1. Introduction to R

FISH 560: Applied Multivariate Statistics for Ecologists

Topics

1. What is R
2. Installing R
3. Libraries, packages and sources for R
4. Importing and exporting data
5. Variables, Data Structure, Graphing

A quick note on font conventions

All R documents provided in class are typed using Trebuchet MS font. However, when R code is presented, referenced, or when R output is given, I have used 10 point **`Bold Courier New Font`**.

WHAT IS R?

R is a language and environment for statistical computing and graphics, similar to the S language originally developed at Bell Labs. It is an open source solution to data analysis that is supported by a large and active worldwide research community. R is exceptional statistical software for ecological analysis as it includes a broad range of multivariate techniques, as well as numerous routines for exploratory data analysis. It handles and analyzes data very effectively and it contains a suite of operators for data manipulation, simulation and calculation. It also has the graphical capabilities for very sophisticated graphs and data displays. Finally, it is an elegant, object-oriented programming language. Technically, the language is called S, and R is the open source implementation available for many systems for free. The R project was started by Robert Gentleman and Ross Ihaka (that's where the name "R" is derived) in the Statistics Department at the University of Auckland in 1995. The software has quickly gained a wide audience. The R project web page is http://www.r-project.org.

R is a powerful programming environment, but can be intimidating if you know little about programming. There aren't any flashy buttons to press, just an unforgiving, blinking prompt that offers little comfort and no direction. But don't worry; we're in this together. Manuals and tutorials are available at http://cran.r-project.org/other-docs.html.

Luckily for you, most of the multivariate techniques we'll be performing in this class can be performed by R programs and require relatively little understanding of the R programming language. Program authors typically bundle sets of statistical programs into libraries that can be downloaded from the web and called from R. In the following sections we cover some of the basics of R. Learning these fundamentals is essential for FISH 560 because it will help you unleash the power of R for multivariate analysis.

WHY USE R?

R has many features to recommend it:

- Most commercial statistical software platforms cost thousands, if not tens of thousands of dollars. R is free! If you are a teacher or a student, the benefits are obvious
- R runs on a wide variety of platforms including Windows, UNIX and Mac OS X.
- R is a comprehensive statistical platform, offering all manner of data analytic techniques.
- R has state-of-the-art graphics capabilities.
- R provides an unparalleled platform for programming statistical methods in an easy manner.
- R contains advanced statistical routines not yet available in other packages.

OBTAINING AND INSTALLING R

R is freely available from the Comprehensive R Archive Network (CRAN) at http://www.r-project.org. Precompiled binaries are available for Linux, Mac OS X, and Windows. Follow directions for installing the base product on the platform of your choice. Later we'll talk about adding additional functionality through optional modules called packages (also available from CRAN).

LIBRARIES AND PACKAGES FOR MULTIVARIATE ANALYSIS

R comes with extensive capabilities right out of the box. However, some of its most exciting features are available as optional modules that you can download and install. There are thousands of user contributed modules called packages that you can download from the CRAN website. They provide a tremendous range of new capabilities, from the analysis of geostatistical data to multivariate data! We will use many of these optional packages in FSH 560.

What are packages?

Packages are collections of R functions, data, and compiled code in a well-defined format. The directory where packages are stored on your computer is called the library. The function .libPaths() will show you where your library is located, while the function library() will show you what packages you have saved in your library.

Installing a package

There are a number of R functions that let you manipulate packages. To install a package for the first time, use the install.packages() command or click on the *packages* pull down menu at the top of the R program. For example, install.packages() without options will bring up a list of CRAN mirror sites.

Once you select a site, you will be presented with a list of all available packages. Selecting one will download and install it. You only need to install a package once. However, like any software, packages are often updated by their authors. Use the command update.packages() to update any packages that you have installed. To see details on your packages, you can use the installed.packages() command.

Loading a package

Installing a package downloads it from a CRAN mirror site and places it in your library. To actually use it in an R session, you need to load the package using the library() command. For example, to use the packaged vegan, issue the command library(vegan). Of course, you must have installed a package before you can load it. You will have to load a package once in each session you want to use it.

At present, there are a number of libraries or packages available specifically for multivariate analyses typically conducted by ecologists, including VEGAN from Jari Oksanen and LABDSV from Dave Roberts. These libraries are available at CRAN website and can be installed using the instructions above. We will make extensive use of them in subsequent lectures/labs, so be sure to install these packages at the beginning of each lab.

Learning about a package

When you load a package, a new set of functions and datasets become available. Small illustrative datasets are provided along with sample code, allowing you to try out the new functionalities. The help system contains a description of each function (along with examples), and information on each dataset included. Entering help(package="name") will provide a brief description of the package named and an index of the functions and datasets included. Using help()with any of these function or dataset names will provide further details. The same information can be downloaded as a PDF manual from CRAN.

SOURCES FOR MULTIVARIATE ANALYSIS

Biostats.R is a suite of R functions written by Kevin McGarigal (University of Massachusetts) to facilitate the analysis of multivariate data sets. This is a pseudo-library that contains a set of functions that must

be sourced by clicking on the *File* menu at the top, and then clicking on *Source R code* …. Locate the file Biostats.R from your directory. Help documentation has been provided.

Common mistakes in R programming

There are some common mistakes made frequently by both beginning and experienced R programmers. If your program generates an error be sure the check for the following:

Using the wrong case. help(), Help(), and HELP() are three different functions (only the first will work).

Forgetting to use quote marks when they are needed. install.packages("vegan") will work, while install.packages(vegan) will generate an error.

Forgetting to include the parentheses in a function call. help() rather than help. Even if there are no options, you still need the ().

Using the \ in a path name on Windows. R sees the backslash character as an escape character. setwd("c:\mydata") will generate an error. Use setwd("c:/mydata") or setwd("c:\\mydata") instead.

Using a function from a package that is not loaded. The function hclust() is contained in the vegan package. If you try to use it before loading the package, you will get an error.

The error messages in R can be cryptic, but if you are careful to follow the points above, you should avoid seeing many of them.

IMPORTING DATA INTO 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 first thing to do before importing any file is to tell R what directory your file is in. Do this by going under the File menu and choosing the "Change dir" option. Type in or browse to the directory of your data file and press the OK button.

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

```
sample elev aspect slope river
   1 1300   240   30   skagit
   2 1640   170   20   cowlitz
   3 1840    NA   24   quinault
   .  .     .     .    .
   .  .     .     .    .
   .  .     .     .    .
 100 1730    70   15    skagit
```

The columns do not need to be straight, but multi-word variables 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 older versions of R. The above file (if named "site.dat" for instance) could be read with the read.table command as follows:

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

The resulting data frame would be named "site", and the columns would be named exactly as in the data file. Note that the value for aspect in the third sample 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:

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

Preferably, data can be organized as a "csv" comma delimited file, as follows:

```
sample,elev,aspect,slope,river
1,1300,240,30,skagit
2,1640,170,20,cowlitz
3,1840,90,24,quinault
. . . . .
. . . . .
. . . . .
100,1730,70,15,skagit
```

In which case it would be read:

`site <- read.table('site.dat',header=TRUE,sep=",")` to tell R that the values were separated by commas. Alternatively, you can use:

`site <- read.csv('site.dat',header=TRUE)` to read the file, as read.csv() calls read.table() with the appropriate parameters as defaults.

Finally, if the data are in a formatted file with no delimiters (spaces or commas) it can be read by specifying the columns that start each field. For example:

```
1130024030skagit
2164017020cowlitz
31840 9024quinault
  . .   .   .   .
  . .   .   .   .
  . .   .   .   .
1001730 7015skagit
```

can be read

```
site <- read.table('site.dat',sep=c(1,4,8,11,13))
```

In this case (or in any case where column headings are absent), they can be entered separately with the names command. For example:

```
names(site) <- c("sample","elev","aspect","slope","river")
```

Row names (such as sample 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 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), using **`row.names= specifier`**. Going back to the original example,

`site <- read.csv('site.dat',header=TRUE, row.names=1)` makes sure that R knows that the first column is identifiers, not data.

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.

Troubleshooting
Sometimes you may have data in a format R will not understand. In such cases using scan () or the data editor to enter your data may be a simple and easy solution. Another trick is to try importing the data into another program, such as a spreadsheet program, and saving it as a different file type. In particular saving spreadsheet data in a text (comma or tab delineated) format is simple and useful. Caution should be used as some spreadsheet programs may restrict the number of data values to be stored.

EXPORTING DATA FROM R

Exporting results from R is usually a less difficult task. The most common method is to write a matrix or data frame to file as a rectangular grid of numbers, possibly with row and column labels. This can be done by the functions write.table and write. Function write just writes out a matrix or vector in a specified number of columns (and transposes a matrix). Function write.table is more convenient, and writes out a data frame (or an object that can be coerced into a data frame) with row and column labels.

USING OUTPUT AS INPUT – REUSING RESULTS

One of the most useful design features of R is that the output of analyses can easily be saved and used as input to additional analyses. Let's walk through an example. If you don't understand the statistics involved, don't worry. We are focusing on the general principle here.

This following code will run a simple linear regression of miles per gallon (mpg) on car weight (wt) using the dataset mtcars. Results are sent to the screen. Nothing is saved.

```
lm(mpg~wt, data=mtcars)
```

This time, the same regression is performed but the results are saved under the name fit.

```
fit <- lm(mpg~wt, data=mtcars)
```

No output is sent to the screen. However, we now can manipulate the results.

The assignment has actually created a list called "fit" that contains a wide range of information from the analysis (including the predicted values, residuals, regression coefficients, and more). Typing **`summary(fit)`** provides details of the analysis, while **`plot(fit)`** produces diagnostic plots. You can generate and save influence statistics with **`cook<-cooks.distance(fit). plot(cook)`** will graph these influence statistics. To predict miles per gallon from car weight in a new set of data use **`predict(fit, mynewdata).`**

To see what a function returns, look at the Value section of the online help for that function. Here we would look at **`help(lm)`**. This will tell you what is saved when you assign the results of that function to a name.

VARIABLES

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. Names can consist of letters, numbers, and the character ".". They may not start with a number, or include the character "$" 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 that 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 <- 'Homo sapien'
human <- 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 1.0^{-10}. 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 37 to be added to species.name because species.name was a character variable.

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. There are 4 primary data structures we will use repeatedly.

- **Vectors:** vectors are one-dimensional ordered sets composed of a single data type. Data types include integers, real numbers, and strings (character variables)
- **Matrices**: matrices are two dimensional ordered sets composed of a single data type, equivalent to the concept of matrix in linear algebra.
- **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."
- **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, list items do not have a row-column structure, and items need not be the same length; some can be a single value, and others a matrix.

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 character "$" 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.vector <- 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, if the number of species per sample site is stored in a vector **`spc.site`**, then

`spc.site[37]` = the number of species in site 37

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

`veg[23,48]` = row 23, column 48 in matrix **`veg`**

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,

`veg[,3]` = third column of matrix **`veg`** represents the third column of matrix **`veg`**, as the row number before the comma was omitted. Similarly,

`veg[5,]` = row 5 of matrix **`veg`** represents row 5, as the column after the comma was omitted. In addition, a number of specialized subscripts can be used.

`veg[]` = all rows and columns of matrix **`veg`**
`spcsite[a:b] = spcsite[a]` through **`spcsite[b]`**
`spcsite[-a]` = all of vector **`spcsite`** except **`spcsite[a]`**
`veg[a:b,c:d]` = a submatrix of **`veg`** from row a to b and column c to d

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

`spcsite[c(1,7,10),c(3,6,12)]` = a submatrix consisting of rows 1,7 and 10, and columns 3, 6, and 12 from matrix **`spcsite`**.

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 example, in the MAHA fish dataset (Mid-Atlantic Highlands Area containing environmental characteristics of the fish sampling sites), there is a column labeled "Elev" that holds the elevation of each sample site. This column can be accessed as **`maha$Elev`**, where "maha" is the name of the data frame, "Elev" is the name of the field or column of interest, and the "$" 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(maha)
```

we can specify the field "Elev" simply as "Elev" rather than "maha$Elev." This is more concise notation, but means that we cannot have a variable with the same name as a field in a data frame that is attached. Data frames are extraordinarily useful in R.

Lists
As noted above, a list is a compound object composed of associated data. Items within a list 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.

R VECTOR AND MATRIC 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 exist for manipulating vectors, and by extension, matrices. For example, if veg is a plant community matrix of 100 sample sites and 200 species (sites as rows and species as columns), we can perform the following:

`x <- max(veg[,3])` --- assigns the maximum value of species 3 among all sites
`y <- sum(veg[,5])` --- assigns the sum of species 5 abundance in all sites to y
`logveg <- log(veg+1)` --- creates a new matrix called "logveg" with all values the log of the respective values in veg (+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:

q <- sum(veg[,8]>10) --- assigns q the number of sites where the abundance of species 8 is greater than 10 (veg[,8]>0 is evaluated as 1 (true) or 0 (false), so that the sum is of 0's and 1's).

r <- sum(veg[,8][veg[,8]>10]) --- assigns r the sum of the abundance for species 8 in sites where species 8 has abundance greater than 10

deep14 <- max(veg[,14][soil=='deep']) --- assigns the maximum abundance for species 14 on sites with deep soils

A final 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. For now lets assume that we have missing values in a vector. To use all of the vector EXCEPT the missing value, use

```
spc.site[!is.na(spc.site)]
```

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(spc.site)
```

Accordingly, the full expression is

```
spc.site[!is.na(spc.site)]
```

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.

ROW OR COLUMN OPERATIONS ON A MATRIX

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

spcmax <- apply(veg,2,max) creates a vector "spcmax" with the maximum value for each species in its respective position. The apply operator is employed as:

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

so that

```
sitesum <- apply(veg,1,sum)
```

creates a vector of total species abundance in each site. The vector is as long as the number of rows in matrix veg.

TRIANGULAR MATRICES

Often in community ecology we work with symmetric matrices (e.g. similarity, dissimilarity, of distance matrices). These matrices take up extra space (since the value of the diagonal is known by definition, and since every other value is stored twice (matrix[x,y]=matrix[y,x]). We can save space by only storing one triangle of the matrix. In addition, some analyses require a vector argument, rather than a matrix, and it's convenient to convert the triangular matrix to a vector. This can be done as follows:

```
triang <- matrix[row(matrix) > col(matrix)]
```

GRAPHING 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 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).

X11
In unix/linux, we will be mostly working with X11. If you give R a plotting command without first opening a device, an X11 window will pop up automatically to contain the plot. This plotting area is usually a convenient size for working, and can be resized with the mouse to almost any size. Normally, this is convenient and sufficient. Sometimes, however, we want absolute control over the aspect ratio of the plot, so that 100 units on the X axis is exactly the same size as 100 units on the Y axis. There is a small number of ways to ensure that the plotting is "square", but all of them assume that the plotting window has not been re-sized with the mouse. Accordingly, it is sometimes important to know how to create a plotting window of a specific size.

In R, the X11 window is controlled by the x11() function. The size of the window is specified in inches as arguments to the function. For example, to get a window 8 inches wide by 6 inches tall

```
x11(height=6,width=8)
```

This is simple, except that you can't control the location. You can, however, move the window with your mouse. As long as you don't resize it you are fine.

Other Devices
The list of other devices you can plot also depends on operating system. R includes postscript, pdf, pictex, and xfig as vector devices, and png and jpeg as raster (pixel) devices as well as x11. 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.

SOME COMMON PLOTTING FUNCTIONS

(borrowed from *Statistics using R with Biological Examples*, Seefeld and Linder)

Function name	Plot produced
boxplot(x)	"Box and whiskers" plot
pie(x)	Circular pie chart
hist(x)	Histogram of the frequencies of x
barplot(x)	Histogram of the values of x
stripchart(x)	Plots values of x along a line
dotchart(x)	Cleveland dot plot
pairs(x)	For a matrix x, plots all bivariate pairs
plot.ts(x)	Plot of x with respect to time (index values of the vector unless specified)
contour(x,y,z)	Contour plot of vectors x and y, z must be a matrix of dimension rows=x and columns=y
image(x,y,z)	Same as contour plot but uses colors instead of lines
persp(x,y,z)	3-d contour plot

HIGH-LEVEL PLOTTING
(borrowed from *R for Beginners*, Emmanuel Paradis)

`plot(x)`	plot of the values of `x` (on the y-axis) ordered on the x-axis
`plot(x, y)`	bivariate plot of `x` (on the x-axis) and `y` (on the y-axis)
`sunflowerplot(x, y)`	id. but the points with similar coordinates are drawn as a flower which petal number represents the number of points
`pie(x)`	circular pie-chart
`boxplot(x)`	"box-and-whiskers" plot
`stripchart(x)`	plot of the values of `x` on a line (an alternative to `boxplot()` for small sample sizes)
`coplot(x~y | z)`	bivariate plot of `x` and `y` for each value (or interval of values) of `z`
`interaction.plot (f1, f2, y)`	if `f1` and `f2` are factors, plots the means of `y` (on the y-axis) with respect to the values of `f1` (on the x-axis) and of `f2` (different curves); the option `fun` allows to choose the summary statistic of `y` (by default `fun=mean`)
`matplot(x,y)`	bivariate plot of the first column of `x` *vs.* the first one of `y`, the second one of `x` *vs.* the second one of `y`, etc.
`dotchart(x)`	if `x` is a data frame, plots a Cleveland dot plot (stacked plots line-by-line and column-by-column)
`fourfoldplot(x)`	visualizes, with quarters of circles, the association between two dichotomous variables for different populations (`x` must be an array with `dim=c(2, 2, k)`, or a matrix with `dim=c(2, 2)` if $k = 1$)
`assocplot(x)`	Cohen–Friendly graph showing the deviations from independence of rows and columns in a two dimensional contingency table
`mosaicplot(x)`	'mosaic' graph of the residuals from a log-linear regression of a contingency table
`pairs(x)`	if `x` is a matrix or a data frame, draws all possible bivariate plots between the columns of `x`
`plot.ts(x)`	if `x` is an object of class `"ts"`, plot of `x` with respect to time, `x` may be multivariate but the series must have the same frequency and dates
`ts.plot(x)`	id. but if `x` is multivariate the series may have different dates and must have the same frequency
`hist(x)`	histogram of the frequencies of `x`
`barplot(x)`	histogram of the values of `x`
`qqnorm(x)`	quantiles of `x` with respect to the values expected under a normal law
`qqplot(x, y)`	quantiles of `y` with respect to the quantiles of `x`
`contour(x, y, z)`	contour plot (data are interpolated to draw the curves), `x` and `y` must be vectors and `z` must be a matrix so that `dim(z)=c(length(x), length(y))` (`x` and `y` may be omitted)
`filled.contour (x, y, z)`	id. but the areas between the contours are coloured, and a legend of the colours is drawn as well
`image(x, y, z)`	id. but the actual data are represented with colours
`persp(x, y, z)`	id. but in perspective
`stars(x)`	if `x` is a matrix or a data frame, draws a graph with segments or a star where each row of `x` is represented by a star and the columns are the lengths of the segments

LOW-LEVEL PLOTTING

R has a set of graphical functions that affect an already existing graph: they are called low-level plotting commands. Here are the main ones (borrowed from *R for Beginners*, Emmanuel Paradis):

`points(x, y)`	adds points (the option `type=` can be used)
`lines(x, y)`	id. but with lines
`text(x, y, labels, ...)`	adds text given by `labels` at coordinates (`x`,`y`); a typical use is: `plot(x, y, type="n"); text(x, y, names)`
`mtext(text, side=3, line=0, ...)`	adds text given by `text` in the margin specified by `side` (see `axis()` below); `line` specifies the line from the plotting area
`segments(x0, y0, x1, y1)`	draws lines from points (`x0`,`y0`) to points (`x1`,`y1`)
`arrows(x0, y0, x1, y1, angle= 30, code=2)`	id. with arrows at points (`x0`,`y0`) if `code=2`, at points (`x1`,`y1`) if `code=1`, or both if `code=3`; `angle` controls the angle from the shaft of the arrow to the edge of the arrow head
`abline(a,b)`	draws a line of slope `b` and intercept `a`
`abline(h=y)`	draws a horizontal line at ordinate `y`
`abline(v=x)`	draws a vertical line at abcissa `x`
`abline(lm.obj)`	draws the regression line given by `lm.obj` (see section 5)
`rect(x1, y1, x2, y2)`	draws a rectangle which left, right, bottom, and top limits are `x1`, `x2`, `y1`, and `y2`, respectively
`polygon(x, y)`	draws a polygon linking the points with coordinates given by `x` and `y`
`legend(x, y, legend)`	adds the legend at the point (`x`,`y`) with the symbols given by `legend`
`title()`	adds a title and optionally a sub-title
`axis(side, vect)`	adds an axis at the bottom (`side=1`), on the left (2), at the top (3), or on the right (4); `vect` (optional) gives the abcissa (or ordinates) where tick-marks are drawn
`box()`	adds a box around the current plot
`rug(x)`	draws the data `x` on the x-axis as small vertical lines
`locator(n, type="n", ...)`	returns the coordinates (x, y) after the user has clicked `n` times on the plot with the mouse; also draws symbols (`type="p"`) or lines (`type="l"`) with respect to optional graphic parameters (`...`); by default nothing is drawn (`type="n"`)

GRAPHICAL PARAMETERS

In addition to low-level plotting commands, the presentation of graphics can be improved with graphical parameters. They can be used either as options of graphic functions (but it does not work for all), or with the function **`par`** to change permanently the graphical parameters, i.e. the subsequent plots will be drawn with respect to the parameters specified by the user. For instance, the following command:

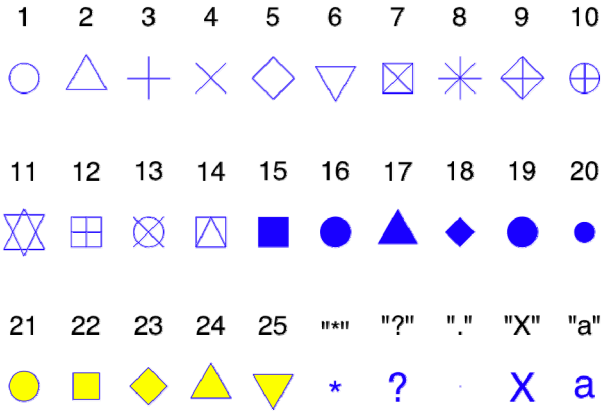
```
par(bg="yellow")
```

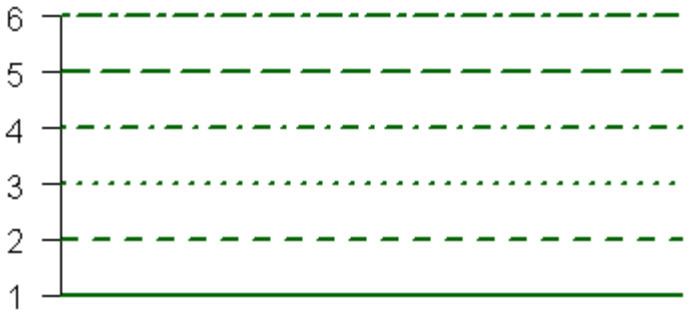
will result in all subsequent plots drawn with a yellow background. There are 73 graphical parameters, some of them have very similar functions. The exhaustive list of these parameters can be read with **`?par`**; I will limit the following table to the most usual ones (borrowed from *R for Beginners*, Emmanuel Paradis).

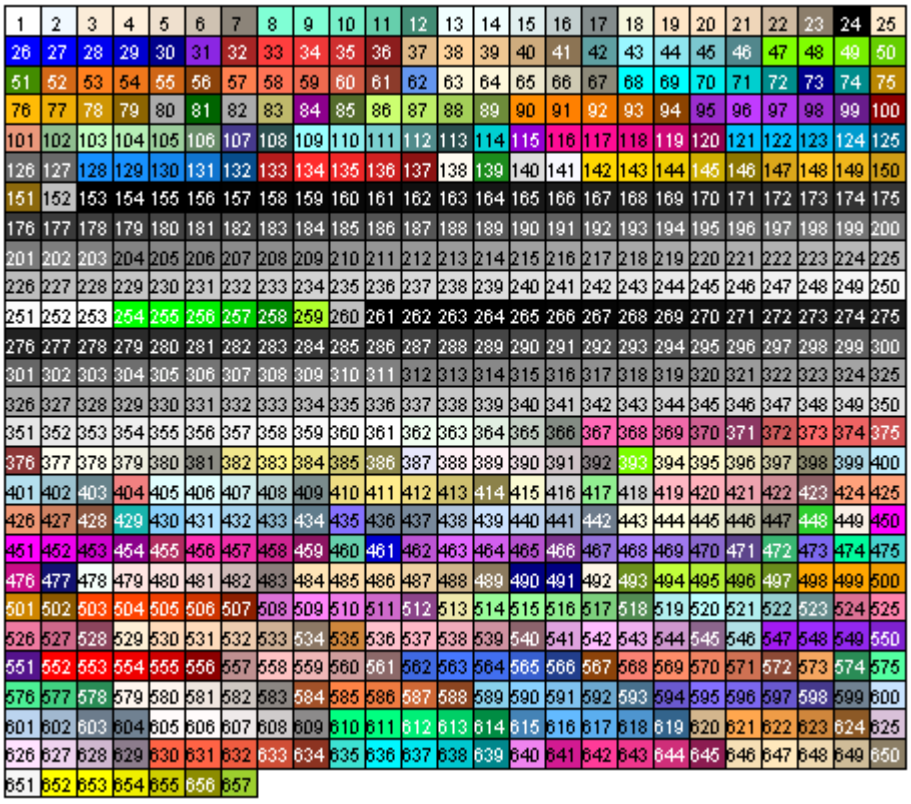
adj	controls text justification with respect to the left border of the text so that 0 is left-justified, 0.5 is centred, 1 is right-justified, values > 1 move the text further to the left, and negative values further to the right; if two values are given (e.g., c(0, 0)) the second one controls vertical justification with respect to the text baseline
bg	specifies the colour of the background (e.g., bg="red", bg="blue"; the list of the 657 available colours is displayed with colors())
bty	controls the type of box drawn around the plot, allowed values are: "o", "l", "7", "c", "u" ou "]" (the box looks like the corresponding character); if bty="n" the box is not drawn
cex	a value controlling the size of texts and symbols with respect to the default; the following parameters have the same control for numbers on the axes, cex.axis, the axis labels, cex.lab, the title, cex.main, and the sub-title, cex.sub
col	controls the colour of symbols; as for cex there are: col.axis, col.lab, col.main, col.sub
font	an integer which controls the style of text (1: normal, 2: italics, 3: bold, 4: bold italics); as for cex there are: font.axis, font.lab, font.main, font.sub
las	an integer which controls the orientation of the axis labels (0: parallel to the axes, 1: horizontal, 2: perpendicular to the axes, 3: vertical)
lty	controls the type of lines, can be an integer (1: solid, 2: dashed, 3: dotted, 4: dotdash, 5: longdash, 6: twodash), or a string of up to eight characters (between "0" and "9") which specifies alternatively the length, in points or pixels, of the drawn elements and the blanks, for example lty="44" will have the same effet than lty=2
lwd	a numeric which controls the width of lines
mar	a vector of 4 numeric values which control the space between the axes and the border of the graph of the form c(bottom, left, top, right), the default values are c(5.1, 4.1, 4.1, 2.1)
mfcol	a vector of the form c(nr,nc) which partitions the graphic window as a matrix of nr lines and nc columns, the plots are then drawn in columns (see section 4.1.2)
mfrow	id. but the plots are then drawn in line (see section 4.1.2)
pch	controls the type of symbol, either an integer between 1 and 25, or any single character within "" (Fig. 2)
ps	an integer which controls the size in points of texts and symbols
pty	a character which specifies the type of the plotting region, "s": square, "m": maximal
tck	a value which specifies the length of tick-marks on the axes as a fraction of the smallest of the width or height of the plot; if tck=1 a grid is drawn
tcl	id. but as a fraction of the height of a line of text (by default tcl=-0.5)
xaxt	if xaxt="n" the x-axis is set but not drawn (useful in conjunction with axis(side=1, ...))
yaxt	if yaxt="n" the y-axis is set but not drawn (useful in conjunction with axis(side=2, ...))

PLOTTING SYMBOLS, LINES AND COLORS

Below are the plotting symbols in R (pch=1:25). The colors were obtained with the options **col="blue"**, **bg="yellow"**, the second option has an effect only for the symbols 21–25. Any character can be used (pch="*", "?", ".", . . .). Borrowed from *R for Beginners* by Emmanuel Paradis.


1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 "*" "?" "." "X" "a"


6
5
4
3
2
1



SAVING GRAPHICS

Notice that when the graphics window is active the main menu is different. On the File menu there are many options for saving a graphic, including different graphical formats (png, bmp, jpg) and other formats (metafile, postscript, pdf). You could also use command line functionality to save, but using the options under the File menu is easier and pops up a save as dialog box allowing you to choose the directory you are saving the graphic file to.

Another option to save a graphic is to simply right mouse click on the graphic, which will produce a pop up menu with options to copy or save the graphic in various formats as well as to directly print the graphic. In a Windows environment the copy options are as a bitmap or metafile, and the save options are as metafile or postscript.

SUMMARY

In this document, we have looked as some of the strengths that make R an attractive option for students, researchers, statistician, and data analysts trying to understand the meaning of their data. We have walked through the program's installation and talked about how to enhance R's capabilities by downloading additional packages. We have explored the basic interface, running programs interactively and in batch, and produced a few sample graphs. We have also learned how to save our work to both text and graphic files. Hopefully, you're getting a sense of how powerful this freely available software can be. Now that we have R up and running, it's time to get our data into the mix.

ADDITIONAL RESOURCES

CRAN	http://www.r-project.org/
Quick-R	http://www.statmethods.net/index.html
Graphing in R	http://www.statmethods.net/graphs/creating.html

GETTING HELP

Virtually everything in R has some type of accessible help documentation. The challenge is finding the documentation that answers your question. This section gives some suggestions for where to look for help.

Program Help Files: The quickest and most accessible source of help when using R is to use the on-line help system that is part of R. This includes on-line documentation for the R base packages, as well as on-line documentation for any loaded packages. Finding help when you know the name of what it is you're asking for help on is easy, just use the help function with the topic of interest as the argument of the help function. For example to get help on function sum:

```
help(sum)
```

As an alternative to calling the help function you can just put a question mark in front of the topic of interest:

```
?sum
```

Note that on-line help is especially useful for describing function parameters, which this document will not discuss in great detail because many functions in R have lots of optional parameters (read.table for example has about 12 different optional parameters). You should become comfortable looking up functions using help to get detailed parameter information for use in R.