CSI606
Introduction to R

Jeff Solka

Additional References

- *Modern Applied Statistics* with S, B. Ripley and W. Veneables
- Introductory Statistics with R, Peter Dalgaard.
- S Programming, W. Veneables and B. Ripley.
- *A Handbook of Statistical Analysis using S-Plus*, B. Everitt

History of R and Its Capabilities

R, S and S-plus

S: an interactive environment for data analysis developed at Bell Laboratories since 1976
1988 - S2: RA Becker, JM Chambers, A Wilks
1992 - S3: JM Chambers, TJ Hastie
1998 - S4: JM Chambers

Exclusively licensed by *AT&T/Lucent* to *Insightful Corporation*, Seattle WA. Product name: "S-plus".

Implementation languages C, Fortran.

See:
http://cm.bell-labs.com/cm/ms/departments/sia/S/history.html

R, S and S-plus

R: initially written by Ross Ihaka and Robert Gentleman at Dep. of Statistics of U of Auckland, New Zealand during 1990s.

Since 1997: international "R-core" team of ca. 15 people with access to common CVS archive.

GNU General Public License (GPL)
- can be used by anyone for any purpose
- contagious

Open Source
-quality control!
-efficient bug tracking and fixing system supported by the user community

What R Does and Does not Do

- data handling and storage: numeric, textual
- matrix algebra
- hash tables and regular expressions
- high-level data analytic and statistical functions
- classes ("OO")
- graphics
- programming language: loops, branching, subroutines

- is not a database, but connects to DBMSs
- has no graphical user interfaces, but connects to Java, Tcl/Tk
- language interpreter can be very slow, but allows to call own C/C++ code
- no spreadsheet view of data, but connects to Excel/MsOffice
- no professional / commercial support

R and Statistics

- Packaging: a crucial infrastructure to efficiently produce, load and keep consistent software libraries from (many) different sources / authors

- Statistics: most packages deal with statistics and data analysis

- State of the art: many statistical researchers provide their methods as R packages

Obtaining R

- Go to http://www.r-project.org/
- Under Linux
  - Install R as an rpm
- Under Windoz
  - Self extracting binary installation

R Syntax Basics

Making it Go

- **Under Unix/LINUX Type**

```
R (or the appropriate path on
  your machine)
```

- **Under Windows**

```
Double click on the R icon
```

Making it Stop

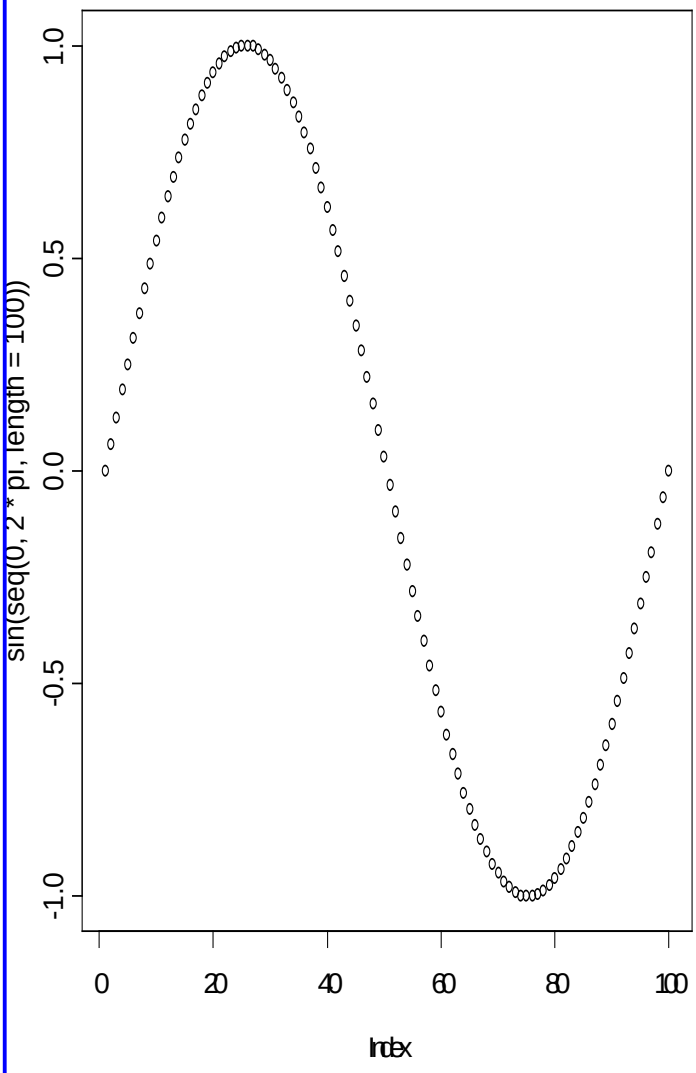
- Type

```
> q()
```

- q()is a function execution
  - Everything in R is a function
  - q merely returns a listing of the function

R as a Calculator

```
> log2(32)

[1] 5

> sqrt(2)

[1] 1.414214


> seq(0, 5,
length=6)

[1] 0 1 2 3 4 5


> plot(sin(seq(0,
 2*pi,
length=100)))
```



Syntax

- Everything that we type in R is an expression

- We may have multiple expressions on each line separated by ;

```
2+3;4*5;6-9
```

- We use <- or = for making assignments

```
b<-5+9 or b = 5+9
```

- R commands are case sensitive

- The result of any expression is an object

Recalling Previous Commands

- **In WINDOWS/UNIX one may use the** `arrow up key` **or the** `history` **command under the menus**

- **Given the history window then one can copy certain commands or else past them into the console window**

Getting Help

- **In both environments we may use**

```
help(command name)
?command name
```

```
> help("ls")
> ? ls
```

- **We may also use**

```
?methods(command name)
```

- **html-based help**

```
help.start()
```

- **For commands with multiple methods based on different object types**

Getting Function Information

- To view information on just the arguments to a function use the command args

```
> args(plot.default)
function (x, y = NULL, type = "p",
   xlim = NULL, ylim = NULL,
     log = "", main = NULL, sub =
   NULL, xlab = NULL, ylab = NULL,
   ann = par("ann"), axes = TRUE,
   frame.plot = axes, panel.first =
   NULL,
     panel.last = NULL, col =
   par("col"), bg = NA, pch =
   par("pch"),
     cex = 1, lty = par("lty"), lab =
   par("lab"), lwd = par("lwd"),
     asp = NA, ...)
NULL
```

Assignments in R

- Some Examples

```
> cat<-45


> dog=66

> cat
[1] 45

> dog
[1] 66

> 77 -> rat
> rat
[1] 77
```

- **Note** = is used for specifying values in function calls

Vectors

- A vector example

```
> a<-c(1,2,3,4)

> length(a)
[1] 4

> a
[1] 1 2 3 4
```

- An example with character strings

```
> name<-c("Jeff","Solka")

> name
[1] "Jeff"  "Solka“

> name[1]
[1] "Jeff"
```

Matrices

- **A matrix example**

```
> b<-matrix(nrow=2,ncol=2)

> b
     [,1] [,2]
[1,]   NA   NA
[2,]   NA   NA

> b[,1]<-c(1,3)
> b[,2]<-c(2,4)

> b
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

Functions

- We will discuss function at length later but for now I point out how to edit a function

```
fix(ftn name) for new functions

edit(ftn name) for existing ones
```

- I have had problems with these under windoz
- It is possible to use other editors (notepad, jot, vi ...)
- Under windoz one can edit with notepad and then save
  - You should save with a .R extension

Editing Data Sets

- We may create and modify data sets on the command line

```
> xx<-seq(from=1,to=5)

> xx
[1] 1 2 3 4 5

> xx[xx>3]
[1] 4 5
```

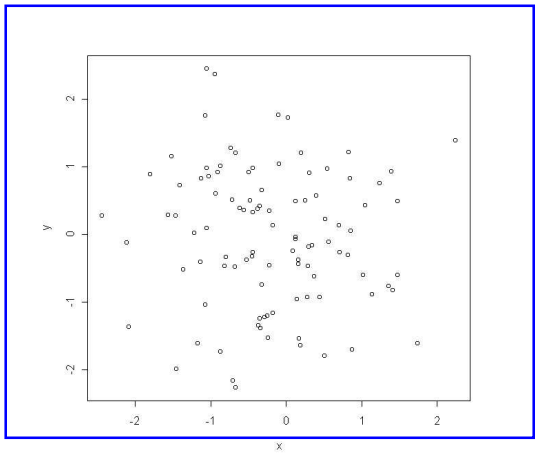
- We may edit our data set in our editor once it is created

```
edit(mydata)
```

Graphics in R

- `win.graph()` **or in UNIX** we say `x11()`
- `dev.list()` - **list currently opened graphics devices**
- `dev.cur()` - **list identifier for the current graphics device**
- `dev.close()` - **close the current graphics window**
- ***A simple plotting example***

```
> x<-rnorm(100)

> y<-rnorm(100)

> plot(x,y)
```



R Search Path

```
> search()
[1] ".GlobalEnv"    "package:ctest"
   "Autoloads"     "package:base"
```

- Organizing your projects under windoz
  - Create a separate shortcut for each project: see Q2.3. All the paths to files used by R are relative to the starting directory, so setting the `Start in' field automatically helps separate projects.
  - Alternatively, start R by double-clicking on a saved .RData file in the directory for the project you want to use, or drag-and-drop a file with extension .RData onto an R shortcut. In either case, the working directory will be set to that containing the file.
  - Alternatively, start R and then use file → change dir to change to your directory of intest
- Organizing your projects under UNIX
  - A separate .Rdata file is used in each directory

Assessing Stored Objects

```
objects()

> objects(pattern="coal*")
[1] "coal.krige"   "coal.mat"
   "coal.mp"
[4] "coal.nl1"     "coal.predict"
   "coal.signal"
[7] "coal.var1"    "coalsig.mat"
```

Removing Stored Objects

```
rm(x, y)
```

```
rm(list=ls(pat = “^x+"))
```

- Removes those objects starting with x

- See http://www.greenend.org.uk/rjk/2002/06/regexp.html for a summary of regular expression rules

- See http://www.anybrowser.org/bbedit/grep.shtml for a brief tutorial on grep

Data Modes

- `logical` - **Binary data mode, with values represented as T or F.**
- `numeric` - **Numeric data mode includes integer, single precision, and double precision representations of numeric values.**
- `complex` - **Complex numeric values (real and imaginary parts).**
- `character` - **Character values represented as strings.**

Data Types

- `vector` - **A set of elements in a specified order.**
- `matrix` - **A matrix is a two-dimensional array of elements of the same mode.**
- `factor` - **A factor is a vector of categorical data.**
- `data frame` - **A data frame is a two-dimensional array whose columns may represent data of different modes.**
- `list` - **A list is a set of components that can be any other object type.**

Vector Creation Functions.

- scan - **Read vaues of any mode.**

```
scan(), scan("mydata")
```

- c - **Combine values of any mode.**

```
c(1,2,3)
```

- rep - **Repeat values of any mode.**

```
rep(1,5)
```

- :, seq - **Generate numeric sequences.**

```
> seq(from=1,by=2,to=10)
[1] 1 3 5 7 9
> 1:4
[1] 1 2 3 4
```

- vector, logical, numeric, complex, character - **Initialize appropriate types.**

```
vector('numeric',4),
logical(3), numeric(5)
```

Matrix Creation Functions.

- matrix - **Create matrix of values.**

```
matrix(1:6,ncol=3,byrow=T)
[,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

- cbind - **Bind together as columns.**

```
c(1,2,3)
cbind(1:10,rep(c(1,2),c(5,5)))
```

- rbind - **Bind together as rows.**

```
rbind(sample(1:10,rep=T),rnorm(10))
```

- data.matrix - **Covert data frame to matrix.**

Data Frames

- `read.table` - **Reads in data from an external file.**

- `data.frame` - **Binds together R objects of various kinds.**

Lists

- The components of a list can be objects of any mode and type including other lists.

- Lists are useful for returning values from functions.

```
> x = 5
> z = list(original=x, square=x^2)
> z$original
[1] 5
> z$square
[1] 25
> attributes(z)
$names
[1] "original" "square"
```

scan Function

- **This is very useful for reading in vectors or matrices.**

```
mat <-
   matrix(scan("mydata"),ncol=4,byrow
   =T)
```

`read.table` Function

- Reads an ascii file and creates a data frame.
- Intended for data in tables of rows and columns.
- If first line in the file contains column labels and the first columns contain row labels then `read.table` will convert to a a data frame naturally.
  - Use `header=T`
- Field separator is white space.
  - There is also `read.csv` and `read.csv2` which assumes , and ; separations
- Treats characters as factors.

www.omegahat.org

- This site implements various R/S interfaces
- Database (Mysql)
- Perl
- Java
- Python
- Glade

data.dump and data.restore

- dump
  - **Used for R Functions**
  - **Mostly Readable by Wetware**
  - **Sourced into another R session**

- save **and** load
  - **Used for R Functions and Objects**
  - **Understandable to load only**

```
> x = 23
> y = 44
>  save(x, y, file = "xy.Rdata")
> load("xy.Rdata")
> ls()
[1] "last.warning" "x"
  "y"
```

Arithmetic Operators

- * - Multiply
- + - Add
- - - Subtract
- / - Divide
- ^ - Exponentiation
- %% - Modulus
- %/% - Integer Divide
- %*% - Matrix Multiply

N.B. - These are all vectorized.

Comparison Operators

- != - Not Equal To
- < - Less Than
- <= - Less Than or Equal to
- == - Equal
- > - Greater Than
- >= - Greater Than or Equal to

Logical Operators

- ! - Not
- | - Or (For Calculating Vectors and Arrays of Logicals)
- || - Sequential or (for Evaluating Conditionals)
- & - And (For Calculating Vectors and Arrays of Logicals)
- && - Sequential And (For Evaluating Conditionals)

Mathematical Functions

- abs - **Absolute Value**
- acos, asin, atan- **Inverse Trig.**
- **acosh, asinh, atanh- Inverse Hyper. Trig.**
- ceiling- **Next Larger Integer**
- floor- **Next Smallest Int.**
- cos, sin, tan - **Trig. Functions**
- exp - **e^x**
- log - **Natural Logarithm**
- log10- **Log Base 10.**
- max- **Maximum**
- min- **Minimum**
- sqrt- **Square Root**

Statistical Summary Functions

- `all`- **Logical Product**
- `any`- **Logical Sum**
- `length`- **Length of Object**
- `max`- **Maximum Value**
- `mean`- **Arithemetic Mean**
- `median`- **Median**
- `min`- **Minimum Value**
- `prod`- **Product of Values**
- `quantile`- **Empirical Quantiles**
- `sum`- **Sum**
- `var`- **Variance**
- `cor`- **Correlation Between Matrices or Vectors**

Sorting and Other Functions

- `rev`- **Put Values of Vectors in Reverse Order**
- `sort`- **Sort Values of Vector**
- `order`- **Permutation of Elements to Produce Sorted Order**
- `rank`- **Ranks of Values in Vector**
- `match`- **Detect Occurences in a Vector**
- `cumsum`- **Cummulative Sums of Values in Vector**
- `cumprod`- **Cumulative Products**

Writing Free-format Files

- `write`
  - Allows one to specify the number of columns
  - Don't forget to use `t` = transpose function and specify number of columns consistent with your original data (default is to write column by column)

- `cat`
  - Less useful than write

- `write.table`
  - Data exporting utilities under the windows file structure

- `dump`
  - Preferable method

Iteration and Flow of Control

- **Conditional Statements**

```
if (cond) {body}
```

- **for and while loops allowed (**but to be avoided if possible**)**

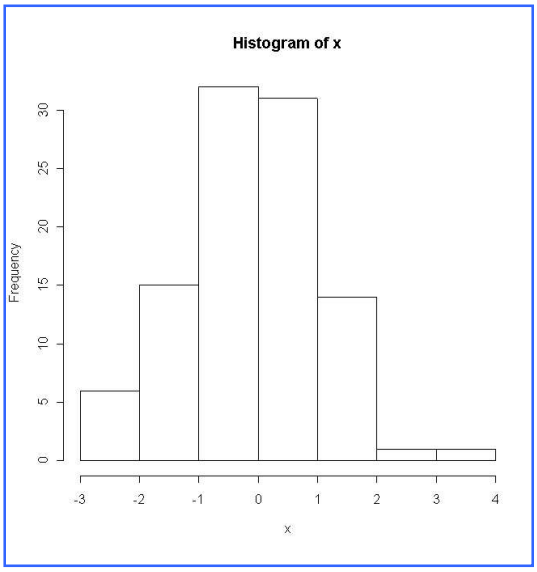
```
for(name in vlaues) {body}
```

R Graphics

High-Level Graphics Functions

- `win.graph(), x11()`
- **All Examples of Calls to Launch Graphics Window**
- *A simple example*

```
> x = rnorm(100)
> win.graph()
> hist(x)
```



Plotting Functions That are Useful for One-Dimensional Data

- `barplot`- **Creates a Bar Plot**
- `boxplot`- **Creates Side-by-Side Boxplots**
- `hist`- **Creates a Histogram**
- `dotchart`- **Creates a Dot Chart**
- `pie`- **Creates a Pie Chart**
- **Note - These commands along with the commands on the next several slides are all high-level graphics calls.**

Plotting Functions That are Useful for Two-Dimensional Data

- `plot` - **Creates a scatter plot**

- `qqnorm` - **Plot quantile-quantile plot for one sample against standard normal**

- `qqplot` - **Plot quantile-quantile plot for two samples**

Three-Dimensional Plotting Function

- `contour`- Creates a contour plot
- `persp`- Creates a perspective or mesh plot
- `image`- Creates an image plot

Apply and Outer

- To perform calculations on each row or column of a matrix use `apply`

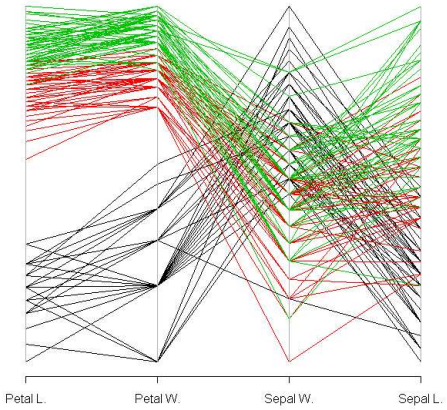
```
apply(mymatrix,2,means)
# Computes column means or mymatrix
```

- To perform the outer product of of two vectors (or matrices)
  - Useful for computing a function over a grid of values

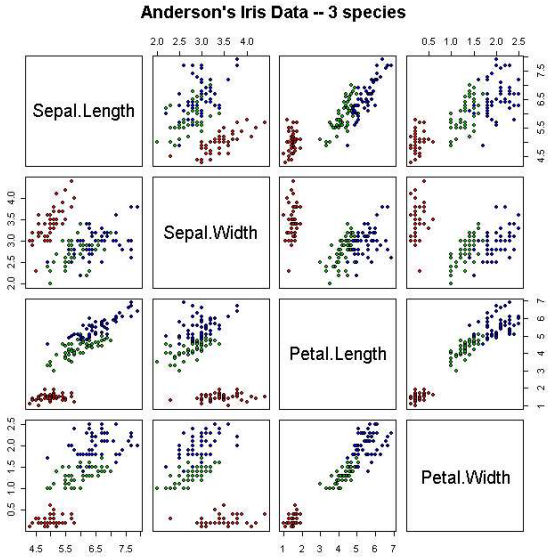
```
surf <- function(x,y) {cos(x) +
  sin(y)}
x<-seq(-2*pi, 2*pi,len=40)
y<- x
z<-outer(x,y,surf)
persp(x,y,z)
```

Multivariate Plotting Function

- `parcoord`- **Plots a parallel coordinates plot of multi-dimensional data (requires `library(MASS)`)**

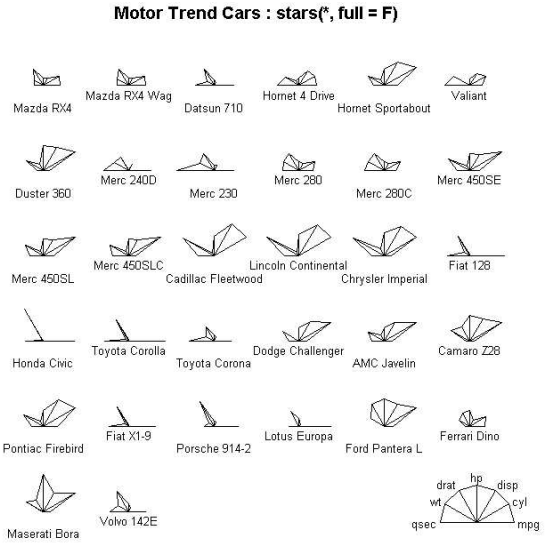


- `pairs`- **Creates a pairs or scatter plot matrix**

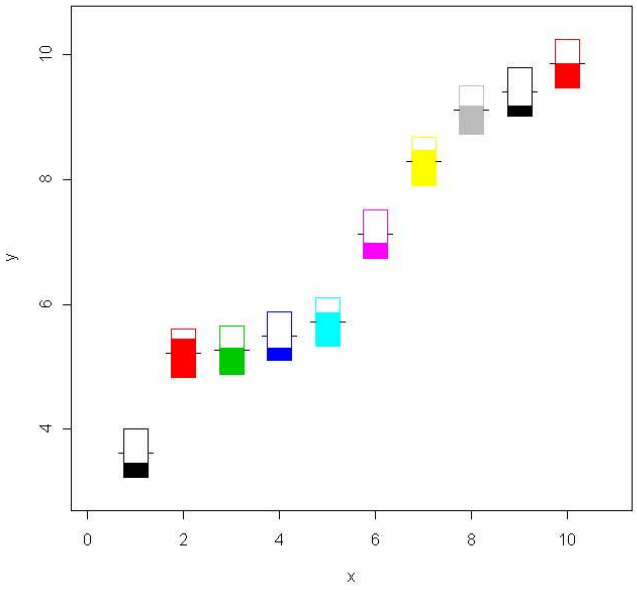


Multivariate Plotting Function

- `stars-` **Starplots**

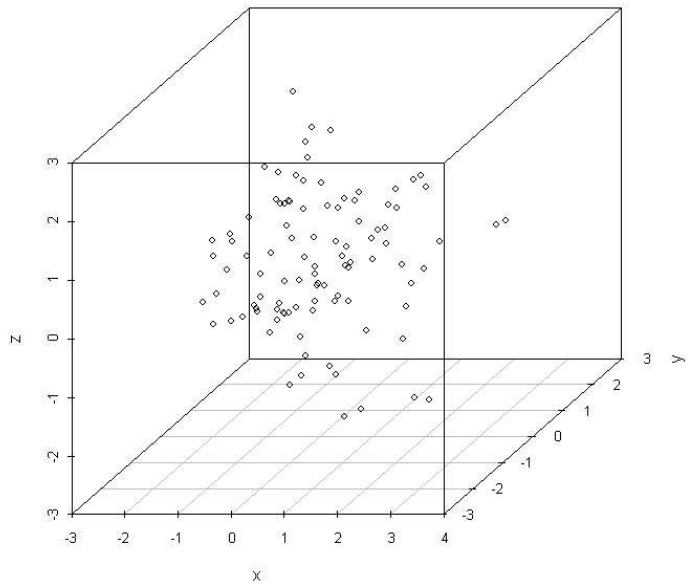


- `symbols` - **Plot symbols at each location.**



Scatterplotting Three-Dimensional Data

```
install.packages("scatterplot3d")
library(scatterplot3d)
> x = rnorm(100)
> y = rnorm(100)
> z = rnorm(100)
> scatterplot3d(x,y,z)
```



The par function

- par
  - Returns current setting on the graphics parameters

- To save the current graphics settings

```
oldsettings<-par()
```

- 4 categories of graphics parameters
  - High-level graphics parameters
    - Control appearance of the plot region
    - Only used as arguments to high-level plotting functions

Graphics Parameter Categories

- High-level graphics parameters
  - Control appearance of the plot region
  - Only used as arguments to high-level plotting functions

- Layout graphics parameters
  - Control the page layout
  - Only set with the par function

- General graphics parameters
  - Set with either call to par or to plotting function
  - When set with par they are set for the current graphics device

- Information graphics parameters
  - Can't bet set by user, but can be queried by par

Multiple Plots Per Page

- `par(mfrow=c(2,2))`
  - **This specifies two rows and two columns of plots**
- `par(mfrow=c(1,1))`
  - **Back to the normal arrangement**
- `plot(x,y,pch="+")`
  - **Override the default plotting symbol**

Adding to Plots

- You can continue to add to plots until you call another high-level plotting function or `frame()`

- We may use love level plot functions to add things to plots
  - `lines`
  - `points`

- **Here is a useful trick**

```
plot(x,y,xlim =
   c(minx,maxx),ylim=c(minx,maxx),type
   =“n”)
```

Printing Graphics

- File-Print Menu
  - Starting Printing Graphics Device
    - Postscript - Postscript
    - Pdf
    - Pictex - Latex
    - Windows - Metafile
    - png - PNG bitmap device
    - Jpeg - JPEG bitmap device
    - Bmp - BMP bitmap device
    - Xfig - Device for XFIG graphics file format

Capturing Graphics to a jpeg File

```
jpeg(file="junk.jpg")

plot(x,y,pch="*")

dev.off()
```

Alternative Screen Printing Approach

#plot in an x11 or wingraph window and then write the output to a file

```
>  dev.print(bmp,
  file="myplot.bmp",
  width=1024, height=768)
```

Functions in R

The Syntax of an R Function

- **R functions are defined using the reserved word function. Following that must come the argument list contained in round brackets, (). The argument list can be empty. These arguments are called the formal arguments to the function.**

- **Then comes the body. The body of the function can be any R expression (and is generally a compound expression).**

- **When the function is called or evaluated the user supplies actual values for the formal arguments which are used to evaluate the body of the function.**

- **All R functions take arguments (the number could be zero, though) and return a value. The value can be returned either by an explicit call to the function return or it can be the value of the last statement in the function.**

A Simple R Function

function() 1

- **This function has no arguments**
- **This function just returns the value 1**
- **This function is not so useful because we did not save it**

A Simple R Function Revisited

simplefun <- function() 1

- **This defines our function**

simplefun()

- **This of course merely returns a 1**

simplefun(1)

- **This does not work because we are offering up an unused argument**

simplefun

- **This of course merely returns the function definition**

Some Slightly More Nontrivial Functions

sf2 <- function(x) x^2

sf2(3)

- **What do you think that this returns?**

sf3 <- function(x) if(x<3) return(x^2) else 4

- **What are the formal arguments to this function?**

> sf3(2)
[1] 4
> sf3(4)
[1] 4
> sf3(-1)
[1] 1

Argument matching in R

- **Argument matching is done in a few different ways.**
  - **One is positional, the arguments are matched by their positions.**
    - **The first supplied argument is matched to the first formal argument and so on.**

- **A second method is by name.**
  - **A named argument is matched to the formal argument with the same name.**
  - **Name matching takes precedence over positional matching.**

- **The specific rules for argument matching are a bit complicated but generally name matching happens first, then positional matching is used for any unmatched arguments.**

- **For name matching a type of partial matching is used { this makes it easy to use long names for the formal arguments when writing a function but does not force the user to type them in}.**

The … Operator

- **There is a special argument named ....**
  - **This argument matches all unmatched arguments and hence it is basically a list.**
  - **It provides a means of writing functions that take a variable number of arguments.**

```
mypower <- function(x, power) x^power
mypower(1, 2)
mypower(p=4, 5) ##5^4 not 4^5
```

Default Arguments

- **The formal arguments can have default values specified for them.**

mypower <- function(x, power=2) x^power

mypower(4)

- **Now, if only one argument is specified then it is x and power has the**

default value of 2.

Partial Argument Matching

- **Partial argument matching requires that you specify enough of the name to uniquely identify the argument.**

```
foo <- function(aa=1, ab=2) aa+ab
foo(a=1, 2)
```

Argument Passing in R

- **R is among a class of languages roughly referred to as having pass by value semantics.**

- **That means that the arguments to a function are copied and the function works on copies rather than on the original values. Because R is a very flexible language this can (like just about everything else) be circumvented.**

- **<u>It is a very bad idea to do so.</u>**

An Interesting Example

```
x<-1:10
foo <- function(x) x[x<5]<-1
foo(x)
x
```

- Notice that x is unchanged.
- Notice also that the expression foo(x) did not seem to return a value.

```
y <- foo(x)
Y
```

- Now, we see that it did, it returned the value 1.
- This is probably not what we intended. What does a function return?
- What is the value of the statement

x[x<5]<-1?

Recursion in R

- Here are two functions that compute the sum of a set of vectors

```
sum1 <- function(x) {
    lenx <- length(x)
    sumx <- 0
    for(i in 1:lenx)
    sumx <- sumx + x[i]
    sumx
}
```

```
sum2 <- function(x) {
    if(length(x) == 1) return(x)
    x[1] + sum2(x[-1])
}
```

Documenting Your Functions

- **The basic object to work with is a package.**
- **Packages are simply a collection of folders that are organized according to some conventions.**
- **A package has a DESCRIPTION file that explains what is in the package.**
- **It will also have two folders.**
  - **One named R that contains your R code**
  - **One named man that contains the documentation for the functions.**

The R Documentation Language

- **R documentation is written in a LATEX like syntax called Rd.**

- **You don't need to know very much about it since you can use the R function prompt to create the documentation and then simply edit it.**

Warnings and Error Messages in R

- **The R system has two main ways of reporting a problem in executing a function.**

- **One is a warning while the other is a simple error.**

- **The main difference between the two is that warnings do not halt execution of the function.**

- **The purpose of the warning is to tell the user that something unusual happened during the execution of this function, but the function was nevertheless able to execute to completion."**

- **One example of getting a warning is when you take the log of a negative number:**

> log(-1)

[1] NaN

Warning message:

NaNs produced in: log(x)

Error Messages in R

```
message <- function(x) {
    if(x > 0)
        print(``Hello'')
    else
        print(``Goodbye'')
}
> x <- log(-1)
Warning message:
NaNs produced in: log(x)
> message(x)
Error in if (x > 0) { : missing value where logical
    needed

> x <- 4
> message(x)
[1] "Hello"
```

Printing the Call Stack With traceback

- The call stack is the sequence of function calls that leads to an error

```
> message(log(-1))
Error in if (x > 0) { : missing value where logical
    needed
In addition: Warning message:
NaNs produced in: log(x)
> traceback()
1: message(log(-1))
```

- Here, traceback shows in which function the error occurred. However, since only one function was in fact called, this information is not very useful. It's clear that the error occurred in the message function. Now, consider the following function definitions:

A More Complex Callback Sequence

```
f <- function(x) {
    r <- x - g(x)
    r
}
```

```
g <- function(y) {
    r <- y * h(y)
    r
}
```

```
h <- function(z) {
    r <- log(z)
    if (r < 10)
        r^2
    else r^3
}
```

```
> f(-1)
Error in if (r < 10) r^2 else r^3 : missing
value where logical needed
In addition: Warning message:
NaNs produced in: log(x)
```

What happened here? First, the function f was halted somewhere because of a bug. Furthermore, we got a warning from taking the log of a negative number. However, it's not immediately clear where the error occurred during the execution. Did f fail at the top level or at some lower level function? Upon receiving this error, we could immediately run traceback to find out:

```
> traceback()
3: h(y)
2: g(x)
1: f(-1)
```

traceback prints the sequence of function calls in reverse order from the top.
So here, the function on the bottom, f, was called first, then g, then h.
From the traceback output, we can see that the error occurred in h and not in f or g.

The R debug Command

- **debug takes a single argument | the name of a function.**
- **When you pass the name of a function to debug, that function is tagged for debugging.**
- **In order to unflag a function, there is the corresponding undebug function. When a function is flagged for debugging, it does not execute on the usual way. Rather, each statement in the function is executed one at a time and the user can control when each statement gets executed. After a statement is executed, the function suspends and the user is free to interact with the environment. This kind of functionality is what most programmers refer to as “using the debugger" in other languages.**

Our Toy Problem

```
SS <- function(mu, x) {
    d <- x - mu
    d2 <- d^2
    ss <- sum(d2)
    ss
}
```

The function SS in Action

- **The function SS simply computes the sum of squares. It is written here in a rather drawn out fashion for demonstration purposes only.**
- **Now we generate a Normal random sample:**

> set.seed(100) ## set the RNG seed so that the results are reproducible

> x <- rnorm(100)

- **Here, x contains 100 Normal random deviates with (population) mean 0 and variance 1. We can run SS to compute the sum of squares for x and a given value of mu. For example,**

> SS(1, x)

[1] 208.1661

SS Under the Microscope of debug

But suppose we wanted to interact with SS and see how it operates line by line. We need to tag SS for debugging:

```
> debug(SS)
```

The following R session shows how SS runs in the debugger:

```
> SS(1, x)
debugging in: SS(1, x)
debug: {
d <- x - mu
d2 <- d^2
ss <- sum(d2)
ss
}
Browse[1]> n
debug: d <- x - mu
Browse[1]> n
debug: d2 <- d^2
Browse[1]> n
debug: ss <- sum(d2)
Browse[1]> n
debug: ss
Browse[1]> n
exiting from: SS(1, x)
[1] 208.1661
```

What Happened?

Browse[1]>

You are now in what is called the \browser". Here you can enter one of four basic debug commands. Typing n executes the current line and prints the next one. At the very beginning of a function there is nothing to execute so typing n just prints the rst line of code. Typing c executes the rest of the function without stopping and causes the function to return. This is useful if you are done debugging in the middle of a function and don't want to step through the rest of the lines. Typing Q quits debugging and completely halts execution of the function. Finally, you can type where to show where you are in the function call stack. This is much like running a traceback in the debugger (but not quite the same). Besides the four basic debugging commands mentioned above, you can also type other relevant commands. For example, typing ls() will show all objects in the local environment.

You can also make assignments and create new objects while in the debugger. Of course, any new objects created in the local environment will disappear when the debugger finishes.

If you want to inspect the value of a particular object in the local environment, you can print its value, either by using print or by simply typing the name of the object and hitting return. If you have objects in your environment with the names n, c, or Q, then you must explicitly use the print function to print their values (i.e. print(n) or print(c)).

Another SS debug Session - I

```
> SS(2, x)
debugging in: SS(2, x)
debug: {
d <- x - mu
d2 <- d^2
ss <- sum(d2)
ss
}
Browse[1]> n
debug: d <- x - mu
Browse[1]> d[1] ## Print the value of first element of d
[1] -0.4856523
Browse[1]> n
debug: d2 <- d^2
Browse[1]> hist(d2) ## Make a histogram (not shown)
Browse[1]> n
debug: ss <- sum(d2)
Browse[1]> n
debug: ss
```

Another SS debug Session - II

Browse[1]> print(ss) ## Show value of ss; using print() is optional here

[1] 503.814

Browse[1]> ls()

[1] "d" "d2" "mu" "ss" "x"

Browse[1]> where

where 1: SS(2, x)

Browse[1]> y <- x^2 ## Create new object

Browse[1]> ls()

[1] "d" "d2" "mu" "ss" "x" "y"

Browse[1]> y

[1] 2.293249e+00 1.043871e+00 5.158531e-01 3.677514e-01 1.658905e+00

[... omitted ...]

Browse[1]> c ## Execute rest of function without stepping

exiting from: SS(2, x)

[1] 503.814

> undebug(SS) ## Remove debugging flag for SS

Invoking debug on the``Fly'' - I

```
> debug(SS)
> SS(2, x)
debugging in: SS(2, x)
debug: {
d <- x - mu
d2 <- d^2
ss <- sum(d2)
ss
}
Browse[1]> n
debug: d <- x - mu
Browse[1]> n
debug: d2 <- d^2
Browse[1]> n
debug: ss <- sum(d2)
Browse[1]> debug(sum) ## Flag sum for debugging
Browse[1]> n
debugging in: sum(d2)
```

Invoking debug on the``Fly'' - II

debug: .Internal(sum(..., na.rm = na.rm))

Browse[1]> where ## Print the call stack; there are 2 levels now

where 1: sum(d2)

where 2: SS(2, x)

Browse[1]> n

exiting from: sum(d2)

debug: ss

Browse[1]> n

exiting from: SS(2, x)

[1] 503.814

> undebug(SS); undebug(sum)

Explicit Calls to browser

- **It is possible to do a kind of \manual debugging" if you don't feel like stepping through a function line by line.**
- **The function browser can be used to suspend execution of a function so that the user can browse the local environment.**
- **Suppose we edited the SS function from above to look like:**

```
SS <- function(mu, x) {
d <- x - mu
d2 <- d^2
browser()
ss <- sum(d2)
ss
}
```

Now, when the function reaches the third statement in the program, execution will suspend and you will get a Browse[1]> prompt, much like in the debugger.

Our Function With a browser Prompt

> SS(2, x)

Called from: SS(2, x)

Browse[1]> ls()

[1] "d" "d2" "mu" "x"

Browse[1]> print(mu)

[1] 2

Browse[1]> mean(x)

[1] 0.02176075

Browse[1]> n

debug: ss <- sum(d2)

Browse[1]> c

[1] 503.814

Final Thoughts

- trace
  - Useful for making modifications to functions on the fly
- recover
  - Allows us to jump up to a higher level in the execution stack