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 do 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.

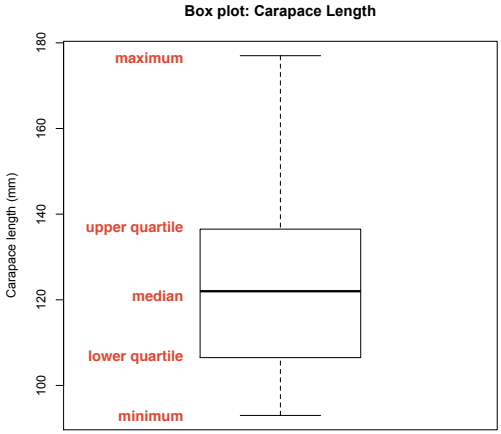

Figure 1.1: A box plot represents a five number summary of a set of observations.

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.

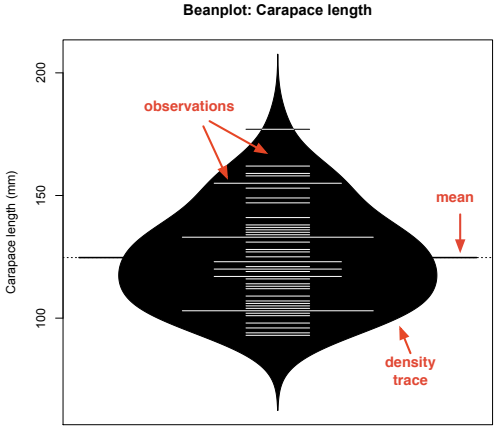

Figure 1.2: Beanplots combine features of density and box plots.

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 an 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 ...
 [59] 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 ...
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.

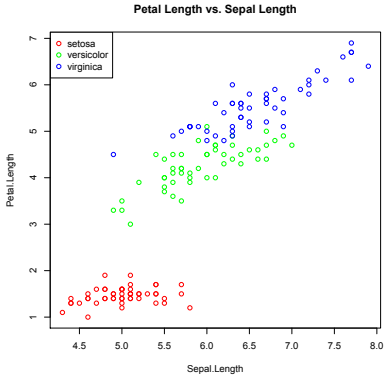

Figure 2.1: Scatter plot created from the iris data set using the `plot` function.

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 thecartesian 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 reprsentations: 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.

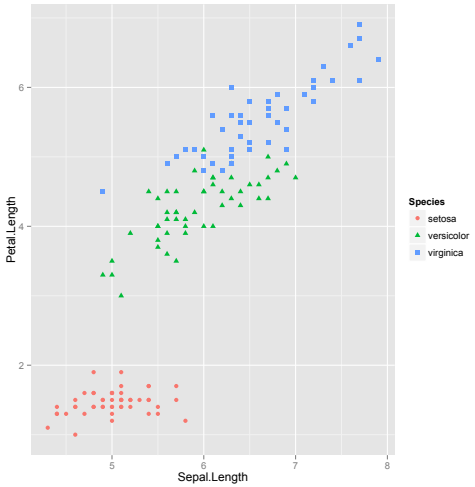

Figure 2.2: Scatter plot created from the iris data set using the `ggplot` function.

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:

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

This produces the plot shown in Figure 2.3.

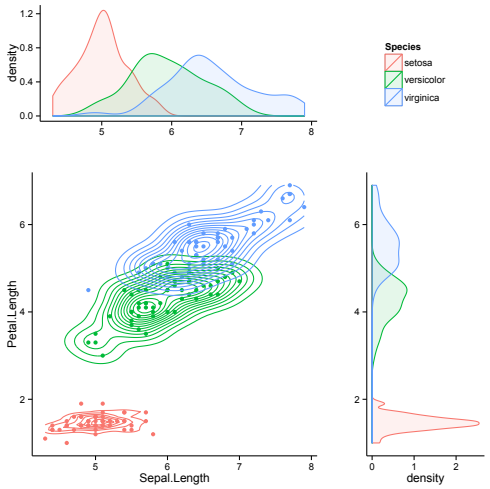

Figure 2.3: Figure produced by the `scatterWithMargins` function from the course wiki.

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

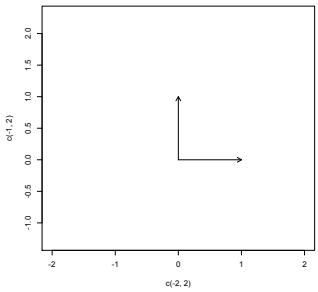

Figure 2.4: A simple vector figure.

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 ($|\vec{a}|^2 = \vec{a} \cdot \vec{a}$)

```
> 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:

```
> myfunc <- function(x,y){
+ # don't type the '+' symbols, these show continuation lines
+   x^2 + y^2
+ }

> a <- 1:5
> b <- 6:10
> a
[1] 1 2 3 4 5
> b
[1]  6  7  8  9 10
> myfunc(a,b)
[1]  37  53  73  97 125
> myfunc
function(x,y){
  x^2 + y^2
}
```

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.

Choose `File > New Script` (or `File > New Document` in OS X) in the R GUI . 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)
}

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) )
}
```

There are two functions defined above, one of which calls the other. Both take single vector arguments. At this point there is no error checking to insure that the argument is reasonable but R's built in error handling will do just fine for now.

Once your functions are in a script file you can make them accesible by using the `source()` function (See also the `File > Source R code...` menu item in the R 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
```

Assignment 2.1

Write a function that uses the dot product and the `acos()` function to calculates 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.

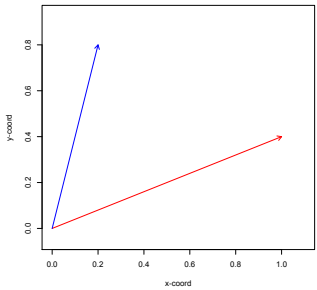

Figure 2.5: Another vector figure.

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)
```

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)
```

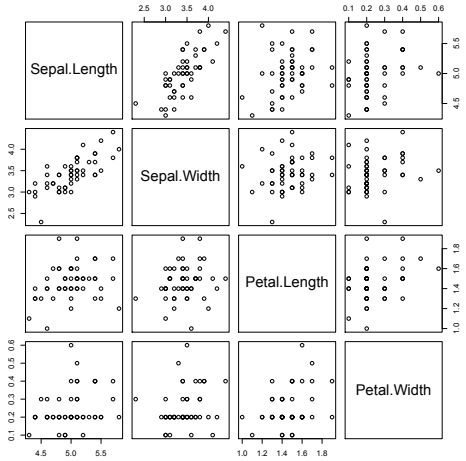

Figure 2.6: Output of the `pairs()` function for the *I. setosa* specimens in the `iris` dataset.

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, $\text{cor}(X,Y) = r_{XY} = cos\,\theta = \frac{\vec{x}\cdot\vec{y}}{|\vec{x}||\vec{y}|}$. 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.

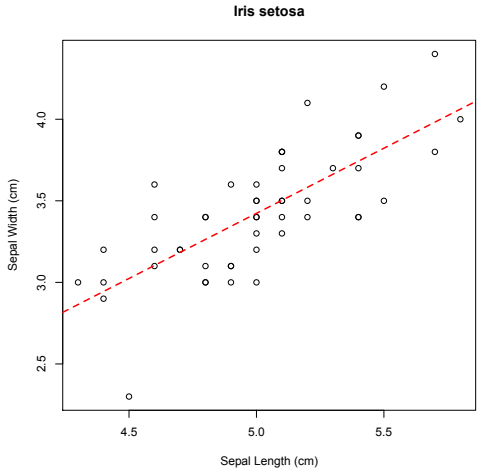

Figure 2.7: Linear regression of Sepal Width on Sepal Length for *I. setosa*.

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 $\vec{\hat{y}} = b\vec{x}$
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.