Some Useful R Pointers

P. Rossi 1/1/05
Revised 4/21/05

Note: these notes assume a Windows environment. All R commands and objects are the same under Windows and LINUX but the install procedure and GUI are different.

Obtaining R

Visit http://cran.r-project.org/ or google "R language."

CRAN is a network of mirror sites that allow you to download precompiled binary versions of R or source.

Look under "Precompiled Binary Distributions" and click on **Windows**. Click on **base** on the next page and download the rwXXXX.exe file (XXXX=2010 at present) – right click and "save target as." Doubleclick the file name under Windows Explorer and R will install itself in the usual fashion for Windows software.

You may also obtain the optimized BLAS version for Pentium 4 chips by visiting the **contributed** link and click on the ATLAS directory. Simply download the RBLAS.dll file for your chip (invariably Pentium 4) and replace the RBLAS.dll file in the R/bin directory (this will be something like C:/Program Files/R/rwXXXX/bin). My experience is that this will double the speed of many common matrix operations.

Customizing the R Shortcut

The standard R install will create a desktop shortcut to invoke R. This will start up R pointing to the rwXXXX directory. It is more useful to modify the shortcut to start up with R pointing to a directory which is "closer" to the directories you plan on working in. To modify the short-cut, right click on it and choose "properties", change the "Startin: " value to any valid directory on your machine. You can also copy the short-cut and make a short-cut for each of your major "projects."

If you have more than 500 MB of memory (I highly recommend min 1 GB memory), you may also want to add the option to increase the memory available on the short cut. This is accomplished by adding a parameter to the command line which invokes R. Again, right click the short-cut and modify the command in the target line. I've set mine to:

```
"C:\Program Files\R\rw2010\bin\Rgui.exe" --max-mem-size=1900Mb
```

Note the closed-quote after Rgui.exe BEFORE the specification of the –max-mem-size parm.

You may also copy this shortcut to your "taskbar" – don't forget to "unlock" it first!

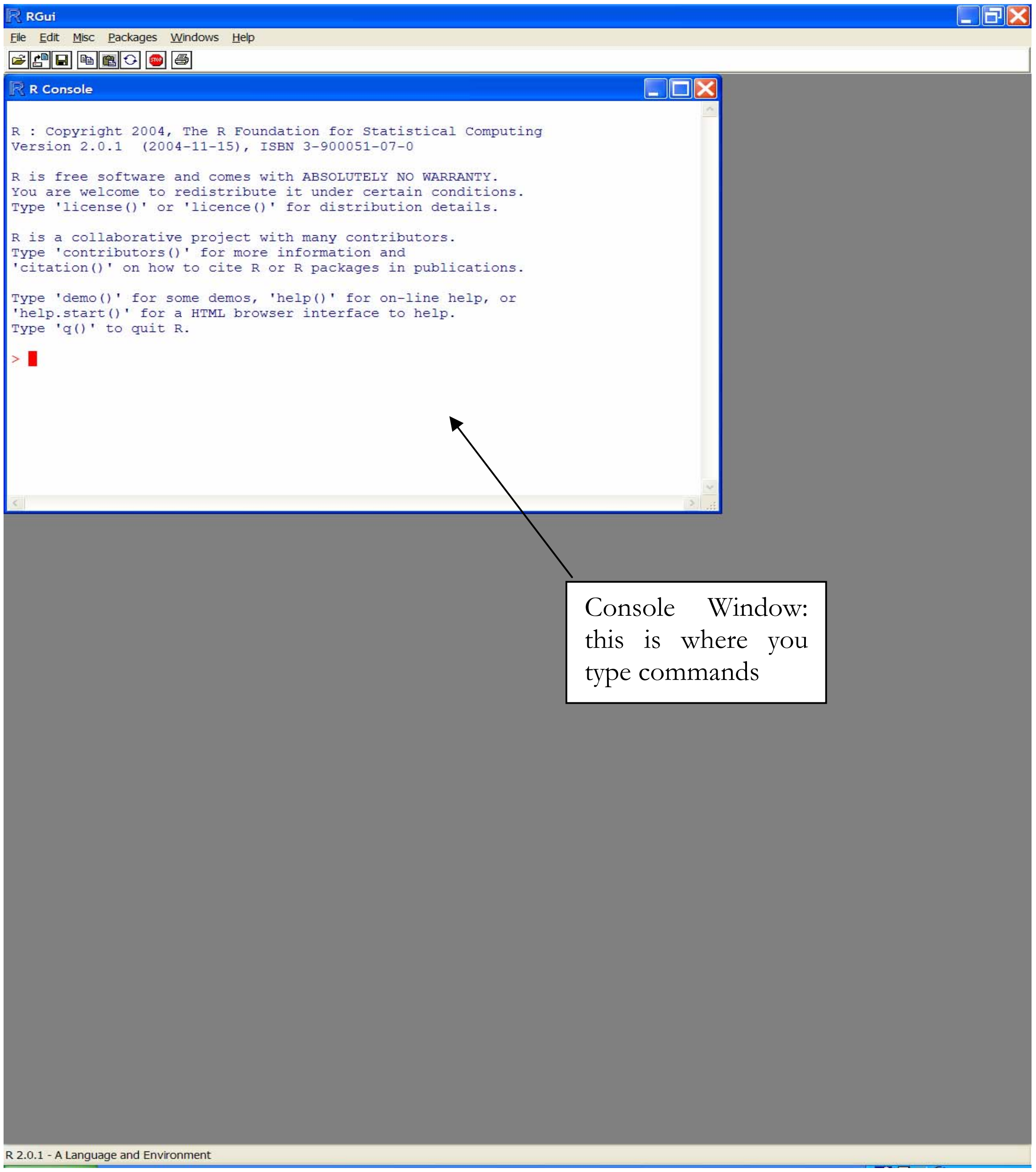
Invoking R and Using the R GUI

Click the short-cut to invoke R. Something like the screen shot below will appear.

File Menu allows you to change directories, "source" or read in files of commands to execute.

Packages Menu allows you to select contributed packages to download and install automatically

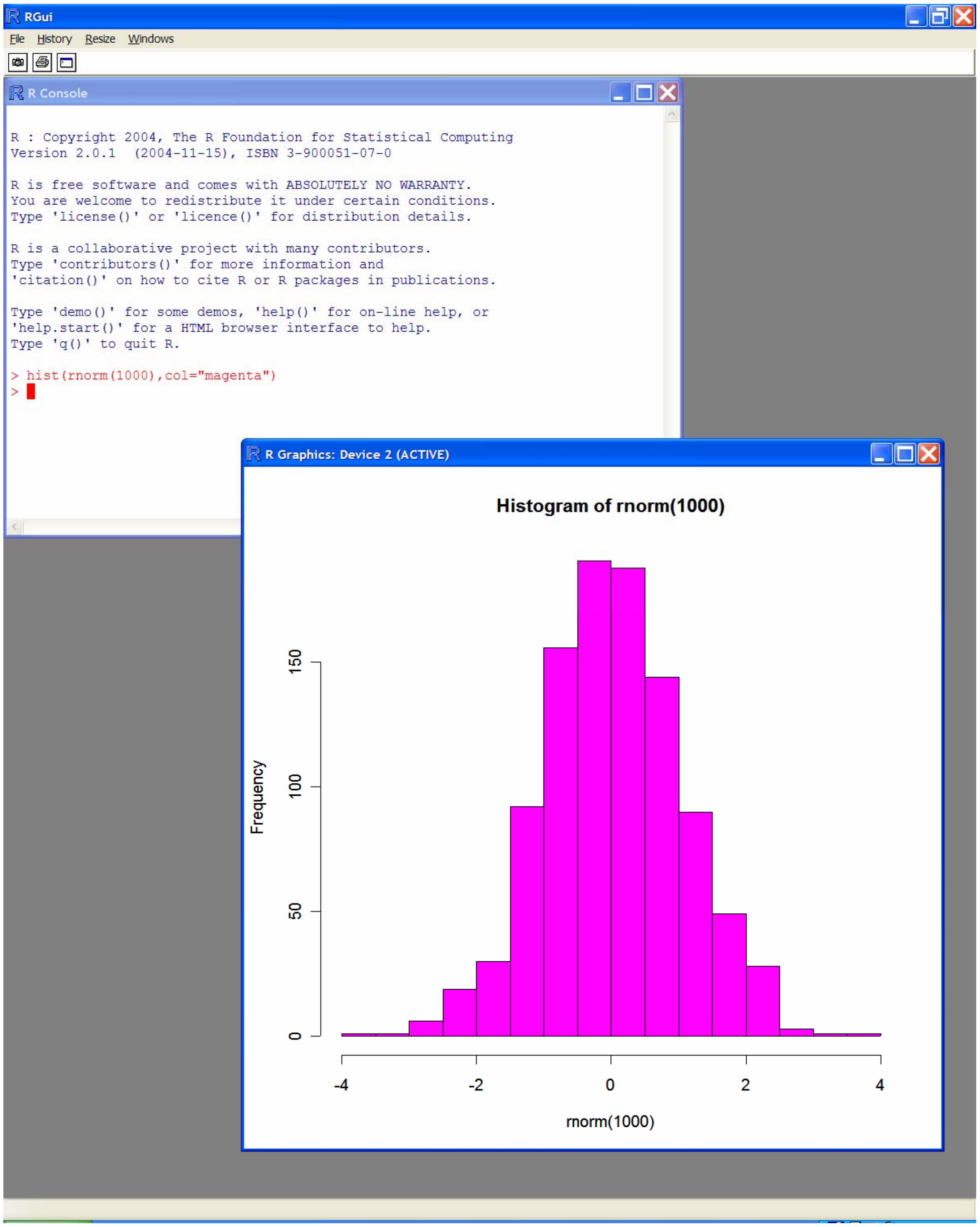
Help Menu allows you access to all of the R manuals in PDF or HTML form.


RGui
File Edit Misc Packages Windows Help
R Console
R 2.0.1 - A Language and Environment

```
R : Copyright 2004, The R Foundation for Statistical Computing
Version 2.0.1  (2004-11-15), ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for a HTML browser interface to help.
Type 'q()' to quit R.

>
```

Console Window: this is where you type commands

If you create graphics, a separate graphics window will appear within the RGUI main window. For example, if we create a histogram of 1000, normal random numbers by using the `hist` command (more below in graphics section). A window will automatically open with the plot. (It is possible to create multiple graphics windows and "paint" different graphics in each – see commands `dev.cur(), dev.list(), dev.set()`)



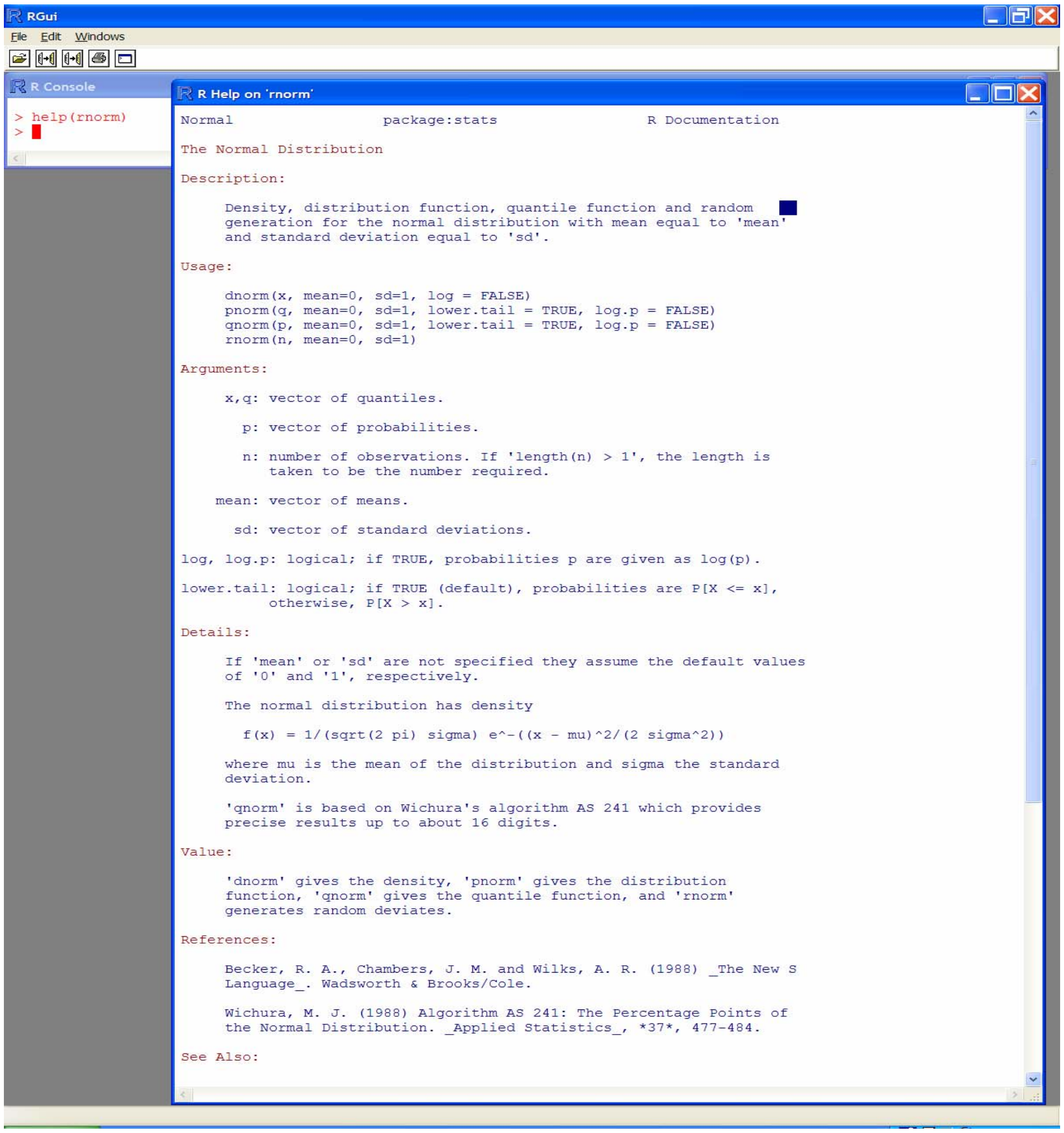
```
R : Copyright 2004, The R Foundation for Statistical Computing
Version 2.0.1  (2004-11-15), ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for a HTML browser interface to help.
Type 'q()' to quit R.

> hist(rnorm(1000),col="magenta")
> 
```

The contents of this graphics window can be "clipped" into the clipboard for pasting into your favorite application by selecting the graph window and using Ctl-W (not Ctl-C) and the paste or Ctl-V to paste into the app.

Obtaining Help in R

There are a number of different ways to obtain help in R. The "Help" menu allows you to access the manuals in PDF or HTML form as well as to search for keywords. Note: the help menu does not appear in the R GUI unless the console window is "active." To make any window active, click on the "button bar" or the blue bar on the top of the window.

You can also use the `help` and `help.search` commands in the R console windows. For example, we just used two commands, `hist` and `rnorm`. `help(rnorm)` produces a window with the following content (a short cut is the command `?rnorm`):



```
> help(rnorm)
>
```

```
Normal                  package:stats                  R Documentation

The Normal Distribution

Description:

     Density, distribution function, quantile function and random
     generation for the normal distribution with mean equal to 'mean'
     and standard deviation equal to 'sd'.

Usage:

     dnorm(x, mean=0, sd=1, log = FALSE)
     pnorm(q, mean=0, sd=1, lower.tail = TRUE, log.p = FALSE)
     qnorm(p, mean=0, sd=1, lower.tail = TRUE, log.p = FALSE)
     rnorm(n, mean=0, sd=1)

Arguments:

     x,q: vector of quantiles.

       p: vector of probabilities.

       n: number of observations. If 'length(n) > 1', the length is
          taken to be the number required.

    mean: vector of means.

      sd: vector of standard deviations.

log, log.p: logical; if TRUE, probabilities p are given as log(p).

lower.tail: logical; if TRUE (default), probabilities are P[X <= x],
          otherwise, P[X > x].

Details:

     If 'mean' or 'sd' are not specified they assume the default values
     of '0' and '1', respectively.

     The normal distribution has density

       f(x) = 1/(sqrt(2 pi) sigma) e^-((x - mu)^2/(2 sigma^2))

     where mu is the mean of the distribution and sigma the standard
     deviation.

     'qnorm' is based on Wichura's algorithm AS 241 which provides
     precise results up to about 16 digits.

Value:

     'dnorm' gives the density, 'pnorm' gives the distribution
     function, 'qnorm' gives the quantile function, and 'rnorm'
     generates random deviates.

References:

     Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) _The New S
     Language_. Wadsworth & Brooks/Cole.

     Wichura, M. J. (1988) Algorithm AS 241: The Percentage Points of
     the Normal Distribution. _Applied Statistics_, *37*, 477-484.

See Also:
```

Each R help window has the same sections:

- Description
- Usage
- Arguments
- Details
- Value
- References
- See Also
- Examples (note this is "cut-off" in the screen shot above)

Usage/Arguments/Examples are the most useful.

If you are not sure what command you need, `help.search("key word")` can be very useful.

Program Editing and R

For all but the simplest tasks, it is useful to edit a file with R commands in it. R syntax is sufficiently complex that it is difficult to write directly into the command window without making numerous syntax errors.

Open a text editor (VIM – improved VI is recommended; this is shareware, http://www.vim.org/); type in R commands and save the file. You can either cut the commands from the editor window and paste into the R console window to run or use the R command, `source` – also available from the File menu in the Windows GUI.

Customizing the R session

When R is invoked, R looks for a file `.Rprofile` in the directory that R is pointing at. If this file is not found, R then looks for the file in the directory specified by the environmental variable HOME or R_USER. To set HOME or R_USER environmental variable, right click on **My Computer** icon on desktop, select properties, selected **advanced** tab, environment variables and add new variables to the user.

Typically, the `.Rprofile` file is used to source various files which contain customized function definitions of use either for a particular project or for a particular user. You can also use it to load an installed package using the `library` command, as in `library(bayesm)`.

For more information on R startup, see `help(Startup)`.

An Introductory Example: Reading in Data and Dataframes

R language is a functionally-oriented language. All commands are functions which act upon objects of various types. All commands produce objects as well.

The basic R command is of the form: object=function(object)

Functions can be composed to produce powerful (but sometimes hard-to-read) expressions. Users can define their own functions. Writing these user functions constitutes R-programming.

Let's start by reading in some data. Suppose we have a file in a spreadsheet that with some regression data on several different units. The file has a UNIT variable to identify which unit the data comes from, a dependent variable Y, and two independent variables X1, X2.

UNIT	Y	X1	X2
A	1	0.23815	0.4373
A	2	0.55508	0.47938
A	3	3.03399	-2.17571
A	4	-1.49488	1.66929
B	10	-1.74019	0.35368
B	9	1.40533	-1.2612
B	8	0.15628	-0.27751
B	7	-0.93869	-0.0441
B	6	-3.06566	0.14486

We write this data out of Excel by saving it as a text (tab-delimited file), data.txt (use the **save as** option on the **file** menu and choose text file in the file type box). Note: there is no simple, direct way to read .XLS files in R[1].

We can read this file into R using the `READ.TABLE` command.

```
> df=read.table("data.txt",header=TRUE)
> df
  UNIT  Y       X1       X2
1    A  1  0.23815  0.43730
2    A  2  0.55508  0.47938
3    A  3  3.03399 -2.17571
4    A  4 -1.49488  1.66929
5    B 10 -1.74019  0.35368
6    B  9  1.40533 -1.26120
7    B  8  0.15628 -0.27751
8    B  7 -0.93869 -0.04410
9    B  6 -3.06566  0.14486
```

[1] Another option is to select a portion of a worksheet in Excel, copy this into the clipboard and use the command, `df=read.table(file="clipboard",header=TRUE)`.

The `read.table` function has two arguments: the name of the file, and the argument "header." There are many other arguments but they are optional and often have defaults. The default for the header argument is the value FALSE.

Using the argument, `header=TRUE`, tells the read.table function to expect that the first line of the file will contain (delimited by spaces or tabs) the names of each variable.

TRUE and FALSE are examples of reserved values in R indicating a logical switch for true or false. Another useful reserved value is NULL which is often used to create an object with nothing in it.

The command `df=read.table(…)` assigns the output of the read.table function to the R object named "df."

`df` is a member of a class or type of object called a data frame. A "data frame" is preferred by R as the format for datasets. A data frame contains a set of observations on variables with the same number of observations in each variable. In this example, each of the variables, Y, X1, and X2, is of type numeric (R does not distinguish between integers and floating point numbers), while the variable UNIT is character.

There are two reasons to store your data as a data frame: 1. most R statistical functions require a data frame and 2. the data frame object allows the user to access the data either via the variables names or by viewing the dataframe as a two-dimensional array.

```
> df$Y
[1]  1  2  3  4 10  9  8  7  6
> mode(df$Y)
[1] "numeric"
> df[,2]
[1]  1  2  3  4 10  9  8  7  6
```

We can refer to the Y variable in df by using the df$XXX notation (where XXX is the name of the variable). The "mode" command confirms that this variable is, indeed, numeric.

We can also access the Y variable by using notation in R for subsetting a portion of an array.

The notation `df[,2]` means the values of the 2nd column of df. Below we will explore the many ways we can subset an array or matrix.

Using Built-In Functions: Running a regression

Let's now use the built-in linear model function in R to run a regression of Y on X1 and X2, pooled across both units A and B.

```
> lmout=lm(Y ~ X1 + X2, data=df)
> names(lmout)
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"
> print(lmout)

Call:
lm(formula = Y ~ X1 + X2, data = df)

Coefficients:
(Intercept)           X1           X2
      5.084       -1.485       -2.221
> summary(lmout)

Call:
lm(formula = Y ~ X1 + X2, data = df)

Residuals:
    Min      1Q  Median      3Q     Max
-3.3149 -2.4101  0.4034  2.5319  3.2022

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   5.0839     1.0194   4.987  0.00248 **
X1           -1.4851     0.8328  -1.783  0.12481
X2           -2.2209     1.3820  -1.607  0.15919
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1

Residual standard error: 2.96 on 6 degrees of freedom
Multiple R-Squared: 0.3607,     Adjusted R-squared: 0.1476
F-statistic: 1.693 on 2 and 6 DF,  p-value: 0.2612
```

`lm` is the function in the package (stats) which fits linear models. Note that the regression is specified via a "formula" that tells `lm` which is the dependent and independent variables.

We assign the output from the `lm` function to the object, `lmout`. `lmout` is a special type of object called a "list." A list is simply an ordered collection of objects of any type.

The names command will list the names of the elements of the list. We can access any element of the list by using the $ notation.

```
> lmout$c
(Intercept)          X1          X2
   5.083871   -1.485084   -2.220859
```

Note: that we only need to specify enough of the name of the list component to uniquely identify it, e.g. `lmout$c` is the same as `lmout$coefficients`.

We can "print" the object `lmout` and get a brief summary of it's contents. Print is a generic command which uses a different "print method" for each type of object. Print recognizes that `lmout` is a list of type lm and uses a specific routine to printout the contents of the list.

A more useful summary of contents of `lmout` can be obtained with the `summary` command.

Inspecting objects and the R workspace

When you start up R, R looks for a file .Rdata in the directory in which R is started from (you can also double-click the file to start R). This file contains a copy of the R "workspace" which is a list of R objects created by the user. For example we just created two R objects in the example above: `df` (the data frame) and `lmout,` the `lm` output object.

To list all objects in the current workspace, use the command `ls()`[2].

```
> ls()
[1] "df"    "lmout"
```

This doesn't tell us too much about the objects. If you just type the object name at the command prompt and return, then you will invoke the default print method for this type of object as we saw above in the data frame example.

As useful command is the `structure` (`str` for short) command.

```
> str(df)
`data.frame':   9 obs. of  4 variables:
 $ UNIT: Factor w/ 2 levels "A","B": 1 1 1 1 2 2 2 2 2
 $ Y   : int  1 2 3 4 10 9 8 7 6
 $ X1  : num   0.238  0.555  3.034 -1.495 -1.740 ...
 $ X2  : num   0.437  0.479 -2.176  1.669  0.354 ...
```

Note that the str command tells us a bit about the variables in the data frame. The UNIT variable is of type "factor" with two levels. Type "factor" is used by many of the built-in R functions and is way to store qualitative variables. A "factor" is usually an identifier of some sort of classification of the observation, e.g. which "UNIT" or which state or which store … `levels()` gives a list of the unique values of this variable. `as.factor` can be used to convert a standard numeric or character vector into a factor.

[2] Note you can specify regular expressions as an argument to `ls` so that you can specify only object whose names match these patterns, e.g. to list all objects whose name begins with a, `ls(pat="^a")`, or all objects whose name includes the string "gibbs," `ls(pat="gibbs")`. See `?regex` for more.

To remove objects, use the `rm()` function. To remove a list of objects, use ls to create the list as follows:

```
rm(list=ls(pat=”*”))
```

This will remove all objects (except the ones with names starting with .), so be careful!

The R workspace exists only in memory. You must either save the workspace when you exist (you will be prompted for this) or you must recreate the objects again.

Vectors, Matrices and Lists

From our point of view, the power of R comes from statistical programming at a relatively high level. To do so, we will need to organize data as vectors, arrays and lists.

Vectors are ordered collections of the same type of object. If we access one variable from our data frame above, it will be a vector.

```
> df$X1
[1]  0.23815  0.55508  3.03399 -1.49488 -1.74019  1.40533  0.15628 -
0.93869 -3.06566
> length(df$X1)
[1] 9
> is.vector(df$X1)
[1] TRUE
```

The function `is.vector` returns a logical flag as to whether or not the input argument is a vector.

We can also create a vector with the `c()` command.

```
> vec=c(1,2,3,4,5,6)
> vec
[1] 1 2 3 4 5 6
> is.vector(vec)
[1] TRUE
```

A matrix is a two dimensional array.

Let’s create a matrix from a vector.

```
> mat=matrix(c(1,2,3,4,5,6),ncol=2)
> mat
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

`matrix()` is a command to create a matrix from a vector. `ncol` is a option to create the matrix with a specified number of columns (see also `nrow`). Note that the matrix is created column by column from the input vector (first subscripts varies the fastest). We can also create a matrix row by row.

```
> mat=matrix(c(1,2,3,4,5,6),byrow=TRUE,ncol=2)
> mat
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

Matrices are simply vectors with the appropriate "dim" attribute (objects in R have attributes or other objects attached to them). We can also create the matrices by changing the dim attribute.

```
> mat=c(1:6)
> dim(mat)=c(2,3)
> mat
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

We can also convert a data frame into a matrix.

```
> dfmat=as.matrix(df)
> dfmat
  UNIT Y    X1         X2
1 "A"  " 1" " 0.23815" " 0.43730"
2 "A"  " 2" " 0.55508" " 0.47938"
3 "A"  " 3" " 3.03399" "-2.17571"
4 "A"  " 4" "-1.49488" " 1.66929"
5 "B"  "10" "-1.74019" " 0.35368"
6 "B"  " 9" " 1.40533" "-1.26120"
7 "B"  " 8" " 0.15628" "-0.27751"
8 "B"  " 7" "-0.93869" "-0.04410"
9 "B"  " 6" "-3.06566" " 0.14486"
> dim(dfmat)
[1] 9 4
```

Note that all of the values of the results matrix are character as one of the variables in the data frame (UNIT) is character-valued.

We can also create matrices from other matrices and vectors using the `cbind` (column bind) and `rbind` (row bind) commands.

```
> mat1
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
> mat2
     [,1] [,2]
[1,]    7   10
[2,]    8   11
```

```
[3,]    9   12
> cbind(mat1,mat2)
     [,1] [,2] [,3] [,4]
[1,]    1    2    7   10
[2,]    3    4    8   11
[3,]    5    6    9   12
> rbind(mat1,mat2)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7   10
[5,]    8   11
[6,]    9   12
> rbind(mat1,c(99,99))
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]   99   99
```

R supports multi-dimensional arrays as well. Below is an example of creating a three dimensional array from a vector.

```
> ar=array(c(1,2,3,4,5,6),dim=c(3,2,2))
> ar
, , 1

     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

, , 2

     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Again, the array is created by using vector for the first dimension, then the second, and then third. A 3 x 2 x 2 array as 12 elements not the six provided as an argument. R will repeat the input vector as necessary until the required number of elements are obtained.

A list is an ordered collection of objects of any type. It is the most flexible object in R that can be indexed. As we have seen in the `lm` function output, lists can also have names.

```
> l=list(1,"a",c(4,4),list(FALSE,2))
> l
[[1]]
[1] 1

[[2]]
```

```
[1] "a"

[[3]]
[1] 4 4

[[4]]
[[4]][[1]]
[1] FALSE

[[4]][[2]]
[1] 2

> l=list(num=1,char="a",vec=c(4,4),list=list(FALSE,2))
> l$num
[1] 1
> l$list
[[1]]
[1] FALSE

[[2]]
[1] 2
```

In the example, we created a list of a numeric value, character, vector and another list. We also can name each component and access them with the $ notation.

Accessing Elements and Subsetting Vectors, Arrays, and Lists

To access an element of a vector, simply enclose index of that element in square brackets.

```
> vec=c(1,2,3,2,5)
> vec[3]
[1] 3
```

To access a sub-set of elements, there are two approaches: 1. specify a vector of integers of the required indices, 2. specify a logical variable which is TRUE for the desired indices.

```
> index=c(3:5)
> index
[1] 3 4 5
> vec[index]
[1] 3 2 5
> index=vec==2
> index
[1] FALSE  TRUE FALSE  TRUE FALSE
> vec[index]
[1] 2 2
> vec[vec!=2]
[1] 1 3 5
```

`c(3:5)` creates a vector from the "pattern" or sequence from 3 to 5. The `seq` command can create a wide variety of different patterns.

To properly understand the example of the logical index, it should be noted that "=" is an assignment operator while "==" is a comparison operator. Vec==2 creates a logical vector with flags for if the elements of vec are 2[3]. The last example uses the "not equal" comparison operator !=.

We can also access the elements not in a specified index vector.

```
> vec[-c(3:5)]
[1] 1 2
```

To access elements of arrays, we can use the same ideas for vectors but we must specify a set of row and column indices. If no indices are specified, we get all of the elements on that dimension. For example, earlier we used the notation `df[,2]` to access the second column of the data frame `df`.

We can pull off the observations corresponding to unit "A" from the matrix version of dfmat using the commands:

```
> dfmat
  UNIT Y    X1         X2
1 "A"  " 1" " 0.23815" " 0.43730"
2 "A"  " 2" " 0.55508" " 0.47938"
3 "A"  " 3" " 3.03399" "-2.17571"
4 "A"  " 4" "-1.49488" " 1.66929"
5 "B"  "10" "-1.74019" " 0.35368"
6 "B"  " 9" " 1.40533" "-1.26120"
7 "B"  " 8" " 0.15628" "-0.27751"
8 "B"  " 7" "-0.93869" "-0.04410"
9 "B"  " 6" "-3.06566" " 0.14486"
> dfmat[dfmat[,1]=="A",2:4]
  Y    X1         X2
1 " 1" " 0.23815" " 0.43730"
2 " 2" " 0.55508" " 0.47938"
3 " 3" " 3.03399" "-2.17571"
4 " 4" "-1.49488" " 1.66929"
```

The result is a 4 x 3 matrix. Note that we are using the values of the dfmat to index into itself. This means that R evaluates the expression `dfmat[,1] == "A"` and passes the result into the matrix subsetting operator `[ ]` which is a function that processes `dfmat`.

To access elements of lists, we can use the $ notation if the element has a name or we can use a special operator `[[ ]]`. To see how this works, let's make a list with two elements, each corresponding to the observations for unit A and B.

[3] It is sometimes desired to obtain the indices for which a logical expression is true. The function which returns these indices, e.g. `which(vec==2)` will return a vector of (2,4) in the example here.

Note that the size of the matrices in correspond to each unit is different – unit A has four obs and unit B has five! This means that we can't use a three dimensional array to store this data (we need would need a "ragged" array).

```
> ldata=list(A=dfmat[dfmat[,1]=="A",2:4],B=dfmat[dfmat[,1]=="B",2:4])
> ldata
$A
  Y    X1         X2
1 " 1" " 0.23815" " 0.43730"
2 " 2" " 0.55508" " 0.47938"
3 " 3" " 3.03399" "-2.17571"
4 " 4" "-1.49488" " 1.66929"

$B
  Y    X1         X2
5 "10" "-1.74019" " 0.35368"
6 " 9" " 1.40533" "-1.26120"
7 " 8" " 0.15628" "-0.27751"
8 " 7" "-0.93869" "-0.04410"
9 " 6" "-3.06566" " 0.14486"

> ldata[1]
$A
  Y    X1         X2
1 " 1" " 0.23815" " 0.43730"
2 " 2" " 0.55508" " 0.47938"
3 " 3" " 3.03399" "-2.17571"
4 " 4" "-1.49488" " 1.66929"

> is.matrix(ldata[1])
[1] FALSE
> is.list(ldata[1])
[1] TRUE
> ldata$A
  Y    X1         X2
1 " 1" " 0.23815" " 0.43730"
2 " 2" " 0.55508" " 0.47938"
3 " 3" " 3.03399" "-2.17571"
4 " 4" "-1.49488" " 1.66929"
> is.matrix(ldata$A)
[1] TRUE
```

If we specify `ldata[1]`, we don't get the contents of the list element (which is a matrix) but we get a list! If we specify `ldata$A`, we obtain the matrix. If we have a long list or we don't wish to name each element, we can use the `[[ ]]` operator to access elements in the list.

```
> is.matrix(ldata[[1]])
[1] TRUE
> ldata[[1]]
  Y    X1         X2
1 " 1" " 0.23815" " 0.43730"
2 " 2" " 0.55508" " 0.47938"
3 " 3" " 3.03399" "-2.17571"
4 " 4" "-1.49488" " 1.66929"
```

Loops

As with all interpreted languages, loops in R are slow. That is, they typically take more time than if implemented in a compiled language. On the other hand, matrix/vector operations are typically faster in R than in compiled language such as C and Fortran unless the optimized BLAS is called. Thus, wherever possible, "vectorization" or writing expressions as only involving matrix/vector arithmetic is desirable. This is more of an art than a science, however.

If a computation is fundamentally iterative (such as maximization or MCMC simulation), a loop will be required.

A simple loop can be accomplished with the `for` structure. The syntax is of the form

```
for (var in range) { }
```

`var` is a numeric loop index. `range` is a range of values of `var`. Enclosed in the braces is any valid R expression. There can be more than one R statement in the R expression.

Let's loop over both units and create a list of lists of the regression output from each.

```
> ldatadf=list(A=df[df[,1]=="A",2:4],B=df[df[,1]=="B",2:4])
> lmout=NULL
> for (i in 1:2) {
+    lmout[[i]]=lm(Y ~ X1+X2,data=ldatadf[[i]])
+    print(lmout[[i]])
+    }

Call:
lm(formula = Y ~ X1 + X2, data = ldatadf[[i]])

Coefficients:
(Intercept)           X1           X2
      4.494       -2.860       -3.180


Call:
lm(formula = Y ~ X1 + X2, data = ldatadf[[i]])

Coefficients:
(Intercept)           X1           X2
      9.309        1.051        1.981
```

Here we subset the data frame directly rather than the matrix created from the data frame to avoid the extra-step of converting character to numeric values and so that we can use the lm function which requires data frame input. We can see that the same sub-setting command that work on arrays will also work on data frames.

Implicit Loops

In many contexts, a loop is used to compute the results of applying a function to either the row or column dimensions of an array. For example, if we wish to find the mean of each variable in a data frame, we want to apply the function "mean" to each column. This can be done with the `apply()` function.

```
> apply(df[,2:4],2,mean)
         Y         X1         X2 
 5.5555556 -0.2056211 -0.0748900
```

The first argument specifies the array, the second the dimension (1=row, 2=col), and the third the function to be applied. In R, the apply function is simply an elegant loop so don't expect to speed things up with this. Of course, we could write this as a matrix operation which would be much faster.

If you want to apply a function to a list, `sapply` can be used. `sapply` will attempt to coerce the output of the function into a vector or array if possible. This is marginally faster than an explicit loop over the elements of the list but it is much more elegant! `sapply` is a simplified version of `lapply` (see also `mapply` for applying functions to multiply lists).

```
betas=sapply(regdata,myreg)
myreg=function(list) { statements to do regression here }
```

This is the same as

```
for (i in 1:regdata)
{
   betas[i,]=myreg(regdata[[i]])
}
```

`regdata` is a list of lists of regression (y, X) data. `myreg` is a function which takes a list of regression data and computes the least squares betahats. `betas` will be a `length(betahat)` x `length(regdata)` array.

Matrix Operations

One of the primary advantages of R is that we can write matrix/vector expressions directly in R code. Let's review some of these operators by computing a pooled regression using matrix statements.

The basic functions needed are:

`%*%`	matrix multiplication e.g. (X %*% Y) note: X or Y or both can be vectors
`chol(X)`	compute "square" or Cholesky root of square, pd matrix X=U'U where `U=chol(X)`; U is upper triangular
`chol2inv(chol(X))`	compute inverse of square pd matrix using its Cholesky root

`crossprod(X,Y)`	t(X) %*% Y -- very efficient
`diag`	extract diagonal of matrix or create diagonal matrix from a vector or create an identity matrix from an integer

Less frequently used are:

`%x%`	Kronecker product (to be used carefully as Kronecker products can create very large arrays)
`backsolve()`	used to compute inverse of a triangular array
`qr`	compute QR decomposition (set `LAPACK=TRUE` for speed)
`qr.coef(q,Y)`	Least Squares coefficients on Y using QR object q q is output of qr (i.e. `q=qr(X)`)

The R statements to compute the regression are:

```
y=as.numeric(dfmat[,2])
X=matrix(as.numeric(dfmat[,3:4]),ncol=2)
X=cbind(rep(1,nrow(X)),X)
XpXinv=chol2inv(chol(crossprod(X)))
bhat=XpXinv%*%crossprod(X,y)
res=as.vector(y-X%*%bhat)
ssq=as.numeric(res%*%res/(nrow(X)-ncol(X)))
se=sqrt(diag(ssq*XpXinv))
```

Note: you can also do this by first computing the root of X'X and then inverting this using backsolve. This will leave you a root of $(X'X)^{-1}$ which can then be used not only to compute regression coefficients but if you need to simulate from various distributions like the posterior distribution of beta or the sampling distribution of betahat. See BSM website for examples.

The first two statements create y and X. Then we add a column of ones for the intercept and compute the regression using Cholesky roots.

To create the vector of ones we use the `rep()` function.

Note that we must convert res to a vector to use the statement res %*% res. We also must convert ssq to a scalar from a 1 x 1 matrix to compute the standard errors in the last statement.

Note: the method above is very stable numerically but some would prefer the QR decomposition. This would be simpler but our experience has shown that the method above is actually faster in R.

Other Useful Built-In R Functions

R has thousands of built-in function and thousands more than can be added from contributed packages. Some functions that I use regularly (aside from the matrix operations above) include

`rnorm`	draw univariate normal random variates
`pnorm/qnorm/dnorm`	(cdf, inverse cdf, density)
`runif`	draw uniform random variates
`rchisq`	draw chi-sq random variates

`mean`	compute mean of a vector
`var`	compute Covariance matrix given matrix input
`quantile`	computes quantiles of a vector

`optim`	general purpose optimizer

`sort`	sort a vector

`seq`	create a sequence, e.g. `seq(1,100,by=.1)`

`unlist`	attempts to coerce a list into a vector
`as.integer`	
`as.double`	
`as.numeric`	
`is.[list,integer,double,matrix,list,dataframe]`	

`if`	standard if statement (includes else clause)
`ifelse`	vectorized if else statement
`while`	while loop

`scan`	read from a file to a vector
`write`	write a matrix to a file
`cat`	write expression to console, e.g. `cat("this is a test; i =",i,fill=TRUE)`
`print`	use default print method to print out object
`paste`	paste together two strings, e.g. `paste("A =",a,sep=" ")`

`sqrt`	square root
`log`	natural log
`%%`	modulo (e.g. 100%%10 = 0)
`round`	round to a specified number of sign digits
`floor`	greatest integer < argument

`dyn.load`	load a library for dynamic linking (more on this in a separate document)
`getLoadedDLLs`	find out current loaded DLLS
`is.loaded`	check if a symbol is loaded from a DLL
`.C`	interface to C and C++ code (more later)

User-defined Functions

The regression example above is a perfect situation for which a user-defined function would be useful.

To create a function object in R, simply enclose the R statements in braces as assign this to a function variable.

```
myreg=function(y,X){
#
# purpose: compute lsq regression
#
# arguments:
#    y -- vector of dep var
#    X -- array of indep vars
#
# output:
#    list containing lsq coef and std errors
#
XpXinv=chol2inv(chol(crossprod(X)))
bhat=XpXinv%*%crossprod(X,y)
res=as.vector(y-X%*%bhat)
ssq=as.numeric(res%*%res/(nrow(X)-ncol(X)))
se=sqrt(diag(ssq*XpXinv))
list(b=bhat,std_errors=se)
}
```

The code above should be executed either by cutting and pasting into R or by sourcing a file containing this code. This will define an object called "myreg"

```
ls()
 [1] "ar"           "bhat"         "df"           "dfmat"        "i"            "index"
 [7] "l"            "last.warning" "ldata"        "ldatadf"      "ldataidf"     "lmout"
[13] "mat"          "mat1"         "mat2"         "myreg"        "names"        "res"
[19] "se"           "ssq"          "vec"          "X"            "XpXinv"       "y"
```

To execute the function, we simply type it in with arguments at the command prompt or in another source file.

```
> myreg(X=X,y=y)
$b
          [,1]
[1,]  5.083871
[2,] -1.485084
[3,] -2.220859

$std_errors
[1] 1.0193862 0.8327965 1.3820287
```

`myreg` results a list with b and the standard errors.

Objects are passed by copy in R rather than by reference. This means that if I give the command `myreg(Z,d)` a copy of Z will be assigned to the "local" variable X in the function myreg and a copy of d to y. In addition, variables created in the function (e.g. XpXinv and res in myreg) are created only during the execution of the function and then erased when the function returns to the calling environment.

The arguments are passed and copied in the order supplied at the time of the call so that you must be careful! `myreg(d,Z)` will bomb.

If I explicitly name the arguments as in `myreg(X=Z,y=d)` then I can give the arguments in any order I desire.

Many functions have default arguments and R has what is called "lazy" function evaluation which means that if an argument is not needed it is not checked. See *Introduction to R* for a more discussion on default and other types of arguments.

If a local variable cannot be found while executing a function, R will look in the environment or workspace that the function was called from. This can be convenient but it can also be dangerous!

Many functions are dependent on other functions. If a function called within a function is only used by that calling function and has no other use, it can be useful to define these utility functions in the calling function definition. This means that they will not be visible to the user of the function.

Example:

```
Myfun= function(X,y) {
#
# define utilty function needed
#
Util=function(X) { … }
#
# main body of myfun
#
…
}
```

<u>Debugging Functions</u>

It is a good practice to define your functions in a file and "source" them into R. This will allow you to recreate your set of function objects for a given project without having to save the workspace.

To debug a function, you can use the brute force method of placing print statements in the function. `cat()` can be useful here. For example, we can define a "debugging" version of myreg which prints out the value of se in the function. The `cat` command prints out a statement reminding us of where the "print" output comes from (note the use of `fill=TRUE` which insures that a new line will be generated on the console).

```
myreg=function(y,X){
#
# purpose: compute lsq regression
#
# arguments:
#    y -- vector of dep var
#    X -- array of indep vars
#
# output:
#    list containing lsq coef and std errors
#
XpXinv=chol2inv(chol(crossprod(X)))
bhat=XpXinv%*%crossprod(X,y)
res=as.vector(y-X%*%bhat)
ssq=as.numeric(res%*%res/(nrow(X)-ncol(X)))
se=sqrt(diag(ssq*XpXinv))
cat("in myreg, se = ",fill=TRUE)
print(se)
list(b=bhat,std_errors=se)
}
```

When run, this new function will produce the output:

```
> myregout=myreg(y,X)
in myreg, se =
[1] 1.0193862 0.8327965 1.3820287
```

R also features a simple debugger. If you "debug" a function, you can step through the function and inspect the contents of local variables. One can also modify their contents.

```
> debug(myreg)
> myreg(X,y)
debugging in: myreg(X, y)
debug: {
    XpXinv = chol2inv(chol(crossprod(X)))
    bhat = XpXinv %*% crossprod(X, y)
    res = as.vector(y - X %*% bhat)
    ssq = as.numeric(res %*% res/(nrow(X) - ncol(X)))
    se = sqrt(diag(ssq * XpXinv))
    cat("in myreg, se = ", fill = TRUE)
    print(se)
    list(b = bhat, std_errors = se)
}
Browse[1]>
debug: XpXinv = chol2inv(chol(crossprod(X)))
Browse[1]> X
[1]  1  2  3  4 10  9  8  7  6
Browse[1]> #OOPS!
debug: bhat = XpXinv %*% crossprod(X, y)
Browse[1]> XpXinv
            [,1]
[1,] 0.002777778
Browse[1]> Q
> undebug(myreg)
```

If there are loops in the function, the debugging command “c” can be used to allow the loop to finish. “Q” quits the debugger. You must turn off the debugger with the undebug command! If you want to debug other functions called by `myreg`, you must `debug()` ‘em first!

Elementary Graphics

Graphics in R can be quite involved as the graphics capabilities are very extensive. For some examples of what is possible issue the commands `demo(graphics)`, `demo(image)` and `demo(persp)`.

We will only cover the bare minimum necessary to function.

Let’s return to our first example – a histogram of a distribution.

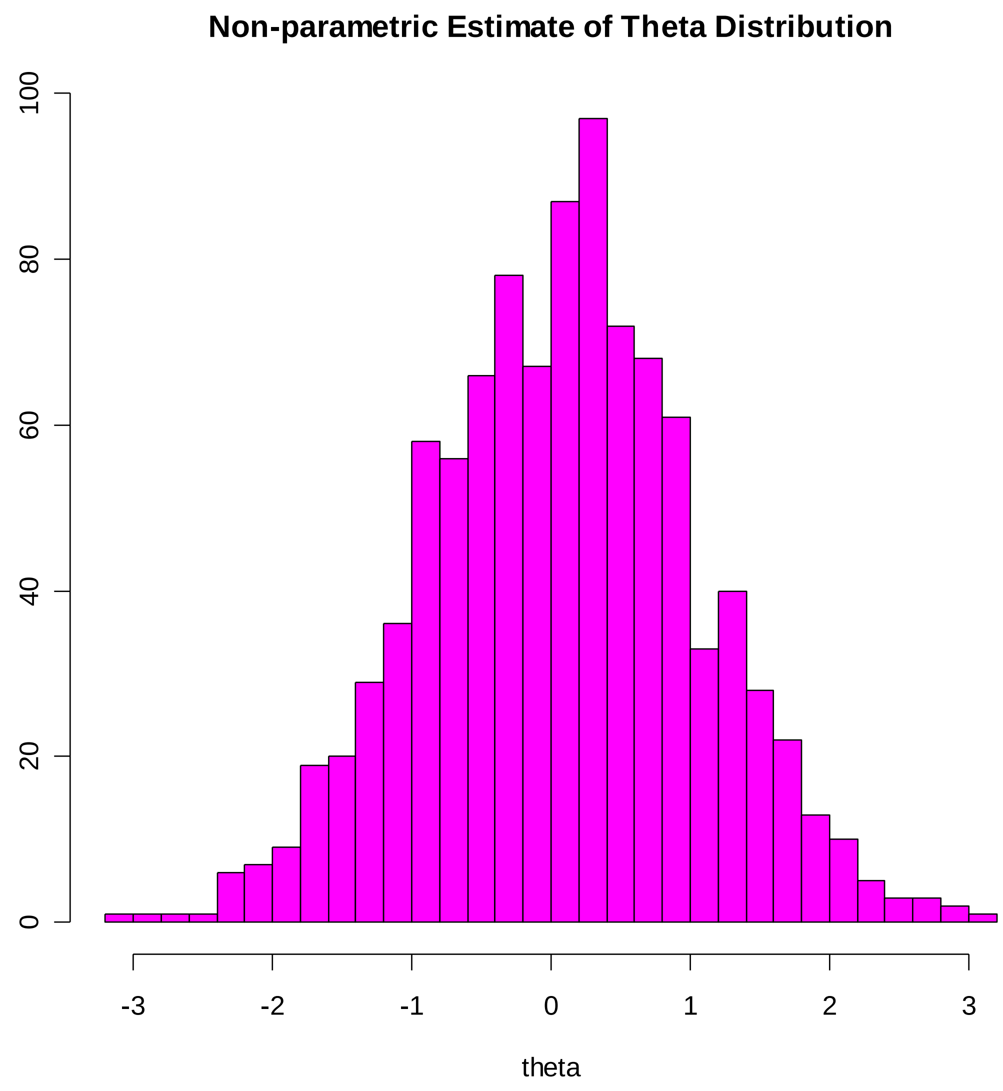
```
hist(rnorm(1000),breaks=50,col=”magenta”)
```

This creates a histogram with 50 bars and with each bar filled in the color “magenta” (type `colors()` to see the list of available colors).

This plot can be improved by inclusion of plot parameters to change the x and y axis labels and well as the “title” of the plot.

```
hist(rnorm(1000),breaks=30,col=”magneta”,xlab=”theta”,ylab=””,main=”Non
-parametric Estimate of Theta Distribution”)
```

produces



Three other basic plots are useful:

`plot(x,y)`	scatterplot of x vs y
`plot(x)`	sequence plot of x
`matplot(X)`	sequence plots of columns of X
`acf(x)`	acf of time series in x
`boxplot(data.frame(X))`	boxplots of data.frame created from X array each column is a plot

The col, xlab, ylab, and main parameters work on all of these plots.

In addition, the parameters

`type=“l”`	connects scatterplot points with a lines
`lwd=x`	specifies the width of lines (1 is default, > 1 is thicker)
`lty=x`	specifies type of line (e.g. solid vs dashed)
`xlim/ylim=c(z,w)`	specifies x/y axis runs from z to w

are useful. `?par` displays all of the graphic parameters available.

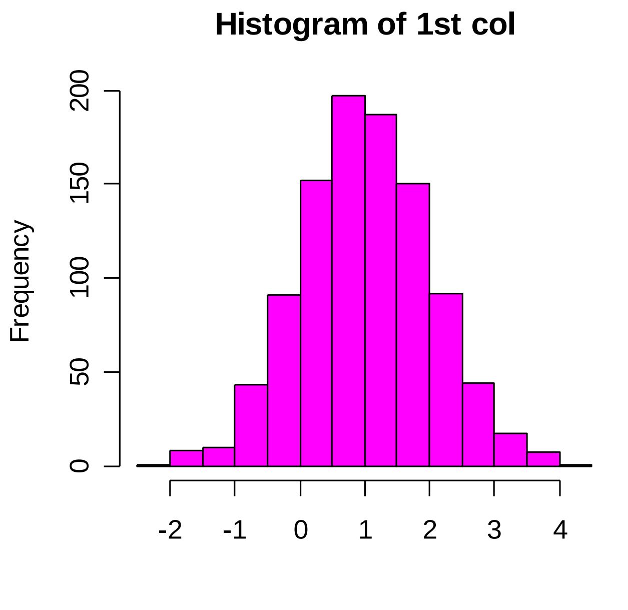
One useful notion is the idea to lay down the basic plot frame using a “plot” command and then add points and lines to it. `abline` adds a line to the current plot frame, `lines` and `points` will add multiple lines to the current frame. In the examples below, we use `abline` to add a line to a plot.

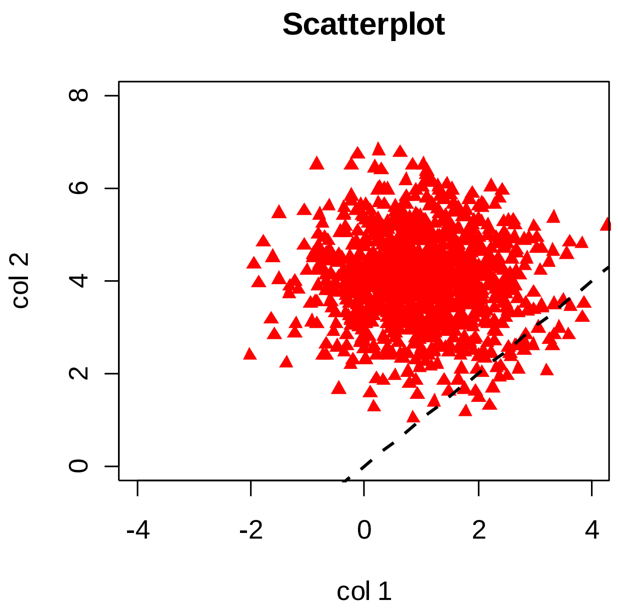
It is often useful to display more than one plot per page. To do this, we must change the global graphic parameters with the command, `par(mfrow=c(x,y))`. This specifies an array of plots x by y plotted row by row.

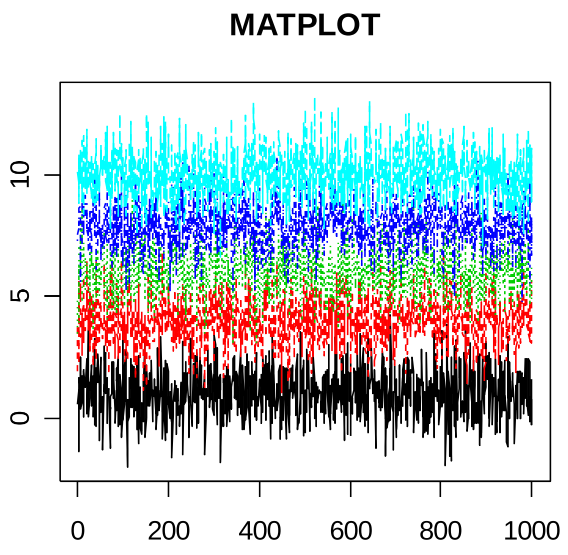
```
par(mfrow=c(2,2))
X=matrix(rnorm(5000),ncol=5)
X=t(t(X)+c(1,4,6,8,10))

hist(X[,1],main="Histogram of 1st col",col="magenta",xlab="")
plot(X[,1],X[,2],xlab="col 1", ylab="col 2",pch=17,col="red",
      xlim=c(-4,4),ylim=c(0,8))
title("Scatterplot")
abline(c(0,1),lwd=2,lty=2)
matplot(X,type="l",ylab="",main="MATPLOT")
acf(X[,5],ylab="",main="ACF of 5th Col")
```

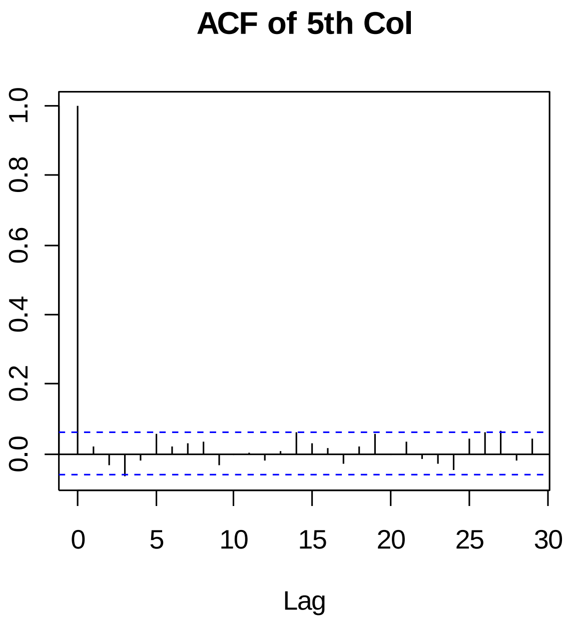
`title()` and `abline()` are examples of commands which modify the current “active” plot. Other userful functions are `points()` and `lines()` to add points and points connected by lines to the current plot.

The commands above will produce


Histogram of 1st col
Frequency
0
50
100
150
200
-2
-1
0
1
2
3
4


Scatterplot
col 2
col 1
0
2
4
6
8
-4
-2
0
2
4


MATPLOT
0
5
10
0
200
400
600
800
1000


ACF of 5th Col
Lag
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20
25
30

System Information

`memory.limit()`	current memory limit
`memory.size()`	current memory size
`system.time(R expression)`	times execution of R expression
`proc.time()[3]`	current R session cpu usage in seconds
`list.files()`	list files in current wd (accepts regular exp)
`getwd()`	obtain current working directory
`setwd()`	set current working directory
`Rprof(file=”filename”)`	turns on profiling and writes to filename
`Rprof(“”)`	turns off profiling
`summaryRprof(file=”filename”)`	summarizes output in profile file

Examples of usage are given below.

```
> memory.size()
[1] 191135504
> getwd()
[1] "C:/userdata/per/class/37904"
> x=matrix(rnorm(1e07),ncol=1000)
> memory.size()
[1] 332070456
> memory.limit()
[1] 1992294400
> begin=proc.time()[3]
> z=crossprod(x)
> end=proc.time()[3]
> print(end-begin)
[1] 6.59
>test=function(n){x=matrix(rnorm(n),ncol=1000);z=crossprod(x);
cz=chol(z)}
> Rprof("test.out")
> test(1e07)
> Rprof()
> summaryRprof("test.out")
$by.self
          self.time self.pct total.time total.pct
rnorm          4.40     48.9       4.40      48.9
crossprod      4.16     46.2       4.16      46.2
matrix         0.22      2.4       4.72      52.4
.Call          0.12      1.3       0.12       1.3
as.vector      0.10      1.1       4.50      50.0
chol           0.00      0.0       0.12       1.3
test           0.00      0.0       9.00     100.0

$by.total
          total.time total.pct self.time self.pct
test            9.00     100.0      0.00      0.0
```

```
matrix          4.72      52.4      0.22      2.4
as.vector       4.50      50.0      0.10      1.1
rnorm           4.40      48.9      4.40     48.9
crossprod       4.16      46.2      4.16     46.2
.Call           0.12       1.3      0.12      1.3
chol            0.12       1.3      0.00      0.0

$sampling.time
[1] 9
```

The profile shows that virtually all of the time in the test function was in the generation of normal random numbers and in computing cross-products. The Cholesky root of a 1000 x 1000 matrix is essentially free! `crossprod` is undertaking 5 billion floating point multiply operations (1/2 of 10,000 x 1,000*1,000).

<u>More Lessons Learned from Timing</u>

If you are going to fill up an array with results, pre-allocate space in the array. Do not append to an existing array.

```
> n=1e04
> x=NULL
> zero=c(rep(0,5))
> begin=proc.time()[3]
> for (i in 1:n) {x=rbind(x,zero) }
> end=proc.time()[3]
> print(end-begin)
[1] 6.62
> x=NULL
> begin=proc.time()[3]
> x=matrix(double(5*n),ncol=5)
> end=proc.time()[3]
> print(end-begin)
[1] 0.07
```