BIO680 - Introduction to Ecological Analysis in R (1 credit)

Instructors: Matthew Lau (and Amy Whipple)

November 4 - December 2 (FALL 2008)

Tuesdays 12pm - 3pm

Room 413

PURPOSE: Statistical analysis is the cornerstone of ecology. The statistical programming language, R, provides a flexible (and free) platform for nearly all aspects of statistical analysis: including modeling, data transformations, plotting and statistical tests. The purpose of this course is to get you programming in R by doing three things. First, I will go over the basics of programming in R. Second, I will cover how to conduct common ecological analyses. Last, and perhaps most important, I will help you develop a knowledge base and a strategy that will allow you to program or learn how to program your own analyses beyond the scope of this class.

It is highly recommended that everyone has a working knowledge of statistical methods; however, no R or other programming experience is required.

MATERIALS:

Required - access to a computer

Recommended - please visit the Comprehensive R Archive Network (CRAN - http://cran.r-project.org/) and go to the manuals and FAQ section

OUTLINE:

Day 1: Introduction, math operations & t-test

Day 2: Data import, Regression, ANOVA, ANCOVA, basic plotting & packages

Day 3: Community analyses

Day 4: Finish community analyses and begin working with your own data

Day 5: Working with your own data OR advanced topics (more standard analyses, AIC or Bayes)

GRADING: PASS/FAIL

OFFICE: PS125 (I will be available on Weds from 1-3pm and by appointment)

CONTACT: mkl48@nau.edu

Introduction to Ecological Analysis in **R** (Day 1)

Matthew Lau

November 4, 2008

Today, we covered lot of ground with the goal of providing the tools to input and manipulate data, conduct analyses and make plots. Below, I have detailed all of the activities that we did in R.

1. Scripting
2. Math Operations
3. Help!
4. Data Entry
5. Statistical Analyses
6. Plotting
7. Next Class

1 Scripting

Perhaps the most important thing we learned above everything else is to use a script editor. Do not work solely in the console command line. Please, open a new script file to work in by going to the file menu and selecting a "New Document". This document can be saved for future use, unlike the R Console, which will save what you have done, but not in a reproducible format.

When you script you can write out your code and then run it by placing the cursor on the line or highlighting all of the script you want to run and running it (Windows users press `CTRL + R` and Mac users press `COMMAND or APPLE + ENTER`). You can add notes by using the pound symbol, #, which tells R not to run any text to the right of that symbol on that line.

2 Math Operations

Math operations are basic and intuitive, but are essential tools for working effectively in R.

2.1 Basic Math Operations:

```
> 1 + 1

[1] 2

> 1 - 1

[1] 0

> 2 * 2

[1] 4

> 2/2

[1] 1

> 2^2

[1] 4

> (2 + 2) * 2

[1] 8
```

2.2 Math Commands

Commands (aka. functions) are the meat of R. They allow us to simplify tasks. The basic structure of a command is `command`(*arguments*). Here we can see the mechanics of any command. Simply, the `command` tells R what you want to be done to what's in the parentheses (i.e. the *arguments*).

```
> sqrt(2)

[1] 1.414214

> log(2)

[1] 0.6931472

> cos(2)

[1] -0.4161468
```

There also several statistically relevant math commands that I'll just mention now, but we'll use them later: `mean`(), `sd`() and `length`() return the mean, standard deviation and number of elements of a vector of numbers.

3 HELP!

You may have already encountered some issues just with trying to do these simple operations and commands. In order to get around obstacles and improve your R knowledge-base it is important to know how to access help resources. There are many books out there including the R "bible", many of which are listed on the R website. I typically rely on two resources, the internal help files within R and internet search engines. You can reach help within R in a number of ways, but one of the easiest is to use the `help()` command. For example:

```
> help(sqrt)
```

This will bring up a help file with information pertaining to the `sqrt()` and other related commands.

4 Data Entry

Handling data is vital to successful analyses. Bringing data into R can be done in at least two ways. First, you can enter data manually, not unlike a calculator, and second, you get input data via file reading commands.

4.1 Entering Data Manually: Making Vectors and Matrixes

R will support many different data formats. Here is how you can create vectors and matrixes.

```
> 1:10

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

> c(1.1, 25, 3.32, 4, 12, 14, 85)

[1]  1.10 25.00  3.32  4.00 12.00 14.00 85.00

> matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3,
+     ncol = 3)

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> array(c(1, 2, 3, 4, 5, 6, 7, 8, 9), c(3, 3))

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

4.2 Reading Data from File

More often than not, we already have our data saved as a data file. Here are two commands that you can use to bring data into R. Note that both of these commands require the data to be in a specific format. Check the help file for the commands to see how the data should be structured.

`read.table`**("file location") will read a text file.**

`read.csv`**("file location") will read a .csv file. This is most useful when you have your data organized as spreadsheets, which many people do. For example:**

```
> read.csv("/Users/artemis/Desktop/rock_lichen_data.csv")
```

```
   moth canopy    A
1     2   14.3  7.2
2     2   28.5  2.8
3     2   34.9  4.6
4     2   43.6  5.2
5     2  169.9 25.4
6     2   38.0  0.0
7     2  102.4  9.7
8     2   64.8  9.9
9     2   39.2  1.8
10    2   39.2  4.7
11    1  320.0 39.9
12    1  206.4 58.0
13    1  194.9 39.1
14    1  135.2 18.3
15    1   89.8 12.4
16    1  417.7 44.1
17    1  164.5 24.1
18    1  155.7 34.9
19    1  155.1 43.1
20    1  169.0 19.0
```

This is the standard formatting for the `read.csv` command. Data are organized into columns with the columns headed by their appropriate names. Column names should not contain spaces or math operations and should not start with numbers.

4.3 Making Objects

Here you may have already noticed a missing peace of the puzzle. Although we can create these nifty vectors and matrixes, R simply spits them out and thats it. It would be very useful to be able to call and use them again. We can use objects to do this for us. Just as we can use symbols in math to represent numbers (e.g. a + b = c),

we can use different symbols to represent or "store" our data. This is done by using either the equals sign "=" or by making a left arrow with the less-than sign "<" and a dash "-". For instance:

```
> a = 10
> a

[1] 10

> a <- 10
> a

[1] 10
```

You can see that the first line above tells R that "a" is the number 10, because when we enter "a" R returns the number 10. Note that object names: a) cannot start with a number, b) are case sensitive and c) cannot contain math operators (e.g. + , - , / , etc.).

We can also make much more complex objects with vectors and matrixes, as well as data frames, data tables and lists. For example, we can create an object from a data matrix read in from a file:

```
> data <- read.csv("/Users/artemis/Desktop/rock_lichen_data.csv")
> data

   moth canopy    A
1     2   14.3  7.2
2     2   28.5  2.8
3     2   34.9  4.6
4     2   43.6  5.2
5     2  169.9 25.4
6     2   38.0  0.0
7     2  102.4  9.7
8     2   64.8  9.9
9     2   39.2  1.8
10    2   39.2  4.7
11    1  320.0 39.9
12    1  206.4 58.0
13    1  194.9 39.1
14    1  135.2 18.3
15    1   89.8 12.4
16    1  417.7 44.1
17    1  164.5 24.1
18    1  155.7 34.9
19    1  155.1 43.1
20    1  169.0 19.0
```

5 Statistical Analyses

5.1 t-test

A t-test is a very common parametric analysis of ecological data, when we are have one or two samples that we would like to analyze. Here we looked at an example where we were interested in testing whether or not habitat restoration had improved the abundance of tigers in a preserve with a set of tiger counts of tigers taken once a month for six months. We did a t-test in two different ways. The first way we used a command that comes in the R base package, `t.test()`:

```
> tigers <- c(10, 14, 11, 12, 10, 18)
> t.test(tigers)

        One Sample t-test

data:  tigers
t = 9.934, df = 5, p-value = 0.0001765
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
  9.265422 15.734578
sample estimates:
mean of x
     12.5
```

This conducts a t-test using the default settings (see the help file) of the population mean different from zero. One problem is that it is a two-tail test, which we don't want (counts can never be less than zero). So we can re-run it specifying the argument "alternative" as "greater" for a one-tail test of the mean greater than zero:

```
> t.test(tigers, alternative = "greater")

        One Sample t-test

data:  tigers
t = 9.934, df = 5, p-value = 8.823e-05
alternative hypothesis: true mean is greater than 0
95 percent confidence interval:
 9.964453      Inf
sample estimates:
mean of x
     12.5
```

Notice that "greater" is inside of quotes. This tells R that it is text, not an object.

We also programmed our own t-test from scratch using what we new about the mechanics of the t-test. First we calculated our observed t, and then we computed a p-value for that t:

```
> tobs <- (mean(tigers) - 0)/(sd(tigers)/sqrt(length(tigers)))
> pt(tobs, df = length(tigers) - 1, lower.tail = FALSE)

[1] 8.82312e-05
```

Here the `mean()` and `sd()` commands are computing our mean and standard deviations for us. The `length()` command yields the number of elements in our data vector "tigers" which is also our sample size (i.e. n). The `pt()` command gives us our p-value. The "lower.tail" argument specification gives us the area under the t distribution to the right of our observed t.

We see that we get both an observed t and p-value exactly equal to the results from our command, `t.test`.

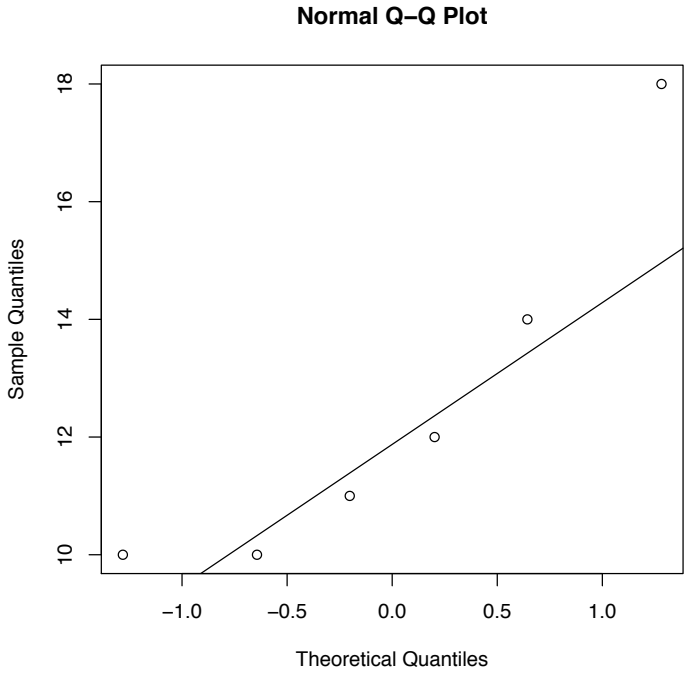
5.2 Checking Assumptions

There are two assumptions of a t-test. The first, that the sample is representative of the population, is perhaps the most important but hard to establish. The second, that the population is normally distributed, can easily be checked with several simple diagnostics: the Shapiro-Wilks test, quantile-quantile (QQ) plotting and histogram. Here is the code to get these three tools:

```
> shapiro.test(tigers)

        Shapiro-Wilk normality test

data:  tigers
W = 0.8502, p-value = 0.158
```

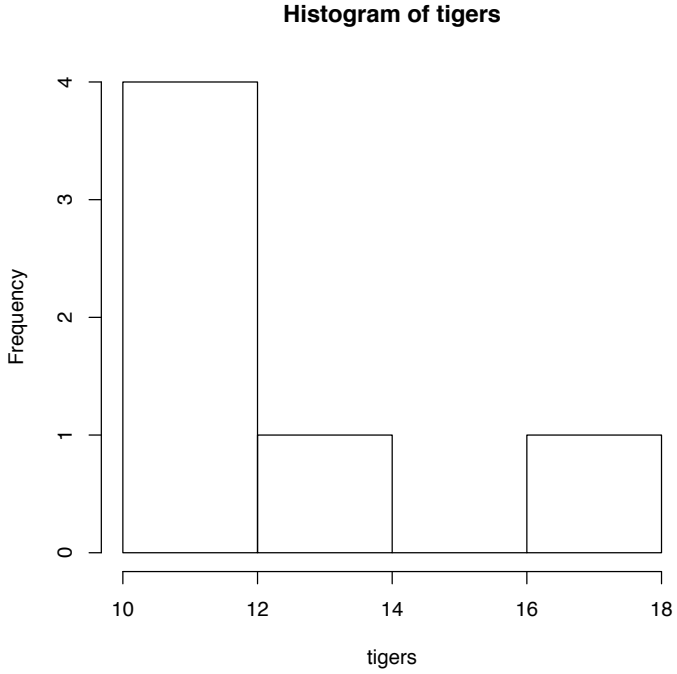
This command, `shapiro.test`, runs a Shapiro-Wilks test with the null-hypothesis that the data were drawn from a normal distribution.

```
> qqnorm(tigers)
> qqline(tigers)
```



The `qqnorm` command produces a QQ plot and the command `qqline` draws a one to one line on top of the points. Note that you must first produce the QQ plot before drawing the QQ line.

```
> hist(tigers)
```



This command is self-explanatory. It produces a histogram of the frequency of various data values.

The Shapiro-Wilks test does not refute the hypothesis of normality, both the QQ plot and the histogram show deviations from normality. In this case, since we have such a small sample size and likely have low power to reject the null hypothesis of normality in the Shapiro-Wilks test, we should go with the evidence from the QQ plot and the histogram that suggest non-normality.

5.3 Non-parametric Test: Wilcoxon Signed-Rank Test

Although the t-test has been shown to be robust to non-normality, we know that count data are not usually normally distributed. Typically they follow a poisson distribution. So, regardless of the conclusions we drew from our normality investigations, it would be prudent to use a test that does not make an assumption of the underlying population distribution. In this case, where we are conducting inference about a single sample, we can use the Wilcoxon Signed-Rank Test. This is easily done by using the `wilcox.test` command from the R base package.

```
> wilcox.test(tigers, alternative = "greater")

        Wilcoxon signed rank test with continuity
        correction
```

```
data:  tigers
V = 21, p-value = 0.01776
alternative hypothesis: true location is greater than 0
```

Here we see that the results, although different, yield the same answer as our t-tests.

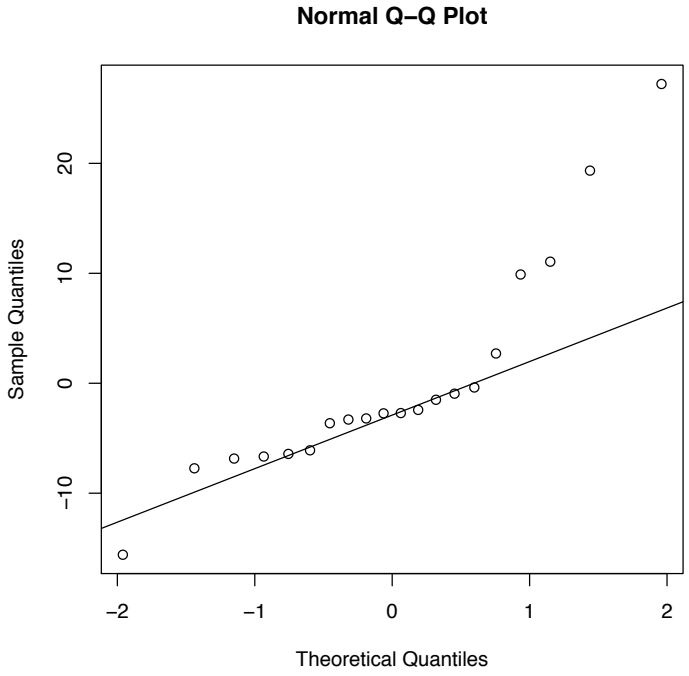
5.4 Regression

In R there are two steps to running a regression. First we need to fit a model. Here we can use the `lm` command to fit a least squares regression model to the data that we loaded earlier:

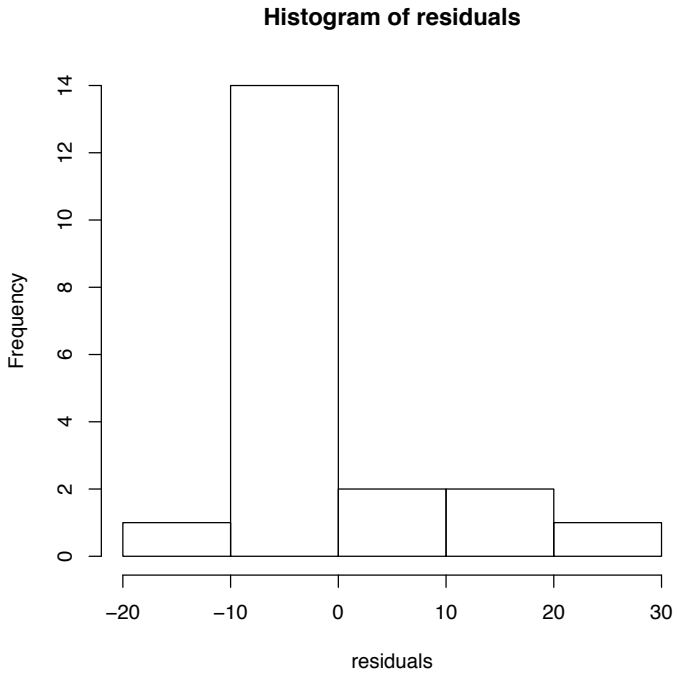
```
> data
> attach(data)

> lm(A ~ canopy)

Call:
lm(formula = A ~ canopy)

Coefficients:
(Intercept)       canopy
     2.5353       0.1368

> fit <- lm(A ~ canopy)
```

Notice that we used the `attach` command. This tells R to make objects from the columns. This object consists of a vector named by the column name. This allows us to use these objects in our `lm` command. Notice the structure of the argument here. This is our model specification. It consists of the response variable (A), a tilde "~" signifying regression and our predictor variable (canopy). The `lm` command fits the model and generates the typical output that you need to conduct the regression analysis. Thus, we save this output to a new object, fit. This will contain different components within it: such as the fitted values and the residuals. We can access these components using a dollar sign "$". For instance, we can pull our residuals out of this output to check the assumption of the residuals being normally distributed:

```
> residuals <- fit$residuals
> shapiro.test(residuals)

        Shapiro-Wilk normality test

data:  residuals
W = 0.843, p-value = 0.004082

> qqnorm(residuals)
> qqline(residuals)
```




```
> hist(residuals)
```



Now we can now use this information to conduct a regression analysis (i.e. a hypothesis test for the slope different from zero), using the `summary` command:

```
> summary(fit)

Call:
lm(formula = A ~ canopy)

Residuals:
     Min       1Q   Median       3Q      Max
-15.5971  -6.1815  -2.7243   0.3875  27.2191

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  2.53526    3.69067   0.687    0.501
canopy       0.13685    0.02246   6.092 9.33e-06 ***
---
Signif. codes:  0 âĂŸ***âĂŹ 0.001 âĂŸ**âĂŹ 0.01 âĂŸ*âĂŹ 0.05 âĂŸ.âĂŹ 0.1 âĂŸ âĂŹ 1

Residual standard error: 10.2 on 18 degrees of freedom
Multiple R-squared: 0.6734,        Adjusted R-squared: 0.6553
F-statistic: 37.12 on 1 and 18 DF,  p-value: 9.332e-06
```

This is our standard F-table, which we can use to conduct our hypothesis test.

5.5 ANOVA

Analysis of Variance (ANOVA) is theoretically very similar to regression. It makes sense then that in R conducting an ANOVA is nearly identical to regression. We do only two things differently. First, we need to make sure that we specify our predictor variable, in this case "moth", as categorical. We can do this with the `factor` command. Second, we can use the `anova` command to produce an F-table that is more useful for conducting an ANOVA. Overall, the process is still very much the same:

```
> fit.anova <- lm(A ~ factor(moth))
> anova(fit.anova)

Analysis of Variance Table

Response: A
             Df Sum Sq Mean Sq F value    Pr(>F)
factor(moth)  1 3421.7  3421.7  26.599 6.612e-05 ***
Residuals    18 2315.6   128.6
---
Signif. codes:  0 âĂŸ***âĂŹ 0.001 âĂŸ**âĂŹ 0.01 âĂŸ*âĂŹ 0.05 âĂŸ.âĂŹ 0.1 âĂŸ âĂŹ 1
```

One note here is that we could have made a new object for "moth" as a factor and then used the new object in our `lm` command. It's merely a matter of personal preference, R doesn't care:

```
> moth <- factor(moth)
> fit.anova <- lm(A ~ moth)
> anova(fit.anova)
```

```
Analysis of Variance Table

Response: A
          Df Sum Sq Mean Sq F value    Pr(>F)
moth       1 3421.7  3421.7  26.599 6.612e-05 ***
Residuals 18 2315.6   128.6
---
Signif. codes:  0 âĂŸ***âĂŹ 0.001 âĂŸ**âĂŹ 0.01 âĂŸ*âĂŹ 0.05 âĂŸ.âĂŹ 0.1 âĂŸ âĂŹ 1
```

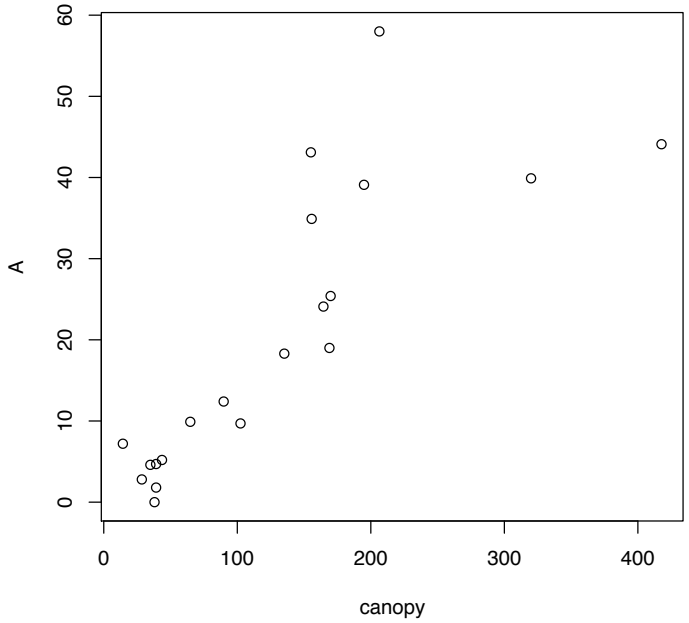
6 Plotting

Visualizing data to look for trends is an important aspect of analysis. We can use one command, `plot`, to create great plots that complement both regression and ANOVA.

6.1 Scatterplots

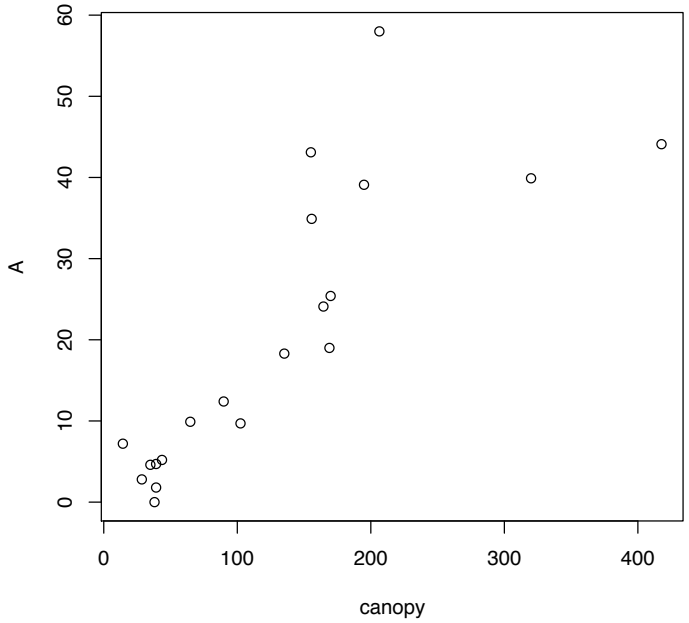
To make a scatterplot, first we simply use the `plot` command:

```
> plot(canopy, A)
```



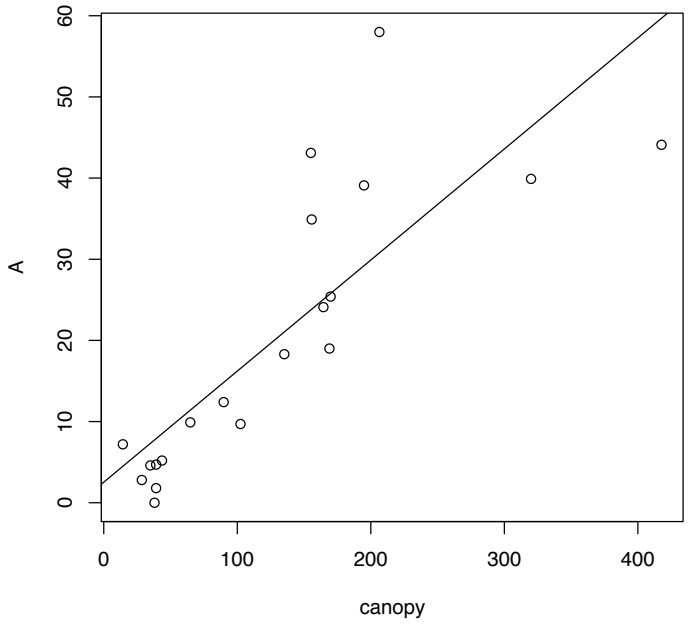
The plot command is expecting the "x" variable first and then the "y" variable second. We could also specify the input as a formula:

```
> plot(A ~ canopy)
```



Either way is valid. I prefer the formula specification, because the text is the same as the model specification in the `lm` command. Now, say we want to add a regression line. We can do this with another command, `abline`:

```
> fit <- lm(A ~ canopy)
> plot(A ~ canopy)
> abline(fit)
```

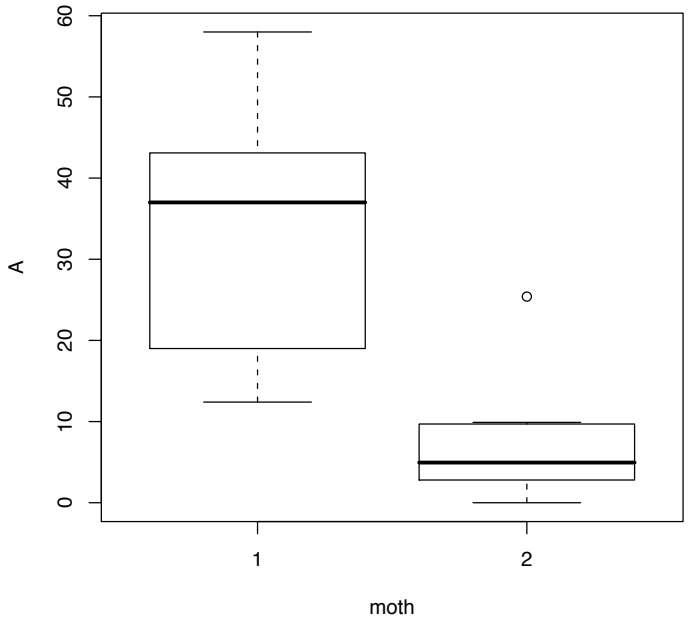


This uses our fitted model, "fit", to draw the regression line.

6.2 Box-and-Whisker and Bar Plots

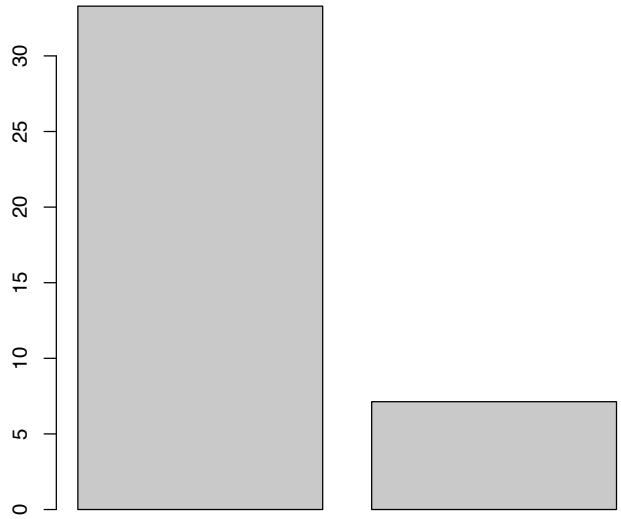
Making a box-and-whisker plot is very much the same process:

```
> moth <- factor(moth)
> plot(A ~ moth)
```



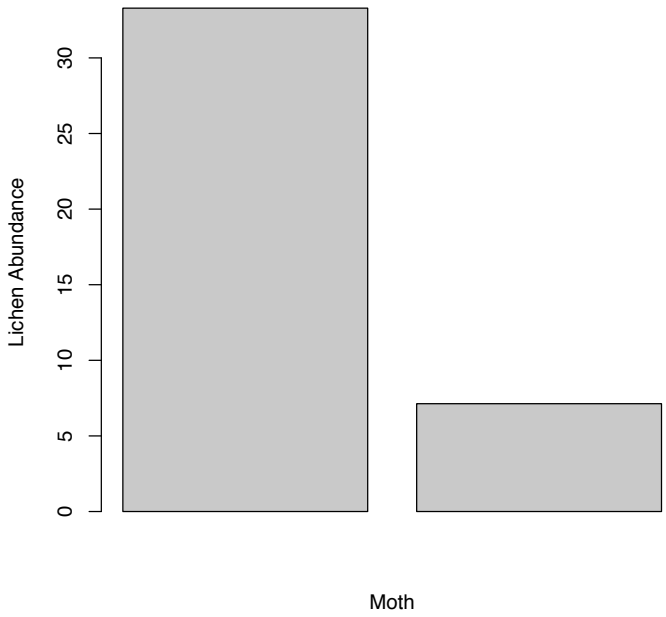
Notice that the plot for "moth" as the predictor is a box-and-whisker plot. If we want a standard barplot we use the `barplot` command, however, we first need the means for each of the levels of "moth"

```
> A.R <- data[11:20, 3]
> A.S <- data[1:10, 3]
> mean.R <- mean(A.R)
> mean.S <- mean(A.S)
> barplot(c(mean.R, mean.S))
```



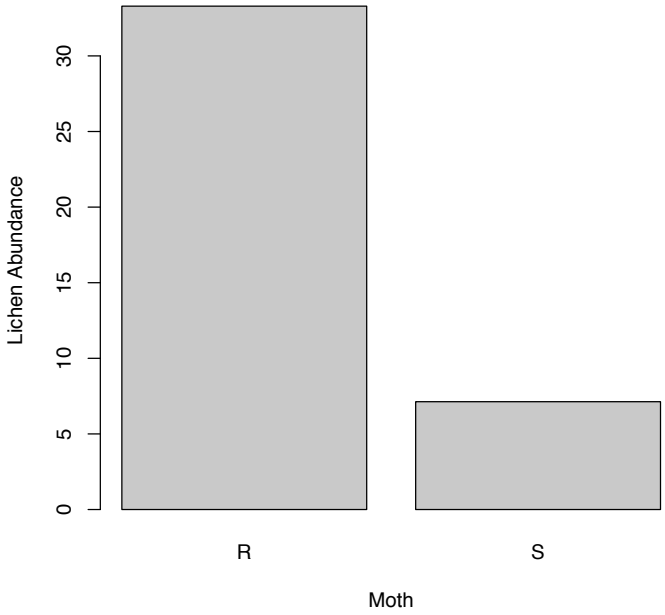
This plot looks pretty poor at this point. There is a whole suite of arguments that can be specified to alter how the plot looks. For instance, you'll notice that the axes have no names. We can add (or change default) axis names by specifying the *xlab* and *ylab* arguments:

```
> barplot(c(mean.R, mean.S), xlab = "Moth", ylab = "Lichen Abundance")
```



We can also add names to the two levels of "moth" by specifying the *names* argument:

```
> barplot(c(mean.R, mean.S), xlab = "Moth", ylab = "Lichen Abundance",
+     names = c("R", "S"))
```



We could continue tweaking all of the various specifications so that we get the plot exactly like we want.

7 Next class:

Next week, we'll go over:

- Improving analysis efficiency
- Changing the working directory
- Analysis of Co-Variance (ANCOVA)
- Making better bar plots
- Begin analyzing community data

Introduction to Ecological Analysis in **R** (Day 2)

Matthew Lau

November 4, 2008

1. Integrative Analysis
2. Making Better Barplots
3. Community Data
4. Species Accumulation Curves
5. Visualizing Data

1 Integrative Analyses

When working in R it's helpful to make sure that you manage objects, functions (i.e. commands) and packages effectively. You can do this by:

1. Setting the working directory to a specific folder where you can keep all of the files pertinent to your analysis
   - You can now enter the file name instead of the whole file path when loading data into **R**
2. Promptly removing objects using the `rm` command
   - `rm(list=ls())` removes all objects visible to **R**
3. Detaching packages when you're through with them
   - `detach(package:`*`package name`*`)`

2 Making a Better Bar Plot

The base function `barplot` does not have a argument specification for making error bars, which is the standard presentation for mean data. Thankfully, someone ran into this issue and made a function that does. This function is a part of a package (i.e. a collection of functions) called *gplots* that is available on the CRAN site. To use this function, first we need to load that package from the CRAN site:

```
> install.packages("gplots")
```

Next, we load the package into R:

```
> library(gplots)
```

Now, we can load our data, make our mean and standard error vectors, which we will be using to make our plot:

NOTE: I have set the working directory to a folder containing the data. Thus, I have only given the file name in the `read.csv` function.

```
> anova.data <- read.csv("PS_Day1_ANOVA.csv")
> attach(anova.data)

> anova.means <- c(mean(Bat_Abundance[Fire_Intensity == 1]), mean(Bat_Abundance[Fire_Inten
+     2]), mean(Bat_Abundance[Fire_Intensity == 3]))
> anova.means

[1]  9.302963 26.189348 15.995588
```

Now we need to create our a vector for our error bars. We do this by first making a vector of standard deviations, which we divide by the square root of our sample size (obtained by the length of one of our response vectors):

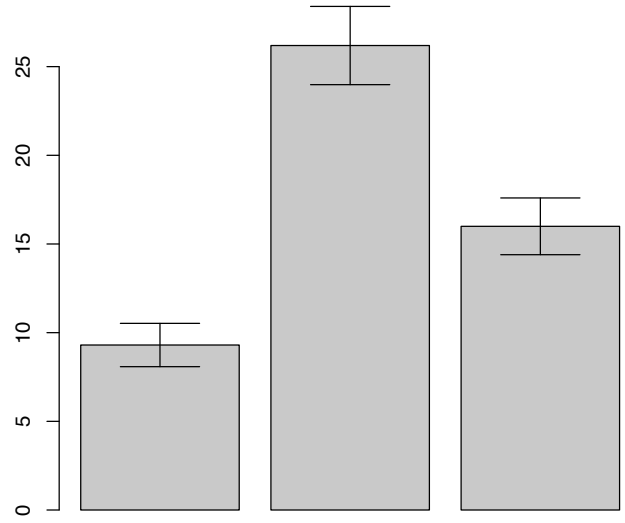
```
> anova.sd <- c(sd(Bat_Abundance[Fire_Intensity == 1]), sd(Bat_Abundance[Fire_Intensity ==
+     2]), sd(Bat_Abundance[Fire_Intensity == 3]))
> n <- length(Bat_Abundance[Fire_Intensity == 1])
> anova.se <- anova.sd/sqrt(n)
> anova.se

[1] 1.221602 2.205972 1.600004
```

We now add and subtract the standard error vector from our means to get our error bar vectors:

```
> anova.ci.u <- anova.means + anova.se
> anova.ci.l <- anova.means - anova.se
> anova.ci.u

[1] 10.52456 28.39532 17.59559

> anova.ci.l

[1]  8.081361 23.983377 14.395584
```

We can now make our plot:

```
> barplot2(anova.means, plot.ci = TRUE, ci.u = anova.ci.u, ci.l = anova.ci.l)
```

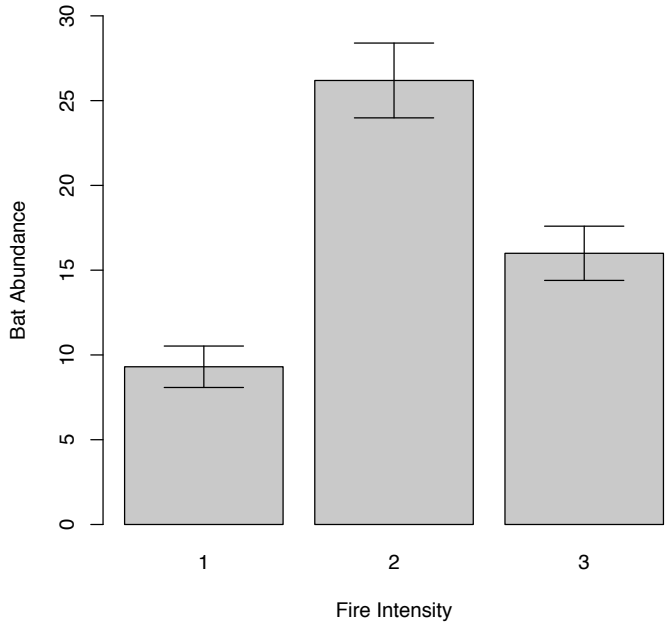


Remember, there are many more arguments in `barplot2`, and just about every other plotting function, that will allow you to specify almost anything you want. For instance, we can improve the plot substantially by just modifying a few argument specifications:

```
> barplot2(anova.means, plot.ci = TRUE, ci.u = anova.ci.u, ci.l = anova.ci.l,
+     xlab = "Fire Intensity", names = c("1", "2", "3"), ylab = "Bat Abundance",
+     col = 8, ylim = c(0, 30))
```

If you look carefully at the argument specification, you can see that to get our error bars we had to specify the logical argument *plot.ci* as TRUE and the *ci.u* and *ci.l* arguments, which define the "upper" and "lower" limits of the error bars, using our error bar vectors. We then also had to specify:

1. *xlab* which is the "x-axis" label
2. *names* which are the levels of x
3. *ylab* which is the "y-axis" label
4. *col* to make the bars grey
5. *ylim* which gives the "y-axis" limits



And now, we clean up:

```
> rm(list = ls())
> detach(anova.data)
```

3 Community Data

Working with community data presents its own set of questions and issues. Here we detail how to:

1. Manage community data
2. Check how well our sample represents the community
3. Obtain univariate community statistics
4. Visualize community data
5. Test for patterns in our community data

```
> com.data <- read.csv("CommData.csv")
```

This file contains both the grouping information as well as the species abundances for each observation in rows. First, we pull out our environmental data and our community matrix:

```
> env <- factor(com.data$env)
> com <- com.data[, 2:ncol(com.data)]
> com <- as.matrix(com)
```

The *as.matrix* function converts our current "matrix" into the matrix format recognized by R (the `read.csv` function imports it as some other format). The dollar sign "$" refers to a particular column (in this case *env*) within a data frame (in this case *com.data*).

4 Species Accumulation Curves

Our inferences from our community analyses rely on whether or not we have adequately sampled the community. To assess this we can create species accumulation curves using functions available in the *vegan* package.

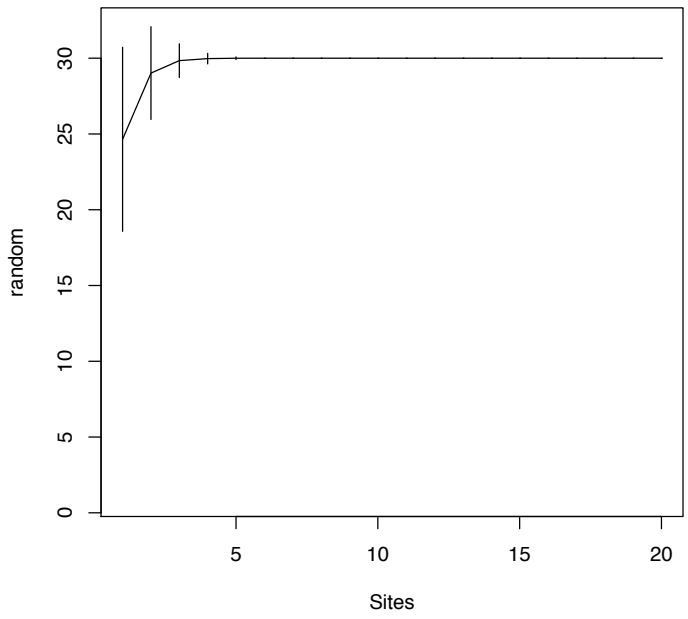
First, we will need to install and load the *vegan* package:

```
> install.packages("vegan")

The downloaded packages are in
        /var/folders/+U/+UN68GjHFRKn4hmJK1Vh0k+++TI/-Tmp-//Rtmp3Gl9wj/downloaded_packages

> library(vegan)
```

Now, we will use the `specaccum` function to generate the species accumulation curve information:

```
> spp.curve <- specaccum(comm = com, method = "random", permutations = 1000)
> plot(spp.curve)
```

Notice how we have specified our community matrix, method and permutation arguments. Simply, this tells the `specaccum` function what the data matrix is called, what method to use to generate the curves and how many permutations to use to generate our confidence inter-



vals.

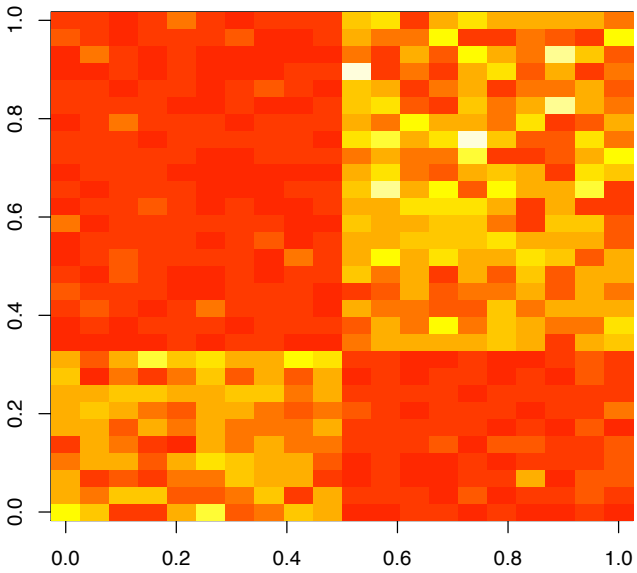
Considering the the the curve levels off way before the number of observations we have, we can confidently say that we have adequately sampled the community. Now we can proceed with our analyses.

5 Visualizing Community Data

5.1 Heatmaps

Heatmaps are typically used to present molecular data (e.g. microarray data), but I have found them to be a useful look at the entirety of the dataset before proceeding to ordination based techniques. To do this, we simply use the *image*:

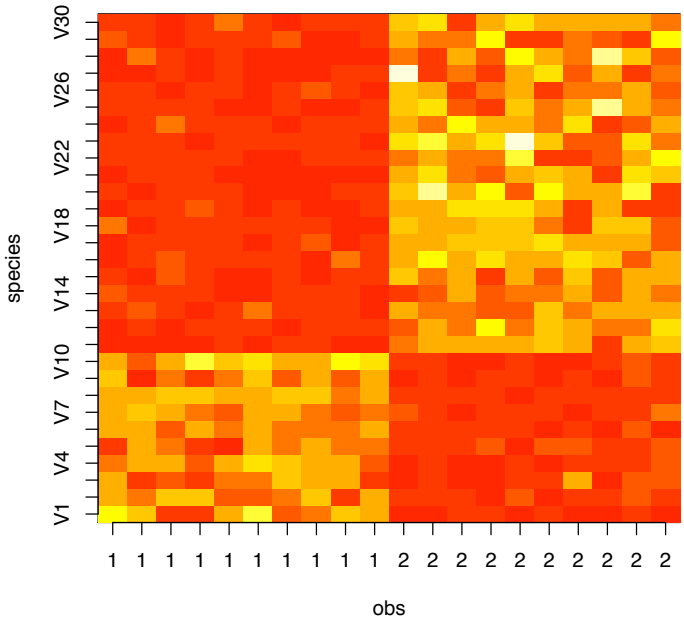
```
> com <- as.matrix(com)
> image(com)
```



You'll notice that we first used the *as.matrix* function on our community matrix. This is because when we imported the data using the *read.csv* function, R formatted it as a "data frame". Unfortunately, some functions, such as the *image* function, require the data to be a "matrix" format. You can check to see if an object is a matrix by using the *is.matrix* function.

Next, we may want to alter this plot in order to make it more comprehensible. For instance, we can change the axis by:

```
> obs = seq(1, nrow(com), by = 1)
> species = seq(1, ncol(com), by = 1)
> image(x = obs, y = species, z = com, xaxt = "n", yaxt = "n")
> axis(side = 1, tick = TRUE, at = obs, labels = env)
> axis(2, tick = TRUE, at = species, labels = colnames(com))
```



We now have a plot with the appropriate lables. What we have done to do this is first make two vectors that define the axis markings for our observations and our species. We then created a plot with no axis labels by specifying the arguments, *xaxt* and *yaxt*, as "n", which is short for "none". We then use the `axis` function to plot both the x (observations) and the y (species) axes. For more information on the argument specifications for these functions, please consult the help files (e.g. help(axis)).

We can also make a legend by adding the script at the bottom specifying the *legend* function:

```
> obs = seq(1, nrow(com), by = 1)
> species = seq(1, ncol(com), by = 1)
> image(x = obs, y = species, z = com, xaxt = "n", yaxt = "n")
> axis(side = 1, tick = TRUE, at = obs, labels = env)
> axis(2, tick = TRUE, at = species, labels = colnames(com))
> legend("topleft", legend = seq(min(com), max(com), by = 50),
+     col = heat.colors(9), pch = c(0, 0, 0, 0, 0, 0, 0, 0, 0,
+         0), bg = "white")
```

5.2 Ordination

The heatmaps are only useful to a certain extent for visualizing community data. The more commonly used method is ordination. This is merely a process of simplifying the multi-dimensional data into

something with fewer dimensions that is presentable and still confers important patterns in the data. Non-metric Multidimensional Scaling (NMS) using Bray-Curtis distance is the most commonly used ordination technique used for community data. This method is robust to typical characteristics of community data. It also represents the data in distance space, preserving the rank order of the observations.

To conduct NMS ordination in R, we will first need to load several packages that contain functions that we will be using:

```
> install.packages("ecodist")

The downloaded packages are in
        /var/folders/+U/+UN68GjHFRKn4hmJK1Vh0k+++TI/-Tmp-//Rtmp3Gl9wj/downloaded_packages

> install.packages("BiodiversityR")

The downloaded packages are in
        /var/folders/+U/+UN68GjHFRKn4hmJK1Vh0k+++TI/-Tmp-//Rtmp3Gl9wj/downloaded_packages

> install.packages("ellipse")

The downloaded packages are in
        /var/folders/+U/+UN68GjHFRKn4hmJK1Vh0k+++TI/-Tmp-//Rtmp3Gl9wj/downloaded_packages
```

As before with the *gplots* package, we now need to load these packages into R:

```
> library(ecodist)
> library(BiodiversityR)
> library(ellipse)
> library(vegan)
```

Notice here that I have also loaded the *vegan* package, even though I didn't install it in the last step. This is because once you have installed a package, as we did when we were generating species area curves above, you do not need to install it again (unless you delete it). You do, however, need to load it again if you have detached it or closed and re-startred R.

```
> dis <- distance(com, method = "bray-curtis")
```

The next step in NMS ordination is to determine the most parsimonious number of axes. That is, we need to find out the lowest number of NMS axes that will adequately represent the data. Here, we can use stress (i.e. the distance of the NMS representation from the original, un-scaled, distances) to assess the departure of the NMS configuration from the actual data.

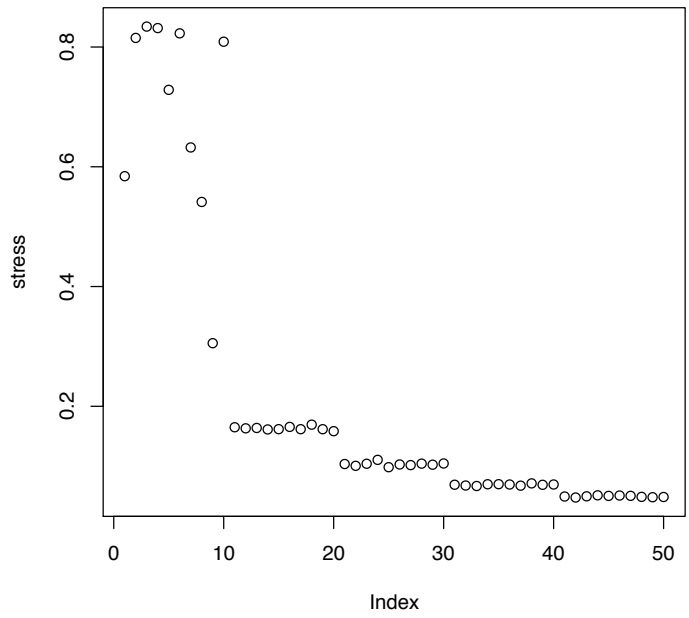
To determine the parsimonious axis number, we generate a scree plot (for more info, consult the PC-ORD manual). We can do this using the *nmds* command in the *ecodist* package:

```
> scree <- nmds(dis, mindim = 1, maxdim = 5, nits = 10)
```

```
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
```

```
Using random start configuration
Using random start configuration
Using random start configuration
```

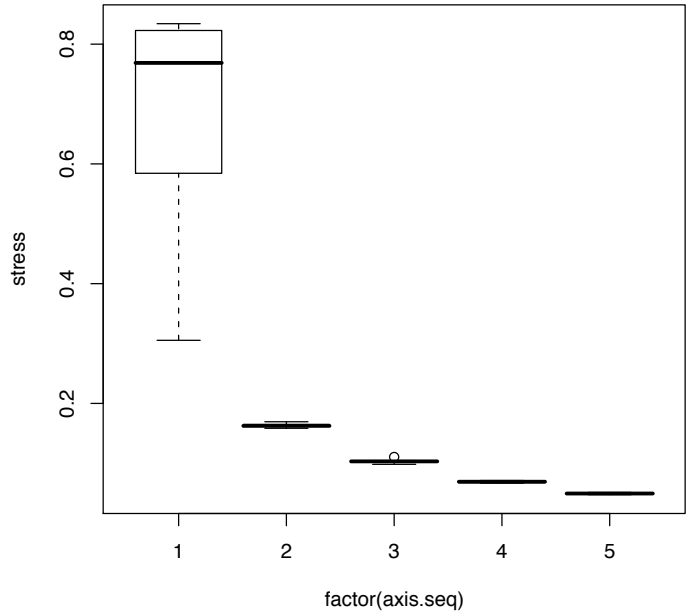
```
> stress <- scree$stress
> plot(stress)
```



This plot is of the stress values from the minimized stress configurations from ten random starts (*inits*=10) at each level of dimensionality, one to five, (*mindim=1*,*maxdim=5*), ranked in order from the first to the last configuration, which means that the first ten stress values are all from 1-D configurations, the second ten are from 2-D, and so on. What we can see is that the stress rapidly decreases as we increase the number of nms axes and that at two axes we have an acceptable stress level with little variation amongst repeated random starting configurations. Thus, 2-D is fine for our purposes.

Here is the code for a more presentable scree plot (dissect it at your leisure):

```
> axis.seq <- c(seq(1, 1, length = 10), seq(2, 2, length = 10),
+     seq(3, 3, length = 10), seq(4, 4, length = 10), seq(5, 5,
+         length = 10))
> plot(stress ~ factor(axis.seq))
```



We can now re-run the *nmds* command with more iterations to "explore" more of the ordination configuration space and ensure that we have the lowest stress configuration possible:

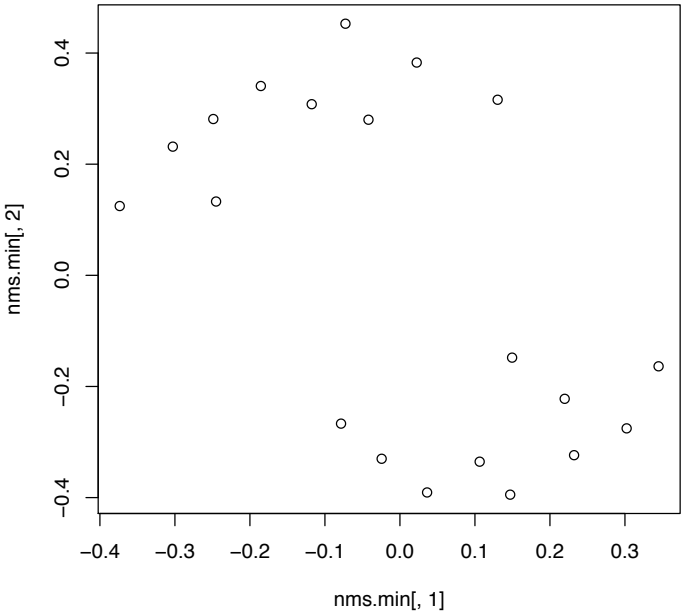
```
> nms.final <- nmds(dis, mindim = 2, maxdim = 2, nits = 50)

Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
```

```
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
Using random start configuration
```

Now we have an object containing a whole bunch of minimized stress configurations generated from random starting configurations. We can use the *nmds.min* function to extract the lowest stress configuration and plot it:

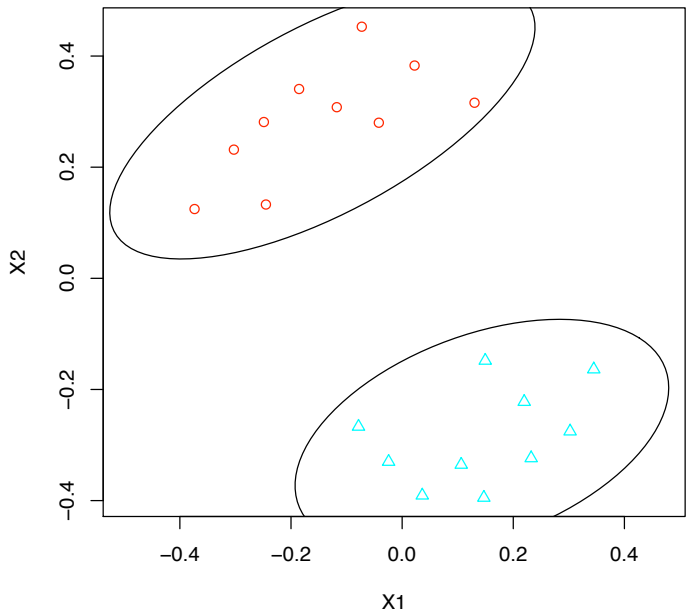
Now we have an object that contains a whole bunch of minimized stress NMS configurations generated from random starts. We can use the *nmds.min* function to extract the lowest stress configuration and plot it:

```
> nms.min <- nmds.min(nms.final, dims = 2)
> plot(nms.min[, 1], nms.min[, 2])
```



This plot is hardly satisfactory. To make a better plot we can use more functions from our packages that are specific to making ordination plots:

```
> nms.plot <- ordiplot(nms.min, type = "n")
> env.frame <- data.frame(env)
> ordisymbol(nms.plot, y = env.frame, factor = "env", rainbow = T,
+     col = env, legend = F)
> ordiellipse(nms.plot, env, kind = "sd", conf = 0.95)
```



Here, we make a plot with no points in it with the `ordiplot` function, saving the plot information into an object for further use in the `ordisymbol` function. Notice the use of the `data.frame` function. This creates a data frame from the object "env", which is the expected format for the `ordisymbol` function. Last, we use the `ordiellipse` function from the *ellipse* package to draw confidence ellipses on our plot.

Last, to get the actual stress value for our final configuration:

```
> min(nms.final$stress)

[1] 0.1565177
```

Next week we will finish up community analysis by going over:

1. Multivariate Analyses (i.e. MRPP, ANOSIM and PerMANOVA)
2. Indicator Species Analysis

Introduction to Ecological Analysis in R (Day 3)

Matthew Lau

November 14, 2008

1. Community Analysis Continued
   (a) Multivariate Tests for Differences in Community Composition
      i. AnoSim
      ii. MRPP
      iii. PerMANOVA
   (b) Indicator Species Analysis
2. Looping: Getting R to Do Many Individual Tests

1 Community Analysis Continued

The data file "CommData.csv" has both our species abundance data and our grouping data combined in one matrix. First we need to separate the two:

```
> com.data <- read.csv("CommData.csv")
> com <- com.data[, -1]
> env <- factor(com.data[, 1])
```

1.1 Multivariate Tests for Differences in Community Composition

We now have a vector "env", which is formatted as a factor using `factor`, containing the grouping data and a matrix "com" composed of our species abundance data. Here are three ways to test for differences in composition between our two groups using commands available in the *vegan* package, which we installed last time:

```
> library(vegan)
```

1.1.1 Analysis of Similarity (AnoSim)

For this test, we first need to convert our species abundance data into dissimilarities (i.e. distances) using the `vegdist` function. We then use the `anosim` function on our distances specifying our groups with our "env" data vector:

```
> dist.com <- vegdist(com, method = "bray")
> anosim(dis = dist.com, grouping = env)

Call:
anosim(dis = dist.com, grouping = env)
Dissimilarity: bray

ANOSIM statistic R:     1
      Significance: < 0.001

Based on  1000  permutations
```

1.1.2 Multiple Response Permutation Procedure (MRPP)

Here, we basically do the exact same thing, except that we don't have to convert our abundance matrix to a distance matrix. The function will do the conversion for us. We just need to specify the distance metric to use:

```
> mrpp(com, env, distance = "bray")

Call:
mrpp(dat = com, grouping = env, distance = "bray")

Dissimilarity index: bray
Weights for groups:  n

Chance corrected within-group agreement A: 0.4165
Based on observed delta 0.2852 and expected delta 0.4887

Significance of delta: < 0.001
Based on  1000  permutations
```

1.1.3 Last, there is a little used permutation multivariate analysis of variance test (PerMANOVA) that is useful when you are interested in testing for the effect of more than one factor. The specification looks very much that same as a regular ANOVA of regression except that we use the function, `adonis`, and we need to specify the number of permutations:

```
> adonis(com ~ env, permutations = 1000)

Call:
adonis(formula = com ~ env, permutations = 1000)

               Df SumsOfSqs   MeanSqs   F.Model     R2  Pr(>F)
env         1.000000  1.903327  1.903327 45.775944 0.7178 < 0.001 ***
Residuals 18.000000  0.748426  0.041579           0.2822
Total     19.000000  2.651753                     1.0000
---
```

NOTE: the number of permutations directly effects the resulting p-value. This is because the p-value is calculated as the number of permutations that result in a F-statistic greater than or equal to our observed F-statistic divided by the total number of permutations. For instance, if you only ran ten permutations the smallest, non-zero, p-value that you could possibly get is 0.10 (i.e. one divided by ten). The rule of thumb is to run at least 200 permutations, but try and run as many as you can. Usually 1000 is sufficient.

1.2 Indicator Species Analysis

The above test will only tell us that there is a difference in composition. They don't tell us what this difference is. The most common method used for this is an Indicator Species Analysis. This will tell us the species that tend to be found in one of our groups versus another. Luckily, the *labdsv* package provides useful functions to do this:

```
> library(labdsv)

> IS <- duleg(com, env)
> summary(IS)
```

	cluster	indicator_value	probability
V1	1	0.9001	0.001
V10	1	0.8668	0.001
V4	1	0.8388	0.001
V8	1	0.8285	0.001
V6	1	0.8180	0.001
V7	1	0.7754	0.001
V2	1	0.7705	0.001
V9	1	0.7311	0.010
V3	1	0.7305	0.022
V21	2	0.9479	0.001
V11	2	0.9273	0.001
V27	2	0.9210	0.001
V23	2	0.8954	0.001
V25	2	0.8902	0.001
V28	2	0.8712	0.001
V20	2	0.8688	0.001
V19	2	0.8661	0.001
V17	2	0.8569	0.001
V15	2	0.8418	0.001
V18	2	0.8329	0.001
V12	2	0.8285	0.001
V30	2	0.8231	0.001
V26	2	0.8073	0.001
V16	2	0.8052	0.001
V14	2	0.7955	0.001
V22	2	0.7873	0.001
V24	2	0.7796	0.002
V13	2	0.7664	0.001

```
V29       2          0.7588       0.001

Sum of probabilities =  0.213

> detach(package:labdsv)
```

This removes all of the *labdsv* functions from the R library.

2 Looping: Getting R to Do Many Individual Tests

There are some situations in which we would like to perform a task repeatedly. We can get R to do a repeat task or set of tasks by doing what is called "looping" in programming lingo. To create a "loop", all we need to know is how to use the `for` function. Essentially, the structure of a "for-loop" says, "for this many steps, do this task." For instance, if we wanted to have R add up the numbers one to ten, we could code it with a for-loop:

```
> a = 0
> for (i in 1:10) {
+     a = a + 1
+ }
> a

[1] 10
```

This is a very simple task, and we most likely would never do this, but it illustrates all you need know to construct almost any for-loop, which is a very powerful thing. Note that inside of the curly-brackets, we are telling R to add one to the object, "a", for ten steps. That is R adds one to "a" then does it again to the "new" value of "a" generated by the last step, and so on, until it has done it ten times. Also note, we created "a" *outside* of the loop. This is important because it needs to be created before we can do anything with it, and if we have it inside the loop, R will make a new "a" at each step.

Now, say for instance our advisor asks us to run tests for the effect of our environmental factor on all of our species. This could take a long time or a lot of code lines to this. However, for-loops can save the day. We simply write the test into a for-loop properly and, *voi-la*, R will do all of tests for us:

```
> results <- list()
> for (i in 1:ncol(com)) {
+     results[i] <- summary(aov(com[, i] ~ env))
+ }
> results[1:10]
```

```
[[1]]
            Df Sum Sq Mean Sq F value    Pr(>F)
env          1 146890  146890  24.496 0.0001036 ***
Residuals   18 107938    5997
---


[[2]]
            Df Sum Sq Mean Sq F value    Pr(>F)
env          1  69502   69502  24.208 0.0001104 ***
Residuals   18  51679    2871
---


[[3]]
            Df Sum Sq Mean Sq F value  Pr(>F)
env          1  36722   36722  8.0937 0.01075 *
Residuals   18  81668    4537
---


[[4]]
            Df Sum Sq Mean Sq F value   Pr(>F)
env          1 112650  112650  47.359 1.95e-06 ***
Residuals   18  42816    2379
---


[[5]]
            Df Sum Sq Mean Sq F value  Pr(>F)
env          1  16416   16416  5.3427 0.03286 *
Residuals   18  55309    3073
---


[[6]]
            Df Sum Sq Mean Sq F value    Pr(>F)
env          1  78250   78250  76.257 6.885e-08 ***
Residuals   18  18470    1026
---


[[7]]
            Df Sum Sq Mean Sq F value    Pr(>F)
env          1  65322   65322  34.603 1.435e-05 ***
Residuals   18  33979    1888
---


[[8]]
```

```
            Df Sum Sq Mean Sq F value    Pr(>F)
env          1 135137  135137  126.88 1.388e-09 ***
Residuals   18  19171    1065
---


[[9]]
            Df Sum Sq Mean Sq F value   Pr(>F)
env          1  57459   57459  14.535 0.001275 **
Residuals   18  71156    3953
---


[[10]]
            Df Sum Sq Mean Sq F value    Pr(>F)
env          1 193258  193258  50.243 1.313e-06 ***
Residuals   18  69236    3846
---
```

You'll notice that we first made a new list object called, "results". We then wrote the for-loop, which consists of two lines:

1. The number of loops
2. The function to be repeated, which say run this test of the column of com equal to "i" (i.e. the loop or step number) and save it as the "ith" component of the list "results"

For-loops will do anything you tell them to and can be extremely useful. Play around and try and make it part of your working R vocabulary.

Introduction to Ecological Analysis in **R**: Problem Set for Day 3

Matthew Lau

today

Using the community dataset that we used in Day 2 (*CommData.csv*), conduct the following analysis:

1. Set the working directory to your data folder and load the data, *CommData.csv*
2. Check that we adequately sampled the community with a species accumulation curve
3. Obtain and test for the effect of our group variable, "env", on species richness and both Shannon's and Simpson's diversity indices
4. Produce an ordination plot
5. Test for the effect of "env" on community composition using the `mrpp` command in the *vegan* package

R-Course Day 4

Matthew K. Lau

November 29, 2008

In class last Tuesday we covered:

1. Review of Analysis in **R**
2. Writing Functions
3. Sorting Data
4. Principal Components Analysis
5. Infromation Theory Based Multi-Model Inference

1 Review of Analysis in R

Before we keep going with progressively complex topics, I would like to bring things back to brass tacks. At the heart of all of these fancy scripts and packages is our desire to conduct efficient inferences about our questions of interest. This inherently includes all aspects of analysis from data management to presentation. It's very easy to lose sight of this in the midst of searching around for the "right" test and presentation format. I constantly remind myself by asking, "what the $#&% is the point of all this?"

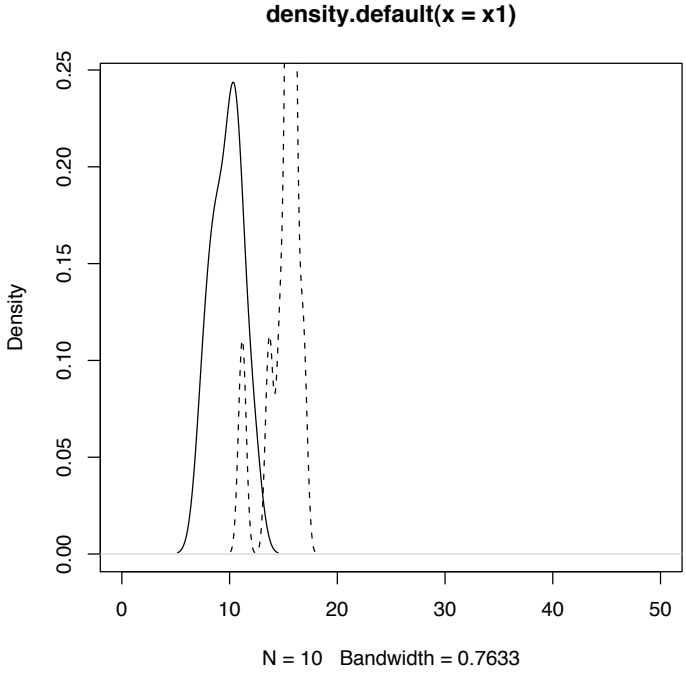
Once you get around the initial difficulties of using **R** it provides an analytical environment that has a greater potential. The goal of making plots and conducting analyses is to generate a coherent, reasoned argument for or against a particular line of inference, not to fulfill some predefined list of rules. Here is a quick run through a hypothetical analysis scenario:

First we create we get data, which in this case are generated by **R**, but we could have also entered them in via any number of data importing functions:

```
> x1 <- rnorm(10, 10, 2)
> x2 <- rnorm(10, 15, 2)
```

Once we have the data inputed, we would most likely want to visualize and summarize the data:

```
> plot(density(x1), xlim = c(0, 50))
> lines(density(x2), lty = 2)
> summary(cbind(x1, x2))

       x1               x2
 Min.   : 7.455   Min.   :11.18
 1st Qu.: 8.838   1st Qu.:14.90
 Median :10.028   Median :15.59
 Mean   : 9.799   Mean   :15.06
 3rd Qu.:10.639   3rd Qu.:15.75
 Max.   :12.211   Max.   :16.84
```



Based on our knowledge of statistical inference, we may decide to conduct a test of that the two means are different from each other via a non-directional, two-sample t-test:

```
> tt <- t.test(x1, x2)
> tt

        Welch Two Sample t-test

data:  x1 and x2
t = -7.705, df = 17.849, p-value = 4.405e-07
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -6.698512 -3.826834
```

```
sample estimates:
mean of x mean of y
 9.798954 15.061627
```

It is important to keep good records of analyses. One way to do this would be to make a file with both the data and the results of the t-test together. This could stave off a huge headache later-on if a repeat analysis is called for:

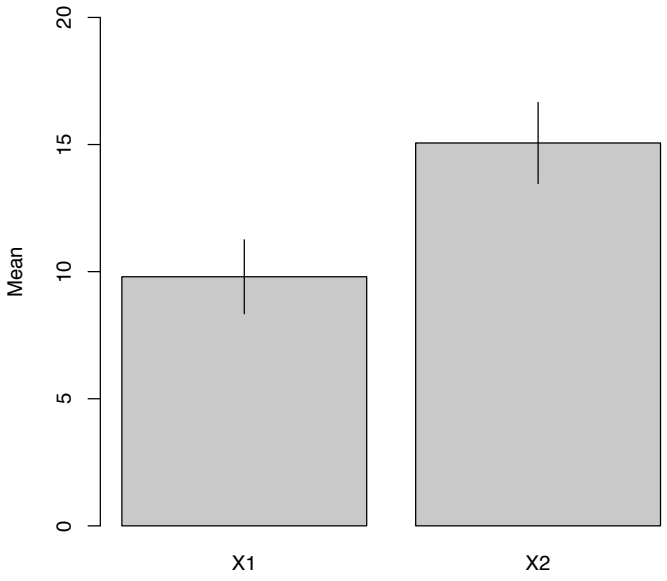
```
> file.create("/Users/artemis/Documents/FALL2008/R-Course_Day4_Example/Day4_data.R")
[1] TRUE
> file.create("/Users/artemis/Documents/FALL2008/R-Course_Day4_Example/Day4_results.R")
[1] TRUE
> write.table(cbind(x1, x2), file = "/Users/artemis/Documents/FALL2008/
R-Course_Day4_Example/Day4_data.R")
> write.table(unlist(tt), file = "/Users/artemis/Documents/FALL2008/
R-Course_Day4_Example/Day4_results.R")
```

Notice that we used the `cbind` and the `unlist` functions to reformat the data and the results before saving them as ".R" files. To double check that everything was saved correctly, we can read the files back into **R**:

```
> read.table("/Users/artemis/Documents/FALL2008/
R-Course_Day4_Example/Day4_data.R")
          x1       x2
1   7.454862 16.06117
2  10.543494 15.67643
3  10.671207 15.48232
4   8.794536 15.51148
5  12.211224 14.69980
6  11.155776 15.73512
7   8.134445 11.17991
8   9.930993 16.83588
9   8.968685 15.74910
10 10.124315 13.68506
> read.table(file = "/Users/artemis/Documents/FALL2008/
R-Course_Day4_Example/Day4_results.R")
                                                 x
statistic.t                          -7.7050266505846
parameter.df                          17.8489694691014
p.value                          4.40486369636702e-07
conf.int1                            -6.6985120477549
conf.int2                           -3.82683448815842
estimate.mean of x                    9.79895376046191
estimate.mean of y                    15.0616270284186
null.value.difference in means                       0
alternative                                  two.sided
method                         Welch Two Sample t-test
data.name                                    x1 and x2
```

Once we have our analyses done, we will most likely want to create a figure to present these results. Previously, we used the `barplot2` function in the *gplots* package, but here we will use the `lines` function to plot our confidence intervals.

```
> barplot(c(mean(x1), mean(x2)), names = c("X1", "X2"), ylab = "Mean",
+     ylim = c(0, 20), col = "grey")
> lines(c(0.7, 0.7), c(mean(x1) - sd(x1), mean(x1) + sd(x1)),
+     lwd = 1)
> lines(c(1.9, 1.9), c(mean(x2) - sd(x2), mean(x2) + sd(x2)))
```



As you can see, the `barplot` function creates an initial plot, which we then "write" on top of with the `lines` functions.

And last, we clean up our mess by removing all of the objects that we created. Had we loaded any packages that we weren't going to use later on, we'd unload them by the `detach` function:

```
> rm(list = ls())
```

2 Writing Functions

We have been using functions left and right. All functions do is group together other functions in an applied way. Because **R** is an "open-source" programming language, many of the functions in its employ have been created by users of **R**

(e.g. `barplots2`). We can see all of the code (i.e. the guts) of any function by typing in the name of the function with no parantheses:

```
> sd

function (x, na.rm = FALSE)
{
    if (is.matrix(x))
        apply(x, 2, sd, na.rm = na.rm)
    else if (is.vector(x))
        sqrt(var(x, na.rm = na.rm))
    else if (is.data.frame(x))
        sapply(x, sd, na.rm = na.rm)
    else sqrt(var(as.vector(x), na.rm = na.rm))
}
<environment: namespace:stats>
```

Function Anatomy

Functions have a structure very similar to loops. In fact, one way of looking at a function is to see it as a function that creates a function from functions. For example, if we want to create a function that will calculate the standard error of a sample for us, we would write:

```
> SE <- function(x = "sample vector") {
+     se <- sd(x)/sqrt(length(x))
+     return(se)
+ }
```

Let's take a look at what's going on. First, there is the `function` function itself. This allows for object oriented arguments to be specified in a general way. When we use our function later on, these arguments will need to be specified. Second, we specify the "action" of the function (i.e. what the function will be doing). This can be anything. Here we calculate the standard error, "se", and then use the `return` function to give us the calculated value. Last, we create an object with the backward arrow, `<-`. This names our function by saving it into as an object. Now, we can use our function:

```
> x <- rnorm(10, 10, 2)
> x

 [1]  8.828547 10.691604 10.516017 11.258912 10.312600  8.651738
 [7] 11.015661 10.920978 12.779844 11.420712

> SE(x)

[1] 0.3822643
```

Notice here that functions can be composed of any other functions, including loops. For example:

```
> loop.func <- function(x = "number of loops") {
+     for (i in 1:100) {
+         x <- x + 1
+     }
+     return(x)
+ }
> loop.func(10)

[1] 110
```

Functions are the atoms of the nucleus that is **R**. Knowing how to write them will allow you to do many things in **R** that you can not do in your standard point and click stats packages.

3 Sorting Data

Sorting data is an essential part of data management. In **R** sorting is very simply achieved by using the bracket specification. For example, if we wanted to sort our data matrix by the central column from lowest to highest:

```
> data <- cbind(1:10, rep(c(1, 2), 10), rnorm(10))
```

We could go in and create a vector of the position of the all the values in the correct order, and put it in our bracket specification:

```
> sorting.vector <- c(1, 3, 5, 7, 9, 11, 13, 15, 17, 19,
+     2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
> data[sorting.vector, ]

      [,1] [,2]       [,3]
 [1,]    1    1 -0.3354490
 [2,]    3    1  0.4490912
 [3,]    5    1 -0.2910797
 [4,]    7    1  2.1438397
 [5,]    9    1  0.4063706
 [6,]    1    1 -0.3354490
 [7,]    3    1  0.4490912
 [8,]    5    1 -0.2910797
 [9,]    7    1  2.1438397
[10,]    9    1  0.4063706
[11,]    2    2 -1.1793572
[12,]    4    2  1.0220319
[13,]    6    2  1.2972297
[14,]    8    2  0.3452335
[15,]   10    2  0.1251559
[16,]    2    2 -1.1793572
[17,]    4    2  1.0220319
```

```
[18,]    6    2  1.2972297
[19,]    8    2  0.3452335
[20,]   10    2  0.1251559
```

Note that we are placing our "sorting vector" in the row portion of our bracket specification. This is because we are telling **R** to give us the rows of the object *data* as specified by our sorting vector, just like we've use it before to pull out portions of data vectors and matrixes.

This way is a very impractical to sort data on a regular basis. Instead, we can use the `order` function to return the "sorting vector" for us, both separately and within the bracket specification itself:

```
> order(data[, 2])

 [1]  1  3  5  7  9 11 13 15 17 19  2  4  6  8 10 12 14 16 18 20

> sorting.vector2 <- order(data[, 2])
> sorting.vector2

 [1]  1  3  5  7  9 11 13 15 17 19  2  4  6  8 10 12 14 16 18 20

> data[sorting.vector2, ]

      [,1] [,2]       [,3]
 [1,]    1    1 -0.3354490
 [2,]    3    1  0.4490912
 [3,]    5    1 -0.2910797
 [4,]    7    1  2.1438397
 [5,]    9    1  0.4063706
 [6,]    1    1 -0.3354490
 [7,]    3    1  0.4490912
 [8,]    5    1 -0.2910797
 [9,]    7    1  2.1438397
[10,]    9    1  0.4063706
[11,]    2    2 -1.1793572
[12,]    4    2  1.0220319
[13,]    6    2  1.2972297
[14,]    8    2  0.3452335
[15,]   10    2  0.1251559
[16,]    2    2 -1.1793572
[17,]    4    2  1.0220319
[18,]    6    2  1.2972297
[19,]    8    2  0.3452335
[20,]   10    2  0.1251559

> data[order(data[, 2]), ]

      [,1] [,2]       [,3]
 [1,]    1    1 -0.3354490
 [2,]    3    1  0.4490912
```

```
 [3,]    5    1 -0.2910797
 [4,]    7    1  2.1438397
 [5,]    9    1  0.4063706
 [6,]    1    1 -0.3354490
 [7,]    3    1  0.4490912
 [8,]    5    1 -0.2910797
 [9,]    7    1  2.1438397
[10,]    9    1  0.4063706
[11,]    2    2 -1.1793572
[12,]    4    2  1.0220319
[13,]    6    2  1.2972297
[14,]    8    2  0.3452335
[15,]   10    2  0.1251559
[16,]    2    2 -1.1793572
[17,]    4    2  1.0220319
[18,]    6    2  1.2972297
[19,]    8    2  0.3452335
[20,]   10    2  0.1251559
```

To sort from highest to lowest, we merely specify the *decreasing* argument in the `order` function:

```
> data[order(data[, 2], decreasing = TRUE), ]
```

```
      [,1] [,2]       [,3]
 [1,]    2    2 -1.1793572
 [2,]    4    2  1.0220319
 [3,]    6    2  1.2972297
 [4,]    8    2  0.3452335
 [5,]   10    2  0.1251559
 [6,]    2    2 -1.1793572
 [7,]    4    2  1.0220319
 [8,]    6    2  1.2972297
 [9,]    8    2  0.3452335
[10,]   10    2  0.1251559
[11,]    1    1 -0.3354490
[12,]    3    1  0.4490912
[13,]    5    1 -0.2910797
[14,]    7    1  2.1438397
[15,]    9    1  0.4063706
[16,]    1    1 -0.3354490
[17,]    3    1  0.4490912
[18,]    5    1 -0.2910797
[19,]    7    1  2.1438397
[20,]    9    1  0.4063706
```

4 Principal Components Analysis (PCA)

The goal of any ordination technique is to reduce data complexity. This either the number of factors or response variables. Typically, ecologist are interested

in the latter.

PCA is one ordination method that uses linear combinations of a subset of variables to represent the full set of variables. This method, although more informative than Non-Metric multidimensional Scaling (NMS), is not typically compatible with community data, which often has strong non-linear relationships. This is typically done through a subjective process of assessing the proportion of variance explained by each "principal component" (i.e. one linear combination of variables). This is actually a very simple process. Although there are multiple PCA functions available, because of its simplicity, it is not much finger-work to do it from scratch by writing loops:

Load *vegan* package and the "dune" data, which comes with it.

```
> library(vegan)
> data(dune)
> dune[1:3, ]
```

```
   Belper Empnig Junbuf Junart Airpra Elepal Rumace Viclat Brarut
2       3      0      0      0      0      0      0      0      0
13      0      0      3      0      0      0      0      0      0
4       2      0      0      0      0      0      0      0      2
   Ranfla Cirarv Hyprad Leoaut Potpal Poapra Calcus Tripra Trirep
2       0      0      0      5      0      4      0      0      5
13      2      0      0      2      0      2      0      0      2
4       0      2      0      2      0      4      0      0      1
   Antodo Salrep Achmil Poatri Chealb Elyrep Sagpro Plalan Agrsto
2       0      0      3      7      0      4      0      0      0
13      0      0      0      9      1      0      2      0      5
4       0      0      0      5      0      4      5      0      8
   Lolper Alogen Brohor
2       5      2      4
13      0      5      0
4       5      2      3
```

Calculate the covariance matrix from the data matrix:

```
> c <- cov(dune)
```

Calculate eigenvalues from the covariance matrix:

```
> dune.e <- eigen(c)
> e.values <- dune.e$values
> is.vector(e.values)
```

```
[1] TRUE
```

```
> e.values
```

```
 [1]  2.479532e+01  1.814662e+01  7.629135e+00  7.152772e+00
 [5]  5.695027e+00  4.333307e+00  3.199365e+00  2.781865e+00
```

```
 [9]  2.481984e+00  1.853767e+00  1.747117e+00  1.313583e+00
[13]  9.905115e-01  6.377937e-01  5.508266e-01  3.505841e-01
[17]  1.995562e-01  1.487978e-01  1.157526e-01  2.477742e-16
[21]  1.265238e-16  9.915497e-17  7.827192e-17  7.767934e-17
[25] -1.615479e-17 -1.670766e-16 -1.790788e-16 -1.994938e-16
[29] -7.571117e-16 -7.735208e-15
```

Calculate the Percent Variance Explained (PVE) by each principal component (NOTE: the eigenvalues are in order from highest to lowest and each principal component is defined by the rank of its associated eigenvalue):

```
> e.values[1]/sum(e.values)

[1] 0.2947484
```

This is more easily done by using a loop:

```
> pve = numeric()
> for (i in 1:length(e.values)) {
+     pve[i] <- e.values[i]/sum(e.values)
+ }
> pve

 [1]  2.947484e-01  2.157136e-01  9.068950e-02  8.502685e-02
 [5]  6.769826e-02  5.151114e-02  3.803168e-02  3.306874e-02
 [9]  2.950399e-02  2.203621e-02  2.076844e-02  1.561490e-02
[13]  1.177447e-02  7.581619e-03  6.547818e-03  4.167483e-03
[17]  2.372176e-03  1.768798e-03  1.375981e-03  2.945356e-18
[21]  1.504021e-18  1.178681e-18  9.304386e-19  9.233944e-19
[25] -1.920362e-19 -1.986082e-18 -2.128757e-18 -2.371435e-18
[29] -8.999983e-18 -9.195042e-17
```

Now, we can calculate the Cumulative Percent Variance explained by each successive principal component using yet another loop:

```
> cpve = numeric()
> for (i in 1:length(pve)) {
+     if (i == 1) {
+         cpve[i] <- pve[i]
+     }
+     else {
+         cpve[i] <- cpve[i - 1] + pve[i]
+     }
+ }
> cpve

 [1] 0.2947484 0.5104620 0.6011515 0.6861783 0.7538766 0.8053877
 [7] 0.8434194 0.8764881 0.9059921 0.9280283 0.9487968 0.9644117
[13] 0.9761861 0.9837677 0.9903156 0.9944830 0.9968552 0.9986240
[19] 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
[25] 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
```

A scree plot is a nice visual summary of this information:
■fig=true■ plot(cpve) abline(a=0.95,b=0)

To get the principal components, we simply multiply the original data matrix by the eigenvector matrix that we can get from the `eigen` function that we used to get our eigenvalues. To do this, we need to first convert the dune data into a matrix (it's currently a data frame), and then we need to use the matrix multiplier "%%" to multiply the two together:

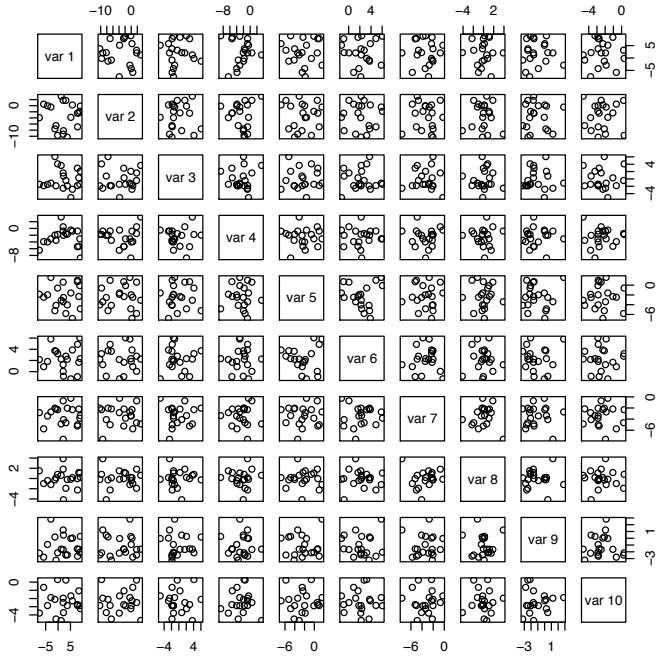
```
> X <- as.matrix(dune)
> V <- eigen(cov(dune))$vector
> PC <- X %*% V
> PC[1:5, ]
```

```
         [,1]       [,2]      [,3]       [,4]       [,5]     [,6]
2   7.8632333 -6.9156455 -1.802930 -0.7592717  1.0038289 4.880190
13 -0.1837002 -9.7166345  4.098879 -1.7969386 -0.5913193 3.223097
4   2.0525466 -9.3132762 -2.539806 -0.7808169 -3.3998013 1.247622
16 -7.5610307 -3.9151554 -1.455246 -6.4509398 -1.8612388 2.165339
6   8.9943182 -0.1316022  2.075440 -8.7597034 -4.8445081 2.326789
        [,7]       [,8]       [,9]      [,10]      [,11]     [,12]
2  -5.013353 -2.2455974 -2.6476661 -2.8427649 -0.4593202 1.2118617
13 -2.296845  0.8166063 -2.1470136  0.3124062 -2.8458103 2.1506161
4  -7.624091  3.8758407  1.2243159 -1.9621888 -1.0915741 0.6602083
16 -2.827212  1.5025837 -2.2036058 -3.7778249  0.3165852 0.5928802
6  -3.365366  0.2061268  0.2744863 -3.2406152 -0.9433286 2.7334334
         [,13]     [,14]       [,15]     [,16]      [,17]      [,18]
2  -0.40358193 0.1596677  1.22251362 1.0052880  0.4765954 -0.6637914
13 -0.73115235 1.6597452 -0.03941401 1.9043549 -0.3642813 -0.7397268
4   0.12584658 1.6981585  0.38371399 0.9805425  0.3957807 -0.8773133
16  0.08986328 1.6755371  1.84847618 2.3480531  0.1527719 -0.4932394
6   0.67422064 0.9965782 -0.47402683 1.3346653  0.3012177 -0.5218971
       [,19]     [,20]     [,21]    [,22]     [,23]     [,24]     [,25]
2  0.8675266 0.4202482 0.6969981 0.349236 -1.300027 0.2059339 0.6115661
13 0.4284945 0.4202482 0.6969981 0.349236 -1.300027 0.2059339 0.6115661
4  0.4281225 0.4202482 0.6969981 0.349236 -1.300027 0.2059339 0.6115661
16 0.3784169 0.4202482 0.6969981 0.349236 -1.300027 0.2059339 0.6115661
6  0.6381589 0.4202482 0.6969981 0.349236 -1.300027 0.2059339 0.6115661
        [,26]      [,27]      [,28]     [,29]     [,30]
2  -0.8477374 -0.1990721 -0.8456182 0.1189631 0.5027105
13 -0.8477374 -0.1990721 -0.8456182 0.1189631 0.5027105
4  -0.8477374 -0.1990721 -0.8456182 0.1189631 0.5027105
16 -0.8477374 -0.1990721 -0.8456182 0.1189631 0.5027105
6  -0.8477374 -0.1990721 -0.8456182 0.1189631 0.5027105
```

And, if we were so inclined, we could plot all pairs of the first 10 principal components using the `pairs` function:

```
> detach(package:vegan)
> rm(list = ls())
```

```
> pairs(PC[, 1:10])
```



5 Information Theory Based Multi-Model Inference

It is far too often in ecology that the model or hypothesis set cannot be narrowed to a dichotomous null-hypothesis test. Without getting to much into the nitty-gritty details, Information Theory provides a framework with which to simultaneously compare multiple competing models by weighting the performance of a model by its complexity (i.e. the number of model parameters). Say for instance that we are comparing five regression models of the effect of various factors on species richness from the *dune* and *dune.env* data-sets in the vegan package:

MODELS

1. Interaction - Richness = Moisture + Manure + Moisture x Manure
2. No Interaction - Richness = Moisture + Manure
3. Moisture Only - Richness = Moisture
4. Manure Only - Richness = Manure
5. Null (intercept Only) - Richness = Intercept

We can get our richness estimate from the *dune* data by using the `specnumber` function:

```
> library(vegan)
> data(dune)
> data(dune.env)
> R = specnumber(dune)
```

Once we have our data and our models of interest. The steps for conducting the analysis are:

1. Fit the models (here we use the `aov` function
2. Generate likelihood based information criteria (in this case Akaike's Infromation Criterion (AIC))
3. Calculate Akaike Weights (W) (here we right a function to do this for use)

```
> model1 <- aov(R ~ Moisture + Manure + Moisture * Manure,
+     data = dune.env)
> model2 <- aov(R ~ Moisture + Manure, data = dune.env)
> model3 <- aov(R ~ Moisture, data = dune.env)
> model4 <- aov(R ~ Manure, data = dune.env)
> model5 <- lm(R ~ 1, data = dune.env)

> aic.values <- c(AIC(model1), AIC(model2), AIC(model3),
+     AIC(model4), AIC(model5))

> weights <- function(ic = "vector of information criterion values") {
+     d <- numeric()
+     for (i in 1:length(ic)) {
+         d[i] <- ic[i] - min(ic)
+     }
+     exp.d <- numeric()
+     for (i in 1:length(d)) {
+         exp.d[i] <- exp(-d[i]/2)
+     }
+     w <- numeric()
+     for (i in 1:length(exp.d)) {
+         w[i] <- exp.d[i]/sum(exp.d)
+     }
+     model <- seq(1, length(ic))
+     out <- data.frame(model, ic, d, exp.d, w)
+     colnames(out) <- c("Model #", "Info. Crit.", "D", "Exp(-D/2)",
+         "Weights")
+     out <- out[order(out$Weights, decreasing = TRUE), ]
+     rownames(out) = seq(1, nrow(out))
+     return(out)
+ }
> w.out <- weights(aic.values)
> w.out
```

	Model #	Info. Crit.	D	Exp(-D/2)	Weights
1	1	85.69719	0.000000	1.000000000	0.977879763
2	4	94.94671	9.249522	0.009805996	0.009589085
3	5	94.95225	9.255069	0.009778840	0.009562530
4	2	98.66867	12.971484	0.001525029	0.001491295
5	3	98.68749	12.990304	0.001510745	0.001477327

From this we can see that the clear winner is Model 1 or the Interaction Model, which has the highest Akaike weight.

Introduction to Ecological Analyses in R (Day 5): Monte-Carlo Methods and Bayesian Beginnings

Matthew K. Lau

1 Monte Carlo Simulations: Coins, Dice and Kangaroos

Back up and forget everything you know about statistics. What do we want to know? Answers to ecological questions. Statistics is a means to make probabilistic statements related to questions of interest, when we don't have complete information. The statistics that are typically taught in introductory courses or frequentist-based, parametric statistics. That is certain aspects about the true population values of interest, such as the form of the distribution, can be assumed. However, there are many instances where we either cannot assume a distributional shape or, even if we do, analytical solutions for mathematical quantities are not approachable or even possible. Monte-Carlo methods provide a way to get around such analytical limitations and have made many non-parametric and Bayesian analyses possible.

Monte-Carlo (MC) methods are rooted in gambling probability (hence the name in reference to the famous gambling city in Europe). Simply, MC methods repeat a physical process in order to obtain quantitative estimates of the probability of particular events. This can be done in "reality" or by simulation. Simulation provides a much more practical and consistent means. At the heart of the simulation is an algorithm (i.e. mathematical representation) that represents the physical process. Here we go over several simple examples of MC methods.

1.1 Coin Games: simulating a coin flip

1.1.1 What is the probability of flipping a heads with a fair coin?

First, we need to make representation of our "fair" coin, which will have two sides that have equal probability of coming "up" when the coin is flipped. We can do this by creating an object with two values that will represent heads and tails:

```
> coin <- c("H", "T")
```

Next, we need to be able to "flip" our coin. This can be achieved by using the `sample` function:

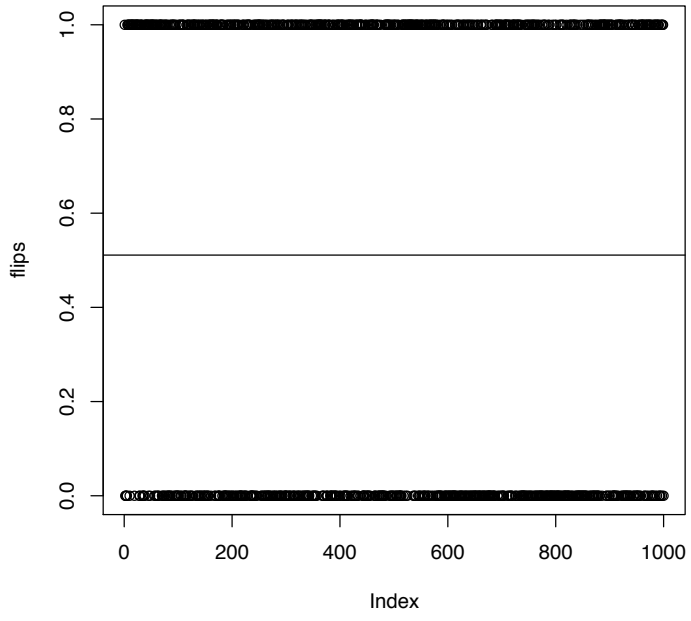
```
> sample(x = coin, size = 1, replace = FALSE, prob = NULL)
```

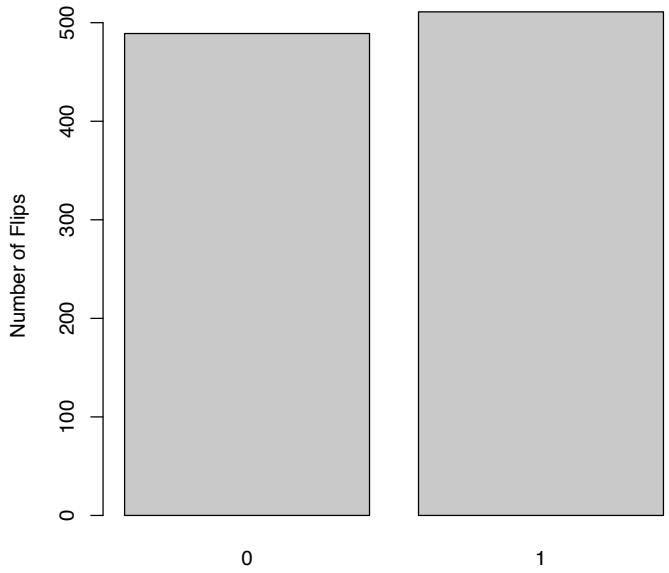
```
[1] "H"
```

Here you can see that the function drew a "sample" of size one from our object "coin." The latter two arguments specify whether or not to choose previously chosen values from the object when the sample size is larger than one and the probability of sampling a particular value in the object being sampled where the default (i.e. NULL) applies equal probability to all values.

Last, we use a for-loop to apply an algorithm that simulates many "flips" of our coin (i.e. repeated sampling of our coin vector).

```
> coin <- c(0, 1)
> flips <- numeric()
> for (i in 1:1000) {
+     flips[i] <- sample(x = coin, size = 1, replace = FALSE, prob = NULL)
+ }
> plot(flips, type = "p")
> abline(h = mean(flips))
```





Looking at the first plot, we can see that the outcome of each one of our flips tend to come up pretty evenly as tails (=0) and heads (=1). The center line here shows the mean value of our simulation, which in this case would be equivalent to the probability of the flip coming up heads. We can calculate the probability of either heads or tails by counting the number of flips that produced either one and dividing by the total number of flips:

```
> p.heads = length(flips[flips == 1])/length(flips)
> p.tails = length(flips[flips == 0])/length(flips)
> cbind(p.heads, p.tails)
```

```
     p.heads p.tails
[1,]   0.511   0.489
```

Both of these probabilities are very close to the analytical solutions: P(heads) = 1/2 = 0.5 and P(tails) = 1/2 = 0.5. However, the values are just a little off of the mark. This is because there is a small amount of random error inherent to the MC algorithm. Theoretically, as we increase the number of simulations toward infinity, this error would diminish to zero. It is generally recommended to not only run a large (>1000) number of simulations, but also assess the MC error of the simulation by looking at the variance of multiple MC simulations (i.e. repeat the MC simulation over and over, each time calculating the probability value of interest and then calculate the variance of the simulated probabilities). This is an instance where writing functions can really help to reduce the length of the script!

1.2 Dice: craps simulation

Here is another example where the analytical solution is easy to obtain, giving us an easy way to confirm our MC simulation. Say we are going to Las Vegas and we would like to try our luck at the craps table. Let's figure out the probability of rolling a 7 or 11 in one roll of two dice.

This is easy to do analytically:

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

The above table shows all of the possible outcomes of one roll of two dice. From this we can calculate the analytical solution which is P(7) + P(11) = 6/36 + 2/36 = 0.17 + 0.06 = 0.23.

Now let's make a simulation to compute this probability using a strategy similar to the coin flip simulation we did above.

1. Make two "fair" dice
2. Simulate 1000 rolls of the two dice
3. Calculate the number of times either a 7 or 11 come up

```
> die1 <- c(1, 2, 3, 4, 5, 6)
> die2 <- c(1, 2, 3, 4, 5, 6)
> roll <- numeric()
> n <- 1000
> for (i in 1:n) {
+     d1 <- sample(die1, 1)
+     d2 <- sample(die2, 1)
+     roll[i] <- d1 + d2
+ }
> length(roll[roll == 7])/length(roll) + length(roll[roll == 11])/length(roll)

[1] 0.203
```

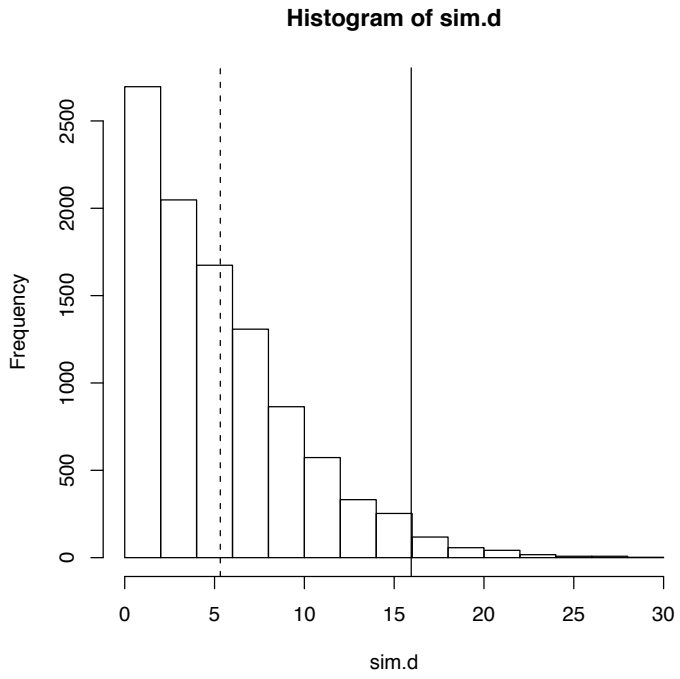
As you can see this MC simulation solution is very close to the analytical solution, but just a little off, as with the coin example above.

1.3 Monte Carlo Test of Two Samples from the Same Population: Boxing Kangaroos

A well-respected biologist suggests to you that kangaroos from W. Australia are better boxers (i.e. better intra-specific competitors) than kangaroos from E. Australia, due to selective pressure from greater resource limitation in the West. You set out to test this hypothesis. After collecting kangaroo boxing ability estimates, which continuous and range from zero to infinity, from both sides of the continent, you wish to test the hypothesis that the western population has no better or worse boxing ability than the eastern population.

1.3.1 We can test this with a Monte-Carlo simulation, where we simulate samples from a common (i.e. "Null") distribution.

```
> W.ability <- c(25, 32, 80, 10, 22, 23.5, 20, 19, 23, 19.5)
> E.ability <- c(10, 11, 12, 11, 10.5, 9, 12, 10, 19, 10)
> null <- c(W.ability, E.ability)
> n = 10000
> obs.d <- abs(mean(W.ability) - mean(E.ability))
> sim.d <- numeric()
> for (i in 1:n) {
+     W <- sample(null, 10, replace = TRUE)
+     E <- sample(null, 10, replace = TRUE)
+     sim.d[i] <- abs(mean(W) - mean(E))
+ }
> length(sim.d[sim.d >= obs.d])/n

[1] 0.0264

> p.value <- length(sim.d[sim.d >= d])/n
> p.value

[1] 0.0264

> hist(sim.d)
> abline(v = c(mean(sim.d), obs.d), lty = c(2, 1))
```



Taking a closer look at the simulation, what's happening is that we mixed together all of our observations and drawing two samples, like we did in pseudo-reality, from this over and over (n=1000 times), each time calculating a value for our statistic (d), which is the absolute value of the difference between our means from our two samples. We can then use this to as a simulated null distribution, which we then use to calculate our p-value by counting the number of times that the simulated d is greater than or equal to our observed d and dividing by the total number of simulations.

Although MC methods are non-parametric (i.e. there is no assumptions about the population distribution), there are still assumptions that are important to be aware of:

1. The samples are representative (i.e. random),
2. The Monte-Carlo algorithm is actually doing what we want (this is often not always possible to assess).

For more information on MC methods, see this online bibliography.

2 The Bayesian Stats MCMC Revolution

Bayesian Statistics are characterized by their use of a subjective definition of probability: the degree of belief in the occurrence of an event. This is distinct and more general than the Frequentist definition of probability, which is based

on the long run frequency of an event. The results of Bayesian Statistical approaches are often more intuitive, since they typically give us direct estimates of our hypotheses of interest, rather than awkwardly tiptoeing around with p-values in the Null Hypothesis Testing framework that is the most prevalent approach. Also, Bayesian methods are logically in line with the scientific method, because they develop a subjective estimate of probability, as a degree of belief, in a particular hypothesis by quantitatively modifying prior beliefs (prior probability) with current observations (likelihood) to generate an updated probability estimate (posterior probability).

But why talk about Bayesian methods in conjunction with MC methods? Because, MC methods are at the core of a modern revolution in Bayesian Statistics. Observe Bayes' theorem, which allows one to calculate the posterior probability:

$$Posterior = \frac{Prior * Likelihood}{Marginal\ Probability}$$

Ignoring the denominator for a second, the numerator shows us that Posterior is simply a modification of the Prior by the Likelihood, which is our data that we have in hand. In other words, we get an estimate of the probability of our hypothesis of interest, which must be quantitative, collect data and calculate the Likelihood, which is the probability of observing our data given our hypothesis, multiply these two together. This however only produces a value proportional to the Posterior probability. To get the true Posterior probability estimate, we must divide by the Marginal Probability, which is simply a normalizing constant.

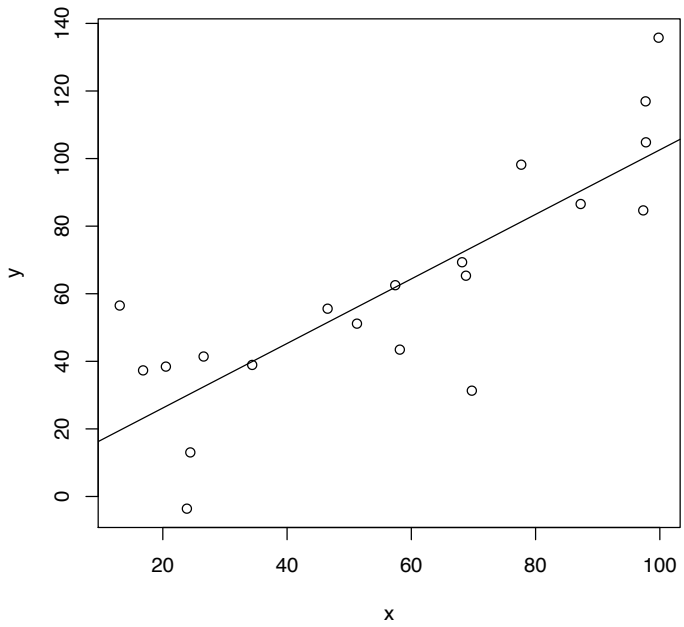
Until MC methods were developed to calculate the Marginal Probability, Bayesian Statistics was often limited, because an analytical solution for the Marginal Probability were intractable at best, and analytically impossible at worst. The development of MC methods, specifically Markov Chain Monte-Carlo (MCMC), have allowed for easy, quick computation of the Marginal Probability.

2.1 Example: Bayesian Regression

Here, we use functions in the *MCMCpack* package to conduct a Bayesian regression.

First we create a simple data set.

```
> x <- runif(20, 10, 100)
> y <- rnorm(20, x, 4^2)
> plot(y ~ x)
> abline(lm(y ~ x))
```



Now, we load up our *MCMCpack* functions and conduct our Bayesian regression:

```
> library(MCMCpack)
> mcmc.output <- MCMCregress(y ~ x)
> xtable(summary(mcmc.output)[1]$statistics, caption = "Bayesian Results1")
```

	Mean	SD	Naive SE	Time-series SE
(Intercept)	7.22	10.59	0.11	0.10
x	0.95	0.17	0.00	0.00
sigma2	465.97	178.26	1.78	2.35

Table 1: Bayesian Results1

```
> xtable(summary(mcmc.output)[2]$quantiles, caption = "Bayesian Results2")
```

	2.5%	25%	50%	75%	97.5%
(Intercept)	-13.53	0.49	7.30	13.98	28.29
x	0.62	0.85	0.95	1.06	1.28
sigma2	236.90	343.20	428.94	543.97	905.77

Table 2: Bayesian Results2

```
> xtable(summary(lm(y ~ x)), caption = "Frequentist Results")
```

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	7.0849	9.9123	0.71	0.4839
x	0.9550	0.1550	6.16	0.0000

Table 3: Frequentist Results

What has happened here is that a MC procedure utilizing a Markov-Chain algorithm has generated posterior probability estimates of our paramters. In this case these are the intercept (which we are not particularly interested in in this situation), the slope and the variance of the slope parameter. We can compare the Bayesian results (upper) to a least squares regression analysis (lower) and see that we come to very similar conclusions. In the case of the Bayesian inference, we get a posterior probability distribution for values of our slope parameter, which can be used to calculate credibility intervals, which can be interpreted as the probability of the true value being within that interval.

We could conclude here that the true slope parameter is not equal to zero with 95% probability, which a similar, but not identical, conclusion that we would come to with the frequentist, null hypothesis approach. However, there is more that can be done with the posterior probability distribution. Two things, which

I will discuss briefly and refer you to the web to learn more about, are multi-model inference and prediction. Bayesian multi-model inference is very similar to frequentist based multi-model inference (often referred to as AIC for the most widely used information criterion, Akaike's Information Criterion) except that the posterior probabilities would be used to compute estimates of relative model performance. Also, the posterior probability distribution can be used to generate predictions about our parameters of interest. This allows one to use all available information at hand in a systematic analytical framework to predict future values, which is especially useful for reducing variance in estimates by using prior information to augment noisy data.

For a great introduction to Bayesian applications in ecology, read either (or both) of these books:

Bayesian Methods for Ecology by Michael A. McCarthy

A Primer of Ecological Statistics by Nicholas Gotelli and Aaron Ellison