Bio 723

Scientific Computing for Biologists

Paul M. Magwene

Fall 2012

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 2012 the latest R release is verison 2.15.1.

The R installer will install appropriate icons under the `Start Menu` (Windows) or `Applications` Folder (OS X). On OS X it will install two icons – "R" and "R64", corresponding to 32-bit and 64-bit versions of the executable. The 64 bit version, which allows access to much larger amounts of your comptuer's RAM, is suitable for dealing with very large data sets.

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 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()`.

1.1.4 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 `Change dir...` item under the `File` menu, while the corresponding item is found under the `Misc` menu on OS X.

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

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

1.1.5 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.

1.1.6 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.

1.1.7 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
```

1.1.8 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
```

1.1.9 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
```

1.1.10 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"
```

1.1.11 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.

1.1.12 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()`.

1.1.13 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.

1.1.14 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
```

One of the advantages of the built in `mean()` function is that it knows how to operate on lists as well as vectors:

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

Can you figure out why the above produced a warning message? 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).

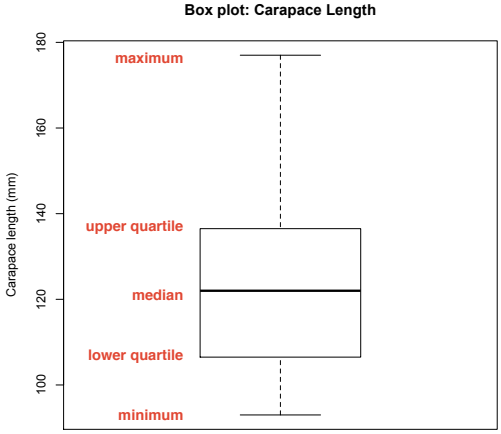


Figure 1.1: A box plot represents a five number summary of a set of observations.

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.

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.

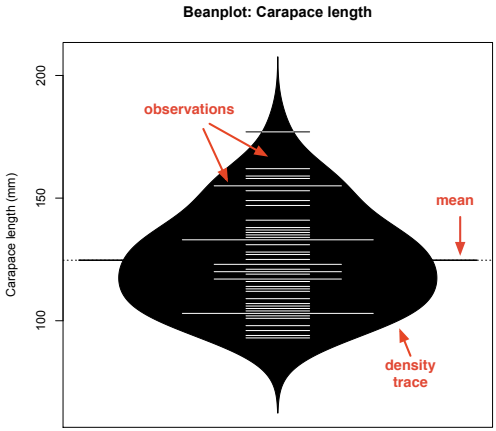


Figure 1.2: Beanplots combine features of density and box plots.

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)
```

2 Vector Operations and Exploring Bivariate Relationships in R

2.1 Vector Operations 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

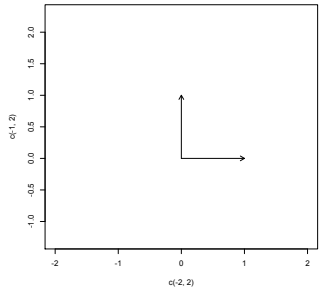


Figure 2.1: A simple vector figure.

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).

2.2 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').

2.2.1 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.

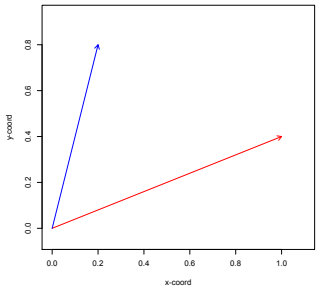


Figure 2.2: Another vector figure.

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
```

2.3 Exploring Bivariate Relationships 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 for specimens from 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
```

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"
```

Notice how we used the `select` argument to `subset()` in order to drop the Species column. Let's explore the setosa subset with some graphs.

```
> plot(setosa$Sepal.Length, setosa$Sepal.Width)
> plot(setosa$Sepal.Width ~ setosa$Sepal.Length)
```

Did you notice what is different between the two versions above? You can also use the `data` argument with plot, like so:

```
> plot(Sepal.Width ~ Sepal.Length, data = setosa)
```

The `xyplot()` function from the `lattice` package does pretty much the same thing:

```
> library(lattice)
> xyplot(Sepal.Width ~ Sepal.Length, data = setosa)
```

Let's also explore a number of the other bivariate relationships in this data set:

```
# an alternate way to generate such a plot, using the data argument to
    specify where the variables are defined
> plot(Petal.Length ~ Sepal.Length, data = setosa)

# same form as the first plot, but changing the character used for the plot
    using the 'pch' argument. The 'cex' argument increases the size of the
    characters by the specified factor (1.5x in this case)
> plot(setosa$Sepal.Length, setosa$Petal.Width, pch = 20, cex=1.5)
```

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

```
> pairs(setosa)
```

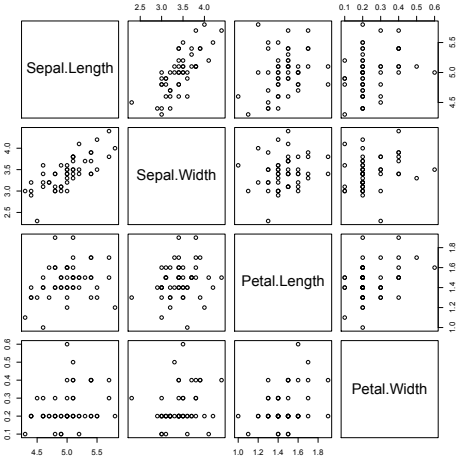


Figure 2.3: Output of the `pairs()` function for the *I. setosa* specimens in the `iris` dataset.

Let's return to our use of the dot product to explore the relationship between variables. First let's add a function to `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 setosa 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 sow 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, $\mathrm{cor}(X,Y) = r_{XY} = cos\,\theta = \frac{\vec{x}\cdot\vec{y}}{|\vec{x}||\vec{y}|}$. 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
```

2.3.1 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"
> 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.

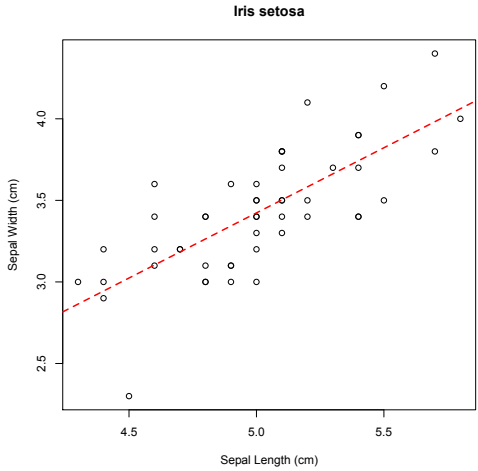


Figure 2.4: Linear regression of Sepal Width on Sepal Length for *I. setosa*.

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 plots in which you use the `plot()` and `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` $\times$ `ncols` the elements will be 'recycled' as discussed in previous lectures). 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)
> tmtx   # note how the elements were all converted to character
   sex length width height
1  "f" " 98"  " 81" "38"
2  "f" "103"  " 84" "38"
3  "f" "103"  " 86" "42"
4  "f" "105"  " 86" "40"
 ... output truncated ...
> tsub <- subset(turtles, select=-sex)
> tmtx <- as.matrix(tsub)
> tmtx    # this is probably more along the lines of what you want
   length width height
1      98    81     38
2     103    84     38
3     103    86     42
4     105    86     40
 ... output truncated ...
```

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
```

Note that the rnorm() function generates random numbers from the standard normal distribution. Use the help to read the documentation for rnorm(). Note that you can use the mean and sd arguments to specify other normal distributions. Since we've introduced the rnorm() function let's go ahead and show how we can useit to simulate draws from a random normal distribution.

```
> x <- rnorm(100) # draw 100 samples from random normal distn
> mean(x)
[1] 0.03198427
> sd(x)
[1] 1.012966
> hist(x)
> abline(v=mean(x),col='red',lwd=2,lty='dashed')
```

Also notice that the rnorm() help file also mentions three other related functions dnorm(), pnorm(), and qnorm(). dnorm() gives the density, pnorm() the distribution function, and qnorm() the quantile function. Here's an example how we can use the dnorm() function to compare our observed sample to the expected distribution:

```
> breakpts <- seq(-3, 3, 0.5)
# note use of freq=F to get density histogram  and user specified
    breakpoints
> h <- hist(x, breakpts, freq=F)
> expected <- dnorm(h$mids)
> lines(h$mids, expected, col='blue',lwd=2)
> abline(v=0, col='blue', lwd=2)
> abline(v=mean(x), col='red', lwd=2,lty='dashed')
```

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
> 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

To calculate a row vector of means, $\mathbf{m}$:

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

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 is given by:

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

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

The data set `yeast-subnetwork-raw.txt` (see class website), consists of gene expression measurements for 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).

The raw data file `yeast-subnetwork-raw.txt` has the genes (variables) arranged by rows and the observations (experiments) in columns. There are also missing values. Using R, show how to read in the data set and then create a matrix where the genes are in columns and the observations in rows. Then replace any missing values (`NA`) in each column with the variable (gene) means (there are better ways to impute missing values but this will do for now). Write a *generic* function, `read.missing()` that will work with any data file with the same organization as that above.

Functions that might come in handy for this assignment include: `read.delim()`, `t()`, `subset()`, `as.matrix()`, and `is.na()`. Note that `t()` applies to data frames as well as matrices. Also take note of the `na.rm` argument of `mean()`. You might consider creating a function that handles the missing value replacement and using it in conjunction with the `apply()` function. `colnames()` and `rownames()` allow you to assign/extract column and row names for a matrix. Use the `write.table()` function to save your results (I recommend you use `"\t"` (i.e. tab) as the `sep` argument). You can check the correctness of your function by comparing it to the `yeast-subnetwork-clean.txt` file available from the course wiki. Use the `all.equal` function to check for approximate equality.

Assignment 3.2

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 the `yeast-subnetwork` 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 (or use the cleaned up data set you created in the assignment above).

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.

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.

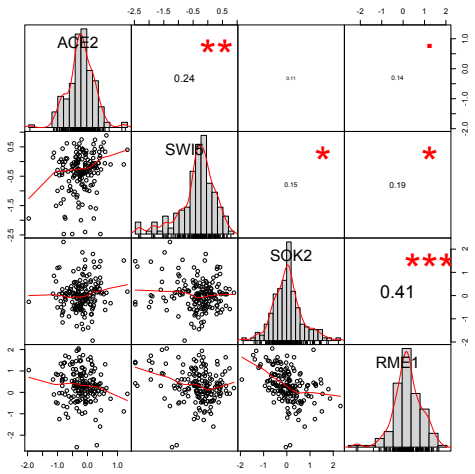


Figure 3.1: Output of the `chart.Correlation()` function in the PerformanceAnalytics package, applied to the yeast expression data set.

```
> 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 th 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')
```

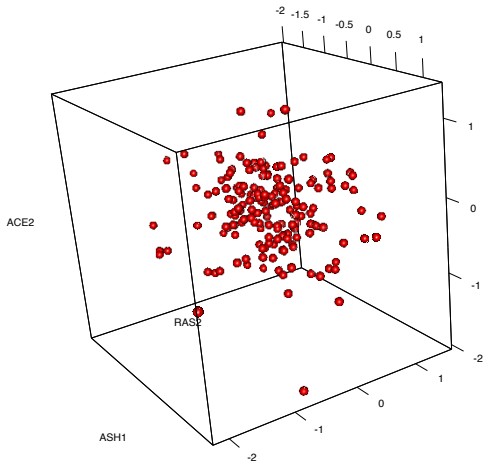


Figure 3.2: Output of the `plot3d()` function in the rgl package.

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)
> 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.

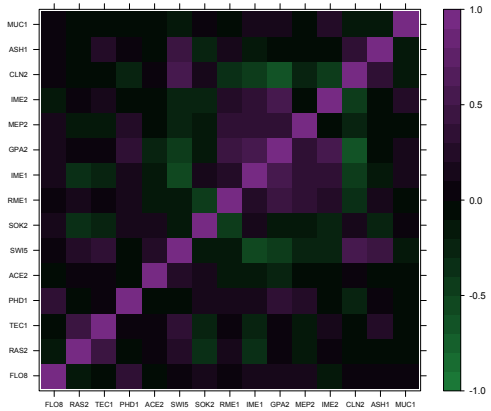


Figure 3.3: A heatmap, representing the correlation matrix for the yeast expression data set, generated by the `levelplot()` function in the lattice package.

4 Multiple Regression in R

4.1 Introduction to Literate Programming Using knitr

`knitr` documents weave together documentation/discussion and code into a single document. The pieces of code and documentation are referred to as 'chunks'. Using `knitr` you can turn the entire document into a nicely formatted report, or you can extract just the code parts.

I recommend you use `knitr` from inside RStudio, which has a Markdown aware editor and is pre-configured to compile `knitr` documents into HTML. The first thing you'll need to do is install the `knitr` and `markdown` packages, either from the command line using`install.packages()` or from the `Tools > Install Packages` menu (make sure you include the dependencies). Once you've installed `knitr` you can create a Markdown document using `File > New > R Markdown`.

RStudio gives you a template file that illustrates the basic Markdown syntax. For more info about Markdown click the 'MD' button, which will bring up a quick reference guide. Replace the template with the following, and save it as `knitr1.Rmd`. Note that the `.Rmd` extension is recognized by RStudio as an R markdown file. I suggest that you get in the habit of using this extension for your markdown files.

````
# Getting started with knitr

This is a very simple knitr Markdown file. It includes only a single
code chunk.

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

The code chunk above generated a random sample of 30 observations
drawn from a normal distribution with mean zero and standard
deviation one.
````

Let's break down the various pieces of the document. The first line is a header. `#` generates a level one header, `##` generates a level two header, etc. This header is than followed by a couple of lines of text, which will appear in the output.

The R code chunk begins and ends with sets of three backticks. The `{r}` immediately after the first set of backticks tells knitr to treat the code as R code (you can also process other languages such as Python). The final set of backticks tells knitr that you're going back to writing documentation chunks.

After saving your document you can compile it using the `Knit HTML` button or from the R console as:

```
> library(knitr)
> knit2html('knit1.Rmd')
```

Either of the above approaches will generate two new file `knit1.md` and `knit1.html`. RStudio will automatically open the HTML file in a built-in viewer, or you can open the HTML file in any browser. Notice how the code from the code chunk is in the output file as well as the output that you would have generated had you typed the code in at the R console.

4.1.1 A fancier knitr document

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

````
# My Second knitr Report
# John Q. Public

This is a still a simple knitr document. However,
now it includes several code chunks and several
markdown formatting commands.

## 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 ($\sigma = 1$).


### Generating figures#

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

```{r fig=TRUE, fig.width=4, fig.height=4}
hist(z)
```
````

In the second document chunk we included some text between dollar signs. knitr recognizes this as mathematical text, using LaTeX based formatting. Also notice how we put an argument, `fig=TRUE` within the second code chunk delimiter. This will tell

knitr to automatically imbed a figure with the histogram graphic we created into our report. We also specified the dimensions of this figure using `fig.width` and `fig.height`. Save the document as `knit2.Rmd` and repeat the above steps to compile it into HTML.

4.1.2 Extracting R Code by Tangling

In addition to generating reports, `knitr` can be used to extract R source code from your literate document. The following example illustrates this. Save the following as `myfuncs.Rmd`

````
# My library of vector functions

### Vector length

Calculate the length of a vector, using the dot product;
$|\vec{x}| = \sqrt{\vec{x} \cdot \vec{x}}$:

```{r}
vec.length <- function(x){
  return (sqrt(x %*% x))
}
```

### Angle between vectors

Calculate the cosine of the angle between
two vectors $\vec{x}$ and $\vec{y}$:

```{r vectorcosine}
vec.cos <- function(x,y){
  lx <- vec.length(x)
  ly <- vec.length(y)
  return ( (x %*% y)/(lx*ly) )
}
```

Calculate the angle in radians between two vectors:

```{r vectorangle}
vec.angle <- function(x,y){
  return ( acos(vec.cos(x,y)) )
}
```
````

To generate a file of pure R code (i.e. something that could be sourced from the R console), do the following:

```
> knit('myfuncs.Rmd', tangle=TRUE)
```

This will generate a corresponding R file named `myfuncs.R` in which the R code chunks have been detangled from the documentation chunks. Open this file in your R environment to see how it corresponds to the markdown document from which it was generated.

For a full overview of knitr's capabilities see the documentation for knitr availabe at http://yihui.name/knitr/.

Assignment 4.1

Convert your library of matrix functions from Assignment 3.2 to an R markdown document. Include documentation and explanatory text as necessary so that somebody looking at your library for the first time can understand how it works. Make sure to use appropriate markup in your document (e.g. headings) so that the HTML output is nicely formatted.

4.2 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
> 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)
> 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.

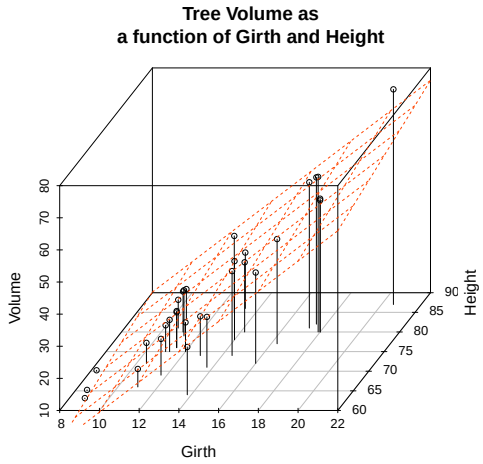


Figure 4.1: Multiple regression plot of cherry tree volume on girth and height, generated using the `scatterplot3d` library

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 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.

4.2.1 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
```

Using those 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.

4.2.2 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 scatter around zero, with no trends as we go from high to low values of the outcome value. 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.

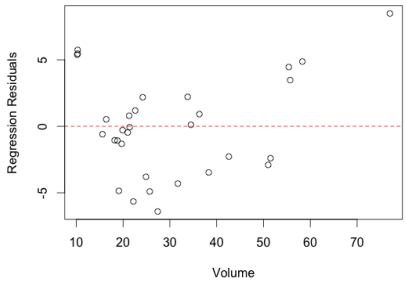


Figure 4.2: Residual plot based on the multiple regression plot of cherry tree volume on girth and height,

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 it's 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 ($V \propto L^3$). This suggests that if we want to fit a linear model we should relate Girth to $\sqrt[3]{\text{Volume}}$. 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()
```

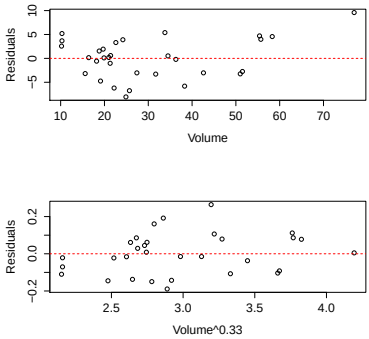


Figure 4.3: Residual plot based on the bivariate regression of tree volume on girth, or $\sqrt[3]{V}$ on girth

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.

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 regression 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(8,20,0.25) # 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)
```

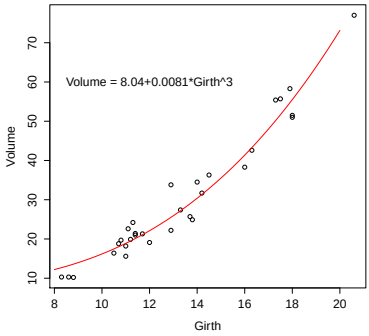


Figure 4.4: Cubic regression of tree volume on girth

Assignment 4.2

In the same R markdown docoument you created for 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.

5 Eigenanalysis and PCA in R

5.1 Eigenanalysis in R

The `eigen()` function computes the eigenvalues and eigenvectors of a square matrix.

```
> A <- matrix(c(2,1,2,3),nrow=2)
> A
     [,1] [,2]
[1,]    2    2
[2,]    1    3
> eigen.A <- eigen(A)
> eigen.A
$values
[1] 4 1
$vectors
           [,1]       [,2]
[1,] -0.7071068 -0.8944272
[2,] -0.7071068  0.4472136
> V <- eigen.A$vectors
> D <- diag(eigen.A$values) # diagonal matrix of eigenvalues
> Vinv <- solve(V)
> V %*% D %*% Vinv  # reconstruct our original matrix (see lecture slides)
     [,1] [,2]
[1,]    2    2
[2,]    1    3
> Vinv %*% A %*% V
              [,1] [,2]
[1,] 4.000000e+00    0
[2,] 2.220446e-16    1
> all.equal(Vinv %*% A %*% V, D) # test 'near equality'
[1] TRUE
> V[,1] %*% V[,2] # note that the eigenvectors are NOT orthogonal. Why?
          [,1]
[1,] 0.3162278
> B <- matrix(c(2,2,2,3),nrow=2) # define another tranformation
> B
     [,1] [,2]
[1,]    2    2
[2,]    2    3
> eigen.B$values
[1] 4.5615528 0.4384472
> eigen.B$vectors
```

```
          [,1]       [,2]
[1,] 0.6154122 -0.7882054
[2,] 0.7882054  0.6154122
> Vb <- eigen.B$vectors
> Vb[,1] %*% Vb[,2] # these eigenvectors ARE orthogonal.
     [,1]
[1,]    0
```

As we discussed in lecture, the eigenvectors of a square matrix, $\mathbf{A}$, point in the directions that are unchanged by the transformation specified by $\mathbf{A}$. The following relationships relate $\mathbf{A}$ to it's eigenvectors and eigenvalues:

$$\mathbf{V}^{-1}\mathbf{A}\mathbf{V} = \mathbf{D}$$

$$\mathbf{A} = \mathbf{V}\mathbf{D}\mathbf{V}^{-1}$$

where $\mathbf{V}$ is a matrix where the columns represent the eigenvectors, and $\mathbf{D}$ is a diagonal matrix of eigenvalues.

Since A and B represent 2D transformations we can visualize the effect of these transformations using points in the plane. We'll show how they distort a set of points that make up a square.

```
# define the corners of a square
> pts <- matrix(c(1,1, 1,-1, -1,-1, -1,1),4,2,byrow=T)
> pts
     [,1] [,2]
[1,]    1    1
[2,]    1   -1
[3,]   -1   -1
[4,]   -1    1
> plot(pts,xlim=c(-6,6),ylim=c(-6,6),asp=1) # plot the corners
> polygon(pts) # draw edges of square
> transA <- A %*% t(pts)
> transA
     [,1] [,2] [,3] [,4]
[1,]    4    0   -4    0
[2,]    4   -2   -4    2
> newA <- t(transA)
> newA
     [,1] [,2]
[1,]    4    4
[2,]    0   -2
[3,]   -4   -4
[4,]    0    2
> points(newA, col='red') # plot the A transformation
> polygon(newA, lty='dashed', border='red')
> newB <- t(B %*% t(pts)) # do the same for the B transformation
> polygon(newB, lty='dashed', border='blue')
> points(newB, col='blue')
```

```
> legend("topleft", c("transformation A","transformation B"),
lty=c("dashed","dashed"),col=c("red","blue"))
```

The code given above will produce the plot show in the figure below.

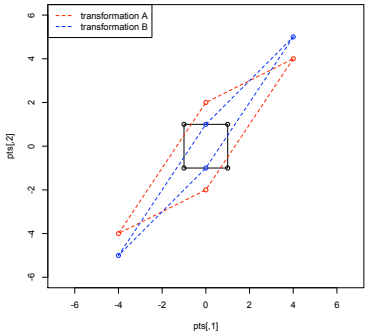


Figure 5.1: Transformation of a square represented by two matrices, A and B

Assignment 5.1

Fig. 5.2 illustrates the geometry of the eigenvectors for matrices A and B as defined above. Note that the lengths of the eigenvector depictions are scaled to be proportional to their eigenvalues. Write R code to reconstruct this figure.

Extra Credit: For extra credit, write a function called `draw_eigenvector()` that will create a similar figure for any arbitrary matrix that represents a 2D transformation. Your function should tak as input a matrix **A**, and a set of points in the plane. Make sure to include code to handle cases where **A** is singular.

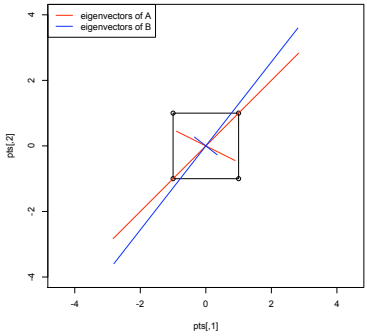


Figure 5.2: Eigenvectors of matrices A and B

5.2 Principal Components Analysis in R

There are two functions in R for carrying out PCA - `princomp()` and `prcomp()`. The `princomp()` function uses the `eigen()` function to carry out the analysis on the covariance matrix or correlation matrix, while `prcomp()` carries out an equivalent analysis, starting from a data matrix, using a technique called singular value decomposition (SVD). The SVD routine has greater numerical accuracy, so the `prcomp()` function should generally be preferred. The `princomp()` function is also useful when you don't have access to the original data, but you do have a covariance or correlation matrix (a frequent situation when re-analyzing data from the literature). We'll concentrate on using the `prcomp()` function.

5.2.1 Bioenv dataset

To demonstrate PCA we'll use a dataset called 'bioenv.txt' (see class wiki), obtained from a book called "Biplots in Practice" (M. Greenacre, 2010). Here is Greenacre's description of the dataset:

> The context is in marine biology and the data consist of two sets of variables observed at the same locations on the sea-bed: the first is a set of biological variables, the counts of five groups of species, and the second is a set of four environmental variables. The data set, called "bioenv", is shown in Exhibit 2.1. The species groups are abbreviated as "a" to "e". The environmental variables are "pollution", a composite index of pollution combining measurements of heavy metal concentrations and hydrocarbons; depth, the depth in metres of the sea-bed where the sample was taken; "temperature", the temperature of the water at the sampling point; and "sediment", a classification of the substrate of the sample into one of three sediment categories.

The first column has no header, and corresponds to the site labels.

```
> b <- read.delim('bioenv.txt',row.names=1) # note use of row.names
    argument
[1] "a"           "b"           "c"           "d"           "e"
[6] "Pollution"   "Depth"       "Temperature" "Sediment"
```

The columns labeled 'a' to 'e' contain the counts of the five species at each site. We'll work with this abundance data for now.

```
> abund <- subset(b, select=c(a,b,c,d,e))
> boxplot(abund, xlab="Species", ylab="Counts", main="Distribution of\
    nSpecies Counts per Site")
```

From the boxplot it looks like the counts for species 'e' are smaller on average, and less variable. The mean and variance functions confirm that.

```
> apply(abund,2,mean)
       a         b         c         d         e
```

```
13.466667  8.733333  8.400000 10.900000  2.966667
> apply(abund,2,var)
        a         b         c         d         e
157.63678  83.44368  73.62759  44.43793  15.68851
```

A correlation matrix suggests weak to moderate associations between the variables, but the scatterplot matrix generated by the `chart.Correlation()` function suggests that many of the relationships have a strong non-linear element.

```
> cor(abund)
           a           b           c           d           e
a  1.0000000  0.67339954 -0.23992888  0.358192050  0.273522301
b  0.6733995  1.00000000 -0.08041947  0.501834036  0.036914702
c -0.2399289 -0.08041947  1.00000000  0.081504483 -0.343540453
d  0.3581921  0.50183404  0.08150448  1.000000000 -0.004048517
e  0.2735223  0.03691470 -0.34354045 -0.004048517  1.000000000

> library(PerformanceAnalytics)
> chart.Correlation(abund)
```

5.2.2 PCA of the Bioenv dataset

Linearity is not a requirement for PCA, as it's simply a rigid rotation of the original data. So we'll continue with our analysis after taking a moment to read the help on the `prcomp()` function.

```
> ?prcomp
> a.pca <- prcomp(abund, center=T, retx=T)
    # center=T mean centers the data
    # retx=T returns the PC scores
    # if you want to do PCA on correlation matrix set scale.=T
    #    -- notice the period after scale!

> summary(a.pca)
Importance of components:
                          PC1    PC2    PC3     PC4     PC5
Standard deviation     14.8653 8.8149 6.2193 5.03477 3.48231
Proportion of Variance  0.5895 0.2073 0.1032 0.06763 0.03235
Cumulative Proportion   0.5895 0.7968 0.9000 0.96765 1.00000
```

We see that approximately 59% of the variance in the data is capture by the first PC, and approximately 90% by the first three PCs.

Let's compare the values return by PCA to what we would get if we carried out eigenanalysis of the covariance matrix that corresponds to our data.

```
> a.pca
Standard deviations:
[1] 14.865306  8.814912  6.219250  5.034774  3.482308

Rotation:
```

```
          PC1         PC2         PC3         PC4         PC5
a  0.81064462  0.07052882 -0.53108427  0.18442140 -0.14771336
b  0.51264394 -0.27799671  0.47711910 -0.63418946  0.17342177
c -0.16235135 -0.88665551 -0.40897655 -0.01149647  0.14173943
d  0.22207108 -0.31665237  0.56250980  0.72941223 -0.04422938
e  0.06616623  0.17696554 -0.08141111  0.17781482  0.96231977
> eigen(cov(abund))
$values
[1] 220.97732  77.70266  38.67908  25.34895  12.12647

$vectors
            [,1]        [,2]        [,3]        [,4]        [,5]
[1,]  0.81064462 -0.07052882  0.53108427  0.18442140 -0.14771336
[2,]  0.51264394  0.27799671 -0.47711910 -0.63418946  0.17342177
[3,] -0.16235135  0.88665551  0.40897655 -0.01149647  0.14173943
[4,]  0.22207108  0.31665237 -0.56250980  0.72941223 -0.04422938
[5,]  0.06616623 -0.17696554  0.08141111  0.17781482  0.96231977
```

Notice that the 'rotation' object returned by the `prcomp` function are the scaled eigenvectors (scaled to have length 1). The standard deviations of the PCA are the square roots of the eigenvalues of the covariance matrix.

5.2.3 Calculating Factor Loadings

Let's calculate the 'factor loadings' associated with the PCs:

```
> V <- a.pca$rotation # eigenvectors
> L <- diag(a.pca$sdev) # diag mtx w/sqrt of eigenvalues on diag.

> a.loadings <- V %*% L
> a.loadings
        [,1]       [,2]       [,3]        [,4]       [,5]
a 12.0504801  0.6217053 -3.3029460  0.92852016 -0.5143835
b  7.6206090 -2.4505164  2.9673232 -3.19300085  0.6039081
c -2.4134024 -7.8157898 -2.5435276 -0.05788214  0.4935804
d  3.3011545 -2.7912626  3.4983893  3.67242602 -0.1540203
e  0.9835813  1.5599356 -0.5063161  0.89525751  3.3510942
```

The magnitude of the loadings is what you want to focus on. For example, species 'a' and 'b' contribute most to the first PC, while species 'c' has the largest influence on PC2.

You can think of the loadings, as defined above, as the components (i.e lengths of the projected vectors) of the original variables with respect to the PC basis vectors. Since vector length is proportional to the standard deviation of the variables they represent, you can think of the loadings as giving the standard deviation of the original variables with respect the PC axes. This implies that the loadings squared sum to the total variance in the original data, as illustrated below.

```
> apply(a.loadings**2, 1, sum)
       a         b         c         d         e
```

```
157.63678  83.44368  73.62759  44.43793  15.68851
> apply(abund, 2, var)
        a         b         c         d         e
157.63678  83.44368  73.62759  44.43793  15.68851
```

5.2.4 Drawing Figures to Represent PCA

PC Score Plots

The simplest PCA figure is to depict the PC scores, i.e. the projection of the observations into the space defined by the PC axes. Let's make a figure with three subplots, depicting PC1 vs PC2, PC1 vs PC3, and PC2 vs. PC3.

```
> plot(a.pca$x[,1], a.pca$x[,2],asp=1,pch=16, xlab='PC1', ylab='PC2',xlim=c
    (-30,30),ylim=c(-30,30))
> plot(a.pca$x[,1], a.pca$x[,3],asp=1,pch=16, xlab='PC1', ylab='PC3',xlim=c
    (-30,30),ylim=c(-30,30))
> plot(a.pca$x[,2], a.pca$x[,3],asp=1,pch=16, xlab='PC2', ylab='PC3',xlim=c
    (-30,30),ylim=c(-30,30))
```

Note that you should always set `asp=1` when plotting PC scores, so that the distances between points are accurate representations. Note too that I used the `xlim` and `ylim` arguments to keep the axis limits the same in all plots; comparable scaling of axes is important when comparing plots. Also note the use of the `mfrow` argument to `par()` in order to setup a multicolumn plot.

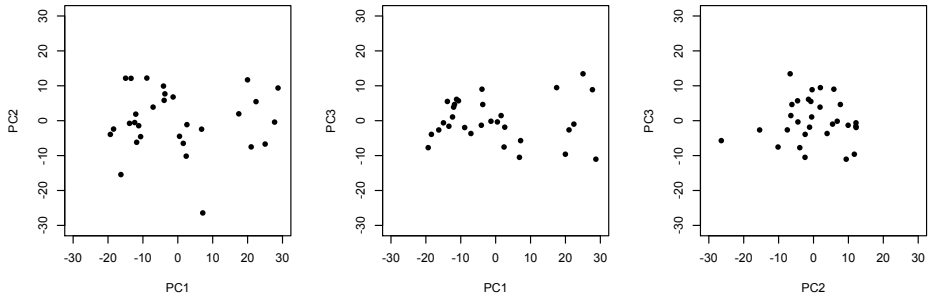


Figure 5.3: Projection of the bioenv dataset into the basis defined by the first three PCs.

As we did in previous weeks we can also use one of the 3D plotting functions to make a 3D scatterplot of the scores.

```
> library(rgl)
> plot3d(a.pca$x[,1:3], asp=1, type='s', xlim=c(-30,30), ylim=c(-30,30),
    zlim=c(-30,30),col='red',size=2)
```

Simultaneous Depiction of Observations and Variables in the PC Space

Let's return to our simple PC score plot. As we discussed above, the loadings are components of the original variables in the space of the PCs. This implies we can depict those loadings in the same PC basis that we use to depict the scores.

```
> plot(a.pca$x[,1], a.pca$x[,2],asp=1,pch=16, xlab='PC1', ylab='PC2',xlim=c
    (-30,30),ylim=c(-30,30))

# get the loadings for each variable w/respect to PCs 1 and 2
> load2d.a <- a.loadings[1,1:2]
> load2d.b <- a.loadings[2,1:2]
> load2d.c <- a.loadings[3,1:2]
> load2d.d <- a.loadings[4,1:2]
> load2d.e <- a.loadings[5,1:2]

# draw arrows depicting loadings
> arrows(0, 0, load2d.a[1], load2d.a[2], length=0.1, col='red')
> text(load2d.a[1], load2d.a[2], 'a',col='red')
> arrows(0, 0, load2d.b[1], load2d.b[2], length=0.1, col='red')
> text(load2d.b[1], load2d.b[2], 'b',col='red')
> arrows(0, 0, load2d.c[1], load2d.c[2], length=0.1, col='red')
> text(load2d.c[1], load2d.c[2], 'c',col='red')
> arrows(0, 0, load2d.d[1], load2d.d[2], length=0.1, col='red')
> text(load2d.d[1], load2d.d[2], 'd',col='red')
> arrows(0, 0, load2d.e[1], load2d.e[2], length=0.1, col='red')
> text(load2d.e[1], load2d.e[2], 'e',col='red')
```

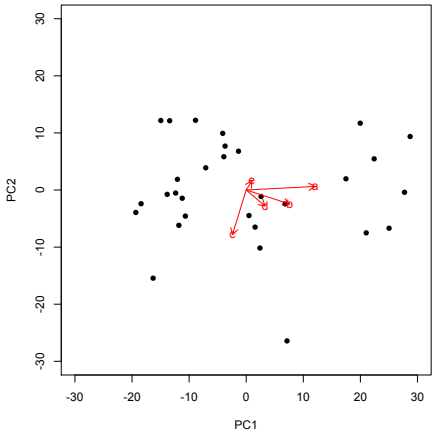


Figure 5.4: PCA of the bioenv dataset. This biplot represents both the observations (black points) and variables (red vectors) in the space of PCs 1 and 2.

The output of the code above should look like Fig. 5.4. Fig. 5.4 is called a 'biplot', as it simultaneously depicts both the observations and variables in the same space. From this biplot we can immediately see that variable 'a' is highly correlated with PC1, but only weakly associated with PC2. Conversely, variable 'c' is strongly correlated with PC2 but only weakly so with PC1. We can also approximate the correlations among the variables themselves – for example 'b' and 'd' are fairly strongly correlated, but weakly correlated with 'c'. Keep in mind however that with respect to the relationships among the variables, this visualization is a 2D projection of a 5D space so the geometry is approximate.

The biplot is a generally useful tool for multivariate analyses and there are a number of different ways to define biplots. We'll study biplots more formally in a few weeks after we've covered singular value decomposition.

Assignment 5.2

Do a PCA analysis on the iris data set with all three species pooled together. Generate a plot showing the projection of the specimens on the first two PC axes as shown in Fig. 5.5. Represent the specimens from a given species with different colors. Make sure you include a legend for your plot.

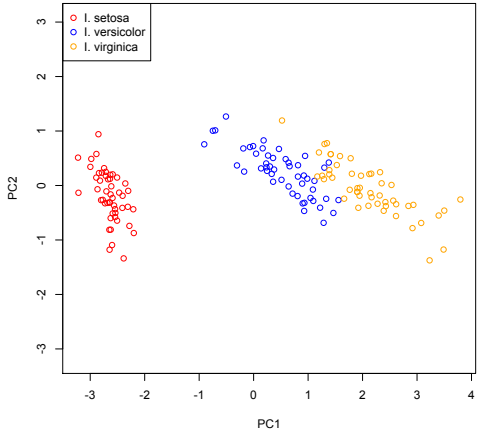


Figure 5.5: PCA of the iris data set. One of your assignments is to reconstruct this figure on your own.

6 Singular value decomposition

6.1 SVD in R

If $\mathbf{A}$ is an $n \times p$ matrix, and the singular value decomposition of $\mathbf{A}$ is given by $\mathbf{A} = \mathbf{USV}^T$, the columns of the matrix $\mathbf{V}^T$ are the eigenvectors of the square matrix $\mathbf{A}^T\mathbf{A}$ (sometimes refered to as the minor product of $\mathbf{A}$). The singular values of $\mathbf{A}$ are equal to the square roots of the eigenvalues of $\mathbf{A}^T\mathbf{A}$.

The `svd()` function computes the singular value decomposition of an arbitrary rectangular matrix. Below I demonstrate the use of the `svd()` function and confirm the relationships described above:

```
> A <- matrix(c(2,1,2,3),nrow=2)
> A
     [,1] [,2]
[1,]    2    2
[2,]    1    3
> a.svd <- svd(A)
> a.svd$u
           [,1]       [,2]
[1,] -0.6618026 -0.7496782
[2,] -0.7496782  0.6618026
# R uses the notation A = u d v' rather than A = u s v'
> a.svd$d
[1] 4.1306486 0.9683709
> all.equal(A, a.svd$u %*% diag(a.svd$d) %*% t(a.svd$v))
[1] TRUE
> AtA <- t(A) %*% A
> eigen.AtA <- eigen(AtA)
> eigen.AtA
$values
[1] 17.0622577  0.9377423
$vectors
          [,1]       [,2]
[1,] 0.5019268 -0.8649101
[2,] 0.8649101  0.5019268
> all.equal(a.svd$d, sqrt(eigen.AtA$values))
[1] TRUE
```

As we discussed in lecture, the eigenvectors of square matrix, $\mathbf{A}$, point in the directions that are unchanged by the transformation specified by $\mathbf{A}$.

6.2 Creating Biplots in R

To illustrate the construction of biplots we'll use the iris data set. The built-in R function is `biplot()`.

```
# leave out the Species variable
> iris.vars <- subset(iris, select=-Species)
# read the prcomp docs and note differnces from princomp
> iris.pca <- prcomp(iris.vars)
> summary(iris.pca)

Importance of components:
                          PC1     PC2    PC3     PC4
Standard deviation     2.0563 0.49262 0.2797 0.15439
Proportion of Variance 0.9246 0.05307 0.0171 0.00521
Cumulative Proportion  0.9246 0.97769 0.9948 1.00000

> biplot(iris.pca, scale=1)  # scale = 1 - alpha
# plot the scores(rows) with slightly smaller font size
> biplot(iris.pca, scale=1, cex=c(0.75,1))
# change the biplot scaling - how does this differ?
> biplot(iris.pca, scale=0, cex=c(0.75,1))
```

Note that the `scale` argument to biplot sets the α value we discussed during lecture, however `scale` $= 1 - \alpha$ (i.e. if `scale` $= 1$, $\alpha = 0$, and if `scale` $= 0$, $\alpha = 1$).

Assignment 6.1

Using R, apply PCA (on the covariances) to the `yeast-subnetwork` data set from week three. Create biplots in the first two principal components using both $\alpha = 0$ and $\alpha = 1$ (i.e. the `scale` argument to biplot). In your biplots change the labels for the observations to integers using the `xlabs` argument to `biplot()` and make the font size for the observations half the size of the varaiable labels. An obvious pattern emerges in the biplot with respect to the gene MEP2. What is this pattern? What subset of conditions (rownames) is most closely related to the vector representing MEP2?

6.2.1 'Seriating' samples using SVD

The term 'seriation' refers to the process of finding an ordering of objects or variables such that they follow a natural ordering with respect to some criteria (e.g. time, similarity, etc.). One way to think about this problem is in terms of ordering objects on a line (i.e. a 1D approximation). Since we've learned that SVD can be used to provide optimal approximations (in the least squares sense) it seems natural to apply the technique to the problem of seriation. We'll illustrate this application by seriating both experimental conditions (samples) and variables (genes) for the yeast expression data set we've been working with. There's some support for the assertion that seriation by SVD is a better method for re-ordering data matrices for heat maps than the more commonly used hierarchical clustering methods that you see in many microar-

ray papers (Wilkinson, L. and M. Friendly. The History of the Cluster Heat Map. The American Statistician. May 1, 2009, 63(2): 179-184. doi link)

I very strongly recommend you use IPython with the `--pylab` option to run the following code.

```
>>> from matplotlib import pyplot
>>> import numpy as np, numpy.linalg as la
>>> # first let's look at the original matrix
>>> yeast = np.loadtxt('yeast-subnetwork-clean.txt',skiprows=1, usecols=
    range(1,15))
>>> yeast.shape
(173, 14)
>>> fig = pyplot.figure(figsize=(4,8))
>>> ax = pylab.imshow(yeast, cmap='seismic')
>>> fig.axes[0].set_aspect(0.2)
>>> fig.show()
```

Since we're going to be creating several figures you essentially the same code let's take a moment to create a function that will take care of the key steps for us.

```
# yeastdraw.py
from matplotlib import pylab, pyplot

def draw_yeast_matrices(matrices = [], titles = [], cmap='seismic'):
    """ draw an image represent of a set of matrices

    matrices and titles should be lists containing np.arrays and strings
    respectively. See the matplotlib docs for color maps other than '
        seismic'
    """
    nmtx = len(matrices)
    width = nmtx * 4
    height = 8

    fig = pyplot.figure(figsize=(width,height))

    # look at the Python docs to read about how the enumerate fxn
    for i, mtx in enumerate(matrices):
        fig.add_subplot(1, nmtx, i+1)
        ax = pylab.imshow(mtx, cmap=cmap)
        fig.axes[i].set_aspect(0.2)
        try:  # try and set title
            fig.axes[i].set_title(titles[i])
        except IndexError:  # if the title doesn't exist
            pass            # just continue with the plotting tasks
    return fig
```

Having created that function we can now put it to use to visualization our seriation of the yeast expression data set.

```
>>> import yeastdraw as yd
```

```
>>> # now do the SVD
>>> u,s,vt = la.svd(yeast)
>>> u1 = u[:,0] # first column of u

# ths specifies how to sort the samples relative to the largest left
# singular vector
>>> u1sort = np.argsort(u1)
# lookup the help for argsort so you understand what it does
>>> help(np.argsort)
>>> s1 = yeast[u1sort] # yeast data with rows sorted by u1

# now create a figure showing original and new ordering
>>> fig = yd.draw_yeast_matrices([yeast, s1],
            ['Original ordering', 'SVD re-ordering of rows'])
>>> fig.show()

# let's repeat it where we sort both rows and cols
>>> v1sort = np.argsort(vt[0])
>>> s2 = s1[:,v1sort]
>>> fig = yd.draw_yeast_matrices([yeast,s1, s2],
        ['Original ordering', 'SVD re-ordering of rows',
        'SVD, rows and cols re-ordered'])
>>> fig.show()
```

6.2.2 Data compression and noise filtering using SVD

Two common uses for singular value decomposition are for data compression and noise filtering. Will illustrate these with two examples involving matrices which represent image data. This example is drawn from an article by David Austin, found on a tutorial about SVD at the American Mathematical Society Website (link).

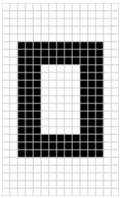
Data compression

Download the file `zeros.dat` from the course wiki. This is a 25×15 binary matrix that represents pixel values in a simple binary (black-and-white) image.

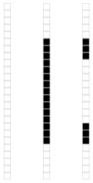
```
> z <- read.delim('zero.dat',header=F)
> z
   V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13 V14 V15
1   1  1  1  1  1  1  1  1  1   1   1   1   1   1   1
2   1  1  1  1  1  1  1  1  1   1   1   1   1   1   1
... output truncated ...

# we'll use the image() function to visualize z
> image(1:15,1:25,t(z),col=c('black','white'),asp=1)
```

This matrix data is shown below in a slightly different form that emphasizes the individual elements of the matrix. As you can see, this matrix can be thought of as being composed of just three types of vectors.



(a) The 'zero' matrix.



(b) The three vector types in the 'zero' matrix.

If SVD is working like expected it should capture that feature of our input matrix, and we should be able to represent the entire image using just three singular values and their associated left- and right-singular vectors.

```
> zsvd <- svd(z)
> round(zsvd$d,2)
 [1] 14.72  5.22  3.31  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
     0.00  0.00  0.00
[15]  0.00
> D <- diag(zsvd$d[1:3])
> D
         [,1]     [,2]     [,3]
[1,] 14.72425 0.000000 0.000000
[2,]  0.00000 5.216623 0.000000
[3,]  0.00000 0.000000 3.314094
> U <- zsvd$u[,1:3]
> V <- zsvd$v[,1:3]
> newZ <- U %*% D %*% t(V)
> all.equal(newZ, zmatx, check.attributes=F)
[1] TRUE

# and let's double check using the image() function
> image(1:15,1:25,t(newZ),col=c('black','white'),asp=1)
```

Our original matrix required 25×15 ($= 375$) storage elements. Using the SVD we can represent the same data using only $15 \times 3 + 25 \times 3 + 3 = 123$ units of storage (corresponding to the truncated U, V, and D in the example above). Thus our SVD allows us to represent the same data with at less than $1/3$ the size of the original matrix. In this case, because all the singular values after the 3rd were zero this is a lossless data compression procedure.

Noise filtering using SVD

The file `noisy-zero.dat` is the same 'zero' image, but now sprinkled with Gaussian noise draw from a normal distribution ($N(0, 0.1)$. As in the data compression case we can use SVD to approximate the input matrix with a lower-dimensional approximation. Here the SVD is 'lossy' as our approximation throws away information. In this

case we hope to choose the approximating dimension such that the information we lose corresponds to the noise which is 'polluting' our data.

```
> nz <- as.matrix(read.delim('noisy-zero.dat',header=F))
> dim(nz)
[1] 25 15
> x <- 1:15
> y <- 1:25
# create a gray-scale representation of the matrix
> image(x,y,t(nz),asp=1,xlim=c(1,15),ylim=c(1,25),col=gray(seq(0,1,0.05)))
> round(nz.svd$d,2)
 [1] 13.63  4.87  3.07  0.40  0.36  0.31  0.27  0.26  0.21  0.19  0.13
     0.11  0.09  0.06
[15]  0.04
# as before the first three singular values dominate
> nD <- diag(nz.svd$d[1:3])
> nU <- nz.svd$u[,1:3]
> nV <- nz.svd$v[,1:3]
> approx.nz <- nU %*% D %*% t(nV)

# now plot the original and approximating matrix side-by-side
> par(mfrow=c(1,2))
> image(x,y,t(nz),asp=1,xlim=c(1,15),ylim=c(1,25),col=gray(seq(0,1,0.05)))
> image(x,y,t(approx.nz),asp=1,xlim=c(1,15),ylim=c(1,25),col=gray(seq
    (0,1,0.05)))
```

As you can see from the images you created the approximation based on the approximation based on the SVD manages to capture the major features of the matrix and filters out much of (but not all) the noise.

6.2.3 Image Approximation Using SVD in Python

The Python Imaging Library (PIL) is a package of routines for manipulating image data in Python. If you're using the Enthought build of Python this was included in your installation. If building packages from scratch you can find the PIL at the PIL home page. Documentation on PIL is available here.

The PIL package provide a more general set of image manipulation features than does R. In the exercises below we will use the PIL package in combination with `numpy` apply SVD to a more complicated image.

Download the module of helper functions `imagehelper.py` from the course wiki and place it somewhere in your PYTHONPATH. Also, download the JPEG image `chesterbw.jpg` and place it in a convenient directory such as `c:/temp` or `~/temp`. I very strongly recommend you use IPython with the `--pylab` option to run the following code.

```
>>> import numpy as np, numpy.linalg as la
>>> from matplotlib import pyplot
>>> import Image # this is the main module in the PIL package
>>> import imagehelper
```

```
>>>  # Let's load and examine our image
>>> cd ~/temp # change to wherever you save the image
>>> im = Image.open("chesterbw.jpg")
>>> im.size
(605, 556) # the image is 605 pixels x 556 pixels
>>> im.show() # assumes you have a default image viewer configured in your
    OS

>>> # matplotlib.pyplot also has an imshow function
>>> pyplot.imshow(im,origin='lower') # (0,0) is in lower left of image
<matplotlib.image.AxesImage object at 0x7aa6d90>
>>> pyplot.imshow(im,origin='lower',cmap='gray') # change the colormap to
    grayscale
<matplotlib.image.AxesImage object at 0x22e87c10>
>>>
>>>   # convert the image to an array
>>> imgarray = np.asarray(im)
>>> imgarray.shape
(556, 605)
>>> fimage = imgarray.astype(np.float32) # svd requires floating points

>>>   # Create a low dimensional approximation to our image
>>> U,S,Vt = la.svd(fimage)
>>> U15 = U[:,:15]
>>> S15 = np.eye(15) * S[:15] # eye() creates an identity matrix
>>> Vt15 = Vt[:15,:]
>>> approx15 = np.dot(U15, np.dot(S15, Vt15)) # 15 dim. approx. of original
     image
>>> approxim = imagehelper.array2image(approx15.astype(np.uint8))
>>> approxim.show()

>>>    # Use pyplot.matshow to view a image representation of a matrix
    directly
>>> pyplot.matshow(fimage, cmap='gray')
>>> pyplot.matshow(approx15, cmap='gray')

>>>    # lets calculate the difference between the two images and plot that
>>> imgdiff = fimage - approx15
>>> pyplot.matshow(imgdiff, cmap='gray')

>>>    # showing off some other features of matplotlib
>>> pyplot.imshow(fimage,cmap='gray_r') # reversed gray scale
>>> fig = pyplot.figure()
>>> fig.add_subplot(1,2,1)
>>> pyplot.imshow(fimage, cmap='gray')
>>> fig.add_subplot(1,2,2)
>>> pyplot.imshow(approx15, cmap='gray')
```

Above we created a rank 15 approximation to the rank 556 original image matrix.

This approximation is crude (as judged by the visual quality of the approximating image) but it does represent a very large savings in space. Our original image required the storage of $605 \times 556 = 336380$ integer values. Our approximation requires the storage of only $15 \times 556 + 15 \times 605 + 15 = 17430$ integers. This is a saving of roughly 95%. Of course, as with any lossy compression algorithm, you need to decide what is the appropriate tradeoff between compression and data loss for your given application.

The Python Image Library has lots of useful functions for manipulating image data. You might spend some time checking out the documenation (see the PIL homepage given above). For more info on the Matplotlib built in color maps see: Matplotlib colormaps.

Assignment 6.2

Write a function (`svd_img()`) in Python that automates the creation of a lower dimensional approximation of a grayscale image using SVD. Test your function on various images using a variety of approximating dimensions (e.g. 5,10, 25, 50, 100, 250 on the `chesterbw.jpg` image). Your function should take as input a floating point array representing the original image and an integer specifying the approximating dimension (i.e. function will be `svd_img(imgarray,dim)`). Your function should return two objects: 1) an array representing the approximated image; and 2) an array representing the difference between the original and approximating images.
In addition to your code consider the following questions:

- When analyzing `chesterbw.jpg`, at some approximating dimensions you'll notice interesting artifacts. How do these relate to the original image?
- What is the lowest approximating dimension where you would you consider the image to be recognizable as a dog?
- At what approximating dimension would you judge the image to be "close enough" to the original by the casual observer? What is the storage saving of this approximation relative to the original image?
- How does the difference array (original - approximating) change as the approximating dimension changes? Is there a particular type of image information that seems most prominent in the difference array?

Extra credit: write a function (`svd_img_plot()`)that creates a single figure in matplotlib that with three subfigures that compare the original, approximating, and difference arrays.