R Reference Guide

Basics of using R for data analysis and visualization

Topher Weiss-Lehman & Lauren Shoemaker

23 September, 2016

What is R?

R is an object-oriented programming language. It is a free, open source platform for statistical computing developed and maintained by the R consortium. Because R is an open source programming language, many people may contribute to its development and potential add-ons. That is how Rstudio developed as an improved graphical interface for R and subsequently developed the use of Rmarkdown for incorporating text, R code, and figures together into pdf documents. If you are curious about who has contributed to R and the story of its development, type the function contributors() in the console.

Because R is a fully functional programming language, as opposed to a standard statistical software package like SAS or JMP, it requires a bit more of a learning curve. However, with just a little bit of work and patience, R has the ability to far outstrip what programs like SAS and JMP are capable of. For this reason, R is quickly surpassing these other programs as the preferred method of analyzing and visualizing data in ecology and evolution as well as other scientific disciplines. Even if you are not considering a career in ecology or evolution, computer programming is a highly desired skill in many careers and the techniques you learn in R can translate to any other programming language like C++, Java, or Python.

Objectives

The goal of this guide is to introduce you to the basics of using R, particularly in the context of analyzing and visualizing data. In the following sections of this guide, we will (1) cover some basic concepts relevant to programming in R, (2) go through the basics of loading, inspecting, and visualizing data, (3) practice constructing different types of graphs, adjusting various graphical parameters in R, and functions to add lines, points, and text to graphs, (4) practice using if statements and for loops, and (5) go over a few miscellaneous functions available in R that can be quite useful.

An additional resource to this reference guide is a series of short videos made by Dr. Maxwell B. Joseph (a former graduate student in the EBIO department at CU) for the Introduction to Quantitative Inference and Thinking course at CU. There are quite a few of these videos and they start out quite basic, but end up covering some quite advanced topics.

1. Basics of R

R as a calculator

At its most basic, R can operate as a calculator. When you open RStudio, your screen will likely be split into four panels. At the bottom is the *console* and at the top, you have your R Markdown or R Scripts open. In the console, you will see a command prompt, >. Go ahead and type in an equation, say $2 + 2$, and hit ENTER. You see that R prints the answer, just like using a calculator. Using the basic mathematical operations for addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (^) you can use R to perform any calculation in the same way you would use a calculator. One thing to note is that R obeys the order of operations (i.e. it will evaluate things inside parentheses first, then exponents, etc.). So, for example, observe how R evaluates the following things differently, based on the placement of the parentheses.

```
(8 * 2) / 4 ^ 2
```

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

```
((8 * 2) / 4) ^ 2
```

```
## [1] 16
```

Math and vectors

In addition to performing such calculations on single numbers, R can perform similar calculations for entire vectors of numbers. For example, if we want to square every value in a vector, we can use essentially the same syntax as if we wanted to square a single number.

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

```
## [1] 1 4 9
```

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

```
## [1] 1 4 9
```

We can take this a step further, and consider what happens when we perform arithmetic with entire vectors. R is designed to automatically *vectorize* such calculations so that each element in each vector can be added, multiplied etc. Additionally, you can use R to perform calculations with a mix of vectors and single numbers. Look through the following code for examples.

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

```
## [1] 4 6 8
```

```
c(1,2,3) * c(3,4,5)
```

```
## [1]  3  8 15
```

```
c(1,2,3) + 5
```

```
## [1] 6 7 8
```

Saving outputs in R

Performing the type of calculations performed above is useful, but it would be even more useful if we could reuse some of the values we calculated later. To do this, we need to save the output of such calculations as an object. This is what is meant by describing R as an "object-oriented programming language." For R to access anything later, it must be saved as an object. When creating an object, you may choose any name for it, but it cannot contain spaces or begin with a number or a special character (though it can contain special characters later in the name). For example, MyObject, my_object, and MyObject1 are all valid names for objects in R, but my object and 1my_object are not valid names. Also remember that R is case sensitive so myobject and MyObject look like two different objects to R. To create an object, you use the assignment operator <- with the name of the object you are creating on the left and the value you are setting that object to on the right. For example,

```
NewObject <- 7 * 8
NewObject
```

```
## [1] 56
```

creates an object called NewObject and sets it to a value of 56. In R, you can also use = as an assignment operator when creating objects, but using <- is generally the preferred method. This is for several reasons, some of them practical and some philosophical. We won't spend time going into this distinction here, except to point out that when setting function arguments (i.e. inputs) in R, you *must* use the = operator.

2. Loading and inspecting data in R

File paths and the working directory

To load a data file into R, first R must know where that data file is on your computer. That means knowing the *file path.* A file path is like a map of your computer. If a data file is in a folder on your desktop, then R needs to know to look there. When R first opens, it sets a default *working directory.* The working directory is the location in your computer where R looks for files and saves any outputs. To see what the working directory is in R, simply enter getwd() in the console and R will output the current working directory. The output will look like a series of locations on your computer separated by slashes. This is the file path to the current working directory. File paths will look slightly different between Linux based operating systems (like Macs) which use forward slashes (/) and Windows operating systems which use back slashes (). If you are unsure of how your file path should look, use the getwd() function to print the working directory and use that as a template.

```
# Example of the getwd() function
getwd()
```

```
## [1] "/Users/Topher/Desktop/CUBoulder/Classes/Evolutionary_Biology"
```

When loading data into R, it is usually most convenient to alter the working directory to be the same folder in which the data is stored. You can do this in two ways. The first, and easiest, is to open R by opening an R script or markdown document from within the same folder as the data. By doing this, R will automatically set that folder to be the working directory (you can confirm this by using the getwd() function). Alternatively, you can use the setwd() function to manually tell R to change the working directory. For example, if the data file is in a folder called "ExampleFolder" on my desktop, then I would type setwd("/Users/Topher/Desktop/ExampleFolder"). Note that I am using back slashes because I work on a Mac. If you are on a Windows machine, your file path will look different as described above. This will change the working directory to the ExampleFolder on my desktop.

Reading in your data

Once the working directory is set to wherever you need it to be, you can use the read.csv() function to read in data. Note that for this function to work, your data must be in the form of a csv (comma separate variable) file. R cannot read in data in the Excel file formats (.xlsx). If your data is located in the current working directory, then you can simply use the read.csv() function with the file name: read.csv("data.csv"). If your data is located in a different folder than the current working directory, then you simply need to supply the complete file path when reading in the data: read.csv("/Users/Topher/Desktop/ExampleFolder/data.csv"). There are two important things to notice in these function calls. The first is that the file extension (.csv in this case) is required to read in the data as it is part of the complete file name. The second is that the file name and file paths are *always* typed within quotations. This is because if there are no quotations, then R will assume there is an object called data.csv that you have previously defined and when it cannot find such an object, it will print an error message telling you that the object 'data.csv' is not found. By using quotations around the file name and file path, you tell R that you are giving it a location on the computer rather than an object.

When reading in your data it is important to assign it to an object which you can reference later. Remember that R is an object oriented programming language and that means that if you don't assign your data to an object, then R will read it in and immediately forget about it. R is not smart enough to automatically know that if you read in data you probably want to use it later. *You* have to tell R exactly what you want it to do. Remember, R is an incredibly powerful tool, but is still just a tool and it relies on you to do the critical thinking. To tell R to save the data you load in as an object you will again need to use the assignment operator (<-). Once you have loaded in the data, there are several things you can do to inspect the data. Look through the code block below at some of the things R can do to inspect the data.

Inspecting the properties of your data

```
# First set the working directory to the correct folder
setwd("/Users/Topher/Desktop/ExampleFolder")
# Now load in the data and save it to an object called MyData
MyData <- read.csv("data.csv")

# First, let's examine the structure of the data with the str function
str(MyData)
```

```
## 'data.frame':    100 obs. of  6 variables:
##  $ treatment: Factor w/ 2 levels "treatment1","treatment2": 1 2 1 2 1 2 1 2 1 2 ...
##  $ variable1: num  48.7 76.6 50.8 70.9 47.9 ...
##  $ variable2: int  1 0 0 1 1 0 1 0 1 1 ...
##  $ variable3: num  38.6 39.3 21.7 45.4 38.2 ...
##  $ variable4: num  59.5 63 37.4 66.7 53.3 ...
##  $ variable5: num  81.7 83.4 98.2 75.8 83.5 ...
```

```
# This creates output with the dimensions of the data, the
#    names of the columns, and the type of data in each column.

# Now let's try another way to examine the data
summary(MyData)
```

```
##       treatment    variable1       variable2      variable3
##  treatment1:50   Min.   :43.05   Min.   :0.00   Min.   :20.28
##  treatment2:50   1st Qu.:50.45   1st Qu.:0.00   1st Qu.:27.36
##                  Median :62.66   Median :1.00   Median :33.80
##                  Mean   :62.36   Mean   :0.55   Mean   :34.83
##                  3rd Qu.:75.29   3rd Qu.:1.00   3rd Qu.:42.38
##                  Max.   :80.85   Max.   :1.00   Max.   :49.07
##    variable4       variable5
##  Min.   :25.07   Min.   : 67.07
##  1st Qu.:40.08   1st Qu.: 83.40
##  Median :48.89   Median : 87.77
##  Mean   :52.42   Mean   : 88.93
##  3rd Qu.:65.19   3rd Qu.: 95.69
##  Max.   :86.50   Max.   :114.97
```

```
# This function gives a more statistical description of the data,
#    with the range, quartiles, median, and mean for each numeric
#    column and the number of each category for each factor column

# Finally, let's take a look at what the data looks like if we print
#    it out. It would be too unwieldy to print the whole data set,
#    but we can use the head() function to just print the first six
#    rows. Similarly the tail() function prints only the last six
#    rows.
head(MyData)
```

```
##    treatment variable1 variable2 variable3 variable4 variable5
```

```
## 1 treatment1  48.65814         1  38.62972  59.51358  81.68555
## 2 treatment2  76.63190         0  39.28123  63.04801  83.39880
## 3 treatment1  50.79845         0  21.66933  37.42456  98.16292
## 4 treatment2  70.86198         1  45.44847  66.66818  75.77936
## 5 treatment1  47.90553         1  38.24682  53.29363  83.52307
## 6 treatment2  77.08571         0  48.67289  81.21074  75.23287
```

```
tail(MyData)
```

```
##      treatment variable1 variable2 variable3 variable4 variable5
## 95  treatment1  51.76027         1  43.69961  69.55527  89.27522
## 96  treatment2  76.66107         1  43.69037  62.17303  71.69886
## 97  treatment1  48.07037         1  22.67929  36.48274 104.87199
## 98  treatment2  77.86641         0  30.40984  39.14664  88.35429
## 99  treatment1  52.13265         1  43.40759  66.88918  85.96975
## 100 treatment2  70.62942         0  27.19230  40.17382  86.34691
```

The functions detailed in the code above are all useful and accomplish different things. It is not always necessary to use all of them to inspect the data you load in, but it is a good idea to always use at least one and ideally two of them to get a feel for your data before you do anything else with it.

3. Visualizing data in R

There are many different ways to visualize data in R. When visualizing your data, the first question you must ask yourself is what type of figure you want to create. This will always be at least partially dependent on the type of data you have. Are you plotting one or two variables together? Are the data points numbers or categories? Do the data represent multiple independent measurements or repeated measurements of the same quantity over a spatial, temporal, or other type of gradient?

We will go through some of the different plots you can use under these conditions and how to create basic versions of them in R. Then, we will go over some of the many ways to customize graphs in R with graphical parameters and ancillary graphing functions.

For this section and the following section on statistical analyses, we will use some of R's built in datasets so that you may follow along on your own computer. R comes with numerous datasets preloaded and ready for you to use. Try typing data() into your console to generate a list and brief description of each dataset. For more information on a particular dataset, you can use the ? operator. For example, try typing ?CO2 and read the brief description of that dataset.

Histograms

If the data you are plotting is only concerned with a single variable, it is called *univariate* data. For example, faithful, one of the preloaded datasets in R concerns, among other things, the waiting times between eruptions for the Old Faithful geyser (try exploring this dataset with the some of the functions discussed above). If asked to visualize the distribution of waiting times between eruptions, that would mean plotting data for only a single variable (waiting times). Almost always when plotting univariate data, the most effective method is a histogram. Histograms bin the data values into pre-specified, equal sized ranges and display the number of data points which fall into each bin. They are quite useful in visualizing the full variability of the data and providing a rough check on what type of distribution it might conform to (e.g. normal vs. uniform or unimodal vs. multimodal). The basics of plotting a histogram are demonstrated below.

```
hist(faithful$waiting)
```

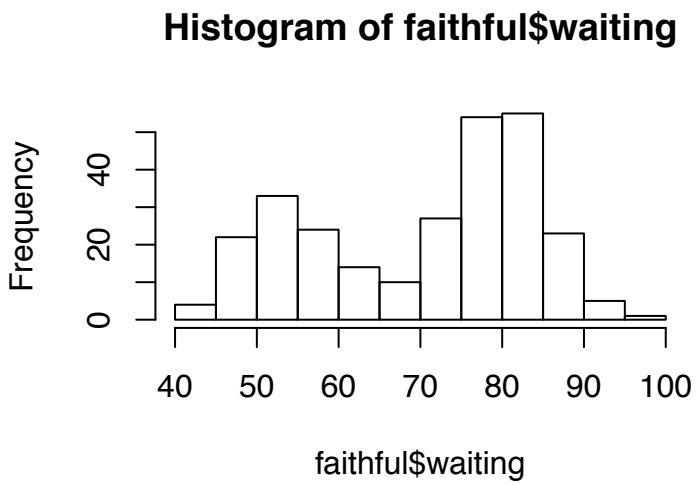

Figure 1: Histogram of the waiting times between eruptions for the Old Faithful geyser in Yellowstone National Park

One important caveat to be aware of when using histograms is that the chosen bin size can drastically affect the appearance, and therefore the available information, of the histogram. Notice that in the histogram with

bin sizes chosen by R using its default settings that the distribution of waiting times appears bimodal (there are two distinct peaks). See what happens when we manually adjust the bin sizes in the code below using the breaks argument.

```
# First adjust the graphical settings to allow three
#    graphs in the same row. This is done using the
#    par() function and the mfrow argument which sets
#    the number of rows and columns to divide the
#    graphing space into (type ?par for more
#    information)
par(mfrow = c(1, 3)) # 1 row and three columns
# Now create three histograms with increasing
#    bin sizes. Bin sizes are set by using the
#    breaks argument which takes in a vector
#    of the break points between bins. We can
#    create such a vector using the seq()
#    function which creates a sequence using
#    a start value, an end value, and a step
#    size (?seq)
hist(faithful$waiting, breaks = seq(from = 40, to = 100, by = 5))
hist(faithful$waiting, breaks = seq(from = 40, to = 100, by = 10))
hist(faithful$waiting, breaks = seq(from = 40, to = 100, by = 15))
```

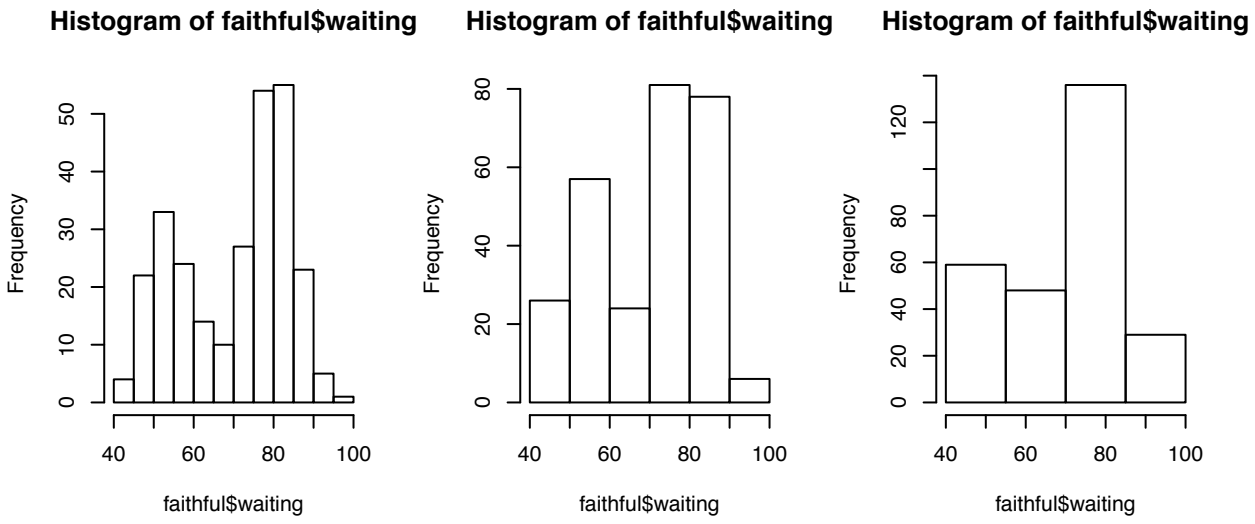

Figure 2: Three different histograms of the waiting times between eruptions with increasing bin sizes from left to right

Notice that as the bin size increases, the distribution of waiting times loses the appearance of bimodality and instead appears to have a single peak. While R usually does a good job of choosing appropriate bin sizes by default for histograms, it is always a good idea to be aware of this feature of this particular type of graph and to explore the consequences of altering bin sizes for your particular data.

Bar plots and Box plots

When creating a plot with two or more variables, your options for the type of figure you may wish to produce increase dramatically. Bar plots and box plots (sometimes called box and whisker plots) are two options

for plotting two variables against each other when one variable is numeric and the other is categorical. Categorical variables are defined by discrete levels, or categories, and do not necessarily have any quantitative relationship to each other (though they can). For example, eye color is often recorded as a categorical variable (brown, blue, etc.) though it could conceivably be defined more quantitatively on a numeric scale by somehow quantifying pigment levels in the iris.

Bar plots tend to be used when the categorical variables have a single summary numeric value associated with them. For example, let's take a look at the HairEyeColor dataset preloaded in R. This data set contains information on hair color and eye color in 592 male and female statistics students. To demonstrate the basics of creating a bar plot, we will focus on only the data from the female students. Notice that even when restricting the data to only female students, there are still two categorical variables considered in the dataset: hair color and eye color. Each of these variables is then broken down by the other variable's categories (e.g. The number of students with brown eyes and black hair, brown eyes and brown hair, etc.). There are two ways to graph this type of information with bar plots: stacked and side-by-side bars. See the code below and the resulting graphs for further explanation of these two types of bar plots.

```
# First let's isolate only the data for females
#    from the HairEyeColor dataset. To understand
#    why the following code works, try taking
#    a look at the structure of the HairEyeColor
#    dataset
FemaleHairEyeColor <- HairEyeColor[,,2]
# Now we'll use the par() function again
#    to make a row with two graphs
par(mfrow = c(1,2))
# Now create a stacked bar plot
barplot(FemaleHairEyeColor)
# Now graph the same data as a side-by-side plot
barplot(FemaleHairEyeColor, beside = TRUE)
```

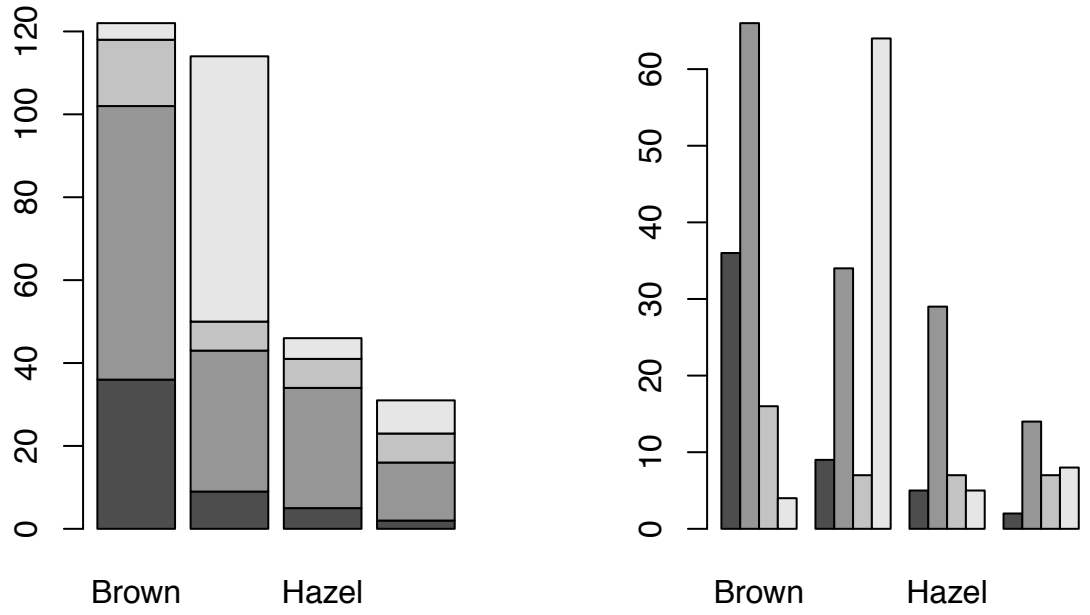

Figure 3: Comparison of the same data plotted as stacked bar plots (left) and side-by-side bar plots (right)

Notice that R constructs the bar plots by first considering the categories defining each column and then the rows as the secondary categories. This will be the same with any matrix given to the barplot function. Also

note that by default, R assigns a grey scale color to each secondary category (hair color in our example). This means that to be useful, these graphs will additionally need a legend denoting which color corresponds to which hair color. We will go over assigning colors and adding legends in a subsequent section. Also notice that R suppresses some of the x axis labels in an attempt to avoid overlapping text. In the section on graphical parameters and functions, we will learn how to manually set the axis labels and adjust their size to avoid this problem.

An alternative type of plot for mixing categorical and numeric variables are box plots (or box and whisker plots). These plots are useful for visualizing the distribution of the numeric variable associated with each level of the categorical variable. To illustrate the use of these graphs we will use the PlantGrowth dataset preloaded in R which contains results from an experiment comparing yields of plants grown under different treatments. We can use a box plot to visualize the difference in yields among all three treatments.

```
# First reset the graphing parameters
#    to not divide up the graphing
#    area
par(mfrow = c(1,1)) # 1 row, 1 column
# To make a box plot with yield graphed
#    across the three different treatments
#    R uses the ~ symbol to denote that
#    one variable should be graphed as
#    a function of the other variable.
boxplot(PlantGrowth$weight ~ PlantGrowth$group)
```

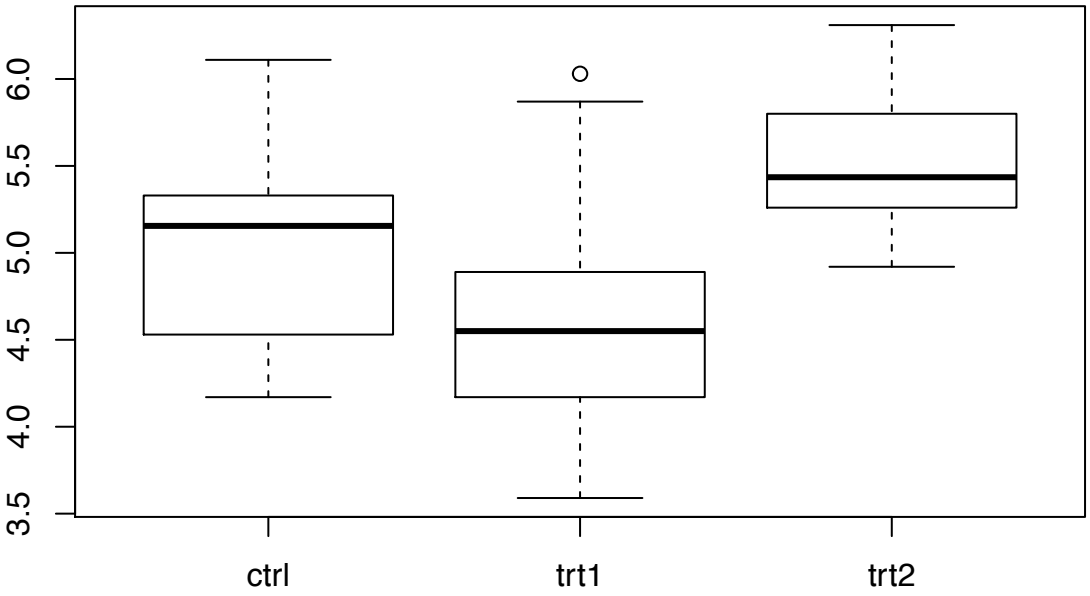

Figure 4: Box plot of the dried weight of plants grown under two experimental treatments and a control treatment

In the box plot created by the above code, the data in each category is visualized by plotting the median (thick horizontal line), the upper and lower quartiles (upper and lower edge of the box), and the extremes of the data (the whiskers and any outlier points that extend beyond the whiskers).

Point and line graphs

When plotting two numeric variables together, you will almost always wish to use either a point or a line graph to represent the data. Point and line graphs are fundamentally similar as both graph types assign

a numeric variable to the x and y axis and individual data points are then graphed as pairs of Cartesian coordinates. This means that for both types of graphs, you need to explicitly tell R which variable is the x variable and which is the y. The only difference between these two types of graphs is that in line graphs the coordinate pairs are connected by a line. While this is a simple difference between the two types of graphs, it is a meaningful one. By connecting the points with a line, you imply a continuity and dependence in the data. For example, if the data represent a series of repeated measurements of the phenotype of the same individual through time, that could make sense to plot as a line graph. If, on the other hand, the data represent measurements of the same phenotype, but from many different individuals through time, that would not make sense to plot as a line graph since each data point is independent of the others.

To illustrate these two graph types, we will use the cars dataset and the LakeHuron dataset. Go ahead and take a look at them on your own before proceeding to the graphing code below.

```
# First let's use the cars dataset to make a point graph
# Since the measurements in this dataset are independent
#    from each other (they do not represent repeated
#    measurements for the same car for example) a
#    point graph is the most logical representation of
#    this data.
plot(x = cars$speed, y = cars$dist)
```

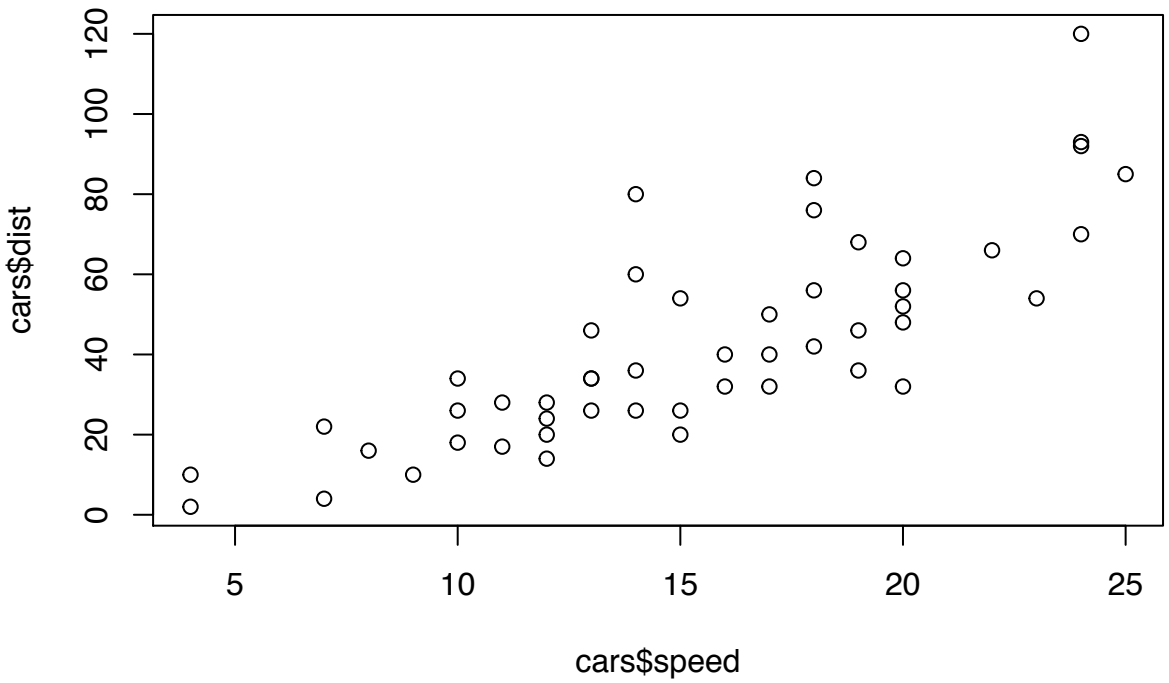

Figure 5: Bivariate point graph of the correlation of speed and stopping distance in cars measured in the 1920s (Ezekiel 1930)

```
# Now let's use the LakeHuron dataset to make a line
#    graph. Since this data represents repeated
#    measurements through time, a line graph is a
#    sensible option.
# First, since the LakeHuron data is a single vector,
#    without an explicit vector of years, we need
#    to create such a vector ourselves using the
```

```r
#    seq() function
Year <- seq(1875, 1972, by = 1)
# Now we can plot the data. Note that by default
#    R creates point graphs so we need to
#    explicitly tell it we want a line graph
#    by setting the argument "type"
plot(x = Year, y = LakeHuron, type = "l")
```

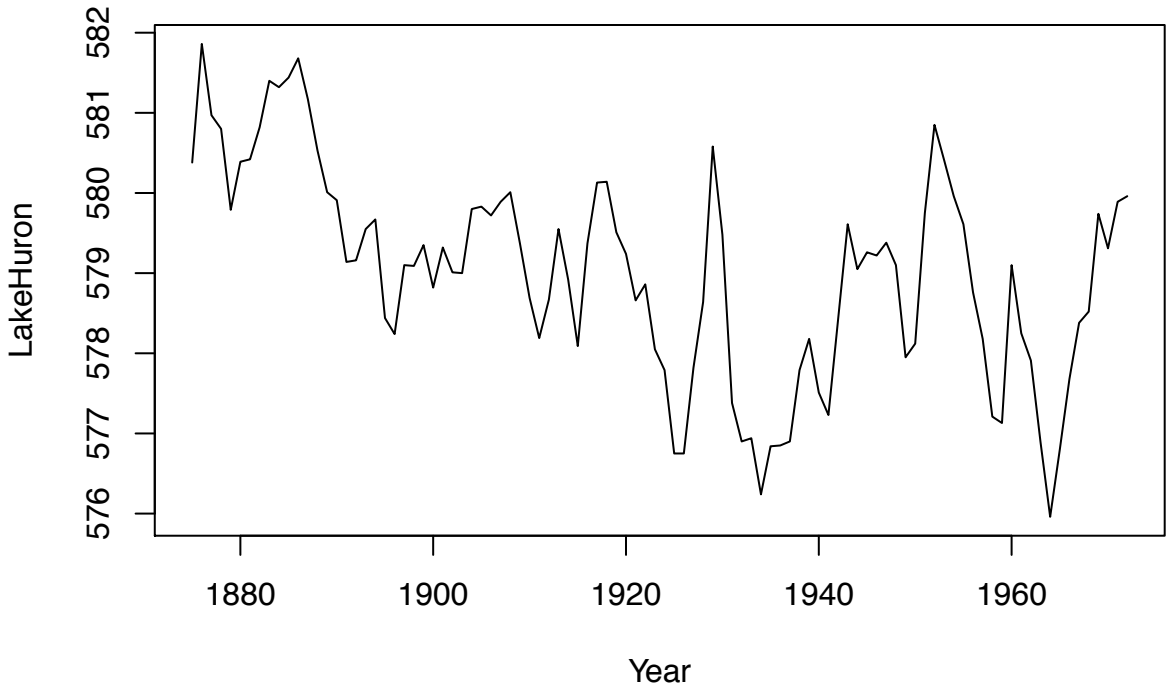

Figure 6: Bivariate line graph of the level (in feet) of Lake Huron measured annually from 1875 through 1972

Additional graphical parameters and functions

You may have noticed that the graphs we've created so far don't look very good. That's because all we've done is give R the most basic information it needs to make each graph. When making a graph in R, whether a histogram or a bivariate plot, there are many graphical parameters that can also be specified to adjust the graph that R creates.

Labels

Axis labels can be set in R using the xlab and ylab arguments. A title for the graph can be specified using the main argument. These arguments are set as follows: xlab = "Independent variable", ylab = "Dependent variable", main = "Figure title" within the plotting function.

Axis limits

Axis limits can also be set from within the plotting functions using the xlim and ylim arguments to set the range of x and y values to be graphed. Each of these arguments must be set to a vector of length two with

the first element corresponding to the minimum and the second element corresponding to the maximum (e.g. xlim = c(0, 10) specifies an x axis plotted from 0 to 10).

Expansion factors

Expansion factors for points and text can be set using the character expansion factor arguments. cex adjusts the size of points, cex.lab adjusts the size of axis labels, cex.axis adjusts the size of the numbers along the axes, and cex.main adjusts the size of the figure title. The values these arguments take in are proportional to the default sizes such that values less than one correspond to smaller sizes and values greater than one correspond to larger sizes.

Color

Color can be set using the col argument. In R, color is specified in any of several ways. First, col can be set to integer values which correspond to different colors (e.g. 1 is black, 2 is red). Second, col can be set to the name of a color as long as it is a recognized color in R (e.g. col = "red"). R recognizes 657 distinct colors which can all be viewed by using the colors() function. Third, R has several functions which can be used to generate colors. The most useful is the rgb() function which takes in four arguments: red, blue, green, and alpha. Each of these can be set to a numeric value between 0 and 1. For the three colors, the value corresponds to the proportion of each that the final color will be composed of (so the three arguments need to add up to one). The alpha argument sets a transparency value for the color with one corresponding to a fully opaque color. Another function is the grey() function which takes input as a single number between 0 and 1 and produces a greyscale color with zero corresponding to black and 1 corresponding to white. Finally, the functions rainbow() and grey.colors() take integer numbers as input and produce a corresponding number of colors either spread evenly across a color spectrum (rainbow()) or on the greyscale (grey.colors()).

Point and line types

In R you can also adjust the shape of points or the type of line (dashed, solid, etc.) using graphical parameters. The pch argument allows you to alter the type of points R plots. The default value is 1 which corresponds to hollow circles. R accepts integer values of 0 through 25 which all correspond to different types of symbols. Type ?points in the console to see what the different values correspond to. Similarly, the lty argument adjusts the type of line drawn on the graph. Type ?par and scroll down to lty to see what line types can be drawn. Furthermore, the argument lwd adjusts the line width and works much the same as cex for points.

Other useful functions for graphing

Sometimes it is necessary to add lines or points to an existing figure. In R, this can be accomplished with the points() and lines() functions. Both of these functions take in x and y coordinates and can accept many of the graphical parameters discussed above. Additionally, the function abline() can add specialized lines to a plot, such as vertical, horizontal, or an a + b*x straight line. The abline() function takes in the following arguments: a, b, h (the y value for a horizontal line), or v (the x value for a vertical line). Additionally, it can also take in many of the graphical parameters described above.

To add a legend to a figure, simply use the legend() function in R. The location of the legend can be specified with x and y locations or with a keyword (e.g. "topright", "right", "bottomright", etc.). The text used in the legend is specified as a vector to the legend argument, and point and/or line type, size and color for the legend are specified using the same graphical parameters described above.

To illustrate the graphical parameters and functions described above, we will use the CO2 dataset from R. This dataset contains the results from an experiment exploring the relationship between the ambient concentration of CO_2 and the uptake of CO_2 in plants from two different locations and subject to two

experimental treatments (Potvin et al. 1990). I encourage you to read more about the dataset by typing ?CO2 into your console.

One further function we will use in the following code is the subset() function. This is a highly useful function when working with complex datasets as it allows you to simply and quickly select only certain sections of the full dataset (e.g. a specific treatment) for individual graphing or analysis.

```
# In the following code we will create a high quality
#    graph of the response curve of CO2 uptake compared
#    to ambient concentrations of CO2 for both experimental
#    treatments. The graph will have labels, a legend,
#    colors, and trend lines.

# First let's establish sensible ranges for the two variables
#    so we can use them when graphing them
xrange <- c(min(CO2$conc), max(CO2$conc))
# We will add 10 to the y upper limit to make room for
#    a legend
yrange <- c(min(CO2$uptake), max(CO2$uptake) + 10)

# Now, let's use the subset function to create two
#    smaller datasets, each corresponding to one of
#    the experimental treatments. Note that the
#    double equals sign (==) is not an assignment
#    operator. Instead it is a logical operator
#    evaluating whether two values are indeed
#    equal to each other. See the following
#    section on if statements for more
#    information.
Chilled <- subset(CO2, Treatment == "chilled")
NonChilled <- subset(CO2, Treatment == "nonchilled")

# Since we will be using the same colors
#    multiple times to set the color for
#    points, lines, and the legend, it will
#    be useful to save the colors we use as
#    objects so they are easy to reuse
ChilledCol <- rgb(red = 0, green = 0.3, blue = 0.7, alpha = 1)
NonChilledCol <- "darkred"

# Now, let's plot the values for the chilled
#    treatment first, but using the variable
#    ranges we calculated with the full data.
#    We will also specify a different point
#    type than the default, adjust the point
#    size and color, and adjust the text size
#    for the axes, tick marks, and labels
plot(x = Chilled$conc, y = Chilled$uptake, xlim = xrange,
     ylim = yrange, xlab = "CO2 concentration",
     ylab = "CO2 uptake", pch = 5, col = ChilledCol,
     cex = 1.5, cex.lab = 1.5, cex.axis = 1.5)

# Now we'll add the points for the other treatment,
#    using the same point symbol and size, but a
#    different color.
```

```
points(x = NonChilled$conc, y = NonChilled$uptake,
       pch = 5, cex = 1.5, col = NonChilledCol)

# Next we will add trend lines to the graph. To do
#    this, first we will need to fit a regression
#    to the data, which we can do with the lm()
#    function. Notice that the data is not linear
#    but instead displays a logarithmic (i.e.
#    saturating) pattern of increase. This
#    means that to calculate the trend line
#    we will fit a model with a logarithmic
#    term.
ChilledModel <- lm(Chilled$uptake ~ log(Chilled$conc))
NonChilledModel <- lm(NonChilled$uptake ~ log(NonChilled$conc))

# Now generate an x sequence to use for calculating and graphing
#    the model predictions. Fitting the model by itself does not
#    generate any predictions. Instead, it calculates the coefficients
#    of the best fitting model. It is up to us to use those coefficients
#    to generate the y values for a trend line. We do this by extracting
#    the model coefficients using the coef() function and the fitted
#    model object. One other ingredient we need is a sequence of values
#    for the independent (or x) variable. We can generate such a sequence
#    again using the seq() function
ConcLevels <- seq(min(CO2$conc), max(CO2$conc), by = 10)
ChilledPreds <- coef(ChilledModel)[1] + coef(ChilledModel)[2] * log(ConcLevels)
NonChilledPreds <- coef(NonChilledModel)[1] + coef(NonChilledModel)[2] * log(ConcLevels)

# Now that we have the predicted values for the trend lines we can add
#    them to the graph using the lines function below
lines(x = ConcLevels, y = ChilledPreds, lty = 1, lwd = 2,
      col = ChilledCol)
lines(x = ConcLevels, y = NonChilledPreds, lty = 1, lwd = 2,
      col = NonChilledCol)

# Finally, we can add a legend to the graph using the legend()
#    function. There are two things to note in our call to
#    legend(). First, notice that the vector of colors used
#    must be in the same order as the vector of legend text
#    for the colors to match the correct text. Second, notice
#    that we can further specify the type of line used to draw
#    the box with the box.lty argument. Here we set it to 0,
#    meaning no line will be drawn around the box.
legend("topleft", legend = c("Chilled", "Non chilled"),
       col = c(ChilledCol, NonChilledCol), pch = 5, lty = 1,
       lwd = 2, box.lty = 0)
```

The graphical parameters and functions demonstrated in the above code are not just restricted to point and line graphs like the one demonstrated. They can also be used with histograms, bar plots, and any other graph produced in R.

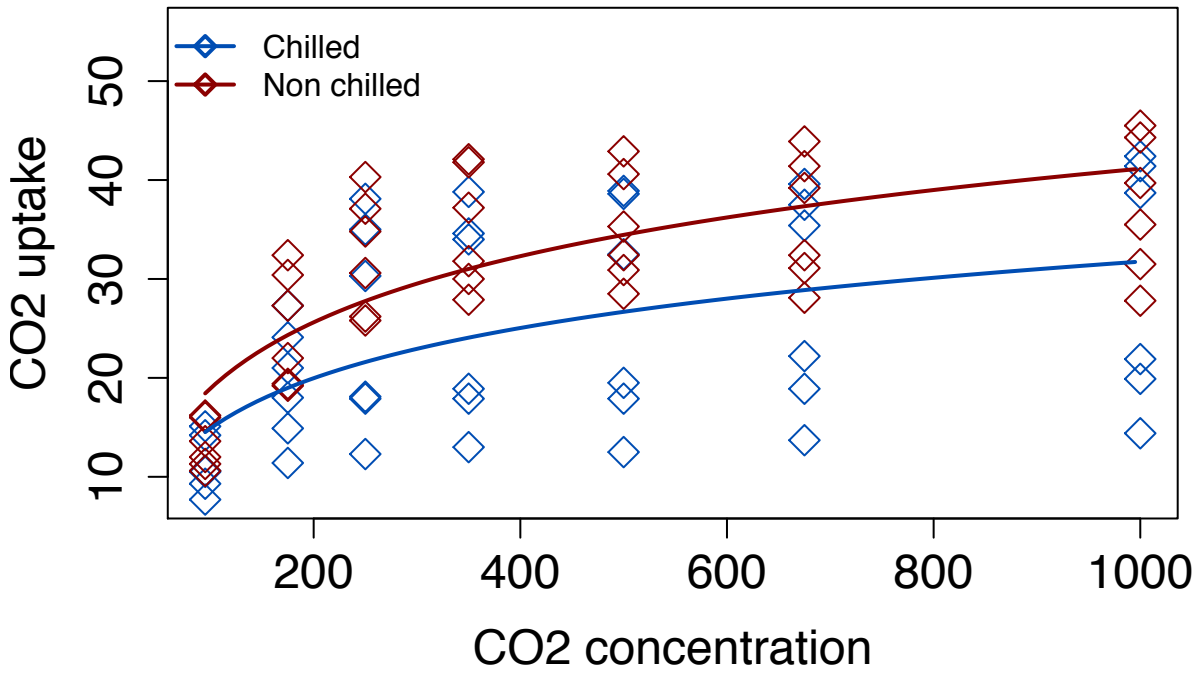

Figure 7: Data and trend lines for the relationship between CO2 concentration and uptake in experimentally chilled and non-chilled plants

4. Loops and if statements

Loops

In programming languages there are ways to do the same task multiple times without having to code the same thing again and again. These are called loops, or *repetition structures*, and usually come in two varieties: for-loops and while-loops.

For-loops are *counter controlled* where the number of repetitions is known ahead of time. For example, say you need to sample your data 100 times. While-loops, in comparison, are *sentinel controlled* reptetion, where the number of repetitions can be unknown. Instead, the loop ends when a given condition is completed. For example, say you want your code to continue running until your error rate is less than 0.001. You likely wouldn't know how many repetitions this would take ahead of time, so it would be best to use a while-loop rather than a for-loop.

The general structure for a for-loop and a while-loop tends to be similar with three general parts:

1. Initiation. This gives the initial, or starting conditions, that you use to begin running your loop. This is where you also define your parameters and set up all vectors/matrices, etc. that you will use to store your results.
2. Process phase. This is when you use a loop to actually do your calculations.
3. Termination phase. This is when your loop terminates and you clean your output for future use, analyses, graphing, etc.

Let's now look at the basic R structure for a for-loop.

```
# Print the numbers 1 through 10
```

```r
# initiation phase
start <- 1 # set the first value we will use to begin
end <- 10 # set when the loop will terminate

# process phase
for (i in start:end) {
  print(i)
}
```

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

Say you wanted to store the numbers rather than print them. You would want to make a vector to store all the values before the loop (note: you can actually do this in the loop, but it makes your code run slow, so we want to avoid this if possible).

```r
# Store the numbers 1 through 10

# initiation phase
start <- 1 # set the first value we will use to begin
end <- 10 # set when the loop will terminate
# We can create an empty vector to store the results by
#    using the rep() function. This function creates a
#    vector with entries equal to the first argument
#    and a length equal to the second argument.
#    Essentially, it creates a vector by repeating
#    the value it's given a certain number of times.
storage <- rep(NA, end)

# process phase
for (i in start:end) {
  storage[i] <- i
}

# print our results to make sure our loop worked correctly
storage
```

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

Let's compare this to the while-loop in R. We will use an example where we want to graph the function $y = y/2 + 1$ from 100 until y is less than 5. We will also record how many times our loop is used.

```
counter <- 0 # keeps track of how many times we loop through our while-statement
y <- 100     # initialization

# Since we want to graph our results, we'll need to make a vector containing each value.
# Our starting value is 100, so we'll start with a vector of length one containing
# the value 100.
results <- c(100)

while (y >= 5) {
  y <- y/2 + 1 # calculation

  # add y to the end of the vector containing all results
  results <- c(results, y)

  # update our counter with how many times we've used the loop
  counter <- counter + 1
}

# print how many times we went through the loop
counter
```

```
## [1] 6
```

```
# print our results
results
```

```
## [1] 100.00000  51.00000  26.50000  14.25000   8.12500   5.06250   3.53125
```

```
# graph results
plot(x = seq(0,counter), y = results, type="l", col="blue",
     xlab="Repetition Step", ylab="Output", lwd=2)
```

Take a look at our vector *results*. Our loop is supposed to stop when y is less than 5, but notice that our last value is 3.53, which is less than 5. Why is this?

Well, R evaluates all code *in order*. Therefore, after $y = 5.06250$, R will evaluate if $y >= 5$. Since the answer is yes, R will then go through the loop and calculate the new y value, 3.53125 and update *results* and *counter*. Keep this in mind when using while-loops in R.

When using while-loops, it is possible to get stuck in what is called an infinite loop. This is a loop that never ends because you gave it conditions that will never be met. If, in the above example, you wanted the loop to stop when y is less than 1, that would result in an infinite loop. Since y will never be less than 2 because $2/2 + 1 = 2$, this condition will never be reached. If (or when) you end up accidently coding an infinite loop, click on the red STOP button on the top right in your console. Or, you can force R to stop running code by hitting ESC on your keyboard.

If statements

If statements allow you to execute code *only* if a condition is true. This is helpful for parsing your data, when you only want to run code on certain portions of your data. It also is often used when you need to subset your data in complicated ways.

To evaluate whether a condition is true or not, R relies on certain *logical operators* defining a relationship between two quantities (e.g. greater than, equal to, etc.). The logical operators that R recognizes and their meanings in English are given in the table below.

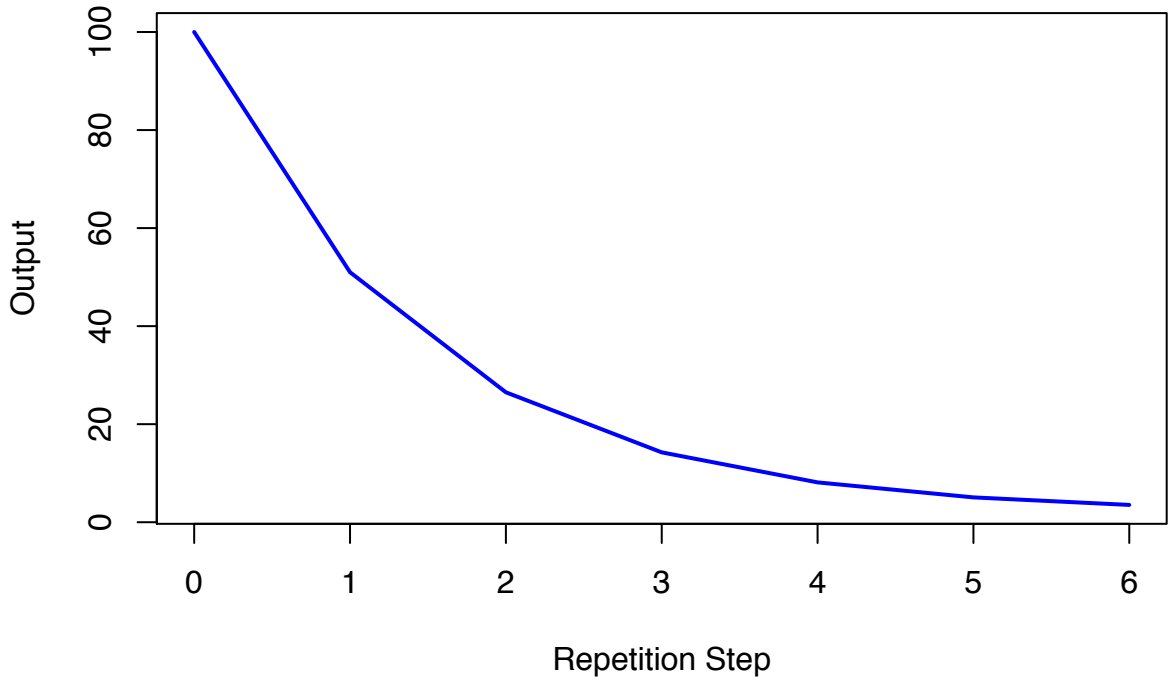

Figure 8: Example output generated using a while-loop to calculate output from the function y = y/2 + 1

Logical relationship	**Logical operator in R**
Strictly less than (<)	<
Less than or equal to (≤)	<=
Strictly greater than (>)	>
Greater than or equal to (≥)	>=
Exactly equal to (=)	==
Not equal to (≠)	!=
And (This operator combines two different logical conditions. It only evaluates as true if *both* X and Y are true)	&
Or (Evaluates as true if *either* X or Y is true)	|

Let's look at the basic R structure for an if statement. We will use an example where we draw numbers at random from a normal distribution and check to see if the mean of our random numbers is sufficiently close to the expected mean.

```
# Draw 10 random numbers from a normal distribution
samples <- rnorm(10, mean=5, sd=3)

# calculate the mean, rounded to the first decimal
samples_mean <- round(mean(samples), digits=1)

# Notice that the conditions given to the if-statements
#    take the form of a comparison between two values.
#    The logical operators that R accepts for such
```

```
#    comparisons are < (less than), > (greater than)
#    <= (less than or equal to), >= (greater than or
#    equal to), == (exactly equal to), and != (not equal)
if (samples_mean < 5) {
  print('mean is less than expected')
}
if (samples_mean > 5) {
  print('mean is more than expected')
}
```

```
## [1] "mean is more than expected"
```

```
if (samples_mean == 5) {
  print('mean is exactly what is expected!')
}
```

The above code could be simplified by using a slight variation on the if-statement, called the if-else statement. In these statements, if an argument is true, the loop will execute all code in brackets after the if-statement. However, if the argument is false, the code after the else-statement will be executed instead. Using the same example as above, we can see how to modify our code for an if-else statement. Note that the else has to go on the same line as the closing bracket, otherwise you will get an error.

```
if (samples_mean < 5) {
  print('mean is less than expected')
} else if (samples_mean > 5) {
  print('mean is more than expected')
} else {
  print('mean is exactly what is expected!')
}
```

```
## [1] "mean is more than expected"
```

Boolean (logical) variables

If statements like the ones above are one example of R using a *Boolean variable*. Boolean variables (named for nineteenth century mathematician George Boole) are logical variables that can only take two values (true or false; equivalent to 1 or 0). In the if statements above, the condition evaluated by the if statement evaluates to a Boolean variable and this variable (true or false) determines if R will execute the code within the if statement. In the section on creating high quality graphics, we saw another example of how R can use Boolean variables with the subset() function. R uses the condition given to the subset function to determine for which rows of the data set the condition yields a true value of the Boolean variable and then returns only those rows of the data.

Boolean variables can serve many purposes in R and we will not cover all of the myriad ways they can be useful. However, we will note one other useful application of Boolean variables. When manipulating data, it can sometimes be necessary to create a new vector of transformed data. While these types of transformations often involve a simple mathematical transformation (e.g. log transforming your data), there are other times when the desired transformation may depend on a logical condition (e.g. reset all values above or below a given threshold, or replace a text based category with numeric values). For these cases, it is necessary to use Boolean variables to identify which elements in your original data need to be transformed and/or how to transform them to a new value. This can be accomplished in several different ways and we demonstrate some of them below in the context of changing a vector of text based survival categories ("dead" or "alive")

to binary, numeric values (0s or 1s) which could then be used, for example, to perform a logistic regression testing how the probability of survival might vary with some independent variable (e.g. a phenotypic trait).

```
# Create an example vector of survival data
survival <- c("alive", "alive", "dead", "alive",
              "dead", "dead", "alive", "dead")

# Suppose we want to convert the above vector to a
#    new vector with 0s and 1s instead of "alive"
#    and "dead" to use in a logistic regression.
#    There are at least three different ways to
#    accomplish this that we will demonstrate here.

# Option 1: For loops and if statements
# First create a new, empty vector of the
#    same length as the survival vector
NumericSurvival1 <- rep(NA, length(survival))
# Now create a for loop to step through
#    each entry of the survival vector
for(i in 1:length(survival)){
     # Now within the for loop, use an
     # if statement to assign either
     # a 1 or a 0 to the NumericSurvival
     # vector depending on the corresponding
     # entry in survival
     if(survival[i] == "alive"){
          NumericSurvival1[i] <- 1
     } else{
          NumericSurvival1[i] <- 0
     }
}

# Option 2: Use the which() function
# The which() function in R can be used
#    to identify the indices of a vector
#    for which the elements correspond
#    to a given condition. We can
#    therefore use the which() function
#    to identify the indices of the
#    survival vector for which the
#    entries are either "dead" or
#    "alive" and use those to
#    set the values in our new vector.
# First we will create a new vector again
NumericSurvival2 <- rep(NA, length(survival))
# Now use which() to get the appropriate
#    indices
AliveIndices <- which(survival == "alive")
DeadIndices <- which(survival == "dead")
# Now use those indices to set the values
#    in NumericSurvival2
NumericSurvival2[AliveIndices] <- 1
NumericSurvival2[DeadIndices] <- 0
```

```
# Option 3: Use Boolean variables directly
#    in the vector
# We can directly use Boolean variables
#    when calling a specific vector
#    to directly select the indices
#    in one vector that correspond
#    to a given condition in another
#    vector.
# First, we again create a new vector
NumericSurvival3 <- rep(NA, length(survival))
# Now we can directly use the survival
#    vector to set values in
#    NumericSurvival3
NumericSurvival3[survival == "alive"] <- 1
NumericSurvival3[survival == "dead"] <- 0

# We can even use Boolean variables to confirm
#    that all three of these methods produced
#    the same vector
NumericSurvival1 == NumericSurvival2
```

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

```
NumericSurvival2 == NumericSurvival3
```

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

```
# Notice that when comparing two vectors using
#    a logical operator, R produces a vector
#    Boolean variables. To ensure that each
#    pair of entries in the vectors are identical,
#    all entries of the boolean vector must be
#    TRUE. We can check this by calculating
#    the sum of the boolean vector (remember
#    that TRUE values are equivalent to 1 and
#    FALSE values are equivalent to 0 in R) and
#    comparing it to its length. If each entry in the
#    vectors are TRUE, the sum of the Boolean
#    vector should equal its length
BooleanVector <- NumericSurvival1 == NumericSurvival2
sum(BooleanVector) == length(BooleanVector)
```

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

As you can see in the above code, all three of the options presented accomplish the same goal. There are slight differences among them in terms of efficiency (some of them run faster than others), but this will only matter if you are attempting to use these in the context of a massive dataset (e.g. thousands of entries) and even then, the differences will usually only manifest as a difference of a few seconds in run time. When deciding which of the above techniques to use, the most important consideration is determining which of them make the most sense to you. If one technique makes more logical sense to you than another, then use that technique. Not only will it give you a better idea of what your code is doing, but it will help you to trouble shoot your code if it runs into problems if you understand what it is doing.

5. Miscellaneous useful functions in R

R function	Input	Details and output
length()	A single dimensional vector	The length of the vector
dim()	A multidimensional matrix, array, or data frame	The dimensions of the input object
c()	Any number of single values or single dimensional vectors, separated by commas	A single vector consisting of all of the input values combined (concatenated) together in the order they were given
rep()	A single value or vector of values and the number of times the value(s) should be repeated	A vector consisting of the value(s) repeated the specified number of times
seq()	A starting and ending value ("from" and "to" respectively) and further optional inputs: a counting increment ("by") OR a desired length ("length.out")	A vector consisting of a sequence beginning at "from" and ending at "to" and changing by increments of "by" OR by equally sized increments calculated from the start, end, and desired length of the vector
sort()	A single dimensional vector and a logical input called "decreasing"	A vector consisting of the same elements as the input vector, but sorted in order from lowest to highest value (if decreasing = FALSE) or vice versa (if decreasing = TRUE)
is.na()	A single value or a vector of values	A Boolean (logical) value of TRUE if the input is an NA and FALSE otherwise. This is useful for if statements and the subset() function as NAs cause logical operators to return NA rather than a TRUE or FALSE value
na.omit()	A single dimensional vector	A new vector consisting of all the values in the input, but without the NAs
mean()	A single dimensional vector and a Boolean na.rm argument telling R whether NA values in the vector should be ignored in the calculation of the mean	The mean value of the data (will be NA if the data contains NAs and na.rm = FALSE, which is the default)
sd()	A single dimensional vector and the same na.rm argument used in mean()	The standard deviation of the data
var()	A single dimensional vector and the same na.rm argument used in mean()	The variance of the data (the square of the standard deviation)
sum()	A single dimensional vector and the same na.rm argument used in mean()	The sum of the data