Bio 723

Scientific Computing for Biologists

Paul M. Magwene and Colin S. Maxwell

Fall 2013

Contents

1 Getting your feet wet with R

1.1 Getting Acquainted with R

1.1.1 Installing R

The R website is at http://www.r-project.org/. I recommend that you spend a few minutes checking out the resources, documentation, and links on this page. Download the appropriate R installer for your computer from the Comprehensive R Archive Network (CRAN). A direct link can be found at: http://cran.stat.ucla.edu/. As of mid August 2013 the latest R release is verison 3.0.1.

1.1.2 Starting and R Interactive Session

The OS X and Windows version of R provide a simple GUI interface for using R in interactive mode. When you start up the R GUI you'll be presented with a single window, the R console. See the your textbook, The Art of R Programming (AoRP) for a discussion of the difference between R's interactive and batch modes.

1.1.3 R Studio

R Studio http://www.rstudio.com/ is an open source integrated development environment (IDE) that provides a nicer graphical interface to R than does the default GUI. R Studio also has built in support for various literate programming tools like knitr and Sweave.

1.1.4 Accessing the Help System on R

R comes with fairly extensive documentation and a simple help system. You can access HTML versions of R documentation under the Help menu in the GUI. The HTML documentation also includes information on any packages you've installed. Take a few minutes to browse through the R HTML documentation.

The help system can be invoked from the console itself using the `help` function or the `?` operator.

```
> help(length)
> ?length
> ?log
```

What if you don't know the name of the function you want? You can use the `help.search()` function.

```
> help.search("log")
```

In this case `help.search("log")` returns all the functions with the string ‘log’ in them. For more on `help.search` type `?help.search`. Other useful help related functions include `apropos()` and `example()`.

1.1.5 Navigating Directories in R

When you start the R environment your ‘working directory’ (i.e. the directory on your computer’s file system that R currently ‘sees’) defaults to a specific directory. On Windows this is usually the same directory that R is installed in, on OS X it is typically your home directory. Here are examples showing how you can get information about your working directory and change your working directory.

```
> getwd()
[1] "/Users/pmagwene"
> setwd("/Users")
> getwd()
[1] "/Users"
```

Note that on Windows you can change your working directory by using the `Set Working Directory` item under the `Session` menu in R Studio.

To get a list of the file in your current working directory use the `list.files()` function.

```
> list.files()
[1] "Shared" "pmagwene"
```

1.1.6 Using R as a Calculator

The simplest way to use R is as a fancy calculator.

```
> 10 + 2 # addition
[1] 12
> 10 - 2 # subtraction
[1] 8
> 10 * 2 # multiplication
[1] 20
> 10 / 2 # division
[1] 5
> 10 ^ 2 # exponentiation
[1] 100
> 10 ** 2 # alternate exponentiation
[1] 100
> sqrt(10) # square root
[1] 3.162278
> 10 ^ 0.5 # same as square root
[1] 3.162278
> exp(1) # exponential function
```

```
[1] 2.718282
> 3.14 * 2.5^2
[1] 19.625
> pi * 2.5^2 # R knows about some constants such as Pi
[1] 19.63495
> cos(pi/3)
[1] 0.5
> sin(pi/3)
[1] 0.8660254
> log(10)
[1] 2.302585
> log10(10) # log base 10
[1] 1
> log2(10) # log base 2
[1] 3.321928
> (10 + 2)/(4-5)
[1] -12
> (10 + 2)/4-5 # compare the answer to the above
[1] -2
```

Be aware that certain operators have precedence over others. For example multiplication and division have higher precedence than addition and subtraction. Use parentheses to disambiguate potentially confusing statements.

```
> sqrt(pi)
[1] 1.772454
> sqrt(-1)
[1] NaN
Warning message:
NaNs produced in: sqrt(-1)
> sqrt(-1+0i)
[1] 0+1i
```

What happened when you tried to calculate `sqrt(-1)`?, -1 is treated as a real number and since square roots are undefined for the negative reals, R produced a warning message and returned a special value called `NaN` (Not a Number). Note that square roots of negative complex numbers are well defined so `sqrt(-1+0i)` works fine.

```
> 1/0
[1] Inf
```

Division by zero produces an object that represents infinite numbers.

1.1.7 Comparison Operators

You've already been introduced to the most commonly used arithmetic operators. Also useful are the comparison operators:

```
> 10 < 9  # less than
[1] FALSE
> 10 > 9  # greater than
```

```
[1] TRUE
> 10 <= (5 * 2) # less than or equal to
[1] TRUE
> 10 >= pi # greater than or equal to
[1] TRUE
> 10 == 10 # equals
[1] TRUE
> 10 != 10 # does not equal
[1] FALSE
> 10 == (sqrt(10)^2) # Surprised by the result? See below.
[1] FALSE
> 4 == (sqrt(4)^2) # Even more confused?
[1] TRUE
```

Comparisons return boolean values. Be careful to distinguish between == (tests equality) and = (the alternative assignment operator equivalent to <-).

How about the last two statement comparing two values to the square of their square roots? Mathematically we know that both $(\sqrt{10})^2 = 10$ and $(\sqrt{4})^2 = 4$ are true statements. Why does R tell us the first statement is false? What we're running into here are the limits of computer precision. A computer can't represent $\sqrt{10}$ exactly, whereas $\sqrt{4}$ can be exactly represented. Precision in numerical computing is a complex subject and beyond the scope of this course. Later in the course we'll discuss some ways of implementing sanity checks to avoid situations like that illustrated above.

1.1.8 Working with Vectors in R

Vectors are the core data structure in R. Vectors store an ordered list of items all of the same type. Learning to compute effectively with vectors and one of the keys to efficient R programming. Vectors in R always have a length (accessed with the `length()` function) and a type (accessed with the `typeof()` function).

The simplest way to create a vector at the interactive prompt is to use the `c()` function, which is short hand for 'combine' or 'concatenate'.

```
> x <- c(2,4,6,8)
[1] "double"
> length(x)
[1] 4
> y <- c('joe','bob','fred')
> typeof(y)
[1] "character"
> length(y)
[1] 3
> z <- c() # empty vector
> length(z)
[1] 0
> typeof(z)
[1] "NULL"
```

You can also use `c()` to concatenate two or more vectors together.

```
> v <- c(1,3,5,7)
> w <- c(-1, -2, -3)
> vwx <- c(v,w,x)
> vwx
 [1]  1  3  5  7 -1 -2 -3  2  4  6  8
```

Vector Arithmetic and Comparison

The basic R arithmetic operations work on vectors as well as on single numbers (in fact single numbers *are* vectors).

```
> x <- c(2, 4, 6, 8, 10)
> x * 2
[1]  4  8 12 16 20
> x * pi
[1]  6.283185 12.566371 18.849556 25.132741 31.415927
> y <- c(0, 1, 3, 5, 9)
> x + y
[1]  2  5  9 13 19
> x * y
[1]  0  4 18 40 90
> x/y
[1]      Inf 4.000000 2.000000 1.600000 1.111111
> z <- c(1, 4, 7, 11)
> x + z
[1]  3  8 13 19 11
Warning message:
longer object length
        is not a multiple of shorter object length in: x + z
```

When vectors are not of the same length R 'recycles' the elements of the shorter vector to make the lengths conform. In the example above `z` was treated as if it was the vector `(1, 4, 7, 11, 1)`.

The comparison operators also work on vectors as shown below. Comparisons involving vectors return vectors of booleans.

```
> x > 5
[1] FALSE FALSE  TRUE  TRUE  TRUE
> x != 4
[1]  TRUE FALSE  TRUE  TRUE  TRUE
```

If you try and apply arithmetic operations to non-numeric vectors, R will warn you of the error of your ways:

```
> w <- c('foo', 'bar', 'baz', 'qux')
> w**2
Error in w^2 : non-numeric argument to binary operator
```

Note, however that the comparison operators can work with non-numeric vectors. The results you get will depend on the type of the elements in the vector.

```
>  w == 'bar'
[1] FALSE  TRUE FALSE FALSE
> w < 'cat'
[1] FALSE  TRUE  TRUE FALSE
```

Indexing Vectors

For a vector of length n, we can access the elements by the indices $1 \ldots n$. We say that R vectors (and other data structures like lists) are ‘one-indexed’. Many other programming languages, such as Python, C, and Java, use zero-indexing where the elements of a data structure are accessed by the indices $0 \ldots n - 1$. Indexing errors are a common source of bugs. When moving back and forth between different programming languages keep the appropriate indexing straight!

Trying to access an element beyond these limits returns a special constant called NA (Not Available) that indicates missing or non-existent values.

```
> x <- c(2, 4, 6, 8, 10)
> length(x)
[1] 5
> x[1]
[1] 2
> x[4]
[1] 8
> x[6]
[1] NA
> x[-1]
[1]  4  6  8 10
> x[c(3,5)]
[1]  6 10
```

Negative indices are used to exclude particular elements. x[-1] returns all elements of x except the first. You can get multiple elements of a vector by indexing by another vector. In the example above x[c(3,5)] returns the third and fifth element of x.

Combining Indexing and Comparison

A very powerful feature of R is the ability to combine the comparison operators with indexing. This facilitates data filtering and subsetting. Some examples:

```
> x <- c(2, 4, 6, 8, 10)
> x[x > 5]
[1]  6  8 10
> x[x < 4 | x > 6]
[1]  2  8 10
```

In the first example we retrieved all the elements of x that are larger than 5 (read as ‘x where x is greater than 5’). In the second example we retrieved those elements of x that were smaller than four *or* greater than six. The symbol | is the ‘logical or’ operator. Other logical operators include & (‘logical and’ or ‘intersection’) and !

(negation). Combining indexing and comparison is a powerful concept and one you'll probably find useful for analyzing your own data.

Generating Regular Sequences

Creating sequences of numbers that are separated by a specified value or that follow a particular patterns turns out to be a common task in programming. R has some built-in operators and functions to simplify this task.

```
> s <- 1:10
> s
 [1]  1  2  3  4  5  6  7  8  9 10
> s <- 10:1
> s
 [1] 10  9  8  7  6  5  4  3  2  1
> s <- seq(0.5,1.5,by=0.1)
> s
 [1] 0.5 0.6 0.7 0.8 0.9 1.0 1.1 1.2 1.3 1.4 1.5
# 'by' is the 3rd argument so you don't have to specify it
> s <- seq(0.5, 1.5, 0.33)
> s
[1] 0.50 0.83 1.16 1.49
```

rep() is another way to generate patterned data.

```
> rep(c("Male","Female"),3)
[1] "Male"   "Female" "Male"   "Female" "Male"   "Female"
> rep(c(T,T, F),2)
[1]  TRUE  TRUE FALSE  TRUE  TRUE FALSE
```

1.1.9 Some Useful Functions

You've already seem a number of functions (c(), length(), sin(), log, length(), etc). Functions are called by invoking the function name followed by parentheses containing zero or more *arguments* to the function. Arguments can include the data the function operates on as well as settings for function parameter values. We'll discuss function arguments in greater detail below.

Creating longer vectors

For vectors of more than 10 or so elements it gets tiresome and error prone to create vectors using c(). For medium length vectors the scan() function is very useful.

```
> test.scores <- scan()
1: 98 92 78 65 52 59 75 77 84 31 83 72 59 69 71 66
17:
Read 16 items
> test.scores
 [1] 98 92 78 65 52 59 75 77 84 31 83 72 59 69 71 66
```

When you invoke `scan()` without any arguments the function will read in a list of values separated by white space (usually spaces or tabs). Values are read until `scan()` encounters a blank line or the end of file (EOF) signal (platform dependent). We'll see how to read in data from files below.

Note that we created a variable with the name `test.scores`. If you have previous programming experience you might be surprised that this works. Unlike most languages, R allows you to use periods in variable names. Descriptive variable names generally improve readability but they can also become cumbersome (e.g. `my.long.and.obnoxious.variable.name`). As a general rule of thumb use short variable names when working at the interpreter and more descriptive variable names in functions.

Useful Numerical Functions

Let's introduce some additional numerical functions that are useful for operating on vectors.

```
> sum(test.scores)
[1] 1131
> min(test.scores)
[1] 31
> max(test.scores)
[1] 98
> range(test.scores) # min,max returned as a vec of len 2
[1] 31 98
> sorted.scores <- sort(test.scores)
> sorted.scores
 [1] 31 52 59 59 65 66 69 71 72 75 77 78 83 84 92 98
> w <- c(-1, 2, -3, 3)
> abs(w) # absolute value function
```

1.1.10 Function Arguments in R

Function arguments can specify the data that a function operates on or parameters that the function uses. Some arguments are required, while others are optional and are assigned default values if not specified.

Take for example the `log()` function. If you examine the help file for the `log()` function (type `?log` now) you'll see that it takes two arguments, refered to as '`x`' and '`base`'. The argument `x` represents the numeric vector you pass to the function and is a required argument (see what happens when you type `log()` without giving an argument). The argument `base` is optional. By default the value of `base` is $e = 2.71828\ldots$. Therefore by default the `log()` function returns natural logarithms. If you want logarithms to a different base you can change the `base` argument as in the following examples:

```
> log(2) # log of 2, base e
[1] 0.6931472
> log(2,2) # log of 2, base 2
```

```
[1] 1
> log(2, 4) # log of 2, base 4
[1] 0.5
```

Because base 2 and base 10 logarithms are fairly commonly used, there are convenient aliases for calling `log` with these bases.

```
> log2(8)
[1] 3
> log10(100)
[1] 2
```

1.1.11 Lists in R

R lists are like vectors, but unlike a vector where all the elements are of the same type, the elements of a list can have arbitrary types (even other lists).

```
> l <- list('Bob', pi, 10, c(2,4,6,8))
```

Indexing of lists is different than indexing of vectors. Double brackets (`x[[i]]`) return the element at index i, single bracket return a list containing the element at index i.

```
> l[1] # single brackets
[[1]]
[1] "Bob"

> l[[1]] # double brackets
[1] "Bob"
> typeof(l[1])
[1] "list"
> typeof(l[[1]])
[1] "character"
```

The elements of a list can be given names, and those names objects can be accessed using the `$` operator. You can retrieve the names associated with a list using the `names()` function.

```
> l <- list(name='Bob', age=27, years.in.school=10)
> l
$name
[1] "Bob"

$age
[1] 27

$years.in.school
[1] 10

> l$years.in.school
[1] 10
> l$name
```

```
[1] "Bob"
> names(l)
[1] "name"            "age"             "years.in.school"
```

1.1.12 Simple Input in R

The `c()` and `scan()` functions are fine for creating small to medium vectors at the interpreter, but eventually you'll want to start manipulating larger collections of data. There are a variety of functions in R for retrieving data from files.

The most convenient file format to work with are tab delimited text files. Text files have the advantage that they are human readable and are easily shared across different platforms. If you get in the habit of archiving data as text files you'll never find yourself in a situation where you're unable to retrieve important data because the binary data format has changed between versions of a program.

1.1.13 Using `scan()` to input data

`scan()` itself can be used to read data out of a file. Download the file `algae.txt` from the class website and try the following (after changing your working directory):

```
> algae <- scan('algae.txt')
Read 12 items
> algae
 [1] 0.530 0.183 0.603 0.994 0.708 0.006 0.867 0.059 0.349 0.699 0.983
     0.100
```

One of the things to be aware of when using `scan()` is that if the data type contained in the file can not be coerced to doubles than you must specify the data type using the `what` argument. The `what` argument is also used to enable the use of `scan()` with columnar data. Download `algae2.txt` and try the following:

```
> algae.table <- scan('algae2.txt', what=list('',double(0)))
                        # note use of list argument to what
> algae.table

> algae.table
[[1]]
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov"
[12] "Dec"

[[2]]
 [1] 0.530 0.183 0.603 0.994 0.708 0.006 0.867 0.059 0.349 0.699 0.983
[12] 0.100

> algae.table[[1]]
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov"
[12] "Dec"
> algae.table[[2]]
```

```
 [1] 0.530 0.183 0.603 0.994 0.708 0.006 0.867 0.059 0.349 0.699 0.983
[12] 0.100
```

Use help to learn more about `scan()`.

1.1.14 Using `read.table()` to input data

`read.table()` (and it's derivates - see the help file) provides a more convenient interface for reading tabular data. Download the `turtles.txt` data set from the class wiki. The data in `turtles.txt` are a set of linear measurements representing dimensions of the carapace (upper shell) of painted turtles (*Chrysemys picta*), as reported in Jolicoeur and Mosimmann, 1960; Growth 24: 339-354.

Using the file `turtles.txt`:

```
> turtles <- read.table('turtles.txt', header=T)
> turtles
   sex length width height
1    f     98    81     38
2    f    103    84     38
3    f    103    86     42
  # output truncated
> names(turtles)
[1] "sex"    "length" "width"  "height"
> length(turtles)
[1] 4
> length(turtles$sex)
[1] 48
```

What kind of data structure is `turtles`? What happens when you call the `read.table()` function without specifying the argument `header=T`?

You'll be using the `read.table()}`function frequently. Spend some time reading the documentation and playing around with different argument values (for example, try and figure out how to specify different column names on input).

Note: `read.table()` is more convenient but `scan()` is more efficient for large files. See the R documentation for more info.

1.1.15 Basic Statistical Functions in R

There are a wealth of statistical functions built into R. Let's start to put these to use.

If you wanted to know the mean carapace width of turtles in your sample you could calculate this simply as follows:

```
> sum(turtles$width)/length(turtles$width)
[1] 95.4375
```

Of course R has a built in `mean()` function.

```
mean(turtles$width) [1] 95.4375
```

What if you wanted to calculate the mean of each variables in the data set? R has a set of 'apply' functions (lapply, sapply, mapply, etc) that facilitate applying a function repeatedly to different variables in a list or data frame. `sapply` is the one you'll probably use most often. Here's how to use `sapply` to calculate means for the turtle data set:

```
> sapply(turtles, mean)
      sex    length     width    height
       NA 124.68750  95.43750  46.33333
Warning message:
In mean.default(X[[1L]], ...) :
  argument is not numeric or logical: returning NA
```

Can you figure out why the above produced a warning message? Spend some time reading the documentation for `lapply` and `sapply`, as they will become increasingly handy as you get into writing your own R functions.

Let's take a look at some more standard statistical functions:

```
> min(turtles$width)
[1] 74
> max(turtles$width)
[1] 132
> range(turtles$width)
[1]  74 132
> median(turtles$width)
[1] 93
> summary(turtles$width)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  74.00   86.00   93.00   95.44  102.00  132.00 
> var(turtles$width) # variance
[1] 160.6769
> sd(turtles$width)  # standard deviation
[1] 12.67584
```

1.2 Exploring Univariate Distributions in R

1.2.1 Histograms

One of the most common ways to examine a the distribution of observations for a single variable is to use a histogram. The `hist()` function creates simple histograms in R.

```
> hist(turtles$length) # create histogram with fxn defaults
> ?hist # check out the documentation on hist
```

Note that by default the `hist()` function plots the frequencies in each bin. If you want the probability densities instead set the argument `freq=FALSE`.

```
> hist(turtles$length,freq=F) # y-axis gives probability density
```

Here's some other ways to fine tune a histogram in R.

```
> hist(turtles$length, breaks=12) # use 12 bins
> mybreaks = seq(85,185,8)
> hist(turtles$length, breaks=mybreaks) # specify bin boundaries
> hist(turtles$length, breaks=mybreaks, col='red') # fill the bins with red
```

1.2.2 Density Plots

One of the problems with histograms is that they can be very sensitive to the size of the bins and the break points used. You probably noticed that in the example above as we changes the number of bins and the breakpoints to generate the histograms for the `turtles$length` variable. This is due to the discretization inherent in a histogram. A 'density plot' or 'density trace' is a continuous estimate of a probability distribution from a set of observations. Because it is continuous it doesn't suffer from the same sensitivity to bin sizes and break points. One way to think about a density plot is as the histogram you'd get if you averaged many individual histograms each with slightly different breakpoints.

```
> d <- density(turtles$length)
> plot(d)
```

A density plot isn't entirely parameter free – the parameter you should be most aware of is the 'smoothing bandwidth'.

```
> d <- density(turtles$length) # let R pick the bandwidth
> plot(d,ylim=c(0,0.020)) # gives ourselves some extra headroom on y-axis
> d2 <- density(turtles$length, bw=5) # specify bandwidth
> lines(d2, col='red') # use lines to draw over previous plot
```

The bandwidth determines the standard deviation of the 'kernel' that is used to calculate the density plot. There are a number of different types of kernels you can use; a Gaussian kernel is the R default and is the most common choice. In the example above, R picked a bandwidth of 8.5 (the black line in our plot). When we specified a smaller bandwith of 5, the resulting density plot (red) is less smooth. There exists a statistical literature on picking 'optimum' kernel sizes. In general, larger data sets support the use of smaller kernels. See the R documentation for more info on the `density()` function and references to the literature on density estimators.

The `lattice` package is an R library that makes it easier to create graphics that show conditional distributions. Here's how to create a simple density plot using the `lattice` package.

```
> library(lattice)
> densityplot(turtles$length) # densityplot defined in lattice
```

Notice how by default the `lattice` package also drew points representing the observations along the x-axis. These points have been 'jittered' meaning they've been randomly shifted by a small amount so that overlapping points don't completely hide each other. We could have produced a similar plot, without the lattice package, as so:

```
> d <- density(turtles$length)
> plot(d)
> nobs <- length(turtles$length)
> points(jitter(turtles$length), rep(0,nobs))
```

Notice that in our version we only jittered the points along the x-axis. You can also combine a histogram and density trace, like so:

```
> hist(turtles$length, 10, xlab='Carapace Length (mm)',freq=F)
> d <- density(turtles$length)
> lines(d, col='red', lwd=2) # red lines, with pixel width 2
```

Notice the use of the `freq=F` argument to scale the histogram bars in terms of probability density.

Finally, let's some of the features of `lattice` to produce density plots for the 'length' variable of the turtle data set, conditional on sex of the specimen.

```
> densityplot(~length | sex, data = turtles)
```

There are a number of new concepts here. The first is that we used what is called a 'formula' to specify what to plot. In this case the formula can be read as 'length conditional on sex'. We'll be using formulas in several other contexts and we discuss them at greater length below. The `data` argument allows us to specify a data frame or list so that we don't always have to write arguments like `turtles$length` or `turtles$sex` which can get a bit tedious.

1.2.3 Box Plots

Another common tool for depicting a univariate distribution is a 'box plot' (sometimes called a box-and-whisker plot). A standard box plot depicts five useful features of a set of observations: the median (center most line), the upper and lower quartiles (top and bottom of the box), and the minimum and maximum observations (ends of the whiskers).

There are many variants on box plots, particularly with respect to the 'whiskers'. It's always a good idea to be explicit about what a box plot you've created depicts.

Here's how to create box plots using the standard R functions as well as the lattice package:

```
> boxplot(turtles$length)
> boxplot(turtles$length, col='darkred', horizontal=T) # horizontal version
> title(main = 'Box plot: Carapace Length', ylab = 'Carapace length (mm)')
> bwplot(~length,data=turtles) # using the bwplot function from lattice
```

Note how we used the `title()` function to change the axis labels and add a plot title.

Historical note – The box plot is one of many inventions of the statistician John W. Tukey. Tukey made many contributions to the field of statistics and computer science, particularly in the areas of graphical representations of data and exploratory data analysis.

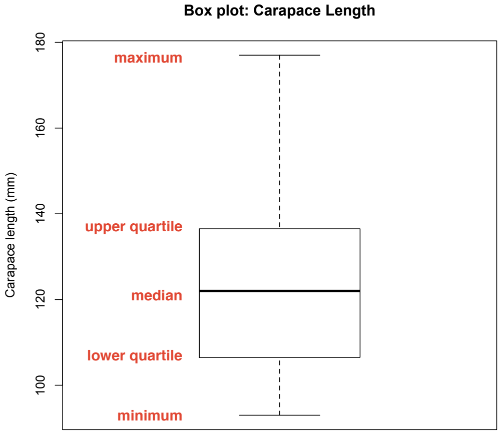

Figure 1.1: A box plot represents a five number summary of a set of observations.

1.2.4 Bean Plots

My personal favorite way to depict univariate distributions is called a 'beanplot'. Beanplots combine features of density plots and boxplots and provide information rich graphical summaries of single variables. The standard features in a beanplot include the individual observations (depicted as lines), the density trace estimated from the observations, the mean of the observations, and in the case of multiple beanplots an overall mean.

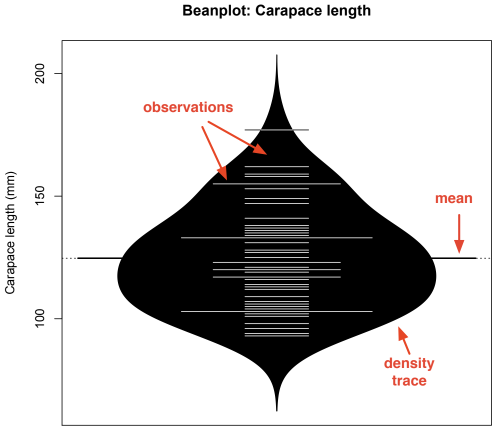

Figure 1.2: Beanplots combine features of density and box plots.

The `beanplot` package is not installed by default. To download it and install it use the R package installer under the `Packages & Data` menu. If this is the first time you use the package installer you'll have to choose a CRAN repository from which to download package info (I recommend you pick one in the US). Once you've done so you can search for 'beanplot' from the Package Installer window. You should also check the 'install dependencies' check box.

Once the beanplot package has been installed check out the examples to see some of the capabilities:

```
> library(beanplot)
```

Note the use of the `library()` function to make the functions in the `beanplot` library available for use. Here's some examples of using the `beanplot` function with the turtle data set:

```
> beanplot(turtles$length) # note the message about log='y'
> beanplot(turtles$length, log='') # DON'T do the automatic log transform
> beanplot(turtles$length, log='', col=c('white','blue','blue','red'))
```

In the final version we specified colors for the parts of the beanplot. See the explanation of the `col` argument int he beanplot function for details.

We can also compare the carapace length variable for male and female turtles.

```
> beanplot(length ~ sex, data = turtles, col=list(c('red'),c('black')),
names = c('females','males'),xlab='Sex', ylab='Caparace length (mm)')
```

Note the use of the formula notation to compare the carapace length variable for males and females. Note the use of the list argument to `col`, and the use of vectors within the list to specify the colors for female and male beanplots.

There is also a asymmetrical version of the beanplot which can be used to more directly compare distributions between two groups. This can be specified by using the argument `side='both'` to the beanplot function.

```
> beanplot(length~sex, data=turtles, col=list(c('red'),c('black')),names=c(
    'females','males'),xlab='Sex', ylab='Carapace length (mm)',side='both')
```

Plots like this one are very convenient for comparing distributions between samples grouped by treatment, sex, species, etc.

We can also create a beanplot with multiple variables in the same plot if the variables are measured on the same scale.

```
> beanplot(turtles$length, turtles$width, turtles$height, log='',
names=c('length','width','height'), ylab='carapace dimensions (mm)')
```

1.2.5 Demo Plots in R

To get a sense of some of the graphical power of R try the `demo()` function:

```
> demo(graphics)
```

1.3 Getting started with literate programming in R

1.3.1 knitr for R

knitr documents weave together documentation/discussion and code into a single document. The pieces of code and documentation are referred to as 'chunks'. Knitr comes with a set of tools that allow you to extract just the code, or to turn the entire document into a nicely formatted report.

You can install knitr using the 'Packages' tab in the R studio IDE or at the command line as follows:

```
install.packages('knitr', dependencies = TRUE)
```

Restart R Studio after installing knitr.

Once knitr is installed, you can create your first knitr document. knitr documents are just plain text files, but R Studio includes some convenient tools to compile such documents in HTML. In R Studio select `New > R Markdown` to create a new knitr document, delete the template text, and and enter the text shown below:

~~~

My First knitr Document
========================

This is very simple knitr document. It includes some *emphasized* and **
bold** text, and a single code chunk.

```{r}
z <- rnorm(30, mean=0, sd=1)
summary(z)
```
~~~

Save this as a markdown file `knit1.Rmd` and 'knit' the document using the `Knit HTML` button in the R Studio IDE. If you entered everything correctly, R Studio will pop up a preview window showing the HTML document that was created from your knitr source code.

As you can see, knitr uses a simple way to markup text (using a formatting convention called 'Markdown'), and code chunks are delineated from text using three backticks. In the HTML output notice that your text blocks includes some formatted italic and bold text, and that the code chunks are shown in grey boxes. Note that there's also a table below the code chunk. This shows the result of evaluating the code chunk.

If you knit the document a second time you'll find that the table output changes slightly. Figure out why this is so by reading the documentation for the `rnorm` function.

A fancier kintr document

Let's get a little bit fancier and show how we can create graphics and use some knitr's formatting features to produce a nicer document.

~~~
My Second knitr Document
========================

This is a still a simple knitr file. However, now it includes several code
chunks, graphics, and mathematical symbols.

## Sampling from the random normal distribution

```{r}
z <- rnorm(30, mean=0, sd=1)
summary(z)
~~~

That code chunk generated a random sample of 30 observations drawn from a
normal distribution with mean zero ($\mu = 0$) and standard deviation one (
$\sigma = 1$).

Note the use of the hashmarks to indicate section headings.

### Mathematical notation

knitr uses standard LaTeX conventions for writing mathematical formulas in
text blocks.

## Generating figures

We canautomatically imbed graphics in our report. For example, the
following will generate a histogram.

```{r}
hist(z)
```
```

For a full overview of knitr’s capabilities see the documentation and examples at the knitr website http://yihui.name/knitr/.
```

## Assignment 1.1

Find an example univariate data set of your choice from the literature. Read it into R using either `scan` or `read.table`. Plot the data using either a histogram or a density plot. Try several different bin widths or kernel sizes. Discuss why you think the plot and bin width/kernel you chose is the best way to represent your data. Also provide an example of a misspecification of the kernel or the bin width. Submit your assignment as a knitr literate programming document. Be sure to include a line of code to read in your data based on a relative path name and submit the accompanying data file.

Note that the `scan` and `read.table` functions can also take a url as input as illustrated below:

```
> algae <- scan(url("http://biology.duke.edu/magwenelab/algae.txt", "r"))
```

Unfortunately, that only works with normal HTTP connections, *not* HTTPS, so you can't open files directly from the Bio 723 Github site.

2 Bivariate Data

2.1 Plotting Bivariate Data in R

Let's use a dataset called iris (included in the standard R distribution) to explore bivariate relationships between variables. This data set was made famous by R. A. Fisher who used it to illustrate many of the fundamental statistical methods he developed. The data set consists of four morphometric measurements on specimens of three different iris species. Use the R help to read about the iris data set (?iris). We'll be using this data set repeatedly in future weeks so familiarize yourself with it.

```
> ?iris
> names(iris)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
[5] "Species"
> unique(iris$Species)
[1] setosa     versicolor virginica
Levels: setosa versicolor virginica
> dim(iris)
[1] 150   5
```

2.1.1 Bivariate scatter plots

We'll start with the conventional 'variable space' representation of bivariate relationships – the scatter plot.

```
> plot(iris$Sepal.Length, iris$Sepal.Width)
```

This plots Sepal Length on the x-axis and Petal Length on the y-axis. Here's an alternate way to generate the same plot:

```
> plot(Petal.Length ~ Sepal.Length, data = iris)
```

Did you notice what is different between the two versions above? In the second version, you can think of the tilde ('~') as short-hand for 'function of'. So the plotting call above can be translated roughly as "Plot Petal.Length as a function of Sepal.Length, where these variables can be found in the iris data set".

From these plot it is immediately obvious that these two variables are positively associated (i.e. when one increases the other tends to increase). You will also notice there seem to be distinct clusters of points in the plot. Recall that the iris data set consists of three different species. Let's regenerate the plot, this time coloring the points according to the species names. First, let's note that the Species column is a categorical variable, which in R we refer to as a 'factor'.

```
> iris$Species
  [1] setosa     setosa     setosa     setosa ...
 [51] versicolor versicolor versicolor versicolor ...
[101] virginica  virginica  virginica  virginica  ...
  ....
Levels: setosa versicolor virginica
> is.factor(iris$Species)
[1] TRUE
> levels(iris$Species)
[1] "setosa"     "versicolor" "virginica"
> nlevels(iris$Species)
[1] 3
> typeof(iris$Species)
[1] "integer"
```

The `is.factor()` function tests whether a vector is a factor, the `levels()` function returns the categorical labels associated with the factor, and `nlevels()` gives the total number of levels. Factor levels are represented internally as integers, as the `typeof()` function call illustrates. You can use the function `unclass()` to show the corresponding integer representations for a vector of factors:

```
> unclass(iris$Species)
  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...
 [59] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 ...
[117] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 ...
attr(,"levels")
[1] "setosa"     "versicolor" "virginica"
```

As you can see, the ‘setosa’ specimens have the value 1, ‘versicolor’ have the value 2, and ‘virginica’ the value 3.

Because of the mapping between factor levels and integers, we can use a variable of factors as indices into another vector, effectively creating a mapping between the factor levels, and the elements of the vector that is being indexed. This is shown below:

```
> clrs <- c('red','green','blue')
> clrs[iris$Species]
  [1] "red"   "red"   "red"   "red"   "red"   "red"   "red" ...
 [57] "green" "green" "green" "green" "green" "green" "green" ...
 [99] "green" "green" "blue"  "blue"  "blue"  "blue"  "blue" ...
```

With that mapping in mind, let’s reconstruct our scatter plot:

```
> plot(Petal.Length ~ Sepal.Length, data = iris, col = clrs[iris$Species],
    main="Petal Length vs. Sepal Length")
> legend( "topleft", pch = 1, col = clrs, legend = levels(iris$Species ))
```

In addition to plotting and coloring the bivariate scatter, we added a title to the plot using the `main` argument and created a legend, using the `legend()` function. Your output should look like Figure 2.1.

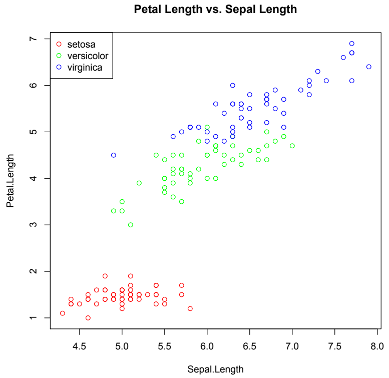

Figure 2.1: Scatter plot created from the iris data set using the `plot` function.

2.2 Introducing ggplot2

Pretty much any statistical plot can be thought of as a mapping between data and one or more visual representations. For example, in a bivariate scatter plot we map two ordered sets of numbers (the variables of interest) to points in the Cartesian plane (x,y-coordinates). In our example above, we further embellished our plot with another mapping in which we mapped the Species labels to different colors.

This notion of representing plots in terms of their mappings is a powerful idea which is central to an approach for plotting that is represented in the R package `ggplot2`.

2.2.1 Installing ggplot2

Like all R packages, `ggplot2` can be installed either from the command line or via the GUI. Here's a reminder of how to do so from the command line:

```
> install.packages("ggplot2", dependencies=T)
```

2.2.2 Aesthetic and Geometric mappings in ggplot2

ggplot2 considers two types of mappings from data to visual representations: 1) 'aesthetic mappings', which determine the way that data are represented in a plot (e.g. symbols, colors) and 2) 'geometry' or 'geom' mappings which determine the type of geometric representation that a plot uses.

The primary plotting function in `ggplot2` is `ggplot()`. The first argument to `ggplot` is always a data frame. The data frame is the one that ggplot will use to look for all the mappings that you define in the subsequent pieces of the plot. The nice thing about this is that there is no need to use the dollar sign notation. As you've seen, you can get similar behavior in base plots by specifying the 'data' argument.

The second argument to `ggplot()` is always a function called `aes()`. `aes()` takes named arguments. Each argument name is the 'aesthetic' that you want mapped to a particular variable (column) in the data.

The final piece of information that we need to draw our plot is the 'geom'. All geoms are encoded as R functions. The syntax used to add them to a plot is simply a '+' sign. There are many different ggplot geoms for different plot types. We'll explore a few of the built-in geoms in this chapter; additional geoms will come up in later weeks.

2.2.3 Scatter plots using ggplot2

Let's recreate our iris scatter plot using the function `ggplot` from the ggplot2 library:

```
> library(ggplot2)
> ggplot(iris, aes(x = Sepal.Length, y = Petal.Length,
                 col = Species)) + geom_point()
```

Following the requirement outline above, `iris` is our data frame, the call to `aes` set's up our aesthetic mapping, and we're specifying the use of the point geom (`geom_point()`) to map the x- and y-values in the aesthetic mapping to points in the Cartesian plane. In the function call above, we told `ggplot` that we wanted the sepal length on the x axis, the petal length on the y axis, and the colors to be encoded by the species. However, we could choose any number of other aesthetic mappings. For example, could use shape instead of color to represent the Species labels:

```
> ggplot(iris, aes(x = Sepal.Length, y = Petal.Length,
                 shape = Species)) + geom_point()
```

or alternately, size:

```
> ggplot(iris, aes(x = Sepal.Length, y = Petal.Length,
                 size = Species)) + geom_point()
```

We can even combine multiple aesthetics in a single plot:

```
> ggplot(iris, aes(x = Sepal.Length, y = Petal.Length,
                 col = Species, shape = Species)) + geom_point()
```

The resulting plot is shown in Figure 2.2.

There's a number of advantages to using `ggplot` rather than trying to replicate this plot with base graphics functions in R:

1. The legend is automatically drawn for you.
2. The code is very easy to change. Rather than having to figure out how to manually map a point size onto a variable using some difficult R code, it's just as simple as saying to set the 'size' equal to a 'variable'.
3. It's easy to swap around variables from one aesthetic mapping to another.

Having a good understanding of both the base plotting functions and a powerful package like `ggplot2` allows you maximum flexibility in terms of the statistical graphics you are able to produce.

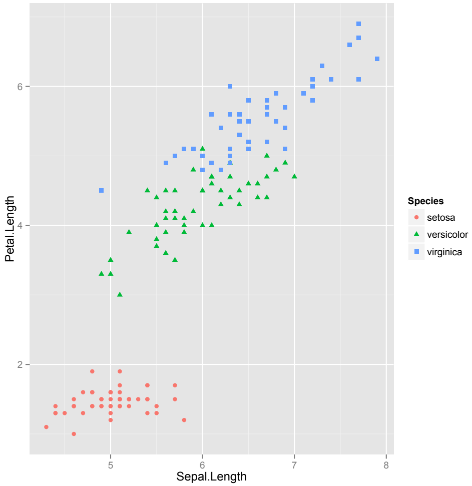

Figure 2.2: Scatter plot created from the iris data set using the `ggplot` function.

2.2.4 Some additional ggplot geoms

So far we've only looked at a single geom (`geom_point()`). Let's revisiting some of the univariate plots from last week using ggplot.

Boxplots `geom_boxplot()` constructs boxplots in ggplot.

```
> ggplot(iris, aes(x = Species, y = Sepal.Length, col=Species)) +
      geom_boxplot()
```

Histograms `geom_histogram()` is used to construct histogram plots in ggplot.

```
> ggplot(iris, aes(x = Sepal.Length)) + geom_histogram()
```

Here we let `ggplot` pick the default bin widths. Below we show how to change the bin width:

```
> ggplot(iris, aes(x = Sepal.Length)) + geom_histogram(binwidth=0.25)
```

If we want to color histogram by species identity you need to set the `position = 'identity'` in the call to `geom_histogram`:

```
> ggplot(iris, aes(x = Sepal.Length, fill=Species)) +
        geom_histogram(binwidth=0.25, position='identity',alpha=0.65)
```

The above code also set the transparency of the bar fills using the `alpha` argument. As an alternative to overlaying the histogram bins for each species, you can show the bins side-by-side using the argument `position = 'dodge'`.

```
> ggplot(iris, aes(x = Sepal.Length, fill=Species)) +
        geom_histogram(binwidth=0.25, position='dodge')
```

Density plots geom_density() creates density plots in ggplot.

```
> ggplot(iris, aes(x = Sepal.Length, fill=Species)) +
        geom_density(alpha=0.65)
```

There's also a 2D version of the density plot, created using geom_density2d(). This can be usefully combined with geom_points() to create a bivariate scatter plot with density contours.

```
> ggplot(iris, aes(x = Sepal.Length, y = Petal.Length, col = Species)) +
    geom_point() + geom_density2d(alpha=0.25)
```

Scatter plots with marginal density plots The file scatterWithMargins.R from the course wiki contains a function that uses multiple calls to ggplot() to combine two marginal density plots with a scatter plot. To use this function you'll need to install a package called "gridExtra":

```
> install.packages("gridExtra", dependencies=T)
```

Then import the new function from scatterWithMargins.R and use it as so:

```
> source('scatterWithMargins.R')
> scatterWithMargins(iris, "Sepal.Length", "Petal.Length", "Species")
```

This produces the plot shown in Figure 2.3.

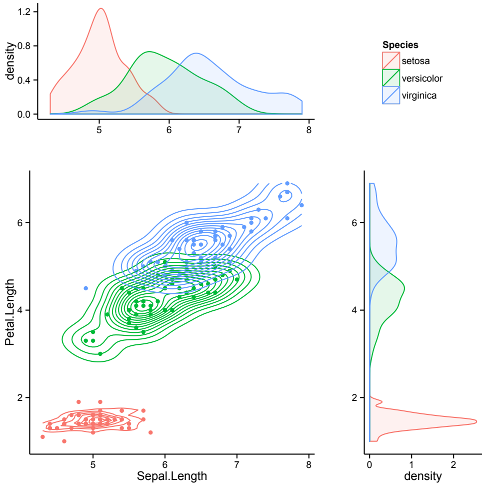

Figure 2.3: Figure produced by the scatterWithMargins function from the course wiki.

2.3 Vector Mathematics in R

As you saw last week R vectors support basic arithmetic operations that correspond to the same operations on geometric vectors. For example:

```
> x <- 1:15
> y <- 10:24
> x
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
> y
 [1] 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

> x + y             # vector addition
 [1] 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
> x - y             # vector subtraction
 [1] -9 -9 -9 -9 -9 -9 -9 -9 -9 -9 -9 -9 -9 -9 -9
> x * 3             # multiplication by a scalar
 [1]  3  6  9 12 15 18 21 24 27 30 33 36 39 42 45
```

R also has an operator for the dot product, denoted `%*%`. This operator also designates matrix multiplication, which we will discuss next week. By default this operator returns an object of the R matrix class. If you want a scalar (or the R equivalent of a scalar, i.e. a vector of length 1) you need to use the `drop()` function.

```
> z <- x %*% x
> class(z)      # note use of class() function
[1] "matrix"
> z
     [,1]
[1,] 1240
> drop(z)
[1] 1240
```

In lecture we saw that many useful geometric properties of vectors could be expressed in the form of dot products. Let's start with some two-dimensional vectors where the geometry is easy to visualize:

```
> a <- c(1, 0) # the point (1,0)
> b <- c(0, 1) # the point (0,1)
```

Now let's draw our vectors:

```
# create empty plot w/specified x- and y- limits
# the 'asp=1' argument maintains the scaling of the x- and y-axes
# so that units are equivalent for both axes (i.e. squares remain squares)
> plot(c(-2,2),c(-1,2),type='n', asp=1)

# draw an arrow from origin (0,0) to x,y coordinates of vector "a"
# the length argument changes the size of the arrowhead
# use the R help to read more about the arrows function
> arrows(0, 0, a[1], a[2], length=0.1)
```

```
# and now for the vector "b"
> arrows(0, 0, b[1], b[2], length=0.1)
```

You should now have a figure that looks like the one below: Let's see what the dot product can tell us about these vectors. First recall that we can calculate the length of a vector as the square-root of the dot product of the vector with itself ($|\vec{a}|^2 = \vec{a} \cdot \vec{a}$)

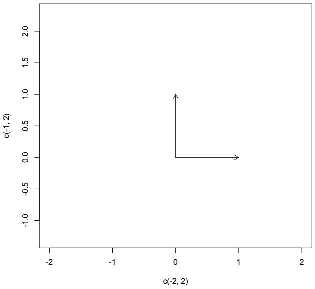

Figure 2.4: A simple vector figure.

```
> len.a <- drop(sqrt(a %*% a))
> len.a
[1] 1
> len.b <- drop(sqrt(b %*% b))
```

How about the angle between a and b?

```
> dot.ab <- a %*% b
> dot.ab
     [,1]
[1,]    0
> cos.ab <- (a %*% b)/(len.a * len.b)
> cos.ab
     [,1]
[1,]    0
```

A key point to remember dot product of two vectors is zero if, and only if, they are orthogonal to each other (regardless of their dimension).

2.4 Writing Functions in R

So far we've been mostly using R's built in functions. However the power of a true programming language is the ability to write your own functions.

The general form of an R function is as follows:

```
funcname <- function(arg1, arg2) {
 # one or more expressions
 # last expression is the object returned
 # or you can explicitly return an object
}
```

To make this concrete, here's an example where we define a function in the interpreter and then put it to use:

```
> my.dot <- function(x,y){
+ # don't type the '+' symbols, these show continuation lines
+   return(sum(x*y))
+ }

> a <- 1:5
> b <- 6:10
> a
[1] 1 2 3 4 5
> b
[1]  6  7  8  9 10
> my.dot(a,b)
[1] 130
> my.dot
function(x,y){
  return(sum(x*y))
}
```

If you type a function name without parentheses R shows you the function's definition. This works for built-in functions as well (thought sometimes these functions are defined in C code in which case R will tell you that the function is a '.Primitive').

2.4.1 Putting R functions in Scripts

When you define a function at the interactive prompt and then close the interpreter your function definition will be lost. The simple way around this is to define your R functions in a script that you can than access at any time.

I R Studio choose `File > New > R Script`. This will bring up a blank editor window. Enter your function into the editor and save the source file in your R working directory with a name like `vecgeom.R`.

```
# functions defined in vecgeom.R

veclength <- function(x) {
  # Given a numeric vector, returns length of that vector
  sqrt(drop(x %*% x))
}

unitvector <- function(x) {
  # Return a unit vector in the same direction as x
  x/veclength(x)
}
```

There are two functions defined above, one of which calls the other. Both take single vector arguments. These functions have no error checking to insure that the arguments passed to the functions are reasonable but R's built in error handling will do just fine for most cases.

Once your functions are in a script file you can make them accesible by using the `source()` function (See also the `Source` tab button in the R Studio GUI):

```
> source("vecgeom.R")
> x <- c(1,0.4)
> veclength(x)
[1] 1.077033
> ux <- unitvector(x)
> ux
[1] 0.9284767 0.3713907
> veclength(ux)
[1] 1
> a
[1] 1 2 3 4 5
> veclength(a)
[1] 7.416198
> ua <- unitvector(a)
> ua
[1] 0.1348400 0.2696799 0.4045199 0.5393599 0.6741999
> veclength(ua)
[1] 1
```

Note that our functions work with vectors of arbitrary dimension.

Assignment 2.1

Write a function that uses the dot product and the `acos()` function to calculate the angle (in radians) between two vectors of arbitrary dimension. By default, your function should return the angle in radians. Also include a logical (Boolean) argument that will return the answer in degrees. Test your function with the following two vectors: `x = [-3, -3, -1, -1, 0, 0, 1, 2, 2, 3]` and `y = [-8, -5, -3, 0, -1, 0, 5, 1, 6, 5]`. The expected angle for these test vectors is 0.441 radians (25.3 degrees).

Let's also add the following function to `vecgeom.R` to aid in visualizaing 2D vectors:

```
draw.vectors <- function(a, b, colors=c('red', 'blue'), clear.plot=TRUE){

    # figure out the limits such that the origin and the vector
    # end points are all included in the plot
    xhi <- max(0, a[1], b[1])
    xlo <- min(0, a[1], b[1])
    yhi <- max(0, a[2], b[2])
    ylo <- min(0, a[2], b[2])

    xlims <- c(xlo, xhi)*1.10 # give a little breathing space around
        vectors
    ylims <- c(ylo, yhi)*1.10
    if (clear.plot){
        plot(xlims, ylims, type='n', asp=1, xlab="x-coord", ylab="y-coord")
    }
    arrows(0, 0, a[1], a[2], length=0.1, col=colors[1])
    arrows(0, 0, b[1], b[2], length=0.1, col=colors[2])
}
```

You can use this new function as follows:

```
# you need to source the file everytime you change it
> source("/Users/pmagwene/Downloads/vecgeom.R")
> x <- c(1,0.4)
> y <- c(0.2, 0.8)
> draw.vectors(x,y)  # draw the original vectors
```

The resulting figure should resemble the one below.

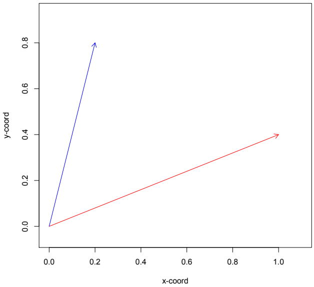

Figure 2.5: Another vector figure.

Notice that we included a `clear.plot` argument in our `draw.vectors` function. I included this so we could add additional vectors to our plot, without overwriting the old vectors, as demonstrated below:

```
# draw the unit vectors that point in the same directors as the original
    vectors
> ux <- unitvector(x)
> uy <- unitvector(y)
> draw.vectors(ux, uy, colors=c('black', 'green'), clear.plot=F)
```

Unlike the other functions we wrote, `draw.vectors` only works properly with 2D vectors. Since any pair of vectors defines a plane, it is possible to generalize this function to work with arbitrary pairs of vectors.

Assignment 2.2

Write a function, `vproj()`, that takes two vectors, $\vec{x}$ and $\vec{y}$, and returns a list containing the projection of $\vec{y}$ on $\vec{x}$ and the component of $\vec{y}$ in $\vec{x}$:

$$P_{\vec{x}}(\vec{y}) = \left(\frac{\vec{x} \cdot \vec{y}}{|\vec{x}|}\right) \frac{\vec{x}}{|\vec{x}|}$$

and

$$C_{\vec{x}}(\vec{y}) = \frac{\vec{x} \cdot \vec{y}}{|\vec{x}|}$$

Use the test vectors from Assignment 2.1 to test your function. The list returned by your function for these test vectors should resemble that shown below:

```
> vproj(x, y)

$proj
 [1] -6 -6 -2 -2  0  0  2  4  4  6

$comp
[1] 12.32883
```

2.5 Vector Geometry of Correlation and Regression

Let's return to our use of the dot product to explore the relationship between variables. First let's add a function to our module, `vecgeom.R`, to calculate the cosine of the angle between to vectors.

```
# add to vecgeom.R

vec.cos <- function(x,y) {
  # Calculate the cos of the angle between vectors x and y
  len.x <- veclength(x)
  len.y <- veclength(y)
  return( (x %*% y)/(len.x * len.y) )
}
```

We can then use this function to examine the relationships between the variables in the iris dataset. For now let's just work with the *I. setosa* specimens. Read the help file for `subset()`.

```
> setosa <- subset(iris, Species == 'setosa', select = -Species)
> dim(setosa)
[1] 50  4
> names(setosa)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
```

Often times it's useful to look at many bivariate relationships simultaneously. The `pairs()` function allows you to do this:

```
> pairs(setosa)
```

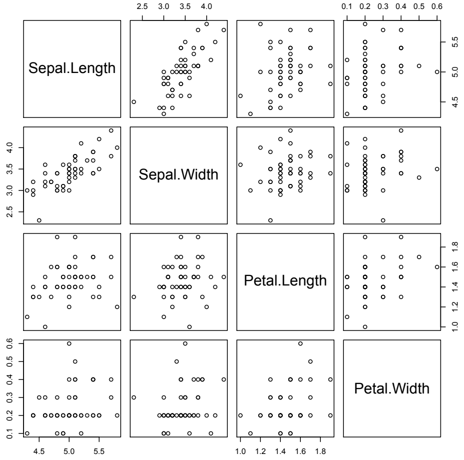

Figure 2.6: Output of the `pairs()` function for the *I. setosa* specimens in the `iris` dataset.

First we'll center the setosa dataset using the `scale()` function. `scale()` has two logical arguments `center` and `scale`. By default both are `TRUE` which will center *and* scale the variables. But for now we just want to center the data. `scale()` returns a matrix object so we use the `data.frame` function to cast the object back to a data frame.

```
> source("/Users/pmagwene/Downloads/vecgeom.R")
> ctrd <- scale(setosa,center=T,scale=F)
> class(ctrd)
[1] "matrix"
> names(ctrd)
NULL
> ctrd <- data.frame(scale(setosa,center=T,scale=F))
> class(ctrd)
[1] "data.frame"
> names(ctrd)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
> vec.cos(ctrd$Sepal.Length, ctrd$Sepal.Width)
          [,1]
[1,] 0.7425467
> vec.cos(ctrd$Sepal.Length, ctrd$Petal.Length)
          [,1]
[1,] 0.2671758
> vec.cos(ctrd$Sepal.Length, ctrd$Petal.Width)
          [,1]
[1,] 0.2780984
```

Consider the values above in the context of the scatter plots you generated with the `pairs()` function; and then recall that for mean-centered variables, $\texttt{cor}(X,Y) = r_{XY} = cos\,\theta = \frac{\vec{x}\cdot\vec{y}}{|\vec{x}||\vec{y}|}$. So our `vec.cos()` function, when applied to centered data, is equivalent to calculating the correlation between x and y. Let's confirm this using the built in `cor()` function in R:

```
> cor(setosa$Sepal.Length, setosa$Sepal.Width)
[1] 0.7425467
> cor(setosa)  # called like this will calculate all pairwise correlations
             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length    1.0000000   0.7425467    0.2671758   0.2780984
Sepal.Width     0.7425467   1.0000000    0.1777000   0.2327520
Petal.Length    0.2671758   0.1777000    1.0000000   0.3316300
Petal.Width     0.2780984   0.2327520    0.3316300   1.0000000
```

2.5.1 Bivariate Regression in R

R has a flexible built in function, `lm()` for fitting linear models. Bivariate regression is the simplest case of a linear model.

```
> setosa.lm <- lm(Sepal.Width ~ Sepal.Length, data=setosa)
> class(setosa.lm)
[1] "lm"
> names(setosa.lm)
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"
> coef(setosa.lm)
 (Intercept) Sepal.Length
  -0.5694327    0.7985283
```

The function `coef()` will return the intercept and slope of the line representing the bivarariate regression. For a more complete summary of the linear model you've fit use the `summary()` function:

```
> summary(setosa.lm)

Call:
lm(formula = Sepal.Width ~ Sepal.Length, data = setosa)

Residuals:
     Min       1Q   Median       3Q      Max
-0.72394 -0.18273 -0.00306  0.15738  0.51709

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   -0.5694     0.5217  -1.091    0.281
Sepal.Length   0.7985     0.1040   7.681 6.71e-10 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘’. 0.1 ‘’  1
```

```
Residual standard error: 0.2565 on 48 degrees of freedom
Multiple R-squared: 0.5514, Adjusted R-squared: 0.542
F-statistic: 58.99 on 1 and 48 DF,  p-value: 6.71e-10
```

As demonstrated above, the `summary()` function spits out key diagnostic information about the model we fit. Now let's create a plot illustrating the fit of the model.

```
> plot(Sepal.Width ~ Sepal.Length, data=setosa, xlab="Sepal Length (cm)",
    ylab="Sepal Width (cm)", main="Iris setosa")
> abline(setosa.lm, col='red', lwd=2, lty=2)  # see ?par for info about lwd
     and lty
```

Your output should resemble the figure below. Note the use of the function `abline()` to plot the regression line. Calling `plot()` with an object of class `lm` shows a series of diagnostic plots. Try this yourself.

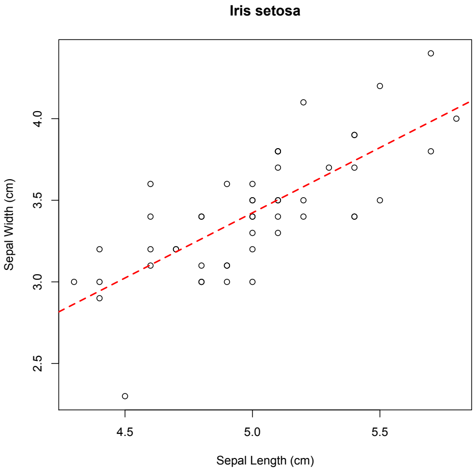

Figure 2.7: Linear regression of Sepal Width on Sepal Length for *I. setosa*.

Assignment 2.3

Write your own regression function (i.e. your code shouldn't refer to the built in regression functions) for mean centered vectors in R. The function will take as it's input two vectors, $\vec{x}$ and $\vec{y}$. The function should return:

1. a list containing the mean-centered versions of these vectors
2. the regression coefficient b in the mean centered regression equation $\hat{\vec{y}} = b\vec{x}$
3. the coefficient of determination, R^2

Demonstrate your regression function by using it to carry out regressions of Sepal.Length on Sepal.Width separately for the 'versicolor' and 'virginica' specimens from the iris data set. Include `ggplot` created plots in which you use the `geom_point()` and `geom_abline()` functions to illustrate your calculated regression line. To test your function, compare your regression coefficients and coefficient of determination to the same values returned by the built in `lm()` function.

3 Matrices and matrix operations in R

3.1 Matrices in R

In R matrices are two-dimensional collections of elements all of which have the same mode or type. This is different than a data frame in which the columns of the frame can hold elements of different type (but all of the same length), or from a list which can hold objects of arbitrary type and length. Matrices are more efficient for carrying out most numerical operations, so if you're working with a very large data set that is amenable to representation by a matrix you should consider using this data structure.

3.1.1 Creating matrices in R

There are a number of different ways to create matrices in R. For creating small matrices at the command line you can use the `matrix()` function.

```
> X <- matrix(1:5)
> X
      [,1]
 [1,]    1
 [2,]    2
 [3,]    3
 [4,]    4
 [5,]    5
> X <- matrix(1:12, nrow=4)
> X
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> dim(X) # give the shape of the matrix
[1] 4 3
```

`matrix()` takes a data vector as input and the shape of the matrix to be created is specified by using the `nrow` and `ncol` arguments. If the number of elements in the input data vector is less than `nrows` × `ncols` the elements will be 'recycled' as discussed in previous chapters. Without any shape arguments the `matrix()` function will create a column vector as shown above. By default the `matrix()` function fills in the matrix in a column-wise fashion. To fill in the matrix in a row-wise fashion use the argument `byrow=T`.

If you have a pre-existing data set in a list or data frame you can use the `as.matrix()` function to convert it to a matrix.

```
> turtles <- read.table('turtles.txt', header=T)
> tmtx <- as.matrix(turtles)
> head(tmtx)   # see ?head and ?tail
     sex length width height
[1,] "f" " 98"  " 81" "38"
[2,] "f" "103"  " 84" "38"
[3,] "f" "103"  " 86" "42"
[4,] "f" "105"  " 86" "40"
[5,] "f" "109"  " 88" "44"
[6,] "f" "123"  " 92" "50"
# NOTE: the elements were all converted to character

> tmtx <- as.matrix(subset(turtles, select=-sex))
> head(tmtx)
  length width height
1     98    81     38
2    103    84     38
3    103    86     42
4    105    86     40
5    109    88     44
6    123    92     50
# This is probably more along the lines of what you want
```

You can use the various indexing operations to get particular rows, columns, or elements. Here are some examples:

```
> X <- matrix(1:12, nrow=4)
> X
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> X[1,] # get the first row
[1] 1 5 9
> X[,1] # get the first column
[1] 1 2 3 4
> X[1:2,] # get the first two rows
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
> X[,2:3] # get the second and third columns
     [,1] [,2]
[1,]    5    9
[2,]    6   10
[3,]    7   11
[4,]    8   12
```

```
> Y <- matrix(1:12, byrow=T, nrow=4)
> Y
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
> Y[4] # see explanation below
[1] 10
> Y[5]
[1] 2
> dim(Y) <- c(2,6)
> Y
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    7    2    8    3    9
[2,]    4   10    5   11    6   12
> Y[5]
[1] 2
```

The example above where we create a matrix Y is meant to show that matrices are stored internally in a column wise fashion (think of the columns stacked one atop the other), regardless of whether we use the byrow=T argument. Therefore using single indices returns the elements with respect to this arrangement. Note also the use of assignment operator in conjuction with the dim() function to reshape the matrix. Despite the reshaping, the internal representation in memory hasn't changed so Y[5] still gives the same element.

You can use the diag() function to get the diagonal of a matrix or to create a diagonal matrix as show below:

```
> Z <- matrix(rnorm(16), ncol=4)
> Z
           [,1]       [,2]         [,3]        [,4]
[1,] -1.7666373  2.1353032 -0.903786375 -0.70527447
[2,] -0.9129580  1.1873620  0.002903752  0.51174408
[3,] -1.5694273 -0.5670293 -0.883259848  0.05694691
[4,]  0.9903785 -1.6138958  0.408543336  2.39152400
> diag(Z)
[1] -1.7666373  1.1873620 -0.8832598  2.3915240
> diag(5) # create the 5 x 5 identity matrix
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    1
> s <- sqrt(10:13)
> diag(s)
         [,1]     [,2]     [,3]     [,4]
[1,] 3.162278 0.000000 0.000000 0.000000
```

```
[2,] 0.000000 3.316625 0.000000 0.000000
[3,] 0.000000 0.000000 3.464102 0.000000
[4,] 0.000000 0.000000 0.000000 3.605551
```

Matrix operations in R

The standard mathematical operations of addition and subtraction and scalar multiplication work element-wise for matrices in the same way as they did for vectors. Matrix multiplication uses the operator `%*%` which you saw last week for the dot product. To get the transpose of a matrix use the function `t()`. The `solve()` function can be used to get the inverse of a matrix (assuming it's non-singular) or to solve a set of linear equations.

```
> A <- matrix(1:12, nrow=4)
> A <- matrix(1:12, nrow=4)
> A
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> t(A)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
> B <- matrix(rnorm(12), nrow=4)
> B
           [,1]        [,2]        [,3]
[1,] -2.9143953  0.38204730 -1.33207235
[2,]  0.1778266 -0.44563686  0.76143612
[3,]  1.7226235  0.03320553 -0.06652767
[4,]  0.5291281 -0.13145408  0.14108766
> A + B
          [,1]     [,2]      [,3]
[1,] -1.914395 5.382047  7.667928
[2,]  2.177827 5.554363 10.761436
[3,]  4.722623 7.033206 10.933472
[4,]  4.529128 7.868546 12.141088
> A - B
         [,1]     [,2]      [,3]
[1,] 3.914395 4.617953 10.332072
[2,] 1.822173 6.445637  9.238564
[3,] 1.277377 6.966794 11.066528
[4,] 3.470872 8.131454 11.858912
> 5 * A
     [,1] [,2] [,3]
[1,]    5   25   45
```

```
[2,]   10   30   50
[3,]   15   35   55
[4,]   20   40   60
> A %*% B  # do you understand why this generated an error?
Error in A %*% B : non-conformable arguments
> A %*% t(B)
          [,1]     [,2]     [,3]     [,4]
[1,] -12.99281 4.802567 1.289902 1.141647
[2,] -16.85723 5.296193 2.979203 1.680408
[3,] -20.72165 5.789819 4.668505 2.219170
[4,] -24.58607 6.283445 6.357806 2.757932
> C <- matrix(1:16, nrow=4)
> solve(C)  # not all square matrices are invertible!
Error in solve.default(C) : Lapack routine dgesv: system is exactly
    singular
> C <- matrix(rnorm(16), nrow=4)  # you'll get a different matrix than I
    did
> C
           [,1]       [,2]       [,3]       [,4]
[1,] -1.6920758 -0.8104245  0.9940420  0.3592050
[2,]  1.5949448 -0.9508142 -0.1960434 -0.5678855
[3,] -1.2443831  0.6400100  0.2645679 -0.8733987
[4,]  0.2129116  0.6719323  0.7494698 -0.3856085
> Cinv <- solve(C)  # this should return something that looks like an
    identity matrix
> C %*% Cinv
             [,1]          [,2]          [,3]          [,4]
[1,] 1.000000e+00 -2.360850e-17  6.193505e-17  4.189425e-18
[2,] 2.710844e-17  1.000000e+00  3.577867e-18 -7.264493e-17
[3,] 4.944640e-17  7.643625e-17  1.000000e+00  5.134714e-17
[4,] 1.978161e-17 -1.187201e-17 -4.022390e-17  1.000000e+00
> all.equal(C %*% Cinv, diag(4)) # test approximately equality
[1] TRUE
```

We expect that CC^{-1} should return the above should return the 4×4 identity matrix. As shown above this is true up to the approximate floating point precision of the machine you're operating on.

3.2 Descriptive statistics as matrix functions

Assume you have a data set represented as a $n \times p$ matrix, X, with observations in rows and variables in columns. Below I give formulae for calculating some descriptive statistics as matrix functions.

3.2.1 Mean vector and matrix

You can calculate a row vector of means, **m**, as:

$$\mathbf{m} = \frac{1}{n}\mathbf{1}^T X$$

where 1 is a $n \times 1$ vector of ones.

A $n \times p$ matrix M where each column is filled with the mean value for that column is:

$$M = \mathbf{1}\mathbf{m}$$

3.2.2 Deviation matrix

To re-express each value as the deviation from the variable means (i.e. each columns is a mean centered vector) we calculate a deviation matrix:

$$D = X - M$$

3.2.3 Covariance matrix

The $p \times p$ covariance matrix can be expressed as a matrix product of the deviation matrix:

$$S = \frac{1}{n-1} D^T D$$

3.2.4 Correlation matrix

The correlation matrix, R, can be calculated from the covariance matrix by:

$$R = VSV$$

where V is a $p \times p$ diagonal matrix where $V_{ii} = 1/\sqrt{S_{ii}}$.

3.2.5 Concentration matrix and Partial Correlations

If the covariance matrix, S is invertible, than inverse of the covariance matrix, S^{-1}, is called the 'concentration matrix' or 'precision matrix'. We can relate the concentration matrix to partial correlations as follow. Let

$$P = S^{-1}$$

Then:

$$\text{corr}(x_i, x_j \mid X \backslash \{x_i, x_j\}) = -\frac{p_{ij}}{\sqrt{p_{ii} p_{jj}}}$$

where $X \backslash \{x_i, x_j\}$ indicates all variables other than x_j and x_i. You can read this as 'the correlation between x and y conditional on all other variables.'

Assignment 3.1

Create an R library that includes functions that use matrix operations to calculate each of the descriptive statistics discussed above (except the concentration matrix / partial correlations). Calculate these statistics for `iris` data set and check the results of your functions against the built-in R functions.

3.3 Visualizing Multivariate data in R

Plotting and visualizing multivariate data sets can be challenge and a variety of representations are possible. We cover some of the basic ones here.

Get the file `yeast-subset-clean.txt` from the class website. This data set consists of gene expression measurements on 15 genes from 173 two-color microarray experiments (see Gasch et al. 2000). These genes are members of a gene regulatory network that determines how yeast cells respond to nitrogen starvation. The values in the data set are expression ratios (treatment:control) that have been transformed by applying the log_2 function (so that a ratio of 1:1 has the value 0, a ratio of 2:1 has the value 1, and a ratio of 1:2 has the value 0.5).

3.3.1 Scatter plot matrix

We already been introduced to the `pairs()` function which creates a set of scatter plots, arranged like a matrix, showing the bivariate relationships for every pair of variables. The size of this plot is p^2 where p is the number of variables so you should only use it for relatively small subsets of variables (maybe up to 7 or 8 variables at a time).

```
> yeast.clean <- read.delim("yeast-subnetwork-clean.txt")
> names(yeast.clean)
 [1] "FLO8" "RAS2" "TEC1" "PHD1" "ACE2" "SWI5" "SOK2" "RME1" "IME1" "GPA2"
     "MEP2" "IME2" "CLN2"
[14] "ASH1" "MUC1"
> pairs(yeast.clean[1:4]) # create a scatter plot matrix of the first 4
    variables
```

The pairs function can be extended in various ways. The package PerformanceAnalytics, which is mostly geared for econometrics analyses, has a very nice extended pairs function. As discussed in a previous class session you can install packages from the `Packages & Data` menu in the GUI or from the command line as shown below:

```
> install.packages('PerformanceAnalytics', dependencies=T)
> library(PerformanceAnalytics)
> chart.Correlation(yeast.clean[5:8])
```

The output of the `chart.Correlation()` function for this subset of the yeast data is shown in Fig. 3.1. The diagonal of this scatterplot matrix shows the univariate distributions. The lower triangle shows the bivariate relationships, over which has been superimposed curves representing the 'LOESS' regressions for each variable (we'll discuss LOESS in a later lecture). The upper triangle gives the absolute value of the correlations, with starts indicating significance of the p-value associated with each correlation. So for example, you can see from the figure that the genes SOK2 and RME1 are negatively correlated, and this correlation is significantly different from zero (under the assumption of bivariate normality). Note that there is no correction for multiple comparisons.

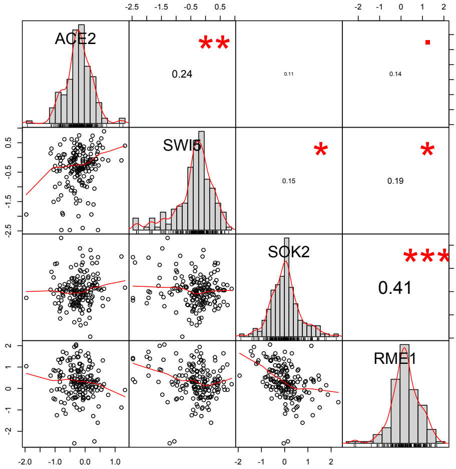

Figure 3.1: Output of the `chart.Correlation()` function in the PerformanceAnalytics package, applied to the yeast expression data set.

3.3.2 3D Scatter Plots

A three-dimensional scatter plot can come in handy. The R library `lattice` has a function called `cloud()` that allows you to make such plots.

```
> library(lattice)
> cloud(ACE2 ~ ASH1 * RAS2, data=yeast.clean)
> cloud(ACE2 ~ ASH1 * RAS2, data=yeast.clean, screen=list(x=-90, y=70)) # same plot from different angle
```

See the help file for `cloud()` and `panel.cloud()` for information on setting parameters.

3.3.3 Scatterplot3D

There is also a package available on CRAN called `scatterplot3d` with similar functionality.

```
> attach(yeast.clean) # so we can access the variables directly
> install.packages('scatterplot3d',dependencies=T) # installs scatterplot3d
> library(scatterplot3d) # assumes package is properly installed
> scatterplot3d(ASH1, RAS2, ACE2)
> scatterplot3d(ASH1, RAS2, ACE2, highlight.3d=T, pch=20,angle=25)
```

The `highlight.3d` argument colors points to help the viewer determine near and far points. Points that are closer to the viewer are lighter colors (more red in the default color scheme).

Using Package Vignettes

The Scatterplot3D package is quite flexible but this flexibility is hard to grok from the standard R help files (try `?scatterplot3d` to see for yourself). Luckily the Scatterplot3D package includes a 'vignette' – a PDF document that discusses the design of the package and illustrates it's use. Many packages include such vignettes. To see the list of vignettes available for your installed packages do the following:

```
> vignette(all=T)
```

You should see that the vignette for the Scatterplot3D package is called `s3d`. You can access this vignette as follows, which should open the document in your default PDF viewer.

```
> vignette("s3d")
```

In this case, the 'good stuff' (i.e. the examples) starts on page 9 of the vignette.

3.3.4 The rgl Package

The 3D plots in `lattice` and `scatterplot3d` are fairly nice, but they don't allow the user to interact with the figures. For example, wouldn't it be nice to be able to rotate a 3D scatter of points around to understand the relationships? The `rgl` package allows you to do this, and can produce figures like that shown in Fig. 3.2. Most R figures can be saved using the `Save` option under the file menu. That's not the case for `rgl` plots. Instead we need to use the `rgl.postscript()` (creates a postscript or PDF version of the figure) or `snapshot3d()` (creates a screenshot) functions.

```
> install.packages('rgl',dependencies=T)
> library(rgl)
> plot3d(ASH1, RAS2, ACE2, col='red', size=1, type='s')
> rgl.postscript('rgl3d-example.pdf', fmt='pdf')
```

3.3.5 Colored grid plots

A colored grid (or 'heatmap') is another way of representing 3D data. It most often is used to represent a variable of interest as a function of two parameters. Grid plots can created using the `image()` function in R.

```
> x <- seq(0, 2*pi, pi/20)
> y <- seq(0, 2*pi, pi/20)
```

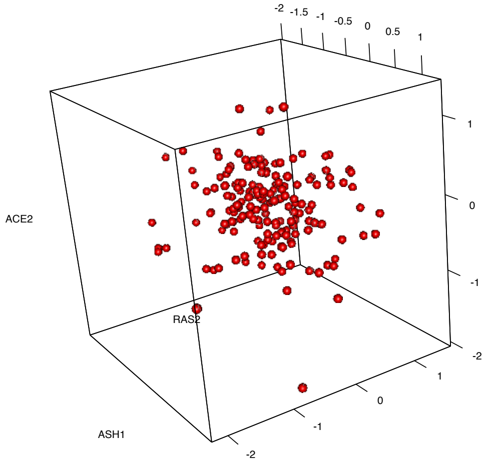

Figure 3.2: Output of the `plot3d()` function in the rgl package.

```
> coolfxn <- function(x,y){
+    cos(x) * cos(y)}
> z <- outer(x,y,coolfxn) # the outer product of two matrices or vectors,
    see docs
> dim(z)
[1] 41 41
> image(x,y,z)
```

The `x` and `y` arguments to `image()` are vectors, the `z` argument is a matrix (in this case created using the outer product operator in conjunction with our function of interest).

A somewhat more flexible function called `levelplot()` is found in the `lattice` package. For example, we can create a similar heatmap using `levelplot()` as follows:

```
> library(lattice)
> levelplot(z)  # just the colors
> levelplot(z, contour=T) # colors plus contour lines
```

We can also apply the levelplot function to creat a representation of a correlation matrix, as shown here:

```
> levelplot(cor(yeast.clean))
```

The default `levelplot()` colors are decent, but let's see how we can change the colors used to our liking. The `colorRampPalette()` function returns a function that interpolates between the values given as arguments to `colorRampPalette()`. So in the example below, it will create a series of colors from blue to white to red.

```
> lvls <- seq(-1,1,0.1)  # set thresholds for our colors
> colors <- colorRampPalette(c('blue', 'white', 'red'))(length(lvls))
> levelplot(cor(yeast.clean), col.regions=colors, at=lvls)
```

The colorRampPallete() function can also take hexadecimal colors, as is commonly used in HTML. For a list of R colors see http://research.stowers-institute.org/efg/R/Color/Chart/. For a list of color schemes, developed by a geographer for effective cartographic representations, see the ColorBrewer web page. For example, here's how to create the representation of the yeast data set correlation matrix shown in Fig. 3.3:

```
# this generates a color ramp from green to black to purple
> colors <- colorRampPalette(c('#1B7837', 'black', '#762A83'))(length(lvls)
    )
> levelplot(cor(yeast.clean), col.regions=colors, at=lvls, scales=list(cex
    =0.6), xlab="", ylab="",main="Correlation Matrix\nYeast Expression Data
    ")
```

The scales argument to levelplot changes the scaling of the tick marks and labels on the axes.

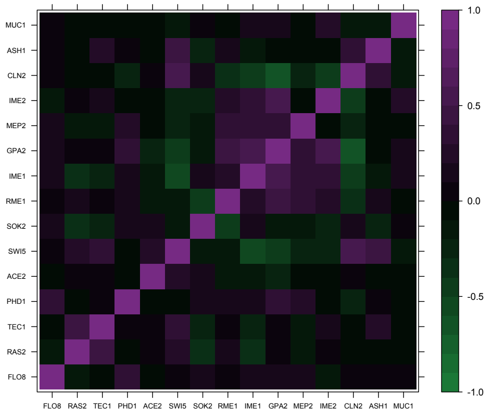

Figure 3.3: A heatmap, representing the correlation matrix for the yeast expression data set, generated by the levelplot() function in the lattice package.

3.4 The Reshape package

reshape2 is an R package for restructuring, transforming, and summarizing multivariate data sets. reshape2 was written by Hadley Wickham, a statistician at Rice University, who is also the author of ggplot. In this section we'll give a brief overview of the reshape2 package; for a more detailed discussion see the documentation available on reshape web page – reshape. Install the reshape2 package before proceeding.

The reshape2 package allows us to restructure and aggregate data more easily than the built-in R functions. There are two primary functions associated with the package

– melt() and cast(). We use melt() to restructure a data frame or list into a generic structure that can then be cast() into the form we want.

melt

Import the reshape2 package with library(reshape2) and read the docs for the melt() function. melt() needs at least three arguments: 1) a data frame or list, 2) a vector specifying which columns to treat as 'identification variables' (id.vars), and 3) a vector specifying which columns to use as 'measured variables' (measured.vars). ID variables are typically the fixed variables that represent aspects of the experimental design, while measured variables represent the variables that were measured on each unit of interest. If id.vars and measured.vars aren't specified, the melt() function will try and infer the id.vars based on those columns that are factors, and treat the remaining variables as measured.vars. If only id.vars is specified, the remaining variables will be treated as measured.vars.

Let's create a simple data set that we can use to explore melt() and cast() functions.

```
> group1 <- c(rep("A", 9), rep("B",9))
> group2 <- rep(c("1","2","3"),6)
> data1 <- c(rnorm(9,mean=0), rnorm(9,mean=1))
> data2 <- as.vector(t(mapply(rnorm, n = c(3,3,3,3,3,3), mean = c(0,1,3))))
> test.data <- data.frame(species=as.factor(group1),
                           treatment=as.factor(group2),
                           v1=data1, v2=data2)
> test.data
   species treatment          v1          v2
1        A         1  0.75357665 -1.83527194
2        A         2  0.40335481  1.22663973
3        A         3  1.18084161  4.47310654
4        A         1 -0.18393749 -1.61953719
5        A         2 -0.85328571  2.75230914
6        A         3  0.99141392  3.39575430
7        A         1 -1.12026845 -0.61442409
8        A         2 -0.01716075  2.08970130
9        A         3 -1.84389967  2.08822132
10       B         1 -0.30507545 -0.01171179
11       B         2  0.54634457 -0.31179921
12       B         3  2.38310469  4.02453740
13       B         1  0.95729799  0.14273026
14       B         2  3.14992630  1.01718329
15       B         3  1.28400301  2.16849192
16       B         1  0.94616082 -0.26436515
17       B         2  1.19047574  0.58964302
18       B         3  0.35085358  2.46990925
```

Let's apply melt(), specifying the 'species' and 'treatment' columns as the id.vars.

```
> melt.test <- melt(test.data, id.vars = c("species","treatment"))
> melt.test
   species treatment variable       value
1        A         1       v1  0.75357665
2        A         2       v1  0.40335481
3        A         3       v1  1.18084161
....
13       B         1       v1  0.95729799
14       B         2       v1  3.14992630
15       B         3       v1  1.28400301
....
19       A         1       v2  1.91999253
20       A         2       v2  1.76509214
21       A         3       v2  3.33803728
....
28       B         1       v2  3.63924057
29       B         2       v2  1.81988816
30       B         3       v2  2.00163369
```

Examining the melted data set, you'll see that the columns representing the measured variables have been collapsed into a single new column called 'value'. There is also another column called 'variable' which specifies which of the measured variables the items in 'value' came from.

cast

Having melted our data set, we can then use the `cast()` function to reshape and aggregate the data into the form we desire. Read the docs for `cast()`. Note that the `cast()` function is actually called as `dcast()` or `acast()` depending on whether you want the function to return a data frame or a vector/array. Minimally, `cast()` takes: 1) a melted data set, 2) a formula specifying how to shape the melted data; and 3) a function to apply to any aggregates that are specified for the cast formula. These are most easily illustrated by example, as shown below.

In the first example we're going to aggregate the measurements of each variable for each species and calculate the species mean. Notice the form of the formula – `species ~ variable`.

```
> recast.test <- dcast(melt.test, species ~ variable, mean)
> recast.test
  species          v1       v2
1       A -0.07659612 1.328500
2       B  1.16701014 1.091624
```

In the second example, we want to aggregate across species *and* experimental treatments. The resulting values in the table show the per-species-per-treatment means.

```
> recast.test <- dcast(melt.test, species + treatment ~ variable, mean)
> recast.test
  species treatment         v1          v2
1       A         1 -0.1835431 -1.35641107
2       A         2 -0.1556972  2.02288339
```

```
3       A         3  0.1094520  3.31902738
4       B         1  0.5327945 -0.04444889
5       B         2  1.6289155  0.43167570
6       B         3  1.3393204  2.88764619
```

Yeast NanoString Dataset

To illustrate the use of the `reshape2` package in conjunction with `ggplot2` we will use a gene expression data set my lab has generated. This data set includes time series expression measurements on 192 genes, collected on each of four different yeast strains grown under two different media conditions. Each combination of treatments (time point, media condition, strain) was replicated three times. The expression platform used for this study is a technology called NanoString.

Download the data set `yeast-timeseries.csv` from the course wiki. The data file is a plain text file that uses the ”comma separated values” format. Use the `read.csv` function to read this data into R.

```
> yeast.time <- read.csv('yeast-timeseries.csv')
> dim(yeast.time)
[1] 108 196
> names(yeast.time)
  [1] "sample.id"  "media"  "strain"  "time.pt"   "replicate"
  [7]  "ACE2"   "ACT1"   "ADR1"   "AGA2"   "AMN1"   "ASG7"   "ASH1"
...
```

We want to treat `sample.id`, `media`, `strain`, and `replicate` as factors. Of these, `strain` is the only variable that is not automatically treated as a factor, because the strain names are numbers. Let’s change that as follows:

```
> yeast.time$strain <- as.factor(yeast.time$strain)
```

Now let’s melt the data set:

```
> yeast.melt <- melt(yeast.time,
        id.vars = c("sample.id","strain","media","replicate","time.pt"))
> dim(yeast.melt)
[1] 20628     7
```

For our example we’ll aggregate across species, media conditions, and time points and calculate the respective means. Below is the appropriate `dcast()` call and the first few rows and columns of the reshaped matrix.

```
> yeast.cast <- dcast(yeast.melt, strain + media + time.pt ~ variable, mean
    )
> dim(yeast.cast)
[1]  36 194
> yeast.cast[1:10,1:5]
   strain media time.pt      ACE2      ACT1
1     144 YEPLD      24 479.99997 95666.580
2     144 YEPLD      48 198.66663 50803.660
3     144 YEPLD      72 119.83327 25328.243
```

```
4      144 YEPLD       96  53.19444 18782.137
5      144   YPD        0 470.88887 92196.660
6      144   YPD       24 481.80550 95400.580
7      144   YPD       48 122.55554 40365.830
8      144   YPD       72  53.55555 18454.160
9      144   YPD       96  53.33333  8317.555
10     497 YEPLD       24 510.72220 95666.580
```

Now let's generate a time series plot for the first gene in the data set, ACE2:

```
> ggplot(subset(yeast.cast, media == 'YPD'),
        aes(x=time.pt, y=ACE2, col=strain)) + geom_line() + geom_point()
```

Notice the use of the `subset()` function to focus specifically on the YPD media treatment. However, it would be much more useful to be able to compare the two media treatments, YPD and YEPLD, side by side. To do so we can use what ggplot calls a 'facet'. A facet specifies one or more variables to condition against. So if we treat the variable `media` as a facet, we will generate a set of plots that differ only by media type. Here's how to do this with the `facet_wrap()` function from ggplot:

```
> ggplot(yeast.cast, aes(x=time.pt, y=ACE2, col=strain)) + geom_line() +
    geom_point() + facet_wrap(facets=c('media'))
```

Your plot should resemble Fig. 3.4. You can examine the times series for different genes by changing the `y` variable in the ggplot aesthetic (do `name(yeast.cast)` to see all the variable names).

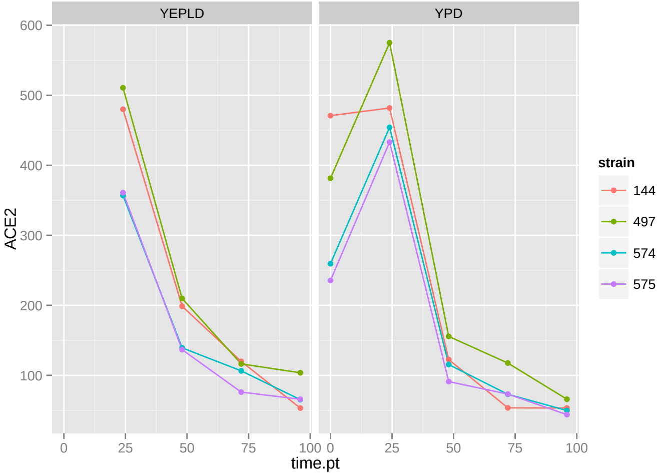

Figure 3.4: Gene expression time series for four yeast strains, grown in two different media conditions.

Assignment 3.2

Rather than plotting the time series for the two media conditions side-by-side, as in Fig. 3.4, you can place them on the same plot and use different line type (e.g. solid vs. dashed) to distinguish between them (see Fig. 3.5).

Write a function the encapsulates the steps needed to produce a figure shown like that shown in Fig. 3.5. This function should take as input: 1) a dataset with column names as in `yeast.cast` used above; and 2) a string giving the name of the gene you want to plot, e.g. `"ACE2"` or `"MUC1"`. The function `aes_string()` will be useful for writing this function. For an example of the use of `aes_string()` see the `scatterWithMargins()` function from week 2.

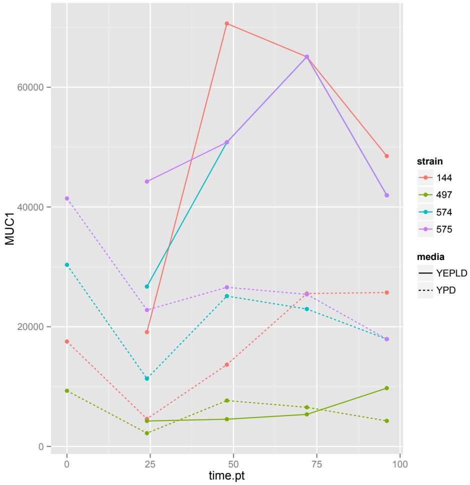

Figure 3.5: An alternate representation of the expression time series data set.

4 Multiple Regression in R

4.1 Multiple Regression in R

To illustrate multiple regression in R we'll use a built in dataset called `trees`. `trees` consists of measurements of the girth, height, and volume of 31 black cherry trees (`?trees` for more info). We'll start with some summary tables and diagnostic plots to familiarize ourselves with the data:

```
> names(trees)
[1] "Girth"  "Height" "Volume"
> dim(trees)
[1] 31  3
> summary(trees)
     Girth           Height       Volume
 Min.   : 8.30   Min.   :63   Min.   :10.20
 1st Qu.:11.05   1st Qu.:72   1st Qu.:19.40
 Median :12.90   Median :76   Median :24.20
 Mean   :13.25   Mean   :76   Mean   :30.17
 3rd Qu.:15.25   3rd Qu.:80   3rd Qu.:37.30
 Max.   :20.60   Max.   :87   Max.   :77.00

# we'll use the chart.Correlation fxn that we introduced last week
> library(PerformanceAnalytics)
> chart.Correlation(trees)
```

As one might expect, the scatterplot matrix shows that all the variables are positively correlated, and girth and volume have a particularly strong correlation.

Let's assume we're lumberjacks, but our permit only allows us to harvest a fixed number of trees. We get paid by the total volume of wood we harvest, so we're interested in predicting a tree's volume (hard to measure directly) as a function of its girth and height (relatively easy to measure), so we can pick the best trees to harvest. We'll therefore calculate a multiple regression of volume on height and width. Let's start by taking a look at the 3D scatter of the data using the plot3d function from the `rgl` package.

```
> library(rgl)
> plot3d(trees, col='red', size=1, type='s') # use your mouse to rotate the
     plot
```

From the 3D scatter plot it looks like we ought to be able to find a plane through the data that fits the scatter fairly well. Let's use the `lm()` function to calculate the multiple regression:

```
> l <- lm(Volume ~ Girth + Height, data=trees)
```

To visualize the multiple regression, let's use the `scatterplot3d` package to draw the 3D scatter of plots and the plane that corresponds to the regression model:

```
> library(scatterplot3d)  # install this package first if needed
> p <- scatterplot3d(trees,angle=55,type='h')
> title('Tree Volume as\na function of Girth and Height')
> p$plane3d(l, col='orangered')
> dev.copy(pdf, 'trees-regrfit.pdf')  # copy plot to a pdf file
> dev.off()  # write the file
```

Notice the use of `dev.copy()` and `dev.off()` to save the plot from the console. The output this generates should look similar to Fig. 4.1.

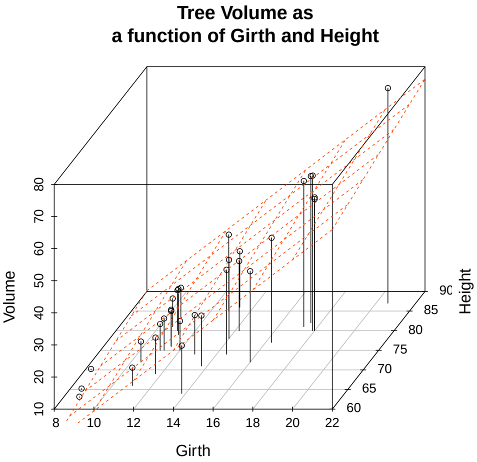

Figure 4.1: Multiple regression plot of cherry tree volume on girth and height, generated using the `scatterplot3d` library

From the figure it looks like the regression model fits pretty well, as we anticipated from the pairwise relationships. Let's use the `summary()` function to obtain details of the model:

```
> summary(l)

Call:
lm(formula = Volume ~ Girth + Height, data = trees)

Residuals:
    Min      1Q  Median      3Q     Max
-6.4065 -2.6493 -0.2876  2.2003  8.4847

Coefficients:
```

```
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -57.9877     8.6382  -6.713 2.75e-07 ***
Girth         4.7082     0.2643  17.816  < 2e-16 ***
Height        0.3393     0.1302   2.607   0.0145 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘’. 0.1 ‘’  1

Residual standard error: 3.882 on 28 degrees of freedom
Multiple R-squared: 0.948,  Adjusted R-squared: 0.9442
F-statistic:   255 on 2 and 28 DF,  p-value: < 2.2e-16
```

The regression equation is: $\hat{y} = 4.71x_1 + 0.34x_2$, where y is Volume, and x_1 and x_2 are Girth and Height respectively. Since they're on different scales the coefficients for Girth and Height aren't directly comparable. Both coefficients are significant at the $p < 0.05$ level, but note that Girth is the much stronger predictor. In fact the addition of height explains only a minor additional fraction of variation in tree volume, so from the lumberjack's perspective the additional trouble of measuring height probably isn't worth it.

4.1.1 Exploring the Vector Geometry of a Regression Model

The object returned by the `lm()` function hold lots of useful information:

```
> names(l)
 [1] "coefficients"  "residuals"     "effects"
     "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "xlevels"
     "call"          "terms"         "model"
```

The `fitted.values` correspond to the predicted values of the outcome variable ($\hat{y}$). Let's use our knowledge of vector geometry to further explore the relationship between the predicted Volume and the predictor variables. By definition the vector representing the predicted values lies in the plane defined by Height and Girth, so let's do some simple calculations to understand their length and angular relationships:

```
# proportional to length of vectors
> sd(l$fitted.values)
[1] 16.00434
> sd(trees$Height)
[1] 6.371813
> sd(trees$Girth)
[1] 3.138139

# cosines of angles btw vectors
> cor(trees$Height, trees$Girth)
[1] 0.5192801
> cor(trees$Height, l$fitted.values)
[1] 0.6144545
> cor(trees$Girth, l$fitted.values)
[1] 0.9933158
```

```
# angles btw vectors in degrees
> acos(cor(trees$Height, l$fitted.values)) * (180/pi)
[1] 52.08771
> acos(cor(trees$Girth, l$fitted.values)) * (180/pi)
[1] 6.628322
> acos(cor(trees$Girth, trees$Height)) * (180/pi)
[1] 58.71603
```

In class assignment

Using the calculations above you should now be able to sketch out by hand, a diagram depicting the vector relationships between Height, Girth, and the predicted Volume . Once you've finished with your sketch, discuss it with your fellow classmates. Did you get similar answers? If not, discuss it and try to come up with an agreed upon representation.

4.1.2 Exploring the Residuals from the Model Fit

Now let's look at the residuals from the regression. The residuals represent the 'unexplained' variance:

```
> plot(trees$Volume,l$residuals, xlab='Volume',ylab='Regression Residuals')
> abline(h=0, lty='dashed', col='red')
```

Ideally the residuals should be evenly scattered around zero, with no trends as we go from high to low values of the dependent variable. As you can see in Fig. 4.2 it looks like that the residuals on the left tend to be below zero, while those on the far right of the plot are consistently above zero, suggesting that there may be a non-linear aspect of the relationship that our model isn't capturing.

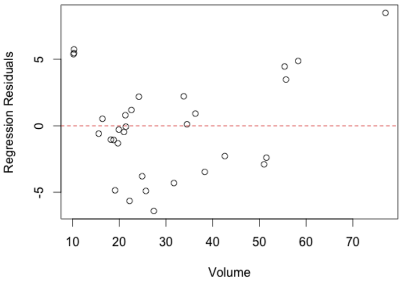

Figure 4.2: Residual plot based on the multiple regression plot of cherry tree volume on girth and height,

Let's think about the relationships we're actually modeling for a few minutes. For the sake of simplicity let's consider the trunk of a tree to be a cylinder. How do the dimensions of this cylinder relate to its volume? You can look up the formula for the volume of a cylinder, but the key thing you'll want to note is that volume of the cylinder should be proportional to a characteristic length of the cylinder cubed ($V \propto \mathrm{L}^3$). This suggests that if we want to fit a linear model we should relate Girth to $\sqrt[3]{\mathrm{Volume}}$. Let's explore this a little. Since our initial multiple regression suggested that height had relatively little predictive power, we'll simplify our model down to a single predictor:

```
> cuberoot.V <- trees$Volume^0.33
> cor(trees$Volume, trees$Girth)
[1] 0.9671194
> cor(cuberoot.V, trees$Girth)
[1] 0.9777078
> l.orig <- lm(trees$Volume~ trees$Girth)
> l.transf <- lm(cuberoot.V ~ trees$Girth)
> summary(l.orig)

Call:
lm(formula = trees$Volume ~ trees$Girth)

Residuals:
   Min     1Q Median     3Q    Max 
-8.065 -3.107  0.152  3.495  9.587 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) -36.9435     3.3651  -10.98 7.62e-12 ***
trees$Girth   5.0659     0.2474   20.48  < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘’. 0.1 ‘’  1 

Residual standard error: 4.252 on 29 degrees of freedom
Multiple R-squared: 0.9353, Adjusted R-squared: 0.9331 
F-statistic: 419.4 on 1 and 29 DF,  p-value: < 2.2e-16 

> summary(l.transf)

Call:
lm(formula = cuberoot.V ~ trees$Girth)

Residuals:
     Min       1Q   Median       3Q      Max 
-0.18919 -0.09775 -0.01488  0.07855  0.26427 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)  0.82543    0.08856   9.321 3.18e-10 ***
trees$Girth  0.16324    0.00651  25.076  < 2e-16 ***
---
```

```
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘’. 0.1 ‘’  1

Residual standard error: 0.1119 on 29 degrees of freedom
Multiple R-squared: 0.9559, Adjusted R-squared: 0.9544
F-statistic: 628.8 on 1 and 29 DF,  p-value: < 2.2e-16
```

Comparing the summary tables, we see indeed that using the cube root of Volume improves the fit of our model some. Let’s examine the residuals.

```
> layout(c(1,2), widths=c(3,3), heights=c(2,2))
> plot(trees$Volume, l.orig$residuals, xlab='Volume', ylab="Residuals")
> abline(h = 0, col='red', lty='dashed')
> plot(cuberoot.V, l.transf$residuals,
       xlab='Volume^0.33', ylab='Residuals')

> abline(h = 0, col='red', lty='dashed')
> dev.copy(pdf, 'compare-residuals.pdf')
> dev.off()
> layout(c(1,1))   # reset the layout
```

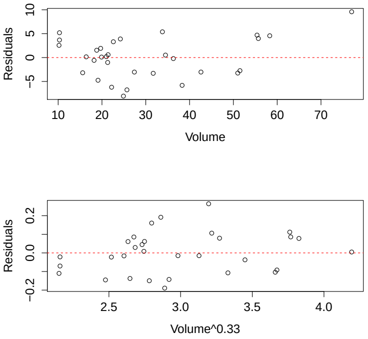

Figure 4.3: Residual plot based on the bivariate regression of tree volume on girth, or $\sqrt[3]{V}$ on girth

As we can see the transformation we applied to the data did seem to make our residuals more uniform across the range of observations. Note the use of the `layout()` function to put multiple plots in the same figure.

4.1.3 Fitting a curvilinear model using lm()

Above we transformed the volume data in order to fit a straight line relationship between $\sqrt[3]{V}$ and Girth. However, we could just as easily have applied a cubic regression to the original variables as shown below (remember this is still linear in the coefficients):

```
> lm.3 <- lm(Volume ~ I(Girth^3), data=trees)
> summary(lm.3)

Call:
lm(formula = Volume ~ I(Girth^3), data = trees)

Residuals:
   Min     1Q Median     3Q    Max 
-4.526 -3.036  0.215  2.419  8.291 

Coefficients:
             Estimate Std. Error t value Pr(>|t|)    
(Intercept) 8.0426960  1.0426698   7.714 1.66e-08 ***
I(Girth^3)  0.0081365  0.0003118  26.098  < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘’. 0.1 ‘’  1

Residual standard error: 3.379 on 29 degrees of freedom
Multiple R-squared: 0.9592, Adjusted R-squared: 0.9578 
F-statistic: 681.1 on 1 and 29 DF,  p-value: < 2.2e-16

> lm.3$coefficients
(Intercept)  I(Girth^3) 
8.042696007 0.008136533 
> a0 = lm.3$coefficients[[1]]
> B1 = lm.3$coefficients[[2]]
> x <- seq(8,25,0.25) # range of values to evaluate model over
> fit <- a0 + B1*x^3
> plot(Volume ~ Girth, data=trees)
> lines(x,fit,col='red')
> figtext <- paste(c("Volume = ", round(a0,2), "+", round(B1,4), "*Girth^3"
    ), collapse='')
> text(12, 60, figtext)
```

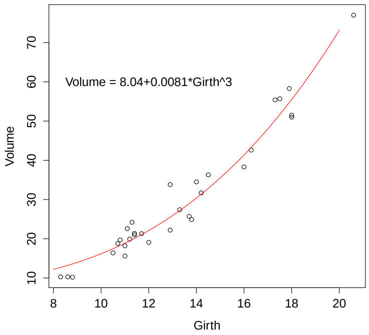

Figure 4.4: Cubic regression of tree volume on girth

The `I()` function used above requires a little explanation. Normally, the R formula syntax (see `?formula`) treats the carat symbol, `'^'`, as short-hand for factor crossing to the specified degree. For example, the formula `(a+b+c)^2` would be interpretted as the model with main effects and all second order interaction terms, i.e. `a + b + c + a:b + a:c + b:c` where the colons indicate interactions. The `I()` function 'protects' the object in it's argument; in this case telling the regression function to treat this as Girth raised to the third power as opposed to trying to construct interaction terms for Girth.

Assignment 4.1

Write a function, `mult.regr(X,y)` that calculates the multiple regression of y on multiple predictors, $x_1, x_2, \ldots x_k$ *using matrix operations*. Your function should take two arguments, `X` and `y`, where `X` is a matrix representing the predictor variables and `y` is a vector for the outcome variable. Your function should return a list containg the vector of regression coeffients, B, the coefficient of determination (R^2), and a vector, $\hat{y}$, representing the fitted values. Refer to the slides from lecture 4 (and possibly lecture 2 if you need a refresher) to review the matrix solution to the regression problem.

4.2 Exploring the impact of nearly collinear predictors on regression

In lecture we discussed the problems that can arise in regression when your predictor variables are nearly collinear. In this section we'll illustrate some of these issues.

Consider again the `trees` data set. Recall that two of the variables – Girth and Volume – are highly correlated and thus nearly collinear.

```
> cor(trees)
           Girth    Height    Volume
Girth  1.0000000 0.5192801 0.9671194
Height 0.5192801 1.0000000 0.5982497
Volume 0.9671194 0.5982497 1.0000000
```

Let's explore what happens when we treat Height as the dependent variable, and Girth and Volume as the predictor variables.

```
> lm.H <- lm(Height ~ Girth + Volume, data = trees)
> summary(lm.H)

Call:
lm(formula = Height ~ Girth + Volume, data = trees)

Residuals:
    Min      1Q  Median      3Q     Max 
-9.7855 -3.3649  0.5683  2.3747 11.6910 

Coefficients:
```

```
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  83.2958     9.0866   9.167 6.33e-10 ***
Girth        -1.8615     1.1567  -1.609   0.1188
Volume        0.5756     0.2208   2.607   0.0145 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘’. 0.1 ‘’  1

Residual standard error: 5.056 on 28 degrees of freedom
Multiple R-squared:  0.4123,    Adjusted R-squared:  0.3703
F-statistic:  9.82 on 2 and 28 DF,  p-value: 0.0005868
```

We can, of course, fit the linear model despite the collinearity, and we find that the model does have some predictive power, with $R^2 = 0.41$, and with Volume being the more significant predictor.

Now, let’s created a slightly different version of the trees data set by add some noise to the three variables. Our goal here is to simulate a data set we might have created had we measured a slightly different set of trees during our sampling. We’ll use the `jitter` function to add uniform noise to the data set.

```
> jitter.Girth <- jitter(trees$Girth, amount= 0.25 * sd(trees$Girth))
> jitter.Height <- jitter(trees$Height, amount= 0.25 * sd(trees$Height))
> jitter.Volume <- jitter(trees$Volume, amount= 0.25 * sd(trees$Volume))
> jitter.trees <- data.frame(Girth = jitter.Girth,
                             Height = jitter.Height,
                             Volume = jitter.Volume)
```

Here we added uniform noise proportional to the one-quarter the standard deviation of each variable. Let’s take a moment to convince ourselves that our new data set, `jitter.trees`, is not too different from the `trees` data set from which it was derived.

```
# compare this to summary(trees)
# You will get slightly different answers because jitter adds random noise

> summary(jitter.trees)
     Girth             Height          Volume
 Min.   : 7.913   Min.   :62.31   Min.   :10.75
 1st Qu.:10.971   1st Qu.:72.37   1st Qu.:18.99
 Median :12.606   Median :76.54   Median :22.38
 Mean   :13.170   Mean   :75.84   Mean   :29.77
 3rd Qu.:15.183   3rd Qu.:80.63   3rd Qu.:37.71
 Max.   :20.722   Max.   :85.91   Max.   :77.69

# correlations among jittered variables are
# similar to those of the original variables

> cor(jitter.trees)
           Girth    Height    Volume
Girth  1.0000000 0.4924240 0.9433214
Height 0.4924240 1.0000000 0.5531763
Volume 0.9433214 0.5531763 1.0000000
```

```
## jittered variables are highly correlatd with original variables

> cor(trees$Height, jitter.trees$Height)
[1] 0.9861006
> cor(trees$Girth, jitter.trees$Girth)
[1] 0.9928097
> cor(trees$Volume, jitter.trees$Volume)
[1] 0.9883385

> plot(trees$Height, jitter.trees$Height)
> plot(trees$Girth, jitter.trees$Girth)
> plot(trees$Volume, jitter.trees$Volume)
```

Now that we've convinced ourselves that our jittered data set is a decent approximation to our original data set, let's re-calculate the linear regression, and compare the coefficients of the jittered model to the original model:

```
> lm.H.jitter <- lm(Height ~ Girth + Volume, data = jitter.trees)
> coefficients(lm.H.jitter)
(Intercept)       Girth      Volume
 73.3492169  -0.5437115   0.3241854
> coefficients(lm.H)
(Intercept)       Girth      Volume
 83.2957705  -1.8615109   0.5755946
```

We see that the coefficients of the linear model have changed quite a bit between the original data and the jittered data. Our model is unstable to relatively modest changes to the data!

Let's draw some plots to illustrate how different the models fit to the original and jittered data are:

```
# draw 3d scatter plots with small points so as not to obscure regression
    planes
> p <- scatterplot3d(x=trees$Girth, y=trees$Volume, z=trees$Height,
                      angle=15, type='p', pch='.')

# original model
> p$plane3d(lm.H, col='orangered')

# jittered model
> p$plane3d(lm.H.jitter, col='blue')
```

The figure you generated should look something like Fig. 4.5.

Let's do the same comparison for the multiple regression of Volume on Height and Girth. In this case the predictor variables are *not* nearly collinear.

```
> lm.V <- lm(Volume ~ Girth + Height, data = trees)
> lm.V.jitter <- lm(Volume ~ Girth + Height, data = jitter.trees)
> coefficients(lm.V)
(Intercept)       Girth      Height
```

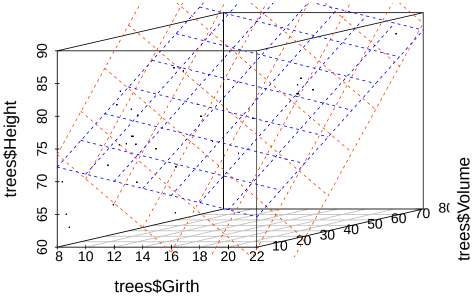

Figure 4.5: Multiple regression plot of cherry tree height on girth and volume, for the original data (red) and the jittered data (blue).

```
-57.9876589   4.7081605   0.3392512
> coefficients(lm.V.jitter)
(Intercept)       Girth      Height
-51.2670818   4.4798268   0.2906203
```

For this model, we see that the coefficients have changed only a small amount. The underlying data, `jitter.trees`, is the same in both cases, but now our model is stable because the predictor variables are only modestly correlated with each other.

Let's generate another plot to illustrate the similarity of the models fit to the original and jittered data when Girth and Height are used to predict Volume. The corresponding output is shown in Fig. 4.6.

```
> p <- scatterplot3d(x=trees$Girth, y=trees$Height, z=trees$Volume,
                     angle=55, type='p', pch='.')
> p$plane3d(lm.V, col='orangered')
> p$plane3d(lm.V.jitter, col='blue')
```

Finally, let's do some vector calculations to quantify how the angular deviation between the fit data and the predictor variables changes between the original and jittered data set for the two different multiple regressions:

```

# write a quickie fxn to express angle between vectors in degrees
> vec.angle <- function(x,y) { acos(cor(x,y)) * (180/pi)}

# vector angles for fit of Height ~ Girth + Volume (orig)
> vec.angle(lm.H$fitted.values, trees$Girth)
[1] 36.02644
> vec.angle(lm.H$fitted.values, trees$Volume)
[1] 21.29297

# vector angles for fit of Height ~ Girth + Volume (jittered)
```

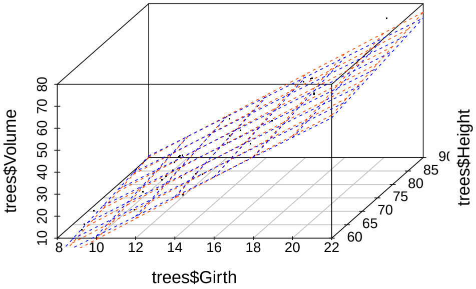

Figure 4.6: Multiple regression plot of cherry tree volume on girth and height, for the original data (red) and the jittered data (blue).

```
> vec.angle(lm.H.jitter$fitted.values, jitter.trees$Girth)
[1] 28.48079
> vec.angle(lm.H.jitter$fitted.values, jitter.trees$Volume)
[1] 9.097828

# CONCLUSION -- angular changes of about 8 and 12 degrees


# vector angles for fit of Volume ~ Girth + Height (orig)
> vec.angle(lm.V$fitted.values, trees$Girth)
[1] 6.628322
> vec.angle(lm.V$fitted.values, trees$Height)
[1] 52.08771

# vector angles for fit of Volume ~ Girth + Height (jittered)
> vec.angle(lm.V.jitter$fitted.values, jitter.trees$Girth)
[1] 6.163463
> vec.angle(lm.V.jitter$fitted.values, jitter.trees$Height)
[1] 54.33651

# CONCLUSION -- angular changes of about 0.5 and 2 degrees
```

4.3 Manipulating data using split

Last week we introduced the `reshape()` function from the reshape2 package. `reshape()` is good for computing simple statistics across multiple ‘facets’ of data. However, more complicated statistics are made possible by using the `split()` function, which is defined in the R base package.

split() takes two arguments: 1) a vector or data frame to split and 2) a character vector defining what to split the first argument by. For example, we can split the iris data set by species in order to get a list containing three data frames; one for each species.

```
> iris.split <- split(iris, iris$Species)
> names(iris.split)
[1] "setosa"     "versicolor" "virginica"
> str(iris.split)  # see the documentation for the str() function
List of 3
 $ setosa    :'data.frame': 50 obs. of  5 variables:
  ..$ Sepal.Length: num [1:50] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
  ..$ Sepal.Width : num [1:50] 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
  ..$ Petal.Length: num [1:50] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
  ..$ Petal.Width : num [1:50] 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
  ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1
      1 1 1 1 1 ...
 $ versicolor:'data.frame': 50 obs. of  5 variables:
  ..$ Sepal.Length: num [1:50] 7 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 ...
  ..$ Sepal.Width : num [1:50] 3.2 3.2 3.1 2.3 2.8 2.8 3.3 2.4 2.9 2.7 ...
  ..$ Petal.Length: num [1:50] 4.7 4.5 4.9 4 4.6 4.5 4.7 3.3 4.6 3.9 ...
  ..$ Petal.Width : num [1:50] 1.4 1.5 1.5 1.3 1.5 1.3 1.6 1 1.3 1.4 ...
  ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 2 2 2 2 2
      2 2 2 2 2 ...
 $ virginica :'data.frame': 50 obs. of  5 variables:
  ..$ Sepal.Length: num [1:50] 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 ...
  ..$ Sepal.Width : num [1:50] 3.3 2.7 3 2.9 3 3 2.5 2.9 2.5 3.6 ...
  ..$ Petal.Length: num [1:50] 6 5.1 5.9 5.6 5.8 6.6 4.5 6.3 5.8 6.1 ...
  ..$ Petal.Width : num [1:50] 2.5 1.9 2.1 1.8 2.2 2.1 1.7 1.8 1.8 2.5 ...
  ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 3 3 3 3 3
      3 3 3 3 3 ...
```

Now that we have a split data frame, it's easy to use lapply or sapply to calculate complicated summary statistics. For example, this function calculates the mean ratio of Sepal.Length to Petal.Length:

```
> ratio.sepal2petal <- function(x) {
+    mean( x$Sepal.Length / x$Petal.Length)
+ }
> sapply(iris.split, ratio.sepal2petal)
    setosa versicolor  virginica
  3.464906   1.400896   1.188350
```

We could also write a function to return the coefficients of fitting a linear model to each facet of the data:

```
> sepal.on.petal.coeff <- function(x){
+       model <- lm(Sepal.Length ~ Petal.Length, data=x)
+       return(model$coeff)
+ }
> sapply(iris.split, sepal.on.petal.coeff)
```

```
                setosa versicolor virginica
(Intercept)  4.2131682   2.407523 1.0596591
Petal.Length 0.5422926   0.828281 0.9957386
```

Of course, no analysis would be complete without examining the fit of the linear models. In order to visualize whether the linear model is a good representation of the data, we'll write another function to return a data frame containing the fitted values, residuals, and species names for each element of the list.

```
> sepal.on.petal.lm.fit <- function(x){
+       model <- lm(Sepal.Length ~ Petal.Length, data=x)
+       data.frame(fitted = fitted(model),
+                  residuals = residuals(model),
+                  species = x$Species)
+ }
> iris.fit <- lapply(iris.split, sepal.on.petal.lm.fit)
> str(iris.fit)
List of 3
 $ setosa    :'data.frame': 50 obs. of  3 variables:
  ..$ fitted   : num [1:50] 4.97 4.97 4.92 5.03 4.97 ...
  ..$ residuals: num [1:50] 0.1276 -0.0724 -0.2181 -0.4266 0.0276 ...
  ..$ species  : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1
       1 1 1 ...
 $ versicolor:'data.frame': 50 obs. of  3 variables:
  ..$ fitted   : num [1:50] 6.3 6.13 6.47 5.72 6.22 ...
  ..$ residuals: num [1:50] 0.7 0.265 0.434 -0.221 0.282 ...
  ..$ species  : Factor w/ 3 levels "setosa","versicolor",..: 2 2 2 2 2 2 2
       2 2 2 ...
 $ virginica :'data.frame': 50 obs. of  3 variables:
  ..$ fitted   : num [1:50] 7.03 6.14 6.93 6.64 6.83 ...
  ..$ residuals: num [1:50] -0.734 -0.338 0.165 -0.336 -0.335 ...
  ..$ species  : Factor w/ 3 levels "setosa","versicolor",..: 3 3 3 3 3 3 3
       3 3 3 ...
>
```

Next, we'll join the data back into a data frame using the `do.call` and `rbind` functions. Read the documentation to figure out what they do.

```
> iris.joined <- do.call('rbind', iris.fit)
> str(iris.joined)
'data.frame': 150 obs. of  3 variables:
 $ fitted   : num  4.97 4.97 4.92 5.03 4.97 ...
 $ residuals: num  0.1276 -0.0724 -0.2181 -0.4266 0.0276 ...
 $ species  : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1
     1 1 ...
```

Finally, we'll visualize our model fits by plotting our data using ggplot:

```
> library(ggplot2)
> ggplot(iris.joined, aes(x=fitted, y=residuals))+
+   geom_point()+
+   facet_wrap(~species, scale='free') +
+   ggtitle("Residuals from Regression of \nSepal Length on Petal Length
    for 3 Iris Species")
```

Examining the residuals (Fig. 4.7), we see they look fairly uniform across the range of fit values. The term that statiscians use for this is ‘homoscedastic’; when the residuals are non uniform we say they are ‘heteroscadistic’.

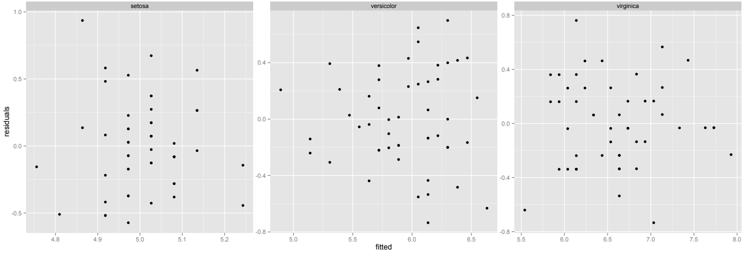

Figure 4.7: Residuals from regressions of Sepal Length on Petal Length, for the Iris data set split by species.

An alternate way to visualize the fits, without the benefit of getting the info on the model fits back for further examination, is to use the `stat_smooth()` to plot a linear fit of our data for each facet (Fig. 4.8). Read the `stat_smooth` documentation to how this works.

```
> ggplot( iris, aes(x=Petal.Length, y=Sepal.Length))+
   geom_point()+
   stat_smooth(method="lm")+
   facet_wrap(~Species, scale='free')+
   ggtitle("Regressions of Sepal Length on Petal Length")
```

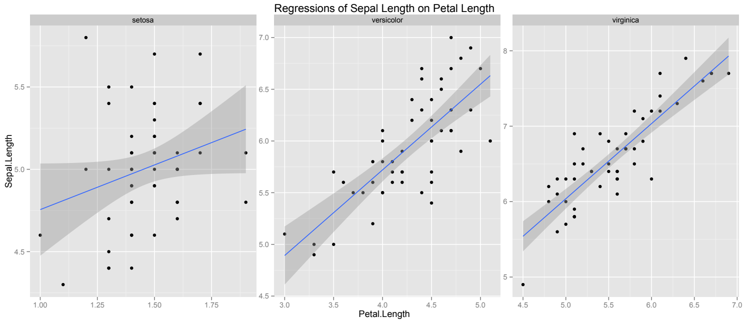

Figure 4.8: Regressions of Sepal Length on Petal Length, for the Iris data, produced using the `stat_smooth()` function in ggplot2.