KUANT Guides

Guide No.
KUANT 20.2

Introduction to R

A guide for the absolute beginner: R installation, programming, and data handling.

Miller, P., & Jorgensen, T. D. (2013)

crmda.ku.edu

This guide is designed to introduce brand new users of R to its basic functionality. It will cover installation and loading of packages; importing data; working with vectors, data frames, and functions; and summarizing data in tables and figures. It will not cover statistical analyses or more advanced topics such as simulating data, creating user-defined functions, or using control-flow operators such as `for` loops or `if` logic.

Introduction to R

R is an open source software environment for statistical computing. One of the primary advantages to using R for statistical computing is that it is highly extensible—not only can you create your own functions and packages, but the R community as a whole builds and maintains thousands of such packages. For example, there are several packages designed for latent variable analysis including lavaan, OpenMx, ltm, and sem.

Another primary advantage for using R is that it is a programming language designed for statistical analysis. Using a programming language allows for very low-level customization (the analytical process tends to feel "hands on") but can also increase the learning curve. In this guide we will examine the very basics of the R language, which are necessary to use the many packages for data analysis in R.

First Steps

If you have not yet installed R, find it at this link: http://cran.r-project.org/

Using installation defaults should generally be sufficient for most users.

To install packages (for instance, *lavaan*), open R and type this at the command prompt:

```
> install.packages("lavaan")
```

Select a CRAN mirror such as University of Kansas when prompted (any in the same continent as your computer should be sufficient). Load the package by typing the following:

```
> library(lavaan)
```

The R Programming Language

Typically, it is best to learn programming languages by actually programming. So as you read through this guide, do the examples and play around by trying different things on your own. (*Note*: Type out the examples yourself, as copy/pasting from a PDF to the R terminal may not result in the expected plain-text format of certain characters, such as quotation marks.)

Have some fun! You will find cited at the end of the guide several resources I've found helpful (including a reference card). The following links are also basic introductions, but more thorough and extensive than what is provided here:

http://scc.stat.ucla.edu/page_attachments/0000/0141/10S-basicR.pdf
http://www.stat.auckland.ac.nz/~stat782/downloads/01-Basics.pdf

The R Environment

R is primarily an interpreted language, which means that syntax is executed and run through an interpreter (in this case the R console window) rather than compiled into machine code. Because you work in R by simply typing in commands and having them evaluated, it is best to use a text editor to save your commands so you can run, review, and change your code as necessary.

The RGui interface contains a basic text editor which you can open by selecting "New Script" under the "File" menu. From the text editor, you can evaluate lines of your code by hitting Ctrl-R.

The R Interpreter

In interpreted languages, what you type in at the prompt gets evaluated as a computation. When starting R, you are greeted with some text describing the version information, citation instructions, and instructions for help. Below that is the prompt: ">"

```
>
```

Expressions typed for computation at the prompt get evaluated immediately. To see this, type the simple mathematical expression `2 + 3` into the interpreter:

```
> 2 + 3
[1] 5
```

The result returned is 5. In this guide, expressions typed by the user will be in italics, while the result returned by evaluating the expression will be un-italicized. Anything following the pound symbol (or hash mark) "#" is a comment and is not evaluated.

R Programming Fundamentals

Vectors

Programming languages are built around data, and doing operations on that data. In many programming languages, data is stored in variables—just think of them as boxes. Each box has a name, and you access the contents of the box by referencing its name. In R, the basic data storage device is a vector, which can contain any number of values. You can think of a vector as a stack of boxes, each of which you can access by referencing its name.

A vector can be named any sequence of characters. To assign to (store things in) a vector, you use the assignment key "<-", which is obviously 2 characters: "<" and "-" (less than and dash).

The following are examples of creating vectors and storing things in them. Access the value you stored by typing the name of the vector you created into the interpreter.

```
> A <- 3
> A
[1] 3
> B12 <- "a string"
> B12
[1] "a string"
```

Vectors can also store lists of values.

```
> x <- c(1, 2, 3)
> x
[1] 1 2 3
> y <- c("one", "two", "three")
> y
[1] "one" "two" "three"
```

You can access individual items in lists using indexing.

```
> x[1]
[1] 1
> y[2]
[1] "two"
```

Data Frames
The second primary way you store data in R is the data frame, which is an object with rows and columns (a list of vectors). The rows are observations, the columns are variables. You can build a data frame by first creating several individual vectors and then combining them. In the following example, we create three different vectors of lists and then combine them into a data frame.

```
> x <- 1:3 # Creates a vector of integers from n1:n2, count by 1
> y <- seq(from = 4, to = 6, by = 1) # another vector from 4:6
> z <- rep(7, times = 3) # a vector of 3 sevens
> grp <- c(0, 1, NA) # "NA" indicates a missing value
> dat <- data.frame(x, y, z, grp) # put each vector in a column
> dat
  x y z grp
1 1 4 7   0
2 2 5 7   1
3 3 6 7  NA
```

If each row is an observation and each column a variable, then observation 1 has a score of 1 on x, 4 on y, and 7 on z Later we will create data frames by reading a file, and describe how to do some basic operations on them.

Functions

In R, functions are the way you get work done. Often, they take a value, do something to it, and return a new value. They have the general form `function_name( argument(s) )`. R has many built in functions:

```
> range(x)      # range (min and max) of values in a vector
[1] 1 3
> sum(x)        # sum of all values in a vector
[1] 6
> prod(x)       # product of all values in one vector
[1] 6
```

As you saw, the function applied itself to all the values in the vector. This also works for data frames:

```
> sum(dat)
[1] NA
```

The `sum()` function returns a missing value (`NA`) because there are missing values in the data frame. To remove those missing values and calculate the sum of all observed values, include the `na.rm` (NA-remove) argument:

```
> sum(dat, na.rm = TRUE)
[1] 43
```

Functions typically have a large number of arguments that allow you to specify in more detail how the function runs, and what output it produces. To see these options for a function, type a question mark in front of the function name:

```
> ?mean
```

Setting the Working Directory

Using R requires you to think in a fundamentally different way about working with your files. Instead of opening a script or reading in data by browsing to that file, you can actually tell R to work within that directory itself. Essentially, you can think about it as moving R to your files instead of moving your files to R. To do this, first find your current working directory.

```
> getwd() # notice backslashes in Windows are changed to forward
[1] "C:/Users/Username/Documents"
```

Then set your new working directory (say, where you have your data files & R scripts), e.g.:

```
> setwd("C:/Users/Username/Documents/My_Project")
> setwd("E:/SEM/Project_1") # on Windows, must change "\" to "/"
```

Now you can open files by name in this directory rather than by file path. To show a list of filenames in the current directory you can use the following function:

```
> dir()
```

Reading Data into R

To read in data, you will use one of the following functions. Each takes a filename as one of the arguments, and returns a data frame object.

```
> dat <- read.table("file") # Reads any file in table format
> PosNeg <- read.csv("file", header = TRUE) # Reads a *.csv file
```

If you don't have a header (i.e., no variable names), set `header = FALSE`. The `read.csv()` function (CSV stands for "comma-separated values) is simply the `read.table()` function with the argument `sep = ","` instead of the default `sep = " "` (a space). If your data is separated by a tab instead of a space, set `sep = "\t"` or use the `read.delim()` function:

```
> myData <- read.delim("filename")
```

All these functions have important options, including *`sep`*, `quote, row.names, na.strings,` and `skip`. Particularly important is the option *`na.strings`*, which tells R what to interpret missing values as. For instance, if the value −999 means it is a missing value, specify the argument `na.strings = "-999"`. See how to use these options by typing

```
> ?read.table
```

For ease of use, stick to text files or *.csv, and use excel to convert file formats when necessary. Once you have read in your data, you can work with your data frame with some of the following functions/methods. This is an important step that helps you verify that the data were read in correctly.

```
> names(dat)         # or colnames(dat) for names of variables
> head(dat, 10)      # view the first 10 lines of the data.frame
> summary(dat)       # a statistical summary of each variable
```

It may be necessary or desirable to change the default names of the variables in the data frame after it has been read in. To do this, first create a vector of variable names –

```
> varnames <- c("name1","name2","name3","name4","name5","name6")
```

Then assign this vector of variable names to the variable names of the data frame. To do this, we get the original column names, and assign to it the new vector we created.

```
> colnames(myData) <- varnames
> head(myData)

name1    name2    name3    name4    name5    name6
1.00000  0.61251  0.61733 -0.00204 -0.17237 -0.18014
0.61251  1.00000  0.65757 -0.11445 -0.27750 -0.25120
```

Summarizing Data in Tables and Figures
To plot univariate or bivariate categorical data, you can print a table or frequencies/counts. Just specify the two vectors you want to describe. Here is a univariate table of a variable "`u1`" with 3 categories (coded 0, 1, and 2):

```
> myData <- read.table(
         "http://www.statmodel.com/usersguide/chap3/ex3.12.dat")
> names(myData) <- c("u1","u2","u3","x1","x2","x3")
> table(myData$u1)

   0   1   2
 199 113 188
```

Here is a bivariate table of variables "`u1`" & "`u2`" with 3 and 2 categories, respectively:

```
> table(myData$u1, myData$u2)

     0   1
 0 161  38
 1  61  52
 2  42 146
```

To add marginal counts to the table, save the table as an object, then use the `addmargins()` function:

```
> myTable <- table(myData$u1, myData$u2)
> addmargins(myTable)

       0   1 Sum
 0   161  38 199
 1    61  52 113
 2    42 146 188
 Sum 264 236 500
```

You can also see percentages instead of counts by using the `prop.table()` function:

```
> prop.table(myTable)

       0     1
```

```
  0 0.322 0.076
  1 0.122 0.104
  2 0.084 0.292
```

To specify row percentages or column percentages, specify the first or second dimension of the table:

```
> prop.table(myTable, 1)

            0         1
  0 0.8090452 0.1909548
  1 0.5398230 0.4601770
  2 0.2234043 0.7765957

> prop.table(myTable, 2)

            0         1
  0 0.6098485 0.1610169
  1 0.2310606 0.2203390
  2 0.1590909 0.6186441
```

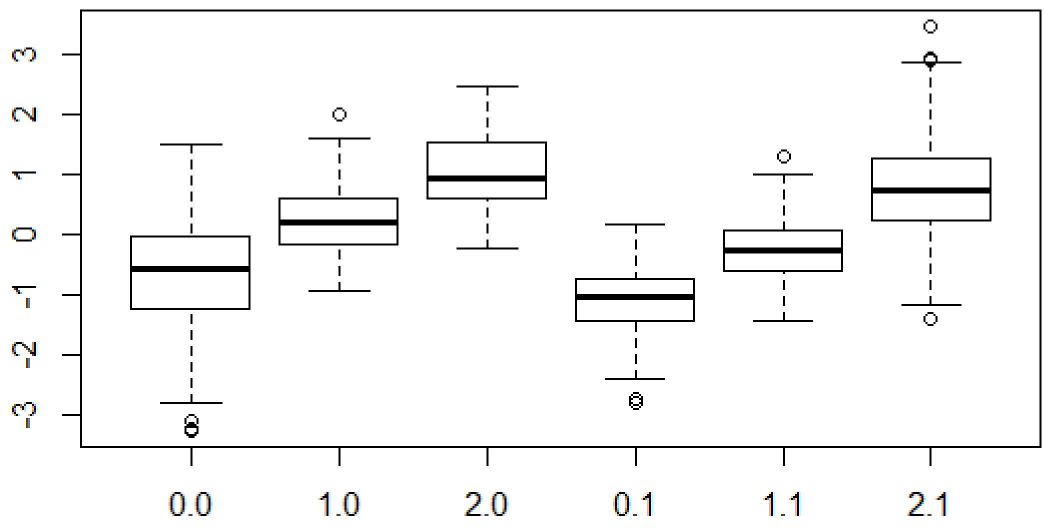
The `aggregate()` function can be used to make a table of continuous measures (e.g., mean or standard deviation) in each level of one or more categorical variables:

```
> aggregate(x1 ~ u1, data = myData, mean)
  u1          x1
1  0 -0.70744944
2  1  0.03680895
3  2  0.84773632

> aggregate(x1 ~ u1 + u2, data = myData, sd)
  u1 u2        x1
1  0  0 0.8881932
2  1  0 0.6171643
3  2  0 0.6392670
4  0  1 0.7372694
5  1  1 0.6219115
6  2  1 0.8917143
```

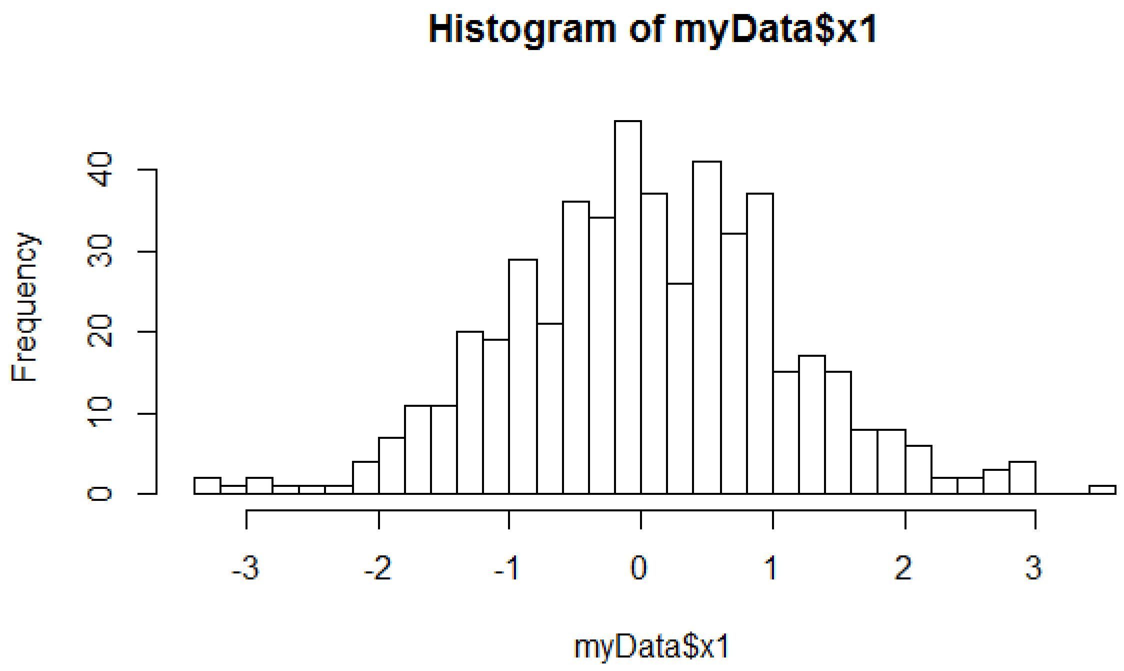
A corresponding boxplot can be made using `boxplot()` function:

```
> boxplot(x1 ~ u1 + u2, data = myData)
```



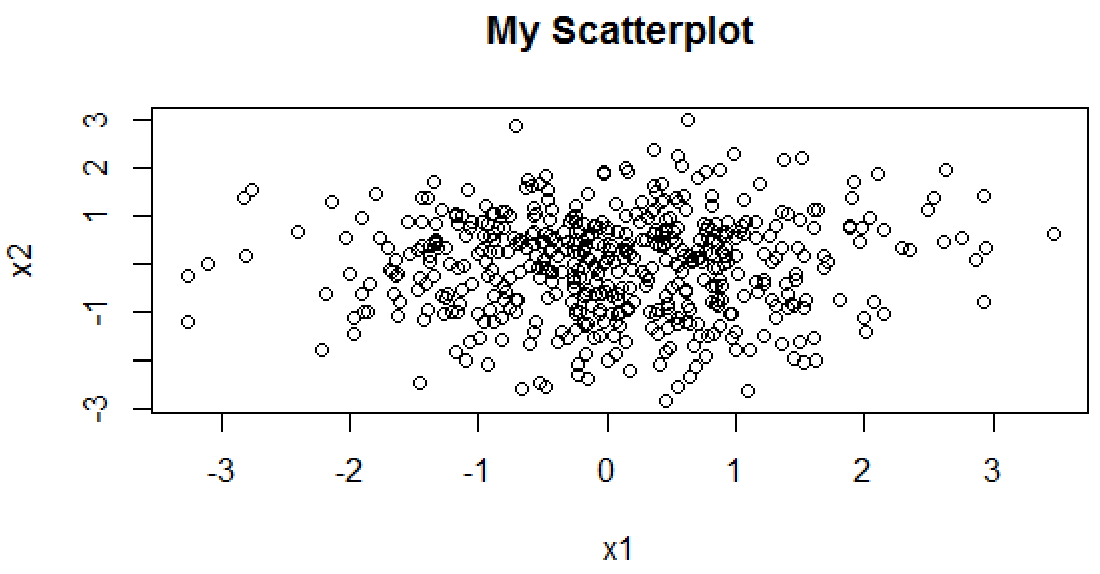
An appropriate univariate plot of a continuous variable is a histogram:

```
> hist(myData$x1, breaks = 30) # "breaks = __" is optional
```



A bivariate relationship between two continuous variables (e.g., a correlation) can be represented with a scatterplot:

```
> plot(x2 ~ x1, data = myData, main = "My Scatterplot")
```



Note: A title can be added to any graph with `main`, and labels can be added to the *x* and *y* axes using `xlab` and `ylab`. See `?plot` for more options.

Distinguishing Data Frames and Matrices: Choosing Subsets
Matrices are like vectors, but they are 2-dimensional. (It is also possible to have 3 or more dimensional arrays in R.) Like a vector, every element in a matrix must be of the same type (e.g., all numeric or all characters). A data frame, on the other hand, is a merely list of vectors, so each column/vector of a data frame can be a different type (one numeric, one character, etc.). Because all vectors in a data frame must have the same length, it resembles a matrix in that it is square (*N* rows by *P* columns, where each column in a data frame is really a separate vector).

In both matrices and data frames, you can access specific rows and columns using brackets after the name of the object, separating row numbers/names and column numbers/names with a comma: `myData[row, col]`. Reading the first four rows of data is the same as using the `head()` function:

```
> head(myData, 4)
  u1 u2 u3        x1        x2        x3
1  1  0  2  0.573051 -0.175230 -1.339954
2  1  1  2 -0.577052  0.425472  0.179867
3  0  0  0 -0.694153 -0.766538  0.455033
4  0  0  0 -0.817974 -1.559255  0.579605

> myData[1:4, ]
  u1 u2 u3        x1        x2        x3
1  1  0  2  0.573051 -0.175230 -1.339954
2  1  1  2 -0.577052  0.425472  0.179867
3  0  0  0 -0.694153 -0.766538  0.455033
4  0  0  0 -0.817974 -1.559255  0.579605
```

Notice that leaving the columns space empty means that you select all columns (likewise if you leave the rows space empty). You can select rows or columns by specifying the appropriate numbers or the corresponding names:

```
> myData[1:4, c(1:3, 6)]
  u1 u2 u3        x3
1  1  0  2 -1.339954
2  1  1  2  0.179867
3  0  0  0  0.455033
4  0  0  0  0.579605

> myData[1:4, c("u1", "u2", "u3", "x3")]
  u1 u2 u3        x3
1  1  0  2 -1.339954
2  1  1  2  0.179867
3  0  0  0  0.455033
4  0  0  0  0.579605
```

Data frames (but not matrices) are also lists (i.e., a list of column vectors), so their columns can also be selected using the dollar sign after the object name. (Note that only one column at a time can be specified this way)

```
> head(myData$x1)
[1]  0.573051 -0.577052 -0.694153 -0.817974  0.463916 -0.096545
```

Logical vectors can be used to specify rows that match certain criteria. Logical vectors consist of values that are either TRUE or FALSE (in all caps). For example, to see whether each value of "`x1`" is positive:

```
> myData$x1 >= 0
  [1]  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
 [11] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
     ...
[491]  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE
```

To compare two values, the possible commands are

`==`	equal
`!=`	not equal
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to
`A %in% B`	test whether each element in vector `A` is one of the elements in vector `B`

We can conduct logical tests using Boolean operators:

`&`	and	`(T & T = T ;  T & F = F ;  F & T = F ;  F & F = F)`
`|`	or	`(T | T = T ;  T | F = T ;  F | T = T ;  F | F = F)`
`!`	not	`(!T = F ;  !F = T)`

You can place a vector of logical values corresponding to some criterion (say, only levels 0 and 1 of the variable “u1”) in the rows bracket, and for every Nth element that is TRUE, that Nth row will be selected:

```
> # the 5th, 7th, 8th, and 10th rows are FALSE
> myData$u1 %in% c(0, 1)
  [1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE ...

> # the 5th, 7th, 8th, and 10th rows are omitted
> myData[ myData$u1 %in% c(0, 1), ]
    u1 u2 u3        x1        x2        x3
1    1  0  2  0.573051 -0.175230 -1.339954
2    1  1  2 -0.577052  0.425472  0.179867
3    0  0  0 -0.694153 -0.766538  0.455033
4    0  0  0 -0.817974 -1.559255  0.579605
6    0  0  1 -0.096545 -0.352276  0.253673
9    0  0  0  0.761720 -1.901134 -2.223851
11   1  1  0 -0.295120  0.881524  0.966334
...
```

Add the additional condition that values of “x1” are positive:

```
> myRows <- myData$u1 %in% c(0, 1) & myData$x1 >= 0
> myData[ myRows, ]
    u1 u2 u3       x1        x2        x3
1    1  0  2 0.573051 -0.175230 -1.339954
9    0  0  0 0.761720 -1.901134 -2.223851
21   0  0  2 0.446963  0.080041 -0.838566
30   1  0  3 0.361164  0.628857 -0.597038
```

List of Functions/Operators

Name	Description	Example Usage
install.packages()	Installs packages	*`install.packages("packageName")`*
library()	Loads packages	*`library(packageName)`*
c()	Builds a vector	*`x <- c(1, 2, 3)`*
list()	Builds a list	*`y <- list(1:3, 1, 2, 3)`*
data.frame()	Builds data frame from variables/vectors	*`myData <- data.frame(x, y, z)`*

:	Creates a vector with values from x : y	x <- 1:3 [1] 1 2 3
range()	Returns the min and max elements in a vector	range(x) [1] 1 3
sum()	Calculates the sum of elements in a vector	sum(x) [1] 6
mean()	Calculates the mean of the elements in a vector	mean(x) [1] 2
prod()	Calculates the product of elements in a vector	prod(x) [1] 6
cor()	Calculates correlations of 2 vectors or all data	cor(x, y) # two vectors cor(myData) # all variables
getwd()	Returns the current working directory	getwd() "C:/User/Specific/Path"
setwd()	Sets the current working directory	setwd("c:/mypath/data")
dir()	Returns files in the current working directory	dir()
?	Launches the documentation for a function	?mean

read.table()
read.csv()
read.delim()
	Read data from files (in or relative to the current working directory)	myData <- read.table("x.dat")
		myData <- read.csv("x.csv")
		myData <- read.delim("x.dat")

names()
colnames()
	Returns the names of variables in a data frame	names(myData)
		colnames(myData)

colnames()	Can also assign the column names	newNames <- c("x1","x2","y") colnames(myData) <- newNames

head()
tail()
	Returns the first or last *N* (e.g., 6 or 10) rows in a data frame	head(myData, n = 10)
		tail(myData, 6)

summary()
aggregate()
	Calculates summary statistics for variables in a data frame	summary(data)
		aggregate(y ~ group, data = myData, mean)

Relational Operators	comparison of values in vectors	x < y x > y x <= y x >= y x == y x != y x %in% y
Binary Arithmetic Operators	Perform arithmetic on numeric	x + y x - y x * y x / y x ^ y OR x**y (power)

		x %% y (modulo) *x %/% y (integer division)*
Logical Operators	Boolean tests of logical values	*!x* *x & y* *x | y*
Extractors	Extract elements of a matrix or a data frame	*x[i] (for 1-dimensional vectors)* *x[i, j]* *x$j (only for data.frames)*

Plot Functions	Create figures	*plot(y ~ x)*
		hist(y)
		boxplot(y ~ x)

Frequency and Contingency Tables	View tables of counts/percentages of categorical variable(s)	*table(x)*
		addmargins(table(x))
		prop.table(table(x), 1:2)

References

Crawley, M. J. (2007). *The R Book.* West Sussex, England: John Wiley & Sons, Ltd.

Hornick, K. (2009). *The R FAQ.* Retrieved from http://CRAN.R-project.org/doc/FAQ/R-FAQ.html

R Development Core Team (2009). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

Short, T. (2004, 11 07). *R Reference Card.* Retrieved 07 2010, from http://cran.r-project.org/doc/contrib/Short-refcard.pdf