Analysis of Environmental Data

Introduction to R

(Written by Kevin McGarigal, but borrowed heavily from Dave Roberts, University of Montana)

The purpose of this lab exercise is to familiarize you with the R language and environment for statistical computing. Specifically, you will learn about variable types and data structures in R, vector and matrix operators in R, functions in R, getting data into and out of R, plotting in R, and libraries and packages in R. Note, these skills are the minimum essentials for statistical data analysis using R; however, additional skills will be needed to become proficient in the use of R. Here is outline of what is included in this lab exercise:

1. What is R?

R is a complete programming language and statistical computing environment for data manipulation, calculation and graphical display. Technically, the language is called S, and R is the open source implementation available for many systems for free.

R is free software because it is part of the GNU project.

R is free to run, copy, distribute, study, change and improve.

The R foundation guides its development (www.r-project.org).

The major difference between R and other environments (e.g., SAS) is that:

- In R, all objects are stored in virtual memory (data limited to memory size).
- In R, results of statistical analyses are stored in objects.
- SAS and other environments will give lots of output, but can't manipulate most output; whereas R will give minimal output and stores the results in a fit object for subsequent interrogation by other functions.

R is exceptional statistical software for ecological analysis as it includes a broad range of analyses employed in ecological analysis, as well as numerous routines for exploratory data analysis (EDA).

The syntax of R is moderately quirky unless you are a C programmer. The following is a general guide to R with hints for performing the exercises in in the accompanying labs.

2. Installation of R

R can be downloaded and installed from any of many available CRAN sites from the R foundation website (www.r-project.org).

From the R-project website, click on the CRAN (Comprehensive R Archive Network) link for 'Downloads' and choose a CRAN mirror closest to you. Choose an operating system (e.g., Windows) from the 'Download and Install R' list and select 'base'. Click on the appropriate setup program for the most recent release (e.g., R-2.11.1-win32.exe) and follow the instructions on screen.

3. A Few Important Syntax Conventions in R

R is case sensitive - so be very careful in the use of upper and lower case.
/ - forward slash is used in all path names (as opposed to the backward slash '\').
' and " (single and double quotes) are used interchangeably as long as they are paired.
() refers to functions and contains the arguments of the corresponding function.
[] refers to indexing and references row and/or column elements of a data structure.

4. Variables and Types

Like most programming languages, R allows users to create variables, which are essentially named computer memory. For example, you may store the number of species in a sample in a variable. Variables are identified by a name assigned when they are created. Names should be unique, and long enough to clearly identify the contents of the variable. You may work with the same data weeks or months later, and variable names like x or data are not very helpful.

Variable names can consist of letters, numbers, and the character ".". They may not start with a number, or include the characters "$" or "_" or any arithmetic symbols as these have special meaning in R.

Variables are assigned a value in an assignment statement, which in R has the variable name to the left of a left-pointing arrow (typed with the "less than" followed by a "dash") with the value behind the arrow. For example,

```
number.species <- 137
```

Recent versions of R allow you to use the = sign for an assignment, i.e. number.species = 137 but I will stick with the older, more elegant arrow.

R allows the creation of variables which contain numeric values (both integers and floating point or real numbers), characters, or special characters interpreted as "logical" values. For example

```
pi<-3.14159
small.value<-1.0e-10
species.name<-'American robin'
conifer<-TRUE
```

Notice that real or floating point numbers can be entered with just a decimal point, or in exponential notation, where 1.0e-10 means .0000000001. Notice also that character variables, called "strings" should be entered in quotes (single or double, it doesn't matter as long as they match). Finally, note that the word TRUE is NOT surrounded by quotes. This is not the WORD TRUE, but rather the VALUE TRUE. Logical variables can only take the values TRUE or FALSE.

Unlike many programming languages (e.g., FORTRAN or C) you do not have to tell R what kind of value (integer, real, or character) a variable will contain; it can tell when the variable is assigned. R will only allow the appropriate operations to be performed on a variable. For example:

```
species.name + 37
Error in species.name + 37 : non-numeric argument to binary operator
```

R did not allow us to add 37 to species.name because species.name was a character variable.

5. Data Structures

R is a 4th generation language, meaning that it includes high-level routines for working with data structures, rather than requiring extensive programming by the analyst. In R there are 4 primary data structures we will use repeatedly.

1. *vectors* --- vectors are one-dimensional ordered sets composed of a single data type. Data types include integers, real numbers, and strings (character variables).

2. *matrices* --- matrices are two dimensional ordered sets composed of a single data type, equivalent to the concept of a matrix in linear algebra.

3. *data frames* --- data frames are one to multi-dimensional sets, and can be composed of different data types (although all data in a single column must be of the same type). In addition, each column and row in a data frame may be given a label or name to identify it. Data frames are equivalent to a flat file database, and similar to spreadsheets. Accordingly, we often refer to specific columns in a data frame as "fields."

4. *lists* --- lists are compound objects of associated data. Like data frames, they need not contain only a single data type, but can include strings (character variables), numeric variables, and even such things as matrices and data frames. In contrast to data frames, lists items do not have a row-column structure, and items need not be the same length; some can be a single values, and others a matrix. It's a little hard to imagine how lists operate in the abstract, but you will see that many of the results of analyses in R are returned as lists, so we'll introduce them as necessary that way.

5.1 Vectors and Matrices

Vectors, matrices, data frames and lists are identified by a name given the data structure at the time it is created. Names should be unique, and long enough to clearly identify the contents of the structure. Names can consist of letters, numbers, and the character ".". They may not start with a number, or include the characters "$" or "_" or any arithmetic symbols as these have special meaning in R.

Vectors are often read in as data or produced as the result of analysis, but you can produce one simply using the c() function, which stands for "combine." For example

```
demo.vector1<-c(1,4,2,6,12)
```

produces a vector of length 5 with the values 1, 4, 2, 6, 12.

Individual items within a vector or matrix can be identified by subscript (numbered 1 - n), which is indicated by a number (or numeric variable) within square brackets. For example, in the demo.vector:

```
demo.vector1[4]
```
= the fourth element of demo.vector, 6

A *factor* is a special type of vector object. The vector data type is typically either integer or character, and the vector can be assigned to a special class called "factor", which is used to distinguish variables that can have a finite number of values (gender, social class, etc.). A factor object has a number of attributes that distinguish it from other object types. For example, a factor may be purely nominal or may have ordered categories, and it may contain a "contrasts" attribute which controls the parameterization used when the factor is used in a modeling function. We will not worry about these details; for now, what is important is that you recognize that some vectors can be classified as "factors" and this will influence how they are interpreted and used in many statistical functions. It is also important to recognize that any variable in a data set that contains a character (i.e., non-numeric value) will automatically be classified as a factor by R when the data is imported into R.

A *matrix* can be created by simply binding together two or more vectors of the same type and length. For example, if we create a second demo.vector

```
demo.vector2<-c(4,2,1,2,4)
```

we can then bind the two vectors together using the cbind() function to create a matrix

demo.matrix<-cbind(demo.vector1,demo.vector2)

Matrices are specified in the order "row, column", so that

demo.matrix[4,2] = row 4, column 2 in matrix demo.matrix

Individual rows or columns within a matrix can be referred to by implied subscript, where the value of the desired row or column is specified, but other values are omitted. For example,

demo.matrix[,2] = second column of matrix demo.matrix

represents the second column of matrix demo.matrix, as the row number before the comma was omitted.

Similarly,

demo.matrix[5,] = row 5 of matrix demo.matrix

represents row 5, as the column after the comma was omitted.

In addition, a number of specialized subscripts can be used.

demo.matrix[] = all rows and columns of matrix demo.matrix
demo.vector1[1:3] = demo.vector1[1] through demo.vector1[3]
demo.vector1[-1] = all of vector demo.vector1 except demo.vector[1]
demo.matrix[1:3,2] = a submatrix of demo.matrix from row 1 to 3 and column 2

It's even possible to specify specific subsets of rows and columns that are not adjacent.

demo.matrix[c(1,2,5),2] = a submatrix consisting of rows 1,2 and 5, and columns 2.

5.2 Data Frames

Data frames can be accessed exactly as can matrices, but can also be accessed by data frame and column or field name, without knowing the column number for a specific data item. For illustration, let's load the birds.csv dataset (note, we will come back to importing and exporting data later):

First, let's set the default working directory to the folder containing the birds.csv file using the setwd() function, as follows (but note that the path specified should correspond to where the file is located on your machine):

setwd('c:/work/stats/ecodata/lab/R.intro/')

Next, read in the birds.csv dataset:

birds<-read.csv('birds.csv',header=TRUE)

Because the variable types are mixed in this incoming data set (containing both numeric and character fields), the data structure will be classed as a data frame automatically.

In the birds dataset, there is a column labeled 'AMRO' that holds the abundance of American Robin on each sample plot. This column can be accessed as:

```
birds$AMRO
```

where 'birds' is the name of the data frame, 'AMRO' is the name of the field or column of interest, and the '$' is a separator to distinguish data frame from field. If you are routinely working with one or a few data frames, R can be told the name(s) of the data frames in an 'attach' statement, and the data frame name and separator can be omitted. For example, if we give the command

```
attach(birds)
```

we can specify the field 'AMRO' simply as

```
AMRO
```

rather than 'birds$AMRO.' This is more concise notation, but means that we cannot have a variable with the same name in another attached data frame. Data frames are extraordinarily useful in R.

5.3 Lists

As noted above, a list is a compound object composed of associated data. Items within a lists are generally referred to as components. Similar to data frames, components in a list can be given a name, and the component can be specified by name at any time. In addition, components can be specified by their position in the list, similar to a subscript in a vector. However, in contrast to a vector, lists components are specified in double [[ ]] delimiters. We will ultimately find it quite handy to create our own lists, but for the first few labs we will just see them as results from analyses, so we'll take them as they come and demonstrate their properties by example.

For the time being, I'll give a very simple example. Using the demo.vector1 vector above, and the names of the birds data frame.

```
list.demo<-list(demo.vector1,names(birds))  — creates the list from the two components
names(list.demo)<-c('species per plot','field names') — assigns names to the two list components
list.demo — prints the list to the console
$`species per plot`
[1]  1  4  2  6 12
$`field names`
 [1] "BASIN" "SUB"   "BLOCK" "AMGO"  "AMRO"  "BCCH"  "BEKI"  "BEWR"  "BGWA"
[10] "BHGR"
```

In this case, the first component 'species per plot' has 5 numbers, and the second component has 10 strings.

5.4 Checking objects

All data objects (vectors, matrices, data frames and lists) have an internal structure that defines the type of object (e.g., data frame) and its components. It is important to always be aware of the structure of the objects you are working with, and this is especially so with data frames which can contain mixtures of data types (e.g., integers, real numbers, factors) and lists which can contain mixtures of object types.

One way to view the structure of an object is via the str() function. For example, we can view the structure of the birds data frame as follows:

```
str(birds)
```

Note, the result tells us that the object is a data frame containing 32 observations and 10 variables. Furthermore, it lists each variable (columns), in order, the associated class or data type (e.g., factor or integer in this case), and a sample of the first 10 values. For factors, it also gives the number of levels (i.e., unique values) and what they are. This information can be incredibly useful in determining why some operation doesn't work, because it usually has to do with having an ineligible data type.

For large data sets, it is sometimes useful to view a portion of the object to make sure it looks right. Two functions, head() and tail() will print out the first and last 6 six records of the object, respectively, although you change the number of records to print by adding an argument to the function. For example, we can print out the first 10 records of the birds data frame as follows:

```
head(birds,10)
```

Lastly, it is sometimes useful to view the entire dataset in a spreadsheet-like format. The function fix() can be used for this purpose as follows:

```
fix(birds)
```

This function opens the data object in a data editor window that has the look and feel of a spreadsheet. You can browse the object and/or edit the values. Importantly, when you close the editor, any changes that you made will be preserved in the object in memory. However, changing values in the object stored in memory does not change the original data file stored on disk, if there is one. The change is made to the object stored in memory and is preserved only for the duration of the R session.

6. R Operators

Because R is a 4th generation language, it is often possible to perform fairly sophisticated routines with little programming. The key is to recognize that R operates best on vectors, matrices, or data fames, and to capitalize on that. A large number of functions exists for manipulating vectors, and by extension, matrices and data frames (usually). For example, birds is a bird abundance data frame containing 32 sample plots (rows) and 10 fields (columns). The first 3 fields (columns) contain plot

identifiers. The first two are character fields (BASIN and SUB) and the third field is numeric (BLOCK). The remaining 7 fields (columns) are numeric and contain abundances for 7 different bird species. Given this data structure, we can perform the following:

max(birds[,5]) --- assigns the maximum value of the second species (fifth column) among all plots.

Alternatively, because birds is a data frame, we can accomplish the same thing with the following:

max(birds$AMRO)

sum(birds[,5]) --- assigns the sum of second listed species abundance in all plots to y
log(birds[,4:10]+1) --- assigns the log of the respective values in columns 4 through 10 in birds (+1 to avoid log(0) which is undefined)

In addition, R supports logical subscripts, where the subscript is applied whenever the logical function is true. Logical operators include:

> for 'greater than'
>= for 'greater than or equal to'
< for 'less than'
<= for 'less than or equal to'
== for 'equal to'
!= for 'not equal to'
& for 'and'
| for 'or'

For example,

sum(birds[,5]>1) --- assigns the number of plots where the abundance of the species in column 5 is greater than 1 (birds[,5]>1 is evaluated as 1 (true) or 0 (false), so that the sum is of 0's and 1's).

sum(birds[,5][birds[,5]>1]) --- assigns the sum of the abundance for the species in column 5 in plots where species in column 5 has abundance greater than 1.

max(birds[,5][birds$BHGR==5]) --- assigns the maximum abundance for the species in column 5 for plots with the abundance of BHGR equal to 5.

6.1 Missing Values

A final special case is of special note. Missing values in a vector or matrix are always a problem in ecological data sets. Sometimes it is best simply to remove samples with missing data, but often only one or a few values are missing, and it's best to keep the sample in the matrix with a suitable missing value code. We'll discuss how to deal missing values in more detail in the next lab, but for now lets assume that we want to drop the missing values from a vector. First, select the fourth column from the birds data frame, which contains a single missing value:

```
x<-birds[,4]
```

To use all of the vector EXCEPT the missing value, use:

```
y<-x[!is.na(x)]
```

That's complicated enough to merit some discussion. The R function to identify a missing value is:

```
is.na( )
```

so that to say all of a vector except missing values, we set a logical test to be true when values are not missing. Since the R operator for 'not' is !, the correct test is:

```
!is.na( )
```

and to specify which vector we're testing for missing value, we put the vector in parentheses as follows:

```
!is.na(x)
```

Accordingly, the full expression is

```
x[!is.na(x)]
```

While the symbol for a missing value in a vector or matrix is NA, using

```
x[x!='NA']
```

will NOT work.

Since dropping missing records is a very common operation, R has a built in function, na.omit(), that makes it even easier to accomplish the same thing as above. Try the following:

```
na.omit(x)
```

This use of missing values is critical to R because all operations on vectors or matrices must have the same number of elements. So, if there are missing values in any field we're using in a calculation, the same record (row) must be omitted from all the other fields as well.

7. Creating Subsets of a Matrix or Data Frame

It is frequently useful or necessary to subset a data set by selecting certain columns, rows or both. For example, you may need to select a set of numeric variables (columns) from a data frame containing a mixture of variable types in order to conduct subsequent analyses that require numeric data. Or you may wish to select a set of observations (rows) that meet certain criteria. There are many ways to subset a data set depending on whether you are selecting columns or rows and

whether the data structure is a matrix or data frame. Here we will assume that the initial data structure is a data frame, since this is what we will typically be working with. Here are several options for creating a subset of the initial data set:

Selecting a suite of columns:
birds.new<-birds[,4:10] — select all rows and columns 4 through 10
birds.new<-birds[,-c(1:3)] — same
birds.new<-subset(birds,select=AMGO:BHGR) — same
birds.new<-birds[,c(4,5,10)] — select all rows and columns 4,5 and 10
birds.new<-subset(birds,select=c(AMGO,AMRO,BHGR)) — same

Selecting a suite of rows:
birds.new<-birds[1:4,] — select rows 1 through 4 and all columns
birds.new<-birds[-c(5:10),] — same
birds.new<-birds[c(4,5,10),] — select rows 4,5 and 10 and all columns
birds.new<-subset(birds,subset=SUB=='AL') — select all rows where SUB=='AL'
birds.new<-subset(birds,subset=BLOCK>10) — select all rows where BLOCK>10
birds.new<-subset(birds,subset=BLOCK>10&SUB=='AL') — select rows meeting both criteria

Selecting a suite of rows and columns:
combine any of the above

8. Row or Column Operations on a Matrix or Data Frame

Vector operators can be applied to every row or column of a matrix to produce a vector with the apply() function. For example:

bird.max<-apply(birds[,4:10],2,max) —creates a vector 'bird.max' with the maximum value for each species (in column 4 through 10) in its respective position.

bird.max — prints the vector to the console.

```
AMGO AMRO BCCH BEKI BEWR BGWA BHGR
  NA    5    2    5    1   NA   25
```

Note that the max value for AMGO and BGWA are given as NA. This is because by default the max function returns a missing value (NA) if any of the vector elements are missing. If we want to find the max values, ignoring the missing values, we have to add a special argument to the function, as follows:

```
bird.max<-apply(birds[,4:10],2,max,na.rm=TRUE)
bird.max
AMGO AMRO BCCH BEKI BEWR BGWA BHGR
   1    5    2    5    1    2   25
```

The 'na.rm=TRUE' tells the function to remove missing values before finding the max.

The apply operator is employed as:

apply('matrix name',1(rowwise) or 2(columnwise),vector operator)

so that

```
bird.sum<-apply(birds[,4:10],1,sum,na.rm=TRUE)
bird.sum
 [1] 26  4 10  3  7  5  3  3  3  3  5  5  6  8  6  6  5  5  6  7  7  3  9  7  7
[26]  7  1  3  3  7  5  5
```

creates a vector of total species abundance in each plot. The vector is as long as the number of rows in the birds data frame.

We can also use an expression within the apply function, for example to sum the number of plots where species x is greater than 0, and x is assigned to each column (species) in turn,

```
bird.sum<-apply(birds[,4:10]>0,2,sum,na.rm=TRUE)
bird.sum
AMGO AMRO BCCH BEKI BEWR BGWA BHGR
   1   11    3    1    2   20   32
```

where the birds[,4:10]>0 converts the birds data frame columns 4 through 10 to a matrix of TRUE and FALSE, and the sum() function treats TRUE as 1 and FALSE as 0.

9. Functions in R

A function consists of a name and one or more parameters (or arguments) contained in parentheses that are required to process the function. A simple function that we have already used is the sum() function, which returns the sum of all the values present in its arguments. In its simplest, sum() contains two arguments:

```
sum(x, na.rm=FALSE)
```

The first argument, x, is the data set (either a vector, matrix, or data frame containing all numeric variables) you wish to sum, and the second argument indicates whether missing values should be ignored. The default na.rm=FALSE will return NA if there are any missing values, whereas na.rm=TRUE will ignore the missing values when calculating the sum.

In the case of the birds data set, applying the sum function to the species abundance fields (columns 4-10) with the default argument of na.rm=FALSE, returns the following:

```
sum(birds[,4:10])
[1] NA
```

Note, there is no need to include the arguments if you wish to use the defaults provided. Applying

the sum function with the missing values argument set to TRUE, returns the following:

```
sum(birds[,4:10],na.rm=TRUE)
[1] 190
```

Note, in this case the sum is across all elements in all rows and in columns 4 through 10.

The apply() function we used above is a special function that allows us to apply other functions to each column or row of the matrix. In this case, we applied the sum() function to each species column of the birds data set and returned a vector of values containing the sum of abundance for each species.

When using functions it is important to understand the arguments of the function. The arguments of a function are all defined in the associated help file (see below). Each function has one or more named arguments. Some or all of the arguments may come with default values, in which case you do not need to specify any arguments inside the () when calling the function. However, in most cases one or more of the arguments will not have a default value and thus you must provide a value for the argument. For example, the sum() function requires that you specify a data set (an object, either a vector, matrix, or data frame containing all numeric variables). If you do not specify a value for this argument, you will get an error message.

In addition, if you specify values for arguments in the order that they are given in the written function, then the arguments do not need to be named explicitly in the function call. For example, in the apply() function, the following two calls are equivalent:

```
apply(X=birds[,4:10], MARGIN=2, FUN=sum)
apply(birds[,4:10],2,sum)
```

This is because in the second call the arguments are given in the same order as expected. If, however, you want to specify the arguments in a different order from the default, then the argument names must be included in the function call, e.g.:

```
apply(birds[,4:10], FUN=sum, MARGIN=2)
```

In practice, explicitly naming the arguments often is required when you want to only specify say the first and fourth argument of the function and accept the default values for the second and third. In this case, you do not need to name the first argument if given first in your call, but you must name the fourth argument. In general, I find it good practice to always name the arguments in a function call to make it explicit what you are doing, albeit at the cost of verbosity in your code.

Functions are essential to working with R. You will be using functions constantly to manipulate, summarize, analyze, and graphically display your data. For most of the things you will need to do in this course, functions have already been written by others and you will simply need to know how to call these functions and interpret their output. However, you can't work long in R without confronting the need to construct your own functions. In most cases, these will be functions that call or make use of existing R functions, but in particular ways suited to your applications.

Throughout this course, we will make extensive use of existing R functions to complete projects, but there may be a need or opportunity for you to create your own functions. Any time you issue a set of commands that you anticipate having to repeat or reuse in the future, you should consider writing a function. Although we will not go into the details of writing functions here, you can easily review the code for a function by simply typing the function name at the console.

10. Getting Data Into and Out of R

Getting data into any program is often the hardest part about using the program. For R, this is generally not true, as long as the data are reasonably formatted. The R Development Core Team has developed a special manual to cover the ins and outs of getting data into and out of R. It's available as a PDF or HTML at http://cran.r-project.org.

The easiest way is to format the data in columns, with column headings, and blanks or tabs between. For example:

```
BASIN SUB BLOCK AMGO AMRO BCCH BEKI BEWR BGWA BHGR
D     AL  1     0    1    0    5    0    NA   25
D     AL  2     NA   0    0    0    0    0    4
D     AL  3     0    5    2    0    0    2    1
etc.
```

The columns do not need to be straight, but multi-word variables like 'clay loam' need to be connected or put in quotes. The R convention (but it is just a convention) is to connect with a period. It CANNOT be connected with '$'. Recent versions of R allow connections with '_', but this will cause problems with earlier versions of R. The above file (if named 'birds.dat' for instance) could be read with the read.table command as follows:

```
birds<-read.table('.../birds.dat',header=TRUE)
```

where .../ is replaced with the full path to the folder containing the birds.dat file. Alternatively, it is customary to set the working directory at the beginning of a session to direct all reading and writing to the desired location, e.g., as follows:

```
setwd('c:/work/stats/ecodata/lab/R.intro/')
```

The resulting data frame would be named 'birds', and the columns would be named exactly as in the data file. Note that the value for BGWA in the third plot is NA. This is a missing value code, and will cause R to treat that value as missing, rather than as a code NA. It's possible to use other codes as missing values if you specify them in the read.table command. For example, suppose in your data set you used -999 as the missing value code. To tell R to set -999 to missing, add the na.strings= argument as follows:

```
bogus<-read.table('bogus.dat',header=TRUE,na.strings='-999')
```

Alternatively, data can be organized as in traditional spreadsheet 'csv' comma delimited files, as

follows:

BASIN,SUB,BLOCK,AMGO,AMRO,BCCH,BEKI,BEWR,BGWA,BHGR
D,AL,1,0,1,0,5,0,,25
D,AL,2,,0,0,0,0,0,4
D,AL,3,0,5,2,0,0,2,1
etc.

In which case it would be read:

```
birds<-read.table('birds.csv',header=TRUE,sep=',')
```

to tell R that the values were separated by commas. Alternatively, you can use

```
birds<-read.csv('birds.csv',header=TRUE)
```

to read the file, as read.csv() calls read.table() with the appropriate parameters as defaults.

If the data do not contain a header row with the field (col) names, the data can be read in and column names subsequently assigned. For example:

```
birds<-read.csv('birds.csv',header=FALSE,skip=1)
```

reads in the birds.csv file but specifies no header (treating the first row as data) and then to skip the first row (which in this case does actually contain the header), resulting in the input of just data with no column names. Once this is done (or in any case where column headings are absent), they can be entered separately with the names command. For example:

```
names(birds)<- c('BASIN','SUB','BLOCK','AMGO','AMRO',...)
```

Row names (such as plot IDs) can also be added if desired, using the row.names() function in a similar way.

There is one element of read.table() that sometimes causes problems. Ordinarily, read.table() will use the first column that contains all unique values as the row names/labels. Generally (but not universally) this is the first column. It is often best to explicitly specify which column contains row identifiers (as opposed to data), if you have them, using the row.names= specifier. For example, the birds.rowids.csv dataset contains row names in the first column. To read this in correctly, assigning the first column as the row names as opposed to data, we would do the following:

```
birds<-read.csv('birds.rowids.csv',header=TRUE,row.names=1)
```

The beauty of the read.table() function is the way it handles variables. If any value in a column is alphabetic, it treats the column as composed of 'factors,' or categorical variables. There is NEVER a reason to convert a categorical variable to numeric. However, if you already have categorical variables coded as integers, you can explain that to R with the as.factor() function after you read the

data in. If all values in a column are numeric, it treats that variable as numeric.

For example, let's say we wanted to convert the BLOCK field from an integer (numeric) to a factor (categorical), since it represents plot ID and does not contain any quantitative information:

```
birds$BLOCK<-as.factor(birds$BLOCK)
```

To verify that we made the desired change, let's check the structure of the birds object using the str() function, as follows:

```
str(birds)
'data.frame':   32 obs. of  10 variables:
 $ BASIN: Factor w/ 2 levels "D","N": 1 1 1 1 1 1 1 2 2 2 ...
 $ SUB  : Factor w/ 2 levels "AL","BC": 1 1 1 1 1 1 1 1 1 1 ...
 $ BLOCK: Factor w/ 16 levels "1","2","3","4",..: 1 2 3 4 5 6 7 8 9 10 ...
 $ AMGO : int  0 NA 0 0 0 1 0 0 0 0 ...
 $ AMRO : int  1 0 5 0 3 1 0 0 0 0 ...
 $ BCCH : int  0 0 2 0 0 0 0 0 0 0 ...
 $ BEKI : int  5 0 0 0 0 0 0 0 0 0 ...
 $ BEWR : int  0 0 0 0 1 0 0 0 0 0 ...
 $ BGWA : int  NA 0 2 2 2 2 2 2 2 2 ...
 $ BHGR : int  25 4 1 1 1 1 1 1 1 1 ...
```

Note that BLOCK is now a factor with 16 levels.

Getting data out of R is just as easy, if not easier, as getting data into R. While there are numerous options for outputting R objects, here we will focus on the most common task, outputting a data frame using the generic write.table() function.

For simplicity, let's take the birds data set that we just read into R and stored in the object named "birds" and write it back out as a comma-delimited file:

```
write.table(birds, 'out.csv', row.names=FALSE, sep=',')
```

This statement will write the object birds to a file named out.csv in the current working directory. Note, if you haven't set the working directory to the desired location using the setwd() function, you can include the full path in the write.table() function, as follows:

```
write.table(birds, 'c:/work/stats/ecodata/lab/R.intro/out.csv', row.names=FALSE, sep=',')
```

A few things about the write.table statement above function are worth noting. First, the write.table() function by default will overwrite a file by the same name if it exists in the output directory. So, be careful in naming the output file. On the other hand, if you wish to append the object to the existing file, you can include the append=TRUE argument in the statement.

Second, in the statement above, I set the row.names argument to FALSE (the default is TRUE)

which will omit the row names from the output file. I find that if I include the row names in the output file, then the column names (which are output by default) end up being shifted to the left one column (because there is no column name for the row names column.

Third, in the statement above, I included the sep= argument to specify exactly what I wanted to use as column delimiters in the output file. In this case, I set the separator to ',' which will produce a comma-delimited text file. For writing comma-delimited output files, you can also use the write.csv() which always uses a comma delimiter, which has the added advantage of not automatically including the row names in the output file, thus avoiding the need to specify the row.names argument to FALSE.

There are lots of other options for outputting data using the write.table() function, so it is worth familiarizing yourself with the help file for this function (?write.table).

Instead of outputting data in an ascii format that can easily be read by other programs, another option is to save the output in a uniquely R formatted file using the save() function. This allows you to save R objects in an optimized fashion that reduces the file size on disk and may make it more efficient to read and write very large files or complex objects such as lists and multi-dimensional arrays that don't collapse into flat tables. To save an R object with the default extension .RData and read it back in, try the following:

```
save(birds,file='c:/work/stats/ecodata/lab/R.intro/birds.RData')
load('c:/work/stats/ecodata/lab/R.intro/birds.RData')
```

11. Plotting in R

R has a powerful graphics capability that is much of the appeal to using the system. Many of the analyses have special plotting capabilities that allow you to plot results without storing multiple intermediate products. (R likes to point out that it is 'object oriented', and that this object orientation is what allows the generality of its plotting routines. While that is generally true, the SYNTAX of R is more appropriately viewed as functional, rather than object oriented, and we will concern ourselves largely with syntax, rather than implementation). R supports a fairly broad range of graphic devices in addition to excellent on-screen plotting. Reflecting its origins on unix computers, it is quite good at Postscript output, but also includes other formats. The devices available to you for plotting will depend to some extent on your operating system (Windows versus unix/linux). R includes postscript, pdf, pictex, and xfig as vector devices, and png and jpeg as raster (pixel) devices. Simply type:

```
?Devices or
help(Devices)
```

to get a list of available devices and their names (note the capital D on Devices). Each of the devices has options that can be set to control plot size, orientation (landscape or portrait), font size, etc.

To get a quick feel for how easy it is to create plots, let's first create a simple data set containing three numeric variables:

x<-1:50 — creates an ordered vector with elements 1 through 50
y<-rnorm(50,0,1) — creates a vector of random numbers; length 50; mean 0; variance 1

Now we can produce a simple scatter plot of x against y using the basic plot () function. Simply type:

plot(x,y) — note, a call to any of the plotting functions will automatically open up a graphics device and display the results in that device.

We can change just about any aspect of the plot with a bewildering array of graphical controls given as arguments to the plot function. Here are some examples for you to try:

plot(x,y,type='o') — to change the type of plot, try type='l','o','b', and 's'
plot(x,y,type='o',lty=2) — to change the line type try lty=1,2,3,4,...
plot(x,y,type='o',col='blue') — to change the line color try col='blue','red','green', ...
plot(x,y,type='o',pch=2) — to change the point symbols try pch=l,2,3,4,...
plot(x,y,type='o',cex=2) — to change the point size try cex=2,3,4,... (#x normal)
plot(x,y,type='o',lwd=2) — to change the line width try lwd=2,3,4,... (#x normal)

To see a complete list of plot controls, look at the help file for the par() function:

help(par)

Most or all of the par commands to control the graphics can be given as arguments to the plot function (as above). However, it is also possible to set these graphics controls for the graphics device being use so that all plots to that device will adopt these same controls. This is done by issue a par() command before a plot command. Some examples are as follows:

par(mfrow=c(2,3)) — partitions the page into 6 sections, 2 rows and 3 columns.
par(new=TRUE) — used to overlay different plot types
par(mai=c(0.6,0.5,0.1,0.5)) — specifies margin size in inches (bottom, left, top, right)
etc.

Of course there are many more options and these can all be specified in a single command, for example:

par(mfrow=c(2,3),new=TRUE,mai=c(0.6,0.5,0.1,0.5))

Once a par command is given, the graphics controls given by the function are set for the current graphics device until changed by a subsequent par() command or by changing the controls within a plot function.

Of course there are many different kinds of plots for displaying data. The basic plot() function is simply a starting point. There are many different so-called "high-level" plotting functions, for example:

plot() — line/point plots; output depends on class of data

hist() — histogram of single variable
boxplot() — box-and-whisker plot of single variable
qqnorm() — quantile-quantile plot of single variable
coplot() — graphs of 3 or more variables
image() — draws grid of rectangles using 3 variables
contour() — draws contours using 3 variables
persp() — draws 3D surface
pairs() — all pairwise plots between multiple variables
etc.

In addition, there are many so-called "low-level" plotting functions used to plot additional elements over an existing plot (i.e., overlays). These low-level functions are always called after a high-level command in order to supplement the high-level plot. Some examples of low-level commands include:

points() — add points to an existing plot
lines() — add lines to an existing plot
text() —add text to an existing plot
abline() — draw a line in intercept and slope form across an existing plot
polygon() — draw a polyon on an existing plot
legend() — add a legend to an existing plot
title() — add a title to an existing plot
axis() — add further axis scales to an existing plot

There is of course much more detail to plotting in R, but this should suffice for now. We will be making extensive use of the plotting capabilities of R throughout this course.

12. Getting Help in R

There are lots of ways to get help in R. Check out the Help drop-down menu options in the R window. The pdf manuals are an invaluable source of information and provide an introduction to R and a basic description of all the basic R functions. In addition, you can access help on particular functions more directly from the console, as follows:

To see help page for a particular function, type:

help(function) or
?function

To search help for keywords, type:

help.search('mahalanobis distance') or
??'mahalanobis distance'

Note, the quotes around the phrase are necessary because it contains spaces. For a single key work search, the quotes are unnecessary.

To search the R-project list serve for information on a particular function or, more importantly, on a topic or keyword, type:

```
RSiteSearch("mahalanobis distance")
```

This will conduct a targeted search of documented R functions, documents and the help list. This is a great way to see what other people have done to solve a particular problem.

Lastly, you can always a search engine like Google, but it is helpful to precede the search query with the letter R to prioritize results pertinent to R, e.g.,:

```
R mahalanobis distance
```

13. Libraries and Packages

While R is an expansive language with a large number of routines already included, it doesn't include everything. Fortunately, the core routines are easily augmented with additional user-written routines which can be loaded into your copy of R. These routines are usually provided in what R calls a 'package,' which is a package with the routine itself (which may be partially implemented in FORTRAN or C, as well as S), help files, often test data, and other items as necessary. Accordingly, it's necessary to know how to load packages to make the most of R.

Under Windows OS, click on the Packages menu and scroll down to the Load package item. This will pop up a widget listing all available packages. To load the library, simply click on the desired library. Alternatively, to load the library include the library name as listed in the library function. For example, enter:

```
library(MASS)
```

to load the MASS library of Venables and Ripley.

To see a list of installed libraries, enter:

```
library()
```

If the library you want is not installed, you will have to install it yourself. Again, depending on operating system and program, the details are somewhat different.

The best repository for R packages is CRAN at http://cran.r-project.org/. R generally refers to 'packages' rather than 'libraries.' The R site has separate areas for source code (S functions and FORTRAN or C code in uncompiled ASCII) and binaries (compiled code for a specific machine). If your machine and operating system are supported, it's usually simpler to use the pre-compiled binaries.

If your machine is on the internet, R has routines available to automatically install or update libraries or packages from CRAN. This is one of the areas where R really outshines S-Plus.

Under Windows OS, click on the Packages menu and scroll down to Install packages from CRAN. This will pop up a widget that lists all the packages available for DOS/R. Simply click on the desired package and it will install. It's wonderful!

14. Script editors

While it is occasionally useful to type code into the R console and view the results, as we have been doing all along in this lab exercise, it is almost always more useful to type code into a text editor. The text editor allows you to efficiently type in code and run it as before, but also allows you to save the script as a file that can be easily updated, rerun and shared with others. The script is an effective way to document your analysis.

There are several text editor options. The simplest text editor is the one provided in R. Simply open a new script or a saved script from the file drop-down menu. For our purposes, we can open a saved script that contains all the R code used in this lab exercise. From the file drop-down menu, select 'open script', navigate to the directory containing the files for this lab exercise, and select the R.intro.R file. Note, the navigation window will by default show only files with an .R or .r extension. This will open the script in the R text editor window.

There are a few things to note about working with scripts.

- First, the pound sign (#) denotes a comment line.
- Second, to submit or run a single line of code from the script, simply place the cursor at any point in the line (doesn't matter where) and enter control R (for Windows OS) on the keyboard. This will send the line to the R console just as if we had typed it in the console. There are alternative ways to send a line to the R console: 1) right click mouse and select 'run line or selection', or 2) from the edit drop-down menu select 'run line or selection'. If we want to send multiple lines to the console, simply block select the lines using the mouse and then enter control R, or right click mouse, or use the drop-down menu. If we want to send the entire script to the console, simple block select everything and then control R, or right click mouse and select 'select all' and then control R, or select the 'run all' option from the edit drop-down menu.
- Third, the R text editor is not visually appealing for writing and editing script because it does not support syntax coloring. Thus, it is generally preferable to use a different text editor and there are lots of options in this regard. One option is to use a third party text editor, such as EditPad Pro, that offers R syntax coloring and can be configured to submit lines to the R console much like the R text editor. This is perhaps a good option if you have text editor that you really like and are used to and it offers an interface with R. Another option is to use an R library such as TinnR that must be installed first just like any other library, but then can be used as a multi-functional text editor for your scripts. Yet another option is to install RStudio™ is a new integrated development environment (IDE) for R. RStudio combines an intuitive user interface with powerful coding tools to help you get the most out of R. Thus, Rstudio is more than just a text editor, but is a way to manage the entire R workspace.

15. R Workspaces

When we open R, we also open a workspace. Any libraries that we load or objects that we create are stored in that workspace, which is really nothing more than named computer memory. When we are

done with a session, it may be convenient to save the workspace so that we pick up right were we left off at some later time. R allows us to save the current workspace and load previously saved workspaces. An R workspace is given a user specified file name with an .RData extension. This is most useful if the session involved lots of processing to create temporary files (i.e., not save to disk) and we don't want to have to repeat that processing the next time we are ready to continue working. So, we simply save the workspace using the file drop-down menu and the next time we are ready to continue working on this particular project, we simply load the saved workspace and pick up right where we left off. If the processing time to get back to the point of interest is very short, we may not need to save the workspace and instead simply rerun the script back to the point of interest.

16. Tutorials for Learning R

A new set of teaching modules for using the R programming environment in ecology and epidemiology is available through the open-access on-line peer-reviewed journal The Plant Health Instructor (PHI).

http://www.apsnet.org/education/AdvancedPlantPath/Topics/RModules/default.html

You might explore these modules on your own to better prepare yourself for your future life in R.