EME 109 Tutorial for Data Analysis With R

Benjamin D. Shaw
Mechanical and Aerospace Engineering Department
University of California
Davis, CA 95616

02 August 2023

Introduction

R is a software package that is useful for statistical analysis of experimental data. It is not the only software available for data analysis, but R is free and it has become extremely popular over the last several years. There are many people worldwide who have contributed to the development of R.

Many useful routines have been written in R and we will make use of some of these routines for analysis of data. R also has programming capabilities generally associated with a computer language, e.g., defining functions, making decisions, loops, etc. We will consider these topics as well, but only as needed. A goal of this tutorial is to allow you to be able to write simple R scripts rather than always relying on code that someone else has written. This tutorial focuses on topics relevant to EME 109. It covers basic concepts that will be expanded upon as the quarter progresses.

Downloading R

R is compatible with Windows, Macs, and Linux machines. R should be available on the computers in the MAE CAE lab on the second floor of Bainer Hall. You can also download R from the following website.

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

Documentation of R

R is described in many books (try a web search). R documentation can also be found at the above website, e.g., click on the "manuals" link.

Using R

Rather than using a menu-based system, which can be limited in what it can do, we will use the command window that appears when R is started. The symbol ">" generally appears as the prompt in the command window. This prompt is used in the text below. Note that # is the comment symbol, and any text following # on a line of input is ignored by R. The output from R is also shown below each command. The output can include numbers enclosed by square braces, e.g., "[1]", at the beginning of each line of output - the number in braces simply gives the location of the first element in each line when the output is a list of elements.

Note that in the R examples provided here, the command prompt (>) is included so that you can easily determine what you should type – just don't type the command prompt when you go through the commands in this tutorial.

The R code examples we will use are often not the only possibilities for a certain type of data analysis. I have selected ways of doing things that I thought were the most transparent (at least to me).

Getting Help

R provides options for getting help with the commands help() and example(). You just need to fill in whatever you are asking about. For example, suppose we want to learn something about the plot() command. Just type the following commands and information should appear on your computer screen.

```
> help(plot)

> example(plot)
```

You can also get help by typing a question mark followed by whatever you want to be helped with.

```
> ?plot
```

Variables

Variables in R are case sensitive, i.e., "x" and "X" are different variables. A variable can be assigned a value using the assignment operator "<-", which is a less-than symbol followed by a minus sign. For example, the variable named X can be assigned the value 5.2 by typing the following command.

```
> X <- 5.2
```

The assignment operator can also be used in the opposite direction, i.e., "->", as shown below.

```
> 3.8 -> Y
```

This command assigns the value 3.8 to Y.

It is also possible to assign values using the equal sign "=" but this is not recommended because there are (rare) instances where an equal sign may not work correctly.

Vectors

Variables in R are vectorized in the sense that a single variable can hold many values. For example, the variable Y can be assigned the vector (1,2,3,4,5) with the following command.

```
> Y <- c(1,2,3,4,5)
```

Note that c() is actually a function that combines (concatenates) its arguments, in this case the numbers 1, 2, 3, 4, and 5, into a vector.

In some cases you may want to generate vectors with large numbers of elements, which is impractical to do manually. The operator ":" generates a sequence of numbers. The command below generates the number sequence 1, 2, 3, ..., 50 and then stores the resulting vector in the variable Z.

```
> Z <- 1:50
```

The seq() command generates a vector of numbers with specified increments.

```
> seq(from = -0.2, to = 1.7, by = 0.1)
 [1] -0.2 -0.1  0.0  0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1.0  1.1  1.2  1.3  1.4  1.5  1.6  1.7
```

By typing a variable's name (and pressing return) we can see the values associated with the variable.

```
> Z
  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
 [26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

Basic Arithmetic

R has the usual arithmetic operations found in a programming language (+,-,*,/,^). These operations can be used for calculations in an interactive manner as well as in scripts.

```
> 2^3                              # raise to a power
[1] 8
```

R does arithmetic element-by-element with vectors. Below is an example with multiplication of vectors.

```
> x <- c(1,2,3,4,5)
> y <- c(2,4,6,8,10)
> x*y
[1]  2  8 18 32 50
```

Note that R recycles vector entries if an arithmetic operation is supposed to occur between two vectors of different length. What this means is that R will append copies of the shortest vector onto itself, until it is the same length as the longer vector. The arithmetic operation is then performed. For this to work in a predictable manner, the longer vector should have a length that is an integer multiple of the length of the shorter vector. If it does not, then a warning or error message may be generated. Some example code is shown below.

```
> x_short <- c(1,2,3,4)                # a 4 element vector
> x_medium <- c(7,8,9,2,3)             # a 5 element vector
> x_long <- c(2,4,6,8,10,12,14,16)     # an 8 element vector

> x_short+x_long
[1]  3  6  9 12 11 14 17 20

> x_short+x_medium
[1]  8 10 12  6  4
Warning message:
In x_short + x_medium :
  longer object length is not a multiple of shorter object length
```

If a vector is added to a number, each vector element is added to the same number. R treats the number as a vector of length 1 and then recycles it. This also applies to other arithmetic operations (-, *, /,^).

```
> x <- c(1,2,3,4,5)

> x + 0.1
[1] 1.1 2.1 3.1 4.1 5.1
```

Mathematical Functions

Many common mathematics functions are included in R, e.g., exponentials, sines, cosines, etc. You can also define your own functions as described elsewhere in this tutorial.

```
> log(10)                    # the log function is base "e"
[1] 2.302585
```

If you use a vector as the argument of a function, the function operates on each vector component separately.

```
> x <- c(1,2,3,4,5)
> z <- exp(-x)
> z
[1] 0.367879441 0.135335283 0.049787068 0.018315639 0.006737947
```

Writing Your Own Functions

It is relatively straightforward to write functions. The basic structure is shown below for a function we have named fcn. This function takes two vectors A and B, which have to be in the argument list of "function", and calculates another vector C defined as sin(A*B). The part that does the actual calculation is contained between the curly braces "{ }" and you can change this to satisfy your own requirement.

```
> fcn <- function(A,B) {
> C <- sin(A*B)
> }
```

Here is some code that uses this function. Note that A and B need to appear in the function argument list.

```
> A <- 1:20
> B <- rnorm(length(A))
> C <- fcn(A,B)
> C
 [1]  0.31229702 -0.50163372 -0.75735144  0.99787367  0.45329664
 [6]  0.80427810  0.84782315  0.04487570  0.81399208  0.47231956
[11] -0.75590916  0.74246384  0.08321959 -0.25278893  0.56127944
[16] -0.34513298 -0.86995703 -0.94831176  0.67115865  0.98742702
```

The variables used in a function disappear after the function has finished its calculations, so we have assigned the result to another vector C in order to be able to use results from the function. In general, the last quantity calculated in the function is returned. If you want to return several quantities, you can store them in an object such as a vector, list, or data frame and then return this object using the command return(object) inside the function.

For Loops

Sometimes a mathematical operation needs to be performed many times. A way to accomplish this is with a "for" loop. The structure of such a loop is illustrated here with an example where we add a sequence of numbers.

```
> ans <- 0
> for (i in 1:100) {
>         ans <- ans + i
>  }
> ans
[1] 5050
```

In this example, the variable i is incremented by 1 as it goes from 1 to 100. Any commands between the curly braces "{ }" are executed every time i is incremented. The loop ends when i reaches a value of 100.

While Loops

Sometimes a mathematical operation needs to be performed as long as a condition is met. A way to accomplish this is with a "while" loop. The structure of such a loop is illustrated here with an example where we add a sequence of numbers.

```
> j <- 0
> ans <- 0
> while (j <= 100) {
+         ans <- ans + j
+         j <- j + 1
+ }
> ans
[1] 5050
```

In this example, the loop is executed as long as j <= 100. Within the loop (the commands between the curly braces "{ }"), the variable j is incremented by 1 each time the loop executes. The loop is exited when j reaches 100.

Data Frames

Data frames are quite useful. They can be considered to be matrices but where the columns have names (headers). This allows for relatively easy access to specific columns of data for analysis. We can create a data frame as shown below, but in most cases data frames are encountered when data files are imported into R.

Suppose we create the following three vectors.

```
> x <- 1:10
> y <- sqrt(x)
> z <- sqrt(y)
```

A data frame can be created using the data.frame() command. Here, this data frame is associated with the object named W.

```
> W <- data.frame(x,y,z)
> W

    x        y        z
1   1 1.000000 1.000000
2   2 1.414214 1.189207
3   3 1.732051 1.316074
4   4 2.000000 1.414214
5   5 2.236068 1.495349
6   6 2.449490 1.565085
7   7 2.645751 1.626577
8   8 2.828427 1.681793
9   9 3.000000 1.732051
10 10 3.162278 1.778279
```

Note that the column names are x, y, and z. Specific columns from a data frame can be accessed using the $ (dollar sign). This is accomplished by typing the name of the data frame, then $, then the column header. For example we can access the z column using W$z.

```
> W$z
 [1] 1.000000 1.189207 1.316074 1.414214 1.495349 1.565085 1.626577 1.681793 1.732051 1.778279
```

If we want to work with only certain rows of a data frame, we can use the subset() function to pull these rows out of the original data frame and store them in another one.

Exporting Data

We can export data into a file on a computer. For example, suppose we create two new vectors, t_out and x_out as shown below.

```
> t_out <- 0:20
> x_out <- cos(t_out/3.14159)
```

We can export the manipulated data into a csv file with the following commands.

```
> for.export <- data.frame(t_out,x_out)    # create a data frame named for.export with headers t_out and x_out
> # write the data into a csv file named EME_107A_Sample_R_Dataset
> write.csv(for.export, file="EME_107A_Sample_R_Dataset.csv")
```

The resulting file should now be available in the working directory.

Reading from Data Files

R can access data files in several different formats. We will work with data files in the comma-separated values (csv) format. The file we will use in this case is named EME_107A_Sample_R_Dataset.csv. This data file, which you created above, needs to be in the R working directory. You can set the working directory using menu commands in R or with the following command.

```
> setwd("~/Desktop/EME_107A_R_Tutorial")  # fill in your own filepath to the data file
```

Once the working directory has been set, you can read in the data.

```
> mydata <- read.csv("EME_107A_Sample_R_Dataset.csv",header = TRUE,sep = ",")
```

The read.csv() command loads the data from the data file into the object we have named "mydata". The entry header=TRUE tells R that the first entry in each column of data is actually a header while sep="',' tells R that the entries in each row are separated by commas. Here are the first few rows of this data set.

```
    X t_out       x_out
1   1     0  1.00000000
2   2     1  0.94976563
3   3     2  0.80410951
4   4     3  0.57766552
5   5     4  0.29318420
6   6     5 -0.02075296
7   7     6 -0.33260510
8   8     7 -0.61104082
9   9     8 -0.82808605
10 10     9 -0.96193451
```

Using the following commands gives us information about the object "mydata".

```
> class(mydata)
[1]  "data.frame"

> names(mydata)
[1] "X"     "t_out" "x_out"

> sapply(mydata, class)
        X     t_out     x_out
"integer" "integer" "numeric"
```

Note that "mydata" is a data frame and has three columns: X, t_out, and x_out. You can access the data in the columns by typing mydata$t_out and mydata$x_out.

Mean, Median, Standard Deviation, and Variance of a Sample

Consider the following data set.

```
> x <- c(2.3,4.2,3.2,4.1,1.1,5.4,3.3,4.4,3.7,2.7,3.0,1.9,7.9,4.6,3.4)           # a data set
```

R has <u>built-in</u> functions for these calculations.

```
> mean(x)                                  # mean
[1] 3.68
> median(x)                                # median
[1] 3.4
> sd(x)                                    # standard deviation
```

```
[1] 1.60766
> var(x)                                  # variance
[1] 2.584571
```

Covariance and Correlation between Two Samples

Consider the following data sets.

```
> x <- c(2.3,4.2,3.2,4.1,1.1,5.4,3.3,4.4,3.7,2.7,3.0,1.9,7.9,4.6,3.4)       # a data set
> y <- c(2.1,3.2,3.9,4.8,1.9,5.0,3.2,4.7,3.0,2.6,2.5,2.9,7.8,4.7,3.2)       # another data set
```

The covariance and correlation between these two data sets can be calculated with built-in functions.

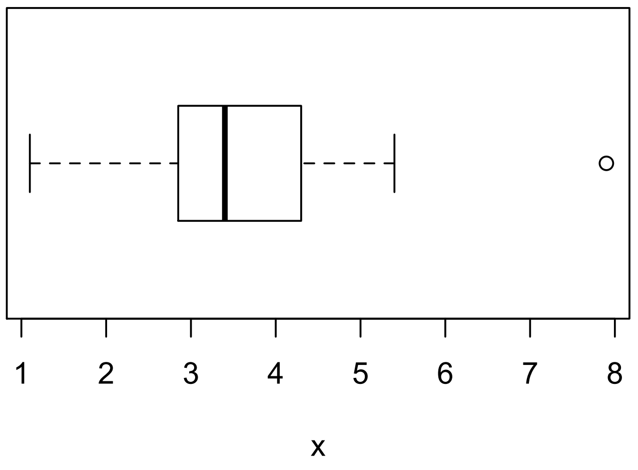
```
> cov(x,y)                                # covariance
[1] 2.27
> cor(x,y)                                # correlation
[1] 0.93
```

Note that the covariance of a data set with itself is the variance of the data set and the correlation of a data set with itself is unity. The correlation of x and y is defined as the ratio of the covariance of x and y to the product of the standard deviations of x and y.

Boxplots

Boxplots are useful for visualizing univariate data distributions. The "box" in a boxplot shows the median (the thick line inside the box) and top and bottom of the box denote data quartiles. The whiskers extend out a user-specified distance, i.e., some number times the interquartile range (IQR), above and below the box. Boxplots can also be useful for identifying data points that can be considered potential outliers by marking points that are outside the whiskers.

```
> x <- c(2.3,4.2,3.2,4.1,1.1,5.4,3.3,4.4,3.7,2.7,3.0,1.9,7.9,4.6,3.4)       # a data set
> boxplot(x,range=1.5,horizontal=TRUE,xlab="x")                             # generate the boxplot
```



The "range=1.5" term tells boxplot how many IQRs to extend the whiskers from the sides of the box. In this example, a possible outlier is indicated by the circle. Using "range=0" extends the whiskers to the data extremes.

We can print numerical data corresponding to this boxplot with the fivenum() command. The first and last entries in the output are the minimum and maximum values, respectively, and the middle three values are the quartiles (hinges) Q_1, Q_2, and Q_3, where Q_2 is the sample median.

```
> fivenum(x)
```

```
[1] 1.10 2.85 3.40 4.30 7.90
```

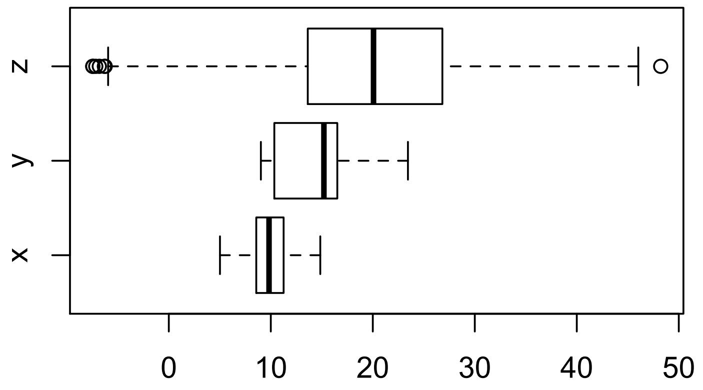
Boxplots can also be used to compare the distributions of several data sets. For example, consider the three data sets below, which are generated using a normal pseudorandom number generator.

```
> x <- rnorm(n=100,mean=10,sd=2)
> y <- rnorm(n=20,mean=15,sd=4)
> z <- rnorm(n=1000,mean=20,sd=10)
```

A boxplot showing all three data sets can be created with the following command.

```
boxplot(x,y,z,horizontal=TRUE,names=c('x','y','z'))
```

The resulting boxplot is shown below.



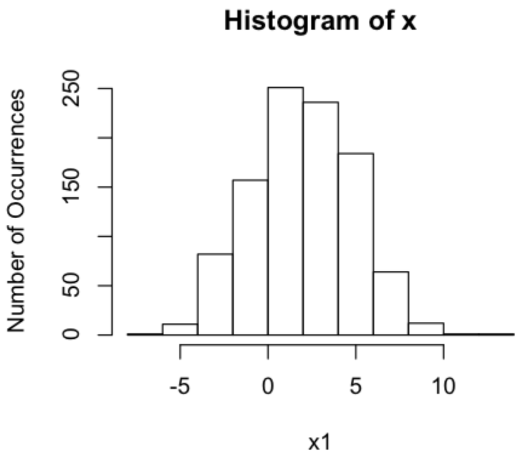
Histograms

A histogram is useful for visualizing the general distribution of a univariate data set. We will first generate a data set named x.

```
> x <- rnorm(n=1000,mean=2,sd=3)        # generate a set of 1000 random numbers using a normal pdf
```

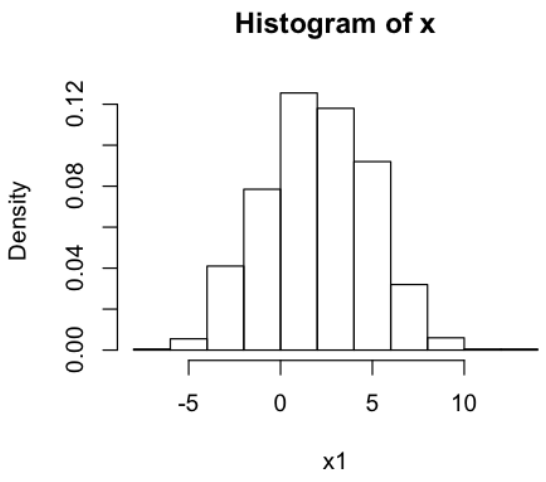
Here is a histogram for this data set.

```
> hist(x,breaks=10,xlab="x1",ylab="Number of Occurrences")
```



A histogram can be displayed as a probability density function (pdf) by using the prob=TRUE setting. In this case the default title for the vertical axis is "Density".

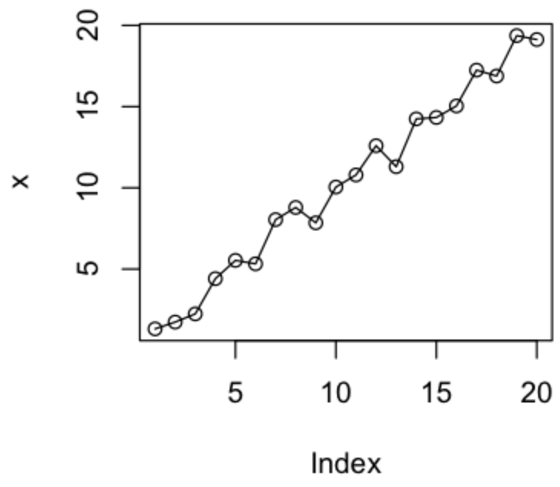
```
> hist(x,breaks=10,xlab="x1", prob=TRUE)
```



Plotting Univariate Data Sets

We can generate a plot of univariate data using the plot() function and then connect the data points with straight lines using the lines() function.

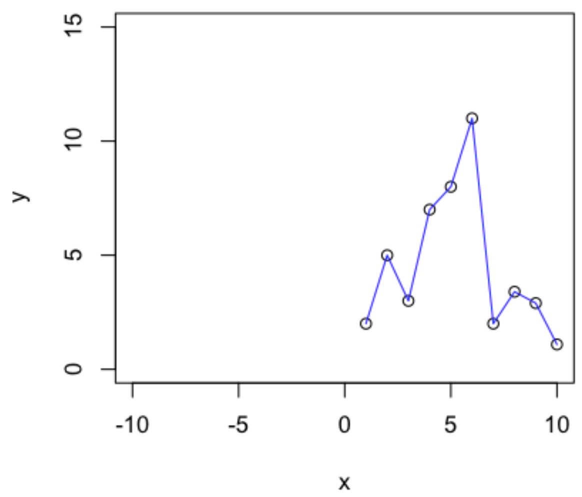
```
> x <- 1:20                                                # generate some data
> x <- x + rnorm(length(x))                                # generate some data
> x
 [1]  1.317610  1.737257  2.239218  4.409005  5.534226  5.320658  8.042150
 [8]  8.791010  7.847137 10.049192 10.792314 12.593327 11.301964 14.242654
[15] 14.331953 15.040276 17.246865 16.883476 19.369419 19.128125
> plot(x)                                                  # plot the data points
> lines(x)                                                 # connect the data points
```



Plotting Bivariate Data Sets

To plot a bivariate set of data, we can use the plot() command.

```
> x <- c(1,2,3,4,5,6,7,8,9,10)                             # define a vector of x values
> y <- c(2,5,3,7,8,11,2,3.4,2.9,1.09)                      # define a vector of y values
> plot(x,y,xlim=c(-10,10),ylim=c(0,15),xlab="x",ylab="y")  # generate a scatter plot of x vs. y
> lines(x,y,col="blue")                                    # connect the data points with blue lines
```



This plot() command plots the data points as circles(as the default) and the limits for the axes are set by the xlim=c(-10,10) and ylim=c(0,15) entries. Blue lines then are added to connect the data points using the lines(x,y,col="blue") command.

Parallel Processing

If your calculations take too long, a way to speed things up is to employ parallel processing. If you need to repeat a calculation M times and you have a computer with N cores, then you can have each core perform M/N calculations simultaneously. The results from each core can then be combined so you can have the results from all M calculations.

Parallel processing is not useful for all types of calculations, but in cases where a computer does the same thing over and over and where a particular calculation does not depend on the results of prior calculations, as with a Monte Carlo calculation, then using it can result in a significant reduction in the time that you wait for results.

The R commands below show a simple example that I ran on a 4-core Mac computer. A function named fcn() is defined. A for() loop, which uses a single core, is first used and the time to execute the function four times is printed along with the results of the calculations, which are stored in the variable ans1. After this, a 4-node cluster is created and used, where each node (core) executes the function only once. The function parSapply() handles all of the details regarding the nodes and their calculations and the results are placed into ans2, which is then printed.

```
> rm(list=ls())              # clear everything
>
> library(parallel)          # a parallel processing library
> library(snow)              # a parallel processing library
>
> # place the code to be executed on each node in a function
> fcn <- function(x) {
        M <- 100
        r <- numeric(M)
        N <- length(x)
        for (m in 1:M) {
                a <- sample(x=x,size=N,replace=TRUE)
                r[m] <- mean(a)
        }
        return(mean(r))
 }
>
> # find the number of cores
> numCores <- detectCores()
> numCores
```

```
[1] 4
>
> # create some data
> x_data <- rnorm(n=1e5,mean=10,sd=2)
>
> # store identical data sets in a list
> x <- rep(list(x_data),numCores)
>
> # measure the time to run the code using a for loop
> tic()
> ans1 <- numeric(numCores)
> for (n in 1:numCores) {
+         ans1[n] <- fcn(x_data)
+ }
> ans1
[1] 10.000670 10.001023 10.001598  9.999956
> toc()
1.846 sec elapsed
>
> # create the cluster
> cl <- makeSOCKcluster(numCores)
>
> # measure the time to run the code in parallel using parSapply()
> tic()
> ans2 <- parSapply(cl,x,fcn)
> ans2
[1] 10.00182 10.00152 10.00024 10.00092
> toc()
0.763 sec elapsed
>
> # shut down the cluster
> stopCluster(cl)
```

The time to execute the calculations was reduced from 1.846 s to 0.763 s as a result of using parallel processing, which is a decrease of better than a factor of two. When I ran these same calculations on a 32-core computer, the time decreased by more than a factor of 20!

Some Functions Operations, and Packages

```
> attach()                        # make the columns in a data frame accessible by name
> D <- data.frame(X,Y,Z)          # create data frame D with vectors X, Y, and Z
> fix(D)                          # open a spreadsheet to allow editing of data frame D
> head()                          # display the first few rows of a data frame
> subset()                        # create a subset of a data frame
> tail()                          # display the last few rows of a data frame

> X <- c(3.2,4.6,2.3,6.5,1.1)     # define a set of data as the vector X
> fivenum(X)                      # print the min, quartiles (median and hinges), and max of the data
> summary(X)                      # print a summary of the data statistics
> X[2]                            # access the 2nd element of the vector X
> X[2:4]                          # access elements 2 through 4 of the vector X
> X <- c(X,10.9)                  # add a data point to the vector X (it is added onto the end of X)
> X[-6]                           # remove data point number 6
> length(X)                       # number of elements
> max(X)                          # largest data value
> mean(X)                         # data average
> median(X)                       # data median
> min(X)                          # smallest data value
> prod(X)                         # multiply all elements of X by each other
> range(X)                        # minimum and maximum values in X as a two-element vector
> sample()                      # draw a random sample
> sd(X)                         # data standard deviation (sample, not population)
> seq(from=-1,to=2,by=0.1)      # generate a sequence of numbers from -1 to 2 in increments of 0.1
> sort(X)                       # sort the data points from smallest to largest
> sum(X)                        # add all the elements of X
> var(X)                        # data variance (sample, not population)

> pnorm(1.5)                    # normal probability that z < 1.5
> qchisq(0.95,df=10)            # chi-square value for 10 degrees of freedom and a probability of 0.95
> qnorm(0.95)                   # z value corresponding to a normal probability of 0.95
> qt(0.75,df=2)                 # student's t value for 2 degrees of freedom and a probability of 0.75
> quantile(X, probs = 0.4)      # estimate a value from a pdf for X below which 40% of the data exist
> range(X)                      # smallest and largest values of the data set X
> rnorm(n=5,mean=0,sd=1)        # generate 5 random numbers using a normal pdf
> runif(n=5,min=-1,max=1)       # generate 5 random numbers between -1 and 1 using a uniform pdf


> abline()                      # draw a straight line
> boxplot()                     # generate a boxplot
> curve()                       # plot a mathematical function
> density()                     # generate a pdf curve
> identify()                    # interactive plotting
> lines()                       # join data points with straight lines (call after plot())
> locator()                     # interactive plotting
> pairs()                       # generate a matrix of plots for a data frame or matrix
> par()                         # can be used to query or set graphical parameters
> plot()                        # plot univariate or bivariate data sets

> builtins()                    # list built-in functions
> data()                        # list built-in data sets
> data(cars)                    # load the data set named "cars"
> dim()                         # retrieve or set the dimension(s) of an object
> fix()                         # open an editor for a data frame
> library()                     # display R packages that have been installed on your computer
> library(shiny)                # load the R package named "shiny"
> ls()                          # list objects in the workspace
> ls.str()                      # list information about objects in the workspace
> rm(X,Z)                       # remove objects X and Z from the workspace
> search()                      # display R packages that have been loaded
> str()                         # display the structure of an object
> source("commands.R")          # run the script "commands.R"
```

Packages

locpol: local polynomial curve fitting
lm: linear regression curve fitting
parallel: parallel processing
snow: parallel processing