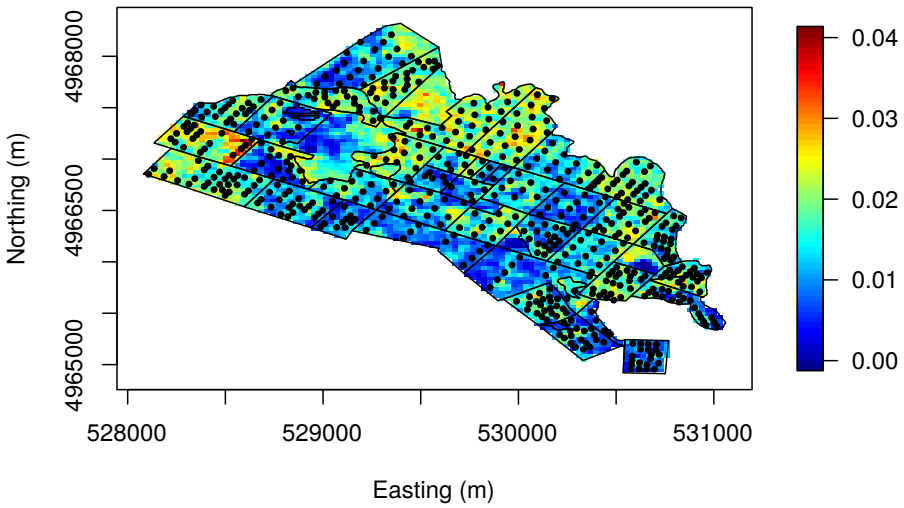


FIGURE 1.1: Surface of LiDAR energy returns at 12 m above the ground, forest inventory plot locations, and management unit boundaries on the PEF.

1.4 Zurichberg Forest inventory data set

Measuring tree diameter and height is a time consuming process. This fact makes the Zurichberg Forest inventory data set a rare and impressive investment. These data comprise a complete enumeration of the 589 trees in the Zurichberg Forest, including species, diameter at breast height, basal area, and volume. The stem map colored by species is shown in Figure 1.2.

1.5 Looking Forward

The four examples above illustrate a variety of data sets that might be encountered in practice, and each provides its own challenges. For the FACE data, the challenges are more statistical in nature. Complications could arise related to the complex study design and how that design might affect methods of analysis and conclusions drawn from the study. The other data sets present different challenges, such as how to:

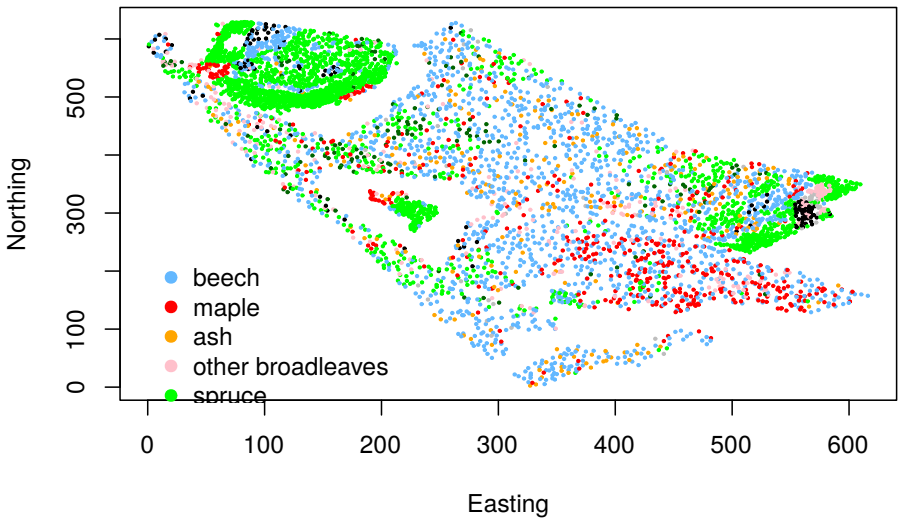


FIGURE 1.2: Location and species of all trees in the Zurichberg Forest.

1. Develop biomass equations suitable for population inference from the FEF's small sample of 88 trees
2. Work with spatially indexed data in the case of the PEF and Zurichberg inventory data
3. Effectively and efficiently process the PEF's high-dimentional LiDAR signal data for use in predictive models of forest variables.

This book and associated material introduce tools to tackle some of the challenges in working with real data sets within the context of the R statistical system. We will focus on important topics such as

- Obtaining and manipulating data
- Summarizing and visualizing data
- Communicating findings about data that support reproducible research
- Programming and writing functions
- Working with specialized data structures, e.g., spatial data and databases

1.6 How to Learn (The Most Important Section in This Book!)

There are several ways to engage with the content of this book and associated materials.

One way is not to engage at all. Leave the book closed on a shelf and do something else with your time. That may or may not be a good life strategy, depending on what else you do with your time, but you won't learn much from the book!

Another way to engage is to read through the book "passively", reading all that's written but not reading the book with R open on your computer, where you could enter the R commands from the book. With this strategy you'll probably learn more than if you leave the book closed on a shelf, but there are better options.

A third way to engage is to read the book while you're at a computer with R, and to enter the R commands from the book as you read about them. You'll likely learn more this way.

A fourth strategy is even better. In addition to reading and entering the commands given in the book, you think about what you're doing, and ask yourself questions (which you then go on to answer). For example, after working through some R code computing the logarithm of positive numbers you might ask yourself, "What would R do if I asked it to calculate the logarithm of a negative number? What would R do if I asked it to calculate the logarithm of a really large number such as one trillion?" You could explore these questions easily by just trying things out in the R Console window.

If your goal is to maximize the time you have to binge-watch *Stranger Things* Season 2 on Netflix, the first strategy may be optimal. But if your goal is to learn a lot about computational tools for data science, the fourth strategy is probably going to be best.

2

Introduction to R and RStudio

Various statistical and programming software environments are used in data science, including R, Python, SAS, C++, SPSS, and many others. Each has strengths and weaknesses, and often two or more are used in a single project. This book focuses on R for several reasons:

1. R is free
2. It is one of, if not the, most widely used software environments in data science
3. R is under constant and open development by a diverse and expert core group
4. It has an incredible variety of contributed packages
5. A new user can (relatively) quickly gain enough skills to obtain, manage, and analyze data in R

Several enhanced interfaces for R have been developed. Generally such interfaces are referred to as *integrated development environments (IDE)*. These interfaces are used to facilitate software development. At minimum, an IDE typically consists of a source code editor and build automation tools. We will use the RStudio IDE, which according to its developers "is a powerful productive user interface for R.[1] RStudio is widely used, it is used increasingly in the R community, and it makes learning to use R a bit simpler. Although we will use RStudio, most of what is presented in this book can be accomplished in R (without an added interface) with few or no changes.

2.1 Obtaining and Installing R

It is simple to install R on computers running Microsoft Windows, macOS, or Linux. For other operating systems users can compile the source code directly.[2]

[1] http://www.rstudio.com/

[2] Windows, macOS, and Linux users also can compile the source code directly, but for most it is a better idea to install R from already compiled binary distributions.

Here is a step-by-step guide to installing R for Microsoft Windows.[3] macOS and Linux users would follow similar steps.

1. Go to http://www.r-project.org/
2. Click on the CRAN link on the left side of the page
3. Choose one of the mirrors.[4]
4. Click on Download R for Windows
5. Click on base
6. Click on Download R 3.5.0 for Windows
7. Install R as you would install any other Windows program

2.2 Obtaining and Installing RStudio

You must install R prior to installing RStudio. RStudio is also simple to install:

1. Go to http://www.rstudio.com
2. Click on the link RStudio under the Products tab, then select the Desktop option
3. Click on the Desktop link
4. Choose the DOWNLOAD RSTUDIO DESKTOP link in the Open Source Edition column
5. On the ensuing page, click on the Installer version for your operating system, and once downloaded, install as you would any program

2.3 Using R and RStudio

Start RStudio as you would any other program in your operating system. For example, under Microsoft Windows use the Start Menu or double click on the shortcut on the desktop (if a shortcut was created in the installation process). A (rather small) view of RStudio is displayed in Figure 2.1.

Initially the RStudio window contains three smaller windows. For now our main focus will be the large window on the left, the Console window, in which

[3] New versions of R are released regularly, so the version number in Step 6 might be different from what is listed below.

[4] The http://cran.rstudio.com/ mirror is usually fast. Otherwise choose a mirror in Michigan.

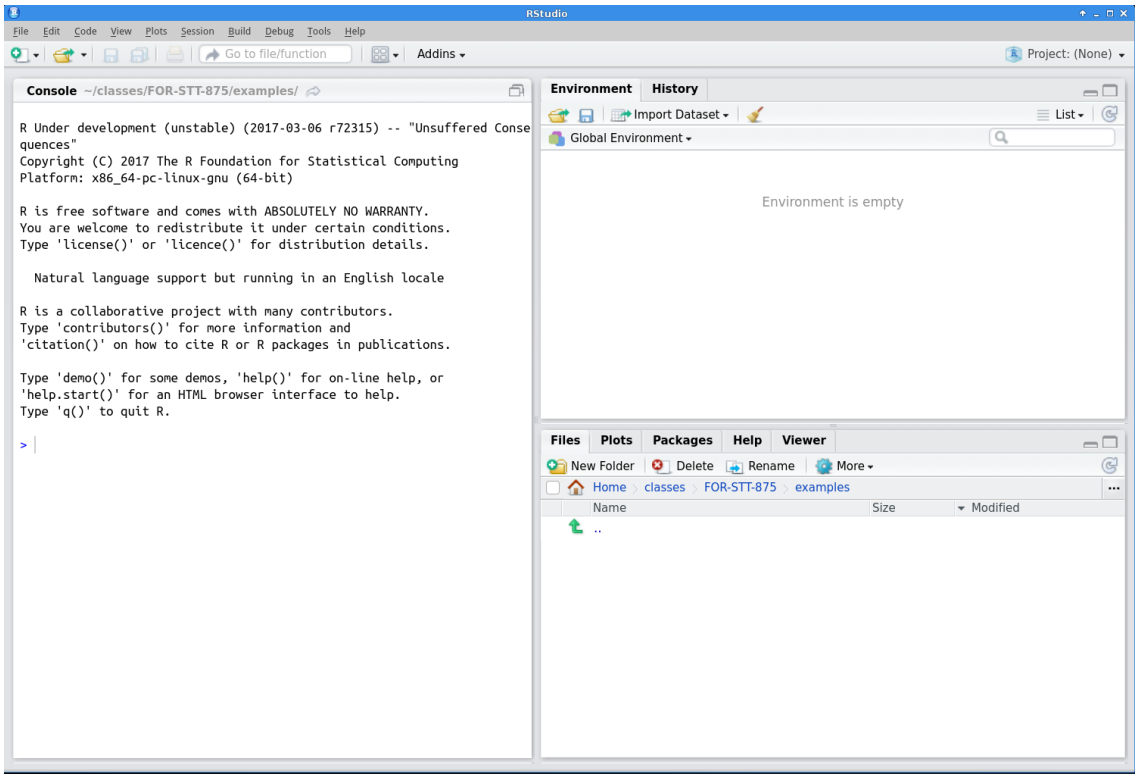


FIGURE 2.1: The Rstudio IDE.

R statements are typed. The next few sections give simple examples of the use of R. In these sections we will focus on small and non-complex data sets, but of course later in the book we will work with much larger and more complex sets of data. Read these sections at your computer with R running, and enter the R commands there to get comfortable using the R console window and RStudio.

2.3.1 R as a calculator

R can be used as a calculator. Note that `#` is the comment character in R, so R ignores everything following this character. Also, you will see that R prints `[1]` before the results of each command. Soon we will explain its relevance, but ignore this for now. The command prompt in R is the greater than sign `>`.

```
> 34 + 20 * sqrt(100)  ## +,-,*,/ have the expected meanings
```

```
[1] 234
```

```
> exp(2)  ##The exponential function
```

```
[1] 7.389
```

```
> log10(100)  ##Base 10 logarithm
```

```
[1] 2
```

```
> log(100)  ##Base e logarithm
```

```
[1] 4.605
```

```
> 10^log10(55)
```

```
[1] 55
```

Most functions in R can be applied to vector arguments rather than operating on a single argument at a time. A *vector* is a data structure that contains elements of the same data type (i.e. integers).

```
> 1:25  ##The integers from 1 to 25
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
[18] 18 19 20 21 22 23 24 25
```

```
> log(1:25)  ##The base e logarithm of these integers
```

```
 [1] 0.0000 0.6931 1.0986 1.3863 1.6094 1.7918 1.9459
 [8] 2.0794 2.1972 2.3026 2.3979 2.4849 2.5649 2.6391
[15] 2.7081 2.7726 2.8332 2.8904 2.9444 2.9957 3.0445
[22] 3.0910 3.1355 3.1781 3.2189
```

```
> 1:25 * 1:25  ##What will this produce?
```

```
 [1]   1   4   9  16  25  36  49  64  81 100 121 144
[13] 169 196 225 256 289 324 361 400 441 484 529 576
[25] 625
```

```
> 1:25 * 1:5  ##What about this?
```

```
 [1]   1   4   9  16  25   6  14  24  36  50  11  24
[13]  39  56  75  16  34  54  76 100  21  44  69  96
[25] 125
```

```
> seq(from = 0, to = 1, by = 0.1)  ##A sequence of numbers from 0 to 1
```

```
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```
> exp(seq(from = 0, to = 1, by = 0.1))  ##What will this produce?
```

```
 [1] 1.000 1.105 1.221 1.350 1.492 1.649 1.822 2.014
 [9] 2.226 2.460 2.718
```

Now the mysterious square bracketed numbers appearing next to the output make sense. R puts the position of the beginning value on a line in square brackets before the line of output. For example if the output has 40 values, and 15 values appear on each line, then the first line will have `[1]` at the left, the second line will have `[16]` to the left, and the third line will have `[31]` to the left.

2.3.2 Basic descriptive statistics and graphics in R

Of course it is easy to compute basic descriptive statistics and to produce standard graphical representations of data. For illustration consider the first 14 observations of tree height and DBH (diameter at breast height) from the FEF data set. We will begin by entering these data "by hand" using the `c()` function, which concatenates its arguments into a vector. For larger data sets we will clearly want an alternative way to enter data.

A style note: R has two widely used methods of assignment: the left arrow, which consists of a less than sign followed immediately by a dash: `<-` and the equals sign: `=`. Much ink has been used debating the relative merits of the two methods, and their subtle differences. Many leading R style guides (e.g., the Google style guide at https://google.github.io/styleguide/Rguide.xml and the Bioconductor style guide at http://www.bioconductor.org/developers/how-to/coding-style/) recommend the left arrow `<-` as an assignment operator, and we will use this throughout the book.

Also you will see that if a command has not been completed but the ENTER key is pressed, the command prompt changes to a `+` sign.

```
> dbh <- c(6, 6.9, 6.4, 6.5, 7.2, 3.1, 2, 4.1, 2.4, 2.7, 3.7,
+   6.3, 5.2, 5.1, 6.4)
> ht <- c(48, 48, 48, 49, 51, 40, 30.5, 50, 28, 40.4, 42.6,
+   53, 55, 50, 50)
> dbh
```

```
 [1] 6.0 6.9 6.4 6.5 7.2 3.1 2.0 4.1 2.4 2.7 3.7 6.3
[13] 5.2 5.1 6.4
```

```
> ht
```

```
 [1] 48.0 48.0 48.0 49.0 51.0 40.0 30.5 50.0 28.0 40.4
[11] 42.6 53.0 55.0 50.0 50.0
```

Next we compute some descriptive statistics for the two numeric variables

```
> mean(dbh)
```

```
[1] 4.933
```

```
> sd(dbh)
```

```
[1] 1.782
```

```
> summary(dbh)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   2.00    3.40    5.20    4.93    6.40    7.20
```

```
> mean(ht)
```

```
[1] 45.57
```

```
> sd(ht)
```

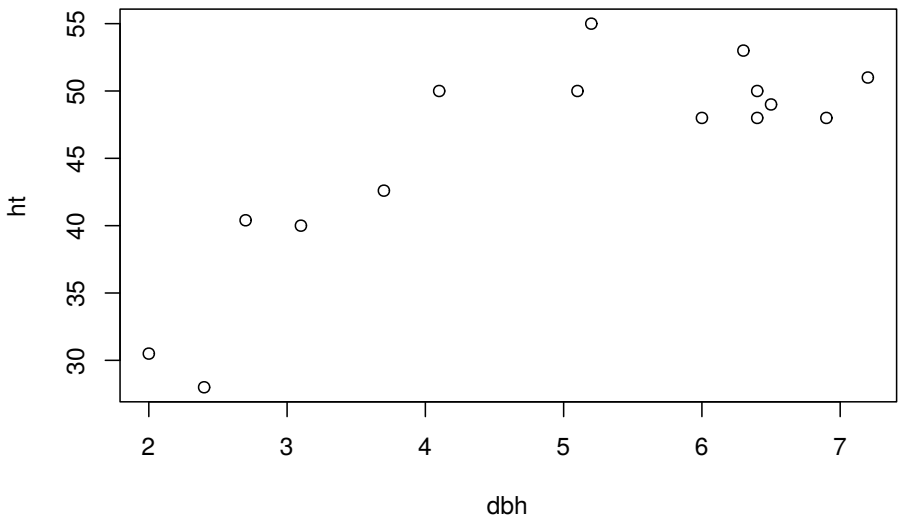
```
[1] 7.857
```

```
> summary(ht)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   28.0    41.5    48.0    45.6    50.0    55.0
```

Next, a scatter plot of `dbh` versus `ht`:

```
> plot(dbh, ht)
```



Unsurprisingly as DBH increases, height tends to increase. We'll investigate this further using simple linear regression in the next section.

2.3.3 Simple linear regression in R

The `lm()` function is used to fit linear models in R, including simple linear regression models. Here it is applied to the DBH height data.

```
> ht.lm <- lm(ht ~ dbh)  ##Fit the model and save it in ht.lm
> summary(ht.lm)  ##Basic summary of the model


Call:
lm(formula = ht ~ dbh)

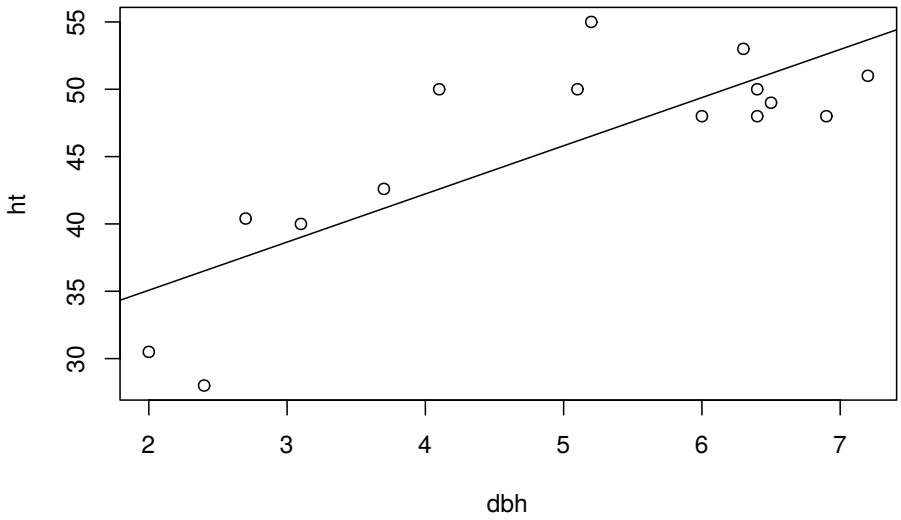
Residuals:
   Min     1Q Median     3Q    Max
-8.507 -2.742 -0.812  2.683  8.480

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
```

```
(Intercept)   27.925      3.739    7.47  4.7e-06 ***
dbh            3.576      0.716    5.00  0.00024 ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4.77 on 13 degrees of freedom
Multiple R-squared:  0.658, Adjusted R-squared:  0.631
F-statistic:   25 on 1 and 13 DF,  p-value: 0.000244
```

```
> plot(dbh, ht)  ##Scatter plot of the data
> abline(ht.lm)  ##Add the fitted regression line to the plot
```



We will work extensively with such models later in the text. We will also talk about why it might not be a good idea to assume a linear relationship between DBH and height—can you guess why this is by looking at the data scatter and model fitted line in the plot above?

2.4 How to Learn

There are several ways to engage with the content of this book and associated learning materials.

A comprehensive, but slightly overwhelming, cheatsheet for RStudio is available here https://www.rstudio.com/wp-content/uploads/2016/01/rstudio-IDE-cheatsheet.pdf. As we progress in learning R and RStudio, this cheatsheet will become more useful. For now you might use the cheatsheet to locate the various windows and functions identified in the coming chapters.

2.5 Getting help

There are several free (and several not free) ways to get R help when needed.

Several help-related functions are built into R. If there's a particular R function of interest, such as `log`, `help(log)` or `?log` will bring up a help page for that function. In RStudio the help page is displayed, by default, in the `Help` tab in the lower right window.[5] The function `help.start` opens a window which allows browsing of the online documentation included with R. To use this, type `help.start()` in the console window.[6] The `help.start` function also provides several manuals online and can be a useful interface in addition to the built in help.

Search engines provide another, sometimes more user-friendly, way to receive answers for R questions. A Google search often quickly finds something written by another user who had the same (or a similar) question, or an online tutorial that touches on the question. More specialized is https://rseek.org/, which is a search engine focused specifically on R. Both Google and https://rseek.org are valuable tools, often providing more user-friendly information then R's own help system.

In addition, R users have written many types of contributed documentation. Some of this documentation is available at http://cran.r-project.org/other-docs.html. Of course there are also numerous books covering general and specialized R topics available for purchase.

[5] There are ways to change this default behavior.

[6] You may wonder about the parentheses after `help.start`. A user can specify arguments to any R function inside parentheses. For example `log(10)` asks R to return the logarithm of the argument 10. Even if no arguments are needed, R requires empty parentheses at the end of any function name. In fact if you just type the function name without parentheses, R returns the definition of the function. For simple functions this can be illuminating.

2.6 Workspace, working directory, and keeping organized

The *workspace* is your R session working environment and includes any objects you create. Recall these objects are listed in the `Global Environment` window. The command `ls()`, which stands for list, will also list all the objects in your workspace (note, this is the same list that is given in the `Global Environment` window). When you close RStudio, a dialog box will ask you if you want to save an image of the current workspace. If you choose to save your workspace, RStudio saves your session objects and information in a `.RData` file (the period makes it a hidden file) in your *working directory*. Next time you start R or RStudio it checks if there is a `.RData` in the working directory, loads it if it exists, and your session continues where you left off. Otherwise R starts with an empty workspace. This leads to the next question—what is a working directory?

Each R session is associated with a working directory. This is just a directory from which R reads and writes files, e.g., the `.RData` file, data files you want to analyze, or files you want to save. On Mac when you start RStudio it sets the working directory to your home directory (for me that's `/Users/andy`). If you're on a different operating system, you can check where the default working directory is by typing `getwd()` in the console. You can change the default working directory under RStudio's `Global Option` dialog found under the `Tools` dropdown menu. There are multiple ways to change the working directory once an R session is started in RStudio. One method is to click on the `Files` tab in the lower right window and then click the `More` button. Alternatively, you can set the session's working directory using the `setwd()` in the console. For example, on Windows `setwd("C:/Users/andy/for472/exercise1")` will set the working directory to `C:/Users/andy/for472/exercise1`, assuming that file path and directory exist (Note: Windows file path uses a backslash, `\`, but in R the backslash is an escape character, hence specifying file paths in R on Windows uses the forward slash, i.e., `/`). Similarly on Mac you can use `setwd("/Users/andy/for472/exercise1")`. Perhaps the most simple method is to click on the `Session` tab at the top of your screen and click on the `Set Working Directory` option. Later on when we start reading and writing data from our R session, it will be very important that you are able to identify your current working directory and change it if needed. We will revisit this in subsequent chapters.

As with all work, keeping organized is the key to efficiency. It is good practice to have a dedicated directory for each R project or exercise.

2.7 Quality of R code

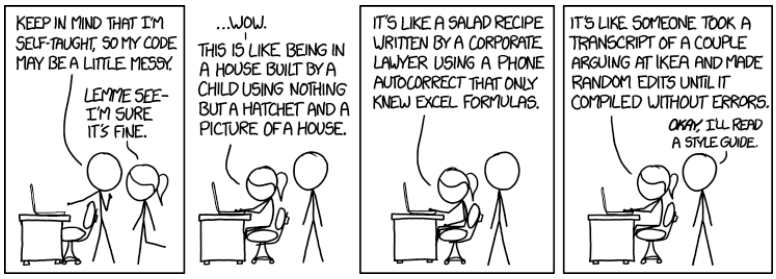


FIGURE 2.2: xkcd: Code Quality

Writing well-organized and well-labeled code allows your code to be more easily read and understood by another person. (See xkcd's take on code quality in Figure **??**.) More importantly, though, your well-written code is more accessible to you hours, days, or even months later. We are hoping that you can use the code you write in this class in future projects and research.

Google provides style guides for many programming languages. You can find the R style guide here[7]. Below are a few of the key points from the guide that we will use right away.

2.7.1 Naming Files

File names should be meaningful and end in `.R`. If we write a script that analyzes a certain species distribution:

- GOOD: `african_rhino_distribution.R`
- GOOD: `africanRhinoDistribution.R`
- BAD: `speciesDist.R` (too ambiguous)
- BAD: `species.dist.R` (too ambiguous and two periods can confuse operating systems' file type auto-detect)
- BAD: `speciesdist.R` (too ambiguous and confusing)

2.7.2 Naming Variables

- GOOD: `rhino.count`
- GOOD: `rhinoCount`

[7] https://google.github.io/styleguide/Rguide.xml

- GOOD: `rhino_count` (We don't mind the underscore and use it quite often, although Google's style guide says it's a no-no for some reason)
- BAD: `rhinocount` (confusing)

2.7.3 Syntax

- Keep code lines under 80 characters long.
- Indent your code with two spaces. (RStudio does this by default when you press the TAB key.)

3

Scripts, R Markdown, and Reproducible Research

Doing work in data science, whether for homework, a project for a business, or a research project, typically involves several iterations. For example, creating an effective graphical representation of data can involve trying out several different graphical representations, and then tens if not hundreds of iterations when fine-tuning the chosen representation. And each of these representations may require several R commands to create. Although this all could be accomplished by typing and re-typing commands at the R Console, it is easier and more effective to write the commands in a *script file* that can then be submitted to the R console either a line at a time or all together.[1]

In addition to making the workflow more efficient, R scripts provide another large benefit. Often we work on one part of a homework assignment or project for a few hours, then move on to something else, and then return to the original part a few days, months, or sometimes even years later. In such cases we may have forgotten how we created a graphical display that we were so proud of, and will again need to spend a few hours to recreate it. If we save a script file, we have the ingredients immediately available when we return to a portion of a project.[2]

Next consider a larger scientific endeavor. Ideally a scientific study will be reproducible, meaning that an independent group of researchers (or the original researchers) will able to duplicate the study. Thinking about data science, this means that all the steps taken when working with the data from a study should be reproducible, from selection of variables to formal data analysis. In principle this can be facilitated by explaining, in words, each step of the work with data. In practice, on the other hand, it is typically difficult or impossible to reproduce a full data analysis based on a written explanation. It is much more effective to include the actual computer code that accomplished the data work in the report, whether the report is a homework assignment or a research paper. Tools in R such as *R Markdown* facilitate this process.

[1] Unsurprisingly it is also possible to submit several selected lines of code at once.

[2] In principle the R history mechanism provides a similar record. But with history we have to search through a lot of other code to find what we're looking for, and scripts are a much cleaner mechanism to record our work.

3.1 Scripts in R

As noted above, scripts help to make working with data more efficient and provide a record of how data were managed and analyzed. Here we describe an example using the FEF data.[3] First we read the FEF data into R using the code below.

```
> face.dat <- read.csv(
+     file="http://blue.for.msu.edu/FOR472/data/FACE_aspen_core_growth.csv"
+ )
```

Next we print the names of the variables in the data set. Don't be concerned about the specific details. Later we will learn much more about reading in data and working with data sets in R.

```
> names(face.dat)
```

```
 [1] "Rep"                "Treat"
 [3] "Clone"              "E.Clone"
 [5] "Row"                "Col"
 [7] "ID.."               "X1997Initial_Height"
 [9] "X1997Initial_Diam"  "X1997Final_Height"
[11] "X1997Final_Diam"    "X1998_Height"
[13] "X1998_Diam"         "X1999_Height"
[15] "X1999_Diam"         "X2000_Height"
[17] "X2000_Diam"         "X2001_Height"
[19] "X2001_AvgDiam"      "X2001_Diam.3cm"
[21] "X2001_Diam.10cm"    "X2002_Height"
[23] "X2002_Diam.10cm"    "X2003_Height"
[25] "X2003_Diam.10cm"    "X2003_DBH"
[27] "X2004_Height"       "X2004_Diam.10cm"
[29] "X2004_DBH"          "X2005_Height"
[31] "X2005_Diam.10cm"    "X2005_DBH"
[33] "X2006_Height"       "X2006_DBH"
[35] "X2007_Height"       "X2007_DBH"
[37] "X2008_Height"       "X2008_DBH"
[39] "Notes"              "Comment1"
[41] "Comment2"           "Comment3"
[43] "Comment4"           "Comment5"
```

[3] The example uses features of R that we have not yet discussed, so don't worry about the details but rather about how it motivates the use of a script file.

```
[45] "Comment6"
```

Let's create a scatter plot of 2008 DBH versus height. To do this we'll first create variables for DBH and height taken in the year 2008 and print out the first ten values of each variable.[4]

```
> dbh <- face.dat$X2008_DBH
> ht <- face.dat$X2008_Height
> dbh[1:10]
```

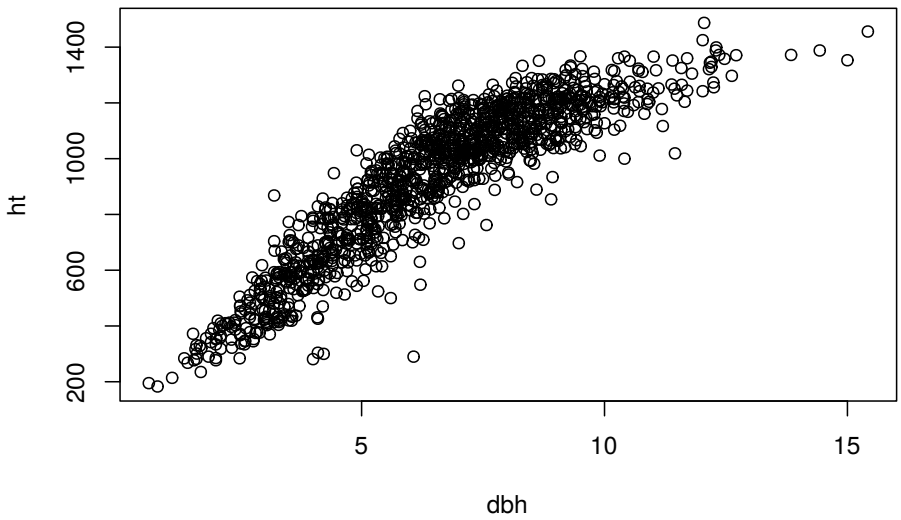
```
 [1]   NA 9.55 2.00 9.00 3.11 6.35 4.60   NA   NA 1.42
```

```
> ht[1:10]
```

```
 [1]   NA 1225  334 1079  370  859  818   NA   NA  268
```

The `NA` is how missing data are represented in R. Their presence here suggests several trees in this data set are dead or not measured for some reason in 2008. Of course at some point it would be good to investigate which trees have missing data and why. The `plot()` function in R will omit missing values, and for now we will just plot the non-missing data. A scatter plot of the data is drawn next.

```
> plot(dbh, ht)
```



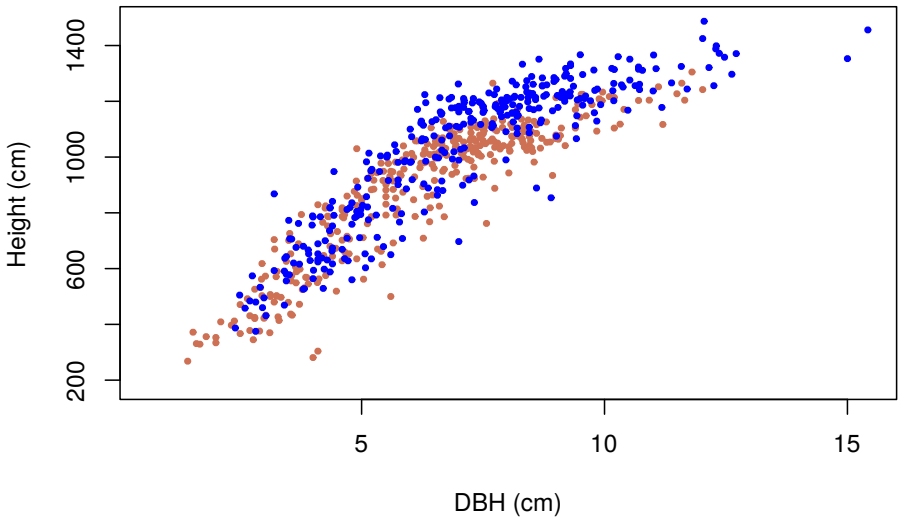
[4]Neither of these steps are necessary, but are convenient for illustration.

Not surprisingly, the scatter plot shows that DBH and height are positively correlated and the relationship is nonlinear. Now that we have a basic scatter plot, it is tempting to make it more informative. We will do this by adding a feature that identifies which trees belong to the control and elevated CO_2 environment treatments. We do this by first separating DBH and height into their respective treatment groups.

```
> treat <- face.dat$Treat
> dbh.treat.1 <- dbh[treat == 1]  ##Treatment 1 is the control
> ht.treat.1 <- ht[treat == 1]
>
> dbh.treat.2 <- dbh[treat == 2]  ##Treatment 2 is the elevated CO2
> ht.treat.2 <- ht[treat == 2]
```

To make a more informative scatter plot we will do two things. First make a plot for treatment 1 data, but ensure the plot region is large enough to include the treatment 2 data. This is done by specifying the range of the plot axes via `xlim` and `ylim` arguments in the `plot()` function. Here the `xlim` and `ylim` are set to the range of `dbh` and `ht` values, respectively, using `range()` (try and figure out what the `na.rm` argument does in the range function). Second we add treatment 2 data via the `points()` function. There are several other arguments passed to the plot function, but don't worry about these details for now.

```
> plot(dbh.treat.1, ht.treat.1, xlim = range(dbh, na.rm = TRUE),
+   ylim = range(ht, na.rm = TRUE), pch = 19, col = "salmon3",
+   cex = 0.5, xlab = "DBH (cm)", ylab = "Height (cm)")
> points(dbh.treat.2, ht.treat.2, pch = 19, col = "blue",
+   cex = 0.5)
```



Of course we should have a plot legend to tell the viewer which colors are associated with the treatments, as well as many other aesthetic refinements. For now, however, we will resist such temptations.[5]

Some of the process leading to the completed plot is shown above. We read in the data, created an intermediate plot by adding treatment identifiers, creating variables representing the 2008 measurements of DBH and height, and so on. However, a lot of the process isn't shown. For example, I made several mistakes in the process of getting the code and plot the way I wanted it—forgot the `na.rm=TRUE` initially then fiddled around with the treatment colors a bit.

Now imagine trying to recreate the plot a few days later. Possibly someone saw the plot and commented that it would be interesting to see similar plots for each year in the study period. If we did all the work, including all the false starts and refinements, at the console it would be hard to sort things out. This would take much longer than necessary to create the new plots. This would be especially true if a few months had passed, rather than just a few days.

Creating the new scatter plots would be much easier with a script file, especially if it had a few well-chosen comments. Fortunately it is quite easy to create and work with script files in RStudio.[6] Just choose `File > New File > New script` and a script window will open up in the upper left of the full RStudio window.

An example of a script window (with some R code already typed in) is shown

[5] As an aside, by only looking at the plotted data and thinking about basic plant physiology, can you guess which color is associated with the elevated CO_2 treatment?

[6] It is also easy in R without RStudio. Just use `File > New script` to create a script file, and save it before exiting R.

in Figure 3.1. From the script window the user can, among other things, save the script (either using the `File` menu or the icon near the top left of the window) and can run one or more lines of code from the window (using the `run` icon in the window, or by copying and pasting into the console window). In addition, there is a `Source on Save` checkbox. If this is checked, the R code in the script window is automatically read into R and executed when the script file is saved.

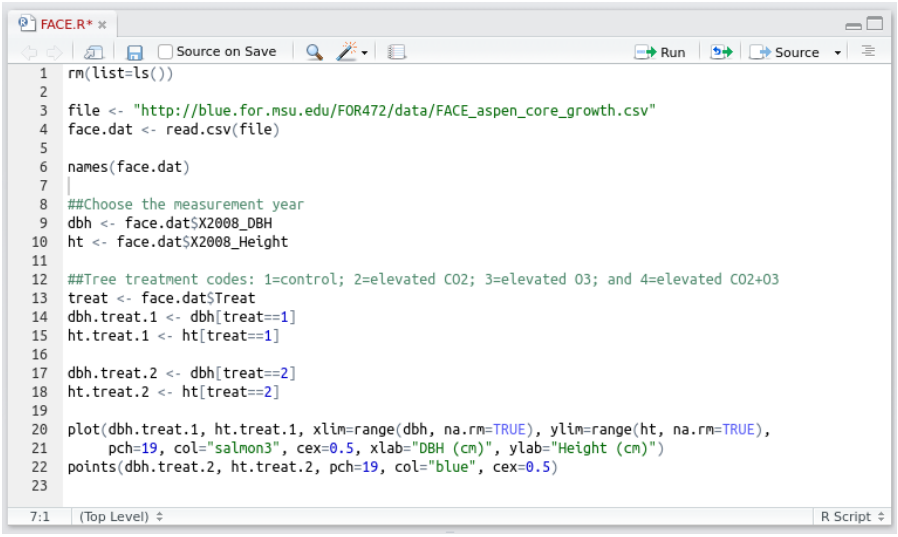


FIGURE 3.1: A script window in RStudio

3.2 R Markdown

People typically work on data with a larger purpose in mind. Possibly the purpose is to understand a biological system more clearly. Possibly the purpose is to refine a system that recommends movies to users in an online streaming movie service. Possibly the purpose is to complete a homework assignment and demonstrate to the instructor an understanding of an aspect of data analysis. Whatever the purpose, a key aspect is communicating with the desired audience.

One possibility, which is somewhat effective, is to write a document using software such as Microsoft Word [7] and to include R output such as computations

[7] Or possibly LaTeX if the document is more technical

and graphics by cutting and pasting into the main document. One drawback to this approach is similar to what makes script files so useful: If the document must be revised it may be hard to unearth the R code that created graphics or analyses, to revise these.[8] A more subtle but possibly more important drawback is that the reader of the document will not know precisely how analyses were done, or how graphics were created. Over time even the author(s) of the paper will forget the details. A verbal description in a "methods" section of a paper can help here, but typically these do not provide all the details of the analysis, but rather might state something like, "All analyses were carried out using R version 3.3.1."

RStudio's website provides an excellent overview of R Markdown capabilities for reproducible research. At minimum, follow the `Get Started` link at `http://rmarkdown.rstudio.com/` and watch the introduction video.

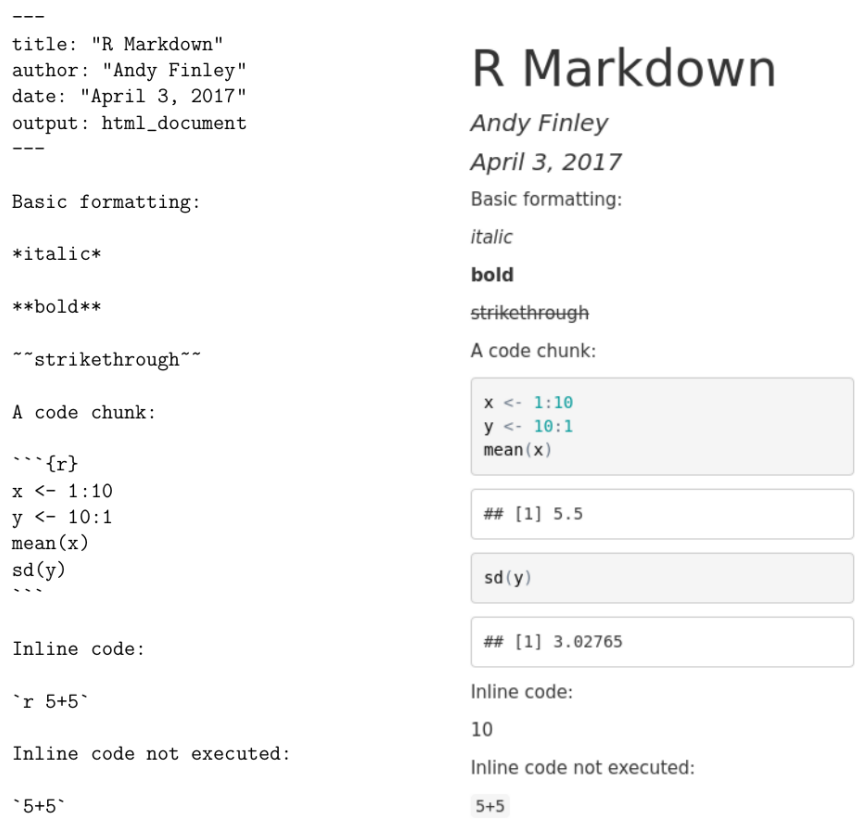
Among other things, R Markdown provides a way to include R code that read in data, create graphics, or perform analyses, all in a single document that is processed to create a research paper, homework assignment, or other written product. The R Markdown file is a plain text file containing text the author wants to show in the final document, simple commands to indicate how the text should be formatted (for example boldface, italic, or a bulleted list), and R code that creates output (including graphics) on the fly. Perhaps the simplest way to get started is to see an R Markdown file and the resulting document that is produced after the R Markdown document is processed. In Figure 3.2 we show the input and output of an example R Markdown document. In this case the output created is an HTML file, but there are other possible output formats, such as Microsoft Word or PDF.

At the top of the input R Markdown file are some lines with `---` at the top and the bottom. These lines are not needed, but give a convenient way to specify the title, author, and date of the article that are then typeset prominently at the top of the output document. For now, don't be concerned with the lines following `output:`. These can be omitted (or included as shown).

Next are a few lines showing some of the ways that font effects such as italics, boldface, and strikethrough can be achieved. For example, an asterisk before and after text sets the text in *italics*, and two asterisks before and after text sets the text in **boldface**.

More important for our purposes is the ability to include R code in the R Markdown file, which will be executed with the output appearing in the output document. Bits of R code included this way are called *code chunks*. The beginning of a code chunk is indicated with three backticks and an "r" in curly

[8] Organizing the R code using script files and keeping all the work organized in a well-thought-out directory structure can help here, but this requires a level of forethought and organization that most people do not possess . . . including myself.



```
---
title: "R Markdown"
author: "Andy Finley"
date: "April 3, 2017"
output: html_document
---

Basic formatting:

*italic*

**bold**

~~strikethrough~~

A code chunk:

```{r}
x <- 1:10
y <- 10:1
mean(x)
sd(y)
```

Inline code:

`r 5+5`

Inline code not executed:

`5+5`
```

FIGURE 3.2: Example R Markdown Input and Output

braces: ```{r}. The end of a code chunk is indicated with three backticks ```. For example, the R Markdown file in Figure 3.2 has one code chunk:

```
```{r}
x = 1:10
y = 10:1
mean(x)
sd(y)
```
```

In this code chunk two vectors `x` and `y` are created, and the mean of `x` and the standard deviation of `y` are computed. In the output in Figure 3.2 the R code is reproduced, and the output of the two lines of code asking for the mean and standard deviation is shown.

3.2.1 Creating and processing R Markdown documents

RStudio has features which facilitate creating and processing R Markdown documents. Choose `File > New File > R Markdown...`. In the ensuing dialog box, make sure that `Document` is highlighted on the left, enter the title and author (if desired), and choose the Default Output Format (HTML is good to begin). Then click OK. A document will appear in the upper left of the RStudio window. It is an R Markdown document, and the title and author you chose will show up, delimited by `---` at the top of the document. A generic body of the document will also be included.

For now just keep this generic document as is. To process it to create the HTML output, click the `Knit HTML` button at the top of the R Markdown window[9]. You'll be prompted to choose a filename for the R Markdown file. Make sure that you use `.Rmd` as the extension for this file. Once you've successfully saved the file, RStudio will process the file, create the HTML output, and open this output in a new window. The HTML output file will also be saved to your working directory. This file can be shared with others, who can open it using a web browser such as Chrome or Firefox.

There are many options which allow customization of R Markdown documents. Some of these affect formatting of text in the document, while others affect how R code is evaluated and displayed. The RStudio web site contains a useful summary of many R Markdown options at `https://www.rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf`. A different, but mind-numbingly busy, cheatsheet is at `https://www.rstudio.com/wp-content/uploads/2016/03/rmarkdown-cheatsheet-2.0.pdf`. Some of the more commonly used R Markdown options are described next.

3.2.2 Text: Lists and Headers

Unordered (sometimes called bulleted) lists and ordered lists are easy in R Markdown. Figure 3.3 illustrates the creation of unordered and ordered lists.

- For an unordered list, either an asterisk, a plus sign, or a minus sign may precede list items. Use a space after these symbols before including the list text. To have second-level items (sub-lists) indent four spaces before indicating the list item. This can also be done for third-level items.
- For an ordered list use a numeral followed by a period and a space (1. or 2. or 3. or …) to indicate a numbered list, and use a letter followed by a period and a space (a. or b. or c. or …) to indicate a lettered list. The same four

[9] If you hover your mouse over this Knit button after a couple seconds it should display a keyboard shortcut for you to do this if you don't like pushing buttons

```
An unordered list:

* List item 1
* List item 2
    + Second level list item 1
    + Second level list item 2
        + Third level list item
* List item 3

An ordered list:

1. List item 1
2. List item 2
    c. Sub list item 1
    q. Sub list item 2
17. List item 3
```

An unordered list:

- List item 1
- List item 2
  - Second level list item 1
  - Second level list item 2
    - Third level list item
- List item 3

An ordered list:

1. List item 1
2. List item 2
   c. Sub list item 1
   d. Sub list item 2
3. List item 3

FIGURE 3.3: Producing Lists in R Markdown

space convention used in unordered lists is used to designate ordered sub lists.

- For an ordered list, the first list item will be labeled with the number or letter that you specify, but subsequent list items will be numbered sequentially. The example in Figure 3.3 will make this more clear. In those examples notice that for the ordered list, although the first-level numbers given in the R Markdown file are 1, 2, and 17, the numbers printed in the output are 1, 2, and 3. Similarly the letters given in the R Markdown file are c and q, but the output file prints c and d.

R Markdown does not give substantial control over font size. Different "header" levels are available that provide different font sizes. Put one or more hash marks in front of text to specify different header levels. Other font choices such as subscripts and superscripts are possible, by surrounding the text either by tildes or carets. More sophisticated mathematical displays are also possible, and are surrounded by dollar signs. The actual mathematical expressions are specified using a language called LaTeX See Figures 3.4 and 3.5 for examples.

3.2.3 Code Chunks

R Markdown provides a large number of options to vary the behavior of code chunks. In some contexts it is useful to display the output but not the R code leading to the output. In some contexts it is useful to display the R prompt, while in others it is not. Maybe we want to change the size of figures created

```
# A first *level* ~~header~~

## A second level header

### A third level header

Text subscripts and superscripts:

x~2~ + y~2~

10^3^ = 1000

Mathematics examples:

$x_a$

$x^a$

$\int_0^1 x^2 dx$

$\frac{x}{y}$

$\sqrt{x}$

$\sqrt[n]{x}$

$\sum_{k=1}^n$

$\prod_{k=1}^n$
```

A first *level* ~~header~~

A second level header

A third level header

Text subscripts and superscripts:

$x_2 + y_2$

$10^3 = 1000$

Mathematics examples:

x_a

x^a

$\int_0^1 x^2 dx$

$\frac{x}{y}$

$\sqrt{x}$

$\sqrt[n]{x}$

$\sum_{k=1}^n$

$\prod_{k=1}^n$

FIGURE 3.4: Headers and Some LaTeX in R Markdown

by graphics commands. And so on. A large number of code chunk options are described in http://www.rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf.

Code chunk options are specified in the curly braces near the beginning of a code chunk. Below are a few of the more commonly used options are described. The use of these options is illustrated in Figure 3.6.

1. `echo=FALSE` specifies that the R code itself should not be printed, but any output of the R code should be printed in the resulting document.
2. `include=FALSE` specifies that neither the R code nor the output should be printed. However, the objects created by the code chunk will be available for use in later code chunks.
3. `eval=FALSE` specifies that the R code should not be evaluated. The

$\leq$	\leq	$\geq$	\geq	$\neq$	\neq	$\approx$	\approx
$\times$	\times	$\div$	\div	$\pm$	\pm	$\cdot$	\cdot
$^\circ$	^{\circ}	$\circ$	\circ	$\prime$	\prime	$\cdots$	\cdots
∞	\infty	$\neg$	\neg	$\wedge$	\wedge	$\vee$	\vee
$\supset$	\supset	$\forall$	\forall	$\in$	\in	$\rightarrow$	\rightarrow
$\subset$	\subset	$\exists$	\exists	$\notin$	\notin	$\Rightarrow$	\Rightarrow
$\cup$	\cup	$\cap$	\cap	$\mid$	\mid	$\Leftrightarrow$	\Leftrightarrow
$\dot{a}$	\dot a	$\hat{a}$	\hat a	$\bar{a}$	\bar a	$\tilde{a}$	\tilde a
α	\alpha	β	\beta	γ	\gamma	δ	\delta
ϵ	\epsilon	ζ	\zeta	η	\eta	ε	\varepsilon
θ	\theta	ι	\iota	κ	\kappa	ϑ	\vartheta
λ	\lambda	μ	\mu	ν	\nu	ξ	\xi
π	\pi	ρ	\rho	σ	\sigma	τ	\tau
υ	\upsilon	ϕ	\phi	χ	\chi	ψ	\psi
ω	\omega	Γ	\Gamma	Δ	\Delta	Θ	\Theta
Λ	\Lambda	Ξ	\Xi	Π	\Pi	Σ	\Sigma
Υ	\Upsilon	Φ	\Phi	Ψ	\Psi	Ω	\Omega

FIGURE 3.5: Other useful LaTeX symbols and expressions in R Markdown

code will be printed unless, for example, `echo=FALSE` is also given as an option.

4. `error=FALSE` and `warning=FALSE` specify that, respectively, error messages and warning messages generated by the R code should not be printed.

5. The `comment` option allows a specified character string to be prepended to each line of results. By default this is set to `comment = '##'` which explains the two hash marks preceding the results in Figure 3.2. Setting `comment = NA` presents output without any character string prepended. That is done in most code chunks in this book.

6. `prompt=TRUE` specifies that the R prompt `>` will be prepended to each line of R code shown in the document. `prompt = FALSE` specifies that command prompts should not be included.

7. `fig.height` and `fig.width` specify the height and width of figures generated by R code. These are specified in inches. For example, `fig.height=4` specifies a four inch high figure.

Figures 3.6 gives examples of the use of code chunk options.

````
No options:
```{r}
x <- 1:10
x
```

echo=FALSE:
```{r, echo=FALSE}
x <- 1:10
x
```

comment=NA:
```{r, comment=NA}
x <- 1:10
x
```
comment='#', prompt=TRUE:
```{r, comment='#', prompt=TRUE}
x <- 1:10
x
```

echo=FALSE, fig.height=4, fig.width=4:

```{r, echo=FALSE, fig.height=4, fig.width=4}
y <- 10:1
plot(x,y)
```
````

No options:

```
x = 1:10
x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

echo=FALSE:

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

comment=NA:

```
x = 1:10
x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

comment='#', prompt=TRUE:

```
> x = 1:10
> x
```

```
#  [1]  1  2  3  4  5  6  7  8  9 10
```

echo=FALSE, fig.height=4, fig.width=4:

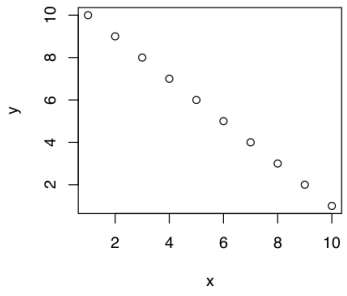


FIGURE 3.6: Output of Example R Markdown

3.2.4 Output formats other than HTML

It is possible to use R Markdown to produce documents in formats other than HTML, including Word and PDF documents. Next to the `Knit HTML` button is a down arrow. Click on this and choose `Knit Word` to produce a Microsoft word output document. Although there is also a `Knit PDF` button, PDF output requires additional software called TeX in addition to RStudio.[10]

3.2.5 LaTeX, knitr, and bookdown

While R Markdown provides substantial flexibility and power, it lacks features such as cross-referencing, fine control over fonts, etc. If this is desired, a variant of R Markdown called `knitr`, which has very similar syntax to R Markdown for code chunks, can be used in conjunction with the typesetting system LaTeX to produce documents. We originally created this book using knitr and LaTeX. For simpler tasks, however, R Markdown is sufficient, and substantially easier to learn.

As you know (since you are reading this) we are currently converting this book into R Markdown using the package `bookdown` written by Yihui Xie. This package utilizes the R Markdown style that is described above, and also incorporates numerous other features that R Markdown alone does not have (see the previous paragraph). Perhaps the best part about `bookdown` (in addition to it's lovely formatting style) is that we can make it interactive, so as you read the html version of this book you can interact with the code itself. You will experience this first hand when you work through the spatial data and databases chapters.

[10] It isn't particularly hard to install TeX software. For a Microsoft Windows system, MiKTeX is convenient and is available from https://miktex.org. For a Mac system, MacTeX is available from https://www.tug.org/mactex/

4

Data Structures

A data structure is a format for organizing and storing data. The structure is designed so that data can be accessed and worked with in specific ways. Statistical software and programming languages have methods (or functions) designed to operate on different kinds of data structures.

This chapter's focus is on data structures. To help initial understanding, the data in this chapter will be relatively modest in size and complexity. The ideas and methods, however, generalize to larger and more complex data sets.

The base data structures in R are vectors, matrices, arrays, data frames, and lists. The first three, vectors, matrices, and arrays, are *homogeneous*, meaning that all elements are required to be of the same type (e.g., all numeric or all character). Data frames and lists are *heterogeneous*, allowing elements to be of different types (e.g., some elements of a data frame may be numeric while other elements may be character). These base structures can also be organized by their dimensionality, as shown in Table 4.1.

R has no scalar types (0-dimensional). Individual numbers or strings are actually vectors of length one.

An efficient way to understand what comprises a given object is to use the `str()` function. `str()` is short for structure and prints a compact, human-readable description of any R data structure. For example, in the code below, we prove to ourselves that what we might think of as a scalar value is actually a vector of length one.

```
> a <- 1
> str(a)
```

TABLE 4.1: Dimension and Type Content of Base Data Structures in R.

Dimension	Homogeneous	Heterogeneous
1	Atomic Vector	List
2	Matrix	Data Frame
N	Array	

```
 num 1
```

```
> is.vector(a)
```

```
[1] TRUE
```

```
> length(a)
```

```
[1] 1
```

Here we assigned `a` the scalar value one. The `str(a)` prints `num 1`, which says `a` is numeric of length one. Then just to be sure we used the function `is.vector()` to test if `a` is in fact a vector. Then, just for fun, we computed the length of `a` which again returns one. There are a set of similar logical tests for the other base data structures, e.g., `is.matrix()`, `is.array()`, `is.data.frame()`, and `is.list()`. These will all come in handy as we encounter different R objects.

4.1 Vectors

Think of a vector[1] as a structure to represent one variable in a data set. For example a vector might hold the DBH, in inches, of six trees in a data set, and another vector might hold the species of those six trees. The `c()` function in R is useful for creating vectors and for modifying existing vectors. Think of `c` as standing for "combine"" or "concatenate."

```
> dbh <- c(20, 18, 13, 16, 10, 14)
> dbh
```

```
[1] 20 18 13 16 10 14
```

```
> spp <- c("Acer rubrum", "Acer rubrum", "Betula lenta", "Betula lenta",
+          "Prunus serotina", "Prunus serotina")
> spp
```

```
[1] "Acer rubrum"     "Acer rubrum"
```

[1] Technically the objects described in this section are "atomic" vectors (all elements of the same type), since lists are also actually vectors. This will not be an important issue in this course, and the shorter term vector will be used for atomic vectors.

```
[3] "Betula lenta"    "Betula lenta"
[5] "Prunus serotina" "Prunus serotina"
```

Notice that elements of a vector are separated by commas when using the `c()` function to create a vector. Also notice that character values are placed inside quotation marks.

The `c()` function also can be used to add to an existing vector. For example, if a seventh tree were included in the data set, and its DBH was 13 inches, the existing vectors could be modified as follows.

```
> dbh <- c(dbh, 13)
> spp <- c(spp, "Acer rubrum")
> dbh
```

```
[1] 20 18 13 16 10 14 13
```

```
> spp
```

```
[1] "Acer rubrum"     "Acer rubrum"
[3] "Betula lenta"    "Betula lenta"
[5] "Prunus serotina" "Prunus serotina"
[7] "Acer rubrum"
```

4.1.1 Types, Conversion, and Coercion

Clearly it is important to distinguish between different types of vectors. For example, it makes sense to ask R to calculate the mean of the DBH stored in `dbh`, but does not make sense to ask R to compute the mean of the species stored in `spp`. Vectors in R may have one of six different "types": character, double, integer, logical, complex, and raw. Only the first four of these will be of interest below, and the distinction between double and integer will not be of great import. To illustrate logical vectors, imagine the field technician who measured the trees also indicated if the tree was acceptable growing stock (`ags`) and the call was coded as TRUE if the tree was acceptable and FALSE if the tree was not acceptable.

```
> typeof(dbh)
```

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

```
> typeof(spp)
```

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

```
> ags <- c(TRUE, TRUE, FALSE, TRUE, FALSE, FALSE, TRUE)
> ags
```

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

```
> typeof(ags)
```

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

It may be surprising to see the DBH variable `dbh` is of type `double`, even though its values are all integers. By default R creates a double type vector when numeric values are given via the `c()` function.

When it makes sense, it is possible to convert vectors to a different type. Consider the following examples.

```
> dbh.int <- as.integer(dbh)
> dbh.int
```

```
[1] 20 18 13 16 10 14 13
```

```
> typeof(dbh.int)
```

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

```
> dbh.char <- as.character(dbh)
> dbh.char
```

```
[1] "20" "18" "13" "16" "10" "14" "13"
```

```
> ags.double <- as.double(ags)
> ags.double
```

```
[1] 1 1 0 1 0 0 1
```

```
> spp.oops <- as.double(spp)
```

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

```
> spp.oops
```

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

```
> sum(ags)
```

```
[1] 4
```

The integer version of `dbh` doesn't look any different, but it is stored differently, which can be important both for computational efficiency and for interfacing with other languages such as C++. As noted above, however, we will not worry about the distinction between integer and double types. Converting `dbh` to character goes as expected—the character representation of the numbers replace the numbers themselves. Converting the logical vector `ags` to double is pretty straightforward too—`FALSE` is converted to zero, and `TRUE` is converted to one. Now think about converting the character vector `spp` to a numeric double vector. It's not at all clear how to represent "Acer rubrum" as a number. In fact in this case what R does is to create a double vector, but with each element set to `NA`, which is the representation of missing data [2]. Finally consider the code `sum(ags)`. Now `ags` is a logical vector, but when R sees that we are asking to sum this logical vector, it automatically converts it to a numerical vector and then adds the zeros and ones representing FALSE and TRUE.

R also has functions to test whether a vector is of a particular type.

```
> is.double(dbh)
```

```
[1] TRUE
```

```
> is.character(dbh)
```

```
[1] FALSE
```

```
> is.integer(dbh.int)
```

```
[1] TRUE
```

```
> is.logical(ags)
```

```
[1] TRUE
```

[2] Missing data will be discussed in more detail later in the chapter.

4.1.1.1 Coercion

Consider the following examples.

```
> xx <- c(1, 2, 3, TRUE)
> xx
```

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

```
> yy <- c(1, 2, 3, "dog")
> yy
```

```
[1] "1"   "2"   "3"   "dog"
```

```
> zz <- c(TRUE, FALSE, "cat")
> zz
```

```
[1] "TRUE"  "FALSE" "cat"
```

```
> dbh + ags
```

```
[1] 21 19 13 17 10 14 14
```

Vectors in R can only contain elements of one type. If more than one type is included in a `c()` function, R silently *coerces* the vector to be of one type. The examples illustrate the hierarchy—if any element is a character, then the whole vector is character. If some elements are numeric (either integer or double) and other elements are logical, then the whole vector is numeric. Note what happened when R was asked to add the numeric vector `dbh` to the logical vector `ags`. The logical vector was silently coerced to be numeric, so that FALSE became zero and TRUE became one, and then the two numeric vectors were added.

4.1.2 Accessing Specific Elements of Vectors

To access and possibly change specific elements of vectors, refer to the position of the element in square brackets. For example, `dbh[4]` refers to the fourth element of the vector `dbh`. Note that R starts the numbering of elements at 1, i.e., the first element of a vector `x` is `x[1]`.

```
> dbh
```

```
[1] 20 18 13 16 10 14 13
```

```
> dbh[5]
```

```
[1] 10
```

```
> dbh[1:3]
```

```
[1] 20 18 13
```

```
> length(dbh)
```

```
[1] 7
```

```
> dbh[length(dbh)]
```

```
[1] 13
```

```
> dbh[]
```

```
[1] 20 18 13 16 10 14 13
```

```
> dbh[3] <- 202  ##crazy big tree
> dbh
```

```
[1]  20  18 202  16  10  14  13
```

```
> dbh[1:3] <- c(16, 8, 2)
> dbh
```

```
[1] 16  8  2 16 10 14 13
```

Note that including nothing in the square brackets results in the whole vector being returned. We can also assign values to vectors by accessing the position(s) where the new values will be assigned. For example, in the above code chunk `dbh[3]` is changed to 202, then the values in the first three elements of `dbh` are changed to 10, 8, and 2, respectively.

Negative numbers in the square brackets tell R to omit the corresponding value. And a zero as a subscript returns nothing (more precisely, it returns a length zero vector of the appropriate type).

```
> dbh[-3]
```

```
[1] 16  8 16 10 14 13
```

```
> dbh[-length(dbh)]
```

```
[1] 16  8  2 16 10 14
```

```
> fewer.dbh <- dbh[-c(1, 3, 5)]
> fewer.dbh
```

```
[1]  8 16 14 13
```

```
> dbh[0]
```

```
numeric(0)
```

```
> dbh[c(0, 2, 1)]
```

```
[1]  8 16
```

```
> dbh[c(-1, 2)]
```

```
Error in dbh[c(-1, 2)]: only 0's may be mixed with negative subscripts
```

Note that mixing zero and other nonzero subscripts is allowed, but mixing negative and positive subscripts is not allowed.

What about the (usual) case where we don't know the positions of the elements we want? For example possibly we want the DBH of all acceptable growing stock trees in the data. Later we will learn how to subset using logical indices, which is a very powerful way to access desired elements of a vector [3].

4.2 Factors

Categorical variables such as `spp` can be represented as character vectors. In many cases this simple representation is sufficient. Consider, however, two

[3] We had a prelude to this in the temperature data exercise.

other categorical variables, one representing crown class `S`, `I`, `C`, and `D` (i.e., **S**uppressed, **I**ntermediate, **C**odominate, **D**ominant), and another representing grade of the first log via categories `Grade 1`, `Grade 2`, and `Grade 3`. Suppose that for the small data set considered here, all trees are either dominant or codominant crown class. If we just represented the variable via a character vector, there would be no way to know that there are two other categories, representing suppressed and intermediate, because they happen to not be present in the data set. In addition, for the log grade variable the character vector representation does not explicitly indicate that there is an ordering of the levels.

Factors in R provide a more sophisticated way to represent categorical variables. Factors explicitly contain all possible levels, and allow ordering of levels.

```
> crown.class <- c("D", "D", "C", "D", "D", "C", "C")
> grade <- c("Grade 1", "Grade 1", "Grade 3", "Grade 2", "Grade 3",
+   "Grade 3", "Grade 2")
> crown.class
```

```
[1] "D" "D" "C" "D" "D" "C" "C"
```

```
> grade
```

```
[1] "Grade 1" "Grade 1" "Grade 3" "Grade 2" "Grade 3"
[6] "Grade 3" "Grade 2"
```

```
> crown.class <- factor(crown.class, levels = c("S", "I",
+   "C", "D"))
> crown.class
```

```
[1] D D C D D C C
Levels: S I C D
```

```
> grade <- factor(grade, levels = c("Grade 1", "Grade 2",
+   "Grade 3"), ordered = TRUE)
> grade
```

```
[1] Grade 1 Grade 1 Grade 3 Grade 2 Grade 3 Grade 3
[7] Grade 2
Levels: Grade 1 < Grade 2 < Grade 3
```

In the factor version of crown class the levels are explicitly listed, so it is clear that the two included levels are not all the possible levels. In the factor version of log grade, the ordering is explicit as well.

In many cases the character vector representation of a categorical variable is sufficient and easier to work with. In this book factors will not be used extensively. It is important to note that R often by default creates a factor when character data are read in, and sometimes it is necessary to use the argument `stringsAsFactors = FALSE` to explicitly tell R not to do this. This will be seen below when data frames are introduced.

4.3 Names of objects in R

Continuing the discussion about code quality from Section 2.7, there are a few hard and fast restrictions on the names of objects (such as vectors) in R. Note also that there are good practices, and things to avoid.

From the help page for `make.names` in R, the name of an R object is "syntactically valid" if the name "consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number" and is not one of the "reserved words" in R such as `if`, `TRUE`, `function`, etc. For example, `c45t.le_dog` and `.ty56` are both syntactically valid (although not very good names) while `4DislikeCats` and `log#@sparty` are not.

A few important comments about naming objects follow:

1. It is important to be aware that names of objects in R are case-sensitive, so `dbh` and `DBH` do not refer to the same object.

```
> dbh
```

```
[1] 16  8  2 16 10 14 13
```

```
> DBH
```

```
Error in eval(expr, envir, enclos): object 'DBH' not found
```

2. It is unwise to create an object with the same name as a built-in R object such as the function `c` or the function `mean`. In earlier versions of R this could be somewhat disastrous, but even in current versions, it is definitely not a good idea!
3. As much as possible, choose names that are informative. When creating a variable you may initially remember that `x` contains DBH and `y` contains crown class, but after a few hours, days, or weeks, you probably will forget this. Better options are `dbh` and `crown.class`.

4. As much as possible, be consistent in how you name objects. In particular, decide how to separate multi-word names. Some options include:
    - Using case to separate: `CrownClass` or `crownClass` for example
    - Using underscores to separate: `crown_class` for example
    - Using a period to separate: `crown.class` for example

4.4 Missing Data, Infinity, etc.

Most real-world data sets have variables where some observations are missing. In longitudinal studies of tree growth (i.e., where trees are measured over time), it is common that trees die or cannot be located in subsequent remeasurements. Statistical software should be able to represent missing data and to analyze data sets in which some data are missing.

In R, the value `NA` is used for a missing data value. Since missing values may occur in numeric, character, and other types of data, and since R requires that a vector contain only elements of one type, there are different types of `NA` values. Usually R determines the appropriate type of `NA` value automatically. It is worth noting the default type for `NA` is logical, and that `NA` is NOT the same as the character string `"NA"`.

```
> missingCharacter <- c("dog", "cat", NA, "pig", NA, "horse")
> missingCharacter
```

```
[1] "dog"   "cat"   NA      "pig"   NA      "horse"
```

```
> is.na(missingCharacter)
```

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

```
> missingCharacter <- c(missingCharacter, "NA")
> missingCharacter
```

```
[1] "dog"   "cat"   NA      "pig"   NA      "horse"
[7] "NA"
```

```
> is.na(missingCharacter)
```

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

```
> allMissing <- c(NA, NA, NA)
> typeof(allMissing)
```

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

How should missing data be treated in computations, such as finding the mean or standard deviation of a variable? One possibility is to return `NA`. Another is to remove the missing value(s) and then perform the computation.

```
> mean(c(1, 2, 3, NA, 5))
```

```
[1] NA
```

```
> mean(c(1, 2, 3, NA, 5), na.rm = TRUE)
```

```
[1] 2.75
```

As this example shows, the default behavior for the `mean()` function is to return `NA`. If removal of the missing values and then computing the mean is desired, the argument `na.rm` is set to `TRUE`. Different R functions have different default behaviors, and there are other possible actions. Consulting the help for a function provides the details.

4.4.1 Infinity and NaN

What happens if R code requests division by zero, or results in a number that is too large to be represented? Here are some examples.

```
> x <- 0:4
> x
```

```
[1] 0 1 2 3 4
```

```
> 1/x
```

```
[1]    Inf 1.0000 0.5000 0.3333 0.2500
```

```
> x/x
```

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

```
> y <- c(10, 1000, 10000)
> 2^y
```

```
[1]  1.024e+03 1.072e+301        Inf
```

`Inf` and `-Inf` represent infinity and negative infinity (and numbers which are too large in magnitude to be represented as floating point numbers). `NaN` represents the result of a calculation where the result is undefined, such as dividing zero by zero. All of these are common to a variety of programming languages, including R.

4.5 Data Frames

Commonly data is rectangular in form, with variables as columns and cases as rows. Continuing with the species, DBH, and acceptable growing stock data, each of those variables would be a column of the data set, and each tree's measurements would be a row. In R, such data are represented as a *data frame*.

```
> trees <- data.frame(Spp=spp, Dbh=dbh, Ags=ags,
+                     stringsAsFactors=FALSE)
> trees
```

```
              Spp Dbh   Ags
1     Acer rubrum  16  TRUE
2     Acer rubrum   8  TRUE
3    Betula lenta   2 FALSE
4    Betula lenta  16  TRUE
5 Prunus serotina  10 FALSE
6 Prunus serotina  14 FALSE
7     Acer rubrum  13  TRUE
```

```
> names(trees)
```

```
[1] "Spp" "Dbh" "Ags"
```

```
> colnames(trees)
```

```
[1] "Spp" "Dbh" "Ags"
```

```
> names(trees) <- c("species", "DBH", "good.stock")
> colnames(trees)
```

```
[1] "species"    "DBH"        "good.stock"
```

```
> rownames(trees)
```

```
[1] "1" "2" "3" "4" "5" "6" "7"
```

```
> names(trees) <- c("spp", "dbh", "ags")
> dim(trees)
```

```
[1] 7 3
```

The `data.frame` function can be used to create a data frame (although it's more common to read a data frame into R from an external file, something that will be introduced later). The names of the variables in the data frame are given as arguments, as are the vectors of data that make up the variable's values. The argument `stringsAsFactors=FALSE` asks R not to convert character vectors into factors, which R does by default, to the dismay of many users. Names of the columns (variables) can be extracted and set via either `names` or `colnames`. In the example, the variable names are changed to `species, DBH, good.stock` and then changed back to what I like better `spp, dbh, ags` in this way. Rows can be named also. In this case since specific row names were not provided the default row names of `"1", "2"`, etc. are used. Finally, I take a look at the data frame's dimensions (where the `dim` function returns a vector comprised of number of rows and number of columns, respectively). Also, try the functions `nrow` and `ncol` on the data frame and see what happens.

In the next example a built-in R data set called `Loblolly` is made available by the `data` function, and then the first and last six rows are displayed using `head` and `tail`.

```
> data("Loblolly")
> head(Loblolly)
```

```
   height age Seed
1    4.51   3  301
15  10.89   5  301
```

```
29  28.72  10  301
43  41.74  15  301
57  52.70  20  301
71  60.92  25  301
```

```
> tail(Loblolly)
```

```
   height age Seed
14   3.46   3  331
28   9.05   5  331
42  25.85  10  331
56  39.15  15  331
70  49.12  20  331
84  59.49  25  331
```

Note the `Loblolly` data frame has row names that are not ordered (which really doesn't matter) and simply suggests the data set author might have subset these data from a larger data set or sorted them by a variable, e.g., `height` or `age`. Row names can be generally ignored (unless they hold some specific meaning). Find out more about the Loblolly data set by running `?Loblolly` on the command line or, equivalently, looking it up in the RStudio's search window on the help tab.

4.5.1 Accessing specific elements of data frames

Data frames are two-dimensional, so to access a specific element (or elements) we need to specify both the row and column indices.

```
> Loblolly[1, 3]
```

```
[1] 301
14 Levels: 329 < 327 < 325 < 307 < 331 < ... < 305
```

```
> Loblolly[1:3, 3]
```

```
[1] 301 301 301
14 Levels: 329 < 327 < 325 < 307 < 331 < ... < 305
```

```
> Loblolly[1:3, 2:3]
```

```
   age Seed
1    3  301
```

```
15   5  301
29  10  301
```

```
> Loblolly[, 1]
```

```
 [1]  4.51 10.89 28.72 41.74 52.70 60.92  4.55 10.92
 [9] 29.07 42.83 53.88 63.39  4.79 11.37 30.21 44.40
[17] 55.82 64.10  3.91  9.48 25.66 39.07 50.78 59.07
[25]  4.81 11.20 28.66 41.66 53.31 63.05  3.88  9.40
[33] 25.99 39.55 51.46 59.64  4.32 10.43 27.16 40.85
[41] 51.33 60.07  4.57 10.57 27.90 41.13 52.43 60.69
[49]  3.77  9.03 25.45 38.98 49.76 60.28  4.33 10.79
[57] 28.97 42.44 53.17 61.62  4.38 10.48 27.93 40.20
[65] 50.06 58.49  4.12  9.92 26.54 37.82 48.43 56.81
[73]  3.93  9.34 26.08 37.79 48.31 56.43  3.46  9.05
[81] 25.85 39.15 49.12 59.49
```

Note that `Loblolly[,1]` returns ALL elements in the first column. This agrees with the behavior for vectors, where leaving a subscript out of the square brackets tells R to return all values. In this case we are telling R to return all rows, and the first column.

As we have seen in class, we can also access the columns (or rows) using their names.

```
> Loblolly[1:4, "height"]
```

```
[1]  4.51 10.89 28.72 41.74
```

```
> Loblolly[1:4, c("age", "Seed")]
```

```
   age Seed
1    3  301
15   5  301
29  10  301
43  15  301
```

For a data frame there is another way to access specific columns, using the `$` notation.

```
> Loblolly$height
```

```
 [1]  4.51 10.89 28.72 41.74 52.70 60.92  4.55 10.92
 [9] 29.07 42.83 53.88 63.39  4.79 11.37 30.21 44.40
```

```
[17] 55.82 64.10  3.91  9.48 25.66 39.07 50.78 59.07
[25]  4.81 11.20 28.66 41.66 53.31 63.05  3.88  9.40
[33] 25.99 39.55 51.46 59.64  4.32 10.43 27.16 40.85
[41] 51.33 60.07  4.57 10.57 27.90 41.13 52.43 60.69
[49]  3.77  9.03 25.45 38.98 49.76 60.28  4.33 10.79
[57] 28.97 42.44 53.17 61.62  4.38 10.48 27.93 40.20
[65] 50.06 58.49  4.12  9.92 26.54 37.82 48.43 56.81
[73]  3.93  9.34 26.08 37.79 48.31 56.43  3.46  9.05
[81] 25.85 39.15 49.12 59.49
```

```
> Loblolly$age
```

```
 [1]  3  5 10 15 20 25  3  5 10 15 20 25  3  5 10 15 20
[18] 25  3  5 10 15 20 25  3  5 10 15 20 25  3  5 10 15
[35] 20 25  3  5 10 15 20 25  3  5 10 15 20 25  3  5 10
[52] 15 20 25  3  5 10 15 20 25  3  5 10 15 20 25  3  5
[69] 10 15 20 25  3  5 10 15 20 25  3  5 10 15 20 25
```

```
> height
```

```
Error in eval(expr, envir, enclos): object 'height' not found
```

```
> age
```

```
Error in eval(expr, envir, enclos): object 'age' not found
```

Notice that typing the variable name, such as `height`, without the name of the data frame (and a dollar sign) as a prefix, does not work. This is sensible. There may be several data frames that have variables named `height`, and just typing `height` doesn't provide enough information to know which is desired.

4.6 Lists

The third main data structure we will work with is a list. Technically a list is a vector, but one in which elements can be of different types. For example a list may have one element that is a vector, one element that is a data frame, and another element that is a function. Consider designing a function that fits a simple linear regression model to two quantitative variables. We might want that function to compute and return several things such as

- The fitted slope and intercept (a numeric vector with two components)

- The residuals (a numeric vector with n components, where n is the number of data points)
- Fitted values for the data (a numeric vector with n components, where n is the number of data points)
- The names of the dependent and independent variables (a character vector with two components)

In fact R has a function, `lm`, which does this (and much more).

```
> htAgeLinMod <- lm(height ~ age, data = Loblolly)
> mode(htAgeLinMod)
```

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

```
> names(htAgeLinMod)
```

```
 [1] "coefficients"  "residuals"     "effects"
 [4] "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "xlevels"
[10] "call"          "terms"         "model"
```

```
> htAgeLinMod$coefficients
```

```
(Intercept)         age
     -1.312       2.591
```

```
> tail(htAgeLinMod$residuals)
```

```
    14     28     42     56     70     84
-2.999 -2.590  1.257  1.605 -1.378 -3.961
```

The `lm` function returns a list (which in the code above has been assigned to the object `htAgeLinMod`)[4]. One component of the list is the length 2 vector of coefficients, while another component is the length 84 vector of residuals. The code also illustrates that named components of a list can be accessed using the dollar sign notation, as with data frames.

The `list` function is used to create lists.

```
> temporaryList <- list(first = dbh, second = trees, pickle = list(a = 1:10,
+   b = trees))
> temporaryList
```

[4] The `mode` function returns the the type or storage mode of an object.

```
$first
[1] 16  8  2 16 10 14 13

$second
              spp dbh   ags
1     Acer rubrum  16  TRUE
2     Acer rubrum   8  TRUE
3    Betula lenta   2 FALSE
4    Betula lenta  16  TRUE
5 Prunus serotina  10 FALSE
6 Prunus serotina  14 FALSE
7     Acer rubrum  13  TRUE

$pickle
$pickle$a
 [1]  1  2  3  4  5  6  7  8  9 10

$pickle$b
              spp dbh   ags
1     Acer rubrum  16  TRUE
2     Acer rubrum   8  TRUE
3    Betula lenta   2 FALSE
4    Betula lenta  16  TRUE
5 Prunus serotina  10 FALSE
6 Prunus serotina  14 FALSE
7     Acer rubrum  13  TRUE
```

Here, for illustration, I assembled a list to hold some of the R data structures we have been working with in this chapter. The first list element, named `first`, holds the `dbh` vector we created in Section 4.1. The second list element, named `second`, holds the `trees` data frame. The third list element, named `pickle`, holds a list with elements named `a` and `b` that hold a vector of values 1 through 10, and another copy of the `trees` data set, respectively. As this example shows, a list can contain another list.

4.6.1 Accessing specific elements of lists

We already have seen the dollar sign notation works for lists. In addition, the square bracket subsetting notation can be used. But with lists there is an added somewhat subtle wrinkle—using either single or double square brackets.

```
> temporaryList$first
```

```
[1] 16  8  2 16 10 14 13
```

```
> mode(temporaryList$first)
```

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

```
> temporaryList[[1]]
```

```
[1] 16  8  2 16 10 14 13
```

```
> mode(temporaryList[[1]])
```

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

```
> temporaryList[1]
```

```
$first
[1] 16  8  2 16 10 14 13
```

```
> mode(temporaryList[1])
```

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

Note the dollar sign and double bracket notation return a numeric vector, while the single bracket notation returns a list. Notice also the difference in results below.

```
> temporaryList[c(1, 2)]
```

```
$first
[1] 16  8  2 16 10 14 13

$second
              spp dbh   ags
1     Acer rubrum  16  TRUE
2     Acer rubrum   8  TRUE
3    Betula lenta   2 FALSE
4    Betula lenta  16  TRUE
5 Prunus serotina  10 FALSE
6 Prunus serotina  14 FALSE
7     Acer rubrum  13  TRUE
```

```
> temporaryList[[c(1, 2)]]
```

```
[1] 8
```

The single bracket form returns the first and second elements of the list, while the double bracket form returns the second element in the first element of the list. Generally, do not put a vector of indices or names in a double bracket, you will likely get unexpected results. See, for example, the results below [5].

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

```
Error in temporaryList[[c(1, 2, 3)]]: recursive indexing failed at level 2
```

So, in summary, there are two main differences between using the single bracket `[]` and double bracket `[[]]`. First, the single bracket will return a list that holds the object(s) held at the given indices or names placed in the bracket, whereas the double brackets will return the actual object held at the index or name placed in the innermost bracket. Put differently, a single bracket can be used to access a range of list elements and will return a list, while a double bracket can only access a single element in the list and will return the object held at the index.

4.7 Subsetting with Logical Vectors

Consider the `Loblolly` data frame. How can we access only those trees with heights more than 50 m? How can we access the age of those trees taller than 50 m? How can we compute the mean height of all trees from seed source 301? The data set is small enough that it would not be too onerous to extract the values by hand. But for larger or more complex data sets, this would be very difficult or impossible to do in a reasonable amount of time, and would likely result in errors.

R has a powerful method for solving these sorts of problems using a variant of the subsetting methods that we already have learned. When given a logical vector in square brackets, R will return the values corresponding to `TRUE`. To begin, focus on the `dbh` and `spp` vectors created in Section 4.1.

The R code `dbh > 15` returns `TRUE` for each value of `dbh` that is more than 15, and a `FALSE` for each value of `dbh` that is less than or equal to 15. Similarly `spp`

[5] Try this example using only single brackets... it will return a list holding elements `first`, `second`, and `pickle`.

`== "Betula lenta"` returns `TRUE` or `FALSE` depending on whether an element of `spp` is equal to `Betula lenta`.

```
> dbh
```

```
[1] 16  8  2 16 10 14 13
```

```
> dbh > 15
```

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

```
> spp[dbh > 15]
```

```
[1] "Acer rubrum"  "Betula lenta"
```

```
> dbh[dbh > 15]
```

```
[1] 16 16
```

```
> spp == "Betula lenta"
```

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

```
> dbh[spp == "Betula lenta"]
```

```
[1]  2 16
```

Consider the lines of R code one by one.

- `dbh` instructs R to display the values in the vector `dbh`.
- `dbh > 15` instructs R to check whether each value in `dbh` is greater than 15, and to return `TRUE` if so, and `FALSE` otherwise.
- The next line, `spp[dbh > 15]`, does two things. First, inside the square brackets, it does the same thing as the second line: it returns `TRUE` or `FALSE` depending on whether a value of `dbh` is or is not greater than 15. Second, each element of `spp` is matched with the corresponding `TRUE` or `FALSE` value, and is returned if and only if the corresponding value is `TRUE`. For example the first value of `spp` is Acer rubrum. Since the first `TRUE` or `FALSE` value is `TRUE`, the first value of `spp` is returned. So this line of code returns the species names for all trees with DBH greater than 15; hence, the first and the fourth values of `spp` are returned.
- The fourth line of code, `dbh[dbh > 15]`, again begins by returning `TRUE` or `FALSE` depending on whether elements of `dbh` are larger than 15. Then those

elements of `dbh` corresponding to `TRUE` values are returned. So this line of code returns the DBH of all trees whose DBH is greater than 15.

- The fifth line returns `TRUE` or `FALSE` depending on whether elements of `spp` are equal to `Betula lenta` or not.
- The sixth line returns the DBH of those all Betula lenta trees.

There are six comparison operators in R, `>, <, >=, <=, ==, !=`. Note that to test for equality a "double equals sign"" is used, while `!=` tests for inequality.

5

Manipulating Data with `dplyr`

Much of the effort (a figure of 80% is sometimes suggested) in data analysis is spent cleaning the data and getting it ready for analysis. Having effective tools for this task can save substantial time and effort. The R package `dplyr` written by Hadley Wickham is designed, in Hadley's words, to be "a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges." Functions provided by `dplyr` do in fact capture key data analysis actions (i.e., verbs). Below we describe a few of the key functions available in `dplyr`:

- `filter()` extracts rows based on their values
- `arrange()` changes the ordering of the rows
- `select()` extracts variables based on their names
- `mutate()` adds new variables that are functions of existing variables‘
- `summarize()` reduces multiple values down to a single summary

These all combine naturally with a `group_by` function that allows you to perform any operation grouped by values of one or more variables. All the tasks done using `dplyr` can be accomplished using more traditional R syntax; however, `dplyr`'s functions provide a potentially more efficient and convenient framework to accomplish these tasks. RStudio provides a convenient data wrangling cheat sheet[1] that covers many aspects of the `dplyr` package.

5.1 Minnesota tree growth data

We'll use some tree growth data to motivate the methods presented in this chapter. The data were collected in 2010 from 35 forest stands in and around Superior National Forest in northeastern Minnesota. See Foster et al. (2014) for details about data collection and preparation.

The tree growth data set consists of radial growth increments (collected using

[1] `http://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf%7D%7Bdata%20wrangling%20cheat%20sheet`

an increment borer) for 521 trees located in 105 forest plots in northeastern Minnesota from 1978 to 2007. The forest plots are distributed across 35 forest stands (3 plots per stand). Each stand represents an area with similar species composition and approximately homogeneous forest characteristics (e.g., trees/acre, tree size distribution, tree age distribution). In total, 15 species are present in the sample data. The "mn_trees.csv"" file that is read into the `mn.trees` data frame below contains the dated (`Year`) annual radial growth increment (`rad.inc` annual growth ring width in mm) and basal area increment (`BA.inc` cross-sectional area of annual growth in cm^2) estimates for each tree, along with ancillary tree-level information including species, diameter at breast height (DBH), and age. The data frame also includes a stand, plot, and tree identification number for each tree, i.e., `StandID`, `PlotID`, and `TreeID`, respectively. Each tree can be uniquely identified by its combined stand, plot, and tree ID values. For illustration, each line in Figure 5.1 is an individual tree's diameter growth over time from one stand colored by species. We also load the `dplyr` library for subsequent use[2].

```
> mn.trees <- read.csv("http://blue.for.msu.edu/FOR472/data/mn_trees.csv")
> str(mn.trees)
```

```
'data.frame':   15301 obs. of  9 variables:
 $ StandID: int  1 1 1 1 1 1 1 1 1 ...
 $ PlotID : int  1 1 1 1 1 1 1 1 1 ...
 $ TreeID : int  1 1 1 1 1 1 1 1 1 ...
 $ Species: Factor w/ 15 levels "ABBA","ACRU",..: 1 1 1 1 1 1 1 1 1 ...
 $ Year   : int  1978 1979 1980 1981 1982 1983 1984 1985 1986 1987 ...
 $ Age    : int  19 20 21 22 23 24 25 26 27 28 ...
 $ DBH    : num  5.23 5.44 5.56 5.75 5.99 ...
 $ rad.inc: num  0.92 1.035 0.61 0.935 1.245 ...
 $ BA.inc : num  1.48 1.73 1.05 1.66 2.3 ...
```

```
> library(dplyr)
```

```
Attaching package: 'dplyr'

The following objects are masked from 'package:stats':

    filter, lag
```

[2]The text printed immediately after `library(dplyr)` means the `stats` and `base` packages, which are automatically loaded when you start R, have functions with the same name as functions in `dplyr`. So, for example, if you call the `filter()` or `lag()`, R will use `library(dplyr)`'s functions. Use the `::` operator to explicitly identify which packages' function you want to use, e.g., if you want `stats`'s `lag()` then call `stats::lag()`.

```
The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

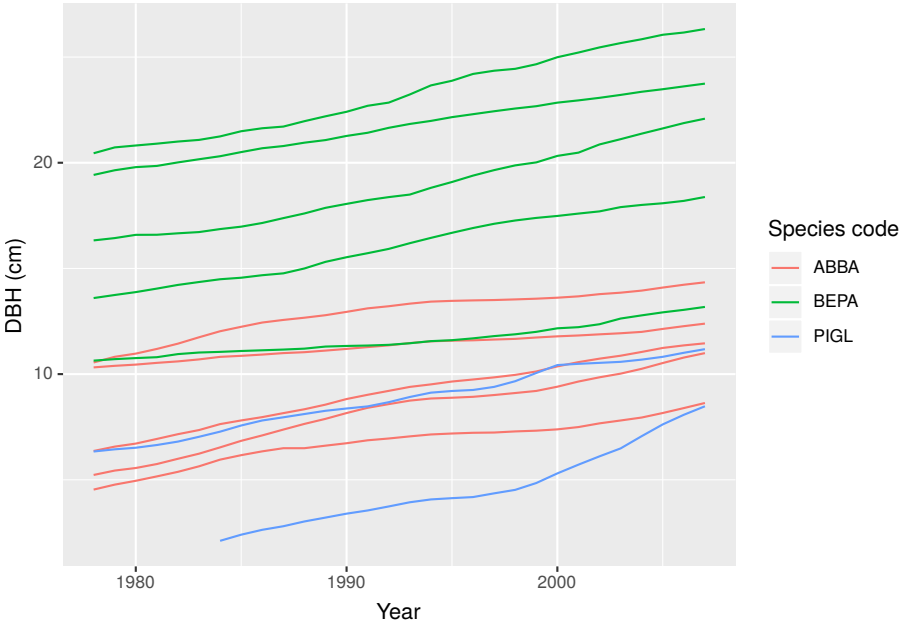


FIGURE 5.1: Tree core derived diameter at breast height (DBH cm) by year for sampled trees in Stand 1

5.2 Improved Data Frames

The `dplyr` package provides two functions that offer improvements on data frames. First, the `data_frame` function is a trimmed down version of the `data.frame` that is somewhat more user friendly, and won't be discussed here. Second, the `tbl_df` function creates a data frame like object called a tibble[3]. A tibble has two advantages over a data frame. First, when printing, it only prints the first ten rows and the columns that fit on the page, as well as some additional information about the table's dimension, data type of variables, and non-printed columns. Second, recall that subsetting a data frame can sometimes return a vector rather than a data frame (if only one row or column

[3]Reminds me of The Trouble with Tribbles[4]

is the result of the subset). A tibble does not have this behavior. Here is an example using the `mn.trees` data frame.

```
> is.data.frame(mn.trees[, 1])
```

```
[1] FALSE
```

```
> is.vector(mn.trees[, 1])
```

```
[1] TRUE
```

```
> mn.trees.tbl <- tbl_df(mn.trees)
> is.tbl(mn.trees.tbl[, 1])
```

```
[1] TRUE
```

```
> is.data.frame(mn.trees.tbl[, 1])
```

```
[1] TRUE
```

```
> mn.trees.tbl[, 1]
```

```
# A tibble: 15,301 x 1
   StandID
     <int>
 1       1
 2       1
 3       1
 4       1
 5       1
 6       1
 7       1
 8       1
 9       1
10       1
# ... with 15,291 more rows
```

Note, above that once the data frame is reduced to one dimension by subsetting to one column, it is no longer a data frame it has been *simplified* to a vector. This might not seem like a big deal; however, it can be very frustrating and potentially break your code when you expect an object to behave like a data frame and it doesn't because it's now a vector. Alternatively, once we convert `mn.trees` to a tibble via the `tbl_df` function the object remains a

data frame even when subset down to one dimension (there is no automatic simplification). Converting data frames using `tbl_df()` is not required for using `dplyr` but is convenient. Also, it is important to note that `tbl_df` is simply a wrapper around a data frame that provides some additional behaviors. The newly formed `tbl_df` object will still behave like a data frame (because it technically still is a data frame) but will have some added niceties (some of which are illustrated below).

5.3 Filtering Data By Row

Filtering creates row subsets of data that satisfy a logical statement. Considering the `mn.trees` data, the `filter` function can be used to examine a particular set of stands, plots, trees, measurement years, species, etc. The first argument in the `filter` function is the data frame or tibble from which you want to select the rows based on the logical statement given in the second argument. As you work through this chapter, you'll notice the data are specified in the first argument of all `dplyr`'s functions. Below are several illustrative applications of the `filter` function.

```
> filter(mn.trees.tbl, StandID == 5)
```

```
# A tibble: 450 x 10
   StandID PlotID TreeID Species  Year   Age   DBH
     <int>  <int>  <int> <fct>   <int> <int> <dbl>
 1       5     13     57 ABBA     1978     2  1.71
 2       5     13     57 ABBA     1979     3  2.12
 3       5     13     57 ABBA     1980     4  2.53
 4       5     13     57 ABBA     1981     5  2.97
 5       5     13     57 ABBA     1982     6  3.48
 6       5     13     57 ABBA     1983     7  3.98
 7       5     13     57 ABBA     1984     8  4.33
 8       5     13     57 ABBA     1985     9  4.70
 9       5     13     57 ABBA     1986    10  5.07
10       5     13     57 ABBA     1987    11  5.39
# ... with 440 more rows, and 3 more variables:
#   rad.inc <dbl>, BA.inc <dbl>, UTreeID <int>
```

```
> filter(mn.trees.tbl, Species %in% c("ABBA", "PIST"))
```

```
# A tibble: 2,876 x 10
```

```
   StandID PlotID TreeID Species  Year   Age   DBH
     <int>  <int>  <int> <fct>   <int> <int> <dbl>
 1       1      1      1 ABBA     1978    19  5.23
 2       1      1      1 ABBA     1979    20  5.44
 3       1      1      1 ABBA     1980    21  5.56
 4       1      1      1 ABBA     1981    22  5.75
 5       1      1      1 ABBA     1982    23  5.99
 6       1      1      1 ABBA     1983    24  6.24
 7       1      1      1 ABBA     1984    25  6.53
 8       1      1      1 ABBA     1985    26  6.84
 9       1      1      1 ABBA     1986    27  7.10
10       1      1      1 ABBA     1987    28  7.38
# ... with 2,866 more rows, and 3 more variables:
#   rad.inc <dbl>, BA.inc <dbl>, UTreeID <int>
```

```
> filter(mn.trees.tbl, DBH > 12 & Year == 1980)
```

```
# A tibble: 228 x 10
   StandID PlotID TreeID Species  Year   Age   DBH
     <int>  <int>  <int> <fct>   <int> <int> <dbl>
 1       1      1      4 BEPA     1980    74  16.6
 2       1      2      8 BEPA     1980    92  19.8
 3       1      2      9 BEPA     1980    75  13.9
 4       1      2     10 BEPA     1980    98  20.8
 5       2      4     14 BEPA     1980    98  19.5
 6       2      4     15 BEPA     1980   124  21.3
 7       2      4     16 BEPA     1980   116  18.4
 8       2      4     17 PIGL     1980    34  12.6
 9       2      5     18 PIGL     1980    34  12.4
10       2      5     19 PIRE     1980   242  58.4
# ... with 218 more rows, and 3 more variables:
#   rad.inc <dbl>, BA.inc <dbl>, UTreeID <int>
```

```
> filter(mn.trees.tbl, Species %in% c("ABBA", "PIST") & Year ==
+   1985)
```

```
# A tibble: 87 x 10
   StandID PlotID TreeID Species  Year   Age   DBH
     <int>  <int>  <int> <fct>   <int> <int> <dbl>
 1       1      1      1 ABBA     1985    26  6.84
 2       1      1      2 ABBA     1985    41  7.81
 3       1      2      6 ABBA     1985    19  6.16
 4       1      2      7 ABBA     1985    16 10.9
 5       1      3     11 ABBA     1985    40 12.2
```

```
 6       2      5     21 PIST     1985   149 35.9
 7       2      6     22 ABBA     1985    33 12.3
 8       3      7     28 PIST     1985    49 10.5
 9       3      7     29 PIST     1985    48 17.7
10       3      7     30 PIST     1985    52 22.4
# ... with 77 more rows, and 3 more variables:
#   rad.inc <dbl>, BA.inc <dbl>, UTreeID <int>
```

Notice the full results are not printed. For example, when we asked for the data for stand 5, only the first ten rows were printed. This is an effect of using the `tbl_df` function. Of course if we wanted to analyze the results (as we will below) the full set of data would be available.

5.4 Selecting Variables by Column

Another common task is to restrict attention to some subset of variables in the data set. This is accomplished using the `select` function. Like `filter`, the data frame or tibble is the first argument in the `select` function, followed by additional arguments identifying variables you want to include or exclude. Consider the examples below.

```
> select(mn.trees.tbl, Year, DBH)
```

```
# A tibble: 15,301 x 2
    Year   DBH
   <int> <dbl>
 1  1978  5.23
 2  1979  5.44
 3  1980  5.56
 4  1981  5.75
 5  1982  5.99
 6  1983  6.24
 7  1984  6.53
 8  1985  6.84
 9  1986  7.10
10  1987  7.38
# ... with 15,291 more rows
```

```
> select(mn.trees.tbl, 2:4)
```

```
# A tibble: 15,301 x 3
```

```
   PlotID TreeID Species
    <int>  <int> <fct>
 1      1      1 ABBA
 2      1      1 ABBA
 3      1      1 ABBA
 4      1      1 ABBA
 5      1      1 ABBA
 6      1      1 ABBA
 7      1      1 ABBA
 8      1      1 ABBA
 9      1      1 ABBA
10      1      1 ABBA
# ... with 15,291 more rows
```

```
> select(mn.trees.tbl, -c(2, 3, 4))
```

```
# A tibble: 15,301 x 7
   StandID  Year   Age   DBH rad.inc BA.inc UTreeID
     <int> <int> <int> <dbl>   <dbl>  <dbl>   <int>
 1       1  1978    19  5.23   0.92    1.48       1
 2       1  1979    20  5.44   1.03    1.73       1
 3       1  1980    21  5.56   0.61    1.05       1
 4       1  1981    22  5.75   0.935   1.66       1
 5       1  1982    23  5.99   1.25    2.30       1
 6       1  1983    24  6.24   1.20    2.31       1
 7       1  1984    25  6.53   1.50    3.00       1
 8       1  1985    26  6.84   1.54    3.25       1
 9       1  1986    27  7.10   1.28    2.81       1
10       1  1987    28  7.38   1.38    3.14       1
# ... with 15,291 more rows
```

```
> select(mn.trees.tbl, ends_with("ID"))
```

```
# A tibble: 15,301 x 4
   StandID PlotID TreeID UTreeID
     <int>  <int>  <int>   <int>
 1       1      1      1       1
 2       1      1      1       1
 3       1      1      1       1
 4       1      1      1       1
 5       1      1      1       1
 6       1      1      1       1
 7       1      1      1       1
 8       1      1      1       1
```

```
 9       1      1      1       1
10       1      1      1       1
# ... with 15,291 more rows
```

Notice a few things. Variables can be selected by name or column number. As usual a negative sign tells R to leave something out. Also, there are special *helper* functions such as `ends_with` that provide ways to match part of a variable's name. Other very handy helper functions you should investigate are `begins_with`, `contains`, `matches`, `num_range`, `one_of`, and `everything`.

5.5 Pipes

Consider selecting the `Age` and `rad.inc` for the two aspen species POTR or POGR (*Populus tremuloides* and *Populus grandifolia*). One possibility is to nest a `filter` call within `select`.

```
> select(filter(mn.trees.tbl, Species %in% c("POTR", "POGR")),
+   Age, rad.inc)
```

```
# A tibble: 719 x 2
     Age rad.inc
   <int>   <dbl>
 1    55   0.935
 2    56   0.7
 3    57   0.25
 4    58   0.595
 5    59   1.28
 6    60   1.34
 7    61   1.14
 8    62   1.20
 9    63   1.09
10    64   0.975
# ... with 709 more rows
```

Even a two-step process like this becomes hard to follow in this nested form, and often we will want to perform more than two operations. There is a nice feature in `dplyr` that allows us to "feed" results of one function into the first argument of a subsequent function. Another way of saying this is that we are "piping" the results into another function. The `%>%` operator does the piping.

```
> mn.trees.tbl %>%
+   filter(Species %in% c("POTR", "POGR")) %>%
+   select(Age, rad.inc)
```

```
# A tibble: 719 x 2
     Age rad.inc
   <int>   <dbl>
 1    55   0.935
 2    56   0.7
 3    57   0.25
 4    58   0.595
 5    59   1.28
 6    60   1.34
 7    61   1.14
 8    62   1.20
 9    63   1.09
10    64   0.975
# ... with 709 more rows
```

It can help to think of `%>%` as representing the word "then." The above can be read as, "Start with the data frame `mn.trees.tbl`, *then* filter it to select data from the species POTR and POGR, *then* select the variables age and radial growth increment from these data."

The pipe operator `%>%` is not restricted to functions in `dplyr`. In fact the pipe operator itself was introduced in another package called `magrittr`, but is included in `dplyr` as a convenience.

5.6 Arranging Data by Row

By default the `mn.trees` data are arranged in ascending order by `StandID`, then `PlotID`, then `TreeID`, then `Year`.

```
> head(mn.trees.tbl, 5)
```

```
# A tibble: 5 x 10
  StandID PlotID TreeID Species  Year   Age   DBH
    <int>  <int>  <int> <fct>   <int> <int> <dbl>
1       1      1      1 ABBA     1978    19  5.23
2       1      1      1 ABBA     1979    20  5.44
```

```
3       1      1      1 ABBA     1980    21  5.56
4       1      1      1 ABBA     1981    22  5.75
5       1      1      1 ABBA     1982    23  5.99
# ... with 3 more variables: rad.inc <dbl>,
#   BA.inc <dbl>, UTreeID <int>
```

```
> tail(mn.trees.tbl, 5)
```

```
# A tibble: 5 x 10
  StandID PlotID TreeID Species  Year   Age   DBH
    <int>  <int>  <int> <fct>   <int> <int> <dbl>
1      35    105    521 POTR     2003    32  17.5
2      35    105    521 POTR     2004    33  17.6
3      35    105    521 POTR     2005    34  17.9
4      35    105    521 POTR     2006    35  18.1
5      35    105    521 POTR     2007    36  18.2
# ... with 3 more variables: rad.inc <dbl>,
#   BA.inc <dbl>, UTreeID <int>
```

This is convenient ordering for these data. But what if we wanted to change the order to be by `Species` then `Year`? The `arrange` function makes this easy. The following examples illustrate `arrange` but also use pipes to simplify code and `select` to focus attention on the columns of interest.

Let's start with arranging in ascending order `BA.inc` for tree 1 in plot 1 in stand 1.

```
> mn.trees.tbl %>%
+   filter(StandID == 1 & PlotID == 1 & TreeID == 1) %>%
+   select(StandID, PlotID, TreeID, BA.inc) %>%
+   arrange(BA.inc)
```

```
# A tibble: 30 x 4
   StandID PlotID TreeID BA.inc
     <int>  <int>  <int>  <dbl>
 1       1      1      1  0.501
 2       1      1      1  0.615
 3       1      1      1  1.05
 4       1      1      1  1.18
 5       1      1      1  1.35
 6       1      1      1  1.38
 7       1      1      1  1.42
 8       1      1      1  1.48
 9       1      1      1  1.66
10       1      1      1  1.73
```

```
# ... with 20 more rows
```

Possibly we want these data to be in decreasing (descending) order. Here, `desc()` is one of many `dplyr` helper functions.

```
> mn.trees.tbl %>%
+   filter(StandID == 1 & PlotID == 1 & TreeID == 1) %>%
+   select(StandID, PlotID, TreeID, BA.inc) %>%
+   arrange(desc(BA.inc))
```

```
# A tibble: 30 x 4
   StandID PlotID TreeID BA.inc
     <int>  <int>  <int>  <dbl>
 1       1      1      1   4.45
 2       1      1      1   4.33
 3       1      1      1   3.67
 4       1      1      1   3.65
 5       1      1      1   3.59
 6       1      1      1   3.46
 7       1      1      1   3.25
 8       1      1      1   3.23
 9       1      1      1   3.16
10       1      1      1   3.14
# ... with 20 more rows
```

Passing multiple variables to `arrange` results in nested ordering. The subsequent code orders first by `Species`, then `Year` within `Species`, then `BA.inc` within `Year` and `Species`.

```
> mn.trees.tbl %>%
+   select(Species, Year, BA.inc) %>%
+   arrange(Species, Year, BA.inc)
```

```
# A tibble: 15,301 x 3
   Species  Year BA.inc
   <fct>   <int>  <dbl>
 1 ABBA     1978  0.253
 2 ABBA     1978  0.368
 3 ABBA     1978  0.416
 4 ABBA     1978  0.560
 5 ABBA     1978  0.620
 6 ABBA     1978  0.650
 7 ABBA     1978  0.672
 8 ABBA     1978  0.676
 9 ABBA     1978  0.699
```

```
10 ABBA     1978  0.760
# ... with 15,291 more rows
```

For analyzing data in R, the order shouldn't matter. But for presentation to human eyes, the order is important.

5.7 Renaming Variables

The `dplyr` package has a `rename` function that makes renaming variables in a data frame quite easy. Below, I rename the `rad.inc` and `BA.inc` to remind myself of their measurement units (i.e., millimeters and centimeters squared, respectively).

```
> mn.trees.tbl <- rename(mn.trees.tbl, rad.inc.mm = rad.inc,
+                        BA.inc.cm2 = BA.inc)
> head(mn.trees.tbl)
```

```
# A tibble: 6 x 10
  StandID PlotID TreeID Species  Year   Age   DBH
    <int>  <int>  <int> <fct>   <int> <int> <dbl>
1       1      1      1 ABBA     1978    19  5.23
2       1      1      1 ABBA     1979    20  5.44
3       1      1      1 ABBA     1980    21  5.56
4       1      1      1 ABBA     1981    22  5.75
5       1      1      1 ABBA     1982    23  5.99
6       1      1      1 ABBA     1983    24  6.24
# ... with 3 more variables: rad.inc.mm <dbl>,
#   BA.inc.cm2 <dbl>, UTreeID <int>
```

5.8 Creating New Variables

We routinely want to create new variables and add them to an existing data frame. This task is done using the `mutate` function. `mutate` adds new columns to the right side of your data frame or tibble. This function is particularly useful because it can make a new variable by simply referencing variables that exist in the data frame. Let's start with a simple example. Below I create a small data frame called `df` with two numeric columns `a` and `b`. Next, I add a

new variable c that is the square root of the sum of a and b. We can of course use mutate to add variables that are not a function of existing variables, e.g., see the addition of the logical variable d below (this time using a pipe).

```
> df <- data.frame("a"=1:4, "b"=c(8, 12, 19, 76))
> df <- mutate(df, c = log(a+b))
> df
```

```
  a  b     c
1 1  8 2.197
2 2 12 2.639
3 3 19 3.091
4 4 76 4.382
```

```
> df <- df %>%
+   mutate(d = c("Jerry","Jerry","Bobby","Bobby"))
> df
```

```
  a  b     c     d
1 1  8 2.197 Jerry
2 2 12 2.639 Jerry
3 3 19 3.091 Bobby
4 4 76 4.382 Bobby
```

Sometimes, we want to create new variables that are a function of existing variables but not add them to the data frame. In this case we use the transmute function. Here, I create a new data frame df.2 that comprises two new variables, e and f where e is a+c and f is just a copy of d.

```
> df.2 <- df %>% transmute(e = a + c, f = d)
> df.2
```

```
      e     f
1 3.197 Jerry
2 4.639 Jerry
3 6.091 Bobby
4 8.382 Bobby
```

5.9 Data Summaries and Grouping

The `summarize` function computes summary statistics or user provided functions for one or more variables in a data frame. Below, the `summarize` function is used to calculate the mean and sum of variable `a` in the data frame created in the previous section.

```
> summarize(df, a.mean = mean(a), a.sum = sum(a))
```

```
  a.mean a.sum
1    2.5    10
```

```
> ##or
> df %>%
+   summarize(a.mean = mean(a), a.sum = sum(a))
```

```
  a.mean a.sum
1    2.5    10
```

By itself, `summarize` is of limited use. Often we want row summaries for specific components of the data. For example, say we want to sum variable `a` for each category of variable `d`. One option is to subset and summarize for each category in `b`:

```
> df %>%
+   filter(d == "Jerry") %>%
+   summarize(a.sum = sum(a))
```

```
  a.sum
1     3
```

```
> df %>%
+   filter(d == "Bobby") %>%
+   summarize(a.sum = sum(a))
```

```
  a.sum
1     7
```

This is a very tedious approach if the number of subsets is large. Also, we might want the result as a single data frame, which means we would need to then combine the summaries of all the subsets in a subsequent step.

The group_by function used in combination with summarize simplifies this task and makes the output more useful. The summarize function is applied to each category in the *grouping* variable specified in group_by, as illustrated in the code below.

```
> df %>%
+   group_by(d) %>%
+   summarize(a.sum = sum(a))
```

```
# A tibble: 2 x 2
  d     a.sum
  <chr> <int>
1 Bobby     7
2 Jerry     3
```

We can specify multiple variables with group_by to define the categories to summarize. Let's return to the mn.trees data set and find the sum of annual radial growth increment by species within each stand.

```
> stand.spp <- mn.trees.tbl %>%
+   group_by(StandID, Species) %>%
+   summarize(rad.inc.total = sum(rad.inc.mm),
+             BA.inc.total = sum(BA.inc.cm2))
> stand.spp
```

```
# A tibble: 115 x 4
# Groups:   StandID [?]
   StandID Species rad.inc.total BA.inc.total
     <int> <fct>           <dbl>        <dbl>
 1       1 ABBA            109.         309.
 2       1 BEPA            122.         733.
 3       1 PIGL             58.0        121.
 4       2 ABBA             88.8        262.
 5       2 BEPA            101.         488.
 6       2 PIGL             86.3        431.
 7       2 PIRE             35.7        510.
 8       2 PIST             60.3        749.
 9       3 ABBA             57.6        135.
10       3 PIGL             21.1         94.8
# ... with 105 more rows
```

There are several things to notice here. Our code specifies both StandID and Species variables in the group_by function, which causes summarize arguments to be applied to each stand and species combination. For example, looking at the output, we can see that stand 1 contains three species ABBA (*Abies*

balsamea), `BEPA` (*Betula papyrifera*), and `PIGL` (*Picea glauca*). Also, for each of these species, the sum of radial growth increments, i.e., `rad.inc.total`, is 108.735, 121.515, and 58.045, and the sum of basal area growth increments, i.e., `BA.inc.total`, is approximately 308.98, 732.58, and 121.26. The two commented lines above the resulting tibble tell us there are 115 such stand and species combinations, i.e., `# A tibble: 115 x 4`, and the tibble is grouped by `StandID`, i.e., `# Groups: StandID [?]`. This last bit of information is important. Recall a tibble is a data frame with some additional functionality. Not only does the resulting tibble hold the `summarize` output, it also retains all levels of grouping to the left of the last grouping variable specified in `group_by`. In this case we grouped by `StandID` and `Species` prior to calling `summarize`, so the resulting tibble is grouped by `StandID`. If we fed the resulting tibble back into `summarize`, aggregation would occur for each stand. Depending on the analysis, this retention of grouping can be handy or annoying. If necessary, use `ungroup` to remove the grouping from a tibble, e.g., `stand.spp %>% ungroup()`.

5.10 Counts

In many circumstances it is useful to know how many rows are being summarized. Continuing the previous example, say we want to know how many increment measurements comprise a given `StandID` and `Species` combination and, hence, went into the `rad.inc.total` and `BA.inc.total` summaries. The `n` function returns the count of rows per group (or number of rows in an ungrouped tibble). Below, I add a new variable called `n.inc` to the our previous `stand.spp`.

```
> stand.spp <- mn.trees.tbl %>%
+   group_by(StandID, Species) %>%
+   summarize(rad.inc.total = sum(rad.inc.mm),
+             BA.inc.total = sum(BA.inc.cm2),
+             n.inc = n())
> stand.spp
```

```
# A tibble: 115 x 5
# Groups:   StandID [?]
   StandID Species rad.inc.total BA.inc.total n.inc
     <int> <fct>           <dbl>        <dbl> <int>
 1       1 ABBA            109.         309.    150
 2       1 BEPA            122.         733.    150
 3       1 PIGL             58.0        121.     54
```

```
 4       2 ABBA             88.8        262.     45
 5       2 BEPA            101.         488.    180
 6       2 PIGL             86.3        431.     90
 7       2 PIRE             35.7        510.     60
 8       2 PIST             60.3        749.     30
 9       3 ABBA             57.6        135.     30
10       3 PIGL             21.1         94.8    60
# ... with 105 more rows
```

This is helpful. We now know how many individual increment measurements were used to compute the summaries. However, it is not clear how many trees were actually cored to generate these measurements. We can get at the count of unique trees in each group by using the `n_distinct` helper function.

```
> stand.spp <- mn.trees.tbl %>% group_by(StandID, Species) %>%
+     summarize(rad.inc.total = sum(rad.inc.mm),
+               BA.inc.total = sum(BA.inc.cm2),
+               n.inc = n(),
+               n.trees = n_distinct(PlotID, TreeID))
> stand.spp
```

```
# A tibble: 115 x 6
# Groups:   StandID [?]
   StandID Species rad.inc.total BA.inc.total n.inc
     <int> <fct>           <dbl>        <dbl> <int>
 1       1 ABBA            109.         309.    150
 2       1 BEPA            122.         733.    150
 3       1 PIGL             58.0        121.     54
 4       2 ABBA             88.8        262.     45
 5       2 BEPA            101.         488.    180
 6       2 PIGL             86.3        431.     90
 7       2 PIRE             35.7        510.     60
 8       2 PIST             60.3        749.     30
 9       3 ABBA             57.6        135.     30
10       3 PIGL             21.1         94.8    60
# ... with 105 more rows, and 1 more variable:
#   n.trees <int>
```

Recall, `StandID`, `PlotID`, and `TreeID` identifies the set of increment measurements for a specific tree. Therefore, if we group using `StandID`, then `n_distinct(PlotID, TreeID)` returns the unique tree count within each stand. Above, we also group by `Species` so the unique tree count within the stand is also partitioned by species. Now, we can see the new variable `n.trees` is the number of trees by species over which `n.inc` increment measurements were collected.

6

Functions and Programming

We have been working with a wide variety of R functions, from simple functions such as `mean()`, `sum()`, and `length()` to more complex functions such as those found in the `dplyr` package. Gaining a better understanding of existing functions and the ability to write our own functions dramatically increases what you can do with R. Learning about R's programming capabilities is an important step in gaining facility with functions. Further, functional programming basics described in this chapter are found in nearly all programming and scripting languages.

6.1 R Functions

A function needs to have a name, perhaps some arguments, and a body of code that does something. At the end it usually returns an object (although it doesn't have to). An important idea behind functions is that objects created within the function only exist within the function unless they are explicitly returned by the function. Returned means passing a value or object back to the environment from which the function was called. Several examples will make this more clear.

Let's start with some pseudocode for a function[1]:

```
name.of.function <- function(argument.1, argument.2){

    do something

    return(something)
}
```

Here we are assigning the function definition to `name.of.function`. The syntax `function(argument.1, argument.2)` says we are creating a function with two arguments, with names `argument.1` and `argument.2`. What the

[1] Pseudocode is notation resembling a simplified programming language, used in program design.

function will do is defined between the curly brackets, i.e., where I wrote `do something`. Finally, before we close the function definition we return something we've created. This last step is optional, but most functions we write do return something.

The argument can be any type of object (like a single value, a matrix, a data frame, a vector, a logical, etc.). Here are a few simple examples of functions.

```
> my.first.function <- function() {
+   print("Ever stop to think, and forget to start again?")
+ }
>
> my.first.function()
```

```
[1] "Ever stop to think, and forget to start again?"
```

Here we define a function called `my.first.function()`. This function has no arguments, i.e., nothing is defined between the parentheses in `function()`. When the function is called, by writing `my.first.function()` on the command line, the function prints a fun quote by Alan A. Milne. No values or objects are formally returned by this function.

Now let's define a function that takes two arguments, the first value is raised by the second, and the result is returned.

```
> pow <- function(x, v) {
+   result <- x^v
+   return(result)
+ }
>
> pow(2, 5)
```

```
[1] 32
```

```
> pow(5, 0)
```

```
[1] 1
```

```
> pow(TRUE, FALSE)
```

```
[1] 1
```

```
> pow("a", "b")
```

```
Error in x^v: non-numeric argument to binary operator
```

Not surprisingly the last test of this function throws an error. Perhaps we should modify the function to first test if both `x` and `v` are numeric. There are lots of ways to perform this check (several of which we'll cover in a subsequent section), but for now consider this particularly convenient function call `stopifnot()`. If any logical tests in the `stopifnot()` function are `FALSE` then an error message is returned and the function evaluation is stopped. So let's revise the `pow()` function by adding a test that ensures both arguments are numeric.

```
> pow <- function(x, v) {
+   stopifnot(is.numeric(x), is.numeric(v))
+   result <- x^v
+   return(result)
+ }
>
> pow(2, 5)
```

```
[1] 32
```

```
> pow("a", "b")
```

```
Error in pow("a", "b"): is.numeric(x) is not TRUE
```

Let's motivate learning some more function features using data from one plot worth of tree measurements from the PEF. Below we read in tree measurements taken on plot 4 within management units 7A.

```
> file.name <- "http://blue.for.msu.edu/FOR472/data/PEF-mu7A-plot4.csv"
> trees.p4 <- read.csv(file.name, header = TRUE)
> trees.p4
```

```
   TreeID dbh Expf         CommonName
1  620377 4.6    5         balsam fir
2  620378 5.3    5         balsam fir
3  620379 4.8    5         balsam fir
4  620380 5.3    5       white spruce
5  620385 6.1    5 eastern white pine
6  620384 5.6    5 eastern white pine
7  620388 5.3    5 eastern white pine
8  620382 5.2    5 eastern white pine
9  620383 4.8    5 eastern white pine
10 620386 4.6    5 eastern white pine
```

```
11 620387 5.1    5 eastern white pine
12 620381 6.4    5 eastern white pine
13 620389 5.6    5 eastern white pine
14 620390 5.0    5      quaking aspen
15 620391 4.8    5      quaking aspen
16 620392 5.6    5      quaking aspen
```

Here, `TreeID` is a unique tree identifier, `dbh` is the diameter at breast height measured in inches, `Expf` is the expansion factor used to convert each stem to a per acre basis, and `CommonName` is tree species. Note, this is a 1/5-th acre fixed area plot, hence the expansion factor is 5 (i.e., the inverse of the plot area).

We will work toward developing a function that returns basal area and stem biomass given a tree's DBH and species. Let's start simple with a function that returns basal area given DBH.

```
> ba <- function(dbh) {
+   ba.sq.ft <- pi * dbh^2/(4 * 144)
+   return(ba.sq.ft)
+ }
>
> tree.1.dbh <- trees.p4$dbh[1]
> ba(tree.1.dbh)
```

```
[1] 0.1154
```

```
> ba(trees.p4$dbh)
```

```
 [1] 0.1154 0.1532 0.1257 0.1532 0.2029 0.1710 0.1532
 [8] 0.1475 0.1257 0.1154 0.1419 0.2234 0.1710 0.1364
[15] 0.1257 0.1710
```

A few things to notice here. Our function name is `ba`, it takes DBH as an argument, and returns basal area in square feet. The formula in the `ba()` function assumes DBH is given in inches. We tested the function twice. The first call to `ba()` passes in DBH for the first tree in `trees.p4` and returns the corresponding basal area. The second call to `ba()` passes in the DBH vector for trees in `trees.p4` and returns the corresponding basal area vector.

Now let's think about modifying our function to return biomass estimates given DBH and species. We can use allometric equations published by Jenkins et al. (2003) that take the form

$$bm = \exp(\beta_0 + \beta_1 \ln(DBH)) \tag{6.1}$$

TABLE 6.1: Parameters for estimating total aboveground biomass for species in the United States

species	beta.0	beta.1
quaking aspen	-2.209	2.387
balsam fir	-2.538	2.481
white spruce	-2.077	2.332
eastern white pine	-2.536	2.435

where

- *bm* is total aboveground biomass (km) for trees 2.5 cm DBH and larger
- DBH is diameter at breast height in cm
- exp is the exponential function
- ln is the natural logarithm, i.e., inverse function of the exponential function

Table 4 in Jenkins et al. (2003) provides species specific values for the regression coefficients β_0 and β_1. Table 6.1 below provides the regression coefficients for the species in our PEF plot 4 data.

Here are some things we'll need to accommodate in our function:

1. Jenkins allometric equations are parameterized for cm DBH, so the PEF's DBH will need to be converted from inches to cm.
2. There is a different set of regression coefficients for each species, so we'll need to pass in species and have the function apply the correct set of regression coefficients.

No problem addressing the first point in the list above, we can easily convert inches to cm. However, the second point will require some conditional statement tests. We'll tackle this in the next section.

6.2 Programming: Conditional Statements

We often want to apply different code conditional on characteristics of the data or objects at hand. This can be accomplished using an `if()` function. The argument of the `if()` function is a single logical value, i.e., `TRUE` or `FALSE`. If `TRUE`, the code within the `if()` function is evaluated. If `FALSE`, the `if()` function is skipped. Consider this simple example.

```
> my.if.example <- function(x) {
+   if (x) {
+     print("x is TRUE")
+   }
+
+   if (!x) {
+     print("x is FALSE")
+   }
+ }
>
> my.if.example(TRUE)
```

```
[1] "x is TRUE"
```

```
> my.if.example(FALSE)
```

```
[1] "x is FALSE"
```

To better understand and use `if()` statements, we need to understand comparison operators and logical operators.

6.2.1 Comparison and Logical Operators

We have made use of some of the comparison operators in R. These include

- Equal: `==`
- Not equal: `!=`
- Greater than: `>`
- Less than: `<`
- Greater than or equal to: `>=`
- Less than or equal to: `<=`

Special care needs to be taken with the `=` and `!=` operators because of how numbers are represented on computers.

There are also three logical operators, with two variants of the "and" operator and the "or" operator.

- and: Either `&` or `&&`
- or: Either `|` or `||`
- not: `!`

The "double" operators `&&` and `||` just examine the first element of the two vectors, whereas the "single" operators `&` and `|` compare element by element.

```
> c(FALSE, TRUE, FALSE) || c(TRUE, FALSE, FALSE)
```

```
[1] TRUE
```

```
> c(FALSE, TRUE, FALSE) | c(TRUE, FALSE, FALSE)
```

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

```
> c(FALSE, TRUE, FALSE) && c(TRUE, TRUE, FALSE)
```

```
[1] FALSE
```

```
> c(FALSE, TRUE, FALSE) & c(TRUE, TRUE, FALSE)
```

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

Often we want to evaluate one expression if the condition is true, and evaluate a different expression if the condition is false. That is accomplished using the `else if()` statement. Here we determine whether a number is positive, negative, or zero.

```
> sign <- function(x) {
+   if (x < 0) {
+     print("the number is negative")
+   } else if (x > 0) {
+     print("the number is positive")
+   } else {
+     print("the number is zero")
+   }
+ }
> sign(3)
```

```
[1] "the number is positive"
```

```
> sign(-3)
```

```
[1] "the number is negative"
```

```
> sign(0)
```

```
[1] "the number is zero"
```

Notice the sequence of conditional tests starts with `if()`. If this is not `TRUE`, moves to the `else if()` statement. If this is not `TRUE`, the sequence is terminated in an `else`. That final `else` acts as a catchall when all conditional tests above it are `FALSE`. Above, we have only one `else if` test, but you can have as many as you need, e.g., see the example below.

Okay, let's return to our PEF example and develop a function that applies the regression equation (6.1) using species specific regression coefficients given in Table 6.1.

```
> bio <- function(dbh.in, species) {
+
+   dbh.cm <- dbh.in * 2.54
+
+   if (dbh.cm < 2.5) {
+     stop("Only valid for trees greater than 2.5 cm DBH")
+   }
+
+   if (species == "quaking aspen") {
+     beta.0 <- -2.2094
+     beta.1 <- 2.3867
+   } else if (species == "balsam fir") {
+     beta.0 <- -2.5384
+     beta.1 <- 2.4814
+   } else if (species == "white spruce") {
+     beta.0 <- -2.0773
+     beta.1 <- 2.3323
+   } else if (species == "eastern white pine") {
+     beta.0 <- -2.5356
+     beta.1 <- 2.4349
+   } else {
+     stop("No coefficients available for the given species")
+   }
+
+   bm <- exp(beta.0 + beta.1 * log(dbh.cm))
+
+   return(bm)
+ }
>
> bio(5, "balsam fir")
```

```
[1] 43.31
```

```
> bio(2, "quaking aspen")
```

```
[1] 5.311
```

```
> bio(3, "Quaking aspen")
```

```
Error in bio(3, "Quaking aspen"): No coefficients available for the given species
```

Consider our new function `bio()` above that takes DBH (in) and species common name as arguments, `dbh.in` and `species`, respectively, and returns biomass (kg). The first line in the function's body converts DBH in inches to cm, because that's what the Jenkins et al. (2003) biomass equation expects. The `if()` statement that follows checks if DBH is at least 2.5 cm. If DBH is less than 2.5 cm then the `stop()` function stops the `bio()` function and reports the error message "Only valid for trees greater than 2.5 cm DBH." Next we go into a series of `if()` and `else if()` statements that identify the species specific values for the regression coefficients `beta.0` and `beta.1`. If the `species` argument is not one of the four species for which we have regression coefficients then the last `else` is reached and the code `stop()` is executed, which again terminates the `bio()` function followed by an explanation about why the function was stopped. The second to last line in `bio()`'s body is the regression equation defined in equation (6.1), now with the species specific regression coefficients `beta.0` and `beta.1`. Finally, `return(bm)` returns the resulting biomass (kg).

6.2.2 Functions with Multiple Returns

We often want a function to return multiple objects. This is most easily done by making the return object be a list with elements corresponding to the different objects we want returned. Recall lists can hold any combination of R objects. Here's a simple example. Let's make a function called `quick.summary` that returns the mean, median, and variance of a numeric vector. We can try it out using the DBH vector from `trees.p4`.

```
> quick.summary <- function(x) {
+
+   a <- mean(x)
+   b <- median(x)
+   c <- var(x)
+
+   result <- list(mean = a, median = b, var = c)
+
```

```
+   return(result)
+ }
>
> quick.summary(trees.p4$dbh)
```

```
$mean
[1] 5.256

$median
[1] 5.25

$var
[1] 0.264
```

6.2.3 Creating functions

Creating very short functions at the command prompt is a reasonable strategy. For longer functions, one option is to write the function in a script window and then submit the whole function. Or a function can be written in any text editor, saved as a plain text file (possibly with a `.R` extension), and then read into R using the `source()` command.

6.3 More on Functions

Understanding functions deeply requires a careful study of the scoping rules of R, as well as a good understanding of environments in R. That's beyond the scope of this book, but we will briefly discuss some issues that are most salient. For a more in-depth treatment, see "Advanced R" by Hadley Wickham, especially the chapters on functions and environments.

6.3.1 Calling Functions

When using a function, the function arguments can be specified in three ways:

- By the full name of the argument.
- By the position of the argument.
- By a partial name of the argument.

```
> tmp_function <- function(first.arg, second.arg, third.arg,
+   fourth.arg) {
+   return(c(first.arg, second.arg, third.arg, fourth.arg))
+ }
> tmp_function(34, 15, third.arg = 11, fou = 99)
```

```
[1] 34 15 11 99
```

Positional matching of arguments is convenient, but should be used carefully, and probably limited to the first few and most commonly used arguments in a function. Partial matching also has some pitfalls

```
> tmp_function <- function(first.arg, fourth.arg) {
+   return(c(first.arg, fourth.arg))
+ }
> tmp_function(1, f = 2)
```

```
Error in tmp_function(1, f = 2): argument 2 matches multiple formal arguments
```

A partially specified argument must unambiguously match exactly one argument.

6.3.2 The ... Argument

In defining a function, a special argument denoted by ... can be used. Sometimes this is called the "ellipsis" argument, sometimes the "three dot" argument, sometimes the "dot dot dot" argument, etc. The R language definition https://cran.r-project.org/doc/manuals/r-release/R-lang.html describes the argument in this way:

> The special type of argument '...' can contain any number of supplied arguments. It is used for a variety of purposes. It allows you to write a function that takes an arbitrary number of arguments. It can be used to absorb some arguments into an intermediate function which can then be extracted by functions called subsequently.

Consider for example the `sum()` function.

```
> sum(1:5)
```

```
[1] 15
```

```
> sum(1:5, c(3, 4, 90))
```

```
[1] 112
```

```
> sum(1, 2, 3, c(3, 4, 90), 1:5)
```

```
[1] 118
```

Think about writing such a function. There is no way to predict in advance the number of arguments a user might specify. So the function is defined with `...` as the first argument:

```
> sum
```

```
function (..., na.rm = FALSE)  .Primitive("sum")
```

This is true of many commonly-used functions in R such as `c()` among others.

Next, consider a function which calls another function in its body. For example, suppose that a collaborator always supplies comma delimited files which have five lines of description, followed by a line containing variable names, followed by the data. You are tired of having to specify `skip = 5`, `header = TRUE`, and `sep = ","` to `read.table()` and want to create a function `my.read()` which uses these as defaults.

```
> my.read <- function(file, header = TRUE, sep = ",", skip = 5,
+   ...) {
+   read.table(file = file, header = header, sep = sep,
+     skip = skip, ...)
+ }
```

The `...` in the definition of `my.read()` allows the user to specify other arguments, for example, `stringsAsFactors = FALSE`. These will be passed on to the `read.table()` function. In fact, that is how `read.csv()` is defined.

```
> read.csv
```

```
function (file, header = TRUE, sep = ",", quote = "\"", dec = ".",
    fill = TRUE, comment.char = "", ...)
read.table(file = file, header = header, sep = sep, quote = quote,
```

```
    dec = dec, fill = fill, comment.char = comment.char, ...)
<bytecode: 0x564a51170d60>
<environment: namespace:utils>
```

7

Working with Data Sources

Bringing data into R, exporting data from R in a form that is readable by other software, cleaning and reshaping data, and other data manipulation tasks are an important and often overlooked component of data science. The book Spector (2008), while a few years old, is still an excellent reference for data-related issues. And the *R Data Import/Export* manual, available online at https://cran.r-project.org/doc/manuals/r-release/R-data.html, is an up-to-date (and free) reference on importing a wide variety of datasets into R and on exporting data in various forms.

7.1 Reading Data into R

Data come in a dizzying variety of forms. It might be in a proprietary format such as a .xlsx Excel file, a .sav SPSS file, or a .mtw Minitab file. It might be structured using a relational model[1], for example, the USDA Forest Service Forest Inventory and Analysis database[2]. It might be a data-interchange format such as JSON[3] (JavaScript Object Notation), or a markup language format such as XML[4] (Extensible Markup Language), perhaps with specialized standards for describing ecological information, see, e.g., EML[5] (Ecological Metadata Language). Both XML and EML are common data metadata formats (i.e., data that provides information about other data). Fortunately many datasets are (or can be) saved as plain text files, and most software can both read and write such files, so our initial focus will be on reading plain text files into R and saving data from R in plain text format. RStudio provides a handy data import cheat sheet[6] for many of the read functions detailed in this section.

[1] https://en.wikipedia.org/wiki/Relational_model
[2] https://www.fia.fs.fed.us/tools-data/
[3] http://www.json.org/
[4] https://en.wikipedia.org/wiki/XML
[5] https://en.wikipedia.org/wiki/Ecological_Metadata_Language
[6] http://www.rstudio.com/resources/cheatsheets

The `foreign` R package provides functions to directly read data saved in some of the proprietary formats into R, which is sometimes unavoidable, but if possible it is good to save data from another package as plain text and then read this plain text file into R.

The function `read.table()` and its offshoots such as `read.csv()` are used to read in rectangular data from a text file. For example, the file `FEF-trees.csv` described in Section 1.1 contains biomass information for trees measured on plots within different watersheds across years. Below are the first seven rows and columns of that file:

```
watershed,year,plot,species,dbh_in,height_ft,stem_green_kg
3,1991,29,Acer rubrum,6,48,92.2
3,1991,33,Acer rubrum,6.9,48,102.3
3,1991,35,Acer rubrum,6.4,48,124.4
3,1991,39,Acer rubrum,6.5,49,91.7
3,1991,44,Acer rubrum,7.2,51,186.2
3,1992,26,Acer rubrum,3.1,40,20.8
```

As is evident, the first line of the file contains the names of the variables, separated (delimited) by commas. Each subsequent line contains information about each tree's location (watershed code), measurement year, plot (within watershed), species, DBH, height, and weight of various components of the tree (e.g., `stem_green_kg` is the green weight of the stem in kilograms).[7]

This file is accessible at http://blue.for.msu.edu/FOR472/data/FEF-trees.csv. The `read.table()` function is used to read these data into an R data frame.

```
> fef.file <- "http://blue.for.msu.edu/FOR472/data/FEF-trees.csv"
> fef.trees <- read.table(file = fef.file, header = TRUE,
+   sep = ",")
> head(fef.trees[, 1:6])
```

```
  watershed year plot     species dbh_in height_ft
1         3 1991   29 Acer rubrum    6.0        48
2         3 1991   33 Acer rubrum    6.9        48
3         3 1991   35 Acer rubrum    6.4        48
4         3 1991   39 Acer rubrum    6.5        49
5         3 1991   44 Acer rubrum    7.2        51
6         3 1992   26 Acer rubrum    3.1        40
```

The arguments used in this call to `read.table()` include:

[7] The metadata for the FEF are at http://www.fs.usda.gov/rds/archive/Product/RDS-2016-0016. For convenience I renamed the data file from the original `felled_tree_biomass.csv` to `FEF-trees.csv`.

- `file = fef.file` tells R the location of the file. In this case the string `http://blue.for.msu.edu/FOR472/data/FEF-trees.csv` giving the location is rather long, so it was first assigned to the object `fef.file`.
- `header = TRUE` tells R the first line of the file gives the names of the variables.
- `sep = ","` tells R that a comma separates the fields in the file.

The function `read.csv()` is the same as `read.table()` except the default separator is a comma, whereas the default separator for `read.table()` is whitespace.

The file `FEF-trees.tsv` contains the same data, except a tab is used in place of a comma to separate fields. The only change needed to read in the data in this file is in the `sep` argument (and of course the `file` argument, since the data are stored in a different file):

```
> fef.file <- "http://blue.for.msu.edu/FOR472/data/FEF-trees.tsv"
> fef.trees <- read.table(file = fef.file, header = TRUE,
+   sep = "\t")
> head(fef.trees[, 1:6])
```

```
  watershed year plot     species dbh_in height_ft
1         3 1991   29 Acer rubrum    6.0        48
2         3 1991   33 Acer rubrum    6.9        48
3         3 1991   35 Acer rubrum    6.4        48
4         3 1991   39 Acer rubrum    6.5        49
5         3 1991   44 Acer rubrum    7.2        51
6         3 1992   26 Acer rubrum    3.1        40
```

File extensions, e.g., `.csv` or `.tsv`, are naming conventions only and are there to remind us how the columns are delimited, i.e., they have no influence on R's file read functions.

A third file, `FEF-trees.txt`[8], contains the same data, but also contains a few lines of explanatory text above the names of the variables. It also uses whitespace rather than a comma or a tab as a delimiter. Here are the first several lines of the file and six columns of data.

```
This file includes felled tree biomass by tree component for hardwood
species sampled on the Fernow Experimental Forest (FEF), West Virginia.
A total of 88 trees were sampled from plots within two watersheds.

"watershed" "year" "plot" "species" "dbh_in" "height_ft" "stem_green_kg"
3 1991 29 "Acer rubrum" 6 48 92.2
3 1991 33 "Acer rubrum" 6.9 48 102.3
```

[8] http://blue.for.msu.edu/FOR472/data/FEF-trees.txt

```
3 1991 35 "Acer rubrum" 6.4 48 124.4
3 1991 39 "Acer rubrum" 6.5 49 91.7
3 1991 44 "Acer rubrum" 7.2 51 186.2
3 1992 26 "Acer rubrum" 3.1 40 20.8
```

Notice that in this file column (variable) names are put inside of quotation marks. Also, variable values that are characters are also quoted. This is necessary because character strings could include whitespace, and hence R would assume these are column delimiters.

To read in this file we need to tell R to skip the first four lines above the header and also that whitespace is the delimiter. The `skip` argument handles the first, and the `sep` argument the second.

```
> fef.file <- "http://blue.for.msu.edu/FOR472/data/FEF-trees.txt"
> fef.trees <- read.table(file = fef.file, header = TRUE,
+   sep = " ", skip = 4)
> head(fef.trees[, 1:6])
```

```
  watershed year plot     species dbh_in height_ft
1         3 1991   29 Acer rubrum    6.0        48
2         3 1991   33 Acer rubrum    6.9        48
3         3 1991   35 Acer rubrum    6.4        48
4         3 1991   39 Acer rubrum    6.5        49
5         3 1991   44 Acer rubrum    7.2        51
6         3 1992   26 Acer rubrum    3.1        40
```

For fun, see what happens when the `skip` argument is left out of the `read.table` call.

7.2 Reading Data with missing observations

Missing data are represented in many ways. Sometimes missing data are just that, i.e., the place where they should be in the file is blank. Other times specific numbers such as -9999 or specific symbols are used. The `read.table()` function has an argument `na.string` which allows the user to specify how missing data are indicated in the source file.

The site http://www.wunderground.com/history/ makes weather data available for locations around the world, and for dates going back to 1945. The file `WeatherKLAN2014.csv` contains weather data for Lansing, Michigan for the year 2014. Here are the first few lines of that file:

```
EST,Max TemperatureF,Min TemperatureF, Events
1/1/14,14,9,Snow
1/2/14,13,-3,Snow
1/3/14,13,-11,Snow
1/4/14,31,13,Snow
1/5/14,29,16,Fog-Snow
1/6/14,16,-12,Fog-Snow
1/7/14,2,-13,Snow
1/8/14,17,-1,Snow
1/9/14,21,2,Snow
1/10/14,39,21,Fog-Rain-Snow
1/11/14,41,32,Fog-Rain
1/12/14,39,31,
```

Look at the last line, and notice that instead of an event such as `Snow` or `Fog-Snow` there is nothing after the comma. This observation is missing, but rather than using an explicit code such as `NA`, the site just leaves that entry blank. To read these data into R we will supply the argument `na.string = ""` which tells R the file indicates missing data by leaving the appropriate entry blank.

```
> u.weather <- "http://blue.for.msu.edu/FOR875/data/WeatherKLAN2014.csv"
> WeatherKLAN2014 <- read.csv(u.weather, header=TRUE,
+                             stringsAsFactors = FALSE, na.string = "")
> WeatherKLAN2014[1:15,]
```

```
       EST Max.TemperatureF Min.TemperatureF
1   1/1/14               14                9
2   1/2/14               13               -3
3   1/3/14               13              -11
4   1/4/14               31               13
5   1/5/14               29               16
6   1/6/14               16              -12
7   1/7/14                2              -13
8   1/8/14               17               -1
9   1/9/14               21                2
10 1/10/14               39               21
11 1/11/14               41               32
12 1/12/14               39               31
13 1/13/14               44               34
14 1/14/14               37               26
15 1/15/14               27               18
          Events
1           Snow
2           Snow
```

```
3           Snow
4           Snow
5       Fog-Snow
6       Fog-Snow
7           Snow
8           Snow
9           Snow
10 Fog-Rain-Snow
11      Fog-Rain
12          <NA>
13          Rain
14     Rain-Snow
15          Snow
```

Also, in the code above, notice I use `stringsAsFactors = FALSE` to prevent the character variables to be converted to factors, which R does by default (which can be quite annoying).

8

Spatial Data Visualization and Analysis

8.1 Overview

Recall, a data structure is a format for organizing and storing data. The structure is designed so that data can be accessed and worked with in specific ways. Statistical software and programming languages have methods (or functions) designed to operate on different kinds of data structures.

This chapter focuses on spatial data structures and some of the R functions that work with these data. Spatial data comprise values associated with locations, such as temperature data at a given latitude, longitude, and perhaps elevation. Spatial data are typically organized into **vector** or **raster** data types. (See Figure 8.1).

- Vector data represent features such as discrete points, lines, and polygons.
- Raster data represent surfaces as a rectangular matrix of square cells or pixels.

Whether or not you use vector or raster data depends on the type of problem, the type of maps you need, and the data source. Each data structure has strengths and weaknesses in terms of functionality and representation. As you gain more experience working with spatial data, you will be able to determine which structure to use for a particular application.

There is a large set of R packages available for working with spatial (and space-time) data. These packages are described in the Cran Task View: Analysis of Spatial Data[1]. The CRAN task view attempts to organize the various packages into categories, such as *Handling spatial data*, *Reading and writing spatial data*, *Visualization*, and *Disease mapping and areal data analysis*, so users can quickly identify package options given their project needs.

Exploring the extent of the excellent spatial data tools available in R is beyond the scope of this book. Rather, we would point you to subject texts like *Applied Spatial Data Analysis with R* by Bivand et al. (2013) (available for free via the MSU library system), and numerous online tutorials on pretty much

[1] https://CRAN.R-project.org/view=Spatial

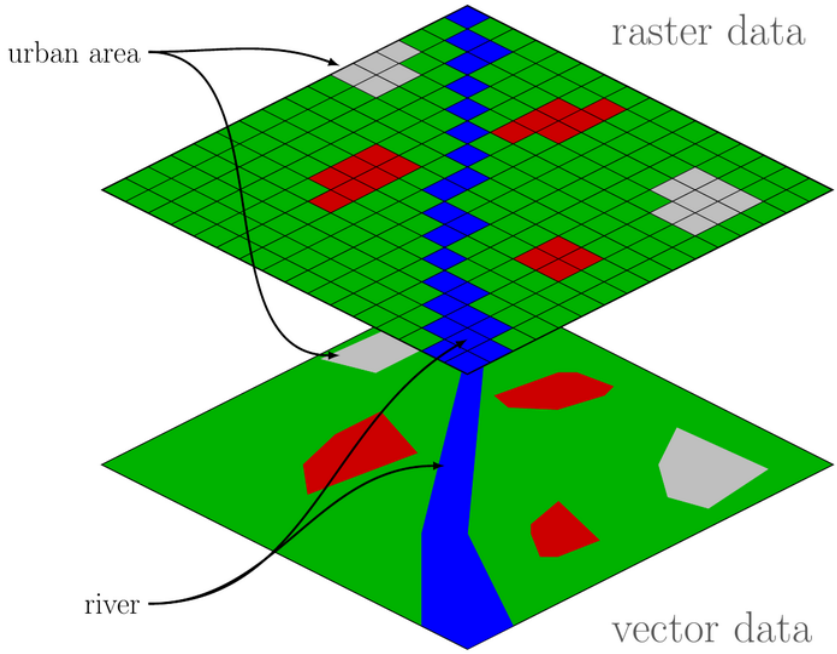


FIGURE 8.1: Raster/Vector Comparison @imageRaster

any aspect of spatial data analysis with R. These tools make R a full-blown Geographic Information System[2] (GIS) capable of spatial data manipulation and analysis on par with commercial GIS systems such as ESRI's ArcGIS[3].

8.1.1 Some Spatial Data Packages

This chapter will focus on a few R packages for manipulating and visualizing spatial data. Specifically we will touch on the following packages

- `sp`: spatial data structures and methods
- `rgdal`: interface to the C/C++ spatial data Geospatial Data Abstraction Library
- `ggmap`: extends `ggplot2` language to handle spatial data
- `leaflet`: generates dynamic online maps

[2] https://en.wikipedia.org/wiki/Geographic_information_system
[3] http://www.esri.com/arcgis/about-arcgis

8.2 Motivating Data

We motivate the topics introduced in this chapter using some forestry data from the Penobscot Experimental Forest[4] (PEF) located in Maine (which you've previously seen throughout the course). The PEF is a long-term experimental forest that is used to understand the effects of silviculture (i.e., science of tending trees) treatments on forest growth and composition. The PEF is divided into non-overlapping management units that receive different harvesting treatments. Within each management unit is a series of observation point locations (called forest inventory plots) where forest variables have been measured. Ultimately, we want to summarize the inventory plots measurements by management unit and map the results.

8.3 Reading Spatial Data into R

Spatial data come in a variety of file formats. Examples of popular vector file formats for points, lines, and polygons, include ESRI's shapefile[5] and open standard GeoJSON[6]. Common raster file formats include GeoTIFF[7] and netCDF[8][9].

The `rgdal` function `readOGR` will read a large variety of vector data file formats (there is also a `writeOGR()` for writing vector data files). Raster data file formats can be read using the `rgdal` function `readGDAL` (yup, also a `writeGDAL()`) or read functions in the `raster` package. All of these functions automatically cast the data into an appropriate R spatial data object (i.e., data structure), which are defined in the `sp` or `raster` packages. Table 8.1 provides an abbreviated list of these R spatial objects[10]. The *Without attributes* column gives the `sp` package's spatial data object classes for points, lines, polygons, and raster pixels that do not have data associated with the

[4] https://www.nrs.fs.fed.us/ef/locations/me/penobscot%7D%7BPenobscot%20Experimental%20Forest

[5] https://en.wikipedia.org/wiki/Shapefile

[6] https://en.wikipedia.org/wiki/GeoJSON

[7] https://en.wikipedia.org/wiki/GeoTIFF

[8] https://en.wikipedia.org/wiki/NetCDF

[9] A longer list of spatial data file formats is available at https://en.wikipedia.org/wiki/GIS_file_formats.

[10] A more complete list of the `sp` package's spatial data classes and methods is detailed in the package's vignette https://cran.r-project.org/web/packages/sp/vignettes/intro_sp.pdf.

TABLE 8.1: An abbreviated list of 'sp' and 'raster' data objects and associated classes for the fundamental spatial data types

	Without Attributes	With Attributes
Polygons	SpatialPolygons	SpatialPolygonsDataFrame
Points	SpatialPoints	SpatialPointsDataFrame
Lines	SpatialLines	SpatialLinesDataFrame
Raster	SpatialGrid	SpatialGridDataFrame
Raster	SpatialPixels	SpatialPixelsDataFrame
Raster		RasterLayer
Raster		RasterBrick
Raster		RasterStack

spatial objects (i.e., without attributes in GIS speak). `DataFrame` is appended to the object class name once data, in the form of variables, are added to the spatial object.

You can create your own spatial data objects in R. Below, for example, we create a `SpatialPoints` object consisting of four points. Then add some data to the points to make it a `SpatialPointsDataFrame`.

```
> library(sp)
> library(dplyr)
>
> x <- c(3, 2, 5, 6)
> y <- c(2, 5, 6, 7)
>
> coords <- cbind(x, y)
>
> sp.pts <- SpatialPoints(coords)
>
> class(sp.pts)
```

```
[1] "SpatialPoints"
attr(,"package")
[1] "sp"
```

```
> some.data <- data.frame(var.1 = c("a", "b", "c", "d"), var.2 = 1:4)
>
> sp.pts.df <- SpatialPointsDataFrame(sp.pts, some.data)
>
> class(sp.pts.df)
```

```
[1] "SpatialPointsDataFrame"
attr(,"package")
[1] "sp"
```

If, for example, you already have a data frame that includes the spatial coordinate columns and other variables, then you can promote it to a `SpatialPointsDataFrame` by indicating which columns contain point coordinates. You can extract or access the data frame associated with the spatial object using `@data`. You can also access individual variables directly from the spatial object using `$` or by name or column number to the right of the comma in `[,]` (analogues to accessing variables in a data frame).

```
> df <- data.frame(x = c(3, 2, 5, 6), y = c(2, 5, 6, 7), var.1 = c("a",
+   "b", "c", "d"), var.2 = 1:4)
> class(df)
```

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

```
> # promote to a SpatialPointsDataFrame
> coordinates(df) <- ~x + y
>
> class(df)
```

```
[1] "SpatialPointsDataFrame"
attr(,"package")
[1] "sp"
```

```
> # access entire data frame
> df@data
```

```
  var.1 var.2
1     a     1
2     b     2
3     c     3
4     d     4
```

```
> class(df@data)
```

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

```
> # access columns directly
> df$var.1
```

```
[1] a b c d
Levels: a b c d
```

```
> df[, c("var.1", "var.2")]
```

```
  coordinates var.1 var.2
1      (3, 2)     a     1
2      (2, 5)     b     2
3      (5, 6)     c     3
4      (6, 7)     d     4
```

```
> df[, 2]
```

```
  coordinates var.2
1      (3, 2)     1
2      (2, 5)     2
3      (5, 6)     3
4      (6, 7)     4
```

```
> # get the bounding box
> bbox(df)
```

```
  min max
x   2   6
y   2   7
```

Here, the data frame `df` is promoted to a `SpatialPointsDataFrame` by indicating the column names that hold the longitude and latitude (i.e., `x` and `y` respectively) using the `coordinates` function. Here too, the `@data` is used to retrieve the data frame associated with the points. We also illustrate how variables can be accessed directly from the spatial object. The `bbox` function is used to return the bounding box that is defined by the spatial extent of the point coordinates. The other spatial objects noted in Table 8.1 can be created, and their data accessed, in a similar way[11].

More than often we find ourselves reading existing spatial data files into R. The code below uses the `downloader` package to download all of the PEF data we'll use in this chapter. The data are compressed in a single zip file, which is then extracted into the working directory using the `unzip` function. A look into the PEF directory using `list.files` shows nine files[12]. Those named `MU-bounds.*` comprise the shapefile that holds the PEF's management

[11] This cheatsheet(www.maths.lancs.ac.uk/~rowlings/Teaching/UseR2012/cheatsheet.html) written by Barry Rowlingson is a useful reference www.maths.lancs.ac.uk/~rowlings/Teaching/UseR2012/cheatsheet.html

[12] The `list.files` function does not read data into R; it simply prints the contents of a directory.

unit boundaries in the form of polygons. Like other spatial data file formats, shapefiles are made up of several different files (with different file extensions) that are linked together to form a spatial data object. The `plots.csv` file holds the spatial coordinates and other information about the PEF's forest inventory plots. The `roads.*` shapefile holds roads and trails in and around the PEF.

```
> library(downloader)
>
> download("http://blue.for.msu.edu/FOR875/data/PEF.zip",
+          destfile="./PEF.zip", mode="wb")
>
> unzip("PEF.zip", exdir = ".")
>
> list.files("PEF")
```

```
 [1] "MU-bounds.dbf" "MU-bounds.prj" "MU-bounds.qpj"
 [4] "MU-bounds.shp" "MU-bounds.shx" "plots.csv"
 [7] "plots.dbf"     "plots.prj"     "plots.qpj"
[10] "plots.shp"     "plots.shx"     "roads.dbf"
[13] "roads.prj"     "roads.shp"     "roads.shx"
```

Next we read the MU-bounds shapefile into R using `readOGR()`[13] and explore the resulting `mu` object. Notice that when we read a shapefile into R, we do not include a file extension with the shapefile name because a shapefile is always composed of multiple files.

```
> library(rgdal)
```

```
rgdal: version: 1.3-6, (SVN revision 773)
 Geospatial Data Abstraction Library extensions to R successfully loaded
 Loaded GDAL runtime: GDAL 2.2.3, released 2017/11/20
 Path to GDAL shared files: /usr/share/gdal/2.2
 GDAL binary built with GEOS: TRUE
 Loaded PROJ.4 runtime: Rel. 4.9.3, 15 August 2016, [PJ_VERSION: 493]
 Path to PROJ.4 shared files: (autodetected)
 Linking to sp version: 1.3-1
```

```
> mu <- readOGR("PEF", "MU-bounds")
```

```
OGR data source with driver: ESRI Shapefile
```

[13] The authors of the `rgdal` library decided to have some information about the version of GDAL and other software specifics be printed when the library is loaded. Don't let it distract you.

```
Source: "/home/jeffdoser/Dropbox/teaching/for472/text/book/PEF", layer: "MU-bounds"
with 40 features
It has 1 fields
```

When called, the `readOGR` function provides a bit of information about the object being read in. Here, we see that it read the MU-bounds shapefile from PEF directory and the shapefile had 40 features (i.e., polygons) and 1 field (i.e., field is synonymous with column or variable in the data frame).

You can think of the resulting `mu` object as a data frame where each row corresponds to a polygon and each column holds information about the polygon[14]. More specifically, the `mu` object is a `SpatialPolygonsDataFrame`.

```
> class(mu)
```

```
[1] "SpatialPolygonsDataFrame"
attr(,"package")
[1] "sp"
```

As illustrated using the made-up point data in the example above, you can access the data frame associated with the polygons using `@data`.

```
> class(mu@data)
```

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

```
> dim(mu@data)
```

```
[1] 40  1
```

```
> head(mu@data)
```

```
  mu_id
0   C15
1   C17
2   C16
3   C27
4   U18
5   U31
```

Above, a call to `class()` confirms we have accessed the data frame, `dim()` shows there are 40 rows (one row for each polygon) and one column, and `head()` shows the first six values of the column named `mu_id`. The `mu_id`

[14] Much of the actual spatial information is hidden from you in other parts of the data structure, but is available if you ask nicely for it (see subsequent sections).

values are unique identifiers for each management unit polygon across the PEF.

8.4 Coordinate Reference Systems

One of the more challenging aspects of working with spatial data is getting used to the idea of a coordinate reference system. A *coordinate reference system* (CRS) is a system that uses one or more numbers, or coordinates, to uniquely determine the position of a point or other geometric element (e.g., line, polygon, raster). For spatial data, there are two common coordinate systems:

1. Spherical coordinate system, such as latitude-longitude, often referred to as a *geographic coordinate system*.
2. Projected coordinate system based on a map projection, which is a systematic transformation of the latitudes and longitudes that aims to minimize distortion occurring from projecting maps of the earth's spherical surface onto a two-dimensional Cartesian coordinate plane. Projected coordinate systems are sometimes referred to as *map projections*.

There are numerous map projections[15]. One of the more frustrating parts of working with spatial data is that it seems like each data source you find offers its data in a different map projection and hence you spend a great deal of time *reprojecting* (i.e., transforming from one CRS to another) data into a common CRS such that they overlay correctly. Reprojecting is accomplished using the `sp` package's `spTransform` function as demonstrated in Section 8.5.

In R, a spatial object's CRS is accessed via the `sp` package `proj4string` function. The code below shows the current projection of `mu`.

```
> proj4string(mu)
```

```
[1] "+proj=utm +zone=19 +datum=NAD83 +units=m +no_defs +ellps=GRS80 +towgs84=0,0,0"
```

The cryptic looking string returned by `proj4string()` is a set of directives understood by the proj.4[16] C library, which is part of `sp`, and used to map

[15] See partial list of map projections at https://en.wikipedia.org/wiki/List/_of/_map/_projections. See a humorous discussion of map projections at http://brilliantmaps.com/xkcd/.

[16] http://proj4.org/

geographic longitude and latitude coordinates into the projected Cartesian coordinates. This CRS tells us the `mu` object is in Universal Transverse Mercator (UTM)[17] zone 19 coordinate system.[18]

8.5 Illustration using ggmap

Let's begin by making a map of PEF management unit boundaries over top of a satellite image using the `ggmap` package. Given an address, location, or bounding box, the `ggmap` package's `get_map` function will query Google Maps, OpenStreetMap, Stamen Maps, or Naver Map servers for a user-specified map type. The `get_map` function requires the location or bounding box coordinates be in a geographic coordinate system (i.e., latitude-longitude). This means we need to reproject `mu` from UTM zone 19 to latitude-longitude geographic coordinates, which is defined by the `'"proj=longlat +datum=WGS84"'` proj.4 string. As seen below, the first argument in `spTransform` function is the spatial object to reproject and the second argument is a CRS object created by passing a proj.4 string into the `CRS` function.

```
> mu <- spTransform(mu, CRS("+proj=longlat +datum=WGS84"))
> proj4string(mu)
```

```
[1] "+proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0"
```

Unfortunately, we cannot just feed the `SpatialPolygonsDataFrame` `mu` into `ggplot` (perhaps some day soon this will possible). Rather, we need to first convert the `SpatialPolygonsDataFrame` into a specially formatted data frame using the `fortify` function that is part of the `ggplot2` package[19]. The `fortify` function will also need a unique identifier for each polygon specified using the `region` argument, which for `mu` is the `mu_id`.

```
> library(ggmap)
```

```
Google Maps API Terms of Service: http://developers.google.com/maps/terms.
```

[17] https://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system

[18] If you start dealing with a lot of spatial data and reprojecting, http://spatialreference.org is an excellent resources for finding and specifying coordinate reference systems.

[19] `ggmap` depends on `ggplot2` so `ggplot2` will be automatically loaded when you call `library(ggmap)`.

```
Please cite ggmap if you use it: see citation("ggmap") for details.
```

```
> mu.f <- fortify(mu, region = "mu_id")
> head(mu.f)
```

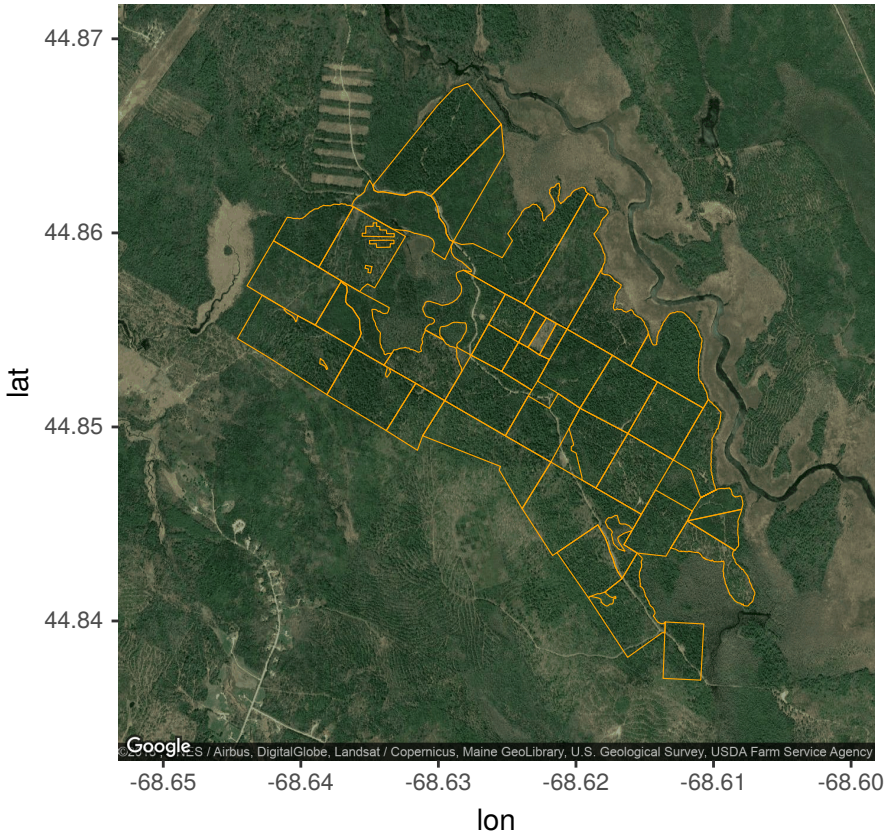
```
    long   lat order  hole piece  id group
1 -68.62 44.86     1 FALSE     1 C12 C12.1
2 -68.62 44.86     2 FALSE     1 C12 C12.1
3 -68.62 44.86     3 FALSE     1 C12 C12.1
4 -68.62 44.86     4 FALSE     1 C12 C12.1
5 -68.62 44.86     5 FALSE     1 C12 C12.1
6 -68.62 44.86     6 FALSE     1 C12 C12.1
```

Notice the `id` column in the fortified version of `mu` holds each polygon's `mu_id` value (this will be important later when we link data to the polygons).

Next, we query the satellite imagery used to underlay the management units (we'll generally refer to this underlying map as the basemap). As of October 2018, Google now requires you to set up a Google API account in order to run the following maps. This is free, but it does require a credit card to obtain the API Key that is required to make the `ggmap` package work. Here I provide you with an API key for a project I created for this class that should allow you to run the following function if you desire. If you are interested in obtaining your own API key, see the page here[20] for learning about how to use Google maps web services.

```
> register_google(key = "AIzaSyBPAwSY5x8vQqlnG-QwiCAWQW12U3CTLZY")
> mu.bbox <- bbox(mu)
>
> basemap <- get_map(location=mu.bbox, zoom = 14, maptype="satellite")
>
> ggmap(basemap) +
+     geom_polygon(data=mu.f, aes(x = long, y = lat, group=group),
+                  fill=NA, size=0.2, color="orange")
```

[20] https://developers.google.com/maps/documentation/geocoding/get-api-key



Looks pretty good! Take a look at the `get_map` function manual page and try different options for `maptype` (e.g., `maptype="terrain"`).

Next we'll add the forest inventory plots to the map. Begin by reading in the PEF forest inventory plot data held in "plots.csv". Recall, foresters have measured forest variables at a set of locations (i.e., inventory plots) within each management unit. The following statement reads these data and displays the resulting data frame structure.

```
> plots <- read.csv("PEF/plots.csv", stringsAsFactors = FALSE)
> str(plots)
```

```
'data.frame':   451 obs. of  8 variables:
 $ mu_id          : chr  "U10" "U10" "U10" "U10" ...
 $ plot           : int  11 13 21 22 23 24 31 32 33 34 ...
 $ easting        : num  529699 529777 529774 529814 529850 ...
 $ northing       : num  4966333 4966471 4966265 4966336 4966402 ...
 $ biomass_mg_ha  : num  96.3 115.7 121.6 72 122.3 ...
```

```
 $ stems_ha        : int  5453 2629 3385 7742 7980 10047 5039 5831 2505 7325 ...
 $ diameter_cm     : num  4.8 6.9 6.1 3.1 4.7 1.6 4.1 5.2 5.7 3.3 ...
 $ basal_area_m2_ha: num  22 23.2 23 16.1 29.2 19.1 14.1 27.4 21.6 15 ...
```

In `plots` each row is a forest inventory plot and columns are:

- `mu_id` identifies the management unit within which the plot is located
- `plot` unique plot number within the management unit
- `easting` longitude coordinate in UTM zone 19
- `northing` latitude coordinate in UTM zone 19
- `biomass_mg_ha` tree biomass in metric ton (per hectare basis)
- `stocking_stems_ha` number of tree (per hectare basis)
- `diameter_cm` average tree diameter measured 130 cm up the tree trunk
- `basal_area_m2_ha` total cross-sectional area at 130 cm up the tree trunk (per hectare basis)

There is nothing inherently spatial about this data structure—it is simply a data frame. We make `plots` into a spatial object by identifying which columns hold the coordinates. This is done below using the `coordinates` function, which promotes the `plots` data frame to a `SpatialPointsDataFrame`.

```
> coordinates(plots) <- ~easting+northing
>
> class(plots)
```

```
[1] "SpatialPointsDataFrame"
attr(,"package")
[1] "sp"
```

Although `plots` is now a `SpatialPointsDataFrame`, it does not know to which CRS the coordinates belong; hence, the `NA` when `proj4string(plots)` is called below. As noted in the `plots` file description above, `easting` and `northing` are in UTM zone 19. This CRS is set using the second call to `proj4string` below.

```
> proj4string(plots)
```

```
[1] NA
```

```
> proj4string(plots) <- CRS("+proj=utm +zone=19 +datum=NAD83 +units=m
+                           +no_defs +ellps=GRS80 +towgs84=0,0,0")
```

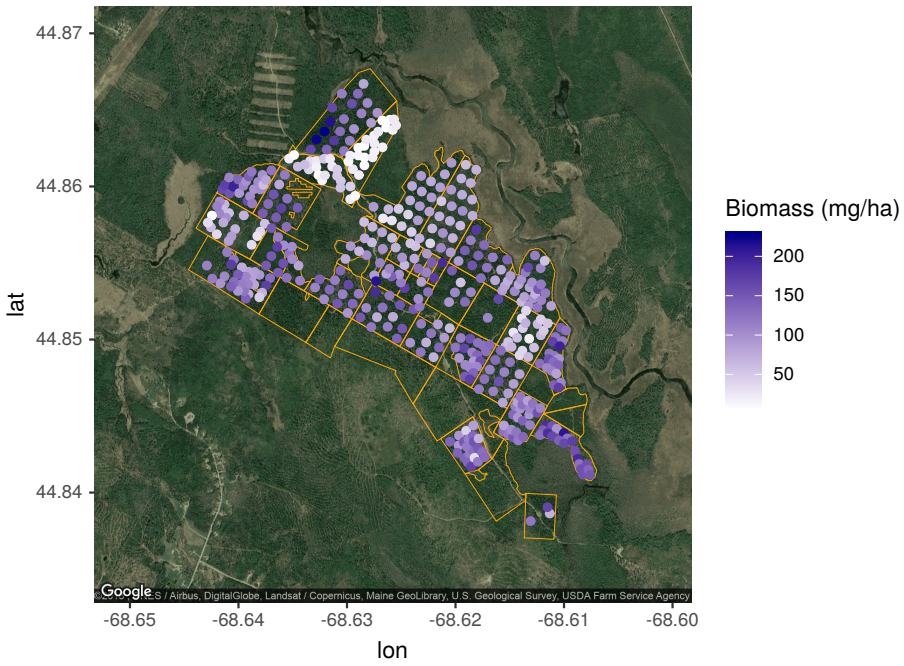
Now let's reproject `plots` to share a common CRS with `mu`

```
> plots <- spTransform(plots, CRS("+proj=longlat +datum=WGS84"))
```

Note, because `mu` is already in the projection we want for `plots`, we could have replaced the second argument in the `spTransform` call above with `proj4string(mu)` and saved some typing.

We're now ready to add the forest inventory plots to the existing basemap with management units. Specifically, let's map the `biomass_mg_ha` variable to show changes in biomass across the forest. No need to fortify `plots`, `ggplot` is happy to take `geom_point`'s `data` argument as a data frame (although we do need to convert `plots` from a `SpatialPointsDataFrame` to a data frame using the `as.data.frame` function). Check out the `scale_color_gradient` function in your favorite `ggplot2` reference to understand how the color scale is set.

```
> ggmap(basemap) +
+     geom_polygon(data=mu.f, aes(x = long, y = lat, group=group),
+                  fill=NA, size=0.2, color="orange") +
+     geom_point(data=as.data.frame(plots),
+                aes(x = easting, y = northing, color=biomass_mg_ha)) +
+     scale_color_gradient(low="white", high="darkblue") +
+     labs(color = "Biomass (mg/ha)")
```



There is something subtle and easy to miss in the code above. Notice the `aes` function arguments in `geom_points` take geographic longitude and latitude, `x` and `y` respectively, from the `points` data frame (but recall `easting` and `northing` were in UTM zone 19). This works because we applied `spTransform` to reproject the `points` `SpatialPointsDataFrame` to geographic coordinates. `sp` then replaces the values in `easting` and `northing` columns with the reprojected coordinate values when converting a `SpatialPointsDataFrame` to a data frame via `as.data.frame()`.

Foresters use the inventory plot measurements to estimate forest variables within management units, e.g., the average or total management unit biomass. Next we'll make a plot with management unit polygons colored by average `biomass_mg_ha`.

```
> mu.bio <- plots@data %>% group_by(mu_id) %>%
+     summarize(biomass_mu = mean(biomass_mg_ha))
> print(mu.bio)
```

```
# A tibble: 33 x 2
   mu_id biomass_mu
   <chr>      <dbl>
 1 C12        124.
 2 C15         49.9
 3 C16        128.
 4 C17        112.
 5 C20        121.
 6 C21        134.
 7 C22         65.2
 8 C23A       108.
 9 C23B       153.
10 C24        126.
# ... with 23 more rows
```

Recall from Section 5.5 this one-liner can be read as "get the data frame from `plots`'s `SpatialPointsDataFrame` *then* group by management unit *then* make a new variable called `biomass_mu` that is the average of `biomass_mg_ha` and assign it to the `mu.bio` tibble."

The management unit specific `biomass_mu` can now be joined to the `mu` polygons using the common `mu_id` value. Remember when we created the fortified version of `mu` called `mu.f`? The `fortify` function `region` argument was `mu_id` which is the `id` variable in the resulting `mu.f`. This `id` variable in `mu.f` can be linked to the `mu_id` variable in `mu.bio` using `dplyr`'s `left_join` function as illustrated below.

```
> head(mu.f, n = 2)
```

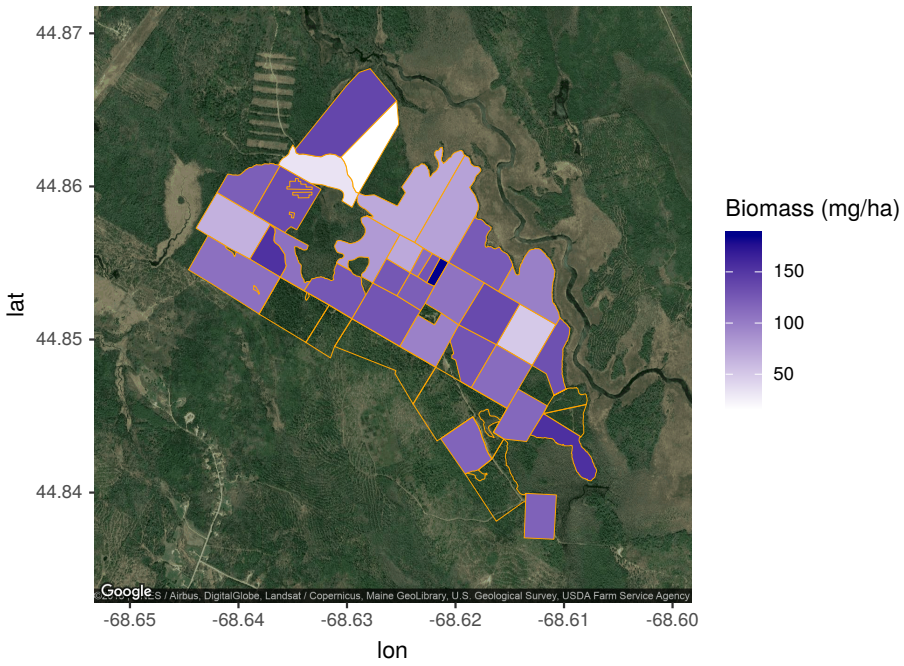
```
    long   lat order  hole piece  id group
1 -68.62 44.86     1 FALSE     1 C12 C12.1
2 -68.62 44.86     2 FALSE     1 C12 C12.1
```

```
> mu.f <- left_join(mu.f, mu.bio, by = c(id = "mu_id"))
>
> head(mu.f, n = 2)
```

```
    long   lat order  hole piece  id group biomass_mu
1 -68.62 44.86     1 FALSE     1 C12 C12.1      123.7
2 -68.62 44.86     2 FALSE     1 C12 C12.1      123.7
```

The calls to `head()` show the first few rows of `mu.f` pre- and post-join. After the join, `mu.f` includes `biomass_mu`, which is used used below for `geom_polygon`'s `fill` argument to color the polygons accordingly.

```
> ggmap(basemap) +
+     geom_polygon(data=mu.f, aes(x = long, y = lat,
+                                 group=group, fill=biomass_mu),
+                  size=0.2, color="orange") +
+     scale_fill_gradient(low="white", high="darkblue",
+                         na.value="transparent") +
+     labs(fill="Biomass (mg/ha)")
```



Let's add the roads and some more descriptive labels as a finishing touch. The roads data include a variable called `type` that identifies the road type. To color roads by type in the map, we need to join the `roads` data frame with the fortified roads `roads.f` using the common variable `id` as a road segment specific identifier. Then `geom_path`'s `color` argument gets this `type` variable as a factor to create road-specific color. The default coloring of the roads blends in too much with the polygon colors, so we manually set the road colors using the `scale_color_brewer` function. The `palette` argument in this function accepts a set of key words, e.g., `"Dark2"`, that specify sets of diverging colors chosen to make map object difference optimally distinct (see, the manual page for `scale_color_brewer`, http://colorbrewer2.org, and blog here[21].)[22]

```
> roads <- readOGR("PEF", "roads")
```

```
OGR data source with driver: ESRI Shapefile
Source: "/home/jeffdoser/Dropbox/teaching/for472/text/book/PEF", layer: "roads"
with 33 features
It has 2 fields
```

[21] https://www.r-bloggers.com/r-using-rcolorbrewer-to-colour-your-figures-in-r

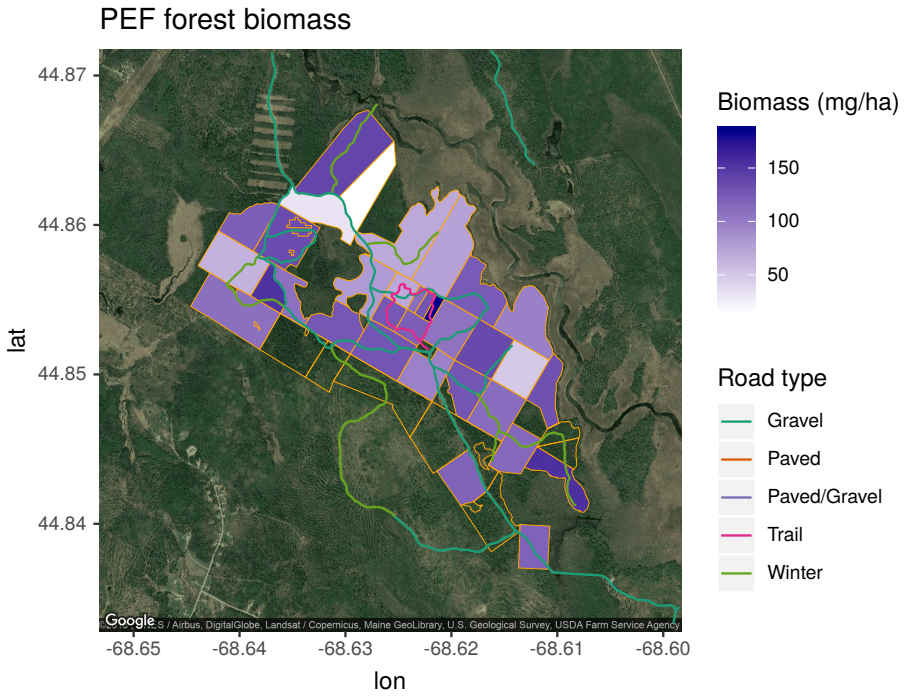
[22] Install the `RColorBrewer` package and run `library(RColorBrewer); display.brewer.all()` to get a graphical list of available palettes.

```
> roads <- spTransform(roads, proj4string(mu))
>
> roads.f <- fortify(roads, region="id")
> roads.f <- left_join(roads.f, roads@data, by = c('id' = 'id'))
```

```
Warning: Column `id` joining character vector and
factor, coercing into character vector
```

```
> ggmap(basemap) +
+     geom_polygon(data=mu.f, aes(x = long, y = lat, group=group,
+                                 fill=biomass_mu),
+                  size=0.2, color="orange") +
+     geom_path(data=roads.f, aes(x = long, y = lat,
+                                 group=group, color=factor(type))) +
+     scale_fill_gradient(low="white", high="darkblue",
+                         na.value="transparent") +
+     scale_color_brewer(palette="Dark2") +
+     labs(fill="Biomass (mg/ha)", color="Road type", xlab="Longitude",
+          ylab="Latitude", title="PEF forest biomass")
```

```
Warning: Removed 616 rows containing missing values
(geom_path).
```



The second, and more cryptic, of the two warnings from this code occurs because some of the roads extend beyond the range of the map axes and are removed (nothing to worry about).

8.6 Illustration using `leaflet`

Leaflet is one of the most popular open-source JavaScript libraries for interactive maps. As noted on the official R leaflet website[23], it's used by websites ranging from *The New York Times* and *The Washington Post* to GitHub and Flickr, as well as by GIS specialists like OpenStreetMap, Mapbox, and CartoDB.

The R leaflet website[24] is an excellent resource to learn leaflet basics, and should serve as a reference to gain a better understanding of the topics we briefly explore below.

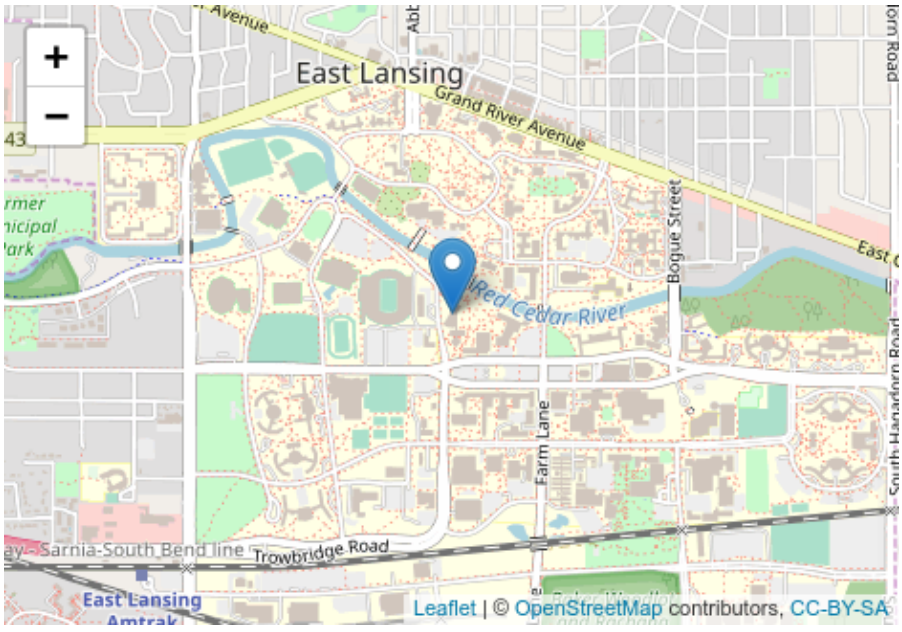
You create a leaflet map using these basic steps:

[23] https://rstudio.github.io/leaflet
[24] https://rstudio.github.io/leaflet

1. Create a map by calling `leaflet()`
2. Add data layers to the map using layer functions such as, `addTiles()`, `addMarkers()`, `addPolygons()`, `addCircleMarkers()`, `addPolylines()`, `addRasterImage()` and other `add...` functions
3. Repeat step 2 to add more layers to the map
4. Print the map to display it

Here's a brief example.

```
> library(leaflet)
>
> m <- leaflet() %>%
+       addTiles() %>%  # Add default OpenStreetMap map tiles
+       addMarkers(lng=-84.482004, lat=42.727516,
+                  popup="<b>Here I am!</b>") # Add a clickable marker
> m  # Print the map
```



There are a couple things to note in the code. First, we use the pipe operator `%>%` just like in `dplyr` functions. Second, the `popup` argument in `addMarkers()` takes standard HTML and clicking on the marker makes the text popup. Third, the html version of this text provides the full interactive, dynamic map, so we encourage you to read and interact with the html version of this textbook for

this section. The PDF document will simply display a static version of this map and will not do justice to how awesome `leaflet` truly is!

As seen in the `leaflet()` call above, the various `add...` functions can take longitude (i.e., `lng`) and latitude (i.e., `lat`). Alternatively, these functions can extract the necessary spatial information from `sp` objects, e.g., Table 8.1, when passed to the data argument (which greatly simplifies life compared with map making using `ggmap`).

8.7 Subsetting Spatial Data

You can imagine that we might want to subset spatial objects to map specific points, lines, or polygons that meet some criteria, or perhaps extract values from polygons or raster surfaces at a set of points or geographic extent. These, and similar types, of operations are easy in R (as long as all spatial objects are in a common CRS). Recall from Chapter 4 how handy it is to subset data structures, e.g., vectors and data frames, using the `[]` operator and logical vectors? Well it's just as easy to subset spatial objects, thanks to the authors of `sp`, `raster`, and other spatial data packages.

8.7.1 Fetching and Cropping Data using `raster`

In order to motivate our exploration of spatial data subsetting and to illustrate some useful functionality of the `raster` package, let's download some elevation data for the PEF. The `raster` package has a rich set of functions for manipulating raster data as well as functions for downloading data from open source repositories. We'll focus on the package's `getData` function, which, given a location in geographic longitude and latitude or location name, will download data from GADM[25], Shuttle Radar Topography Mission[26], Global Climate Data[27], and other sources commonly used in spatial data applications.

Let's download SRTM surface elevation data for the PEF, check the resulting object's class and CRS, and display it using the `raster` package's `image` function along with the PEF forest inventory plots.

```
> library(raster)
```

[25] http://www.gadm.org/%7D%7BGlobal%20Administrative%20Boundaries
[26] https://www2.jpl.nasa.gov/srtm/%7D%7BShuttle%20Radar%20Topography%20Mission
[27] http://www.worldclim.org/

```
Attaching package: 'raster'

The following object is masked from 'package:dplyr':

    select

The following object is masked from 'package:tidyr':

    extract
```

```
> pef.centroid <- as.data.frame(plots) %>%
+     summarize(mu.x = mean(easting), mu.y = mean(northing))
>
> srtm <- getData("SRTM", lon = pef.centroid[, 1], lat = pef.centroid[, 2])
```

```
Error in .SRTM(..., download = download, path = path): file not found
```

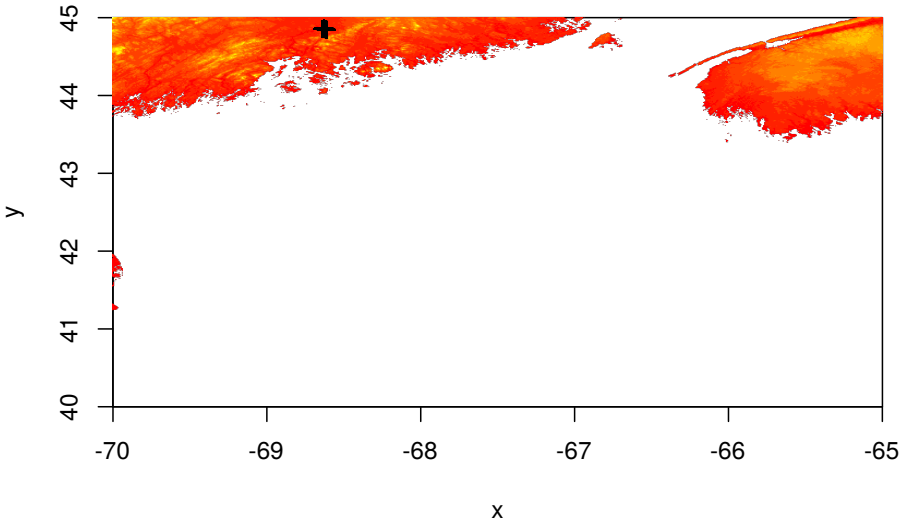
```
> srtm <- raster("srtm_23_04.tif")
> proj4string(srtm) <- "+proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0"
>
> class(srtm)
```

```
[1] "RasterLayer"
attr(,"package")
[1] "raster"
```

```
> proj4string(srtm)
```

```
[1] "+proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0"
```

```
> image(srtm)
> plot(plots, add = TRUE)
```

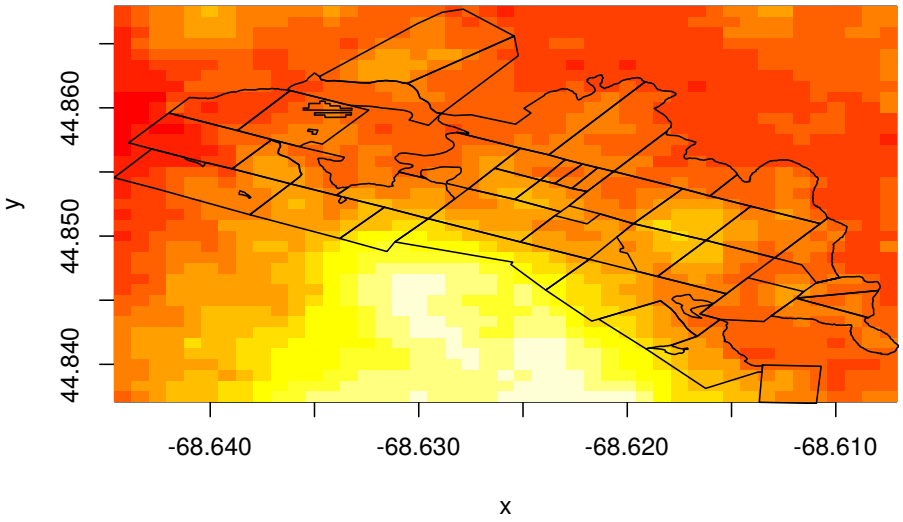


A few things to notice in the code above. First the `getData` function needs the longitude `lon` and latitude `lat` to identify which SRTM raster tile to return (SRTM data come in large raster tiles that cover the globe). As usual, look at the `getData` function documentation for a description of the arguments. To estimate the PEF's centroid coordinate, we averaged the forest inventory plots' latitude and longitude then assigned the result to `pef.centroid`. Second, there is currently a bug with downloading `SRTM` data using the `getData()` function. All the data are downloaded into your current directory, but the function does not properly load them into `R`. If you run this line yourself and get an error, continue going through the code we have. In the next line we load the data in ourselves using the call to `raster("srtm_23_04.tiff")`. I also manually change the coordinate system using the `proj4string` function. The `srtm` object result from our code to get around the bug is a `RasterLayer`, see Table 8.1. Third, `srtm` is in a longitude latitude geographic CRS (same as our other PEF data). Finally, the image shows SRTM elevation along the coast of Maine, the PEF plots are those tiny specks of black in the northwest quadrant, and the white region of the image is the Atlantic Ocean.

Okay, this is a start, but it would be good to crop the SRTM image to the PEF's extent. This is done using `raster`'s `crop` function. This function can use many different kinds of spatial objects in the second argument to calculate the extent at which to crop the object in the first argument. Here, I set `mu` as the second argument and save the resulting SRTM subset over the larger tile (the `srtm` object).

```
> srtm <- crop(srtm, mu)
>
```

```
> image(srtm)
> plot(mu, add = TRUE)
```



The `crop` is in effect doing a spatial setting of the raster data. We'll return to the `srtm` data and explore different kinds of subsetting in the subsequent sections.

8.7.2 Logical, Index, and Name Subsetting

As promised, we can subset spatial objects using the `[]` operator and a logical, index, or name vector. The key is that `sp` objects behave like data frames, see Section 4.5. A logical or index vector to the left of the comma in `[,]` accesses points, lines, polygons, or pixels. Similarly, a logical, index, or name vector to the right of the comma accesses variables.

For example, say we want to map forest inventory plots with more than 10,000 stems per hectare, `stems_ha` (the `min()` was added below to double check that the subset worked correctly.

```
> min(plots$stems_ha)
```

```
[1] 119
```

```
> plots.10k <- plots[plots$stems_ha > 10000, ]
```

```
>
> min(plots.10k$stems_ha)
```

```
[1] 10008
```

You can also add new variables to the spatial objects.

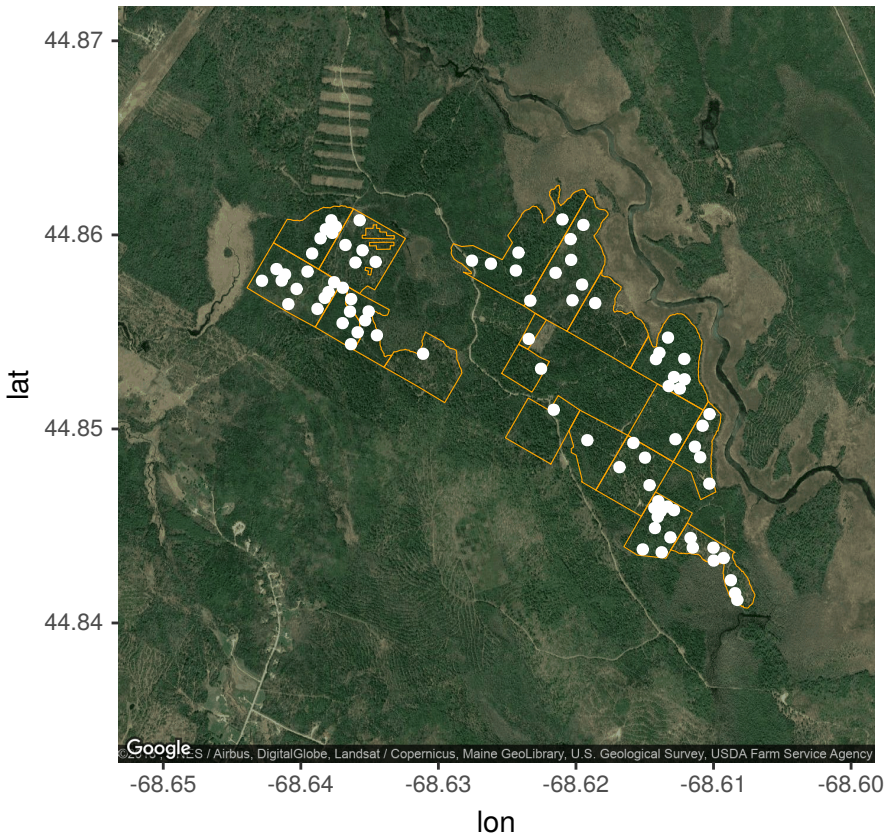
```
> plots$diameter_in <- plots$diameter_cm/2.54
>
> head(plots)
```

```
  mu_id plot biomass_mg_ha stems_ha diameter_cm
1   U10   11         96.35     5453         4.8
2   U10   13        115.70     2629         6.9
3   U10   21        121.58     3385         6.1
4   U10   22         71.97     7742         3.1
5   U10   23        122.26     7980         4.7
6   U10   24         85.85    10047         1.6
  basal_area_m2_ha diameter_in
1             22.0      1.8898
2             23.2      2.7165
3             23.0      2.4016
4             16.1      1.2205
5             29.2      1.8504
6             19.1      0.6299
```

8.7.3 Spatial Subsetting and Overlay

A spatial overlay retrieves the indexes or variables from object A using the location of object B. With some spatial objects this operation can be done using the `[]` operator. For example, say we want to select and map all management units in `mu`, i.e., A, that contain plots with more than 10,000 stems per ha, i.e., B.

```
> mu.10k <- mu[plots.10k, ]  ## A[B,]
>
> mu.10k.f <- fortify(mu.10k, region = "mu_id")
>
> ggmap(basemap) + geom_polygon(data = mu.10k.f, aes(x = long,
+   y = lat, group = group), fill = "transparent", size = 0.2,
+   color = "orange") + geom_point(data = as.data.frame(plots.10k),
+   aes(x = easting, y = northing), color = "white")
```



More generally, however, the `over` function offers consistent overlay operations for `sp` objects and can return either indexes or variables from object A given locations from object B, i.e., `over(B, A)` or, equivalently, `B%over%A`. The code below duplicates the result from the preceding example using `over`.

```
> mu.10k <- mu[mu$mu_id %in% unique(over(plots.10k, mu)$mu_id),
+   ]
```

Yes, this requires more code but `over` provides a more flexible and general purpose function for overlays on the variety of `sp` objects. Let's unpack this one-liner into its five steps.

```
> i <- over(plots.10k, mu)
> ii <- i$mu_id
> iii <- unique(ii)
> iv <- mu$mu_id %in% iii
> v <- mu[iv, ]
```

i. The `over` function returns variables for `mu`'s polygons that coincide with the 85 points in `plots.10k`. No points fall outside the polygons and the polygons do not overlap, so *i* should be a data frame with 85 rows. If polygons did overlap and a point fell within the overlap region, then variables for the coinciding polygons are returned.
ii. Select the unique `mu` identifier `mu_id` (this step is actually not necessary here because `mu` only has one variable).
iii. Because some management units contain multiple plots there will be repeat values of `mu_id` in *ii*, so apply the `unique` function to get rid of duplicates.
iv. Use the `%in%` operator to create a logical vector that identifies which polygons should be in the final map.
v. Subset `mu` using the logical vector created in *iv*.

Now let's do something similar using the `srtm` elevation raster. Say we want to map elevation along trails, winter roads, and gravel roads across the PEF. We could subset `srtm` using the `roads SpatialLinesDataFrame`; however, mapping the resulting pixel values along the road segments using `ggmap` requires a bit more massaging. So, to simplify things for this example, `roads` is first coerced into a `SpatialPointsDataFrame` called `roads.pts` that is used to extract spatially coinciding `srtm` pixel values which themselves are coerced from `raster`'s `RasterLayer` to `sp`'s `SpatialPixelsDataFrame` called `srtm.sp` so that we can use the `over` function. We also choose a different basemap just for fun.

```
> hikes <- roads[roads$type %in% c("Trail", "Winter", "Gravel"),]
>
> hikes.pts <- as(hikes, "SpatialPointsDataFrame")
> srtm.sp <- as(srtm, "SpatialPixelsDataFrame")
>
> hikes.pts$srtm <- over(hikes.pts, srtm.sp)
>
> basemap <- get_map(location=mu.bbox,  zoom = 14, maptype="terrain")
```

```
Warning: bounding box given to google - spatial extent
only approximate.
```

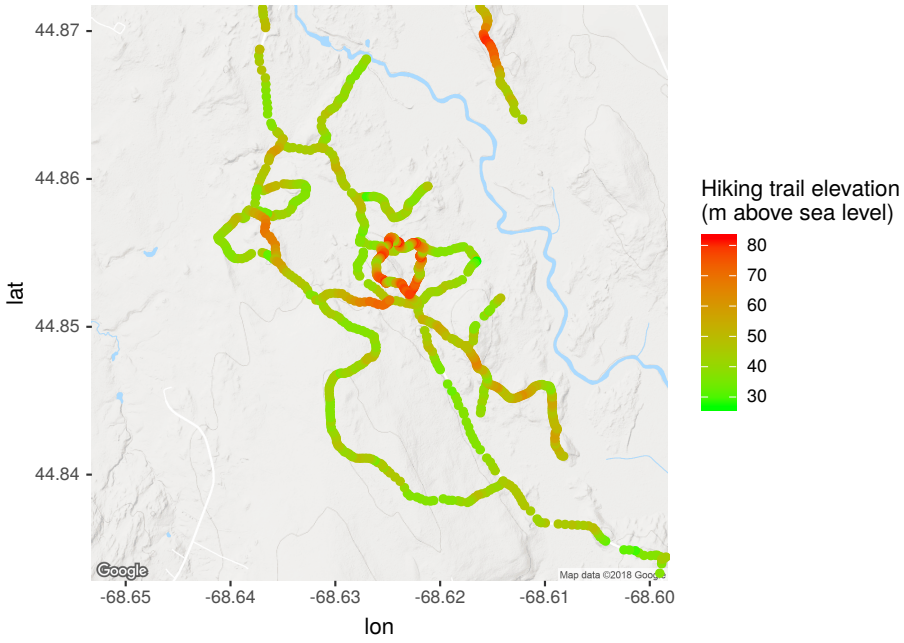
```
converting bounding box to center/zoom specification. (experimental)
```

```
Source : https://maps.googleapis.com/maps/api/staticmap?center=44.852323,-68.625803&zoom=1
```

```
> color.vals <- srtm@data@values[1:length(hikes.pts)]
>
> ggmap(basemap) +
+     geom_point(data=as.data.frame(hikes.pts),
```

```
+                aes(x = coords.x1, y = coords.x2, color = color.vals)) +
+     scale_color_gradient(low="green", high="red") +
+     labs(color = "Hiking trail elevation\n(m above sea level)",
+          xlab="Longitude", ylab="Latitude")
```

```
Warning: Removed 483 rows containing missing values
(geom_point).
```



In the call to `geom_point` above, `coords.x1` `coords.x2` are the default names given to longitude and latitude, respectively, when `sp` coerces `hikes` to `hikes.pts`. These points represent the vertices along line segments. I create the vector `color.vals` that contains the values from `srtm` that I use in the map argument `color`. Normally, I would be able to simply use the argument `color = srtm` in the graph, but since there is a bug in the `getData` function I mentioned earlier, we need to do another workaround here.

Overlay operations involving lines and polygons over polygons require the `rgeos` package which provides an interface to the Geometry Engine - Open Source[28] (GEOS) `C++` library for topology operations on geometries. We'll leave it to you to explore these capabilities.

[28] https://trac.osgeo.org/geos/

8.7.4 Spatial Aggregation

We have seen aggregation operations before when using `dplyr`'s `summarize` function. The `summarize` function is particularly powerful when combined with `group_by()`, which can apply a function specified in `summarize()` to a variable partitioned using a grouping variable. The `aggregate` function in `sp` works in a similar manner, except groups are delineated by the spatial extent of a thematic variable. In fact, the work we did to create `mu.bio` using `dplyr` functions can be accomplished with `aggregate()`. Using `aggregate()` will, however, require a slightly different approach for joining the derived average `biomass_mg_ha` to the fortified `mu`. This is because the `aggregate` function will apply the user specified function to all variables in the input object, which, in our case, results in an `NA` for the linking variable `mu_id` as demonstrated below.

```
> mu.ag <- aggregate(plots[, c("mu_id", "biomass_mg_ha")],
+   by = mu, FUN = mean)
>
> head(mu.ag@data, n = 2)
```

```
  mu_id biomass_mg_ha
0  <NA>         49.86
1  <NA>        112.17
```

With `mu_id` rendered useless, we do not have a variable that uniquely identifies each polygon for use in `fortify`'s `region` argument; hence no way to subsequently join the unfortified and fortified versions of `mu.bio.ag`. Here's the work around. If the `region` is not specified, `fortify()` uses an internal unique polygon ID that is part of the `sp` data object and accessed via `row.names()`[29] So, the trick is to add this unique polygon ID to the `aggregate()` output prior to calling `fortify()` as demonstrated below.

```
> mu.ag$id <- row.names(mu.ag)
>
> mu.ag.f <- fortify(mu.ag)
```

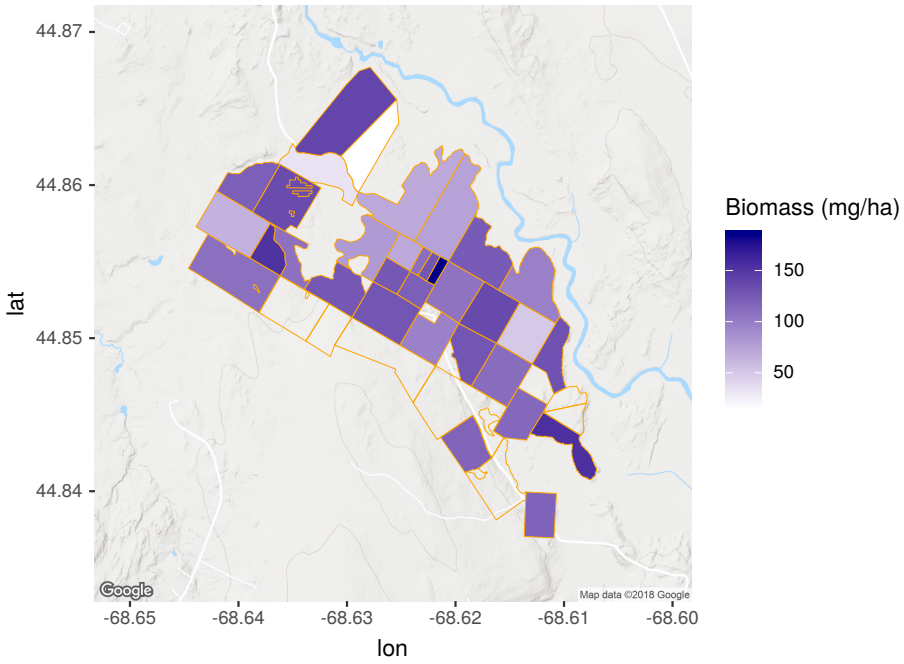
```
Regions defined for each Polygons
```

[29] With other data, there is a chance the row names differ from the unique polygon IDs. Therefore a more reliable approach to getting a unique ID is to use `sapply(slot(mu.ag, 'polygons'), function(x) slot(x, 'ID'))`, but replace `mu.ag` with your `SpatialPolygonsDataFrame`. Also, this approach will work with other `sp` objects in the right column of Table 8.1.

```
> mu.ag.f <- left_join(mu.ag.f, mu.ag@data)
```

```
Joining, by = "id"
```

```
> ggmap(basemap) + geom_polygon(data = mu.ag.f, aes(x = long,
+   y = lat, group = group, fill = biomass_mg_ha), size = 0.2,
+   color = "orange") + scale_fill_gradient(low = "white",
+   high = "darkblue", na.value = "transparent") + labs(fill = "Biomass (mg/ha)")
```



The `aggregate()` function will work with all `sp` objects. For example let's map the variance of pixel values in `srtm.sp` by management unit. Notice that `aggregate()` is happy to take a user-specified function for `FUN`.

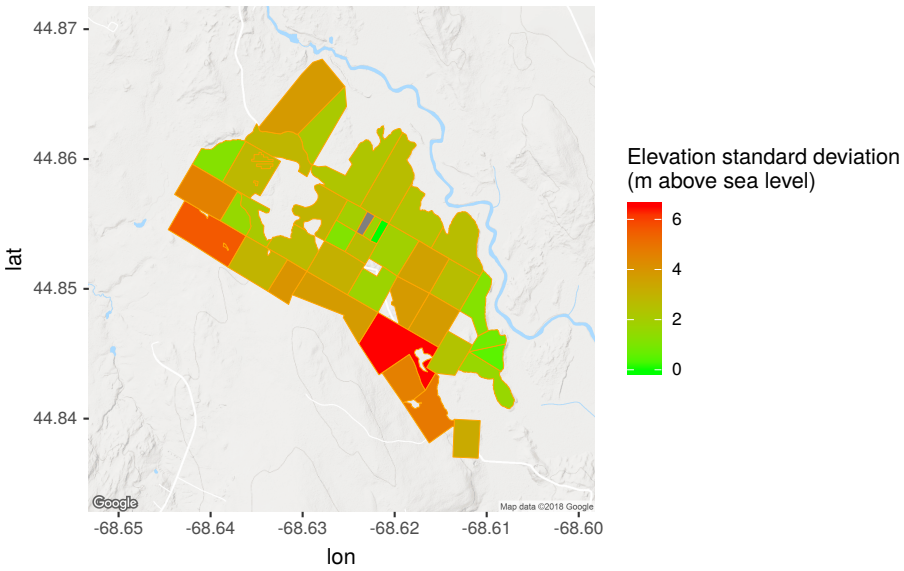
```
> mu.srtm <- aggregate(srtm.sp, by=mu,
+                      FUN=function(x){sqrt(var(x))})
>
> mu.srtm$id <- row.names(mu.srtm)
>
> mu.srtm.f <- fortify(mu.srtm)
```

```
Regions defined for each Polygons
```

```
> mu.srtm.f <- left_join(mu.srtm.f, mu.srtm@data)
```

```
Joining, by = "id"
```

```
> ggmap(basemap) +
+     geom_polygon(data=mu.srtm.f, aes(x = long, y = lat, group=group,
+                                      fill=srtm_23_04),
+                  size=0.2, color="orange") +
+     scale_fill_gradient(low="green", high="red") +
+         labs(fill = "Elevation standard deviation\n(m above sea level)",
+          xlab="Longitude", ylab="Latitude")
```



8.8 Where to go from here

This chapter just scratches the surface of R's spatial data manipulation and visualization capabilities. The basic ideas we presented here should allow you to take a deeper look into `sp`, `rgdal`, `rgeos`, `ggmap`, `leaflet`, and a myriad of other excellent user-contributed R spatial data packages. A good place to start is with Edzer Pebesma's excellent vignette on the use of the map overlay and

spatial aggregation, available here[30], as well as *Applied Spatial Data Analysis with R* by Bivand et al. (2013).

[30] https://cran.r-project.org/web/packages/sp/vignettes/over.pdf

Bibliography

Asner, G., Hughes, R., Varga, T., Knapp, D., and Kennedy-Bowdoin, T. (2009). Environmental and biotic controls over aboveground biomass throughout a tropical rain forest. *Ecosystems*, 12(2):261–278.

Babcock, C., Matney, J., Finley, A., Weiskittel, A., and Cook, B. (2013). Multivariate spatial regression models for predicting individual tree structure variables using lidar data. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 6(1, SI):6–14.

Bivand, R. S., Pebesma, E., and Gomez-Rubio, V. (2013). *Applied spatial data analysis with R, Second edition*. Springer, NY.

Bravo, A., Porzecanski, A., Sterling, E., Bynum, N., Cawthorn, M., Fernandez, D. S., Freeman, L., Ketcham, S., Leslie, T., Mull, J., and Vogler, D. (2016). Teaching for higher levels of thinking: developing quantitative and analytical skills in environmental science courses. *Ecosphere*, 7(4):n/a–n/a. e01290.

Finley, A., Banerjee, S., and MacFarlane, D. (2011). A hierarchical model for quantifying forest variables over large heterogeneous landscapes with uncertain forest areas. *Journal ot the American Statistical Association*, 106(493):31–48.

Foster, J. R., D'Amato, A. W., and Bradford, J. B. (2014). Looking for age-related growth decline in natural forests: Unexpected biomass patterns from tree rings and simulated mortality. *Oecologia*, 175(1):363–374.

Jenkins, J. C., Chojnacky, D. C., Heath, L. S., and Birdsey, R. A. (2003). National-scale biomass estimators for united states tree species. *Forest Science*, 49(1):12–35.

Kenefic, L. S., Rogers, N. S., Puhlick, J. J., Waskiewicz, J. D., and Brissette, J. C. (2015). Overstory tree and regeneration data from the "silvicultural effects on composition, structure, and growth" study at penobscot experimental forest. 2nd edition. fort collins, co: Forest service research data archive.

Kubiske, M. E. (2013). Tree height and diameter data from the aspen face experiment, 1997-2008. newtown square, pa: Usda forest service, northern research station.

Le Toan, T., Quegan, S., Davidson, M., Balzter, H., Paillou, P., Papathanassiou, K., Plummer, S., Rocca, F., Saatchi, S., Shugart, H., et al. (2011). The biomass mission: Mapping global forest biomass to better understand the terrestrial carbon cycle. *Remote Sensing of Environment*, 115(11):2850–2860.

Næsset, E. (2011). Estimating above-ground biomass in young forests with airborne laser scanning. *International Journal of Remote Sensing*, 32(2):473–501.

Neigh, C., Nelson, R., Ranson, K., Margolis, H., Montesano, P., Sun, G., Kharuk, V., Næsset, E., Wulder, M., and Andersen, H. (2013). Taking stock of circumboreal forest carbon with ground measurements, airborne and spaceborne lidar. *Remote Sensing of Environment*, 137:274–287.

Ometto, J. P., Aguiar, A. P., Assis, T., Soler, L., Valle, P., Tejada, G., Lapola, D. M., and Meir, P. (2014). Amazon forest biomass density maps: Tackling the uncertainty in carbon emission estimates. *Climatic Change*, pages 1–16.

Schimel, D. and Keller, M. (2015). Big questions, big science: meeting the challenges of global ecology. *Oecologia*, 177(4):925–934.

Spector, P. (2008). *Data Manipulation With R*. Use R!

UN-REDD (2009). The un-redd programme. http://www.un-redd.org/. Accessed: 5-22-2014.

UNFCCC (2015). United nations framework convention on climate change. http://unfccc.int/2860.php. Accessed: 6-6-15.

West, P. (2004). *Tree and Forest Measurement*. Springer.

Wood, F., Kochenderfer, J. N., and Adams, M. B. (2016). Felled tree biomass for four hardwood species of the central appalachians, west virginia. fort collins, co: Forest service research data archive.