Scientific Computing for Biologists

Bio313, Fall Semester 2011

PAUL M. MAGWENE

Duke University
Durham, North Carolina, USA

Contents

1

Getting your feet wet with R and Python

1.1 Getting Acquainted with R

Starting the default R GUI

Starting R is simple. If you're using Windows simply navigate to the R subfolder from the Start Menu. On a Unix/Linux system invoke the program by typing `R`. On OS X start R by clicking the R icon from the Dock or in the Applications folder.

The OS X and Windows version of R provide a simple GUI interface that simplifies certain tasks. When you start up the R GUI you'll be presented with a single window, the R console. The rest of this document will assume you're using R under Windows or OS X.

R Studio

RStudio is a new IDE for R being developed under an open source model (see the RStudio GitHub Site). It provides a nicely designed graphical interface that is consistent across platforms. It can even run as a server, allowing you to access R via a web interface!

Check out the RStudio Docs for detailed info on configuring Rstudio. We'll go over some of RStudio's nice features in class.

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. 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()`.

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.

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

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

Using R as a Calculator

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

```
> 3.14 * 2.5^2
[1] 19.625
> pi * 2.5^2 # R knows about some mathematical 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.

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 (note that '=' is an alternative assignment operator)
[1] TRUE
> 10 != 10 # does not equal
[1] FALSE
> 10 == (sqrt(10)^2) # Are you surprised by the result? See below..
[1] FALSE
> 4 == (sqrt(4)^2) # Even more confused?
[1] TRUE
```

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

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

Working with Vectors in R

Vector Arithmetic and Comparison

Remember that in 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
```

Indexing Vectors

```
> 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
```

For a vector of length n, we can access the elements by the indices $1 \ldots n$. Trying to access an element beyond these limits returns a special constant called `NA` (Not Available) that indicates missing or non-existent values.

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 '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
> s <- seq(0.5, 1.5, 0.33) # 'by' is the 3rd argument
                           # so you don't have to specify it
> 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
```

Some Useful Functions

You've already seem a number of functions (e.g. `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 vectors

An important function that you've used extensively but we've glossed over is the `c()` function. This is short for 'concatenate' or 'combine' and as you've seen it combines it's arguments to form a vector.

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).

Note that we created a variable with the name '`test.scores`'. If you have previous programming experience you might be surpised that this works. Unlike most languages, R (and S) allow 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 and max returned as a vector of length 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
```

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 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
```

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.

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
[[1]]
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "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
     0.100

> algae.table[[1]]
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "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
     0.100
```

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

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. 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.

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
```

Simple Plots in R

One of the advantages of R is it's ability to produce a variety of plots and statistical graphics. Try out the following:

```
> hist(turtles$width)  # histogram plot
> hist(turtles$width,10) # produces a histogram with 10 bins
> hist(turtles$width,breaks=10, xlab="Carapace Width", probability=T)
>
> boxplot(turtles$width) # simple box plot
> boxplot(list(turtles$length, turtles$width, turtles$height),
+        names=c("Carapace\nLength","Carapace\nWidth","Carapace\nHeight"),
+        ylab="millimeters") # a fancy box plot showing multiple variables
> title("Turtle Shell Variables")
>
> plot(turtles$length, turtles$width)
> plot(turtles$length ~ turtles$width) # how does this differ from the plot
     above?
> plot(turtles$length, turtles$width, xlab="Carapace Length(mm)",
+      ylab="Carapace Width(mm)")
> title("Relationship Between\nLength and Width")
```

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

```
> demo(graphics)
```

1.2 Getting Acquainted with Python

Starting the Python interpretter

The Python interpretter can be started in a number of ways. The simplest way is to open a terminal and type `python`. Go ahead and do this to make sure you have a working version of the default Python interpretter available on your system. From within the default interpretter

you can type `Ctrl-d` (Unix,Mac) or `Ctrl-z` (Windows) to stop the interpretter and return to the command line.

For interactive use, the default interpretter isn't very feature rich, so the Python community has developed a number of GUIs or shell interfaces that provide more functionality. For this class we will be using a shell interface called IPython.

Recent versions of IPython (v0.11) provides both terminal and GUI-based shells. The EPD installer will place a number of shortcuts on your Start Menu or in Launchpad on OS X 10.7, including ones that read `PyLab` and `QtConsole`. These are a terminal based and GUI based versions of IPython respectively, both of which automatically load key numerical and plotting libraries. Click on both of these icons to compare their interfaces.

To get the functionality of `PyLab` from the terminal, run the following command:

```
$ ipython --pylab
```

To get the equivalent of `QtConsole` you can run ipython with the following arguments:

```
$ ipython qtconsole --pylab
```

If you'd prefer a dark background, call QtConsole as so:

```
$ ipython qtconsole --pylab --colors=linux
```

QtConsole is a recent addition to IPython and there may still be bugs to be sorted out, but it provides some very nice features like 'tooltips' (shows you useful information about functions as you type) and the ability to embed figures and plots directly into the console, and the ability to save a console session as a web page (with figures embedded!).

Quick IPython tips

IPython has a wealth of features, many of which are detailed in its documentation. There are also a number of videos available on the IPython page which demonstrate some of it's power. Here are a few key features to get you started and save you time:

- *Don't retype that long command!* — You can scroll back and forth through your previous inputs using the up and down arrow keys (or `Ctrl-p` and `Ctrl-n`); once you find what you were looking forward you can edit or change it. For even faster searching, start to type the beginning of the input and then hit the up arrow.

- *Navigate using standard Unix commands* — IPython lets you use standard Unix commands like `ls` and `cd` and `pwd` to navigate around your file system (even on Windows!)

- *Use <Tab> for command completion* — when your navigating paths or typing function names in you can hit the `<Tab>` key and IPython will show you matching functions or filenames (depending on context). For example, type `cd ./<Tab>` and IPython will show you all the files and subdirectories of your current working directory. Type a few of the letters of the names of one of the subdirectories and hit `<Tab>` again and IPython will complete the name if it finds a unique match. Tab completeion allows you to very quickly navigate around the file system or enter function names so get the hang of using it.

Accessing the Documentation in Python

Python comes with extensive HTML documentation and the Python interpreter has a help function that works similar to R's `help()`.

```
>>> help(sum)
Help on built-in function sum in module __builtin__:

sum(...)
    sum(sequence, start=0) -> value

    Returns the sum of a sequence of numbers (NOT strings) plus the value
    of parameter 'start'.  When the sequence is empty, returns start.
```

IPython also lets you use proceed the function name with a question mark, like in R:

```
In [1]: ?sum
Type:       builtin_function_or_method
Base Class: <type 'builtin_function_or_method'>
String Form:<built-in function sum>
Namespace:  Python builtin
Docstring:
sum(sequence[, start]) -> value

Returns the sum of a sequence of numbers (NOT strings) plus the value
of parameter 'start' (which defaults to 0).  When the sequence is
empty, returns start.
```

Using Python as a Calculator

As with R, the simplest way to use Python is as a fancy calculator. Let's explore some simple arithmetic operations:

```
>>> 2 + 10   # this is a comment
12
>>> 2 + 10.3
 12.300000000000001  # 0.3 can't be represented exactly in floating point
     precision
>>> 2 - 10
-8
>>> 1/2  # integer division
0
>>> 1/2.0  # floating point division
0.5
>>> 2 * 10.0
20.0
>>> 10**2  # raised to the power 2
100
>>> 10**0.5  # raised to a fractional power
3.1622776601683795
>>> (10+2)/(4-5)
-12
>>> (10+2)/4-5  # compare this answer to the one above
-2
```

In addition to integers and reals (represented as floating points numbers), Python knows about complex numbers:

```
>>> 1+2j    # Engineers often use 'j' to represent imaginary numbers
(1+2j)
>>> (1 + 2j) + (0 + 3j)
(1+5j)
```

Some things to remember:

- Integer and floating point division are not the same in Python. Generally you'll want to use floating point numbers.
- The exponentiation operator in Python is **
- 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.
- The standard math functions like `cos()` and `log()` are not available to the Python interpeter by default. To use these functions you'll need to `import` the math library as shown below.

For example:

```
>>> 1/2
0
>>> 1/2.0
0.5
>>> cos(0.5)

Traceback (most recent call last):
  File "<pyshell#2>", line 1, in -toplevel-
    cos(0.5)
NameError: name 'cos' is not defined
>>> import math     # make the math module available
>>> math.cos(0.5) # access the cos() function in the math module
0.87758256189037276
>>> pi      # pi isn't defined in the default namespace

Traceback (most recent call last):
  File "<pyshell#5>", line 1, in -toplevel-
    pi
NameError: name 'pi' is not defined
>>> math.pi # however pi is defined in math
3.1415926535897931
>>> from math import * # bring everything in the math module into the
    current namespace
>>> pi
3.1415926535897931
>>> cos(pi)
-1.0
```

Comparison Operators in Python

The comparison operators in Python work the same was as they do in R (except they don't work on lists default). Repeat the comparison excercises given above.

More Data Types in Python

You've already seen the three basic numeric data types in Python - integers, floating point numbers, and complex numbers. There are two other basic data types - Booleans and strings.

Here's some examples of using the Boolean data type:

```
>>> x = True
>>> type(x)
<type 'bool'>
>>> y = False
>>> x == y
False
>>> if x is True:
...     print 'Oh yeah!'
...
Oh yeah!
>>> if y is True:
...     print 'You betcha!'
... else:
...     print 'Sorry, Charlie'
...
Sorry, Charlie
>>>
```

And some examples of using the string data type:

```
>>> s1 = 'It was the best of times'
>>> type(s1)
<type 'str'>
>>> s2 = 'it was the worst of times'
>>> s1 + s2
'It was the best of timesit was the worst of times'
>>> s1 + ', ' + s2
'It was the best of times, it was the worst of times'
>>> 'times' in s1
True
>>> s3 = "You can nest 'single quotes' in double quotes"
>>> s4 = 'or "double quotes" in single quotes'
>>> s5 = "but you can't nest "double quotes" in double quotes"
  File "<stdin>", line 1
    s5 = "but you can't nest "double quotes" in double quotes"
                                ^
SyntaxError: invalid syntax
```

Note that you can use either single or double quotes to specify strings.

Simple data structures in Python: Lists

Lists are the simplest 'built-in' data structure in Python. List represent ordered collections of arbitrary objects.

```
>>> l = [2, 4, 6, 8, 'fred']
>>> l
[2, 4, 6, 8, 'fred']
>>> len(l)
5
```

Python lists are zero-indexed. This means you can access lists elements `0` to `len(x)-1`.

```
>>> l[0]
2
>>> l[3]
8
>>> l[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

You can use negative indexing to get elements from the end of the list:

```
>>> l[-1] # the last element
'fred'
>>> l[-2] # the 2nd to last element
8
>>> l[-3] # ... etc ...
6
```

Python lists support the notion of 'slices' - a continuous sublist of a larger list. The following code illustrates this concept:

```
>>> y = range(10)  # our first use of a function!
>>> y
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> y[2:8]
[2, 3, 4, 5, 6, 7]
>>> y[2:-1] # the slice
[2, 3, 4, 5, 6, 7, 8]
>>> y[-1:0] # how come this didn't work? see the next example...
[]
>>> y[-1:0:-2]  # slice from last to first, stepping backwards by 2
[9, 7, 5, 3, 1]
```

Using NumPy arrays

As mentioned during lecture, Python does not have a built-in data structure that behaves in quite the same way as do vectors in R. However, we can get very similar behavior using a library called `NumPy`.

NumPy does not come with the standard Python distribution, but it does come as an included package if you use the Enthought Python distribution. Alternately you can download NumPy from the SciPy project page at: http://numpy.scipy.org.

The NumPy package comes with documentation and a tutorial. You can access the documentation at http://docs.scipy.org/doc/

```
>>> from numpy import array # a third form of import
>>> x = array([2,4,6,8,10])
```

```
>>> -x
array([ -2,  -4,  -6,  -8, -10])
>>> x ** 2
array([  4,  16,  36,  64, 100])
>>> pi * x # assumes pi is in the current namespace
array([  6.28318531,  12.56637061,  18.84955592,  25.13274123,
    31.41592654])
>>> y = array([0, 1, 3, 5, 9])
>>> x + y
array([ 2,  5,  9, 13, 19])
>>> x * y
array([ 0,  4, 18, 40, 90])
>>> z = array([1, 4, 7, 11])
>>> x+z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

The last example above shows that, unlike R, NumPy arrays in Python are not 'recycled' if lengths do not match.

Remember that lists and arrays in Python are zero-indexed rather than one-indexed.

```
>>> x
array([ 2,  4,  6,  8, 10])
>>> len(x)
5
>>> x[0]
2
>>> x[1]
4
>>> x[4]
10
>>> x[5]

Traceback (most recent call last):
  File "<pyshell#52>", line 1, in -toplevel-
    x[5]
IndexError: index out of bounds
```

Numpy arrays support the comparison operators and return arrays of booleans.

```
>>> x < 5
array([ True, True, False, False, False], dtype=bool)
>>> x >= 6
array([0, 0, 1, 1, 1])
```

NumPy also supports the combination of comparison and indexing that R vectors can do. There are also a variety of more complicated indexing functions available for NumPy; see the Indexing Routines in the Numpy docs.

```
>>> x[x < 5]
array([2, 4])
>>> x[x >= 6]
```

```
array([ 6,  8, 10])
>>> x[(x<4)+(x>6)]  # 'or'
array([ 2,  8, 10])
```

Note that Boolean addition is equivalent to 'or' and Boolean multiplication is equivalent to 'and'.

Most of the standard mathematical functions can be applied to NumPy arrays however you must use the functions defined in the `numpy` module.

```
>>> x
array([ 2,  4,  6,  8, 10])
>>> import math
>>> math.cos(x)

Traceback (most recent call last):
  File "<pyshell#67>", line 1, in -toplevel-
    math.cos(x)
TypeError: only length-1 arrays can be converted to Python scalars.
>>> import numpy
>>> numpy.cos(x)
array([-0.41614684, -0.65364362,  0.96017029, -0.14550003, -0.83907153])
```

Simple Plots in Python

The Matplotlib package is the de facto standard for producing publication quality scientific graphics in Python. Matplotlib is included with the EPD and was automatically pulled into the interpretter namespace if you're using the IPython PyLab or QtConsole configurations.

Here are some simple plotting examples using matplotlib:

```
# only necessary if not using pylab
>>> from pylab import *

>>> import numpy as np # use a shorter alias

# load the turtle data using the numpy.loadtxt function (see ?np.
    loadtxt)
# we skipped the first row (header) and the first column (info about
    sex)
# we'll see how to deal with more sophisticated data import next week

>>> turt = np.loadtxt('turtles.txt', skiprows=1, usecols=(1,2,3))
>>> turt.shape
(48, 3)

# draw bivariate scatter plot
>>> scatter(turt[:,0], turt[:,1])

# give the axes some labels and a title for the plot
>>> xlabel('Length')
>>> ylabel('Width')
>>> title('Turtle morphometry')
```

Sweave for R

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

Here's a simple Sweave document to get you started:

```
\documentclass{article}
\begin{document}

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

<<>>=
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 and standard
deviation one.

\end{document}
```

Type those lines into a text editor and save the document with the name `sweave1.Rnw`.

Let's break down the various pieces of the document. The first two lines and the last line represent LaTeX commands.

```
\documentclass{article}
\begin{document}
    ....
\end{document}
```

For simple Sweave documents that's all the LaTeX you need to learn. However, learning a little bit more about the document preparation system gives you the option of producing very nicely formatted output as we'll see in a little bit.

The R code is preceeded by the text `<<>>=`. This tells Sweave that you're starting a code chunk. The `@` symbol after the code chunk tells Sweave that you're going back to writing documentation chunks.

If you already have a working installation of LaTeX on your computer you can now compile this into a nicely formatted document from the R interpretter. Change the R working directory so that it's in the same directory where you saved the `sweave1.Rnw` file. The execute the following commands:

```
> library(tools)  # makes the texi2dvi function available
> Sweave('sweave1.Rnw') # compiles our Sweave document into 'sweave1.tex'
> texi2dvi("sweave1.tex", pdf = TRUE)
```

These commands will produce two new documents – `sweave1.tex` and a PDF document, `sweave1.pdf`. You can open the `sweave1.tex` file in any text editor and you'll see that it's

just a slightly modified version of the `sweave1.Rnw` file you created. The PDF file contains the nicely formatted report.

If you got an error message the most likely reason is that R can't find the path to your LaTeX executable. To check this try:

```
> texi2dvi('sweave1.tex', pdf=TRUE, quiet=FALSE)
```

If that's the case, you can fix that by typing the following into the R command line (on OS X):

```
> Sys.setenv("PATH" = paste(Sys.getenv("PATH"),"/usr/texbin",sep=":"))
```

Then try executing the `texi2dvi` command again. If that solved your problem you can make this permanent by adding that line to your .Rprofile (located in `/Users/yourname` on OS X; if the file doesn't already exist go ahead and create it). If that doesn't work please see me for troubleshooting help.

RStudio makes Sweaving easy!

RStudio hides some of the complexity of Sweaving documents. Simply create or open your Sweave document (use the .Rnw extension) in RStudio and then hit the `Compile PDF` button. If your LaTeX setup is working, and the document and code are valid, Rstudio will compile everything behind the scenes and pop up a nice PDF.

A fancier Sweave document

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

```
\documentclass[letterpaper]{article}
\usepackage[margin=0.75in]{geometry}

\title{My Second Sweave Report}
\author{John Q. Public}

\begin{document}
\maketitle
This is a still a simple Sweave file. However,
now it includes several code chunks and several
\LaTeX\ specific commands.

\section{Sampling from the random normal distribution}

<<>>=
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 ($\sigma = 1$).


\section{Generating figures}
```

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

% this tells Sweave to set the graphics
% to be half the width of the text
\setkeys{Gin}{width=0.5\textwidth}
<<fig=TRUE>>=
hist(z)
@

\end{document}
```

Notice how we put an argument, `fig=TRUE` within the second code chunk delimiter. This will tell Sweave to automatically imbed a figure with the histogram graphic we created into our report. Save this as `sweave2.Rnw` and repeat the above steps to compile it into a PDF report.

For a full overview of Sweave's capabilities see the documentation for Sweave availabe at http://www.stat.uni-muenchen.de/~leisch/Sweave/.

Pweave for literate programming in Python

Pweave uses almost exactly the same syntax as Sweave to delimit code and document chunks. Here's a simple Pweave document.

```
\documentclass[letterpaper]{article}
\usepackage[margin=0.75in]{geometry}
\usepackage{graphicx} % unlike Sweave, Pweave doesn't
                      % pull this in automatically

\title{My First Pweave Report}
\author{John Q. Public}

\begin{document}
\maketitle
This is a still a simple Pweave file. As in our Sweave example,
there are several code chunks and we've included a figure.

\section{Sampling from the random normal distribution}

<<>>=
from numpy import random
z = random.normal(loc=0, scale=1, size=30)
@

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


\section{Generating figures}

We can also automatically imbed graphics in our
```

```
report. For example, the following will generate
a histogram.

% this tells Sweave to set the graphics
% to be half the width of the text
\setkeys{Gin}{width=0.5\textwidth}
<<fig=True>>=
import pylab
pylab.hist(z)
@

\end{document}
```

Note that the second code chunk, we wrote `fig=True` in the Pweave document, whereas we wrote `fig=TRUE` for the Sweave document. This minor difference reflects the different syntax for boolean values in R and Python.

Save that code in a text file called `pweave1.Pnw` and from the bash shell (*not* in the Python interpretter) type the following command:

```
Pweave -f "tex" pweave1.Pnw
```

The option `-f "tex"` tells Pweave to output a file (Note: from the Windows command prompt you must use double quotes around "tex", on Unix-based systems either single our double quotes work fine). Assuming you got no error messages, you can then compile this to PDF using the following command:

```
pdflatex pweave1.tex
```

2

Data structures and function in R and Python

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 + 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
```

> **Assignment 1**: In R, use the dot product operator and the `acos()` function to calculate the angle (in radians) between the vectors `x = [-3, -3, -1, -1, 0, 0, 1, 2, 2, 3]` and `y = [-8, -5, -3, 0, -1, 0, 5, 1, 6, 5]`.

Vector Operations in Python

The Python equivalent of the R code above is:

```
>>> import numpy
>>> x = numpy.arange(start=1, stop=16, step=1)
>>> y = numpy.arange(10,25) # default step = 1
>>> x
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
>>> y
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24])
>>> x+y
array([11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39])
>>> x-y
array([-9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9])
>>> 3*x
array([ 3,  6,  9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45])
>>> z = numpy.dot(x,x) # no built-in dot operator, but a dot fxn in numpy
>>> z
1240
```

Note the use of the `numpy.arange()` function. `numpy.arange()` works like R's `sequence()` function and it returns a Numpy array. However, notice that the values go up to but don't include the specified `stop` value. Use `help()` to lookup the documentation for `numpy.arange()`. Python also includes a `range()` function that generates a regular sequence as a Python list object. The `range()` function has `start`, `stop`, and `step` arguments but these can only be integers. Here are some additional examples of the use of `arange()` and `range()`:

```
>>> z = numpy.arange(1,5,0.5)
>>> z
array([ 1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5])
>>> range(1,20,2)
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> range(1,5,0.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: range() integer step argument expected, got float.
```

Writing Functions in R

So far we've been 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
}
```

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){
+   x^2 + y^2     # don't type the '+' symbols, these show continuation
    lines
+ }

> 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').

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 `New Document` in OS X) in the R GUI (`File > New > R Script` in RStudio). 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) {
  # Given a numeric vector, returns a unit vector in the same direction
  x/veclength(x)
}
```

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 (R GUI); `Edit > Source File..` in RStudio):

```
> source("vecgeom.R")
> x <- c(-3,-3,-1,-1,0,0,1,2,2,3)
> veclength(x)
[1] 6.164414
> ux <- unitvector(x)
> ux
 [1] -0.4866643 -0.4866643 -0.1622214 -0.1622214  0.0000000  0.0000000
 [7]  0.1622214  0.3244428  0.3244428  0.4866643
> veclength(ux)
 [1] 1
```

Assignment 2: Write a function in R 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 $\vec{x}$:

Writing Functions in Python

The general form of a Python function is as follows:

```
def funcname(arg1,arg2):
    # one or more expressions
    return someresult # arbitrary python object (could even be another
        function)
```

An important thing to remember when writing functions is that Python is white space sensitive. In Python code indentation indicates scoping rather than braces. Therefore you need to maintain consistent indendation. This may surprise those of you who have extensive programming experience in another language. However, white space sensitivity contributes signficantly to the readability of Python code. Use a Python aware programmer's editor and it will become second nature to you after a short while. I recommend you set your editor to substitute spaces for tabs (4 spaces per tab), as this is the default convention within the python community.

Here's an example of defining and using a function in the Python interpreter:

```
>>> def mypyfunc(x,y):
...     return x**2 + y**2 + 3*x*y
...
>>> mypyfunc(10,12)
604
>>> a = numpy.arange(1,5,0.5)
>>> b = numpy.arange(2,6,0.5)
>>> mypyfunc(a,b)
array([  11.  ,   19.75,   31.  ,   44.75,   61.  ,   79.75,  101.  ,
        124.75])
>>> a = range(1,5)
>>> b = range(1,5)
>>> mypyfunc(a,b)

Traceback (most recent call last):
  File "<pyshell#52>", line 1, in -toplevel-
    mypyfunc(a,b)
  File "<pyshell#45>", line 2, in mypyfunc
    return x**2 + y**2 + 3*x*y
TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
>>>
```

Note that this function works for numeric types (`ints` and `floats`) as well as `numpy.arrays` but not for simple Python lists. If you wanted to make this function work for lists as well you could define the function as follows:

```
>>> def mypyfunc(x,y):
...     x = numpy.array(x)
...     y = numpy.array(y)
...     return x**2 + y**2 + 3*x*y
...
```

```
>>> a
[1, 2, 3, 4]
>>> b
[1, 2, 3, 4]
>>> mypyfunc(a,b)
array([ 5, 20, 45, 80])
```

Putting Python functions in Modules

As with R, you can define your own Python modules that contain user defined functions. Using a programmer's text editor, write your function(s) and save it to a file with a `.py` extension in a directory in your PYTHONPATH (see [[Setting Paths|setting-paths]].

```
# functions defined in vecgeom.py
import numpy

def veclength(x):
    """Given a numeric vector, returns length of that vector"""
    x = numpy.array(x)
    return numpy.sqrt(numpy.dot(x,x))


def unitvector(x):
    """Given a numeric vector, returns a unit vector in the same direction
        """
    x = numpy.array(x)
    return x/veclength(x)
```

To access your function use an `import` statement:

```
>>> import vecgeom
>>> x = [-3,-3,-1,-1,0,0,1,2,2,3]
>>> help(vecgeom.veclength)
Help on function veclength in module vecgeom:

veclength(x)
    Given a numeric vector, returns length of that vector

>>> vecgeom.veclength(x)
6.164414002968976
>>> from vecgeom import * # import all functions from the vecgeom module
>>> print vecgeom.unitvector(x)
[-0.48666426 -0.48666426 -0.16222142 -0.16222142  0.          0.
     0.16222142  0.32444284  0.32444284  0.48666426]
```

> **Assignment 3**: Write Python code for the vector projection and component functions as described in Assignment 2. In your Pweave document illustrate the use of these functions with several examples. Remember that your module will need to have access to the `numpy` module so include an appropriate `import` statement.

Dealing with Data Subsets in R

Manipulating or analyzing subsets of data is one of the most common tasks in R. The `subset()` function comes in handy for such operations. Consider the data set `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"
>
> # Now we'll apply the subset() function
>
> turt.sub <- subset(turtles, select = -sex)
> names(turt.sub)
[1] "length" "width"  "height"
> turt.sub
   length width height
1      98    81     38
2     103    84     38
3     103    86     42
  # output truncated
```

In the example above we create a subset of the original data set by excluding the variable indicating the sex of each individual using the argument `select = -sex`. We can also explicit include only certain variables, like this:

```
> turt.sub2 <- subset(turtles, select=c(height,width))
> turt.sub2
   height width
1      38    81
2      38    84
3      42    86
  # output truncated
```

`subset()` allows you to do more than just select variables to include. You can use the second positional argument to specify matching criteria. For example:

```
# gives only female turtles, all variables except sex
> female.turts <- subset(turtles, sex == "f", select = -sex)
> dim(female.turts)
[1] 24  3

# same for male turtles
> male.turts <- subset(turtles, sex == "m", select = -sex)
> dim(male.turts)
[1] 24  3

# gives only females with length > 125, all variables
> big.females <- subset(turtles, sex == "f" & length > 125)
```

The subset function is especially useful when combined with the function sapply() which allows you to apply a function of interest to each variable. For example:

```
> min(turtles)
Error in Summary.data.frame(..., na.rm = na.rm) :
        only defined on a data frame with all numeric or complex variables
> min(turt.sub)  # unexpected result
[1] 35
> sapply(turt.sub, min) # here's what we were shooting for
length  width height
    93     74     35
> sapply(female.turts, min) # for females
length  width height
    98     81     38
> sapply(male.turts, min) # for males
length  width height
    93     74     35
```

Notice how the min() function chokes on the complete data set because the function is not defined for factor variables. In the second example min(turt.sub) returns a valid result, but also not exactly what we wanted. In this case it looked for the minimum value across all the objects passed to it. In the third case we use the sapply() function and get the minimum on a variable-by-variable basis. Please take a moment to look at the documentation for the sapply() function and cook up some examples of your own.

Anderson's (Fisher's) iris data set. Anderson's (or Fisher's) iris 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.

> **Assignment 4**: Calculate a summary table as well as correlation and covariance matrices for each of the species in the iris data set. Use help.search() and apropos() to lookup any necessary function names.

Exploring Univariate Distributions in R

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
```

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. 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. See the documentation for more info.

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.

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 2.1: 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.

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 2.2: 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 (standard R GUI) or in `Tools > Install Packages...` in RStudio (see also the `Packages` tab in the lower-right window in RStudio). 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)
> example(beanplot)
```

If you ran the examples in RStudio, use the `Clear All` option in the `Plots` tab after running the examples in order to reset parameters that the examples changed.

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. There is also a asymmetrical version of the beanplot which can be used to more directly compare distributions between two groups. We explore this below. Note too 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.

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)')
```

Simple t-tests in R

Student's t-tests can be carried out in R using the function `t.test()`. The `t.test()` function can perform one and two-sample t-tests (i.e. comparing a sample of interest against a hypothesized mean, or comparing the means of two samples). The `t.test()` function also supports a 'formula' interface for two-sample t-tests similar to the `lm()` function.

```
> t.test(width ~ sex, data=turtles)

        Welch Two Sample t-test

data:  width by sex
t = 4.7015, df = 35.355, p-value = 3.862e-05
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
  8.122699 20.460634
sample estimates:
mean in group f mean in group m
      102.58333        88.29167
```

The asymmetric version of the boxplot is very useful for comparing distributions of the same variable between two groups. To generate such plots use the argument `side='both'` as an argument to `beanplot`.

```
> beanplot(width ~ sex, data = turtles, side='both', col=list(c('red'),c('
    black')))
```

As you can see this splits the beanplot in half for each group and puts them back to back to facilitate comparison. The difference in the mean of the two groups is visually obvious from the beanplot.

Assignment 5:

- Prepare beanplots showing samples grouped by `Species` for each of the quantitative variables in the iris data set. Label the x- and y-axes of your boxplots and give each plot a title.
  - **Tip**: since there are three species, you can't use the `side='both'` argument, and you'll need to extend the `col=list` argument to add a third color.
- Carry out two-sample t-tests contrasting `versicolor` and `virginica` for each of the four morphometric variables in the iris data set.
  - **Tip**: use the `subset()` function to create a subset of the iris data containing just these two species.
- Write a brief paragraph interpreting the results of the t-tests you conducted.

Exploring Bivariate Distributions in R

Scatterplots

When dealing with pairs of continuous variables a scatter plot is the obvious choice. The standard `plot` function can be used:

```
> plot(turtles$length, turtles$width)
> plot(turtles$length ~ turtles$width)
```

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

```
> plot(length ~ width, data=turtles)
```

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

```
> xyplot(length ~ width, data = turtles)
```

Regression in R

R has very flexible built in functions for fitting linear models. Bivariate regression is the simplest case of a linear model.

```
> turtles <- read.table('turtles.txt',header=T)
> names(turtles)
[1] "sex"    "length" "width"  "height"
> regr <- lm(turtles$width ~ turtles$length)
> class(regr)
[1] "lm"
> names(regr)
```

```
 [1] "coefficients"  "residuals"     "effects"       "rank"          "
    fitted.values"
 [6] "assign"        "qr"            "df.residual"   "xlevels"       "call"
[11] "terms"         "model"
> summary(regr)

Call:
lm(formula = turtles$width ~ turtles$length)

Residuals:
     Min       1Q   Median       3Q      Max
-5.57976 -1.66578 -0.04471  1.73752  5.97104

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)     19.9434     2.3877   8.353 8.99e-11 ***
turtles$length   0.6055     0.0189  32.033  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.654 on 46 degrees of freedom
Multiple R-Squared: 0.9571,     Adjusted R-squared: 0.9562
F-statistic:  1026 on 1 and 46 DF,  p-value: < 2.2e-16

> plot(turtles$width ~ turtles$length)  # scatter plot with turtles$length
    on x axis
> abline(regr)  # plot the regression line
```

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.

> **Assignment 6**: Write your own regression function (i.e. your code shouldn't refer to the built in regression functions) for mean centered vectors in R. The function will take as it's input two vectors, $\vec{x}$ and $\vec{y}$. The function should return:
>
> 1. a list containing the mean-centered versions of these vectors
> 2. the regression coefficient b in the mean centered regression equation $\hat{\vec{y}} = b\vec{x}$
> 3. the coefficient of determination, R^2
>
> Demonstrate your regression function by using it to carry out regressions of Sepal.Length on Sepal.Width separately for the 'setosa' and 'virginica' specimens from the iris data set (again, `subset()` is your friend). Include plots in which you use the `plot()` and `abline()` functions to illustrate your calculated regression line.

3

Matrices and matrix operations in R and Python

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.

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

```
[1] 2
> dim(Y) <- c(2,6)
> Y
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    7    2    8    3    9
[2,]    4   10    5   11    6   12
> Y[5]
[1] 2
```

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

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

```
> Z <- matrix(rnorm(16), ncol=4)
> Z
           [,1]       [,2]         [,3]        [,4]
[1,] -1.7666373  2.1353032 -0.903786375 -0.70527447
[2,] -0.9129580  1.1873620  0.002903752  0.51174408
[3,] -1.5694273 -0.5670293 -0.883259848  0.05694691
[4,]  0.9903785 -1.6138958  0.408543336  2.39152400
> diag(Z)
[1] -1.7666373  1.1873620 -0.8832598  2.3915240
> diag(5) # create the 5 x 5 identity matrix
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    1
> s <- sqrt(10:13)
> diag(s)
         [,1]     [,2]     [,3]     [,4]
[1,] 3.162278 0.000000 0.000000 0.000000
[2,] 0.000000 3.316625 0.000000 0.000000
[3,] 0.000000 0.000000 3.464102 0.000000
[4,] 0.000000 0.000000 0.000000 3.605551
```

Matrix operations in R

The standard mathematical operations of addition and subtraction and scalar multiplication work element-wise for matrices in the same way as they did for vectors. Matrix multiplication uses the operator `%*%` which you saw last week for the dot product. To get the tranpose 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)
> B <- matrix(rnorm(12), nrow=4)
```

```
> A
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> 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
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)
Error in solve.default(C) : Lapack routine dgesv: system is exactly
    singular
> C <- matrix(rnorm(16), nrow=4)
> 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)
> 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.

Matrices in Python

Matrices in Python are created are created using the `Numeric.array()` function. In Python you need to be a little more aware of the type of the arrays that you create. If the argument you pass to the `array()` function is composed only of integers than Numeric will assume you want an integer matrix which has consequences in terms of operations like those illustrated below. To make sure you're matrix has floating type values you can use the argument `typecode=Numeric.Float`.

```
>>> import numpy as np # I'm 'aliasing' the name so I can type 'np' instead
     of 'numpy'
>>> array = np.array # setup another alias
>>> X = array(range(1,13))
>>> X
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
>>> X.shape = (4,3) # rows, columns
>>> X
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
>>> 1/X # probably not what you expected
array([[1, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
>>> X = array(range(1,13), dtype=np.float)
>>> X.shape = 4,3
>>> X
array([[  1.,   2.,   3.],
       [  4.,   5.,   6.],
       [  7.,   8.,   9.],
       [ 10.,  11.,  12.]])
>>> 1/X # that's more like it
array([[ 1.        ,  0.5       ,  0.33333333],
       [ 0.25      ,  0.2       ,  0.16666667],
       [ 0.14285714,  0.125     ,  0.11111111],
       [ 0.1       ,  0.09090909,  0.08333333]])
>>> X
array([[  1.,   2.,   3.],
```

```
       [  4.,   5.,   6.],
       [  7.,   8.,   9.],
       [ 10.,  11.,  12.]])
>>> X + X
array([[  2.,   4.,   6.],
       [  8.,  10.,  12.],
       [ 14.,  16.,  18.],
       [ 20.,  22.,  24.]])
>>> X - X
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> np.dot(X,np.transpose(X)) # dot fxn in numpy gives matrix
    multiplication for arrays
array([[  14.,   32.,   50.,   68.],
       [  32.,   77.,  122.,  167.],
       [  50.,  122.,  194.,  266.],
       [  68.,  167.,  266.,  365.]])
>>> np.identity(4)
array([[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0],
       [0, 0, 0, 1]])
>>> np.sqrt(X)
array([[ 1.        ,  1.41421356,  1.73205081],
       [ 2.        ,  2.23606798,  2.44948974],
       [ 2.64575131,  2.82842712,  3.        ],
       [ 3.16227766,  3.31662479,  3.46410162]])
>>> np.cos(X)
array([[ 0.54030231, -0.41614684, -0.9899925 ],
       [-0.65364362,  0.28366219,  0.96017029],
       [ 0.75390225, -0.14550003, -0.91113026],
       [-0.83907153,  0.0044257 ,  0.84385396]])
```

The code above also demonstrated the Numpy functions `dot()`, `transpose()` and `identity()`. Note too that Numpy has a variety of functions such as `sqrt()`and `cos()` that work on an element-wise basis.

Indexing of arrays in Numpy is demonstrated below. You'll see that Python arrays support 'slicing' operations. For more on slicing and other array basics see the Numpy documentation at http://docs.scipy.org/doc/.

```
>>> X
array([[  1.,   2.,   3.],
       [  4.,   5.,   6.],
       [  7.,   8.,   9.],
       [ 10.,  11.,  12.]])
>>> X[0,0] # get the 0th row, 0th column (remember that Python sequences
    are zero-indexed!)
1.0
>>> X[3,0] # get the fourth row, 1st column
```

```
10.0
>>> X[:2,:2]  # an example of slicing, get the first two columns and rows (
    i.e. indices 0 and 1)
array([[ 1.,  2.],
       [ 4.,  5.]])
>>> X[1:,:2] # get everything after the 0th row and  the first two columns
array([[  4.,   5.],
       [  7.,   8.],
       [ 10.,  11.]])
```

To calculate matrix inverses in Python you need to import the `numpy.linalg` package.

```
>>> import numpy.linalg as la
>>> import numpy.random as ra  # for matrices with elements from random
    distributions
>>> C = ra.normal(loc=0,scale=1,size=(4,4)) # do help(ra.normal) for
    explanation of argumnets
>>> C
array([[ 0.79525679,  1.11730719, -2.19257712, -0.06289276],
       [ 0.7087366 ,  0.70574975, -1.51599336, -0.90360945],
       [-0.33845153, -0.20109722, -0.75245988, -0.56027025],
       [-0.51692665,  0.59972543,  1.55562234,  1.88639367]])
>>> Cinv = la.inv(C)
>>> np.dot(C, Cinv) # again result is approx the identity matrix due to
    floating point precision
array([[ 1.00000000e+000, -5.55111512e-017, -6.93889390e-017,  2.94902991e
    -017],
       [ 1.11022302e-016,  1.00000000e+000, -1.11022302e-016, -5.55111512e
           -017],
       [ 1.11022302e-016, -2.22044605e-016,  1.00000000e+000,  2.77555756e
           -017],
       [ 0.00000000e+000, -4.44089210e-016,  0.00000000e+000,  1.00000000e
           +000]])
>>> print np.array2string(np.dot(C,Cinv),precision=2, suppress_small=True)
[[ 1. -0.  0.  0.]
 [-0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [-0. -0. -0.  1.]]
```

3.1 Getting ready to analyze a messy data set

The data set `yeast-subnetwork-raw.txt` can be found on the class website. This data set consists of gene expression measurements for 15 genes from 173 two-color microarray experiments (see Gasch et al. 2000, Mol Biol Cell 11(12):4241–57). The genes included in this example 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).

> **Assignment 1**: 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). Extra credit: see if you can encapsulate these steps in a function.

Functions that might come in handy for the assignment above 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).

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.

Mean vector and matrix

To calculate a row vector of means, **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:

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

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:

$$D = X - M$$

Covariance matrix

The $p \times p$ covariance matrix is given by:

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

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 $V_{ii} = 1/\sqrt{S_{ii}}$.

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:

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

where $X \setminus \{x_i, x_j\}$ 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 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).

Scatter plot matrix

A scatter plot shows the relationship between two variables by plotting the observations in the variable space. A scatter of points that falls approximately along a line indicate that the variables of interested are linearly correlated, while a circular scatter indicates a lack of correlation. Other shapes in the scatter can be indicative of non-linear relationships. Scatter plots can also be useful for highlighting outliers.

A scatter plot matrix is a simply 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). The R function `pairs()` will create a scatter plot matrix.

```
> 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
```

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. There is also a package available on CRAN called `scatterplot3d` with similar functionality. I will demonstrate in class how to install packages.

```
> 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
> attach(yeast.clean) # so we can access the variables directly
> library(scatterplot3d) # assumes package is properly installed
> scatterplot3d(ASH1, RAS2, ACE2)
> scatterplot3d(ASH1, RAS2, ACE2, angle=-30)
```

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

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 are 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).

3.4 Plotting in Python

Python doesn't have any 'native' data plotting tools but there are a variety of packages that provide tools for visualizing data. The package we're going to use is called 'Matplotlib'. Matplotlib is one of the many packages that is distributed with the Enthought Python distribution. If you want to explore the full power of Matplotlib check out the example gallery and the documentation at http://matplotlib.sourceforge.net/.

Basic plots using matplotib

If you invoked the Ipython shell using the pylab option than most of the basic matplotlib functions are already available to you. If not, import them as so:

```
>>> from pylab import *
>>> import numpy as np # go ahead and import numpy as well
```

Loading data

First let's load the yeast data set:

```
>>> data = np.loadtxt('yeast-subnetwork-clean.txt',skiprows=1,usecols=range
    (1,16))
>>> data.shape   # check the dimensions of the resulting matrix
(173, 15)
```

The `skiprows` argument tells the function how many rows in the data file you want to skip. In this case we skipped only the first row which gives the variable names. The `usecols` arguments specificies which columns from the data file to use. Here we skipped the first (zeroth) column which had the names of the conditions. The usecols `loadtxt` works when there is no missing data. Use `numpy.genfromtxt` instead when there are missing values. For a full tutorial on how to use the `numpy.genfromtxt` function see http://docs.scipy.org/doc/numpy/user/basics.io.genfromtxt.html.

Histograms in Matplotlib

Matplotlib has a histogram drawing function. Here's how to use it:

```
>>> hist? # in Ipython calls the help function
>>> h = hist(data[:,0]) # plot a histogram of the first variable (column)
    in our data set
>>> clf() # clear the plot window, don't need this if you closed the plot
    window
>>> h = hist(data[:,0], bins=20) # plot histogram w/20 bins
>>> h = hist(data[:,:2])  # histograms of the first two variables
```

There's no built in density plot function, but we can create a function that will do the necessary calculations for us to create our own density plot. This uses a kernel density estimator function in the scipy library (included with EPD). Put the following code in a file called `myplots.py` somewhere on your `PYTHONPATH`:

```
# myplots.py

import numpy as np
from scipy import stats

def density_trace(x):
    kde = stats.gaussian_kde(x)
    xmin,xmax = min(x), max(x)
    xspan = xmax - xmin
    xpts = np.arange(xmin, xmax, xspan/1000.)
    ypts = kde.evaluate(xpts) # evalude the estimate at the xpts
    return xpts,ypts
```

You can then use the `density_trace` function as follows:

```
>>> import myplots
>>> h = hist(data[:,0], normed=True) # use normed=True so histogram
                            # is normalized to form a prob. density
>>> x,y = myplots.density_trace(data[:,0])
>>> plot(x,y, 'red')
```

Boxplots in Matplotlib

Box-and-whisker plots are straightforward in Matplotlib:

```
>>> b = boxplot(data[:,0])
>>> clf()
>>> b = boxplot(data[:,:5]) # boxplots of first 5 variables
```

The `boxplot` function has quite a few facilities for customizing your boxplots. For example, here's how we can create a notched box-plot using 1000 bootstrap replicates (we'll discuss the bootstrap in more detail in a later lecture) to calculate confidence intervals for the median.

```
>>> boxplot(data[:,0], notch=1, bootstrap=True)
```

See the Matplotib docs for more info.

Scatter Plots in Matplotlib

Scatter plots are also easy to create:

```
>>> s = scatter(data[:,0], data[:,1])
```

3D Plots

Recent version of Matplotlib include facilities for creating 3D plots. Here's an example of a 3D scatter plot:

```
>>> from mpl_toolkits.mplot3d import Axes3D
>>> fig = figure()
>>> ax = fig.add_subplot(111, projection = '3d')
>>> ax.scatter(data[:,0],data[:,1],data[:,2])
<mpl_toolkits.mplot3d.art3d.Patch3DCollection object at 0x1a0bbd70>
>>> ax.set_xlabel('Gene 1')
<matplotlib.text.Text object at 0x1a0ae7d0>
>>> ax.set_ylabel('Gene 2')
<matplotlib.text.Text object at 0x1a0bb2b0>
>>> ax.set_zlabel('Gene 3')
<matplotlib.text.Text object at 0x1a0bbcd0>
>>> show()
```

Retyping all those commands is tedious and error prone so let's turn it into a function. Add the following code to `myplots.py`:

```
from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D

def scatter3d(x,y,z, labels=None):
    fig = pyplot.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(x,y,z)

    if labels is not None:
        try:
            ax.set_xlabel(labels[0])
            ax.set_ylabel(labels[1])
```

```
            ax.set_zlabel(labels[2])
        except IndexError:
            print "You specificied less than 3 labels."
    return fig
```

Now reload myplots and call the scatter3d function as so:

```
>>> reload(myplots)
>>> myplots.scatter3d(data[:,0], data[:,1], data[:,2])
>>> myplots.scatter3d(data[:,0], data[:,1], data[:,2], lab)
>>> myplots.scatter3d(data[:,0], data[:,1], data[:,2],labels=('X','Y','Z'))
```

3.5 Plotting Geographic Data using Basemap

There are a number of toolkits available for Matplotlib that extend the functionality of the package. The mplot3d is one of those toolkits which has now been incorporated into the standard distribution. Basemap is another toolkit that provides the ability to plot 2D data on maps. The Basemap toolkit supports a variety of mapping projections and coordinate transformations and has the ability to plot things likes water bodies and political boundaries.

The EPD edition of Python includes Basemap but in the interest of space they have removed the high resolution maps that the normal Basemap distribution includes. In order to use those maps you can download a basemap binary (for Windows) or the source code (on OS X) from the here.

On Windows just run the executable installer (make sure you get the version that is appropriate to your EPD distribution; either 32-bit or 64-bit).

On OS X, once you have downloaded the source tarball (`basemap-1.0.1.tar.gz`), open up a bash shell, navigate to the directory where you saved the tarball, and type:

```
tar xvzf basemap-1.0.1.tar.gz
```

This will decompress and unarchive the source code into a directory called `basemap-1.0.1`. Navigate to the directory where the mapping data is stored:

```
cd basemap-1.0.1/lib/mpl_toolkits/basemap/data
```

And then copy all the `.dat` files to your Python installation:

```
cp *.dat /Library/Frameworks/Python.framework/Versions/Current/lib/python2
    .7/site-packages/mpl_toolkits/basemap/data
```

Using Basemap

In our first basemap example we show how to plot the US lower 48 and we add a red dot to represent the city of Durham, NC. Save this code as `mapex.py` and run it from the command line (`python mapex.py`).

```
# Derived from: Tosi, Sandro. Plotting Geographical Data using Basemap
# url: http://www.packtpub.com/article/plotting-geographical-data-using-
    basemap

import numpy as np
from matplotlib import pyplot
from mpl_toolkits.basemap import Basemap
```

```
# Lambert Conformal map of USA lower 48 states
m = Basemap(llcrnrlon=-119, llcrnrlat=22, urcrnrlon=-64,
  urcrnrlat=49, projection='lcc', lat_1=33, lat_2=45,
  lon_0=-95, resolution='l', area_thresh=10000)

# draw the coastlines of continental area
m.drawcoastlines()
# draw country boundaries
m.drawcountries(linewidth=2)
# draw states boundaries (America only)
m.drawstates()

# fill the background (the oceans)
m.drawmapboundary(fill_color='aqua')
# fill the continental area and lakes
m.fillcontinents(color='coral',lake_color='aqua')

# draw pt. indicating durham/raleigh area
# Durham, latitude:  35deg 52min N, longitude:78deg 47min W
dlat, dlong = 35.86, -78.78 # west is minus

# this maps latitude and longitude to map coordinates
mcoordx, mcoordy = m(dlong,dlat)
pyplot.plot(mcoordx,mcoordy, 'ro') # draw red dot
pyplot.text(mcoordx+36000, mcoordy-18000, 'Durham')

# finally show the file
pyplot.show()
```

In our second example let's assume you've been studying the population genetics of the beautiful and rare North Carolina Blue Snouter (mammals of the order Rhinogradentia; see Stümpke 1967. The snouters: form and life of the Rhinogrades). You've been sampling snouter populations from across NC and you want to make a figure for a paper showing all your sampling locations. Download the file `nc-sites.txt` from the course wiki, and place it in the same directory as the following module (`mapex2.py`).

```
# mapex2.py

import numpy as np
from matplotlib import pyplot
from mpl_toolkits.basemap import Basemap

m = Basemap(llcrnrlon=-85, llcrnrlat=33, urcrnrlon=-75,
  urcrnrlat=37, projection='lcc', lat_0=35.774, lon_0=-78.634,
  resolution='l', area_thresh=10000)

m.drawcoastlines()
m.drawcountries(linewidth=2)
m.drawstates()
m.drawmapboundary(fill_color='aqua')
```

```
m.fillcontinents(color='coral',lake_color='aqua')

sites = np.loadtxt('nc-sites.txt')

for row in sites:
    lat, lon = row[0], row[1]
    x,y = m(lon, lat) # note how longitude (x-direction) comes first
    # use blue +'s to plot sites
    pyplot.plot(x,y, 'b+', markersize=8,markeredgewidth=2)

pyplot.show()
```

The `mapex2.py` code will produce a figure like the one below.

Figure 3.1: Output of the mapex2.py module

4

Regression models in R

This weeks hands-on exercises draw from a nice set of introductory R notes written by John Verzani, entitled "simpleR – Using R for Introductory Statistics." This text used to available online at: http://www.math.csi.cuny.edu/Statistics/R/simpleR/ (the page still exists, but the PDF is not available as of Sept. 2011, but see below).

I've put up a PDF version of the Verzani text on GitHub. Also from the class wiki download the set of R functions and data that are associated with the Verzani text `simpleR.R` and put it in your R working directory.

4.1 Bivariate Linear Regression in R revisited

Recall from previous class sessions that the main function for carrying out regression in R is `lm()` (short for linear-model).

Read and work through the exercises in Verzani, section 13 (starting at p. 100 in the PDF), for a refresher on bivariate regression. As you work through the material you will notice that Verzani uses a number of functions that are defined in the code file `simple.R`, for example `simple.lm()`. Most of these are what me might call "wrapper functions" — they wrap pre-existing R functions to make them more convenient to use, for example automatically creating useful plots.

> **Assignment 1**: Open up `simple.R` in your text editor and read through the code for `simple.lm()`. Write a description of what each of the major steps in the `simple.lm()` accomplishes. If you want you can embed little snippets of code in your explanation by surrounding the code with `\begin{verbatim}...\end{verbatim}`:

```
\begin{verbatim}
x < - 1:10
y <- 10:20
z <- x + y
\end{!verbatim} # don't include the exclamation mark
```

> These verbatim snippets will not get evaluated when you compile your document to a PDF.

4.2 Multiple Regression in R

Like bivariate regression, multiple regression in R is typically conducted using the `lm()` function. Work through Verzani section 14 (p. 109 in the PDF). On p. 114 Verzani demonstrates an application of polynomial regression.

4.3 Logistic Regression in R

Logistic regression is a type of 'Generalized Linear Model' (GLM) and hence is fit using the `glm()` function in R. `glm()` can also be used to fit other types of generalized linear models so one must specify the model type as shown below. We will apply logistic regression to study natural selection on a population of house sparrows.

Bumpus' Sparrow Data

Bumpus (1898) described a sample of house sparrows which he collected after a very severe storm. The sample included 136 birds, sixty four of which perished during the storm. Also included in his description were a variety of morphological measurements on the birds and information about their sex and age (for male birds). This data set has become a benchmark in the evolutionary biology literature for demonstrating methods for analyzing natural selection. The bumpus data set is available from the class website as `bumpus-data.txt`.

Bumpus, H. C. 1898. The elimination of the unfit as illustrated by the introduced sparrow, Passer domesticus. (A fourth contribution to the study of variation.) *Biol. Lectures: Woods Hole Marine Biological Laboratory*, 209–225.

Using logistic regression to analyze the Bumpus data

We'll load the Bumpus data set and rename the variable names to shorter, more convenient ones.

```
> bumpus <- read.delim("bumpus-data.txt")
> names(bumpus)
 [1] "line.number"                "sex"                          "age..a...
     adult..y...young."
 [4] "survived"                   "total.length..mm."            "alar.
     extent..mm."
 [7] "weight..g."                 "length.of.beak.and.head"      "length.of
     .humerus..in."
[10] "length.of.femur..in."       "length.of.tibiotarsus..in."   "width.of.
    skull..in."
[13] "length.of.sternal.keel..in."
> split.names <- strsplit(names(bumpus), ".", fixed=T)
> split.names[1]
[[1]]
[1] "line"   "number"
> split.names[8]
[[1]]
[1] "length" "of"     "beak"   "and"    "head"
> first <- function(x){x[1]}
> third <- function(x){x[3]}
> new.names1 <- unlist(lapply(g[1:7],first))
> new.names2 <- unlist(lapply(g[8:13],third))
```

```
> new.names <- c(new.names1, new.names2)
> new.names
 [1] "line"        "sex"         "age"         "survived"    "total"
     "alar"
 [7] "weight"      "beak"        "humerus"     "femur"       "tibiotarsus"
     "skull"
[13] "sternal"
> names(bumpus) <- new.names
```

Having made the names more convenient we can now proceed to carrying out the logistic regression:

```
> weight <- bumpus$weight
> survived <- bumpus$survived
> plot(survived ~ weight)
> fit <- glm(formula = survived ~ weight, family=binomial(link=logit))
> summary(fit)

Call:
glm(formula = survived ~ weight, family = binomial(link = logit))

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.6331  -1.1654   0.8579   1.0791   1.4626

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)   8.0456     3.2516   2.474   0.0133 *
weight       -0.3105     0.1272  -2.441   0.0146 *
---
Signif. codes:  0 ’***’ 0.001 ’**’ 0.01 ’*’ 0.05 ’.’ 0.1 ’ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 188.07  on 135  degrees of freedom
Residual deviance: 181.58  on 134  degrees of freedom
AIC: 185.58

Number of Fisher Scoring iterations: 4
> newx <- seq(20,35,0.5)
> p <- predict.glm(fit, newdata=data.frame(weight = newx), type="response")
> lines(newx,p)
```

> **Assignment 2:** Repeat the logistic regression of survival as a function of body weight for: 1)the female birds and 2) the adult male birds. Produce plots illustrating these regressions for each sex.

LOESS Models

LOESS (aka LOWESS; ‘Locally weighted scatterplot smoothing’) is a modeling technique that fits a curve (or surface) to a set of data using a large number of local regressions. Local

weighted regressions are fit at numerous regions across the data range, using a weighting function that drops off as you move away from the center of the fitting region (hence the 'local' aspect). LOESS combines the simplicity of least squares fitting with the flexibility of non-linear techniques and doesn't require the user to specify a functional form ahead of time in order to fit the model. It does however require relatively dense sampling in order to produce robust fits.

Formally, at each point x_i we estimate the regression coefficients $\hat{\beta}_j(x)$ as the values that minimize:

$$\sum_{k=1}^{n} w_k(x_i)(y_k - \beta_0 - \beta_1 x_k - \ldots - \beta_d x_k^2)^2$$

where d is the degree of the polynomial (usually 1 or 2) and w_k is a weight function. The most common choice of weighting function is called the "tri-cube" function as is defined as:

> **Assignment 3**: Write an R function that computes the tri-cube function described above. Create a plot of the tri-cube function over the range (-3,3). Create a second R function that calculates the Gaussian function $f(x) = e^{-x^2}$ and plot that function over the same range in the same plot. If you're struggling with writing the functions you may wish to review Chapter 6 of Paradis, R for Beginners (see wiki) and Chapters 9 (Grouping, loops and conditional execution) and 10 (Writing your own functions) in the "R Introduction" included with R.

The primary parameter that a user must decide on when using LOESS is the size of the neighborhood function to apply (i.e. over what distance should the weight function drop to zero). This is referred to as the "span" in the R documentation, or as the parameter α in many of the papers that discuss LOESS. The appropriate span can be determined by experimentation or, more rigorously by cross-validation.

We'll illustrate fitting a Loess model using data on Barack Obama's approval ratings over the period from November 2008 to September 2010 (`obama-polls.txt` available on the class wiki).

```
> polls <- read.table('obama-polls.txt',header=T,sep="\t")
> names(polls)
[1] "Pollster"   "Dates"      "N.Pop"      "Approve"    "Disapprove" "
    Undecided"
> dim(polls)
[1] 854   6
# polls in reverse chronological order so let's reverse them
# so we can look at trend from earliest to most recent dates
> approve <- rev(polls$Approve)
> pollnum <- 1:length(approve)
> loess.app <- loess(approve ~ pollnum)
> pred.app <- predict(loess.app, pollnum)

# illustrate Loess with a smaller neighborhood span
> loess.app2 <- loess(approve ~ pollnum, span=0.25)
> pred.app2 <- predict(loess.app2, pollnum)
> lines(pollnum, pred.app2, lwd=2, col='cyan')
```

Take note of how the loess curve changed when we made the span smaller. By decreasing the span we've increased the sensitivity of the model (perhaps overfitting in this case).

Assignment 4: Write an R function that genereates a plot that simultaneously illustrates trends in both approval and disapproval ratings for Barack Obama, showing both the raw data and corresponding loess fits. Use colors and/or shapes to distinguish the two trends. Make sure both your x- and y-axes are scaled to show the full range of the data. Label your axes and create a title in the plot. Aim for a 'publication quality' figure.

5

Eigenanalysis and PCA

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$values)
> Vinv <- solve(V)
> V %*% D %*% Vinv  # reconstruct our original matrix (see lecture slides)
     [,1] [,2]
[1,]    2    2
[2,]    1    3
> Vinv %*% A %*% V
             [,1] [,2]
[1,] 4.000000e+00    0
[2,] 2.220446e-16    1
> all.equal(Vinv %*% A %*% V, D) # test 'near equality'
[1] TRUE
> V[,1] %*% V[,2] # note that the eigenvectors are NOT orthogonal. Why?
          [,1]
[1,] 0.3162278
> B <- matrix(c(2,2,2,3),nrow=2) # define another tranformation
> B
     [,1] [,2]
[1,]    2    2
[2,]    2    3
> eigen.B$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, **A**, point in the directions that are unchanged by the transformation specified by **A**. The following relationships relate **A** to it's eigenvectors and eigenvalues:

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

$$\mathbf{A} = \mathbf{VDV}^{-1}$$

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"))
```

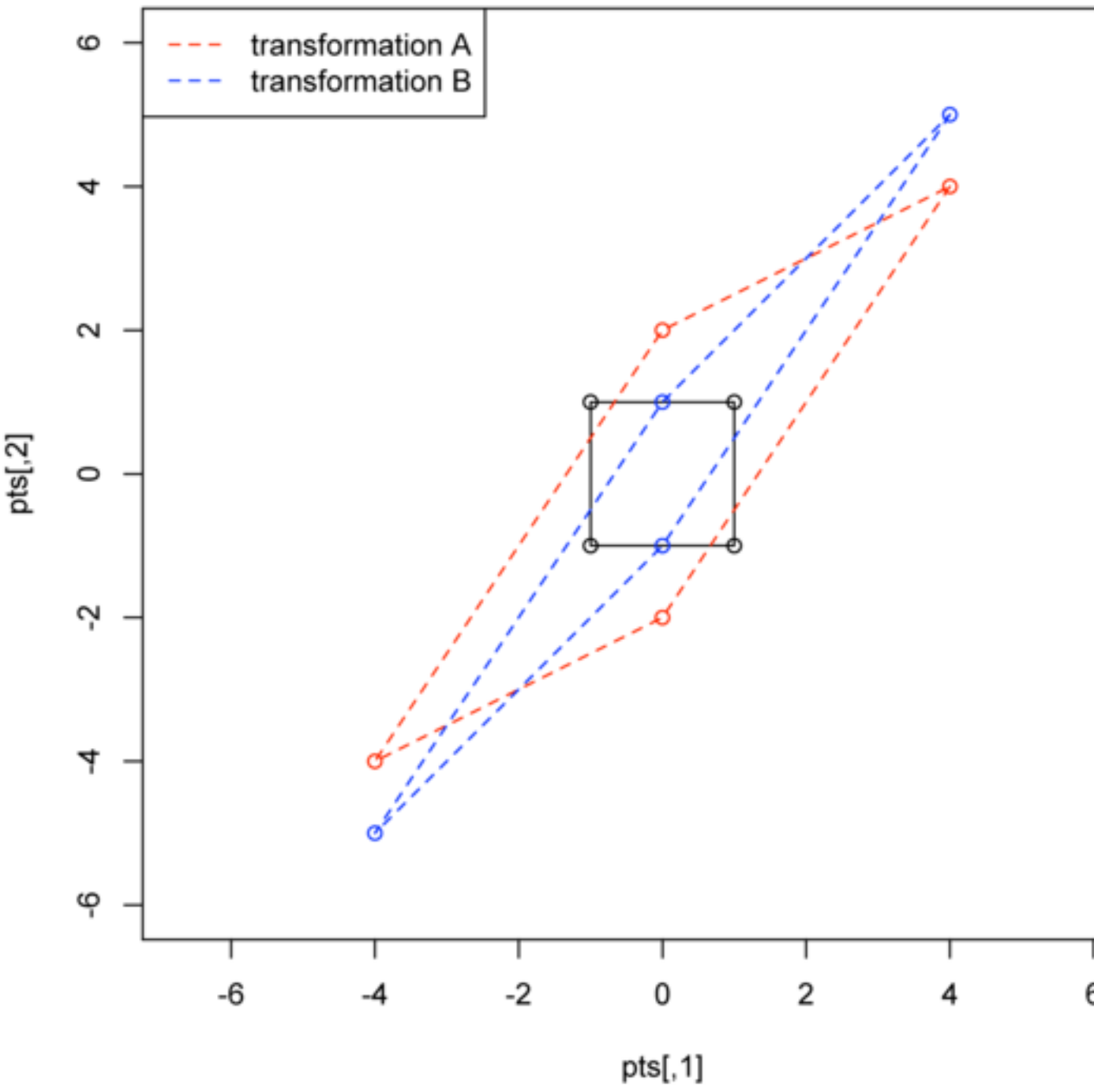

Figure 5.1: Transformation of a square represented by two matrices, A and B

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

> **Extra Credit**: write R code to reconstruct the plot shown in the figure below (Fig. 5.2) which illustrates the geometry of the eigenvectors for matrices A and B. Note that the lengths of the eigenvector depictions are scaled to be proportional to their eigenvalues.

5.2 EIGENANALYSIS IN PYTHON

The eigenanalysis functions in Python are found in the `linalg` module in the `Numpy` package. These functions are specified by `linalg.eig()` (which returns both eigenvalues and eigenvectors) and `linalg.eigvals()` (which only returns the eigenvalues).

```
>>> import numpy as np, numpy.linalg as la
>>> A = np.array([[2,2],[1,3]],np.float)
>>> A
array([[ 2.,  2.],
       [ 1.,  3.]])
>>> evals, evecs = la.eig(A)
>>> evals
array([ 1.,  4.])
>>> evecs
array([[-0.89442719, -0.70710678],
       [ 0.4472136 , -0.70710678]])
```

Note that (somewhat inconveniently) the `eig()` function does not necessarily return the eigenvalues and eigenvectors in sorted fashion. The Numpy documentation states that the normalized eigenvector corresponding to the eigenvalue `w[i]` is the column `v[:,i]`. Also note that both the R `eigen()` function and the Numpy `eig()` function return normalized eigenvectors (i.e. each eigenvector has length 1).

We can get sort the eigenvectors by their eigenvalues as so:

```
>>> colorder = list(np.argsort(evals)) # get back argsort as a list
>>> colorder
[0, 1]
>>> colorder.reverse() # need to reverse the column order because want from
     large to small
>>> colorder
[1, 0]
>>> srtevals = np.take(evals, colorder) #see the Numpy docs on take()
>>> srtevals
array([ 4.,  1.])
>>> srtevecs = np.take(evecs, colorder,axis=1) # sort columns
>>> srtevecs
array([[-0.70710678, -0.89442719],
       [-0.70710678,  0.4472136 ]])
>>> V = srtevecs
>>> Vinv = la.inv(srtevecs)
>>> D = np.diag(srtevals)
>>> np.dot(V, np.dot(D, Vinv))
```

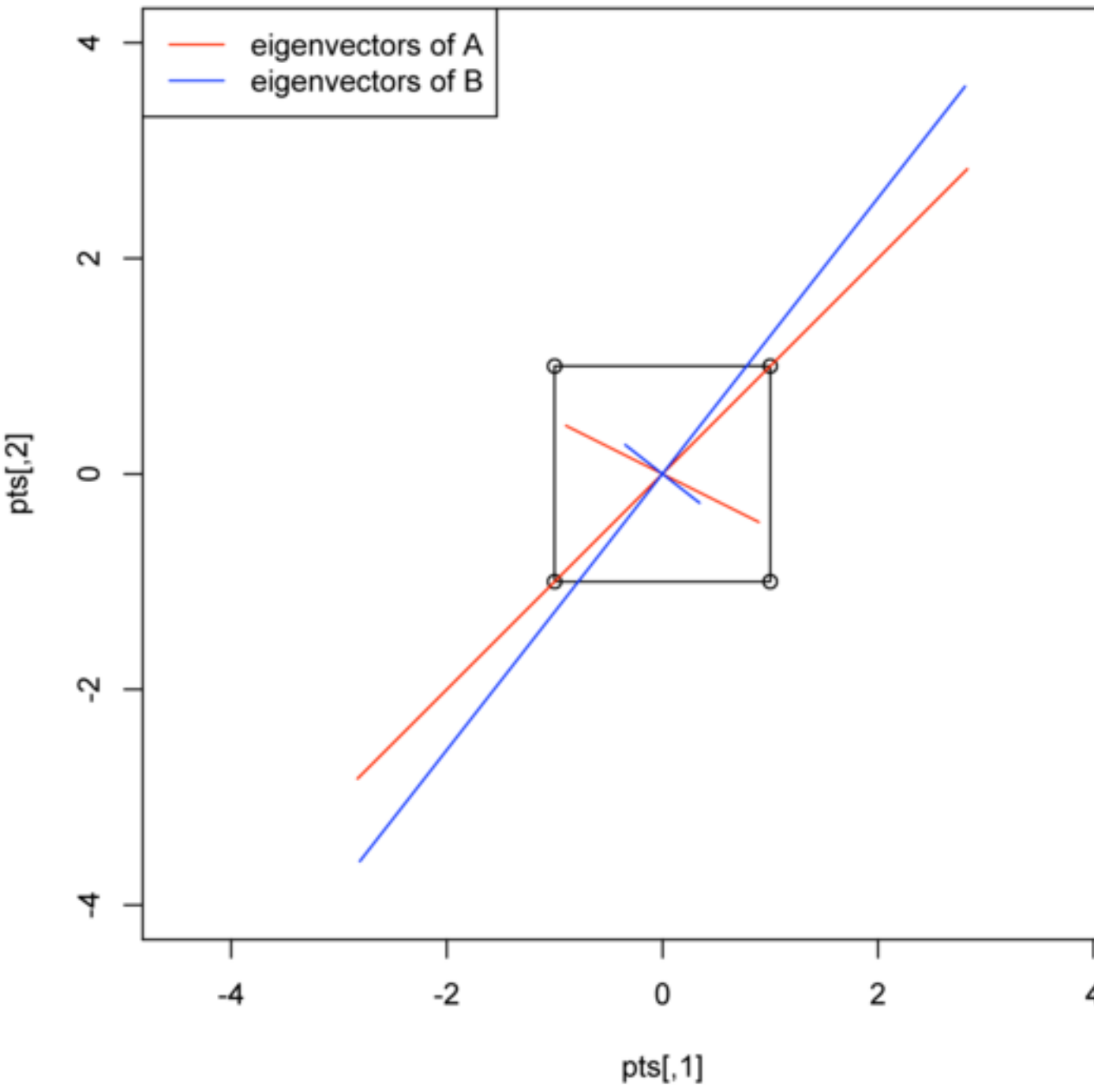

Figure 5.2: Eigenvectors of matrices A and B

```
array([[ 2.,  2.],
       [ 1.,  3.]])
>>> Arecon = np.dot(V, np.dot(D, Vinv))
>>> Arecon == A
array([[ True,  True],
       [ True, False]], dtype=bool)
>>> np.allclose(Arecon, A)  # like R's all.equal()
True
```

See the Numpy docs for more information on the `argsort()` (Numpy: Sorting and searching), `take()` ((Numpy: Indexing Routines), and `allclose()` (Numpy: Logic functions).

> **Assignment 1**: Write a Python function that takes as input a square matrix, and returns a vector of sorted eigenvalues (sorted from largest to smallest) and the corresponding matrix of eigenvectors sorted according to their corresponding eigenvalues.

5.3 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 it should generally be preferred but we'll delay it's use until next week when we introduce the mathematics behind SVD. 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). By default the `princomp()` function carries out PCA on the covariance matrix of the data. If you wish to carry out PCA using the correlation matrix specify the argument `cor=T`.

To demonstrate PCA we'll use Anderson's Iris data set.

```
> names(iris)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
> iris
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1            5.1         3.5          1.4         0.2     setosa
2            4.9         3.0          1.4         0.2     setosa
3            4.7         3.2          1.3         0.2     setosa
.. output truncated ..
> summary(iris) # notice that  Petal.Width is
        # about an order of magnitude smaller than the other variables.
        # We should therefore consider doing the PCA on the correlation
        # matrix rather than the cov matrix. We'll try both
> setosa <- subset(iris, Species=='setosa', select=-Species)
> pca.setosa <- princomp(setosa)
> summary(pca.setosa)
Importance of components:
                          Comp.1    Comp.2    Comp.3     Comp.4
Standard deviation     0.4813799 0.1902114 0.1620508 0.09408823
Proportion of Variance 0.7647237 0.1193992 0.0866625 0.02921456
```

```
Cumulative Proportion  0.7647237 0.8841229 0.9707854 1.00000000
> plot(pca.setosa) # plots histogram plot of eigenvalues
> plot(pca.setosa$scores)  # in space of PC1 and PC2
> plot(pca.setosa$scores, asp=1)  # compare to the previous plot
> plot(pca.setosa$scores[,1],pca.setosa$scores[,3], asp=1) # PC1 and PC3
> pca.setosa$loadings

Loadings:
             Comp.1 Comp.2 Comp.3 Comp.4
Sepal.Length -0.669 -0.598  0.440
Sepal.Width  -0.734  0.621 -0.275
Petal.Length        -0.490 -0.832 -0.240
Petal.Width         -0.131 -0.195  0.970

               Comp.1 Comp.2 Comp.3 Comp.4
SS loadings      1.00   1.00   1.00   1.00
Proportion Var   0.25   0.25   0.25   0.25
Cumulative Var   0.25   0.50   0.75   1.00

# now do the PCA based on the correlation matrix
> pca.setosa.cor <- princomp(setosa,cor=T)
> summary(pca.setosa.cor)
Importance of components:
                          Comp.1    Comp.2    Comp.3     Comp.4
Standard deviation     1.4347614 1.0110283 0.8172027 0.50145917
Proportion of Variance 0.5146351 0.2555446 0.1669551 0.06286532
Cumulative Proportion  0.5146351 0.7701796 0.9371347 1.00000000
> pca.setosa.cor$loadings

Loadings:
             Comp.1 Comp.2 Comp.3 Comp.4
Sepal.Length  0.604  0.335         0.720
Sepal.Width   0.576  0.441        -0.689
Petal.Length  0.375 -0.627 -0.677
Petal.Width   0.403 -0.548  0.733

               Comp.1 Comp.2 Comp.3 Comp.4
SS loadings      1.00   1.00   1.00   1.00
Proportion Var   0.25   0.25   0.25   0.25
Cumulative Var   0.25   0.50   0.75   1.00
> plot(pca.setosa.cor)
> plot(pca.setosa.cor$scores,asp=1)
```

Assignment 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 the figure below. Represent the specimens from a given species with different colors. Make sure you include a legend for your plot.

Apply PCA (on the covariances) to the `yeast-subnetwork` data set from week three. Now do the same analysis based on PCA of the correlation matrix. How

does your interpretation of the data change?

5.4 Principal Components Analysis in Python

There are no built-in functions for carrying out PCA in Python. Luckily you have all the tools you need at your disposal to write your own PCA module.

Note that when performing PCA on a correlation matrix, to get the proper PC scores you need to use mean centered and standardized variates.

Here's an example:

```
>>> import numpy as np, numpy.linalg as la
>>> turt = np.loadtxt('turtles.txt',skiprows=1,usecols=(1,2,3)) # see the
    Numpy docs
>>> turt.shape
(48, 3)
>>> import pylab # convenient interface to Matplotlib
>>> pylab.scatter(turt[:,0], turt[:,1])
<matplotlib.collections.CircleCollection object at 0x2502630>
>>> pylab.show()
>>> pylab.scatter(turt[:,0], turt[:,2])
<matplotlib.collections.CircleCollection object at 0x66d4a30>
>>> pylab.scatter(turt[:,1], turt[:,2])
<matplotlib.collections.CircleCollection object at 0x63296b0>
>>> tmean = np.mean(turt,axis=0) # get the column means
>>> tmean
array([ 124.6875    ,   95.4375    ,   46.33333333])
>>> deviates = turt - tmean # this mean centers each observation
>>> stdized = deviates/np.std(deviates,axis=0) # standardize variates
>>> pylab.scatter(stdized[:,0],stdized[:,1],color='red') # plot the
    standardized observations
>>> tcor = np.corrcoef(turt,rowvar=False)
>>> tcor
array([[ 1.        ,  0.97831162,  0.96469455],
       [ 0.97831162,  1.        ,  0.96057053],
       [ 0.96469455,  0.96057053,  1.        ]])
>>> u,v = la.eig(tcor)
>>> u # the eigenvalues
array([ 2.93573765,  0.02141848,  0.04284387])
>>> v
array([[-0.57879812, -0.74789704, -0.32502731],
       [-0.57798399,  0.65741263, -0.48346989],
       [-0.57526276,  0.09197088,  0.81278171]])
>>> colsort = np.argsort(u)[::-1] # fancy way  to do the
                 # argsort and reversing to sort index in one call
>>> colsort
array([0, 2, 1])
>>> usort = np.take(u,colsort)
>>> vsort = np.take(v,colsort,axis=1)
>>> usort
array([ 2.93573765,  0.04284387,  0.02141848])
```

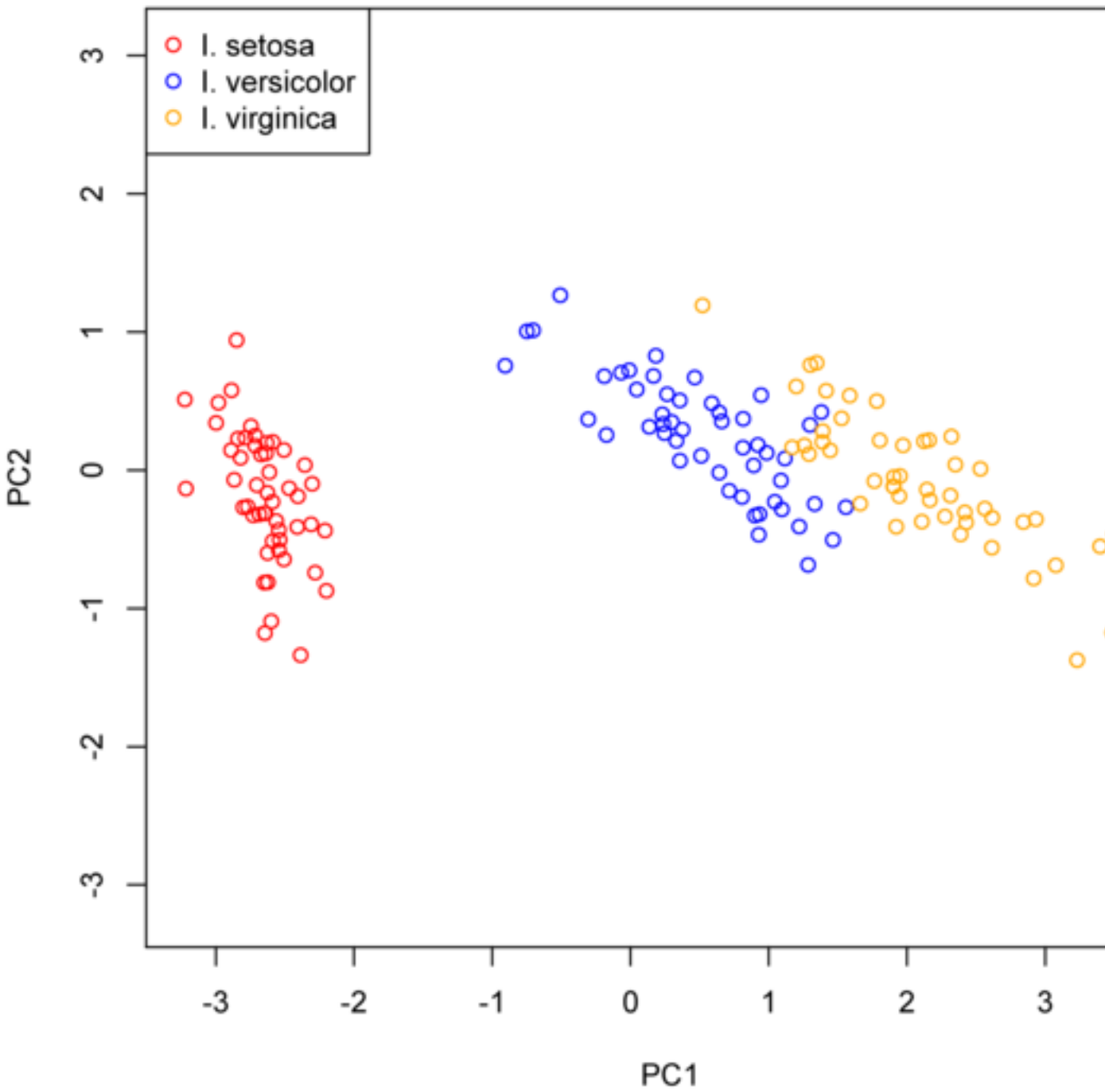

Figure 5.3: PCA of the iris data set. One of your assignments is to reconstruct this figure on your own.

```
>>> vsort # the PC coefficents are the normalized eigenvectors
array([[-0.57879812, -0.32502731, -0.74789704],
       [-0.57798399, -0.48346989,  0.65741263],
       [-0.57526276,  0.81278171,  0.09197088]])
>>> L = np.diag(usort**0.5) # mtx with sqrt of eigenvalues on diagonal
>>> L
array([[ 1.71339944,  0.        ,  0.        ],
       [ 0.        ,  0.2069876 ,  0.        ],
       [ 0.        ,  0.        ,  0.14635054]])
>>> f=np.dot(vsort,L) # "factor loadings" -- gives corr. btw original vars
    and PC axes
>>> f
array([[-0.99171237, -0.06727662, -0.10945514],
       [-0.99031745, -0.10007227,  0.09621269],
       [-0.98565489,  0.16823574,  0.01345999]])
>>> scores = np.dot(stdized,vsort)
>>> scores[:5]  # compare to the values you got in R
array([[ 2.00466025,  0.16892135,  0.13583391],
       [ 1.72362844, -0.02689852,  0.10856089],
       [ 1.35439901,  0.2874806 ,  0.25768234],
       [ 1.43581697,  0.05967203,  0.16172995],
       [ 0.95235368,  0.30990249,  0.16324351]])
>>> pylab.scatter(scores[:,0],scores[:,1]) # don't close the plot
                                          # or else the next line won't work
<matplotlib.collections.CircleCollection object at 0x66d5f50>
>>> pylab.axes().set_aspect(1) # equivalent of the asp=1 argument in R
>>> pylab.xlim(-5,3)  # to make the plot limits more like those we saw in R
(-5, 3)
>>> pylab.ylim(-3,3)
(-3, 3)
```

Assignment 3: Write a Python function that takes as it's input an $n \times p$ data matrix and returns the following four objects (*in this order*):

1. A vector of the **eigenvalues of the correlation matrix** sorted from largest to smallest
2. A corresponding matrix of **eigenvectors of the correlation matrix** sorted relative to their eigenvalues
3. The **principal component scores** for the dataset
4. An array giving the percentage of variation explained by each principal component.

Apply this to the `yeast-subnetwork` data set and check your answers against the R implementation.

6

Singular value decomposition

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
```

```
> a.svd$d  # R uses the notation A = u d v' rather than A = u s v'
```

```
[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, **A**, point in the directions that are unchanged by the transformation specified by **A**.

Creating Biplots in R

To illustrate the construction of biplots we'll use the same iris data set we used last week to introduce PCA. The built-in R function is `biplot()`. We'll also use the `prcomp()` function to do PCA rather than `princomp()`. `prcomp()` uses SVD of the mean-centered data matrix to do the PCA.

```
> iris.vars <- subset(iris, select=-Species) # leave out the Species
    variable
> # ?prcomp # read the docs on prcomp and how it differs 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
```

```
> iris.pca$rotation  # equivalent to 'loadings' of princomp
```

```
                     PC1         PC2         PC3        PC4
```

```
Sepal.Length  0.36138659 -0.65658877  0.58202985  0.3154872
Sepal.Width  -0.08452251 -0.73016143 -0.59791083 -0.3197231
Petal.Length  0.85667061  0.17337266 -0.07623608 -0.4798390
Petal.Width   0.35828920  0.07548102 -0.54583143  0.7536574
```

```
> plot(iris.pca$x)  # 'x' is equivalent to 'scores' of princomp
> biplot(iris.pca, scale=1)
> biplot(iris.pca, scale=1, cex=c(0.75,1)) # plot the scores(rows) with
    slightly smaller font size
> biplot(iris.pca, scale=0, cex=c(0.75,1)) # change the biplot scaling -
    how does this differ?
```

Assignment 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 is most closely related to the vector representing MEP2?

SVD in Python

Like the `eig()` function, the `svd()` function is found in the `numpy.linalg` module.

```
>>> import numpy as np, numpy.linalg as la
>>> A = np.array([[2,2],[1,3]])
>>> A
array([[ 2.,  2.],
       [ 1.,  3.]])
>>> U,S,Vt = la.svd(A)
>>> U
array([[-0.66180256, -0.74967818],
       [-0.74967818,  0.66180256]])
>>> S
array([ 4.13064859,  0.96837093])
>>> Vt
array([[-0.50192682, -0.86491009],
       [-0.86491009,  0.50192682]])
>>> diagS = np.identity(len(S)) * S
>>> diagS
array([[ 4.13064859,  0.        ],
       [ 0.        ,  0.96837093]])
>>> svdA = np.dot(U, np.dot(diagS, Vt))
>>> svdA
array([[ 2.,  2.],
       [ 1.,  3.]])
>>> np.allclose(A, svdA)
True
>>> AtA = np.dot(transpose(A),A)
```

```
>>> eu, ev = la.eig(AtA)
>>> eu
array([  0.93774225,  17.06225775]) # eigenvalues not sorted
>>> S**2
array([ 17.06225775,   0.93774225])
```

'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 microarray 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()
```

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.

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.

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 http://www.pythonware.com/products/pil/ or pre-built binaries for OS X can be found at http://www.pythonmac.org/packages/. 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 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 called as `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.

7

Discriminant Analysis

Discriminant Analysis in R

The function `lda()`, found in the R library `MASS`, carries out linear discriminant analysis (i.e. canonical variates analysis).

```
> library(MASS) #load the MASS package
> z <- lda(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.
    Width,
+           iris, prior=c(1,1,1)/3)
> z
Call:
lda(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width,
    data = iris, prior = c(1, 1, 1)/3)

Prior probabilities of groups:
    setosa versicolor  virginica
 0.3333333  0.3333333  0.3333333

Group means:
           Sepal.Length Sepal.Width Petal.Length Petal.Width
setosa            5.006       3.428        1.462       0.246
versicolor        5.936       2.770        4.260       1.326
virginica         6.588       2.974        5.552       2.026

Coefficients of linear discriminants:
                     LD1         LD2
Sepal.Length  0.8293776  0.02410215
Sepal.Width   1.5344731  2.16452123
Petal.Length -2.2012117 -0.93192121
Petal.Width  -2.8104603  2.83918785

Proportion of trace:
   LD1    LD2
0.9912 0.0088
```

The `prior` argument given in the `lda()` function call isn't strictly necessary because by default the `lda` function will assign equal probabilities among the groups. However I included this argument call to illustrate how to change the prior if you wanted. The output give some simple summary statistics for the group means for each of the variables and then

gives the coefficients of the canonical variates. The 'Proportion of trace' output above tells us that 99.12% of the between-group variance is captured along the first discriminant axis.

Shorthand Formulae in R

You've encountered the use of model formulae in R several times, such as in the call to `lda()` above and when carrying out various regressions. The document "An Introduction to R" (distributed with R and available at the R project website) gives a concise summary and a number of examples of how to construct formulae in R (see Defining statistical models: formulae).

Relevant to our current example is a shorthand way for specifying multiple variables in a formula. In the example above we called the `lda()` function with a formula of the form:

```
Species ~ Sepal.Length + Sepal.Width + ....
```

Writing the names of all those variables is tedious and error prone and would be unmanageable if we were analyzing a data set with tens or hundreds of variables. Luckily we can use the shorthand name '.' to specify all other variables in the data frame except the variable on the left. For example, we can rewrite the `lda()` call above as:

```
> z <- lda(Species ~ ., data = iris, prior = c(1,1,1)/3)
```

Fine Tuning Your Plot

To get a graphical representation of the specimens in the space of the canonical variates you can use the `plot()` function on the object returned by the call to `lda()`.

```
> plot(z) # 2D scatter plot of specimens in CVs 1 and 2
> plot(z, abbrev=T) # use abbreviated group names
```

You can also create a plot to look at group variation along just the first canonical variate:

```
> plot(z, dimen=1,type='both') # plot histograms and density plots for each
     group along 1st CV
```

The plot call on the object returned by `lda()` allows some additional customization of the plot, but the extent of graphical tuning is limited:

```
> plot(z, abbrev=T, xlab='CV1', ylab='CV2') # change the x- and y-axis
    labels
```

If you want to do any more fine tuning of the plot you'll have to calculate the CV scores from the coefficients and reconstruct the plot to your liking. Below I give an example of how to do that:

```
> iris.data <- subset(iris,select=-Species)
> iris.mtx <- as.matrix(iris.data)
> dim(iris.mtx)
[1] 150   4
> iris.cv <- iris.mtx %*% z$scaling # gives scores in the CV space
> dim(iris.cv)
[1] 150   2
> group.symbols <- (1:3)[iris$Species] # specify the symbols to use for
    each group
> plot(iris.cv, pch=group.symbols, asp=1, xlab="CV1", ylab="CV2")
```

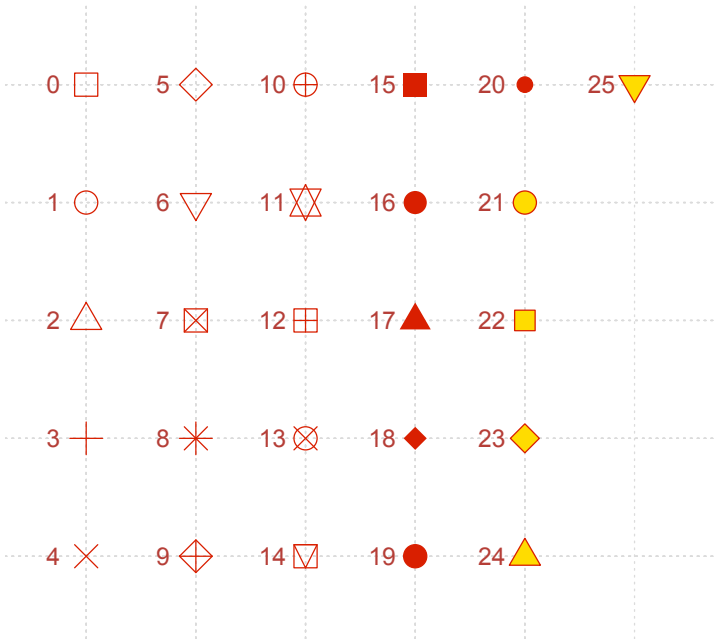

Figure 7.1: Standard R symbols, and their corresponding integer values, accessible via the `pch` argument to plot.

The definition of `group.symbols` and the use of the `pch` argument require a little explanation. `pch` is short-hand for 'plotting character' and specifies the symbols used to represent each observation in the plot. These symbols can either be letters or integers in the range 0-25. The integers refer to a standard set of symbols shapes defined in R. Figure 1 gives those symbols.

If you'd like to see a function that prints out all the standard symbols type `?points` and check out the `pchShow` function defined in the example at the bottom of the documentation page. To see this example in action type `example(points)`. After typing `example(points)` you can call the `pchShow` function directly (that's how I generated the figure).

The `group.symbols <- ...` line constructs a vector of length n (where n is the length of the `iris$Species` vector) where each element of the `group.symbols` vector has the value 1, 2 or 3 according to which species the corresponding specimen represents. A simpler example might make this clearer:

```
> sexes = as.factor(c('M','F','F','M','F'))
> sexes
[1] M F F M F
Levels: F M
> c("a","b")[sexes]
[1] "b" "a" "a" "b" "a"
```

Here I created a simple example involving five specimens where each specimens was categorized by sex. The `as.factor` function tells R to treat the characters in the vector as factor levels. I then assigned each specimen a label, either“a" or “b" depending on its sex. If I wanted to extend that example to our three species iris data set I could do something like:

```
> group.symbols = c("a","b","c")[iris$Species]
> plot(iris.cv, pch=group.symbols, cex = 0.75, asp=1, xlab="CV1", ylab="CV2
    ")
```

This draws each specimen with the label “a", “b", or “c" depending on which species it is assigned to. Notice that in the last example I used the `cex` argument to make the symbols smaller than normal.

What if i wanted to also plot the group means in the canonical variate space? The following example shows how to do that:

```
> group.symbols = c(0,2,4)[iris$Species] # I decided to go back to plotting
     symbols
> group.colors = c('red','darkorange','blue')[iris$Species] # I also want
    to use colors
> cv1.means <- tapply(iris.cv[,1], iris$Species, mean)
> cv1.means
    setosa versicolor  virginica
  5.502493  -3.930156  -7.887657
> cv2.means <- tapply(iris.cv[,2], iris$Species, mean)
> cv2.means
    setosa versicolor  virginica
  6.876606   5.933573   7.174239
> plot(iris.cv, pch=group.symbols, cex = 0.75, asp=1, xlab="CV1", ylab="CV2
    ", col=group.colors)
> points(cv1.means, cv2.means, pch=16, cex=1.5, col='black')
```

Note the use of the `points()` function. This function draws on top of rather than erasing the previous plot. Note too the use of the `col` argument in the `plot()` call to specify different colors. If you'd like to see a chart of all the colors in R check out this web page: A Chart of R Colors.

I stated in lecture that for the canonical variate diagram we can estimate the $100(1-\alpha)$ confidence region for a group mean as a circle centered at the mean having a radius $(\chi^2_{\alpha,r}/n_i)^{1/2}$ where r is the number of canonical variate dimensions considered. Using similar reasoning the $100(1-\alpha)$ confidence region for the whole population is given by a hypersphere centered at the mean with radius $(\chi^2_{\alpha,r})^{1/2}$. To calculate these confidence regions you could look up the appropriate value of the the χ^2 distribution in a book of statistical tables, or we can use the `qchisq()` function which gives the inverse cumulative probability distribution for the χ^2 function:

```
> chi2 = qchisq(0.05,2, lower.tail=F)
> chi2
[1] 5.991465
> group.lengths = tapply(iris$Species, iris$Species, length)
> group.lengths
    setosa versicolor  virginica
        50         50         50
> mean.radii = sqrt(chi2/group.lengths)
```

```
> pop.radii = rep(sqrt(chi2),3)
> help.search("circle")  # I don't remember off hand how to draw circles so
     let's look it up
> library(tripack) # I decided to use the circles function in the 'tripack'
     package
> circles(cv1.means, cv2.means, pop.radii,lty='dashed')
> circles(cv1.means, cv2.means, mean.radii,lty='dotted')
```

Let's put the finishing touch on our plots by adding some color coded rug plots to the first CV axis. For completeness I'll include all the previous steps used to generate the plot:

```
> plot(iris.cv, pch=group.symbols, cex = 0.75, asp=1, xlab="CV1", ylab="CV2
    ", col=group.colors)
> points(cv1.means, cv2.means, pch=16, cex=1.5,col='red')
> circles(cv1.means, cv2.means, pop.radii,lty='dashed')
> circles(cv1.means, cv2.means, mean.radii, lty='dotted')
> rug(iris.cv[,1][iris$Species=="setosa"],col="red")
> rug(iris.cv[,1][iris$Species=="versicolor"],col="darkorange")
> rug(iris.cv[,1][iris$Species=="virginica"],col="blue")
> title("Canonical Variates Analysis\nof Anderson's Iris Data")
```

If you did everything right (and I cut and pasted correctly!) you should get a plot that looks like Figure 2.

If I was going to be repeatedly generate these types of plots I would wrap up the key steps discussed above into a convenient function.

Calculating the Within and Between Group Covariance Matrices

The `lda()` function conveniently carries out the key steps of a canonical variates analysis for you. However, what if we wanted some of the intermediate matrices relevant to the analysis such as the within- and between group covariances matrices? The code below shows you how to calculate these:

```
> g = iris$Species
> group.means <- rowsum(iris.mtx, g)/as.vector(table(g))
> group.means
           Sepal.Length Sepal.Width Petal.Length Petal.Width
setosa            5.006       3.428        1.462       0.246
versicolor        5.936       2.770        4.260       1.326
virginica         6.588       2.974        5.552       2.026
> Dwin <- iris.mtx - group.means[g,]
> nobs <- dim(iris.mtx)[1]
> ngroups <- length(levels(g))
> win.cov <- 1/(nobs-ngroups) * t(Dwin) %*% Dwin
> btw.cov.unweighted <- cov(group.means)
```

Having now calculated the within group covariance matrix we can calculate the Mahalanobis distance between the means of each group as follows:

```
> mahalanobis(group.means, group.means[1,], win.cov)
    setosa versicolor  virginica
   0.00000   89.86419  179.38471
> mahalanobis(group.means, group.means[2,], win.cov)
    setosa versicolor  virginica
```

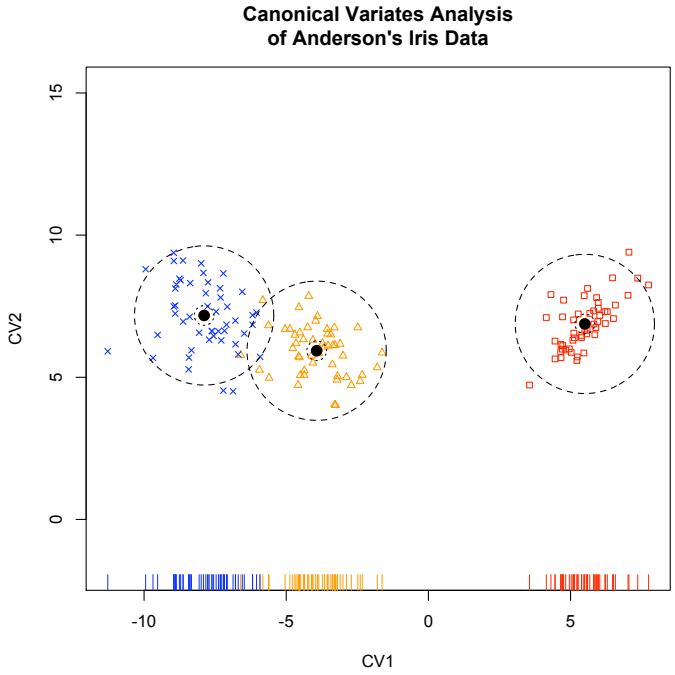

Figure 7.2: Ordination of iris specimens in the space of the first two canonical variates. The dashed circles surrounding each species distribution give the approximate 95% tolerance regions for the population distributions. See text for details on the construction of this plot.

```
  89.86419    0.00000   17.20107
> mahalanobis(group.means, group.means[3,], win.cov)
    setosa versicolor  virginica
 179.38471   17.20107    0.00000
```

A Literature Based Assignment

Assignment:

Identify a paper from the literature (published in the last decade) that employs one or more of the following multivariate statistical techniques:

1. Multivariate regression
2. Principal Component Analysis
3. Singular Value Decomposition
4. Canonical Variate Analysis (or an alternate discriminant function)

Write a short report discussing the use of these techniques in the paper and how the application of these methods contributed to the author's conclusions or understanding of the data. Your report should touch on any assumptions (explicit or implicit) that are relevant to the statistical analysis and discuss whether you feel the author's conclusions are justified or well supported (again based on the statistical anlaysis).

Include in your report a brief outline (bullet points) that lays out the key steps (e.g. handling of missing data, normalization) and the primary R functions that you would use to repeat the analysis yourself. Specify whether the author(s) have made their multivariate data set available.

8

Hierarchical clustering, minimum spanning trees, and multidimensional scaling

Hierarchical Clustering in R

The function `hclust()` provides a simple mechanism for carrying out standard hierarchical clustering in R. The `method` argument determines the group distance function used (single linkage, complete linkage, average, etc.).

The input to `hclust()` is a dissimilarity matrix. The function `dist()` provides some of the basic dissimilarity measures (e.g. Euclidean, Manhattan, Canberra; see method argument of `dist()`) but you can convert an arbitrary square matrix to a distance object by applying the `as.dist()` function to the matrix.

```
> iris.data <- subset(iris, select=-Species)
> iris.cl <- hclust(dist(iris.data), method='single')
> plot(iris.cl) # plot a dendrogram
# let's improve the look a little bit
> plot(iris.cl, labels=iris$Species, cex=0.7)
> # use neg. values of hang to make labels on leaves line up
> plot(iris.cl, labels=iris$Species,  hang=-0.1, cex=0.7)
```

Other functions of interested related to dendrograms include `cuttree()` for cutting the tree at a specified height (or number of groups) and `identify()` for graphically highlighting a cluster of interest in a dendrogram.

```
> plot(iris.cl, labels=iris$Species, cex=0.7)
> interesting.cluster <- identify(iris.cl) # use left-mouse to choose,
    right-mouse to stop choosing
> interesting.cluster
# [output ommitted]
```

Fancy formatting of dendrogram plots in R is awkward. You need to use the `plot()` function in combination with the `as.dendrogram()` function to access many options. See the help for 'dendrogram' in R for a discussion of options and type `example(dendrogram)` to set some possibilities. A few of them are illustrated here:

```
> plot(as.dendrogram(iris.cl)) # contrast this with plot(iris.cl)
> plot(as.dendrogram(iris.cl), horiz=T) # draw horizontally
> # here's one way to change the labels
> iris.cl$labels <- iris$Species
> levels(iris.cl$labels) <- factor(c("S","Ve","Vi"))
```

```
> iris.dend <- as.dendrogram(iris.cl)
> plot(iris.dend)
```

The `heatmap()` function combines a false color image of a matrix with a dendrogram. Here's we apply it to the yeast-subnetwork data set from previous weeks.

```
> yeast <- read.delim('yeast-subnetwork-clean.txt')
> ymap <- heatmap(as.matrix(yeast), labRow=NA) # suppress the numerous row
    labels
> ymap <- heatmap(as.matrix(yeast),labRow=rownames(yeast)) # w/row labels,
    kinda messy
```

The R package `cluster` provides some slightly fancier clustering routines. The basic agglomerative clustering methods in `cluster` are accessed via the function `agnes()`

Compare the results of different hierarchical clustering methods (single linkage, complete linkage, etc.) as applied to the iris data set using the hclust() or agnes() functions. For single and average linkage use both Euclidean and Manhattan distance as the dissimilarity measures.

Neighbor joining in R

The package `ape` provides an implementation of neighbor joining in R (and many other useful phylogenetic methods). Here's a couple of examples of using neighbor joining taken from the `ape` documentation:

```
library(ape) # install ape if need be
### From Saitou and Nei (1987, Table 1):
x <- c(7, 8, 11, 13, 16, 13, 17, 5, 8, 10, 13, 10, 14, 5, 7, 10, 7, 11, 8,
    11, 8, 12, 5, 6, 10, 9, 13, 8)
M <- matrix(0, 8, 8)
# create a symmetric matrix by filling upper and lower triangles
# of the matrix M
M[row(M) > col(M)] <- x
M[row(M) < col(M)] <- x
rownames(M) <- colnames(M) <- 1:8
tree <- nj(M)
plot(tree, "u")

### a less theoretical example
?woodmouse  # check out the info about the
            # woodmouse data set in the ape package
data(woodmouse)
dist <- dist.dna(woodmouse) # see the help on the dist.dna fxn
tree.mouse <- nj(dist)
plot(tree.mouse)
```

Multidimensional scaling in R

Metric MDS

The implementation of classic metric scaling in R is carried out using the `cmdscale()` function. Read the documentation for `cmdscale` and then work through the example

showing the application of MDS to analysis of road distances between US cities available at the following link (but see notes below first):

http://personality-project.org/r/mds.html.

As you work through your example note the following:

- You can use the `source()` function not only with a local file but also with a URL. This is convenient but potentially a security issue so don't run code willy nilly without checking out what it does.

- You can download the code at http://personality-project.org/r/useful.r and check out the functions that it includes. I thought the `read.clipboard()` function was particularly nice.

Minimum Spanning Tree in R

The package `ape` has an `mst()` function. Several others packages, including vegan also have minimum spanning tree functions. The `mst()` function takes a dissimilarity matrix as its input and returns a square adjacency matrix, A, where $A_{ij} = 1$ if (i, j) is an edge in the MST or 0 otherwise.

Here's an application of the MST function to the cities example you completed above.

```
> library(ape) # install ape first if necessary
> city.mst <- mst(as.dist(cities))
> city.mst # see the adjacency matrix return by mst
```

If you want to create a nice looking plot you can use the `mat2listw()` function in the package `spdep`. `mat2listw` converts the adjacency matrix into a form that you can extract the neighbor information from:

```
> library(spdep) # install spdep first if necessary
> plot(city.location, type='n', xlab='PCoord1', ylab='PCoord2')
> text(city.location, labels=names(cities))

# note British spelling of 'neighbours'
> plot(mat2listw(city.mst)$neighbours, city.location, add=T)
```

Non-metric MDS

The `isoMDS()` function in the `MASS` package implements the Shepard-Krusal version of non-metric scaling, while the `sammon()` function in the same package use the criterion proposed by Sammon (1969). You will need to utilize these functions, along with `cmdscale` and the hierarchical clustering functions covered last week for the following assignment.

Assignment:

Harding and Sokal (1998; PNAS 85:9370-9372; see course wiki) used cluster analysis and non-metric MDS to explore the relationship between European language families as measured by genetic distances among the people who speak those languages. The classification they derived at large reflects geographic proximity but there are some language families that have distant genetic relationships to their geographic neighbors.

Harding and Sokal provide a table of genetic distances that they used in their analyses. Use R to reconstruct the cluster analysis they report (Fig. 1) and repeat this analysis using neighbor joining. In a similar manner use both metric scaling and the Shepard-Kruskal and Sammon criteria for non-metric scaling to do an MDS analysis (similar to Harding and Sokal's fig. 2). Try to also recreated the MST shown in their figure 2.

Submit your figures and a brief paragraph describing what differences, if any, you found in your re-analysis of Harding and Sokal data. Are these differences significant (i.e. do they change your interpretation of the data)?

9

Clustering via K-means and Gaussian mixture modeling

K-means Clustering in R

The `kmeans()` function calcualtes standard k-means clusters in R. The input is a data matrix (perhaps transformed before hand) and k, the number of clusters. Alternatively you can specify starting cluster centers. You can also run the algorithm multiple times with different random starting positions by using the `nstart` argument.

```
# generate data set w/two groups (one of size 50, the other of size 75)
# note the different means and std dev between the two groups
> test.data <- rbind(matrix(rnorm(100, mean=0, sd=0.2),ncol=2),
                  matrix(rnorm(150,mean=1,sd=0.5),ncol=2))
> colnames(test.data) <- c("x", "y")
> plot(test.data)
> cl <- kmeans(test.data, 2)
> names(cl)
[1] "cluster"  "centers"  "withinss" "size"
> cl$cluster #  which cluster each object is assigned to
... output deleted ...
> plot(test.data, col = cl$cluster)
> cl$centers  # compare to the "true" means for the groups
            x         y
1 0.009479636 0.1182016
2 1.109641398 1.0427396
> points(cl$centers, col = 1:2, pch = 8, cex=2)

> # what if we pick the wrong number of clusters?
> cl <- kmeans(test.data, 5)
> plot(test.data, col = cl$cluster)
> points(cl$centers, col = 1:5, pch = 8, cex=2)

> # as above but using nstart argument
> cl <- kmeans(test.data, 5, nstart=25)
> plot(test.data, col = cl$cluster)
> points(cl$centers, col = 1:5, pch = 8, cex=2)
```

Applying K-means to the iris data set

Now that we've seen how to apply k-mean clustering to a synthetic data set, let's go ahead and apply it to our old friend the iris data set. Note that this is a four dimensional data set so we'll need to pick a projection in which to depict the cluster. The space of the first two principal components is a natural choice (but note that the fact that we're using the PCA space doesn't impact the k-means clustering in this context).

```
# drop the fifth column (species names)
# we'll assume we know how many groups there are
> k.iris <- kmeans(as.matrix(iris[,-5]), 3)
> iris.pca <- prcomp(iris[,-5])
# the following plot colors the specimens by the group
# they were assigned to by the k-means clustering
> plot(iris.pca$x,col=k.iris$cluster)

# this plot colors speciemsn by k-means grouping
# and chooses plot symbol by real species grouping.
# This can help us quickly pick out the misclasssified
# specimens
> plot(iris.pca$x, col=k.iris$cluster, pch=c(1,2,16)[iris[,5]])
```

Gaussian Mixture Models in R

There are multiple packages for fitting mixture models in R. We'll look at two – `mixtools` and `MCLUST`.

Installing mixtools

The package `mixtools` can be installed via the GUI or the `install.packages` command. A mixtools vignette can be downloaded from the CRAN website.

Using mixtools

We'll look at how to use mixtools using a data set on eruption times for the Old Faithful geyser in Yellowstone National Park (`?faithful` for details). We'll fit a univariate Gaussian mixture model to the time between eruptions data (`faithful$waiting`).

```
# allows us to refer to the variables within waiting time
# without using the standard list "$" syntax
> attach(faithful)

# create a nice histogram
> hist(waiting, main = "Time between Old Faithful eruptions",
xlab = "Minutes", ylab = "", cex.main = 1.5, cex.lab = 1.5, cex.axis = 1.4)

> library(mixtools)
> ?normalmixEM  # read the docs!
> wait.mix <- normalmixEM(waiting)

> names(wait.mix)
[1] "x"          "lambda"     "mu"         "sigma"      "loglik"     "
    posterior"
```

```
[7] "all.loglik" "restarts"   "ft"

# lambda is what we called "pi" in the lecture notes
> wait.mix[c("lambda","mu","sigma")]

> class(wait.mix)
[1] "mixEM"
> ?plot.mixEM  # read about the plotting options for the mixEM object
> plot(wait.mix, density=TRUE)
> plot(wait.mix, loglik=FALSE, density = TRUE, cex.axis = 1.4, cex.lab =
    1.4,
 cex.main = 1.8, main2 = "Time between Old Faithful eruptions", xlab2 = "
     Minutes")
```

Installing MCLUST

The package `MCLUST` is one of another package that provides maximum likelihood based estimation of mixture models. You will need to install the package (and it's dependencies) from the R GUI or using the `install.packages` command.

Using MCLUST

We're going to use two data set to illustrate some of `MCLUST`'s capabilities – the iris data set we've worked with before, and the bivariate version of the Old Faithful data set. We'll start off with the old faithful data set.

```
> plot(faithful$eruptions, faithful$waiting)
```

From visual inspection of the bivariate scatter plot, it looks like there two clusters. Let's apply the `Mclust` function and see what it suggests:

```
> library(mclust)
> fclust <- Mclust(faithful)
> fclust

 best model: elliposidal, equal variance with 3 components
```

Now that we've running the mixture model, let's look at the results graphically. The following call to `plot` will produce a series of plots.

```
> plot(fclust, data=faithful)
```

The first plot gives is a diagnostic plot that shows the likelihood of the model as a function of the number of groups (see BIC below). In general, when considering many possible models you want to pick the simplest model that has a highest likelihood. The second graphically represents the classification. The third plots highlights those objects for which the cluster assignment is most uncertain. The fourth plot gives a graphical representation of the Gaussian densities.

The `Mclust` function used a likelihood criterion called the "Bayesian Information Criterion" (BIC) to estimate the number of components (clusters) in the mixture model. By this criterion it suggested 3 components. BIC, like other information criteria (the Akaike Information Criterion is another popular one), is designed to help choose among parametric models with different numbers of parameters. It tries to choose the simplest model that provides a good fit to the data.

```
> names(fclust)
 [1] "modelName"      "n"              "d"              "G"              "
     BIC"
 [6] "bic"            "loglik"         "parameters"     "z"              "
     classification"
[11] "uncertainty"
> ?mclust  # check out the docs to read about all the returned parameters
```

Of course you don't have to accept the number of clusters that the `Mclust` function estimated. Here's how you'd calculate the mixture model with a user determined number of clusters:

```
> fclust2 <- Mclust(faithful, G=2)
> plot(fclust2, data=faithful)
```

Now let's generate some diagnostic plots for the mixture model:

```
> plot(fclust, data=faithful)
```

If you wanted to generate some of those plots individually you can do the following:

```
# generate a plot showing the classifications predicted by mixture model
> mclust2Dplot(data = faithful, what = "classification", identify = TRUE,
    parameters = fclust$parameters, z = fclust$z)
```

See the docs for the `mclust2Dplot` function for other options.

Mixture Models for the Iris data set

The `MCLUST` package includes the function `clPairs`, a very nice extension of the `pairs` function, for creating scatter plot matrices with group information. The following code illustrates this:

```
> names(iris) # remind yourself of the variables in the iris data set
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"

# 5th variable is the Species classification
> clPairs(data=iris[,-5], classification=iris[,5])
```

Let's see what `Mclust` makes of the iris data set:

```
> iclust <- Mclust(iris[,-5])
> iclust

 best model: ellipsoidal, equal shape with 2 components
> plot(iclust, data=iris[,-5])
```

Blind to the actual group structure the BIC suggests just two components, whereas we know there are three groups (though *I. versicolor* is thought to be an allopolyploid hybrid; see Kim et al. (2007) Ann Bot, 100: 219-224.). Examine the first graph produced by the `plot` call above to see how the 2 and 3 component models compare with respect to the BIC.

Now let's see how the mixture model does when we give it the true number of clusters:

```
> iclust3 <- Mclust(iris[,-5], G=3)
> plot(iclust3,data=iris[,-5])
```

To calculate the classification error rate we can compare the estimated clustering to the "true" (known) classification with the `classError` function (`?classError` for details):

```
> classError(iclust3$classification, iris[,5])
$misclassified
[1] 69 71 73 78 84

$errorRate
[1] 0.03333333
```

The `uncerPlot` command allows us to visualize the uncertainty implied by the mixture model to see how uncertain the model was about the misclassified samples.

```
> uncerPlot(iclust3$z, iris[,5])
```

In the uncertainty plot the vertical lines indicate the misclassified samples. As you can see those tend to be among the observations that the mixture model was most uncertain about with respect to which component they belonged to.

More details on MCLUST

See the MCLUST docs for in depth discussion of the use of `MCLUST`. The examples illustrated above were drawn from this documentation.

10

Building a Bioinformatics Pipeline, Part I

Overview

Many types of analyses, especially those involving genomic data, require the investigator to carry out a large number of sequential steps. For example, given a set of uncharacterized genes in your organism of interest you might want to find out as much as you can about the structure and function of the proteins they encode, search for related proteins in other organisms, and try to identify pathways that they might be involved in. If you had only a single gene of interest you might apply each of the appropriate software tools by hand to carry out such an analysis. However, when the number of genes of interest grows beyond a small number (say 10-15) doing such an analysis by hand starts to become tedious and error prone. A bioinformatics pipeline can help to automate this process, will make the analysis easier to replicate or apply to new sets of genes, and can be modified to include additional tasks. Writing out a series of analysis steps as a pipeline also helps us to achieve the goal of 'reproducible research' in the same way that Sweave helps you to do so in R.

Over the next three class sessions we're going to explore ways to create such a pipeline. Today we're going to start with some of the powerful tools and features that are typically 'built-in' to Unix or Unix-like systems.

Unix Tools and Concepts

The Unix command line

A Unix/Linux command line environment is the de-facto standard for building bioinformatics pipelines. While the command line may not be particularly user friendly, various aspects of how Unix is designed make it very powerful fo constructing analysis pipelines. We'll review some of those design aspects here.

Unix on Windows

If your computer has Linux or Mac OS X installed you already have a native Unix environment. If your computer runs Windows you can have access to a Unix-like environment by installing a program called Cygwin (http://www.cygwin.com). Cygwin is free, open source, and provides a convenient installer for common Unix programs. Download the installer program (`setup.exe`) and place it in a `c:\cygwin` directory (I recommend that you use this directory as the installation directory as well). The Cygwin setup program will install an icon on your desktop that you can double click to open a bash shell. Cygwin creates a set of subdirectories that mirrors the standard Unix file system (`/bin, /usr, /var, /home`, etc). When you start the bash shell you will be in your home directory (`/home/<username>`).

Unix tools

You'll need the following tools. Under Cygwin these are all easily installed using the GUI installation interface. On Linux or OS X many of these are already installed by default.

- autoconf - used to build source code; found under 'Devel' in the cygwin installer.
- less - a 'pager' (convenient for viewing files)
- gcc - provides a C language compiler; found under 'Devel' in the cygwin installer.
- make - used in building source code; found under 'Devel' in the cygwin installer.
- curl - a package for retrieving files using a variety ofInternet protocols; found under 'Net' in the cygwin installer.
- gzip - a file compression utility
- tar - a file archiving utility (usually used in conjunction with gzip)
- awk - a text processing programming language.

Basic Unix commands

You should familiarize yourself with the basic unix commands covered in the UNIX Tutorial for Beginners (see link on class website). Here are some of the more common ones you'll need to navigate around your file system:

- `ls` – list the content of a directory e.g. `ls /home/`
- `cd` – change directory. e.g. `cd /home/pmagwene/tmp`
- `pwd` – display the name of the present working directory.
- `mv` – move a file. e.g. `mv myfile newfile`
- `rm` – remove (delete) a file. Be careful with this one! e.g. `rm tmpfile`
- `find` – find files that match a given patern. e.g. `find . -name "*.txt"` (matches all files in the current directory that end with '.txt').
- `man` – show the manual pages for a command. e.g. `man ls`
- `less` – show the contents of a file, displaying one page at a time. e.g. `less somefile.txt` (use the space bar to advance, `b` to go back, `q` to quit)

The bash shell

The bash shell is the default shell on most Linux systems, Cygwin, and recent versions of OS X. The shell itself provides a useful framework for interacting with the operating system. Shell scripts can be written to make the shell environment even more powerful. We'll explore how to do this in today's exercises.

Here's a few efficiency tips to keep in mind when working in the bash shell:

- Scroll backwards and forwards through your command history using the up and down arrow keys
- Use the tab key to invoke command and file-name completion to keep your typing to a minimum
- Use `<ctrl-r>` and then start typing a word or phrase to search on; this invokes the history search mode to do a reverse incremental search of previous commands
  - Once you've found the command you were searching for hit `<Enter>` to execute it or `<ctrl-j>` to retrieve the command for further editing.
  - Use `<ctrl-g>` or `<ctrl-c>` to cancel the history search mode

The Unix philosophy

Doug McIlroy who invented the concept of the Unix 'pipe' (discussed below) summarized the Unix philosphy as follows:

> "This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface."

This is not some rigid set of specifications, but rather an approach to writing simple programs that can be tied together in useful ways to accomplish larger, more complex tasks. Most of the standard Unix commands are written with this philosophy in mind. The scripts you will develop over the next three class session will follow the same philosophy (and take advantage of other software tools that also use the Unix approach).

Everything is a file or a process

One of the aspects of Unix that makes it easy to tie programs together is that the operating systems treats pretty much everything as either a *file* or a *process*. There are three categories of files in Unix: plain files (e.g. text files, image files, word documents, video files, the code for a program, etc.), directories (e.g. your home directory, the root directory), and devices (e.g. the keyboard, a printer, a display screen, etc). The same basic set of commands can be applied to all three types of files.

A *process* is an instance of a running program. Everytime you start a program the operating system creates a process ID (PID) that is associated with that process. Processes typically operate on data in the form of files (of any of the three types) and return data that is sent to a file (again, any of the three file types). Any given process can start multiple subprocesses (also called child processes), however a process can only have one parent. For example, when you logon to a Unix system you are typically working in a shell process (common shells include bash, tcsh, csh, etc.). When you type a command like `ls` this creates

a child process. The parent process is temporarily suspended until the child process returns its output. The `ls` process takes a file as input (the current directory by default), and returns it's output to the display associated with the shell (represented by a device file).

Redirection and Pipes

Because Unix treats everything as a file or process, it's easy to change the source of input and the destination for output. There are several special operators that allow one to change the source/destination of input and output from a process. These are:

- `>` (redirect output operator)
- `>>` (append output operator)
- `<` (redirect input operator)
- `|` (pipe)

We'll give a few example of redirection and pipes using the commands `ls` (list directory contents), and `grep` (find lines matching a pattern):

```
$ ls -l
total 4184
-rw-r--r--     1 pmagwene  staff     1722 Mar 11  2009 #newfile3.lyx#
drwx------+   43 pmagwene  staff     1462 Nov 13 18:14 Desktop
drwx------+   20 pmagwene  staff      680 Sep 23 12:32 Documents
... output truncated ...
$ ls -l > ex1.out # redirect output to file
```

In the example above we redirected the output of the `ls -l` command to a file named `ex1.out`. Open the file to confirm this. Now we'll 'pipe' the output of `ls` to the `grep` command.

```
$ ls -l | grep 'Nov'
drwx------+   43 pmagwene  staff     1462 Nov 13 18:14 Desktop
drwx------+ 1285 pmagwene  staff    43690 Nov 13 18:48 Downloads
drwx------+   14 pmagwene  staff      476 Nov 10 11:51 Pictures
-rw-r--r--     1 pmagwene  staff     1427 Nov 14 14:13 ex1.out
-rw-r--r--     1 pmagwene  staff   604976 Nov  1 12:21 rolland-etal-2000-
    cAMP.pdf
```

In this example we used grep to show all the lines of the `ls` output that have the string 'Nov' in them.

Now we'll combine those two commands and redirect the output to another file.

```
$ ls -l | grep 'Nov' > ex2.out
```

Finally, let's use the append output operator to append to our file lines with 'Oct' as well.

```
$ ls -l | grep 'Oct' >> ex2.out
```

If we had used the redirection operator rather than append than it would have overwritten the previous contents of the file rather than adding the output to what was already there. Open `ex2.out` to confirm that your commands worked as expected.

Review chapter 3 of the UNIX Tutorial for Beginners (see link on class website) for more examples illustrating the use of redirection and pipes. We'll be using pipes and redirection throughout these hands-on exercises.

Using curl to retrieve files from the net

`curl` is a command line tool for transfering data to or from a server using a variety of different protocols including FTP, HTTP, SCP, etc. `curl` is available by default in recent versions of OS X. If you're running Cygwin on Windows you may have to install it from the "Net" subdirectory in the Cygwin installer.

Using `curl` is relatively straightforward. Here, we'll use it to download a file that we're going to use for today's exercises:

```
$ curl -O http://downloads.yeastgenome.org/curation/chromosomal_feature/
    saccharomyces_cerevisiae.gff
```

Type `man curl` or check out an online version of the manual for more information on using curl: http://curl.haxx.se/docs/manual.html.

The GFF3 File Format

GFF3 (GENERIC FEATURE FORMAT VERSION 3) is a text-based format for representing genomic features. It is widely used by the genomics research community for representing sequence features associated with genome projects. All of the major genome databases provide data in GFF3 format and most of the software tools used by the reserach community can parse GFF3 formatted files.

You can read the details of the GFF3 format here: http://www.sequenceontology.org/gff3.shtml. Notice that a GFF3 file consists 9 columns, separated by tabs. Read the above web page to understand what each of these 9 columns represents.

TOOLS FOR MANIPULATING TEXT

Many types of data, including GFF3 files, are structured text files. Because of this it's useful to have a handle on some of the major tools that Unix provides for manipulating such files.

head and tail

`head` and `tail` respectively show the first n and last n lines of a file (default $n = 10$). These can be useful for quickly checking out what's in a file. `tail` is especially useful for looking at log files to see the last few entries entered in a log.

```
$ head saccharomyces_cerevisiae.gff
... < output truncated > ...
$ tail saccharomyces_cerevisiae.gff
... < output truncated > ...
```

Use the `-n` argument to specify the number of lines you'd like to see:

```
$ head -n 3 saccharomyces_cerevisiae.gff
##gff-version 3
#date Mon Nov 15 19:50:13 2010
#
```

less

`less` is 'pager' program that allows you to scroll through a file (or standard input) page by page.

```
$ less saccharomyces_cerevisiae.gff
```

From within `less` you can scroll forward by hitting the space bar, or the 'f' key, backward by typing 'b'. To search for a particular word or pattern in the file type '/' followed by the word of interest and then hit return. All the instances of that word / pattern will be highlighted. For example, from within `less` type `/gene<RET>`, where `<RET>` means hit the enter or return key, to find all instances of the word 'gene' in the file. Type `q` to quit `less`.

echo

`echo` simply writes it's string argument to standard output (i.e. it echos what you type).

```
$ echo "hello, world"
hello, world
$ echo "These are the times that try men's souls"
These are the times that try men's souls
```

cat

`cat`, short for 'concatenate', is a utility for concatenating and printing text. Here are some examples of it's use:

```
$ echo "some text here" > file1.txt
$ echo "some more text" > file2.txt
$ cat file1.txt file2.txt > file1plus2.txt
$ cat file1plus2.txt
some text here
some more text
```

wc

`wc` is a program that counts the number of words, lines, and characters in a file. You can also specify you only want one of those counts using options like `-l` (count only words).

```
$ wc saccharomyces_cerevisiae.gff
  168490  299825 18871650 saccharomyces_cerevisiae.gff
$ wc -l saccharomyces_cerevisiae.gff
  168490 saccharomyces_cerevisiae.gff
```

cut

`cut` is a utility for subsetting words, bytes or columns of a text file. For example:

```
$ cut -f1-3 saccharomyces_cerevisiae.gff | less
```

In the above we use cut to show the first three fields of the file, and then we pipe it to `less` to examine one page of text at a time . The default field delimiter in `cut` is a tab (`\t`), but you can specify other delimiters with the `-d` option. You don't have to use adjacent columns with `cut`. For example,

```
$ cut -f1,3-5,7 saccharomyces_cerevisiae.gff | less
```

This allows us to look at the first column, and columns 3-5 and 7, corresponding to the seqid (=chromosome), the feature type, the feature start and stop coordinates (1-based), and the strand on which the feature is defined.

Notice how in addition to the fields, `cut` also gave use the header information at the beginning of the file. We can use the `-s` option to suppress lines that don't have the field delimiter character:

```
$ cut -s -f1,3-5,7 saccharomyces_cerevisiae.gff > out.txt
```

Notice this time we redirected the output of the command to a file, `out.txt`.

sort

The `sort` utility sorts lines of text. By default `sort` interprets an entire line of text as the key for sorting and sorts in dictionary order. For example, to see the default sorting:

```
$ sort out.txt | less
```

We can use the `-k` option to specify the field to sort on. For example, this is how we can sort on the second column of `out.txt`:

```
$ sort -k2 out.txt | less
```

Another useful option to sort is `-u` which tells sort to output only the first instance of a set of identical keys. Try and figure out what the following command does before running it:

```
$ cut -s -f3 saccharomyces_cerevisiae.gff | sort -u
```

grep

`grep` is a tool for doing regular expression matching on lines of a file. Regular expressions are a way to specificy search patterns in strings. The simplest type of regular expression is to just search for a specific word, as illustrated here:

```
$ grep "gene" out.txt | less
```

The above command simply returns all the lines in `out.txt` that have the word “gene" in them. Let’s use this in a slightly different way to count instances of different features in the file:

```
$ grep "gene" out.txt | wc -l
   6720
$ grep "pseudogene" out.txt | wc -l
     21
$ grep "telomere" out.txt | wc -l
     32
```

NOTE: the numbers of matches may change somewhat between releases of the curated yeast genome. If the numbers you get above or below are *slightly* different from what is shown here don’t worry.

We can get a little fancier if we use the “extended” `grep` syntax (specified using the `-E` option). Here’s how we can search for lines that match on any of a set of terms (the vertical bar `|` indicates an “OR" operator):

```
$ grep -E "tRNA|rRNA|snRNA|snoRNA" out.txt | wc -l
    409
```

Note that we have to be careful about what `grep` matches, for example:

```
$ grep "chr01" out.txt | grep "gene" | wc -l
   121
```

Note how we piped two `grep` commands together to get the equivalent of AND (“chr01” AND “gene”). However, there’s a very subtle problem with this command as constructed.

We search on the word “gene", but “gene” is also a substring of “psuedogene” and hence “pseudogene” features also generate matches. What we really want is whole word matches. We can do that as follows:

```
$ grep "\<chr01\>" out.txt | grep "\<gene\>" | wc -l
     117
```

This uses what are called “POSIX character classes" to match possible sets of characters. A list of the POSIX character classes is linked to on the course wiki. Here’s the equivalent call for counting genes on chromosome IV:

```
$ grep "\<chr04\>" out.txt | grep "\<gene\>" | wc -l
     836
```

We’ve only just scratched the surface of regular expressions. Regular expressions are a very powerful tool and there are whole books on the topic. I’ll post a number of links on the course wiki to online tutorials on `grep` and regular expressions.

tr

`tr` is a utility for translating characters within a text stream. `tr` can be useful for converting delimiters from one file type to another. For example, let’s say we wanted to analyze the file `out.txt` in a program that expected comma separated values (csv) instead of tab-delimited fields. `tr` makes that conversion easy:

```
$ cat out.txt | tr "\t" "," > out.csv
```

Note that `tr` only reads from standard input so we used the `cat` program to feed the lines of text to `tr`.

awk

`awk` is a programming language designed for processing structured text files. You can use it to write short one liners or to write full blown programs. It turns out that some form of text file manipulation is often a necessary first step in most bioinformatics analyses, so `awk` often comes in very handy. We’ll use `awk` to illustrate how you might transitions from simple command line usage into slightly more complicated scripts.

One simple thing we can do with awk is to use it to re-order fields in a structured data file:

```
awk ’{print $2, $1, $4, $5}’ out.txt | less
```

In the command above the the dollar signs followed by numbers refer to the fields of the file. With it’s default setting `awk` operates line by line, so you can interpret the above statement as saying: “for each line, print the fields 2, 1, 4 and 5".

The basic syntax of `awk` is often depicted in the form `pattern {action}`. The above command only specified an action, so it was applied to every line. By contrast, in the example below we specify a pattern. The pattern can be read as – “if the 3rd field is 'chromosome'". For all lines that match that pattern the correspoding action is applied; in this case “print fields 1 and 5" (the chromosome name and its length):

```
awk ’$3=="chromosome" {print $1, $5}’ saccharomyces_cerevisiae.gff
```

Here’s another `pattern {action}` pair that shows how we could find all gene features with length less than 300:

```
$ awk '$3 == "gene" && ($5 - $4) < 300 {print $0 }' \
  saccharomyces_cerevisiae.gff | wc -l
     446
```

&& is the AND operator. Read this as "if the 3rd fields is 'gene' AND the the 5th field minus the 4th field is less than 300."

In this last example we added one more condition – we looked for the word 'Dubious' in the 9th field. The results indicate that a significant proportion of these small genes are classified as 'Dubious'.

```
$ awk '$3 == "gene" && ($5 - $4) < 300 && match($9, "Dubious") {print $0 }'
     \
    saccharomyces_cerevisiae.gff | wc -l
     163
```

There's lots of powerful things you can do with `awk` one-liners, but writing short command files often makes things easier to understand. You can think of an `awk` command file as a set of `pattern {action}` statements. Our command file will create a table giving both the length of each chromosome and the number of genes on that chromosome. Save the following awk script in a file called `gcount.awk`.

```
# gcount.awk
# length of each chromosome
$3 == "chromosome" {
    clen[$1] = $5
}

# increment the gene count for the given chromosome
$3 == "gene" {
    ngenes[$1] += 1
}

# once we've processed all the records
END {
for (chr in clen) {
    print chr "\t" clen[chr] "\t" ngenes[chr]
    }
}
```

In the example above we create two arrays – `clen` and `ngenes` – to keep track of the chromosome lengths and number of genes on each chromosome. Arrays can be indexed by either integers or strings; when there indxed by strings we can think of them like Python dictionaries. We have two patterns – whether the 3rd field equals 1) "chromosome" or 2) "gene". The final pattern, labeled `END`, says what to do once we've processed all the lines in the file. To run this script do the following:

```
$ awk -f gcount.awk saccharomyces_cerevisiae.gff
```

The `-f` option says to use the pattern/action pairs contained in the specified file. One possible shortcoming (at least on my system) is that the output wasn't sorted. That's easy to solve by piping the results to the sort command:

```
$ awk -f gcount.awk saccharomyces_cerevisiae.gff | sort
```

Our `gcount.awk` script works pretty well, but what if we wanted to count pseudogenes rather than genes, or tRNA features? In its current form the feature type is hardcoded into the script. Let's see how we can get rid of that constraint. Save the following awk script as `fcount.awk`.

```
# fcount.awk
BEGIN {
# if var ftype has NOT been defined, assign it a default value
if (!ftype)
  ftype = "gene"
}

# length of each chromosome
$3 == "chromosome" {
    clen[$1] = $5
}

# increment the feature count for the given chromosome
$3 == ftype {
    ngenes[$1] += 1
}

END {
for (chr in clen) {
    print chr "\t" clen[chr] "\t" ngenes[chr]
    }
}
```

Here we introduced the `BEGIN` pattern. This pattern is carried out before any lines of the file are processed. By default, this new script will count genes like our previous script did, but if you specify the variable `ftype` using the `-v` option on the command line it will count the specified feature type:

```
# count pseudogenes
$ awk -f fcount.awk  -v ftype="pseudogene" saccharomyces_cerevisiae.gff

# counts ARS sequences (origins of replication)
$ awk -f fcount.awk  -v ftype="ARS" saccharomyces_cerevisiae.gff

# count tRNA genes
$ awk -f fcount.awk  -v ftype="tRNA" saccharomyces_cerevisiae.gff
```

As a final example of using `awk`, let's say you wanted to look for all gene features that include reference to a particular gene family in the attribute field (column 9 of the GFF file). You could do something like this:

```
$ awk 'match($9,"FLO") && $3 == "gene" { print $0 }' \
  saccharomyces_cerevisiae.gff  | less
```

This works, but the output is a bit ugly because of how the attribute field is specified. Let's write a simple awk `function` that nicely formats the output, with each of the attributes appearing as a subline. Save the following script as `attribs.awk`.

```
# attribs.awk
```

```
# parse the attributes field of a GFF file

NF >= 9 {
    # print the first 8 fields
    print $1, $2, $3, $4, $5, $6, $7, $8

    # break the attributes field up into individual attributes
    n = split($9, attributes, ";")
    for (i = 1; i <= n; i++){
        tstr = attributes[i]
        gsub(/%20/," ", tstr ) # spaces
        gsub(/%2C/, ",", tstr) # commas
        gsub(/%3B/, ";", tstr) # semi-colons
        gsub(/%2F/, "/", tstr) # forward slash
        print "\t\t" tstr
        }
    print "\n"
}
```

The awk function `gsub()` globally substitutes one string for another. In this case it's replacing HTML type encoding of spaces, commas, semi-colons, etc. with more human friendly versions of the same.

We can use our `attribs.awk` script as follows:

```
$ awk 'match($9,"FLO") && $3 == "gene" { print $0 }'  \
  saccharomyces_cerevisiae.gff  | awk -f attribs.awk | less
```

This produces output that is much nicer for a human reader to interpret, though less easy to parse computationally.

The GFF3 format is used by many organism specific genome projects besides yeast. If we take care to write our scripts to operate on GFF3 files generically then we can apply scripts we write for one organism easily to another organism. Let's test this out by downloading the X-chromosome GFF3 file for *Drosophila melanogaster* from FlyBase:

```
$ curl -O ftp://ftp.flybase.net/genomes/dmel/current/gff/dmel-X-r5.41.gff.
    gz
$ gunzip dmel-X-r5.41.gff.gz  # unzip the compressed file
```

Now let's test our `attribs.awk` script with this new GFF file by generating a report on pseudogenes on the *Drosophila* X-chromosome:

```
$ awk '$3 == "pseudogene"  {print $0}' dmel-X-r5.41.gff | awk -f attribs.
    awk > fly-X-pseudogenes.txt
```

Use `less` or a text editor to view your report.

TIEING IT ALL TOGETHER WITH BASH

To this point all of our examples have involved single command lines or scripts, occasionally tied together with pipes. This works well for quick analyses, but what if you wanted to run an analysis over and over again, say on a monthly basis as a genome project was updated, or as you generated new data as part of your research? In that context a shell script might

be useful. I'm going to assume you're using bash as your shell (the default on OS X, cygwin, and most Linux based systems).

You can confirm that your default shell is bash by doing something like:

```
$ sh --version
GNU bash, version 3.2.48(1)-release (x86_64-apple-darwin10.0)
Copyright (C) 2007 Free Software Foundation, Inc.
```

Assuming, that you've got bash working on your system, enter the following code into your text-editor and save it with the filename `genome_reporter.sh` in the same directory where you've saved `fcount.awk` that we created earlier. Be careful that you enter the text as shown as `bash` is particularly picky about extra spaces around the equal sign (=) in variable assignment so if you get error messages when you try and run this script (see below), that's the first thing to check.

```
#!/bin/bash

URL='http://downloads.yeastgenome.org/curation/chromosomal_feature/
    saccharomyces_cerevisiae.gff'
BASEFILE='saccharomyces_cerevisiae.gff'

# get today's date
TODAY=$(date -u +%Y-%m-%d)

# create filename, prepended w/today's date
FILENAME="$TODAY-$BASEFILE"
REPORT="report-$FILENAME"

# if the GFF file does not already exist then
# use curl to download the file and save if with the name above
if [ ! -e $FILENAME ]
then
    curl -o $FILENAME $URL
fi


# create report with a series of awk calls
echo -e "Genome Report\nPrepared: $TODAY\n" > $REPORT

echo "Total genes: " >> $REPORT
awk '$3 == "gene" {print $0}' $FILENAME | wc -l >> $REPORT

echo -e "\nDubious ORFs: " >> $REPORT
awk '$3 == "gene" && ($5 - $4) < 300 && match($9, "Dubious") {print $0 }'
    $FILENAME | wc -l >> $REPORT

echo -e "\nPseudogenes: " >> $REPORT
awk -f fcount.awk  -v ftype="pseudogene" $FILENAME | wc -l >> $REPORT

echo -e "\nChromosome, length, genes per chromosome: " >> $REPORT
awk -f fcount.awk $FILENAME | sort >> $REPORT
```

Note that the line `#!/bin/bash` needs to be the first line in the file. This tells the operating system to run this script using the bash shell. This line is sometimes referred to as the 'she-bang' line by Unix programmers. We'll see next week how to set this for a Python program.

Having entered and saved that script, make the script executable by typing:

```
$ chmod +x genome_reporter.sh
```

from the command line. Once you've done that you can run the script, from the same directory, by typing:

```
$ ./genome_reporter.sh
```

Assuming you don't have any errors the script will download the GFF file from the Saccharomyces Genome Database, save it with the date prefixed to the file name, and then generate a short report listing some useful summaries generated from the file.

Most of the bottom half of the script should be easy to understand; it simply shows a bunch of `echo` and `awk` commands that you might have typed at the command line. In the top portion of the script we create a set of variables to hold the names of the files we'll be using. One new feature we haven't seen before is the use of the dollar sign (`$`) to dereference variable names. For example, the variable `FILENAME` is constructed by creating a string by joining together the strings held in the variables `TODAY` and `BASEFILE` (and separated by a dash `-`). Depending on the date on which the script is run it generates a different set of file names, as specified by the variables `TODAY`, `BASEFILE`, and `REPORT`. The bottom half of the script is setup to generate the appropriate output given those changing variables.

One other feature to take note of is the `if-then-fi` conditional statement. The portion in the square brackets (`[ ! -e $FILENAME ]`) asks whether the file for that date already exists. If so it doesn't bother downloading the file again, for efficiency reasons.

Like regular expressions, `bash` scripting can be quite involved. We'll create some additional `bash` scripts next class sessions and I'll provide some web links on the course wiki if you want to learn more about working with `bash`.

11

Building a Bioinformatics Pipeline, Part II

BIOINFORMATICS TOOLS

In last weeks class we worked through some examples that illustrated how standard Unix command line tools could be chained together into simple pipelines to analyze genome sequence information. Today we'll examine how to build pipelines using a mix of standard bioinformatics software packages and a Python script.

Installing the tools

For the purposes of these exercises we will install all of our software tools into a directory called `tmp` in your home directory (you might already have such a directory).

```
$ mkdir ~/tmp
```

Recall that the tilde (~) is Unix shorthand for your home directory (`/Users/pmagwene` in my case).

Cygwin on Windows

If you're running Windows, see the Notes on Cygwin link on the course wiki for some information that may be required to get the tools and scripts listed below working properly.

Clustalw

Clustalw is a popular progressive multiple sequence alignment program. There are pre-built binaries of Clustalw available but for the purposes of this exercises we're going to build the program from source. This assumes you have the appropriate command line tools available on your platform, either the Xcode tools on OS X, or Cygwin plus gcc, make, autoconf, etc.

Use the `curl` command to download the source code for Clustalw as I demonstrate below:

```
$ cd ~/tmp
$ curl -O ftp://ftp.ebi.ac.uk/pub/software/clustalw2/2.1/clustalw-2.1.tar.
    gz
```

Once you have the clustalw source code decompress the tarball and make the programs as follows:

```
$ tar xvzf clustalw-2.1.tar.gz
$ cd clustalw-2.1/
$ ./configure  # configures the source code for your operating system
$ make  # compiles the code files
```

```
$ sudo make install # puts the binary executables in standard directories
                    # you don't need sudo on cygwin
```

The command `sudo make install` will install the clustalw binaries you just built into the `/usr/local/bin` directory. `sudo` let's you run this command with 'superuser' (i.e. admin) privileges. Assuming everything compiled without errors you can check that the program installed correctly as follows:

```
$ which clustalw2
/usr/local/bin/clustalw2

$ clustalw2

 ***********************************************************
 ******** CLUSTAL 2.0.12 Multiple Sequence Alignments  ********
 ***********************************************************

     ... output truncated ...
```

MAFFT

MAFFT is another multiple sequence alignment program. It's relatively fast and a number of studies have shown that it is amongst the best performing multiple sequence aligners. MAFFT is usually the sequence aligner I reach for first. Clustalw is the 'classic' alignment tool, so it's useful to have on your system, but MAFFT usually gives better alignments (though Clustalw2 is supposed to address some of the short-comings of the older versions of Clustalw). See the MAFFT website for additional references and information.

There are pre-compiled MAFFT binaries available on the MAFFT website. However, again let's build this program from source. First, we grab the latest version of the source code (as of 10 Nov 2011):

```
$ cd ~/tmp; curl -O http://mafft.cbrc.jp/alignment/software/mafft-6.864-
    with-extensions-src.tgz
```

Notice how I put two commands on the same line by separating them with a semi-colon. We then unpack the tarball and build the source as follows:

```
$ cd mafft-6.864-with-extensions/core
$ make clean
$ make
$ sudo make install
```

Building MAFFT was almost exactly the same as building Clustalw, except we didn't need to use the `configure` command in this case.

Once you've built and installed MAFFT check the installation location and confirm that the binary is working:

```
$ which mafft
/usr/local/bin/mafft
$ mafft  # type ctrl-c to exit from the interactive prompt

--------------------------------------------------------------------------
```

```
   MAFFT v6.864b (2011/11/10)

        Copyright (c) 2011 Kazutaka Katoh
        NAR 30:3059-3066, NAR 33:511-518
        http://mafft.cbrc.jp/alignment/software/
--------------------------------------------------------------------
```

HMMER

HMMER is an implementation of a profile Hidden Markov Model (HMM) for protein sequence analysis. You can read up on HMMER at the HMMER website. We will use it here for finding protein domains in sequences in conjuction with the PFAM database. By now, the steps needed to download and build programs from source are familiar to you:

```
$ cd ~/tmp
$ curl -O http://selab.janelia.org/software/hmmer3/3.0/hmmer-3.0.tar.gz
$ tar xzvf hmmer-3.0.tar.gz
$ cd hmmer-3.0
$ ./configure
$ make
$ make check # optional, not every source package includes such a call
$ sudo make install
```

After compiling from source as above the binaries will be placed in `/usr/local/bin`.

Get the PFAM HMM library

We will be using the Pfam database (Release 25) in conjunction with HMMER to search for known protein domains in our sequences of interest. Since the the Pfam HMM libraries are large I'll try and provide a couple of thumb drives with the necessary library. If you're using this document outside of class you can download the necessay library as follows:

```
$ curl -O ftp://ftp.sanger.ac.uk/pub/databases/Pfam/releases/Pfam25.0/Pfam-
    A.hmm.gz
```

This is a large file (185MB) and decompresses to an even larger file (approx. 986MB). Make sure you have adequate disk space. Unzip it as follows:

```
$ gunzip Pfam-A.hmm.gz
```

Install the BLAST+ suite

NCBI provides a set of command line blast tools called the BLAST+ suite. This used to include a command line program called `blastcl3` that you could use to run a BLAST search against the NCBI BLAST servers. In the latest version of the BLAST+ toolset each of the core blast programs has a `-remote` option that does the same thing. The `-remote` option is useful for a moderate number of files, but if you were building a computationally intensive pipeline you would want to install a local copy of the BLAST databases and the accompanying tools.

Becuase the BLAST+ tools are complicated to build from source, we'll use the precompiled binaries that NCBI provides. Download the appropriate binary distribution for your from the NCBI ftp server (see links on class wiki). Once you've got the file you simply unzip the tarball in your `~/tmp` directory. The binaries can be installed elsewhere but we'll just leave everything in `~/tmp`.

```
$ cd tmp
$ curl -O ftp://ftp.ncbi.nlm.nih.gov/blast/executables/LATEST/ncbi-blast
    -2.2.25+-universal-macosx.tar.gz
$ tar xvzf ncbi-blast-2.2.25+-universal-macosx.tar.gz
```

To confirm that the binary works on your system do the following:

```
$ ./ncbi-blast-2.2.25+/bin/blastp -help
```

If you're on a Mac and get an error like the one shown below, try downloading and installing a slightly older version of the BLAST+ tools from ftp://ftp.ncbi.nlm.nih.gov/blast/executables/blast+/2.2.23/.

```
dyld: lazy symbol binding failed: Symbol not found:
    __ZSt16__ostream_insertIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_PKS3_i

  Referenced from: /Users/pmagwene/tmp/./ncbi-blast-2.2.24+/bin/blastp
  Expected in: flat namespace

dyld: Symbol not found:
    __ZSt16__ostream_insertIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_PKS3_i

  Referenced from: /Users/pmagwene/tmp/./ncbi-blast-2.2.24+/bin/blastp
  Expected in: flat namespace
```

Once you've confirmed that `blastp` works on your system check out the PDF documentation in the directory `~/tmp/ncbi-blast-2.2.25+/doc/` (or wherever you unzipped the file).

Install Biopython

If you're on Windows you can download a prebuilt binary of Biopython from the Biopython website. For OS X you can install Biopython from source relatively easily, assuming you've previously installed setuptools which includes the `easy_install` program:

```
$ sudo easy_install -f http://biopython.org/DIST/ biopython
```

Testing the bionformatics tools

Download the `fungal-ras.fas` file from the class website. This file includes protein sequences of Ras-family proteins from a number of different fungi. Ras proteins are small GTPases that are involved in cellular signal transduction. Ras signaling is often involved in cellular growth and differentiation and mutations that affect Ras signaling often lead to cancer. We'll use this data to do some quick tests to confirm that our software tools are working correctly. Of course, when putting together an analysis pipeline for your own purposes you'll want to spend a fair amount of time reading the documentation (and related papers) for each tool and make sure you understand the various options and settings.

Multiple alignments with Clustalw and MAFFT

Let's run Clustalw and MAFFT to align the Ras sequences:

```
$ clustalw2 -infile=fungal-ras.fas
$ mafft --auto fungal-ras.fas > fungal-ras-mafft.fas
```

The commands above should produce the following files: `fungal-ras.aln` and `fungal-ras.dnd` (Clustalw) and `fungal-ras-mafft.fas` (MAFFT). To visualize the alignments there are a variety of different multiple alignment viewers. One such program is Jalview, a free cross platform, multiple alignment viewer/editor written in Java. Take a look at both the clustalw and mafft alignments using Jalview.

Testing HMMER

To test out HMMER and Pfam download the `Rme1.fas` file from the class website. Rme1 is a transcription factor that regulates sporulation and meiosis in budding yeast, *Saccharomyces cerevisiae*. We'll use HMMER to analyze the domain structure of Rme1.

The first thing you'll need to do is run `Pfam-A.hmm` through the `hmmpress` program which prepares the HMM database for fast scanning by creating binary files. This might take a few minutes depending on the speed of your machine.

```
$ hmmpress Pfam-A.hmm
```

This will create a number of additional files in the same directory as `Pfam-A.hmm`. We can now use `hmmscan` to search for known protein domains included in the Pfam database.

```
$ hmmscan Pfam-A.hmm Rme1.fas | less
```

The default output from `hmmscan` is designed to be human readable. For outputs that are easier to parse computationally use the `--tblout` or `--domtblout` options to save output in a tabular format.

```
$ hmmscan --domtblout Rme1-output.txt -o /dev/null Pfam-A.hmm Rme1.fas
```

This call produces a file `Rme1-output.txt` that contains a space delimited text file summarizing the per-domain output. The `-o` option redirects the main human-readable output (in this case to the 'bit-bucket', /dev/null). You can then manipulate the tabular output in `Rme1-output.txt` using standard Unix tools like `awk`.

See pp. 24-26 of the HMMER user guide for more info on the `hmmscan` program and settings. E-values and bit-scores are the criteria you want to look at when trying to judge which domains you sequence of interest has good matches to. HMMER bit scores reflect the extent to which a sequence is a good match to a profile model (higher bit scores are better matches). See p. 43 of the HMMER 2.3 user guide (use Google to find a copy of the older version of the HMMER manual) for a discussion of E-values and bit scores. If you examine the output file `Rme1-output.txt` you'll see that the model with the lowest E-values and highest bit-scores is a "zinc finger" domain. There are three such domains in the Rme1 protein (see the column labeled "N" in the output), though two of them are weaker matches (larger E-values). Rme1 is a zinc finger transcriptional factor. The two weaker zinc finger domains are weak matches to the HMM model for zinc fingers but are nonetheless functional domains.

Testing the BLAST+ tools

Let's use `blastp` to query the NCBI BLAST servers for proteins with sequence homology to Rme1.

```
$ cd ncbi-blast-2.2.25+/bin
$ ./blastp -db refseq_protein -evalue 2e-10 -remote < ~/tmp/Rme1.fas > ~/
    tmp/rme1-blast.out
```

This runs a remote BLAST search with an E-value cutoff of 2^{-10}. It might take a little while because it will access the NCBI servers to run the search. Note the use of both an input redirection operator as well as an output redirection operator. If you want the output in HTML format for easy reading run it as follows and open the resulting file `rme1-blastout.html` in your web browser:

```
$ ./blastp -db refseq_protein -evalue 2e-10 -remote -html < ~/tmp/Rme1.fas
    > ~/tmp/rme1-blast.html
```

If all of the commands above worked (produced no errors, generated valid output) then you're ready to move on to implementing the pipelines.

The Pipelines

In today's exercises we're going to look at two very simple pipelines. In next week's class we'll combine and expand on these pipelines using Python.

Pipeline I

This first pipeline will do the following:

1. Read in a protein sequence from a FASTA file
2. Query the NCBI BLAST servers for potential orthologues
3. Analyze the query protein for known protein domains using HMMER and Pfam
4. Prepare a report combining the HMMER/Pfam output with the BLAST output

Here's a simple bash script that implements this pipeline. Save this script as `pipeline1.sh`.

```
#!/bin/bash

# change these if these files are located somewhere else
BLASTP=$HOME/tmp/ncbi-blast-2.2.25+/bin/blastp
HMMSCAN=/usr/local/bin/hmmscan  # change as appropriate
PFAMDB=$HOME/tmp/Pfam-A.hmm

progname="$0" # the name of the program, we won't use this for anything
fastafile="$1" # the input fasta file
outfile="$2" # the output file for the report

# run blastp
echo "Running blastp"
"$BLASTP" -db refseq_protein -evalue 2e-10 -remote < "$fastafile" > "
    $outfile"

# run hmmscan and append to outfile
echo "Running hmmscan"
"$HMMSCAN" "$PFAMDB" "$fastafile" >> "$outfile"
```

As we've seen before, the 'she-bang' line simply instructs the operating system to run the script with bash. The next three lines specify where on your system you've installed the `blastp` and `hmmscan` binaries and the Pfam database. This script expects two input arguments – the first of which specifies the input fasta file and the second giving the name of the file in which to create the report. We save these values as the variables `fastafile` and `outfile`. Here the syntax `$1` and `$2` refers to the first and second arguments given at the command line. Once all the variables are defined the script simply runs a remote `blastp` search and `hmmscan` using the given fasta input.

Once you've entered and saved the code for `pipeline1.sh` you need to make it executable:

```
$ chmod +x pipeline1.sh
```

You can then execute the code as follows:

```
$ ./pipeline1.sh Rme1.fas Rme1-report.out
```

Assignment 3: Submit your `Rme1-report.out` file via email.

Pipeline II

Our second pipeline will do the following:

1. Translate a nucleotide FASTA file to amino acid sequences
2. Perform a multiple alignment of the amino acid sequences using MAFFT
3. Place the translated output and multiple sequence alignments in separate files
4. Run HMMMER on the the translated output and produce a report file.

A small Python script

First we're going to write a Python script to take care of step 2, translating the nucleotide sequence to a protein sequence. Save the following code as `aatranslate.py`.

```
#!/usr/bin/env python

import sys
from Bio import Seq, SeqIO
from Bio.SeqRecord import SeqRecord
from Bio.Data.CodonTable import TranslationError

recs = SeqIO.parse(sys.stdin, "fasta")

for rec in recs:
    try:
        newrec = SeqRecord(rec.seq.translate(), id=rec.id+"_translated",
                           description=rec.description + ' (translated
                               )')
        print newrec.format("fasta")
    except TranslationError:
        print rec.format("fasta")
```

After saving the above code as `aatranslate.py`, make it executable by using `chmod +x`. This script takes a set of FASTA nucleotide sequences from from `stdin` and translates them, sending the output to `stdout`. This small script uses a number of classes and functions from the Biopython library that we'll discuss in greater detail in the next class session. Test your function as shown below. Notice how we use `cat` to pipe the fasta file to `aatranslate.py`.

```
$ less unknowns.fas # confirm that the unknowns are nucleotid seqs, q to
    quit
$ cat unknowns.fas | ./aatranslate.py | less
```

Combining Python and MAFFT

Save the following code as `pipeline2.sh`:

```
#!/bin/bash

# change these if these files are located somewhere else
TRANSLATE=$HOME/tmp/aatranslate.py
MAFFT=/usr/local/bin/mafft
HMMSCAN=/usr/local/bin/hmmscan
PFAMDB=$HOME/tmp/Pfam-A.hmm

scriptargs="fastafile aafile alignfile reportfile"
E_WRONG_ARGS=85
nexpargs=4
args=$#

# check that we got the expected number of arguments to the script
if [[ $args -ne $nexpargs ]]
then
    echo
    echo "Usage: `basename $0` $scriptargs"
    echo
    exit $E_WRONG_ARGS
fi

progname="$0" # the name of the program
fastafile="$1" # the input fasta file
aafile="$2" # the output file for the AA translation
alignfile="$3" # the output file for the aligned sequences
reportfile="$4" # output file for the HMMER report

echo "Translating sequences"
cat "$fastafile" | "$TRANSLATE" > "$aafile"

echo "Aligning sequences"
"$MAFFT" --auto --quiet "$aafile" > "$alignfile"

echo "Running hmmscan"
"$HMMSCAN" "$PFAMDB" "$aafile" > "$reportfile"
```

After setting the script to be executable you can run it like so:

```
$ ./pipeline2.sh unknowns.fas unknowns-trans.fas unknowns-align.fas
    unknowns-report.txt
```

This will produce three files, one containing the translated amino acid sequences, the second giving the multiple alignment of those amino acid sequences, and the third with the PFAM output. Use less or a text editor to take a look at the generated files.

At the beginning of this script we added a little bit of error checking code to insure that the correct number of arguments were provided on the command line. If no arguments or the wrong number of arguments are provided the script gives the user a little usage information and exits gracefully. Try running the script as:

```
$ ./pipeline2.sh
```

Compare this to what happens when you run `pipeline1.sh` without the arguments:

```
$ ./pipeline1.sh
```

Assignment 4: Submit your `unknowns-report.txt` file via email.

12

Building a Bioinformatics Pipeline, Part III

Overview

Last week we installed a number of bioinformatics tools and showed how they could be used to build simple analysis pipelines using bash scripts. This week we'll build a more sophisticated pipeline using BioPython. This pipeline will incorporate such features as web based queries and conversion of information between different file formats.

The Pipeline

The tasks carried out by the pipeline will be as follows:

- Read in a nucleotide sequence from a FASTA file
- Translate the nucleotide sequence to an amino acid sequence
- Do a blastp search against human and fly proteins in the Swiss-Prot database using an interface to the NCBI web version of BLAST
- Download protein sequences for the best blast hits from Swiss-Prot
- Use MAFFT to do a multiple alignment of the original amino acid sequence and the presumed orthologs generated via the blast search
- Analyze the query protein for known protein domains using HMMER and Pfam

You will need a working installation of Python (2.6+), IPython, and the BioPython library (1.53+) as well as the command line tools we installed last week (MAFFT, HMMER).

The test files

Download the file `unknown1.fas` and `unknown2.fas` from the class website. I recommend you place these in `~/tmp`.

Reading in a single sequence from a FASTA file

As we build our pipeline I will first demonstrate the use of various modules, classes, and functions in the interactive shell and then I will give a set of functions that consolidate the commands to make them convenient to use.

We'll start by showing how to read sequence data out of a FASTA file:

```
>>> cd ~/tmp
>>> from Bio import SeqIO
>>> u1 = SeqIO.read('unknown1.fas','fasta')
>>> type(u1)
<class 'Bio.SeqRecord.SeqRecord'>
>>> u1
SeqRecord(seq=Seq('ATGATGAATTTTTTTACATCAAAATCGTCGAAT
CAGGATACTGGATTTAGCTCT...TGA', SingleLetterAlphabet()),
id='YHR205W', name='YHR205W', description='YHR205W  Chr 8', dbxrefs=[])
>>> u1.name
'YHR205W'
>>> u1.description
'YHR205W  Chr 8'
>>> u1.seq
Seq('ATGATGAATTTTTTTACATCAAAATCGTCGAATCAGGATACTGG
ATTTAGCTCT...TGA', SingleLetterAlphabet())
>>> u1.seq[:10]
Seq('ATGATGAATT', SingleLetterAlphabet())
>>> u1.seq[0]
'A'
>>> u1.seq[9]
'T'
>>> u1.seq[:10].tostring()
'ATGATGAATT'
>>> u1.seq.translate()[:10]
Seq('MMNFFTSKSS', HasStopCodon(ExtendedIUPACProtein(), '*'))
```

`SeqIO` is a sub-module of the top-level module BioPython module `Bio`. `SeqIO.read` reads a single sequence object from a file and returns an instance of a `SeqRecord` class (defined in the Biopython package). A *class* is a programming concept that groups data and functions that operate on that data into a single object. For example, in the code above we used the `.name` and `.description` attributes to examine information about the sequence (this information was retrieved from the FASTA file itself). A `SeqRecord` holds a `Seq` object (yet another class!) as well as accessory information like the name of the sequence, a description, etc. `Seq` objects act very much like strings in terms of slicing and element access but they also have specialized function like `.translate()` that can be used to translate a nucleotide sequence into a peptide sequence.

Reading in multiple sequences from a FASTA file

In the code above we demonstrated how to read a single sequence from a FASTA file. Here we demonstrate how to read multiple sequences. The key difference is the use of the `SeqIO.parse()` function rather than `SeqIO.read()`.

```
>>> u2 = SeqIO.parse('unknown2.fas', 'fasta')
>>> type(u2)
<type 'generator'>
>>> s1 = u2.next()
>>> type(s1)
<class 'Bio.SeqRecord.SeqRecord'>
>>> s1
SeqRecord(seq=Seq('ATGTCATCAAAACCTGATACTGGTTCGGA
```

```
AATTTCTGGCCCTCAGCGACAGGAA...TGA', SingleLetterAlphabet()),
id='YJL005W', name='YJL005W', description='YJL005W', dbxrefs=[])
>>> s1.seq
Seq('ATGTCATCAAAACCTGATACTGGTTCGGAAATTTCTGGCC
CTCAGCGACAGGAA...TGA', SingleLetterAlphabet())
>>> s2 = u2.next()
>>> s2
SeqRecord(seq=Seq('ATGTCATCAAATCATGCTATTAGTCCAGAA
ACTTCTGGCTCTCATGAGCAACAA...TGA', SingleLetterAlphabet()),
id='MIT_Sbay_c342_13338', name='MIT_Sbay_c342_13338',
description='MIT_Sbay_c342_13338', dbxrefs=[])
>>> s3 = u2.next()
>>> s4 = u2.next()
>>> s5 = u2.next()
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
/Users/pmagwene/Desktop/tmp/<ipython console> in <module>()
StopIteration:
```

In this case the `SeqIO.parse` function returns an object that has *iterator* semantics (technically it's a 'generator' but this is a technical difference that you can ignore for now). An iterator is an object that 'acts like' a sequence (e.g. a list or tuple), but there are some major differences. The most important one is that an iterator does not have to compute the entire sequence at once. In the case of the `SeqIO.parse()` function that means that if you have a FASTA file with thousands of sequence entries it wouldn't try to suck them all into memory. The `.next()` method is used to call successive sequence entries in the FASTA file. When you call `.next()` on the iterator(generator) instance you get back `SeqRecords`, one at a time. However, as the lost call demonstrates if there is no 'next' item in the iterator it raises a `StopIteration` exception. For more info about iterators and generators see Norman Matloff's Tutorial on Python Iterators and Generators.

The steps for reading a FASTA sequence file can be wrapped up in the following function. We'll place each of the functions we develop in a module called `pipeline.py`. As you progress through the pipeline design you will add additional functions to this module.

```
# pipeline.py -- a simple bioinformatics pipeline
from Bio import SeqIO

def read_fasta(infile):
    """Read a single sequence from a FASTA file"""
    rec = SeqIO.read(infile,'fasta')
    return rec

def parse_fasta(infile):
    """Read multiple sequences from a FASTA file"""
    recs = SeqIO.parse(infile,'fasta')
    return [i for i in recs]
```

List comprehensions

The `parse_fasta()` function above introduces another new concept called *list comprehensions*. A list comprehension is a compact way of applying a function to each element in

a sequence. In this case the list comprehension implicitly called `.next()` to get all the `SeqRecords` from the generator returned by `SeqIO.parse()`. You'll recall that most functions in R works in a vector-wise manner. List comprehensions provide similar semantics for Python. Below are some simpler examples of list comprehensions. Try and predict the output of each of these before typing them in:

```
In [1]: x = [2,4,6,8,10]
In [2]: [i**2 for i in x]
Out[2]: ???
In [3]: y = ['bob', 'tab', 'rob', 'snob']
In [4]: def juvenilize(s):
   ...:     return str(s) + "by"
   ...:
In [5]: [juvenilize(i) for i in y]
Out[5]: ???
```

You can use the `read_fast()` function as follows:

```
>>> import pipeline
>>> recs = pipeline.parse_fasta('unknown2.fas')
>>> len(recs)
4
>>> [i.name for i in recs]
['YJL005W', 'MIT_Sbay_c342_13338', 'MIT_Smik_c333_12160', '
    MIT_Spar_c300_12282']
```

Note that the `parse_fasta()` function will return a list of `SeqRecords` even when there is only a single sequence in the file. In contrast, if you use the function `read_fasta()` on a FASTA file with more than one sequence it will raise an error.

Translating nucleotide sequence to a protein sequence

The next step is to translate each DNA sequence into a corresponding protein sequence. This is very easy using the `.translate()` method associated with the `Seq` class.

```
>>> recs[0].seq.translate()
Seq('MSSKPDTGSEISGPQRQEEQEQQIEQSSPTEANDRSIHDEV
PKVKKRHEQNSGH...ST*', HasStopCodon(ExtendedIUPACProtein(), '*'))
```

Note that the above code returns an object of type `Seq`. That's usually what we want if we're manipulating nucleotide or protein sequences but if we want to write our translated sequences back out into a file we need to create new `SeqRecords`. I illustrate this in the function below (add this to `pipeline.py`).

```
from Bio import Seq
from Bio import SeqRecord

def translate_recs(seqrecs):
    """ nucleotide SeqRecords -> translated protein SeqRecords """
    proteins = []
    for rec in seqrecs:
        aaseq = rec.seq.translate()
        protrec = SeqRecord.SeqRecord(aaseq, id=rec.id, name=rec.name,
                                      description=rec.description)
```

```
        proteins.append(protrec)
    return proteins
```

We can then encapsulate the whole process of converting a nucleotide FASTA file to a peptide sequence FASTA file as so:

```
def write_fasta(recs, outfile):
    ofile = open(outfile, 'w')
    SeqIO.write(recs, ofile, 'fasta')

def translate_fasta(infile, outfile):
    """ nucleotide fasta file -> protein fasta file """
    nrecs = parse_fasta(infile)
    precs = translate_recs(nrecs)
    write_fasta(precs, outfile)
```

We can then use this function from the Python interpreter like so:

```
>>> reload(pipeline)
<module 'pipeline' from '/Users/pmagwene/synchronized/pyth/pipeline.py'>
>>> pipeline.translate_fasta('unknown2.fas', 'unknown2-protein.fasta')
```

Take a moment to open the file `unknown2-protein.fasta` in a text editor to confirm that the file now hold amino acid sequences rather than nucleotide sequences.

Globbing to get multiple files of a given type

As an aside, what if we wanted to repeat this for a whole directory full of DNA sequences in separate FASTA files? Here's a function to help accomplish that task:

```
import glob

def inout_pairs(insuffix, outsuffix):
    """ Files in directory with given suffix -> list of tuples w/ (infile,
        outfile)"""
    infiles = glob.glob('*'+insuffix)
    pairs = []
    for infile in infiles:
        inprefix = infile[:-len(insuffix)]
        outfile = inprefix + outsuffix
        pairs.append((infile,outfile))
    return pairs
```

The `glob` module gives you filename 'globbing' functionality. Globbing is a means of matching specified file or pathnames; you can think about this as a simplified class of regular expressions. For example, you're probably familiar with command line searches like:

```
$ ls *.fas   # list all files with the extension .fas
$ ls unk*   # list all files that begin with 'unk'
```

The `inout_pairs()` function we defined above allows us to glob file files with the given `insuffix` and create a corresponding set of names for output files. The following illustrates this:

```
>>> pairs = pipeline.inout_pairs('.fas', '-protein.fasta')
>>> pairs
```

```
[('unknown1.fas', 'unknown1-protein.fasta'),
('unknown2.fas', 'unknown2-protein.fasta')]
>>> for (i,o) in pairs:
...     pipeline.translate_fasta(i,o)
...
...
>>> ls *.fas*  # only works in ipython
unknown1-protein.fasta  unknown1.fas  unknown2-protein.fasta  unknown2.fas
```

Note that I changed the file suffix from `.fas` to `.fasta` on the output files. This isn't necessary but I find that doing so makes it easy to sort through large directories to distinguish generated files from the original files. The `inout_pairs()` function will come in handy when we combine our functions to generate a multi-sequence pipeline.

BLAST searches via the NCBI server

We can use Biopython do network based BLAST searches. Here we will use blastp to search against protein sequences in the Swiss-Prot database.

```
>>> from Bio.Blast import NCBIWWW, NCBIXML
>>> prot1 = pipeline.read_fasta('unknown1-protein.fasta')
>>> results_handle = NCBIWWW.qblast('blastp','swissprot',prot1.seq.tostring
    (), entrez_query='(Homo sapiens[ORGN])')
>>> results = results_handle.read()
>>> sfile = open('prot1_blast.out','w')
>>> sfile.write(results)
>>> sfile.close()
>>> blast_out = open('prot1_blast.out','r')
>>> brec = NCBIXML.read(blast_out)
>>> brec
<Bio.Blast.Record.Blast instance at 0x2ec22d8>
>>> len(brec.alignments) # we got 50 blast hits in the
50
>>> brec.alignments[0]
<Bio.Blast.Record.Alignment instance at 0x2ec23a0>
>>> brec.alignments[0].accession
u'P31749'
```

This code introduces another concept we'll call the *Producer-Consumer* pattern. The Producer-Consumer pattern is a general programming concept, but the key here is that the pattern generalizes the problem of parsing complex biological data types. The producer does the work of getting the information from a file (or from the web in this case). The consumer process the information into a form we can use. In the code above the function `NCBIWWW.qblast()` is the producer and `NCBIXML.read()` plays the role of the consumer. This pattern is used over and over again in Biopython so you should spend some time trying to understand the general idea. See the Biopython tutorial for a more complete discussion.

Our BLAST query returned the information in the form of XML data. XML stands for 'Extensible Markup Language', and is a generic way to encode documents in machine-readable form. XML data is usually plain text – go ahead and open up the file `prot1_blast.out` in a text editor to see the output. Since XML is a generic format, specific types of XML documents need a 'schema' or 'grammar' that specifies how the document is to be read and interpretted.

In the example above, the module `NCBIXML` knows how to handle XML data returned from NCBI, hence our use of the function `NCBIXML.read()`.

In the example given, we limited our query to sequences from humans. If we wanted to include all metazoan sequences we could pass `'(Metazoa[ORGN])'` as the argument to `entrez_query`. If we didn't want to limit our search at all we would simply not include that argument (i.e. accept the default). The BLAST output is fairly complicated. See the BioPython tutorial section 7.5 for a complete breakdown of all the fields in the BLAST output.

Again, the commands above are rather involved so let's wrap them up in a function:

```
from Bio.Blast import NCBIWWW, NCBIXML

def blastp(seqrec, outfile, database='nr', entrez_query='(none)'):
    handle = NCBIWWW.qblast('blastp', database, seqrec.seq.tostring(),
                            entrez_query=entrez_query)
    results = handle.read()
    sfile = open(outfile, 'w')
    sfile.write(results)
    sfile.close()
    bout = open(outfile, 'r')
    brecord = NCBIXML.read(bout)
    return brecord

def summarize_blastoutput(brecord):
    hits = []
    for alignment in brecord.alignments:
        expect = alignment.hsps[0].expect
        accession = alignment.accession
        hits.append((expect,accession))
    hits.sort() # will sort tuples by their first value (i.e. expect)
    return hits
```

We can use this code as follows:

```
>>> humanblast = pipeline.blastp(prot1, 'prot1-hum-blast.out', database='
    swissprot', entrez_query='(Homo sapiens[ORGN])')
>>> flyblast = pipeline.blastp(prot1, 'prot1-fly-blast.out', database='
    swissprot', entrez_query='(Drosophila melanogaster[ORGN])')
>>> humanhits = pipeline.summarize_blastoutput(humanblast)
>>> flyhits = pipeline.summarize_blastoutput(flyblast)
>>> humanhits[0] # the first number is the E-value for the BLAST search
(6.0329099999999998e-84, u'P31749')
>>> print humanhits[0][1]  # prints the swissprot accession number
P31749
>>> flyhits[0]
(3.5325700000000003e-86, u'Q8INB9')
```

Go to the UniProt website and use the search box to lookup those accession numbers.

Getting records from Swiss-Prot

For a small number of accession numbers it's easy to use the web interface to UniProt (Swiss-Prot). For hundred of blast hits that's just not an option. Conveniently, we can use

Biopython to query the Swiss-Prot database to retrieve information about these presumed orthologs. You can access the Swiss-Prot database as follows:

```
>>> from Bio import ExPASy
>>> from Bio import SwissProt
>>> handle1 = ExPASy.get_sprot_raw(humanhits[0][1]) # access with the
    accession number
>>> rec1 = SwissProt.read(handle1)
>>> print rec1.description
RecName: Full=RAC-alpha serine/threonine-protein kinase; EC=2.7.11.1;
    AltName:
Full=RAC-PK-alpha; AltName: Full=Protein kinase B; Short=PKB; AltName: Full
    =C-
AKT;
>>> rec1.comments[0]
"FUNCTION: General protein kinase capable of phosphorylating several known
proteins. Phosphorylates TBC1D4. Signals downstream of phosphatidylinositol
     3-
kinase (PI(3)K) to mediate the effects of various growth factors such as
    platelet-
... output truncated ..."
>>> print dir(rec1) # lets see what other attributes the record has
['__doc__', '__init__', '__module__', 'accessions', 'annotation_update', '
    comments', 'created', 'cross_references', 'data_class', 'description',
    'entry_name', 'features', 'gene_name', 'host_organism', '
    host_taxonomy_id', 'keywords', 'molecule_type', 'organelle', 'organism'
    , 'organism_classification', 'references', 'seqinfo', 'sequence', '
    sequence_length', 'sequence_update', 'taxonomy_id']
>>> print rec1.gene_name
Name=AKT1; Synonyms=PKB, RAC;
>>> print rec1.sequence[:25] # first 25 amino acids
MSDVAIVKEGWLHKRGEYIKTWRPR
```

Here's some functions to make this more convenient:

```
from Bio import ExPASy
from Bio import SwissProt

def get_swissrec(accession):
    handle = ExPASy.get_sprot_raw(accession)
    record = SwissProt.read(handle)
    return record

def swissrec2seqrec(record):
    seq = Seq.Seq(record.sequence, Seq.IUPAC.protein)
    s = SeqRecord.SeqRecord(seq, description=record.description,
                id=record.accessions[0], name=record.entry_name)
    return s
```

And here is an example of how we can apply these functions:

```
>>> reload(pipeline)
>>> ids = [humanhits[0][1], flyhits[0][1]]
```

```
>>> ids
[u'P31749', u'Q8INB9']
>>> swissrecs = [pipeline.get_swissrec(i) for i in ids]
>>> seqs = [pipeline.swissrec2seqrec(i) for i in swissrecs]
>>> seqs[0]
SeqRecord(seq=Seq('MSDVAIVKEGWLHKRGEYIKTWRPRYFLLKNDGTFIGYKERPQDVDQREAPLNN
    ...GTA', IUPACProtein()), id='P31749', name='AKT1_HUMAN', description='
    RecName: Full=RAC-alpha serine/threonine-protein kinase; EC=2.7.11.1;
    AltName: Full=Protein kinase B; Short=PKB; AltName: Full=Protein kinase
     B alpha; Short=PKB alpha; AltName: Full=Proto-oncogene c-Akt; AltName:
     Full=RAC-PK-alpha;', dbxrefs=[])
>>> seqs[1]
SeqRecord(seq=Seq('MNYLPFVLQRRSTVVASAPAPGSASRIPESPTTTGSNIINIIYSQSTHPNSSPT
    ...SMQ', IUPACProtein()), id='Q8INB9', name='AKT1_DROME', description='
    RecName: Full=RAC serine/threonine-protein kinase; Short=DAkt; Short=
    DRAC-PK; Short=Dakt1; EC=2.7.11.1; AltName: Full=Akt; AltName: Full=
    Protein kinase B; Short=PKB;', dbxrefs=[])
>>> seqs.append(prot1)  # add our original protein sequence to the list
>>> pipeline.write_fasta(seqs, 'unknown1-plus-human-fly.fasta')
```

Multiple sequence alignment via MAFFT

We've now generated a new FASTA file that includes our original protein sequence and the sequences for the human and fly BLAST best hits. We will use MAFFT to perform a multiple alignment. Biopython has built in code to simplify command line usage of common alignment programs like CLUSTALW, MAFFT, and MUSCLE. However I'll show you how to do this with your own code using the `subprocess` module. Knowing how the `subprocess` module works is useful because it allows you to interface with any command line program from within Python.

The `subprocess` module allows your Python code to start other programs (child processes) and send/get input and output from those same processes. When we use the subprocess module we're putting the Unix design element of 'Everything is a file or process' to use. Here's a simple example:

```
>>> import subprocess
>>> subprocess.call(["ls","-l"])
# on windows the equivalent command is
# subprocess.call(["dir",],shell=True)
total 11696
-rw-r--r--   1 pmagwene  staff    93514 Nov 22 19:36 prot1-fly-blast.out
-rw-r--r--   1 pmagwene  staff   109635 Nov 22 19:35 prot1-hum-blast.out
-rw-r--r--   1 pmagwene  staff   109635 Nov 22 19:19 prot1_blast.out
-rw-r--r--   1 pmagwene  staff     2308 Nov 22 20:07 unknown1-plus-human-
    fly.fasta
-rw-r--r--   1 pmagwene  staff      854 Nov 22 16:46 unknown1-protein.fasta
-rwx------   1 pmagwene  staff     2535 Nov 22 15:38 unknown1.fas
-rw-r--r--@  1 pmagwene  staff    24849 Nov 22 16:25 unknown2.fas
-rw-r--r--   1 pmagwene  staff     8331 Nov 22 16:46 unknowns-protein.fasta
```

The above code uses a convenience function `call()` in the `subprocess` module. We'll use the same function to run MAFFT:

```
import subprocess

def mafft_align(infile, outfile):
    ofile = open(outfile,'w')
    retcode = subprocess.call(["mafft",infile], stdout=ofile)
    ofile.close()
    if retcode != 0:
        raise Exception("Possible error in MAFFT alignment")
```

And we put it to use as follows:

```
In [8]: reload(pipeline)
In [8]: pipeline.mafft_align('unknown1-plus-human-fly.fasta', 'unknown1-
    alignment.fasta')
```

If all went well this should have created the file `unknown1-alignment.fasta` in your directory. Open this alignment using JalView to examine the alignment in more detail.

Searching for protein domains using HMMER and Pfam

As the final step of our pipeline we'll use HMMER and the Pfam database to search for known protein domains in our original protein. This assumes you have the HMMER binaries and Pfam database installed as demonstrated in last weeks exercises and that you've already run `hmmpress` against the Pfam database. Again we write a small wrapper function using the `subprocess` module. This time we'll use the `Popen` class to illustrate how we can capture the output produced by `hmmpfam`. Note that if you haven't installed the HMMER binaries to one of the standard locations you might need to specify the full path to the `hmmscan` executable in the code below.

```
def hmmer_pfam(infilename, outfilename, pfamdb):
    pipe = subprocess.Popen(["hmmscan", pfamdb, infilename],
            stdout=subprocess.PIPE).stdout
    output = pipe.read() # this gives us the output of our command
    outfile = open(outfilename, 'w')
    outfile.write(output)
    outfile.close()
```

This function can be called like this:

```
# change the last argument to match the path to your Pfam database.
>>> pipeline.hmmer_pfam('unknown1-protein.fasta', 'unknown1-domains.out', '
    /Users/pmagwene/tmp/Pfam-A.hmm')
```

As before this search may take several minutes.

Putting it all together

We've generated a variety of functions that take care of the major steps of our pipeline. It's time to put the steps together to automate the entire process.

```
def oneseq_pipeline(infilename, pfamdb=None,
                    compareto=['Homo sapiens','Drosophila melanogaster'],
                    skipHMMER = True,extension="XX"):
    # translate nucleotide sequence to protein seq
    protout = 'protein-' + infilename + extension
```

```
            # add the extension so all generated files have
            # different extension than input files

    translate_fasta(infilename, protout)

    # run blastp on protein sequence against swissprot and extract best
        hits
    protrec = parse_fasta(protout)[0]
    blastout ='blast-' + protout
    besthitids = []
    for organism in compareto:
        equery = '(%s[ORGN])' % organism # create the entrez organism query
        brecord = blastp(protrec, blastout, database='swissprot',
            entrez_query=equery)
        bhits = summarize_blastoutput(brecord)
        besthitids.append(bhits[0][1])

    # download corresponding records from Swiss-Prot
    swissrecs = [get_swissrec(i) for i in besthitids]
    seqs = [swissrec2seqrec(i) for i in swissrecs]
    seqs.append(protrec)

    # write FASTA file with best hits plus original protein sequence
    plusout = 'blasthits-' + protout + '.XML'
    write_fasta(seqs, plusout)

    # do multiple alignment via mafft
    mafft_align(plusout, 'aligned-' + protout)

    # search for domains via HMMER/Pfam
    if not skipHMMER:
        if pfamdb is not None:
            hmmerout = 'hmmer-' + protout
            hmmer_pfam(protout, hmmerout, pfamdb)
```

Our function can take as input a FASTA file with a single sequence or with multiple sequences. In the case of a multiple sequences it assumes that the 'target' sequence for the search is the first sequence in the file. Also, note the `skipHMMER` argument included in the function. The HMMER search takes a relatively long time and doing it sequence by sequence is not very efficient so by default the pipeline will skip this step. If you want to include the HMMER step than specify the Pfam database file and set `skipHMMER=False`.

Testing out the pipeline

To test out the function we do:

```
>>> reload(pipeline)
>>> pipeline.oneseq_pipeline('unknown1.fas')
```

This will create four new FASTA files:

1) `protein-unknown1.fasXX`
2) `blast-protein-unknown1.fasXX.XML`

3) `blasthits-protein-unknown1.fasXX`
4) `aligned-protein-unknown1.fasXX`

These respectively contain:

1) the amino acid sequence translated from the nucleotide sequence given as input
2) the XML output of the qblast query to NCBI
3) the amino acid sequences for the BLAST hits returned from NCBI
4) the MAFFT multiple alignment of the protein sequences.

Let's now test the pipeline using an alternate set organisms:

```
>>> pipeline.oneseq_pipeline('unknown1.fas', compareto=["Homo sapiens","Mus
    musculus","Caenorhabditis elegans"])
```

For completeness let's also test the pipeline with the HMMER step included:

```
>>> pipeline.oneseq_pipeline('unknown1.fas', '/home/pmagwene/tmp/Pfam-A.hmm
    ',
skipHMMER=False)
```

Extending the pipeline to deal with multiple inputs

Now that we're confident out single sequence pipeline function works it can be easily adapted to deal with multiple input files:

```
def multiseq_pipeline(inext, pfamdb=None,
                compareto=['Homo sapiens','Drosophila melanogaster'],
                skipHMMER=True):
    inout = inout_pairs(inext, 'XX')
    infiles = [i[0] for i in inout]
    for filename in infiles:
        print "Processing %s" % filename
        oneseq_pipeline(filename, pfamdb, compareto, skipHMMER)
```

To test the complete multi-sequence pipeline delete all the generated files (so that only `unknown1.fas` and `unknown2.fas` are in the unknowns directory) and try the following:

```
>>> pipeline.multiseq_pipeline('.fas')
```

Given our example data this function will process just two input files. However, you can add an arbitrary number of additional '.fas' files to the directory and the pipeline will process those as well with exactly the same command.

There are a number of ways the pipeline could be sped up. One obvious improvement would be to utilize a local installation of BLAST and the respective databases. However, optimization is often a complex task. The pipeline we developed here doesn't require us to install BLAST (which can be somewhat involved) and provides adequate performance for a modest number of sequences. It is possible to turn this set of Python functions into a program that you could run from the command line (rather than the Python interpeter) just like any other Unix program.

The pipeline.py module

The pages that follow give the complete code listing for the `pipeline.py` module.

```
"""
pipline.py --  An illustrative example of a bioinformatics pipeline.
Requires Python 2.6+ and BioPython 1.53+
(c) Copyright by Paul M. Magwene, 2009-2011  (mailto:paul.magwene@duke.edu)
"""

from Bio import Seq, SeqIO, SeqRecord
from Bio import ExPASy, SwissProt
from Bio.Blast import NCBIWWW, NCBIXML

import glob, subprocess

def read_fasta(infile):
    """Read a single sequence from a FASTA file"""
    rec = SeqIO.read(infile,'fasta')
    return rec

def parse_fasta(infile):
    """Read multiple sequences from a FASTA file"""
    recs = SeqIO.parse(infile,'fasta')
    return [i for i in recs]

def write_fasta(recs, outfile):
    ofile = open(outfile, 'w')
    SeqIO.write(recs, ofile, 'fasta')

def translate_recs(seqrecs):
    """ nucleotide SeqRecords -> translated protein SeqRecords """
    proteins = []
    for rec in seqrecs:
        aaseq = rec.seq.translate()
        protrec = SeqRecord.SeqRecord(aaseq, id=rec.id, name=rec.name,
                                      description=rec.description)
        proteins.append(protrec)
    return proteins

def translate_fasta(infile, outfile):
    """ nucleotide fasta file -> protein fasta file """
    nrecs = parse_fasta(infile)
    precs = translate_recs(nrecs)
    write_fasta(precs, outfile)

def inout_pairs(insuffix, outsuffix):
    """ Files in directory with given suffix -> list of tuples w/ (infile,
        outfile)"""
    infiles = glob.glob('*'+insuffix)
    pairs = []
    for infile in infiles:
        inprefix = infile[:-len(insuffix)]
        outfile = inprefix + outsuffix
        pairs.append((infile,outfile))
```

```
    return pairs

def blastp(seqrec, outfile, database='nr', entrez_query='(none)'):
    handle = NCBIWWW.qblast('blastp', database, seqrec.seq.tostring(),
                                entrez_query=entrez_query)
    results = handle.read()
    sfile = open(outfile, 'w')
    sfile.write(results)
    sfile.close()
    bout = open(outfile, 'r')
    brecord = NCBIXML.read(bout)
    return brecord

def summarize_blastoutput(brecord):
    hits = []
    for alignment in brecord.alignments:
        expect = alignment.hsps[0].expect
        accession = alignment.accession
        hits.append((expect,accession))
    hits.sort() # will sort tuples by their first value (i.e. expect)
    return hits

def get_swissrec(accession):
    handle = ExPASy.get_sprot_raw(accession)
    record = SwissProt.read(handle)
    return record

def swissrec2seqrec(record):
    seq = Seq.Seq(record.sequence, Seq.IUPAC.protein)
    s = SeqRecord.SeqRecord(seq, description=record.description,
                id=record.accessions[0], name=record.entry_name)
    return s

def mafft_align(infile, outfile):
    ofile = open(outfile,'w')
    retcode = subprocess.call(["mafft",infile], stdout=ofile)
    ofile.close()
    if retcode != 0:
        raise Exception("Possible error in MAFFT alignment")

def hmmer_pfam(infilename, outfilename, pfamdb):
    pipe = subprocess.Popen(["hmmscan", pfamdb, infilename],
            stdout=subprocess.PIPE).stdout
    output = pipe.read() # this gives us the output of our command
    outfile = open(outfilename, 'w')
    outfile.write(output)
    outfile.close()

def oneseq_pipeline(infilename, pfamdb=None,
                    compareto=['Homo sapiens','Drosophila melanogaster'],
                    skipHMMER = True,extension="XX"):
```

```
    # translate nucleotide sequence to protein seq
    protout = 'protein-' + infilename + extension
                # add the extension so all generated files have
                # different extension than input files

    translate_fasta(infilename, protout)

    # run blastp on protein sequence against swissprot and extract best
        hits
    protrec = parse_fasta(protout)[0]
    blastout ='blast-' + protout + '.XML'
    besthitids = []
    for organism in compareto:
        equery = '(%s[ORGN])' % organism # create the entrez organism query
        brecord = blastp(protrec, blastout, database='swissprot',
            entrez_query=equery)
        bhits = summarize_blastoutput(brecord)
        besthitids.append(bhits[0][1])

    # download corresponding records from Swiss-Prot
    swissrecs = [get_swissrec(i) for i in besthitids]
    seqs = [swissrec2seqrec(i) for i in swissrecs]
    seqs.append(protrec)

    # write Fasta file with best hits plus original protein sequence
    plusout = 'blasthits-' + protout
    write_fasta(seqs, plusout)

    # do multiple alignment via mafft
    mafft_align(plusout, 'aligned-' + protout)

    # search for domains via HMMER/Pfam
    if not skipHMMER:
        if pfamdb is not None:
            hmmerout = 'hmmer-' + protout
            hmmer_pfam(protout, hmmerout, pfamdb)

def multiseq_pipeline(inext, pfamdb=None,
                compareto=['Homo sapiens','Drosophila melanogaster'],
                skipHMMER=True):
    inout = inout_pairs(inext, 'XX')
    infiles = [i[0] for i in inout]
    for filename in infiles:
        print "Processing %s" % filename
        oneseq_pipeline(filename, pfamdb, compareto, skipHMMER)
```