Umer Zeeshan Ijaz

Introduction to R

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.

How to download?

- Google it using R or CRAN

(Comprehensive R Archive Network)

- http://www.r-project.org

R Overview

Start R by typing R at prompt

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 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 (=).

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 F T T
> x < 5
> T T T T F F F F F F
> x > 8 | x < 5
> T T T T F F F F T T
> x[c(T,T,T,T,F,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 hold 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

- During your R session you can also explicitly save the workspace image.

```
## 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

```
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 vingettes
    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

- 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 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
```

R Packages

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

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.

```
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
pdf("mygraph.pdf")	pdf file
win.metafile("mygraph.wmf")	windows metafile
png("mygraph.png")	png file
jpeg("mygraph.jpg")	jpeg file
bmp("mygraph.bmp")	bmp file
postscript("mygraph.ps")	postscript file

Redirecting Graphs

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

mtcars data

```
> mtcars
                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
>
```

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.

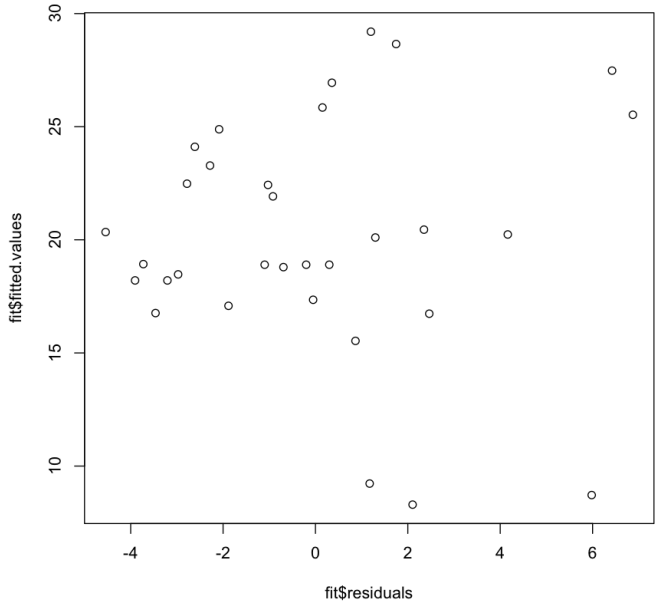
str(fit) # view the contents/structure of "fit“

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)
```

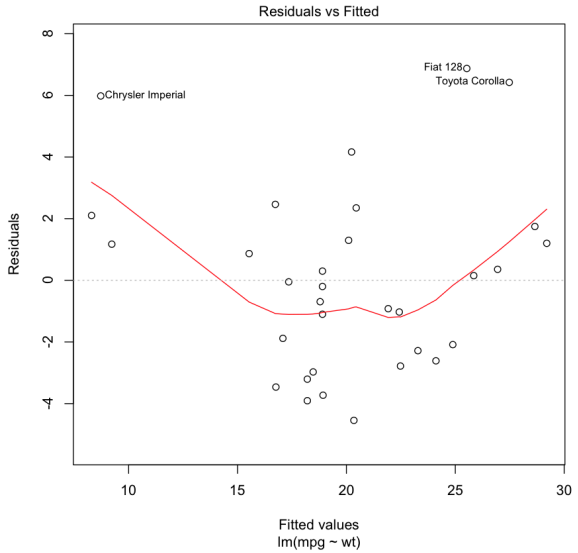
To see what a function returns, look at the **value** section of the online help for that function. Here we would look at **help(lm)**.

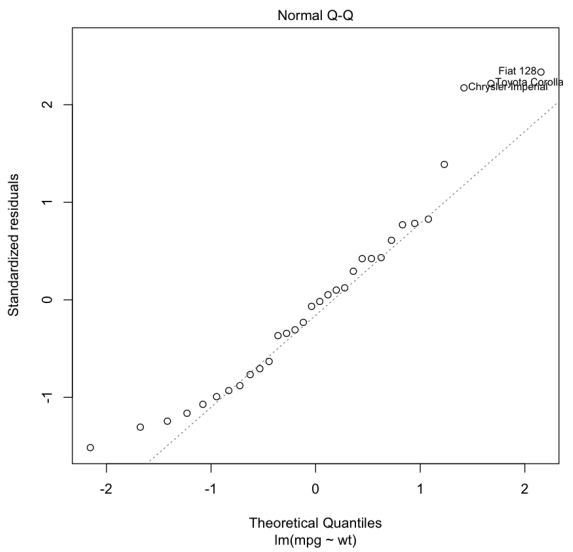


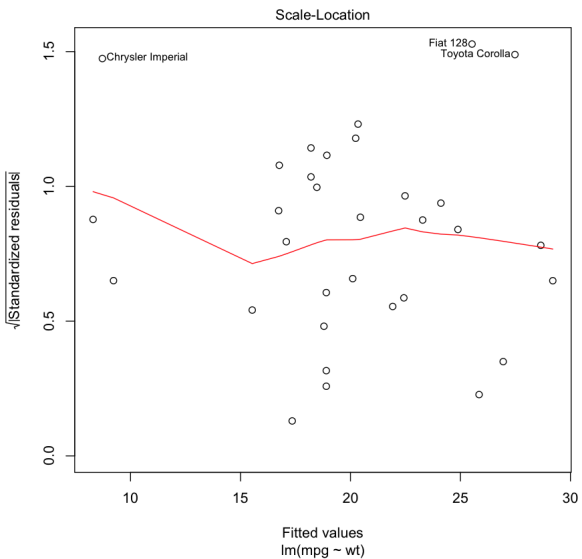
Reusing Results

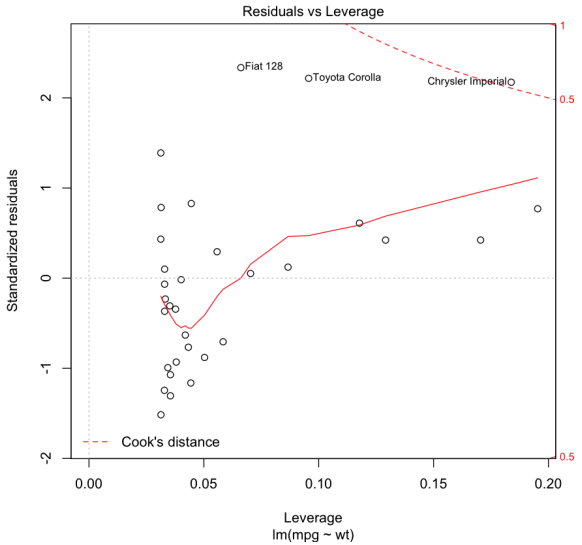
The results can also be used by a wide range of other functions.

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









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.

```
 > as.array(letters)
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"
>array(1:3, c(2,4)) # recycle 1:3 "2 2/3 times"
 #      [,1] [,2] [,3] [,4]
 #[1,]    1    3    2    1
 #[2,]    2    1    3    2
```

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 scaler
    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 newobject
fix(object)             # edit in place
```

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:/myexel.xls")
    mydata <- sqlFetch(channel, "mysheet")
    odbcClose(channel)
```

Keyboard Input

Usually you will obtain a dataframe by importing it **Excel**, 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 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")
```

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

```
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 lables.

```
# 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"))
```

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).

Testing for Missing Values

is.na(x) # returns TRUE of 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)
```

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"
```

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
  )
  ```
- transform(airquality, Ozone = -Ozone)
- transform(airquality, new = -Ozone, Temp = (Temp-32)/1.8)
- ```
  attach(airquality)
  transform(Ozone, logOzone = log(Ozone)) # marginally interesting ...
  detach(airquality)
  ```

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
^ or **	exponentiation
x %% y	modulus (x mod y) 5%%2 is 1
x %/% y	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)

Numeric Functions

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

Character Functions

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

Stat/Prob Functions

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

Function	Description

dnorm(*x*)	normal density function (by default m=0 sd=1)
	# plot standard normal curve
	x <- pretty(c(-3,3), 30)
	y <- dnorm(x)
	plot(x, y, type='l', xlab="Normal Deviate", ylab="Density", yaxs="i")

pnorm(*q*)	cumulative normal probability for q (area under the normal curve to the right of q) pnorm(1.96) is 0.975
qnorm(*p*)	normal quantile. value at the p percentile of normal distribution qnorm(.9) is 1.28 # 90th percentile
rnorm(*n*, m=0,sd=1)	n random normal deviates with mean m and standard deviation sd. #50 random normal variates with mean=50, sd=10 x <- rnorm(50, m=50, sd=10)

dbinom(*x*, *size*, *prob*)
pbinom(*q*, *size*, *prob*)
qbinom(*p*, *size*, *prob*)
rbinom(*n*, *size*, *prob*)
	binomial distribution where size is the sample size
	and prob is the probability of a heads (pi)
	# prob of 0 to 5 heads of fair coin out of 10 flips
	dbinom(0:5, 10, .5)
	# prob of 5 or less heads of fair coin out of 10 flips
	pbinom(5, 10, .5)

dpois(*x*, *lamda*)
ppois(*q*, *lamda*)
qpois(*p*, *lamda*)
rpois(*n*, *lamda*)
	poisson distribution with m=std=lamda
	#probability of 0,1, or 2 events with lamda=4
	dpois(0:2, 4)
	# probability of at least 3 events with lamda=4
	1- ppois(2,4)

dunif(*x*, min=0, max=1) **punif(*q*, min=0, max=1)** **qunif(*p*, min=0, max=1)** **runif(*n*, min=0, max=1)**	uniform distribution, follows the same pattern as the normal distribution above. #10 uniform random variates x <- runif(10)

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

Other Useful Functions

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

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)

Basic Graphs

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

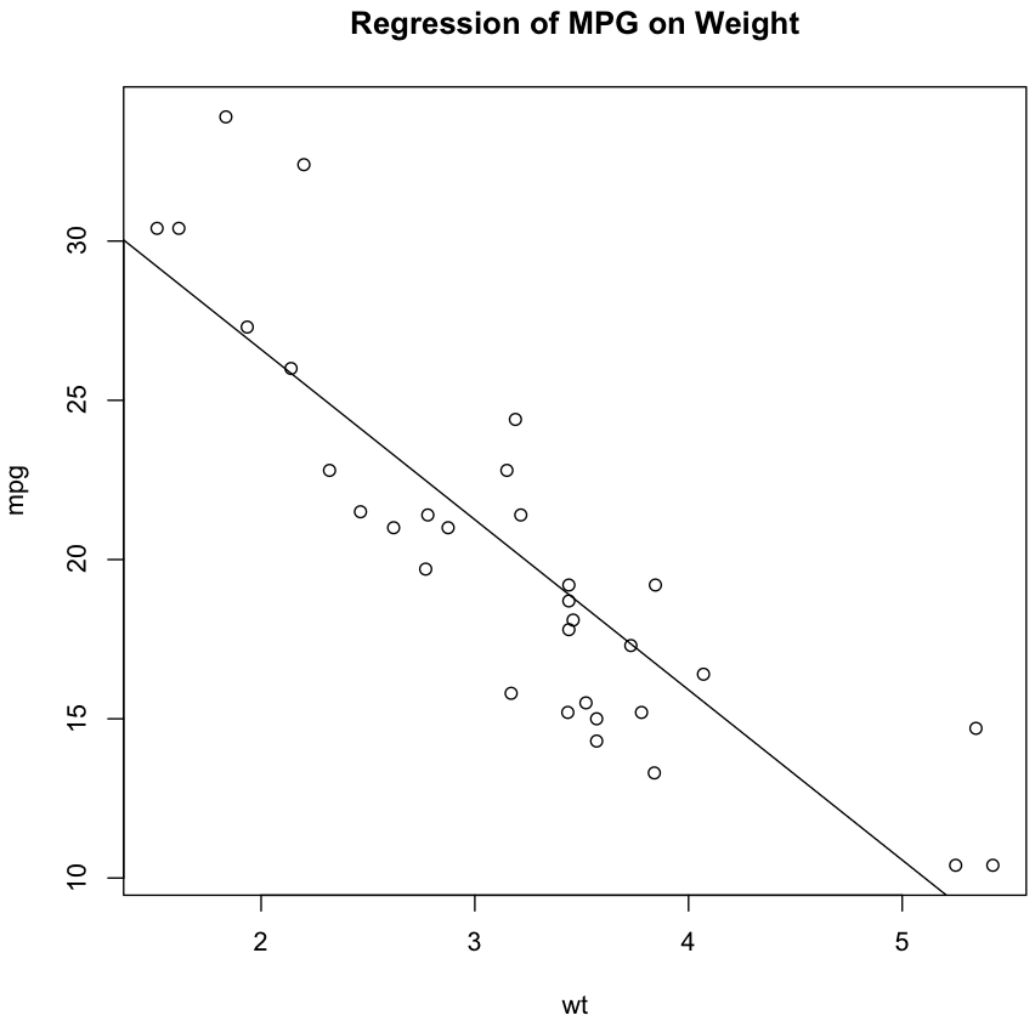
Creating a Graph

- 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



Creating a Graph

Saving Graphs

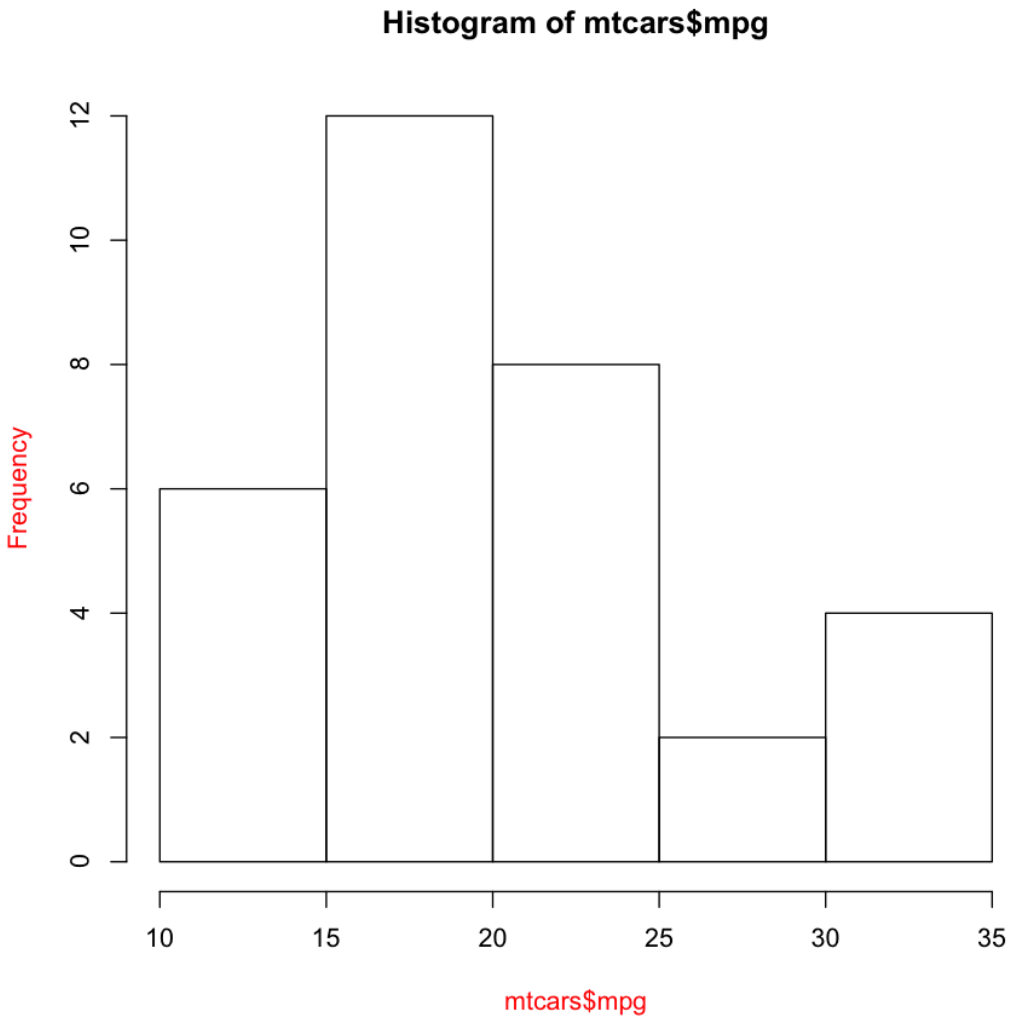
Function	Output to
pdf("mygraph.pdf")	pdf file
win.metafile("mygraph.wmf")	windows metafile
png("mygraph.png")	png file
jpeg("mygraph.jpg")	jpeg file
bmp("mygraph.bmp")	bmp file
postscript("mygraph.ps")	postscript file

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



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.

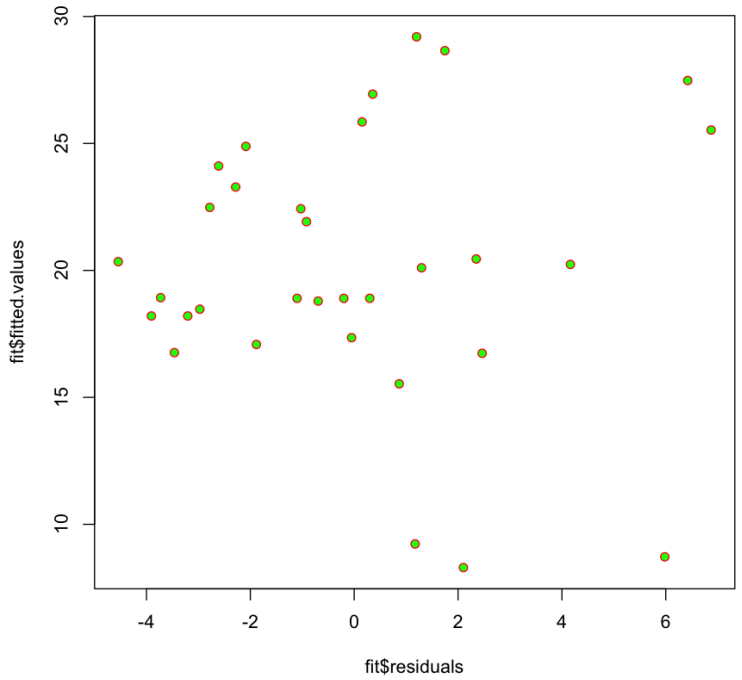
Graphical Parameters

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

option	description
cex	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.
cex.axis	magnification of axis annotation relative to cex
cex.lab	magnification of x and y labels relative to cex
cex.main	magnification of titles relative to cex
cex.sub	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=).
  plot(fit$residuals, fit$fitted.values, pch=21,col="red",bg="green")



Graphical Parameters

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

option	description
lty	line type. see the chart below.
lwd	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
col	Default plotting color. Some functions (e.g. lines) accept a vector of values that are recycled.
col.axis	color for axis annotation
col.lab	color for x and y labels
col.main	color for titles
col.sub	color for subtitles
fg	plot foreground color (axes, boxes - also sets col= to same)
bg	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

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

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

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 *at* 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="")

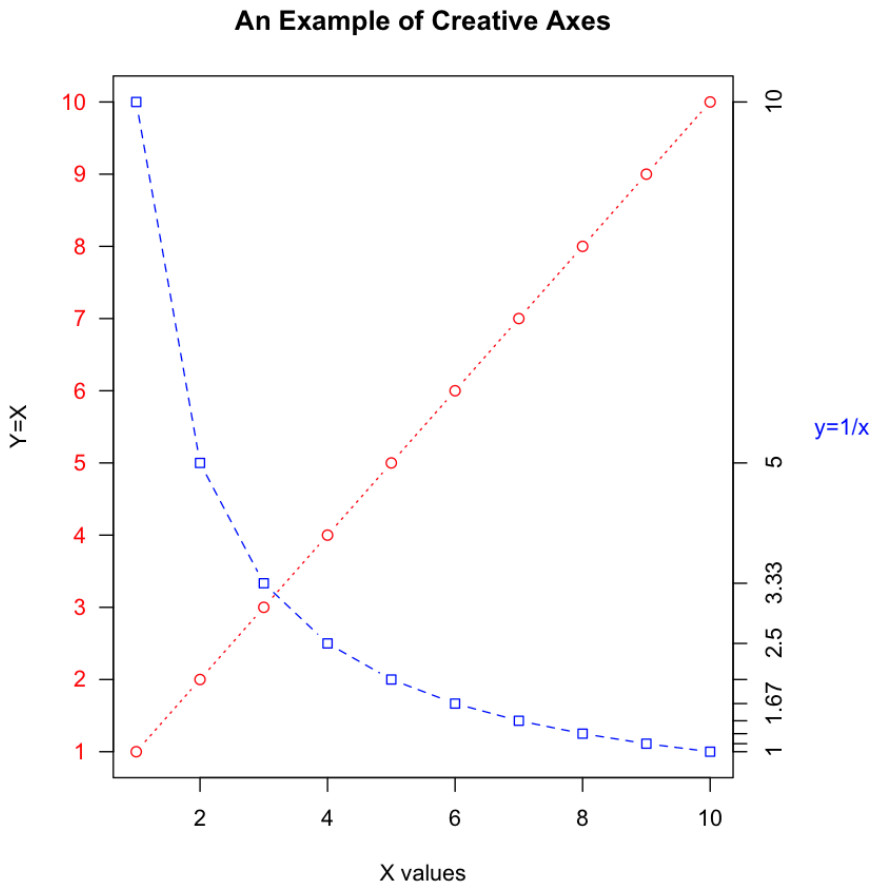
    # add x vs. 1/x
    lines(x, z, type="b", pch=22, col="blue", lty=2)

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

    # draw an axis on the right
    axis(4, at=z,labels=round(z,digits=2))

    # 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 verical 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.

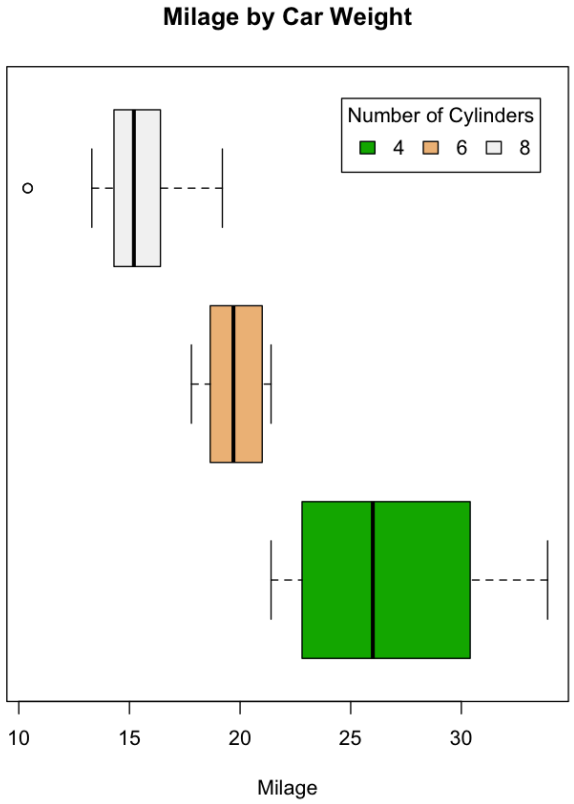
Graphical Parameters

- **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

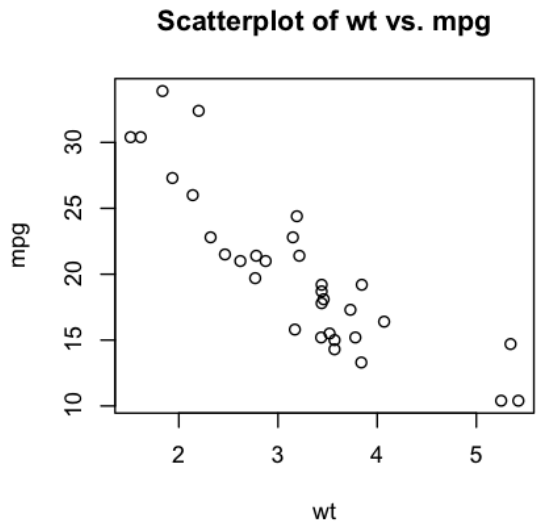
- # Legend Example
  attach(mtcars)
  boxplot(mpg~cyl, main="Milage by Car Weight",
    yaxt="n", xlab="Milage", horizontal=TRUE,
    col=terrain.colors(3))
  legend("topright", inset=.05, title="Number of Cylinders",
    c("4","6","8"), fill=terrain.colors(3), horiz=TRUE)

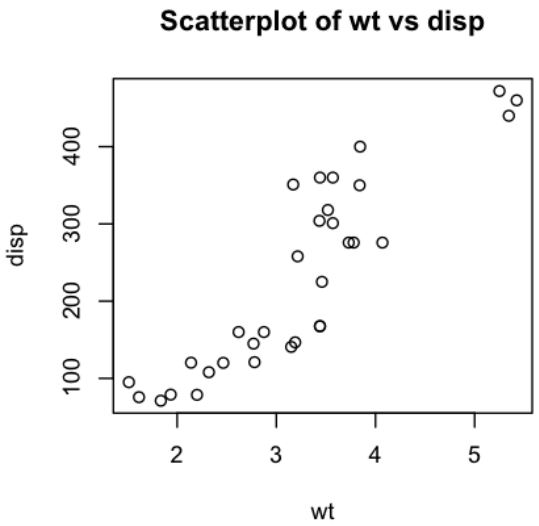


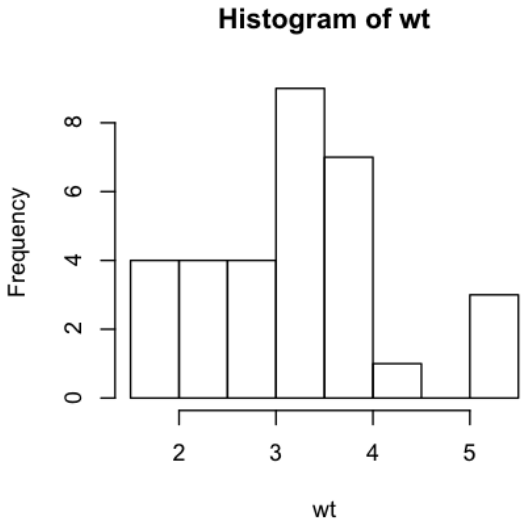
Graphical Parameters

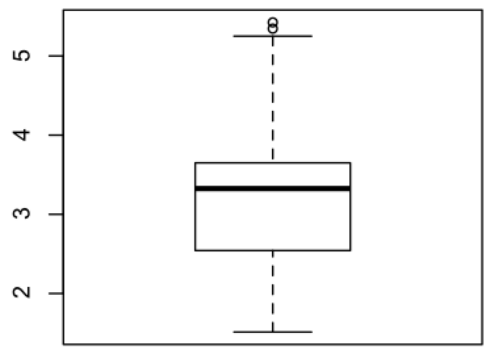
- **Combining Plots**
- **R** makes it easy to combine multiple plots into one overall graph, using either the
  par( ) or **layout( )** function.
- With the **par( )** function, you can include the option **mfrow=c(***nrows*, *ncols***)** to create a matrix of *nrows x ncols* plots that are filled in by row. **mfcol=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



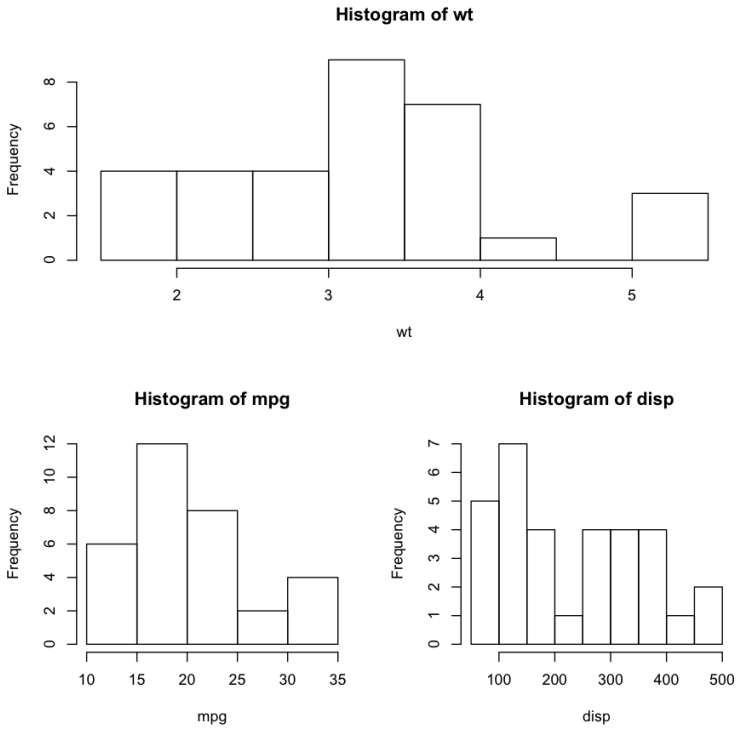






Graphical Parameters

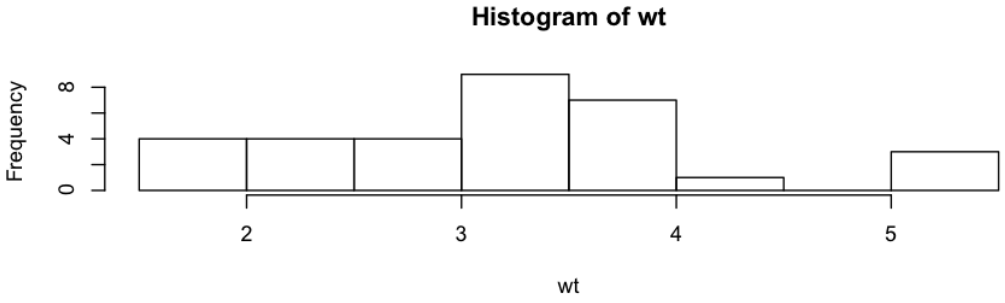
- The **layout( )** function has the form **layout(***mat***)** where *mat* is a matrix object specifying the location of the N figures to plot.
- # One figure in row 1 and two figures in row 2
  attach(mtcars)
  layout(matrix(c(1,1,2,3), 2, 2, byrow = TRUE))
  hist(wt)
  hist(mpg)
  hist(disp)

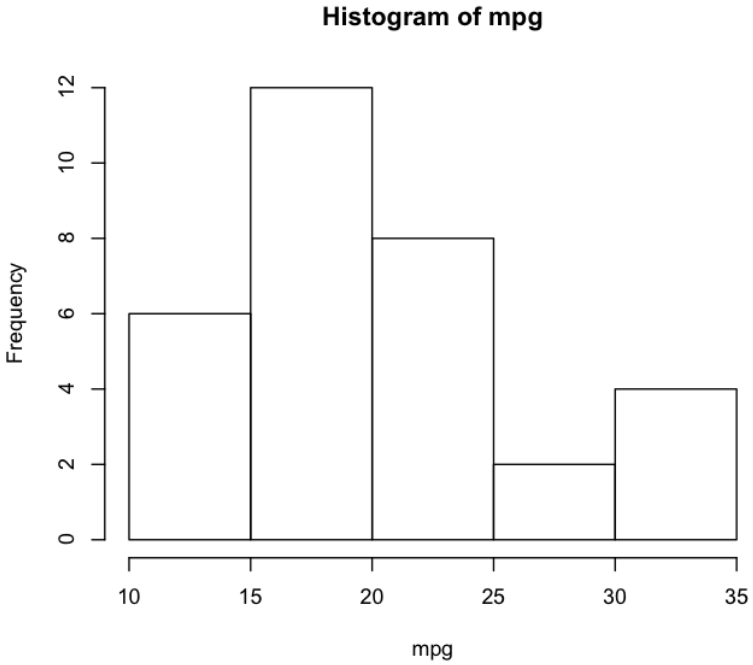


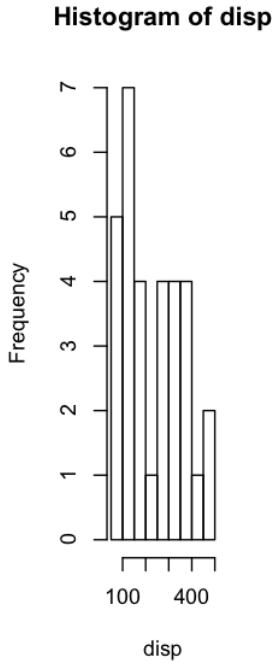
Graphical Parameters

- 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



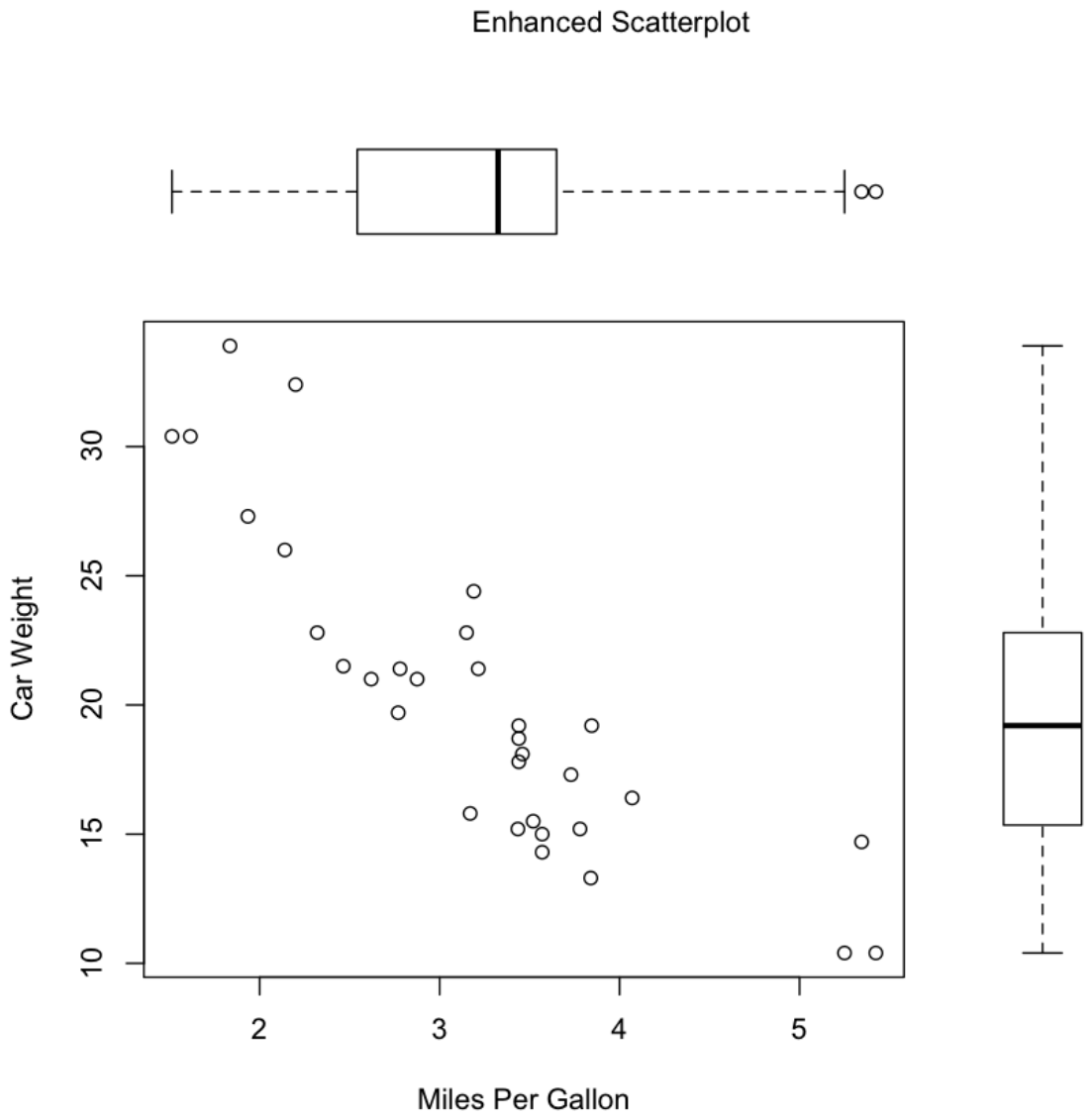




Graphical Parameters

- **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



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.