



Lecture 1: R Basics



Outline

- Why R, and R Paradigm
- References, Tutorials and links
- R Overview
- R Interface
- R Workspace
- Help
- R Packages
- Input/Output
- Reusing Results



Why R?

It's free!

It runs on a variety of platforms including Windows, Unix and MacOS.

It provides an unparalleled platform for programming new statistical methods in an easy and straightforward manner.

It contains advanced statistical routines not yet available in other packages.

It has state-of-the-art graphics capabilities.



R has a Steep Learning Curve

(steeper for those that knew SAS or other software before)

First, while there are many introductory tutorials (covering data types, basic commands, the interface), none alone are comprehensive. In part, this is because much of the advanced functionality of **R** comes from hundreds of user contributed packages. Hunting for what you want can be time consuming, and it can be hard to get a clear overview of what procedures are available.



R has a Learning Curve

(steeper for those that knew SAS or other software before)

The **second** reason is more transient. As users of statistical packages, we tend to run one controlled procedure for each type of analysis. Think of PROC GLM in SAS. We can carefully set up the run with all the parameters and options that we need. When we run the procedure, the resulting output may be a hundred pages long. We then sift through this output pulling out what we need and discarding the rest.



R paradigm is different

Rather than setting up a complete analysis at once, the process is highly interactive. You run a command (say fit a model), take the results and process it through another command (say a set of diagnostic plots), take those results and process it through another command (say cross-validation), etc. The cycle may include transforming the data, and looping back through the whole process again. You stop when you feel that you have fully analyzed the data.



How to download?

- Google it using R or CRAN
(Comprehensive R Archive Network)
- <http://www.r-project.org>



Tutorials

Each of the following tutorials are in PDF format.

- P. Kuhnert & B. Venables, [An Introduction to R: Software for Statistical Modeling & Computing](#)
- J.H. Maindonald, [Using R for Data Analysis and Graphics](#)
- B. Muenchen, [R for SAS and SPSS Users](#)
- W.J. Owen, [The R Guide](#)
- D. Rossiter, [Introduction to the R Project for Statistical Computing for Use at the ITC](#)
- W.N. Venables & D. M. Smith, [An Introduction to R](#)



Web links

- Paul Geissler's [excellent R tutorial](#)
- [Dave Robert's Excellent Labs](#) on Ecological Analysis
- [Excellent Tutorials by David Rossitier](#)
- [Excellent tutorial an nearly every aspect of R](#) (c/o Rob Kabacoff) **MOST of these notes follow this web page format**
- [Introduction to R by Vincent Zoonekynd](#)
- [R Cookbook](#)
- [Data Manipulation Reference](#)



Web links

- [R time series tutorial](#)
- [R Concepts and Data Types](#)
presentation by Deepayan Sarkar
- [Interpreting Output From lm\(\)](#)
- [The R Wiki](#)
- [An Introduction to R](#)
- [Import / Export Manual](#)
- [R Reference Cards](#)



Web links

- [KickStart](#)
- [Hints on plotting data in R](#)
- [Regression and ANOVA](#)
- [Appendices to Fox Book on Regression](#)
- [JGR](#) a Java-based GUI for R
[Mac|Windows|Linux]
- [A Handbook of Statistical Analyses Using R](#) (Brian S. Everitt and Torsten Hothorn)



R Overview

R is a comprehensive statistical and graphical programming language and is a dialect of the S language:

1988 - S2: RA Becker, JM Chambers, A Wilks

1992 - S3: JM Chambers, TJ Hastie

1998 - S4: JM Chambers

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 15 people with access to common CVS archive.



R Overview

You can enter commands one at a time at the command prompt (`>`) or run a set of commands from a source file.

There is a wide variety of data types, including vectors (numerical, character, logical), matrices, dataframes, and lists.

To quit R, use

```
>q()
```



R Overview

Most functionality is provided through built-in and user-created functions and all data objects are kept in memory during an interactive session.

Basic functions are available by default. Other functions are contained in packages that can be attached to a current session as needed



R Overview

A key skill to using **R** effectively is learning how to use the built-in help system. Other sections describe the working environment, inputting programs and outputting results, installing new functionality through packages and etc.

A fundamental design feature of **R** is that the output from most functions can be used as input to other functions. This is described in reusing results.



R Interface

Start the R system, the main window (RGui) with a sub window (R Console) will appear

In the `Console' window the cursor is waiting for you to type in some R commands.

Your First R Session

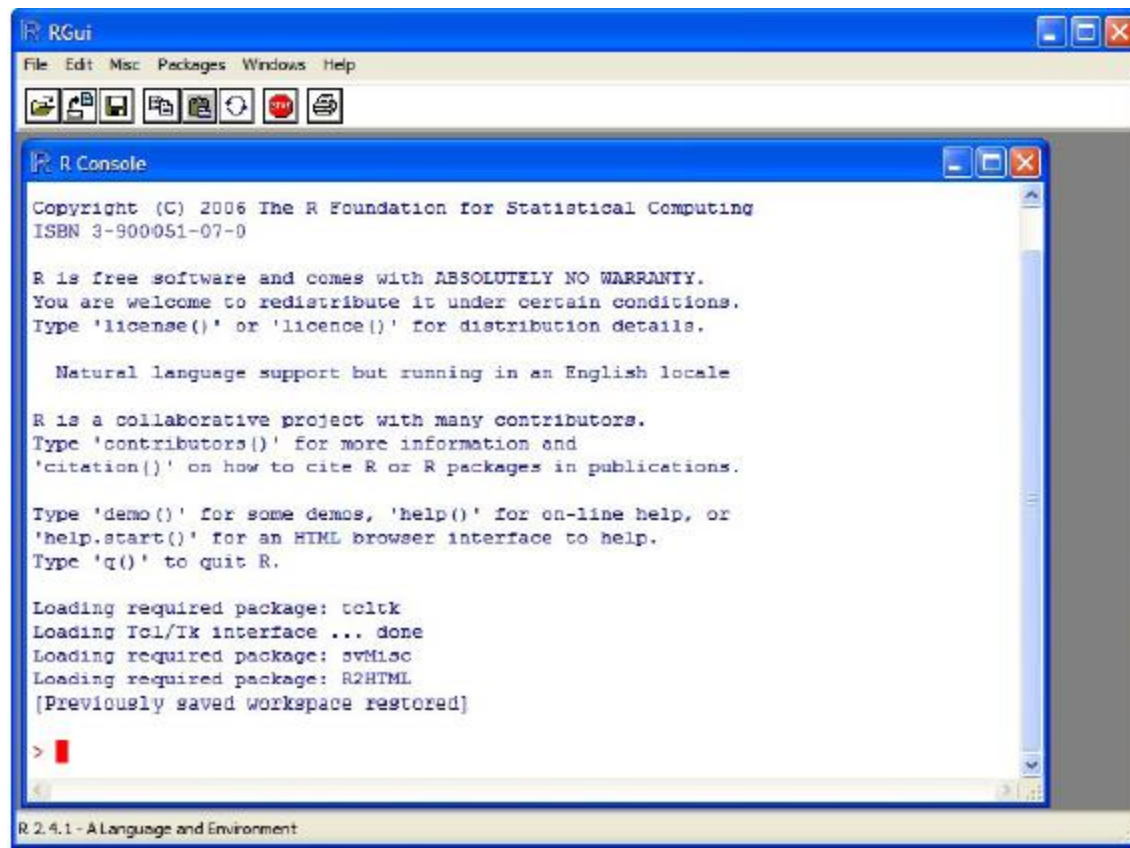


Figure 1.1: The R system on Windows



R Introduction

- Results of calculations can be stored in objects using the assignment operators:
 - An arrow (`<-`) formed by a smaller than character and a hyphen without a space!
 - The equal character (`=`).



R Introduction

■ These objects can then be used in other calculations. To print the object just enter the name of the object. There are some restrictions when giving an object a name:

- Object names cannot contain 'strange' symbols like `!`, `+`, `-`, `#`.
- A dot (`.`) and an underscore (`_`) are allowed, also a name starting with a dot.
- Object names can contain a number but cannot start with a number.
- R is case sensitive, `X` and `x` are two different objects, as well as `temp` and `tempP`.



An example

```
> # An example
> x <- c(1:10)
> x[(x>8) | (x<5)]
> # yields 1 2 3 4 9 10
> # How it works
> x <- c(1:10)
> X
> 1 2 3 4 5 6 7 8 9 10
> x > 8
> F F F F F F F T T
> x < 5
> T T T T F F F F F
> x > 8 | x < 5
> T T T T F F F T T
> x[c(T,T,T,T,F,F,F,T,T)]
> 1 2 3 4 9 10
```



R Introduction

- To list the objects that you have in your current R session use the function `ls` or the function `objects`.

```
> ls()
[1] "x" "y"
```
- So to run the function `ls` we need to enter the name followed by an opening (and and aclosing). Entering only `ls` will just print the object, you will see the underlying R code of the the function `ls`. Most functions in R accept certain arguments. For example, one of the arguments of the function `ls` is `pattern`. To list all objects starting with the letter `x`:

```
> x2 = 9
> y2 = 10
> ls(pattern="x")
[1] "x" "x2"
```



R Introduction

- If you assign a value to an object that already exists then the contents of the object will be overwritten with the new value (without a warning!). Use the function `rm` to remove one or more objects from your session.

```
> rm(x, x2)
```

- Lets create two small vectors with data and a scatterplot.

```
z2 <- c(1,2,3,4,5,6)
```

```
z3 <- c(6,8,3,5,7,1)
```

```
plot(z2,z3)
```

```
title("My first scatterplot")
```



R Warning !

R is a case sensitive language.

FOO, Foo, and foo are three different objects



R Introduction

```
> x = sin(9)/75
> y = log(x) + x^2
> x
[1] 0.005494913
> y
[1] -5.203902
> m <- matrix(c(1,2,4,1), ncol=2)
> m
> [,1] [,2]
[1,] 1 4
[2,] 2 1
> solve(m)
[,1] [,2]
[1,] -0.1428571 0.5714286
[2,] 0.2857143 -0.1428571
```




R Workspace

Objects that you create during an R session are held in memory, the collection of objects that you currently have is called the workspace. This workspace is not saved on disk unless you tell R to do so. This means that your objects are lost when you close R and not save the objects, or worse when R or your system crashes on you during a session.



R Workspace

When you close the RGui or the R console window, the system will ask if you want to save the workspace image. If you select to save the workspace image then all the objects in your current R session are saved in a file `.RData`. This is a binary file located in the working directory of R, which is by default the installation directory of R.



R Workspace

- During your R session you can also explicitly save the workspace image. Go to the `File` menu and then select `Save Workspace...`, or use the `save.image` function.

```
## save to the current working directory
```

```
save.image()
```

```
## just checking what the current working directory is  
getwd()
```

```
## save to a specific file and location
```

```
save.image("C:\\Program Files\\R\\R-  
2.5.0\\bin\\.RData")
```



R Workspace

If you have saved a workspace image and you start R the next time, it will restore the workspace. So all your previously saved objects are available again. You can also explicitly load a saved workspace file, that could be the workspace image of someone else. Go to the 'File' menu and select 'Load workspace...'.



R Workspace

Commands are entered interactively at the **R** user prompt. **Up** and **down arrow keys** scroll through your command history.

You will probably want to keep different projects in different physical directories.



R Workspace

R gets confused if you use a path in your code like

c:|mydocuments|myfile.txt

This is because R sees "\" as an escape character. Instead, use

c:||my documents||myfile.txt

or

c:/mydocuments/myfile.txt



R Workspace

`getwd()` # print the current working directory

`ls()` # list the objects in the current workspace

`setwd(mydirectory)` # change to mydirectory

`setwd("c:/docs/mydir")`



R Workspace

```
#view and set options for the session  
  help(options) # learn about available options  
  options() # view current option settings  
  options(digits=3) # number of digits to print  
  on output  
  
# work with your previous commands  
  history() # display last 25 commands  
  history(max.show=Inf) # display all previous commands
```




R Workspace

```
# save your command history  
savehistory(file="myfile") # default is ".Rhistory"  
  
# recall your command history  
loadhistory(file="myfile") # default is ".Rhistory"
```



R Help

Once **R** is installed, there is a comprehensive built-in help system. At the program's command prompt you can use any of the following:

```
help.start()  # general help
help(foo)     # help about function foo
?foo         # same thing
apropos("foo") # list all function containing string foo
example(foo)  # show an example of function foo
```



R Help

```
# search for foo in help manuals and archived mailing lists
RSiteSearch("foo")

# get vignettes on using installed packages
vignette()      # show available vignettes
vignette("foo") # show specific vignette
```



R Datasets

R comes with a number of sample datasets that you can experiment with. Type

> data()

to see the available datasets. The results will depend on which packages you have loaded. Type

help(*datasetname*)

for details on a sample dataset.



R Packages

- One of the strengths of R is that the system can easily be extended. The system allows you to write new functions and package those functions in a so called 'R package' (or 'R library'). The R package may also contain other R objects, for example data sets or documentation. There is a lively R user community and many R packages have been written and made available on CRAN for other users. Just a few examples, there are packages for portfolio optimization, drawing maps, exporting objects to html, time series analysis, spatial statistics and the list goes on and on.



R Packages

- When you download R, already a number (around 30) of packages are downloaded as well. To use a function in an R package, that package has to be attached to the system. When you start R not all of the downloaded packages are attached, only seven packages are attached to the system by default. You can use the function `search` to see a list of packages that are currently attached to the system, this list is also called the search path.

```
> search()
```

```
[1] ".GlobalEnv" "package:stats" "package:graphics"
```

```
[4] "package:grDevices" "package:datasets" "package:utils"
```

```
[7] "package:methods" "Autoloads" "package:base"
```



R Packages

To attach another package to the system you can use the menu or the library function. Via the menu:

Select the 'Packages' menu and select 'Load package...', a list of available packages on your system will be displayed. Select one and click 'OK', the package is now attached to your current R session. Via the library function:

```
> library(MASS)
```

```
> shoes
```

```
$A
```

```
[1] 13.2 8.2 10.9 14.3 10.7 6.6 9.5 10.8 8.8 13.3
```

```
$B
```

```
[1] 14.0 8.8 11.2 14.2 11.8 6.4 9.8 11.3 9.3 13.6
```



R Packages

- The function library can also be used to list all the available libraries on your system with a short description. Run the function without any arguments

```
> library()
```

```
Packages in library 'C:/PROGRA~1/R/R-25~1.0/library':
```

base	The R Base Package
Boot	Bootstrap R (S-Plus) Functions (Canty)
class	Functions for Classification
cluster	Cluster Analysis Extended Rousseeuw et al.
codetools	Code Analysis Tools for R
datasets	The R Datasets Package
DBI	R Database Interface
foreign	Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, dBase, ...
graphics	The R Graphics Package
	Applied Statistical Computing and Graphics



R Packages

```
install = function() {  
  install.packages(c("moments", "graphics", "Rcmdr", "hexbin"),  
    repos="http://lib.stat.cmu.edu/R/CRAN")  
}  
install()
```



R Conflicting objects

- It is not recommended to do, but R allows the user to give an object a name that already exists. If you are not sure if a name already exists, just enter the name in the R console and see if R can find it. R will look for the object in all the libraries (packages) that are currently attached to the R system. R will not warn you when you use an existing name.

```
> mean = 10
```

```
> mean
```

```
[1] 10
```

- The object mean already exists in the base package, but is now masked by your object mean. To get a list of all masked objects use the function conflicts.

```
>
```

```
[1] "body<-" "mean"
```



R Conflicting objects

The object mean already exists in the base package, but is now masked by your object mean. To get a list of all masked objects use the function `conflicts()`.

```
> conflicts()  
[1] "body<-" "mean"
```

You can safely remove the object mean with the function `rm()` without risking deletion of the mean function.

Calling `rm()` removes only objects in your working environment by default.



Source Codes

you can have input come from a script file (a file containing **R** commands) and direct output to a variety of destinations.

Input

The **source()** function runs a script in the current session. If the filename does not include a path, the file is taken from the current working directory.

```
# input a script  
source("myfile")
```



Output

Output

The **sink()** function defines the direction of the output.

direct output to a file

```
sink("myfile", append=FALSE, split=FALSE)
```

return output to the terminal

```
sink()
```



Output

The **append** option controls whether output overwrites or adds to a file.

The **split** option determines if output is also sent to the screen as well as the output file.

Here are some examples of the **sink()** function.

```
# output directed to output.txt in c:\projects directory.
```

```
# output overwrites existing file. no output to terminal.  
sink("myfile.txt", append=TRUE, split=TRUE)
```



Graphs

To redirect graphic output use one of the following functions. Use **dev.off()** to return output to the terminal.

Function	Output to
<code>pdf("mygraph.pdf")</code>	pdf file
<code>win.metafile("mygraph.wmf")</code>	windows metafile
<code>png("mygraph.png")</code>	png file
<code>jpeg("mygraph.jpg")</code>	jpeg file
<code>bmp("mygraph.bmp")</code>	bmp file
<code>postscript("mygraph.ps")</code>	postscript file



Redirecting Graphs

```
# example - output graph to jpeg file  
jpeg("c:/mygraphs/myplot.jpg")  
plot(x)  
dev.off()
```




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.

Example 1

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

This will run a simple linear regression of miles per gallon on car weight using the dataframe mtcars. Results are sent to the screen. Nothing is saved.



Reusing Results

Example 2

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

This time, the same regression is performed but the results are saved under the name `fit`. No output is sent to the screen. However, you now can manipulate the results.

```
str(fit) # view the contents/structure of "fit"
```

The assignment has actually created a list called "fit" that contains a wide range of information (including the predicted values, residuals, coefficients, and more).



Reusing Results

```
# plot residuals by fitted values  
plot(fit$residuals, fit$fitted.values)
```

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)**.

The results can also be used by a wide range of other functions.

```
# produce diagnostic plots  
plot(fit)
```



Lecture 2: Data Input



Outline

- Data Types
- Importing Data
- Keyboard Input
- Database Input
- Exporting Data
- Viewing Data
- Variable Labels
- Value Labels
- Missing Data
- Date Values



Data Types

R has a wide variety of data types including scalars, vectors (numerical, character, logical), matrices, dataframes, and lists.



Vectors

```
a <- c(1,2,5.3,6,-2,4) # numeric vector
```

```
b <- c("one","two","three") # character vector
```

```
c <- c(TRUE,TRUE,TRUE,FALSE,TRUE,FALSE)  
#logical vector
```

Refer to elements of a vector using subscripts.

```
a[c(2,4)] # 2nd and 4th elements of vector
```



Matrices

All columns in a matrix must have the same mode(numeric, character, etc.) and the same length.

The general format is

```
mymatrix <- matrix(vector, nrow=r, ncol=c,  
  byrow=FALSE,dimnames=list(char_vector_rownames  
  , char_vector_colnames))
```

byrow=TRUE indicates that the matrix should be filled by rows. **byrow=FALSE** indicates that the matrix should be filled by columns (the default). **dimnames** provides optional labels for the columns and rows.



Matrices

```
# generates 5 x 4 numeric matrix
y<-matrix(1:20, nrow=5,ncol=4)

# another example
cells <- c(1,26,24,68)
rnames <- c("R1", "R2")
cnames <- c("C1", "C2")
mymatrix <- matrix(cells, nrow=2, ncol=2,
  byrow=TRUE, dimnames=list(rnames, cnames))

#Identify rows, columns or elements using subscripts.
x[,4] # 4th column of matrix
x[3,] # 3rd row of matrix
x[2:4,1:3] # rows 2,3,4 of columns 1,2,3
```



Arrays

Arrays are similar to matrices but can have more than two dimensions. See **help(array)** for details.



Data frames

A data frame is more general than a matrix, in that different columns can have different modes (numeric, character, factor, etc.).

```
d <- c(1,2,3,4)
```

```
e <- c("red", "white", "red", NA)
```

```
f <- c(TRUE,TRUE,TRUE,FALSE)
```

```
mydata <- data.frame(d,e,f)
```

```
names(mydata) <- c("ID","Color","Passed")  
#variable names
```



Data frames

There are a variety of ways to identify the elements of a dataframe .

```
myframe[3:5] # columns 3,4,5 of dataframe
```

```
myframe[c("ID","Age")] # columns ID and Age from dataframe
```

```
myframe$X1 # variable x1 in the dataframe
```



Lists

An ordered collection of objects (components). A list allows you to gather a variety of (possibly unrelated) objects under one name.

example of a list with 4 components -

a string, a numeric vector, a matrix, and a scalar

```
w <- list(name="Fred", mynumbers=a,  
mymatrix=y, age=5.3)
```

example of a list containing two lists

```
v <- c(list1,list2)
```



Lists

Identify elements of a list using the `[[]]` convention.
`mylist[[2]]` # 2nd component of the list



Factors

Tell **R** that a variable is **nominal** by making it a factor. The factor stores the nominal values as a vector of integers in the range [1... k] (where k is the number of unique values in the nominal variable), and an internal vector of character strings (the original values) mapped to these integers.

```
# variable gender with 20 "male" entries and
```

```
# 30 "female" entries
```

```
gender <- c(rep("male",20), rep("female", 30))
```

```
gender <- factor(gender)
```

```
# stores gender as 20 1s and 30 2s and associates
```

```
# 1=female, 2=male internally (alphabetically)
```

```
# R now treats gender as a nominal variable
```

```
summary(gender)
```



Useful Functions

`length(object)` # number of elements or components

`str(object)` # structure of an object

`class(object)` # class or type of an object

`names(object)` # names

`c(object,object,...)` # combine objects into a vector

`cbind(object, object, ...)` # combine objects as columns

`rbind(object, object, ...)` # combine objects as rows

`ls()` # list current objects

`rm(object)` # delete an object

`newobject <- edit(object)` # edit copy and save a
newobject

`fix(object)` # edit in place



Importing Data

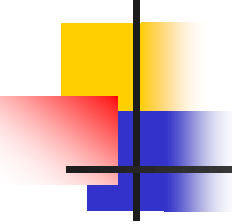
Importing data into **R** is fairly simple.

For Stata and Systat, use the **foreign** package.

For SPSS and SAS I would recommend the **Hmisc** package for ease and functionality.

See the **Quick-R** section on **packages**, for information on obtaining and installing the these packages.

Example of importing data are provided below.



From A Comma Delimited Text File

first row contains variable names, comma is
separator

assign the variable *id* to row names

note the / instead of \ on mswindows systems

```
mydata <- read.table("c:/mydata.csv",  
header=TRUE, sep="," , row.names="id")
```



From Excel

The best way to read an Excel file is to export it to a comma delimited file and import it using the method above.

On windows systems you can use the **RODBC** package to access Excel files. The first row should contain variable/column names.

first row contains variable names

we will read in workSheet *mysheet*

```
library(RODBC)
```

```
channel <- odbcConnectExcel("c:/myexcel.xls")
```

```
mydata <- sqlFetch(channel, "mysheet")
```

```
odbcClose(channel)
```



From SAS

- `# save SAS dataset in transport format`
`libname out xport 'c:/mydata.xpt';`
`data out.mydata;`
`set sasuser.mydata;`
`run;`
- `library(foreign)`
`#bsl=read.xport("mydata.xpt")`



Keyboard Input

Usually you will obtain a dataframe by importing it from **SAS**, **SPSS**, **Excel**, **Stata**, a database, or an ASCII file. To create it interactively, you can do something like the following.

```
# create a dataframe from scratch
age <- c(25, 30, 56)
gender <- c("male", "female", "male")
weight <- c(160, 110, 220)
mydata <- data.frame(age,gender,weight)
```



Keyboard Input

You can also use **R**'s built in spreadsheet to enter the data interactively, as in the following example.

```
# enter data using editor
mydata <- data.frame(age=numeric(0),
gender=character(0), weight=numeric(0))
mydata <- edit(mydata)
# note that without the assignment in the line
above,
# the edits are not saved!
```



Exporting Data

There are numerous methods for exporting **R** objects into other formats . For SPSS, SAS and Stata. you will need to load the foreign packages. For Excel, you will need the xlsReadWrite package.



Exporting Data

To A Tab Delimited Text File

```
write.table(mydata, "c:/mydata.txt", sep="\t")
```

To an Excel Spreadsheet

```
library(xlsReadWrite)  
write.xls(mydata, "c:/mydata.xls")
```

To SAS

```
library(foreign)  
write.foreign(mydata, "c:/mydata.txt",  
"c:/mydata.sas", package="SAS")
```




Viewing Data

There are a number of functions for listing the contents of an object or dataset.

list objects in the working environment

`ls()`

list the variables in mydata

`names(mydata)`

list the structure of mydata

`str(mydata)`

list levels of factor v1 in mydata

`levels(mydata$v1)`

dimensions of an object

`dim(object)`



Viewing Data

There are a number of functions for listing the contents of an object or dataset.

class of an object (numeric, matrix, dataframe, etc)
`class(object)`

print mydata
`mydata`

print first 10 rows of mydata
`head(mydata, n=10)`

print last 5 rows of mydata
`tail(mydata, n=5)`



Variable Labels

R's ability to handle variable labels is somewhat unsatisfying.

If you use the **Hmisc** package, you can take advantage of some labeling features.

```
library(Hmisc)
label(mydata$myvar) <- "Variable label for variable
myvar"
describe(mydata)
```



Variable Labels

Unfortunately the label is only in effect for functions provided by the **Hmisc** package, such as **describe()**. Your other option is to use the variable label as the variable name and then refer to the variable by position index.

```
names(mydata)[3] <- "This is the label for variable 3"  
mydata[3] # list the variable
```



Value Labels

To understand value labels in **R**, you need to understand the data structure factor.

You can use the factor function to create your own value labels.

```
# variable v1 is coded 1, 2 or 3
```

```
# we want to attach value labels 1=red, 2=blue,3=green
```

```
mydata$v1 <- factor(mydata$v1,  
  levels = c(1,2,3),  
  labels = c("red", "blue", "green"))
```

```
# variable y is coded 1, 3 or 5
```

```
# we want to attach value labels 1=Low, 3=Medium, 5=High
```



Value Labels

```
mydata$v1 <- ordered(mydata$y,  
  levels = c(1,3, 5),  
  labels = c("Low", "Medium", "High"))
```

Use the **factor()** function for **nominal data** and the **ordered()** function for **ordinal data**. R statistical and graphic functions will then treat the data appropriately.

Note: factor and ordered are used the same way, with the same arguments. The former creates factors and the latter creates ordered factors.



Missing Data

In **R**, missing values are represented by the symbol **NA** (not available) . Impossible values (e.g., dividing by zero) are represented by the symbol **NaN** (not a number). Unlike SAS, **R** uses the same symbol for character and numeric data.

Testing for Missing Values

`is.na(x)` # returns TRUE if x is missing

```
y <- c(1,2,3,NA)
```

`is.na(y)` # returns a vector (F F F T)



Missing Data

Recoding Values to Missing

```
# recode 99 to missing for variable v1  
# select rows where v1 is 99 and recode column v1  
mydata[mydata$v1==99,"v1"] <- NA
```

Excluding Missing Values from Analyses

Arithmetic functions on missing values yield missing values.

```
x <- c(1,2,NA,3)  
mean(x)           # returns NA  
mean(x, na.rm=TRUE) # returns 2
```




Missing Data

The function **complete.cases()** returns a logical vector indicating which cases are complete.

```
# list rows of data that have missing values  
mydata[!complete.cases(mydata),]
```

The function **na.omit()** returns the object with listwise deletion of missing values.

```
# create new dataset without missing data  
newdata <- na.omit(mydata)
```



Missing Data

Advanced Handling of Missing Data

Most modeling functions in **R** offer options for dealing with missing values. You can go beyond pairwise or listwise deletion of missing values through methods such as multiple imputation. Good implementations that can be accessed through **R** include **Amelia II**, **Mice**, and **mitools**.



Date Values

Dates are represented as the number of days since 1970-01-01, with negative values for earlier dates.

```
# use as.Date( ) to convert strings to dates  
mydates <- as.Date(c("2007-06-22", "2004-02-13"))  
# number of days between 6/22/07 and 2/13/04  
days <- mydates[1] - mydates[2]
```

Sys.Date() returns today's date.

Date() returns the current date and time.



Date Values

The following symbols can be used with the `format()` function to print dates.

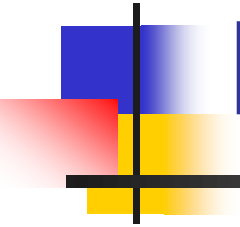
Symbol	Meaning	Example
%d	day as a number (0-31)	01-31
%a	abbreviated weekday	Mon
%A	unabbreviated weekday	Monday
%m	month (00-12)	00-12
%b	abbreviated month	Jan
%B	unabbreviated month	January
%y	2-digit year	07
%Y	4-digit year	2007



Date Values

```
# print today's date  
today <- Sys.Date()  
format(today, format="%B %d %Y")  
"June 20 2007"
```

Lecture 3: Data Manipulation





Outline

- Creating New Variable
- Operators
- Built-in functions
- Control Structures
- User Defined Functions
- Sorting Data
- Merging Data
- Aggregating Data
- Reshaping Data
- Sub-setting Data
- Data Type Conversions



Introduction

Once you have access to your data, you will want to massage it into useful form. This includes creating new variables (including recoding and renaming existing variables), sorting and merging datasets, aggregating data, reshaping data, and subsetting datasets (including selecting observations that meet criteria, randomly sampling observation, and dropping or keeping variables).



Introduction

Each of these activities usually involve the use of **R**'s built-in operators (arithmetic and logical) and functions (numeric, character, and statistical). Additionally, you may need to use control structures (if-then, for, while, switch) in your programs and/or create your own functions. Finally you may need to convert variables or datasets from one type to another (e.g. numeric to character or matrix to dataframe).



Creating new variables

- Use the assignment operator `<-` to create new variables. A wide array of [operators](#) and [functions](#) are available here.
- # Three examples for doing the same computations

```
mydata$sum <- mydata$x1 + mydata$x2  
mydata$mean <- (mydata$x1 + mydata$x2)/2
```

```
attach(mydata)  
mydata$sum <- x1 + x2  
mydata$mean <- (x1 + x2)/2  
detach(mydata)
```

- ```
mydata <- transform(mydata,
 sum = x1 + x2,
 mean = (x1 + x2)/2
)
```



# Creating new variables

---

## Recoding variables

- In order to recode data, you will probably use one or more of R's control structures.

- # create 2 age categories  
mydata\$agecat <- ifelse(mydata\$age > 70,  
c("older"), c("younger"))  
# another example: create 3 age categories  
attach(mydata)  
mydata\$agecat[age > 75] <- "Elder"  
mydata\$agecat[age > 45 & age <= 75] <-  
"Middle Aged"  
mydata\$agecat[age <= 45] <- "Young"  
detach(mydata)



# Creating new variables

---

## Recoding variables

- In order to recode data, you will probably use one or more of R's control structures.

- `# create 2 age categories`  
`mydata$agecat <- ifelse(mydata$age > 70,`  
`c("older"), c("younger"))`

`# another example: create 3 age categories`

`attach(mydata)`

`mydata$agecat[age > 75] <- "Elder"`

`mydata$agecat[age > 45 & age <= 75] <- "Middle`  
`Aged"`

`mydata$agecat[age <= 45] <- "Young"`

`detach(mydata)`



# Creating new variables

---

## Renaming variables

- You can rename variables programmatically or interactively.
- `# rename interactively`  
`fix(mydata) # results are saved on close`

```
rename programmatically
library(reshape)
mydata <- rename(mydata, c(oldname="newname"))
```

```
you can re-enter all the variable names in order
changing the ones you need to change.the limitation
is that you need to enter all of them!
names(mydata) <- c("x1","age","y", "ses")
```



# Arithmetic Operators

---

| Operator       | Description                  |
|----------------|------------------------------|
| +              | addition                     |
| -              | subtraction                  |
| *              | multiplication               |
| /              | division                     |
| <b>^ or **</b> | exponentiation               |
| <b>x % y</b>   | modulus (x mod y) 5 % 2 is 1 |
| <b>x %/ y</b>  | integer division 5 %/ 2 is 2 |



# Logical Operators

---

| Operator  | Description              |
|-----------|--------------------------|
| <         | less than                |
| <=        | less than or equal to    |
| >         | greater than             |
| >=        | greater than or equal to |
| ==        | exactly equal to         |
| !=        | not equal to             |
| !x        | Not x                    |
| x   y     | x OR y                   |
| x & y     | x AND y                  |
| isTRUE(x) | test if x is TRUE        |



# Control Structures

---

- **R** has the standard control structures you would expect. **expr** can be multiple (compound) statements by enclosing them in braces { }. It is more efficient to use built-in functions rather than control structures whenever possible.





# Control Structures

---

- **if-else**

- if (*cond*) *expr*  
if (*cond*) *expr1* else *expr2*

- **for**

- for (*var* in *seq*) *expr*

- **while**

- while (*cond*) *expr*

- **switch**

- switch(*expr*, ...)

- **ifelse**

- ifelse(*test*, *yes*, *no*)



# Control Structures

---

- # transpose of a matrix  
# a poor alternative to built-in t() function

```
mytrans <- function(x) {
 if (!is.matrix(x)) {
 warning("argument is not a matrix: returning NA")
 return(NA_real_)
 }
 y <- matrix(1, nrow=ncol(x), ncol=nrow(x))
 for (i in 1:nrow(x)) {
 for (j in 1:ncol(x)) {
 y[j,i] <- x[i,j]
 }
 }
 return(y)
}
```



# Control Structures

---

- # try it  
z <- matrix(1:10, nrow=5, ncol=2)  
tz <- mytrans(z)



# R built-in functions

---

Almost everything in **R** is done through functions. Here I'm only referring to numeric and character functions that are commonly used in creating or recoding variables.

Note that while the examples on this page apply functions to individual variables, many can be applied to vectors and matrices as well.



# Numeric Functions

| Function                                                   | Description                                       |
|------------------------------------------------------------|---------------------------------------------------|
| <b>abs</b> ( $x$ )                                         | absolute value                                    |
| <b>sqrt</b> ( $x$ )                                        | square root                                       |
| <b>ceiling</b> ( $x$ )                                     | ceiling(3.475) is 4                               |
| <b>floor</b> ( $x$ )                                       | floor(3.475) is 3                                 |
| <b>trunc</b> ( $x$ )                                       | trunc(5.99) is 5                                  |
| <b>round</b> ( $x$ , digits= $n$ )                         | round(3.475, digits=2) is 3.48                    |
| <b>signif</b> ( $x$ , digits= $n$ )                        | signif(3.475, digits=2) is 3.5                    |
| <b>cos</b> ( $x$ ), <b>sin</b> ( $x$ ), <b>tan</b> ( $x$ ) | also acos( $x$ ), cosh( $x$ ), acosh( $x$ ), etc. |
| <b>log</b> ( $x$ )                                         | natural logarithm                                 |
| <b>log10</b> ( $x$ )                                       | common logarithm                                  |
| <b>exp</b> ( $x$ )                                         | $e^x$                                             |



# Character Functions

| Function                                                                                                         | Description                                                                                                                                                                                                                                                                     |
|------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>substr</b> ( <i>x</i> , <b>start</b> = <i>n1</i> , <b>stop</b> = <i>n2</i> )                                  | Extract or replace substrings in a character vector.<br><code>x &lt;- "abcdef"</code><br><code>substr(x, 2, 4)</code> is "bcd"<br><code>substr(x, 2, 4) &lt;- "22222"</code> is "a222ef"                                                                                        |
| <b>grep</b> ( <i>pattern</i> , <i>x</i> ,<br><b>ignore.case</b> =FALSE, <b>fixed</b> =FALSE)                     | Search for <i>pattern</i> in <i>x</i> . If <b>fixed</b> =FALSE then <i>pattern</i> is a <u>regular expression</u> . If <b>fixed</b> =TRUE then <i>pattern</i> is a text string. Returns matching indices.<br><code>grep("A", c("b","A","c"), fixed=TRUE)</code> returns 2       |
| <b>sub</b> ( <i>pattern</i> , <i>replacement</i> , <i>x</i> ,<br><b>ignore.case</b> =FALSE, <b>fixed</b> =FALSE) | Find <i>pattern</i> in <i>x</i> and replace with <i>replacement</i> text. If <b>fixed</b> =FALSE then <i>pattern</i> is a regular expression.<br>If <b>fixed</b> = T then <i>pattern</i> is a text string.<br><code>sub("\\s", ".", "Hello There")</code> returns "Hello.There" |
| <b>strsplit</b> ( <i>x</i> , <i>split</i> )                                                                      | Split the elements of character vector <i>x</i> at <i>split</i> .<br><code>strsplit("abc", "")</code> returns 3 element vector "a","b","c"                                                                                                                                      |
| <b>paste</b> (..., <b>sep</b> ="")                                                                               | Concatenate strings after using <i>sep</i> string to separate them.<br><code>paste("x", 1:3, sep="")</code> returns c("x1", "x2" "x3")<br><code>paste("x", 1:3, sep="M")</code> returns c("xM1", "xM2" "xM3")<br><code>paste("Today is", date())</code>                         |
| <b>toupper</b> ( <i>x</i> )                                                                                      | Uppercase                                                                                                                                                                                                                                                                       |
| <b>tolower</b> ( <i>x</i> )                                                                                      | Lowercase                                                                                                                                                                                                                                                                       |



# Stat/Prob Functions

---

- The following table describes functions related to probability distributions. For random number generators below, you can use `set.seed(1234)` or some other integer to create reproducible pseudo-random numbers.

| Function                                                                                                                         | Description                                                                                                                                                                                                                                                     |
|----------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>dnorm(x)</b>                                                                                                                  | normal density function (by default m=0 sd=1)<br># plot standard normal curve<br>x <- pretty(c(-3,3), 30)<br>y <- dnorm(x)<br>plot(x, y, type='l', xlab="Normal Deviate", ylab="Density", yaxs="i")                                                             |
| <b>pnorm(q)</b>                                                                                                                  | cumulative normal probability for q<br>(area under the normal curve to the right of q)<br>pnorm(1.96) is 0.975                                                                                                                                                  |
| <b>qnorm(p)</b>                                                                                                                  | normal quantile.<br>value at the p percentile of normal distribution<br>qnorm(.9) is 1.28 # 90th percentile                                                                                                                                                     |
| <b>rnorm(n, m=0,sd=1)</b>                                                                                                        | n random normal deviates with mean m<br>and standard deviation sd.<br>#50 random normal variates with mean=50, sd=10<br>x <- rnorm(50, m=50, sd=10)                                                                                                             |
| <b>dbinom(x, size, prob)</b><br><b>pbinom(q, size, prob)</b><br><b>qbinom(p, size, prob)</b><br><b>rbinom(n, size, prob)</b>     | binomial distribution where size is the sample size<br>and prob is the probability of a heads (pi)<br># prob of 0 to 5 heads of fair coin out of 10 flips<br>dbinom(0:5, 10, .5)<br># prob of 5 or less heads of fair coin out of 10 flips<br>pbinom(5, 10, .5) |
| <b>dpois(x, lamda)</b><br><b>ppois(q, lamda)</b><br><b>qpois(p, lamda)</b><br><b>rpois(n, lamda)</b>                             | poisson distribution with m=std=lamda<br>#probability of 0,1, or 2 events with lamda=4<br>dpois(0:2, 4)<br># probability of at least 3 events with lamda=4<br>1- ppois(2,4)                                                                                     |
| <b>dunif(x, min=0, max=1)</b><br><b>punif(q, min=0, max=1)</b><br><b>qunif(p, min=0, max=1)</b><br><b>runif(n, min=0, max=1)</b> | uniform distribution, follows the same pattern<br>as the normal distribution above.<br>#10 uniform random variates<br>x <- runif(10)                                                                                                                            |



| Function                                 | Description                                                                                                                                                                                       |
|------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>mean(x, trim=0, na.rm=FALSE)</b>      | mean of object x<br># trimmed mean, removing any missing values and<br># 5 percent of highest and lowest scores<br>mx <- mean(x,trim=.05,na.rm=TRUE)                                              |
| <b>sd(x)</b>                             | standard deviation of object(x). also look at var(x) for variance and mad(x) for median absolute deviation.                                                                                       |
| <b>median(x)</b>                         | median                                                                                                                                                                                            |
| <b>quantile(x, probs)</b>                | quantiles where x is the numeric vector whose quantiles are desired and probs is a numeric vector with probabilities in [0,1].<br># 30th and 84th percentiles of x<br>y <- quantile(x, c(.3,.84)) |
| <b>range(x)</b>                          | range                                                                                                                                                                                             |
| <b>sum(x)</b>                            | sum                                                                                                                                                                                               |
| <b>diff(x, lag=1)</b>                    | lagged differences, with lag indicating which lag to use                                                                                                                                          |
| <b>min(x)</b>                            | minimum                                                                                                                                                                                           |
| <b>max(x)</b>                            | maximum                                                                                                                                                                                           |
| <b>scale(x, center=TRUE, scale=TRUE)</b> | column center or standardize a matrix.                                                                                                                                                            |



# Other Useful Functions

---

| Function                                           | Description                                                                      |
|----------------------------------------------------|----------------------------------------------------------------------------------|
| <b>seq</b> ( <i>from</i> , <i>to</i> , <i>by</i> ) | generate a sequence<br>indices <- seq(1,10,2)<br>#indices is c(1, 3, 5, 7, 9)    |
| <b>rep</b> ( <i>x</i> , <i>ntimes</i> )            | repeat <i>x</i> <i>n</i> times<br>y <- rep(1:3, 2)<br># y is c(1, 2, 3, 1, 2, 3) |
| <b>cut</b> ( <i>x</i> , <i>n</i> )                 | divide continuous variable in factor with <i>n</i> levels<br>y <- cut(x, 5)      |



# Sorting

---

- To sort a dataframe in R, use the **order( )** function. By default, sorting is ASCENDING. Prepend the sorting variable by a minus sign to indicate DESCENDING order. Here are some examples.
- # sorting examples using the mtcars dataset  
data(mtcars)  
# sort by mpg  
newdata = mtcars[order(mtcars\$mpg),]  
# sort by mpg and cyl  
newdata <- mtcars[order(mtcars\$mpg, mtcars\$cyl),]  
# sort by mpg (ascending) and cyl (descending)  
newdata <- mtcars[order(mtcars\$mpg, -mtcars\$cyl),]



# Merging

---

To merge two dataframes (datasets) horizontally, use the **merge** function. In most cases, you join two dataframes by one or more common key variables (i.e., an inner join).

```
merge two dataframes by ID
```

```
total <- merge(dataframeA,dataframeB,by="ID")
```

```
merge two dataframes by ID and Country
```

```
total <-
```

```
merge(dataframeA,dataframeB,by=c("ID","Country"))
```



# Merging

---

## ADDING ROWS

To join two dataframes (datasets) vertically, use the **rbind** function. The two dataframes **must** have the same variables, but they do not have to be in the same order.

```
total <- rbind(dataframeA, dataframeB)
```

If dataframeA has variables that dataframeB does not, then either:

Delete the extra variables in dataframeA or

Create the additional variables in dataframeB and set them to NA  
(missing)

before joining them with rbind.



# Aggregating

---

- **It is relatively easy to collapse data in R using one or more BY variables and a defined function.**
- `# aggregate dataframe mtcars by cyl and vs, returning means`  
`# for numeric variables`  
`attach(mtcars)`  
`aggdata <- aggregate(mtcars, by=list(cyl), FUN=mean, na.rm=TRUE)`  
`print(aggdata)`
- OR use `apply`



# Aggregating

---

- When using the `aggregate()` function, the `by` variables must be in a list (even if there is only one). The function can be built-in or user provided.
- See also:
- `summarize()` in the Hmisc package
- summaryBy() in the doBy package



# Data Type Conversion

---

- Type conversions in R work as you would expect. For example, adding a character string to a numeric vector converts all the elements in the vector to character.
- Use `is.foo` to test for data type *foo*. Returns TRUE or FALSE  
Use `as.foo` to explicitly convert it.
- `is.numeric()`, `is.character()`, `is.vector()`,  
`is.matrix()`, `is.data.frame()`  
`as.numeric()`, `as.character()`, `as.vector()`,  
`as.matrix()`, `as.data.frame()`





# Lecture 4: Graphs

---



# Outline

---

- Basic Graphs
- Trellis
- Scatter plots



# Basic Graphs

---

- One of the main reasons data analysts turn to R is for its strong graphic capabilities.



# Creating a Graph

---

- In R, graphs are typically created interactively.

# Creating a Graph

```
attach(mtcars)
```

```
plot(wt, mpg)
```

```
abline(lm(mpg~wt))
```

```
title("Regression of MPG on Weight")
```

- The `plot( )` function opens a graph window and plots weight vs. miles per gallon. The next line of code adds a regression line to this graph. The final line adds a title.



# Creating a Graph

---

## Saving Graphs

| Function                                 | Output to        |
|------------------------------------------|------------------|
| <code>pdf("mygraph.pdf")</code>          | pdf file         |
| <code>win.metafile("mygraph.wmf")</code> | windows metafile |
| <code>png("mygraph.png")</code>          | png file         |
| <code>jpeg("mygraph.jpg")</code>         | jpeg file        |
| <code>bmp("mygraph.bmp")</code>          | bmp file         |
| <code>postscript("mygraph.ps")</code>    | postscript file  |



# Graphical Parameters

---

- You can customize many features of your graphs (fonts, colors, axes, titles) through graphic options.
- One way is to specify these options in through the `par( )` function. If you set parameter values here, the changes will be in effect for the rest of the session or until you change them again. The format is  
`par(optionname=value, optionname=value, ...)`
- `# Set a graphical parameter using par()`

```
par() # view current settings
opar <- par() # make a copy of current settings
par(col.lab="red") # red x and y labels
hist(mtcars$mpg) # create a plot with these new
settings
par(opar) # restore original settings
```



# Graphical Parameters

---

- A second way to specify graphical parameters is by providing the *optionname=value* pairs directly to a high level plotting function. In this case, the options are only in effect for that specific graph.
- # Set a graphical parameter within the plotting function  
`hist(mtcars$mpg, col.lab="red")`
- See the help for a specific high level plotting function (e.g. `plot`, `hist`, `boxplot`) to determine which graphical parameters can be set this way.
- The remainder of this section describes some of the more important graphical parameters that you can set.



# Graphical Parameters

---

- **Text and Symbol Size**
- The following options can be used to control text and symbol size in graphs.

| option          | description                                                                                                                                                      |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>cex</b>      | number indicating the amount by which plotting text and symbols should be scaled relative to the default. 1=default, 1.5 is 50% larger, 0.5 is 50% smaller, etc. |
| <b>cex.axis</b> | magnification of axis annotation relative to cex                                                                                                                 |
| <b>cex.lab</b>  | magnification of x and y labels relative to cex                                                                                                                  |
| <b>cex.main</b> | magnification of titles relative to cex                                                                                                                          |
| <b>cex.sub</b>  | magnification of subtitles relative to cex                                                                                                                       |

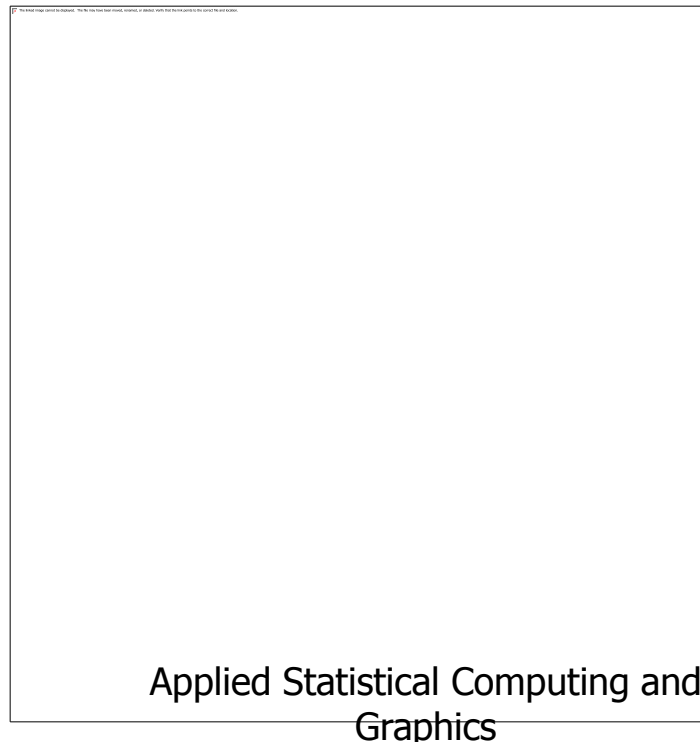




# Graphical Parameters

---

- **PLOTTING SYMBOLS**
- Use the **pch=** option to specify symbols to use when plotting points. For symbols 21 through 25, specify border color (**col=**) and fill color (**bg=**).





# Graphical Parameters

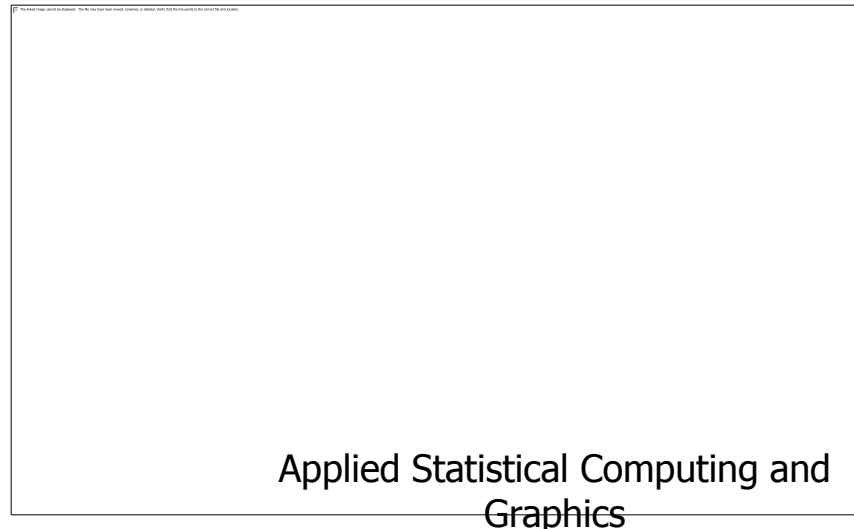
---

- **LINES**
- **You can change lines using the following options. This is particularly useful for reference lines, axes, and fit lines.**

| option | description |
|--------|-------------|
|--------|-------------|

|            |                                 |
|------------|---------------------------------|
| <b>lty</b> | line type. see the chart below. |
|------------|---------------------------------|

|            |                                                                     |
|------------|---------------------------------------------------------------------|
| <b>lwd</b> | line width relative to the default (default=1). 2 is twice as wide. |
|------------|---------------------------------------------------------------------|





# Graphical Parameters

---

- **COLORS**
- Options that specify colors include the following.

| option          | description                                                                                      |
|-----------------|--------------------------------------------------------------------------------------------------|
| <b>col</b>      | Default plotting color. Some functions (e.g. lines) accept a vector of values that are recycled. |
| <b>col.axis</b> | color for axis annotation                                                                        |
| <b>col.lab</b>  | color for x and y labels                                                                         |
| <b>col.main</b> | color for titles                                                                                 |
| <b>col.sub</b>  | color for subtitles                                                                              |
| <b>fg</b>       | plot foreground color (axes, boxes - also sets col= to same)                                     |
| <b>bg</b>       | plot background color                                                                            |



# Graphical Parameters

---

- You can specify colors in R by index, name, hexadecimal, or RGB. For example `col=1`, `col="white"`, and `col="#FFFFFF"` are equivalent.
- The following chart was produced with code developed by Earl F. Glynn. See his [Color Chart](#) for all the details you would ever need about using colors in R. There are wonderful color schemes at [graphviz](#).



# Graphical Parameters



# Graphical Parameters

---

- **fonts**
- **You can easily set font size and style, but font family is a bit more complicated.**

| option           | description                                                                                                          |
|------------------|----------------------------------------------------------------------------------------------------------------------|
| <b>font</b>      | Integer specifying font to use for text.<br>1=plain, 2=bold, 3=italic, 4=bold italic, 5=symbol                       |
| <b>font.axis</b> | font for axis annotation                                                                                             |
| <b>font.lab</b>  | font for x and y labels                                                                                              |
| <b>font.main</b> | font for titles                                                                                                      |
| <b>font.sub</b>  | font for subtitles                                                                                                   |
| <b>ps</b>        | font point size (roughly 1/72 inch)<br>text size=ps*cex                                                              |
| <b>family</b>    | font family for drawing text. Standard values are "serif", "sans", "mono", "symbol".<br>Mapping is device dependent. |



# Graphical Parameters

---

- Axes and Text
- Many high level plotting functions (plot, hist, boxplot, etc.) allow you to include axis and text options (as well as other graphical paramters). For example
- # Specify axis options within plot()  
`plot(x, y, main="title", sub="subtitle",  
      xlab="X-axis label", ylab="y-axis label",  
      xlim=c(xmin, xmax), ylim=c(ymin, ymax))`
- For finer control or for modularization, you can use the functions described below.



# Graphical Parameters

---

- Titles
- Use the `title( )` function to add labels to a plot.
- `title(main="main title", sub="sub-title",  
xlab="x-axis label", ylab="y-axis label")`
- Many other graphical parameters (such as text size, font, rotation, and color) can also be specified in the `title( )` function.
- ```
# Add a red title and a blue subtitle. Make x and y  
# labels 25% smaller than the default and green.  
title(main="My Title", col.main="red",  
sub="My Sub-title", col.sub="blue",  
xlab="My X label", ylab="My Y label",  
col.lab="green", cex.lab=0.75)
```




Graphical Parameters

If you are going to create a custom axis, you should suppress the axis automatically generated by your high level plotting function. The option `axes=FALSE` suppresses both x and y axes. `xaxt="n"` and `yaxt="n"` suppress the x and y axis respectively. Here is a (somewhat overblown) example.



Graphical Parameters

Axes

You can create custom axes using the **axis()** function.
`axis(side, at=, labels=, pos=, lty=, col=, las=, tck=, ...)`
where

option	description
side	an integer indicating the side of the graph to draw the axis (1=bottom, 2=left, 3=top, 4=right)
at	a numeric vector indicating where tic marks should be drawn
labels	a character vector of labels to be placed at the tickmarks (if NULL, the <i>at</i> values will be used)
pos	the coordinate at which the axis line is to be drawn. (i.e., the value on the other axis where it crosses)
lty	line type
col	the line and tick mark color
las	labels are parallel (=0) or perpendicular(=2) to axis
tck	length of tick mark as fraction of plotting region (negative number is outside graph, positive number is inside, 0 suppresses ticks, 1 creates gridlines) default is -0.01
(...)	other graphical parameters



Graphical Parameters

A Silly Axis Example

```
# specify the data
x <- c(1:10); y <- x; z <- 10/x

# create extra margin room on the right for an axis
par(mar=c(5, 4, 4, 8) + 0.1)

# plot x vs. y
plot(x, y, type="b", pch=21, col="red",
     yaxt="n", lty=3, xlab="", ylab="")

# add x vs. 1/x
lines(x, z, type="b", pch=22, col="blue", lty=2)

# draw an axis on the left
axis(2, at=x, labels=x, col.axis="red", las=2)

# draw an axis on the right, with smaller text and ticks
axis(4, at=z, labels=round(z, digits=2),
     col.axis="blue", las=2, cex.axis=0.7, tck=-.01)

# add a title for the right axis
mtext("y=1/x", side=4, line=3, cex.lab=1, las=2, col="blue")

# add a main title and bottom and left axis labels
title("An Example of Creative Axes", xlab="X values",
     ylab="Y=X")
```



Graphical Parameters

Reference Lines

Add reference lines to a graph using the `abline()` function.

`abline(h=yvalues, v=xvalues)`

Other graphical parameters (such as line type, color, and width) can also be specified in the `abline()` function.

add solid horizontal lines at $y=1,5,7$

`abline(h=c(1,5,7))`

add dashed blue vertical lines at $x = 1,3,5,7,9$

`abline(v=seq(1,10,2),lty=2,col="blue")`

Note: You can also use the `grid()` function to add reference lines.



Graphical Parameters

- **Legend**
- Add a legend with the **legend()** function.
- `legend(location, title, legend, ...)`
- Common options are described below.

option	description
location	There are several ways to indicate the location of the legend. You can give an x,y coordinate for the upper left hand corner of the legend. You can use locator(1) , in which case you use the mouse to indicate the location of the legend. You can also use the keywords "bottom", "bottomleft", "left", "topleft", "top", "topright", "right", "bottomright", or "center". If you use a keyword, you may want to use inset= to specify an amount to move the legend into the graph (as fraction of plot region).
title	A character string for the legend title (optional)
legend	A character vector with the labels
...	Other options. If the legend labels colored lines, specify col= and a vector of colors. If the legend labels point symbols, specify pch= and a vector of point symbols. If the legend labels line width or line style, use lwd= or lty= and a vector of widths or styles. To create colored boxes for the legend (common in bar, box, or pie charts), use fill= and a vector of colors.



Graphical Parameters

- ```
Legend Example
attach(mtcars)
boxplot(mpg~cyl, main="Milage by Car Weight",
 yaxt="n", xlab="Milage", horizontal=TRUE,
 col=terrain.colors(3))
legend("topright", inset=.05, title="Number of Cylinders",
 c("4","6","8"), fill=terrain.colors(3), horiz=TRUE)
```



# Graphical Parameters

---

- **Combining Plots**
- **R** makes it easy to combine multiple plots into one overall graph, using either the **par( )** or **layout( )** function.
- With the **par( )** function, you can include the option **mfrow=c(nrows, ncols)** to create a matrix of *nrows*  $\times$  *ncols* plots that are filled in by row. **mfcow=c(nrows, ncols)** fills in the matrix by columns.
- # 4 figures arranged in 2 rows and 2 columns  
attach(mtcars)  
par(mfrow=c(2,2))  
plot(wt,mpg, main="Scatterplot of wt vs. mpg")  
plot(wt,disp, main="Scatterplot of wt vs disp")  
hist(wt, main="Histogram of wt")  
boxplot(wt, main="Boxplot of wt")



# Graphical Parameters

---

- **3 figures arranged in 3 rows and 1 column**  
`attach(mtcars)`  
`par(mfrow=c(3,1))`  
`hist(wt)`  
`hist(mpg)`  
`hist(dis)`





# Graphical Parameters

---

- The **layout( )** function has the form **layout(*mat*)** where *mat* is a matrix object specifying the location of the N figures to plot.
- # One figure in row 1 and two figures in row 2  
attach(mtcars)  
layout(matrix(c(1,1,2,3), 2, 2, byrow = TRUE))  
hist(wt)  
hist(mpg)  
hist(dis)



# Graphical Parameters

---

- Optionally, you can include `widths=` and `heights=` options in the **`layout()`** function to control the size of each figure more precisely. These options have the form  
**`widths=`** a vector of values for the widths of columns  
**`heights=`** a vector of values for the heights of rows.
- Relative widths are specified with numeric values. Absolute widths (in centimetres) are specified with the **`lcm()`** function.
- ```
# One figure in row 1 and two figures in row 2
# row 1 is 1/3 the height of row 2
# column 2 is 1/4 the width of the column 1
attach(mtcars)
layout(matrix(c(1,1,2,3), 2, 2, byrow = TRUE),
        widths=c(3,1), heights=c(1,2))
hist(wt)
hist(mpg)
hist(disp)
```



Graphical Parameters

- **creating a figure arrangement with fine control**
- In the following example, two box plots are added to scatterplot to create an enhanced graph.
- ```
Add boxplots to a scatterplot
par(fig=c(0,0.8,0,0.8), new=TRUE)
plot(mtcars$wt, mtcars$mpg, xlab="Miles Per Gallon",
 ylab="Car Weight")
par(fig=c(0,0.8,0.55,1), new=TRUE)
boxplot(mtcars$wt, horizontal=TRUE, axes=FALSE)
par(fig=c(0.65,1,0,0.8),new=TRUE)
boxplot(mtcars$mpg, axes=FALSE)
mtext("Enhanced Scatterplot", side=3, outer=TRUE, line=-3)
```



# Graphical Parameters

---

- To understand this graph, think of the full graph area as going from (0,0) in the lower left corner to (1,1) in the upper right corner. The format of the `fig=` parameter is a numerical vector of the form `c(x1, x2, y1, y2)`. The first `fig=` sets up the scatterplot going from 0 to 0.8 on the x axis and 0 to 0.8 on the y axis. The top boxplot goes from 0 to 0.8 on the x axis and 0.55 to 1 on the y axis. I chose 0.55 rather than 0.8 so that the top figure will be pulled closer to the scatter plot. The right hand boxplot goes from 0.65 to 1 on the x axis and 0 to 0.8 on the y axis. Again, I chose a value to pull the right hand boxplot closer to the scatterplot. You have to experiment to get it just right.
- `fig=` starts a new plot, so to add to an existing plot use `new=TRUE`.
- You can use this to combine several plots in any arrangement into one graph.



# Trellis Graphs

---

Trellis graphs are available through the **lattice** package. A trellis graph displays a variable or the relationship between variables, conditioned on one or more other variables. Trellis graphs are available for a wide variety of plot types.

- The typical format is
- `graph_type(formula, data=)`
- where ***graph\_type*** is selected from the listed below. ***formula*** specifies the variable(s) to display and any conditioning variables. For example `~x | A` means display numeric variable `x` for each level of factor `A`. `y~x | A*B` means display the relationship between numeric variables `y` and `x` separately for every combination of factor `A` and `B` levels. `~x` means display numeric variable `x` alone.



# Trellis Graphs

| graph_type  | description               | formula examples         |
|-------------|---------------------------|--------------------------|
| barchart    | bar chart                 | $x \sim A$ or $A \sim x$ |
| bwplot      | boxplot                   | $x \sim A$ or $A \sim x$ |
| cloud       | 3D scatterplot            | $z \sim x * y   A$       |
| contourplot | 3D contour plot           | $z \sim x * y$           |
| densityplot | kernal density plot       | $\sim x   A * B$         |
| dotplot     | dotplot                   | $\sim x   A$             |
| histogram   | histogram                 | $\sim x$                 |
| levelplot   | 3D level plot             | $z \sim y * x$           |
| parallel    | parallel coordinates plot | dataframe                |
| splom       | scatterplot matrix        | dataframe                |
| stripplot   | strip plots               | $A \sim x$ or $x \sim A$ |
| xyplot      | scatterplot               | $y \sim x   A$           |
| wireframe   | 3D wireframe graph        | $z \sim y * x$           |



# Trellis Examples

---

```
library(lattice)
attach(mtcars)

create factors with value labels
gear.f<-factor(gear,levels=c(3,4,5),
 labels=c("3gears","4gears","5gears"))
cyl.f <-factor(cyl,levels=c(4,6,8),
 labels=c("4cyl","6cyl","8cyl"))

kernel density plot
densityplot(~mpg,
 main="Density Plot",
 xlab="Miles per Gallon")

kernel density plots by factor level
densityplot(~mpg|cyl.f,
 main="Density Plot by Number of Cylinders",
 xlab="Miles per Gallon")
```



# Trellis Examples

---

# boxplots for each combination of two factors

```
bwplot(cyl.f~mpg|gear.f,
 ylab="Cylinders", xlab="Miles per Gallon",
 main="Mileage by Cylinders and Gears",
 layout=(c(1,3)))
```

# scatterplots for each combination of two factors

```
xyplot(mpg~wt|cyl.f*gear.f,
 main="Scatterplots by Cylinders and Gears",
 ylab="Miles per Gallon", xlab="Car Weight")
```

# 3d scatterplot by factor level

```
cloud(mpg~wt*qsec|cyl.f,
 main="3D Scatterplot by Cylinders")
```

# dotplot for each combination of two factors

```
dotplot(cyl.f~mpg|gear.f,
 main="Dotplot Plot by Number of Gears and Cylinders",
 xlab="Miles Per Gallon")
```

# scatterplot matrix

```
splom(mtcars[c(1,3,4,5,6)],
 main="MTCARS Data")
```





# Customizing Trellis Graphs

- Unlike other **R** graphs, the trellis graphs described here are not effected by many of the options set in the **par( )** function. To view the options that can be changed, look at **help(xyplot)**. It is frequently easiest to set these options within the high level plotting functions described above. Additionally, you can write functions that modify the rendering of panels. Here is an example.
- # Customized Trellis Example

```
library(lattice)
panel.smoother <- function(x, y) {
 panel.xyplot(x, y) # show points
 panel.loess(x, y) # show smoothed line
}
attach(mtcars)
hp <- cut(hp,3) # divide horse power into three bands
xyplot(mpg~wt|hp, scales=list(cex=.8, col="red"),
 panel=panel.smoother,
 xlab="Weight", ylab="Miles per Gallon",
 main="MGP vs Weight by Horse Power")
```



# Scatterplot Matrices

---

There are at least 4 useful functions for creating scatterplot matrices.

# Basic Scatterplot Matrix

```
pairs(~mpg+disp+drat+wt,data=mtcars,
 main="Simple Scatterplot Matrix")
```



# Scatterplot Matrices

---

```
library(car)
scatterplot.matrix(~mpg+disp+drat+wt|cyl,
data=mtcars,
main="Three Cylinder Options")
```

# High Density Scatterplots

- When there are many data points and significant overlap, scatterplots become less useful. There are several approaches that be used when this occurs. The **hexbin(x, y)** function in the **hexbin** package provides bivariate binning into hexagonal cells (it looks better than it sounds).
- ```
library(hexbin)  
x <- rnorm(1000)  
y <- rnorm(1000)  
bin<-hexbin(x, y, xbins=50)  
plot(bin, main="Hexagonal Binning")
```



Lecture 4: Statistics



Outline

- T-test
- Non-parametric tests
- Exact tests
- Multiple Linear Regression
 - Diagnostics
- Anova
 - Diagnostics
- Multiple Comparisons
- Frequency and Cross tabulations
- Power Analysis



T test

- The **t.test()** function produces a variety of t-tests. Unlike most statistical packages, the default assumes unequal variance and applies the Welch df modification.
independent 2-group t-test
`t.test(y~x)` # where y is numeric and x is a binary factor
- # independent 2-group t-test
`t.test(y1,y2)` # where y1 and y2 are numeric
- # paired t-test
`t.test(y1,y2,paired=TRUE)` # where y1 & y2 are numeric
- # one sample t-test
`t.test(y,mu=3)` # $H_0: \mu=3$

Nonparametric Tests of Group Differences

- **R** provides functions for carrying out Mann-Whitney U, Wilcoxon Signed Rank, Kruskal Wallis, and Friedman tests.
independent 2-group Mann-Whitney U Test
`wilcox.test(y~A)`
where y is numeric and A is A binary factor
- # independent 2-group Mann-Whitney U Test
`wilcox.test(y,x)` # where y and x are numeric
- # dependent 2-group Wilcoxon Signed Rank Test
`wilcox.test(y1,y2,paired=TRUE)` # where y1 and y2 are numeric
- # Kruskal Wallis Test One Way Anova by Ranks
`kruskal.test(y~A)` # where y1 is numeric and A is a factor
- # Randomized Block Design - Friedman Test
`friedman.test(y~A|B)`
where y are the data values, A is a grouping factor
and B is a blocking factor



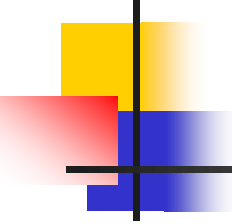
Resampling Statistics

- The [coin](#) package provides the ability to perform a wide variety of re-randomization or permutation based statistical tests. These tests do not assume random sampling from well-defined populations. They can be a reasonable alternative to classical procedures when test assumptions can not be met. See [coin: A Computational Framework for Conditional Inference](#) for details.
- In the examples below, **lower case** letters represent numerical variables and **upper case** letters represent categorical [factors](#). Monte-Carlo simulation are available for all tests. Exact tests are available for 2 group procedures.

Independent Two- and K-Sample Location Tests

- `# Exact Wilcoxon Mann Whitney Rank Sum Test`
`# where y is numeric and A is a binary factor`
`library(coin)`
`wilcox_test(y~A, data=mydata,`
`distribution="exact")`
- `# One-Way Permutation Test based on 9999`
`Monte-Carlo`
`# resamplings. y is numeric and A is a`
`categorical factor`
`library(coin)`
`oneway_test(y~A, data=mydata,`
`distribution=approximate(B=9999))`

Symmetry of a response for repeated measurements



```
# Exact Wilcoxon Signed Rank Test
# where y1 and y2 are repeated measures
library(coin)
wilcoxsign_test(y1~y2, data=mydata,
distribution="exact")

■ # Freidman Test based on 9999 Monte-Carlo
  resamplings.
  # y is numeric, A is a grouping factor, and B is a
  # blocking factor.
  library(coin)
  friedman_test(y~A|B, data=mydata,
    distribution=approximate(B=9999))
```

Independence of Two Numeric Variables

- `# Spearman Test of Independence based on 9999 Monte-Carlo
resamplings. x and y are numeric variables.
library(coin)
spearman_test(y~x, data=mydata,
distribution=approximate(B=9999))`

Independence in Contingency Tables

- # Independence in 2-way Contingency Table based on
9999 Monte-Carlo resamplings. A and B are factors.
`library(coin)`
`chisq_test(A~B, data=mydata,`
`distribution=approximate(B=9999))`
- # Cochran-Mantel-Haenzsel Test of 3-way Contingency
Table
based on 9999 Monte-Carlo resamplings. A, B, are
factors
and C is a stratifying factor.
`library(coin)`
`mh_test(A~B|C, data=mydata,`
`distribution=approximate(B=9999))`
- # Linear by Linear Association Test based on 9999
Monte-Carlo resamplings. A and B are ordered factors.
`library(coin)`
`lbl_test(A~B, data=mydata,`
`distribution=approximate(B=9999))`

Applied Statistical Computing and

Graphics

Independence in Contingency Tables

- # Independence in 2-way Contingency Table based on
9999 Monte-Carlo resamplings. A and B are factors.
`library(coin)`
`chisq_test(A~B, data=mydata,`
`distribution=approximate(B=9999))`
- # Cochran-Mantel-Haenszel Test of 3-way Contingency
Table
based on 9999 Monte-Carlo resamplings. A, B, are
factors
and C is a stratifying factor.
`library(coin)`
`mh_test(A~B|C, data=mydata,`
`distribution=approximate(B=9999))`
- # Linear by Linear Association Test based on 9999
Monte-Carlo resamplings. A and B are ordered factors.
`library(coin)`
`lbl_test(A~B, data=mydata,`
`distribution=approximate(B=9999))`



Frequencies and Crosstabs

- This section describes the creation of frequency and contingency tables from categorical variables, along with tests of independence, measures of association, and methods for graphically displaying results.
- **Generating Frequency Tables**
- **R** provides many methods for creating frequency and contingency tables. Three are described below. In the following examples, assume that A, B, and C represent categorical variables.



Frequencies and Crosstabs

- **table**
- You can generate frequency tables using the **table()** function, tables of proportions using the **prop.table()** function, and marginal frequencies using **margin.table()**.
- # 2-Way Frequency Table
attach(mydata)
mytable <- table(A,B) # A will be rows, B will be columns
mytable # print table

margin.table(mytable, 1) # A frequencies (summed over B)
margin.table(mytable, 2) # B frequencies (summed over A)

prop.table(mytable) # cell percentages
prop.table(mytable, 1) # row percentages
prop.table(mytable, 2) # column percentages



Using with() and by()

There are two functions that can help write simpler and more efficient code.

With

The with() function applies an expression to a dataset. It is similar to DATA= in SAS.

```
# with(data, expression)
# example applying a t-test to dataframe mydata
with(mtcars, t.test(mpg, hp))
```

By

The by() function applies a function to each level of a factor or factors. It is similar to BY processing in SAS.

```
# by(data, factorlist, function)
# example apply a t-test separately for men and women
by(mydata, gender, t.test(y1, y2))
```



Multiple (Linear) Regression

- **R provides comprehensive support for multiple linear regression. The topics below are provided in order of increasing complexity.**
- **Fitting the Model**
- # Multiple Linear Regression Example
`fit <- lm(y ~ x1 + x2 + x3, data=mydata)`
`summary(fit)` # show results
- # Other useful functions
`coefficients(fit)` # model coefficients
`confint(fit, level=0.95)` # CIs for model parameters
`fitted(fit)` # predicted values
`residuals(fit)` # residuals
`anova(fit)` # anova table
`vcov(fit)` # covariance matrix for model parameters
`influence(fit)` # regression diagnostics



Diagnostic Plots

- **Diagnostic plots provide checks for heteroscedasticity, normality, and influential observations.**
- # diagnostic plots
layout(matrix(c(1,2,3,4),2,2)) # optional 4
graphs/page
plot(fit)



Diagnostic Plots

- **Regression Diagnostics**

- An excellent review of regression diagnostics is provided in John Fox's aptly named [Overview of Regression Diagnostics](#).
- Dr. Fox's **car** package provides advanced utilities for regression modeling.
- ```
Assume that we are fitting a multiple linear regression
on the MTCARS data
library(car)
fit <- lm(mpg~disp+hp+wt+drat, data=mtcars)
```
- This example is for **exposition only**. We will ignore the fact that this may not be a great way of modeling the this particular set of data!



# Outliers

---

- # Assessing Outliers  
outlier.test(fit) # Bonferonni p-value for  
most extreme obs  
qq.plot(fit, main="QQ Plot") #qq plot for  
studentized resid  
layout(matrix(c(1,2,3,4,5,6),2,3)) # optional  
layout  
leverage.plots(fit, ask=FALSE) # leverage  
plots



# Influential Observations

---

- ```
# Influential Observations
# added variable plots
av.plots(fit, one.page=TRUE, ask=FALSE)
# Cook's D plot
# identify D values > 4/(n-k-1)
cutoff <- 4/((nrow(mtcars)-length(fit$coefficients)-
2))
plot(fit, which=4, cook.levels=cutoff)
# Influence Plot
influencePlot(fit, main="Influence Plot",
sub="Circle size is proportional to Cook's Distance" )
```



Non-normality

- ```
Normality of Residuals
qq plot for studentized resid
qq.plot(fit, main="QQ Plot")
distribution of studentized residuals
library(MASS)
sresid <- studres(fit)
hist(sresid, freq=FALSE,
 main="Distribution of Studentized Residuals")
xfit<-seq(min(sresid),max(sresid),length=40)
yfit<-dnorm(xfit)
lines(xfit, yfit)
```



# Non-constant Error Variance

---

- `# Evaluate homoscedasticity`  
`# non-constant error variance test`  
`ncv.test(fit)`  
`# plot studentized residuals vs. fitted`  
`values`  
`spread.level.plot(fit)`





# Multi-collinearity

---

- # Evaluate Collinearity  
vif(fit) # variance inflation  
factors  
 $\sqrt{\text{vif(fit)}} > 2$  # problem?



# Nonlinearity

---

- # Evaluate Nonlinearity  
# component + residual plot  
`cr.plots(fit, one.page=TRUE,  
ask=FALSE)`  
# Ceres plots  
`ceres.plots(fit, one.page=TRUE,  
ask=FALSE)`



# Non-independence of Errors

---

- # Test for Autocorrelated Errors  
`durbin.watson(fit)`
- **Additional Diagnostic Help**
- The **`gvlma()`** function in the **`gvlma`** package, performs a global validation of linear model assumptions as well separate evaluations of skewness, kurtosis, and heteroscedasticity.
- # Global test of model assumptions  
`library(gvlma)`  
`gvmodel <- gvlma(fit)`  
`summary(gvmodel)`



# Comparing Models

---

- **You can compare nested models with the `anova( )` function. The following code provides a simultaneous test that `x3` and `x4` add to linear prediction above and beyond `x1` and `x2`.**
- `# compare models`  
`fit1 <- lm(y ~ x1 + x2 + x3 + x4, data=mydata)`  
`fit2 <- lm(y ~ x1 + x2)`  
`anova(fit1, fit2)`

# Cross Validation

- You can do **K-Fold cross-validation** using the `cv.lm( )` function in the **DAAG** package.
- ~~# K-fold cross-validation~~
- `library(DAAG)`  
`cv.lm(df=mydata, fit, m=3) # 3 fold cross-validation`
- **Sum the MSE for each fold, divide by the number of observations, and take the square root to get the cross-validated standard error of estimate.**
- **You can assess R2 shrinkage via K-fold cross-validation. Using the `crossval()` function from the **bootstrap** package, do the following:**
- `# Assessing R2 shrinkage using 10-Fold Cross-Validation`

```
fit <- lm(y~x1+x2+x3,data=mydata)
```

```
library(bootstrap)
define functions
theta.fit <- function(x,y){lsfit(x,y)}
theta.predict <- function(fit,x){cbind(1,x)%*%fit$coef}
```

```
matrix of predictors
X <- as.matrix(mydata[c("x1","x2","x3")])
vector of predicted values
y <- as.matrix(mydata[c("y")])
```

```
results <- crossval(X,y,theta.fit,theta.predict,ngroup=10)
cor(y, fit$fitted.values)**2 # raw R2
cor(y,results$cv.fit)**2 # cross-validated R2
```



# Variable Selection

---

- **Selecting a subset of predictor variables from a larger set (e.g., stepwise selection) is a controversial topic. You can perform stepwise selection (forward, backward, both) using the `stepAIC()` function from the MASS package. `stepAIC()` performs stepwise model selection by exact AIC.**
- **# Stepwise Regression**  
`library(MASS)`  
`fit <- lm(y~x1+x2+x3,data=mydata)`  
`step <- stepAIC(fit, direction="both")`  
`step$anova # display results`



# Variable Selection

---

- **Alternatively, you can perform all-subsets regression using the `leaps( )` function from the `leaps` package. In the following code `nbest` indicates the number of subsets of each size to report. Here, the ten best models will be reported for each subset size (1 predictor, 2 predictors, etc.).**
- **# All Subsets Regression**  

```
library(leaps)
attach(mydata)
leaps<-regsubsets(y~x1+x2+x3+x4,data=mydata,nbest=10)
view results
summary(leaps)
plot a table of models showing variables in each model.
models are ordered by the selection statistic.
plot(leaps,scale="r2")
plot statistic by subset size
library(car)
subsets(leaps, statistic="rsq")
```



# Variable Selection

---

- Other options for `plot( )` are `bic`, `Cp`, and `adjr2`. Other options for plotting with `subset( )` are `bic`, `cp`, `adjr2`, and `rss`.
- Relative Importance
- The **relaimpo** package provides measures of relative importance for each of the predictors in the model. See `help(calc.relimp)` for details on the four measures of relative importance provided.
- ```
# Calculate Relative Importance for Each Predictor
library(relaimpo)
calc.relimp(fit,type=c("lmg","last","first","pratt"),
  rela=TRUE)

# Bootstrap Measures of Relative Importance (1000 samples)
boot <- boot.relimp(fit, b = 1000, type = c("lmg",
  "last", "first", "pratt"), rank = TRUE,
  diff = TRUE, rela = TRUE)
booteval.relimp(boot) # print result
plot(booteval.relimp(boot,sort=TRUE)) # plot result
```




Nonlinear Regression

The `nl` package provides functions for nonlinear regression. See John Fox's [Nonlinear Regression and Nonlinear Least Squares](#) for an overview. Huet and colleagues' [Statistical Tools for Nonlinear Regression: A Practical Guide with S-PLUS and R Examples](#) is a valuable reference book.

- Robust Regression
- There are many functions in R to aid with robust regression. For example, you can perform robust regression with the `rlm()` function in the MASS package. John Fox's (who else?) [Robust Regression](#) provides a good starting overview. The UCLA Statistical Computing website has [Robust Regression Examples](#).
- The [robust](#) package provides a comprehensive library of robust methods, including regression. The [robustbase](#) package also provides basic robust statistics including model selection methods. And David Olive has provided an detailed online review of [Applied Robust Statistics](#) with sample R code.



Comparing Models

- You can compare nested models with the `anova()` function. The following code provides a simultaneous test that `x3` and `x4` add to linear prediction above and beyond `x1` and `x2`.
- ```
compare models
fit1 <- lm(y ~ x1 + x2 + x3 + x4, data=mydata)
fit2 <- lm(y ~ x1 + x2)
anova(fit1, fit2)
```
- **Cross Validation**
- You can do K-Fold cross-validation using the `cv.lm( )` function in the **DAAG** package.
- ```
# K-fold cross-validation
library(DAAG)
cv.lm(df=mydata, fit, m=3) # 3 fold cross-validation
```
- **Sum the MSE for each fold, divide by the number of observations, and take the square root to get the cross-validated standard error of estimate.**
- You can assess R^2 shrinkage via K-fold cross-validation. Using the `crossval()` function from the bootstrap package, do the following:
- ```
Assessing R2 shrinkage using 10-Fold Cross-Validation
fit <- lm(y~x1+x2+x3,data=mydata)
```



# ANOVA

---

- **If you have been analyzing ANOVA designs in traditional statistical packages, you are likely to find R's approach less coherent and user-friendly. A good online presentation on ANOVA in R is available from [Katholieke Universiteit Leuven](#).**



# ANOVA - Fit a Model

---

- **In the following examples lower case letters are numeric variables and upper case letters are factors.**
- # One Way Anova (Completely Randomized Design)  
`fit <- aov(y ~ A, data=mydataframe)`
- # Randomized Block Design (B is the blocking factor)  
`fit <- aov(y ~ A + B, data=mydataframe)`
- # Two Way Factorial Design  
`fit <- aov(y ~ A + B + A:B, data=mydataframe)`  
`fit <- aov(y ~ A*B, data=mydataframe) # same thing`
- # Analysis of Covariance  
`fit <- aov(y ~ A + x, data=mydataframe)`



# ANOVA - Fit a Model

---

- **For within subjects designs, the dataframe has to be rearranged so that each measurement on a subject is a separate observation. See R and Analysis of Variance.**
- # One Within Factor  
fit <-  
aov(y~A+Error(Subject/A),data=mydataframe)
- # Two Within Factors W1 W2, Two Between Factors B1 B2  
fit <-  
aov(y~(W1\*W2\*B1\*B2)+Error(Subject/(W1\*W2))+(B1\*B2),  
data=mydataframe)



## 2. Look at Diagnostic Plots

---

- **Diagnostic plots provide checks for heteroscedasticity, normality, and influential observations.**  
`layout(matrix(c(1,2,3,4),2,2))` # optional layout  
`plot(fit)` # diagnostic plots
- **For details on the evaluation of test requirements, see (M)ANOVA Assumptions.**

# ANOVA - 3. Evaluate Model Effects

- **WARNING:** R provides Type I sequential SS, not the default Type III marginal SS reported by SAS and SPSS. In a nonorthogonal design with more than one term on the right hand side of the equation order will matter (i.e., A+B and B+A will produce *different* results)! We will need use the `drop1()` function to produce the familiar Type III results. It will compare each term with the full model. Alternatively, we can use `anova(fit.model1, fit.model2)` to compare nested models directly.
- `summary(fit)` # display Type I ANOVA table  
`drop1(fit, ~., test="F")` # type III SS and F Tests



# Multiple Comparisons

---

- **You can get Tukey HSD tests using the function below. By default, it calculates post hoc comparisons on each factor in the model. You can specify specific factors as an option. Again, remember that results are based on Type I SS!**
- **# Tukey Honestly Significant Differences**  
`TukeyHSD(fit) # where fit comes from aov()`





# Visualizing Results

---

- Use **box plots** and **line plots** to visualize group differences. There are also two functions specifically designed for visualizing mean differences in ANOVA layouts. `interaction.plot()` in the base stats package produces plots for two-way interactions. `plotmeans()` in the **gplots** package produces mean plots for single factors, and includes confidence intervals.
- # Two-way Interaction Plot  

```
attach(mtcars)
gears <- factor(gears)
cyl <- factor(cyl)
interaction.plot(cyl, gear, mpg, type="b", col=c(1:3),
 leg.bty="o", leg.bg="beige", lwd=2, pch=c(18,24,22),
 xlab="Number of Cylinders",
 ylab="Mean Miles Per Gallon",
 main="Interaction Plot")
```
- # Plot Means with Error Bars  

```
library(gplots)
attach(mtcars)
cyl <- factor(cyl)
plotmeans(mpg~cyl,xlab="Number of Cylinders",
 ylab="Miles Per Gallon, main="Mean Plot\nwith 95% CI")
```



# Power Analysis

---

## Overview

Power analysis is an important aspect of experimental design. It allows us to determine the sample size required to detect an effect of a given size with a given degree of confidence. Conversely, it allows us to determine the probability of detecting an effect of a given size with a given level of confidence, under sample size constraints. If the probability is unacceptably low, we would be wise to alter or abandon the experiment.

The following **four quantities** have an intimate relationship:

1. sample size
2. effect size
3. significance level =  $P(\text{Type I error})$  = probability of finding an effect that is not there
4. power =  $1 - P(\text{Type II error})$  = probability of finding an effect that is there

Given any three, we can determine the fourth.



# Power Analysis in R

---

- The **pwr** package developed by Stéphane Champely, implements power analysis as outlined by **Cohen (!988)**. Some of the more important functions are listed below.

| function              | power calculations for                 |
|-----------------------|----------------------------------------|
| <b>pwr.2p.test</b>    | two proportions (equal n)              |
| <b>pwr.2p2n.test</b>  | two proportions (unequal n)            |
| <b>pwr.anova.test</b> | balanced one way ANOVA                 |
| <b>pwr.chisq.test</b> | chi-square test                        |
| <b>pwr.f2.test</b>    | general linear model                   |
| <b>pwr.p.test</b>     | proportion (one sample)                |
| <b>pwr.r.test</b>     | correlation                            |
| <b>pwr.t.test</b>     | t-tests (one sample, 2 sample, paired) |
| <b>pwr.t2n.test</b>   | t-test (two samples with unequal n)    |



# Power Analysis t-tests

---

For t-tests, use the following functions:

**`pwr.t.test(n = , d = , sig.level = , power = , type =  
c("two.sample", "one.sample", "paired"))`**

where *n* is the sample size, *d* is the effect size, and *type* indicates a two-sample t-test, one-sample t-test or paired t-test. If you have unequal sample sizes, use

**`pwr.t2n.test(n1 = , n2 = , d = , sig.level = , power = )`**

where *n1* and *n2* are the sample sizes.

For t-tests, the effect size is assessed as



Cohen suggests that *d* values of 0.2, 0.5, and 0.8 represent small, medium, and large effect sizes respectively.

You can specify `alternative="two.sided", "less", or "greater"` to indicate a two-tailed, or one-tailed test. A two tailed test is the default.



# Power Analysis - ANOVA

---

For a one-way analysis of variance use

**`pwr.anova.test(k = , n = , f = , sig.level = , power = )`**

where  $k$  is the number of groups and  $n$  is the common sample size in each group.

For a one-way ANOVA effect size is measured by  $f$  where



Cohen suggests that  $f$  values of 0.1, 0.25, and 0.4 represent small, medium, and large effect sizes respectively.

# Power Analysis

## CORRELATIONS



- For correlation coefficients use
- **pwr.r.test(n = , r = , sig.level = , power = )**
- where n is the sample size and r is the correlation. We use the population correlation coefficient as the effect size measure. Cohen suggests that r values of 0.1, 0.3, and 0.5 represent small, medium, and large effect sizes respectively.



# Power Analysis Linear Models

---

For linear models (e.g., multiple regression) use

**`pwr.f2.test(u = , v = , f2 = , sig.level = , power = )`**

where u and v are the numerator and denominator degrees of freedom. We use f2 as the effect size measure.



The first formula is appropriate when we are evaluating the impact of a set of predictors on an outcome. The second formula is appropriate when we are evaluating the impact of one set of predictors above and beyond a second set of predictors (or covariates). Cohen suggests f2 values of 0.02, 0.15, and 0.35 represent small, medium, and large effect sizes.

# Power Analysis

## Tests of Proportions

When comparing two proportions use

**`pwr.2p.test(h = , n = , sig.level = , power = )`**

where h is the effect size and n is the common sample size in each group.



Cohen suggests that h values of 0.2, 0.5, and 0.8 represent small, medium, and large effect sizes respectively.

For unequal n's use

**`pwr.2p2n.test(h = , n1 = , n2 = , sig.level = , power = )`**

To test a single proportion use

**`pwr.p.test(h = , n = , sig.level = , power = )`**

For both two sample and one sample proportion tests, you can specify `alternative="two.sided", "less", or "greater"` to indicate a two-tailed, or one-tailed test. A two-tailed test is the default.

Applied Statistical Computing and





# Power Analysis

---

## Chi-square Tests

For chi-square tests use

**`pwr.chisq.test(w = , N = , df = , sig.level = , power = )`**

where  $w$  is the effect size,  $N$  is the total sample size, and  $df$  is the degrees of freedom. The effect size  $w$  is defined as



Cohen suggests that  $w$  values of 0.1, 0.1, and 0.5 represent small, medium, and large effect sizes respectively.

# Power Analysis

## some Examples



---

```
library(pwr)
```

```
For a one-way ANOVA comparing 5 groups, calculate the
sample size needed in each group to obtain a power of
0.80, when the effect size is moderate (0.25) and a
significance level of 0.05 is employed.
```

```
pwr.anova.test(k=5,f=.25,sig.level=.05,power=.8)
```

```
What is the power of a one-tailed t-test, with a
significance level of 0.01, 25 people in each group,
and an effect size equal to 0.75?
```

```
pwr.t.test(n=25,d=0.75,sig.level=.01,alternative="greater")
```

```
Using a two-tailed test proportions, and assuming a
significance level of 0.01 and a common sample size of
30 for each proportion, what effect size can be detected
with a power of .75?
```

```
pwr.2p.test(n=30,sig.level=0.01,power=0.75)
```



# Power Analysis - examples

## Creating Power or Sample Size Plots

The functions in the **pwr** package can be used to generate power and sample size graphs.

# Plot sample size curves for detecting correlations of  
# various sizes.

```
library(pwr)
```

```
range of correlations
r <- seq(.1,.5,.01)
nr <- length(r)
```

```
power values
p <- seq(.4,.9,.1)
np <- length(p)
```

```
obtain sample sizes
samsize <- array(numeric(nr*np), dim=c(nr,np))
for (i in 1:np){
 for (j in 1:nr){
 result <- pwr.r.test(n = NULL, r = r[j],
 sig.level = .05, power = p[i],
 alternative = "two.sided")
 samsize[j,i] <- ceiling(result$n)
 }
}
```

Applied Statistical Computing and  
Graphics



# Power Analysis - examples

---

```
■ # set up graph
xrange <- range(r)
yrange <- round(range(samsize))
colors <- rainbow(length(p))
plot(xrange, yrange, type="n",
 xlab="Correlation Coefficient (r)",
 ylab="Sample Size (n)")

add power curves
for (i in 1:np){
 lines(r, samsize[,i], type="l", lwd=2, col=colors[i])
}

add annotation (grid lines, title, legend)
abline(v=0, h=seq(0,yrange[2],50), lty=2, col="grey89")
abline(h=0, v=seq(xrange[1],xrange[2],.02), lty=2,
 col="grey89")
title("Sample Size Estimation for Correlation Studies\n
 Sig=0.05 (Two-tailed)")
legend("topright", title="Power", as.character(p),
 fill=colors)
```



# Non-linear Modelling

---

- For dose-response modeling, one of the most common parametric approaches is to use a 3-parameter EMAX model



# Non-linear Modelling

---

For dose-response modeling, one of the most common parametric approaches is to use a 3-parameter EMAX model by fitting the dose response function  $g(D)$

$$g(D) = E(Y|D) = E_0 + \frac{E_{\max} D}{ED_{50} + D}$$

where  $E_0$  is the response  $Y$  at baseline (absence of dose),  $E_{\max}$  is the asymptotic maximum dose effect and  $ED_{50}$  is the concentration that produces 50% of the maximal effect or potency



# Non-linear Modelling

---

For dose-response modeling, one of the most common parametric approaches is to use a 3-parameter EMAX model by fitting the dose response function  $g(D)$

$$g(D) = E(Y|D) = E_0 + \frac{E_{\max} D}{ED_{50} + D}$$

where  $E_0$  is the response  $Y$  at baseline (absence of dose),  $E_{\max}$  is the asymptotic maximum dose effect and  $ED_{50}$  is the concentration that produces 50% of the maximal effect or potency



# Non-linear Modelling

---

- Expansion of equation above is achieved with the 4-parameter EMAX model for

$$g(D) = E(Y|D) = E_0 + \frac{E_{\max} D^{\lambda}}{ED_{50}^{\lambda} + D^{\lambda}}$$

where  $\lambda$  is the 4th parameter which is sometimes called the Hill(Holford and Sheiner[1]) parameter. The Hill parameter affects the shape of the curve and is in some cases very difficult to estimate.



# Non-linear Modelling

