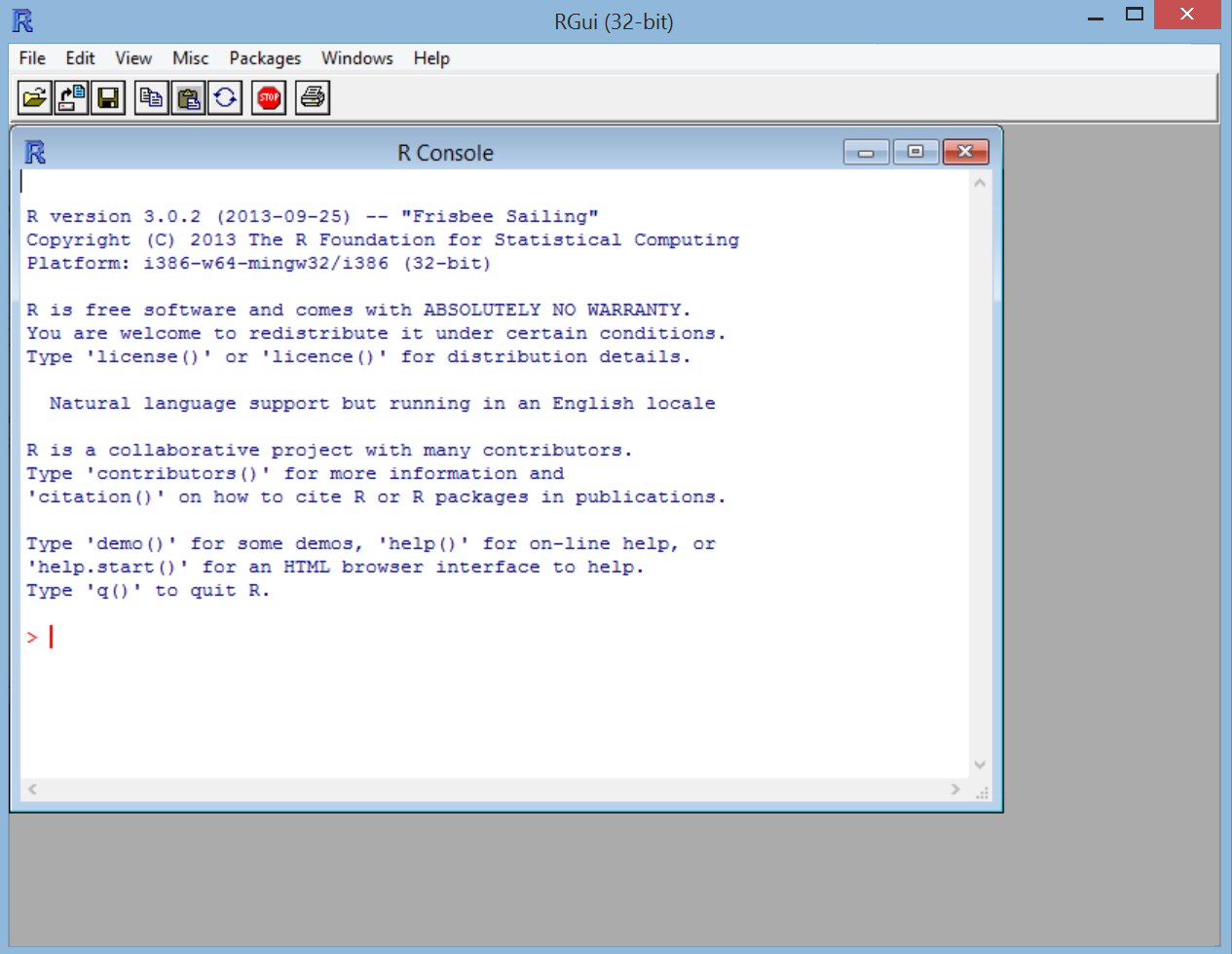
**Introduction to R**

Appendix A of my “Analysis of Categorical Data with R” book contains much of the same content as below. Please note that some of the wording is the same.

The R installation file for Windows can be downloaded from <http://cran.r-project.org/bin/windows/base/>. Select the “Download R 3.\*.\* for Windows” link. You can simply execute the file on your computer to install (all the installation defaults are o.k. to use). Both a 32-bit and 64-bit version of R will be installed.

Basics of R

The R Console window is where commands are typed.



The Console can be used like a calculator. Below are some examples:

> 2+2

[1] 4

> qchisq(0.95,1)

[1] 3.841459

> pnorm(1.96)

[1] 0.9750021

> (2-3)/6

[1] -0.1666667

> 2^2

[1] 4

> sin(pi/2)

[1] 1

> log(1)

[1] 0

Results from these calculations can be stored in an *object*. The <- is used to make the assignment and is read as “gets”.

> save<-2+2

> save

[1] 4

The objects are stored in R’s database. When you close R you will be asked if you would like to save or delete them. This is kind of like the SAS WORK library, but R gives you the choice to save them.

To see a listing of the objects, you can do either of the following:

> ls()

[1] "save"

> objects()

[1] "save"

To delete an object, use rm() and insert the object name in the parentheses.

Functions

R performs calculations using functions. For example, the qchisq() and the pnorm()commands used earlier are functions. Writing your own function is fairly simple. For example, suppose you want to write a function to calculate the standard deviation. Below is an example where 5 observations are saved to an *object* using the *concatenate* or *combine* function c(). A function called sd2() is written to find the standard deviation simply by using the square root of the variance. The sd2 object is now stored in the R database.

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

> sd2<-function(numbers) {

sqrt(var(numbers))

}

> sd2(x)

[1] 1.581139

Note that there already is a function in R to calculate the standard deviation, and this function is sd().

When a function has multiple lines of code in it, the last line corresponds to the returned value. For example,

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

> sd2<-function(numbers) {

cat("Print the data \n", numbers, "\n")

sqrt(var(numbers))

}

> save<-sd2(x)

Print the data   
1 2 3 4 5

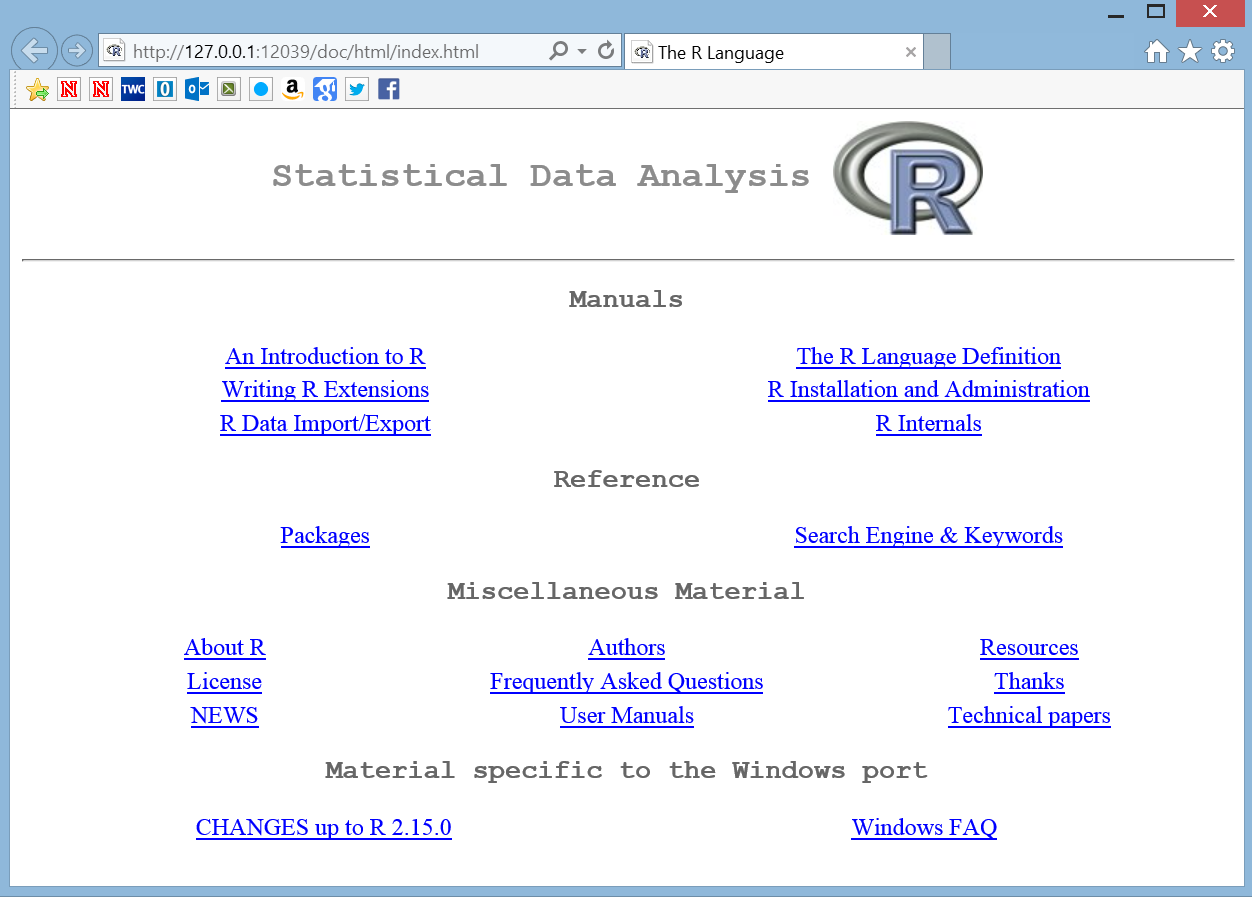
> save

[1] 1.581139

Note that the cat() function is used to print text and the \n character tells R to go to a new line.

Help

To see a listing of all R functions which are “built in”, open the Help by selecting HELP > HTML HELP from the main R menu bar.



Under REFERENCE, select the link called PACKAGES. All built in R functions are stored in a package.



We have been using functions from the base and stats package. By selecting stats, you can scroll down to find help on the pnorm() function. Note the full syntax for pnorm() is

pnorm(q, mean=0, sd=1, lower.tail = TRUE, log.p =

FALSE)

The q value corresponds to the 1.96 that was entered earlier. So

> pnorm(1.96)

[1] 0.9750021

> pnorm(q = 1.96)

[1] 0.9750021

> pnorm(q = 1.96, mean = 0, sd = 1)

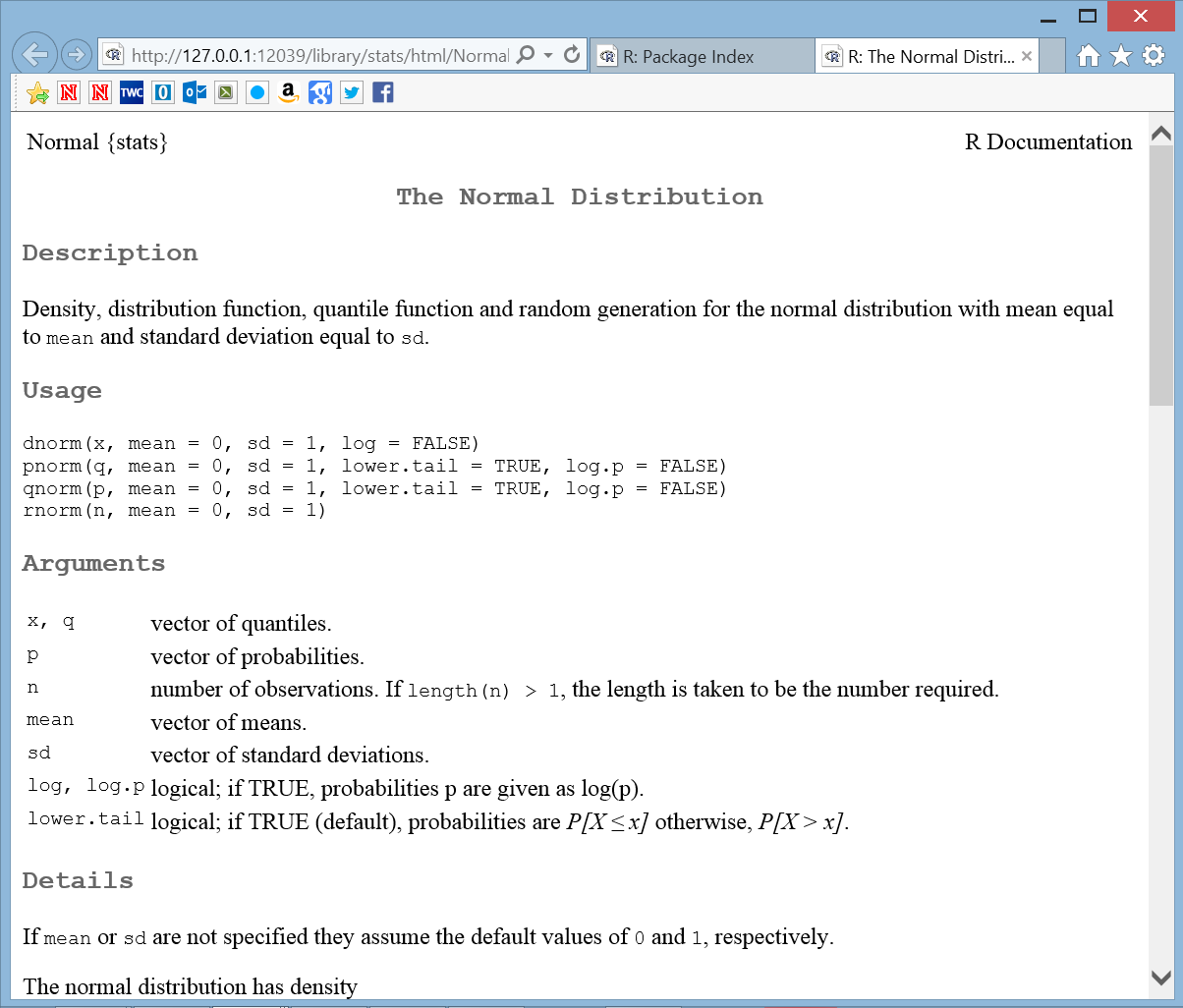
[1] 0.9750021

all produce the same results. The other entries in the function have default values set. For example, R assumes you want to work with the standard normal distribution by assigning mean = 0 and sd = 1 (standard deviation).

If you know the exact name of the function, simply type help(function name) at the R Console command prompt to open its help. For example,

> help(pnorm)

results in



Using R functions on vectors

Many R functions are set up to work directly on vectors. For example,

> pnorm(q = c(-1.96,1.96))

[1] 0.02499790 0.97500210

> qt(p = c(0.025, 0.975), df = 9)

[1] -2.262157 2.262157

The qt() function finds the 0.025 and 0.975 quantiles from a t-distribution with 9 degrees of freedom. As another example, suppose I want to find a 95% confidence interval for a population mean:

> x<-c(3.68, -3.63, 0.80, 3.03, -9.86, -8.66, -2.38,

8.94, 0.52, 1.25)

> x

[1] 3.68 -3.63 0.80 3.03 -9.86 -8.66 -2.38 8.94

0.52 1.25

> mean(x) + qt(p = c(0.025, 0.975), df =

length(x)-1) \* sd(x)/sqrt(length(x))

[1] -4.707033 3.445033

> t.test(x = x, mu = 2, conf.level = 0.95)

One Sample t-test

data: x

t = -1.4602, df = 9, p-value = 0.1782

alternative hypothesis: true mean is not equal to 2

95 percent confidence interval:

-4.707033 3.445033

sample estimates:

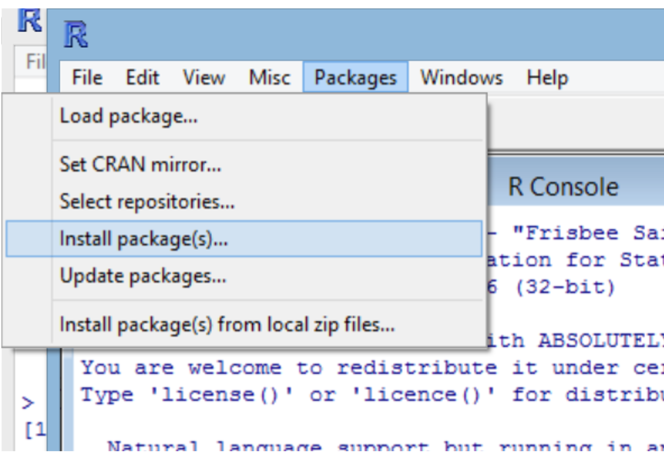
mean of x

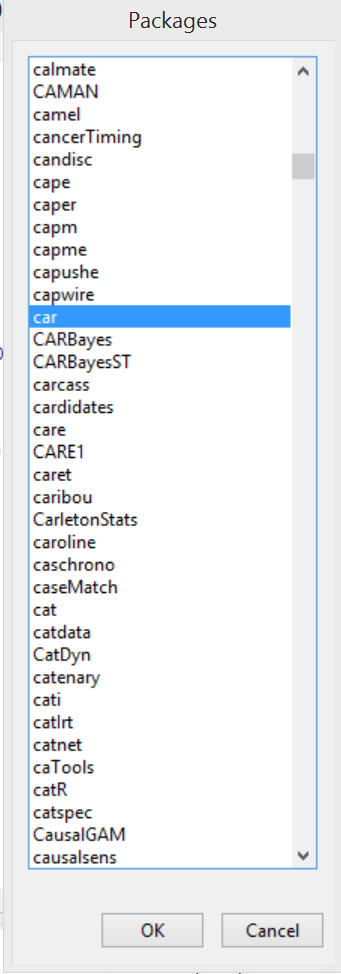
-0.631

Notice how the calculations are done automatically even though the qt() function produces a vector with two elements in it. I checked my confidence interval calculation with the results from t.test(), which automatically calculates the confidence interval and does a hypothesis test for a specified mean (mu). Please be careful when intermixing vectors and scalar values when doing calculations like this so that unintended results do not occur.

Packages

If you want to use functions that are in other packages, you may need to install and then load the package into R. For example, we will be using the car package later in the course. While in the R console, select PACKAGES > INSTALL PACKAGE(S) from the main menu.



A number of locations around the world will come up. Choose one close to you (I usually choose USA(IA), which is at Iowa State U.). Next, the list of packages will appear. Select the car package and select OK.

The package will now be installed onto your computer. This only needs to be done once per computer. To load the package into your current R session, type library(package = car) at the R Console prompt. This needs to be done only once in an R session. If you close R and reopen, you will need to use the library() function again.

Characters

Object names can include periods and underscores. For example, “mod.fit” could be a name of an object and it is often said as “mod dot fit”.

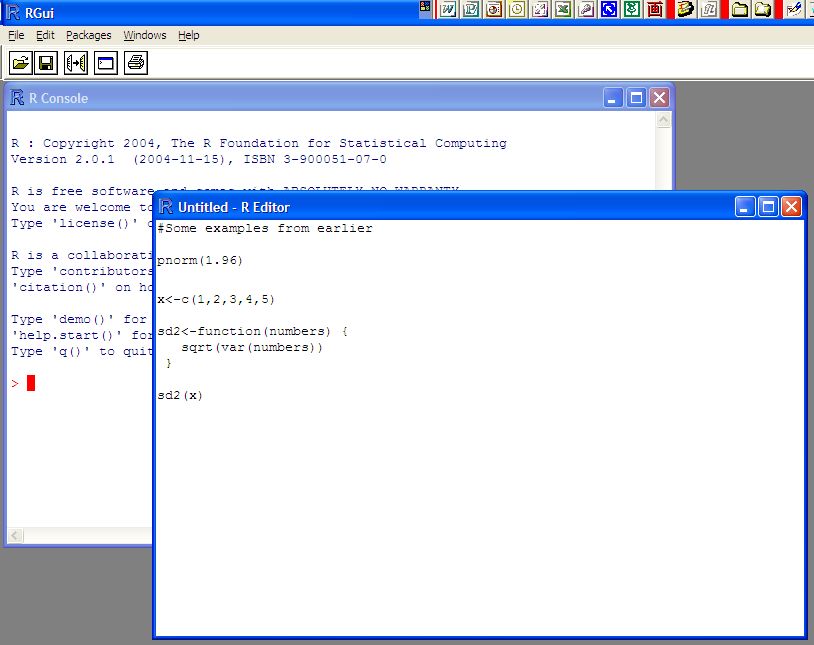
R IS CASE SENSITIVE!

**Program editors**

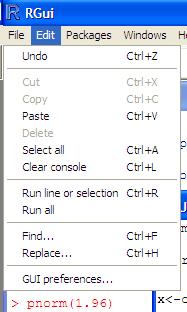
Often, you will have a long list of commands that you would like to execute all at once – i.e., a program. Instead of typing all of the code line by line at the R Console prompt, you could type it in Notepad or some other text editor and copy and paste the code into R.

R’s program editor

Starting with R 2.0, a VERY limited program editor was incorporated into R. Select FILE > NEW SCRIPT to create a new program. Below is what the editor looks like with some of the past examples.



To run the current line of code (where the cursor is positioned) or some highlighted code, select EDIT > RUN LINE OR SELECTION.



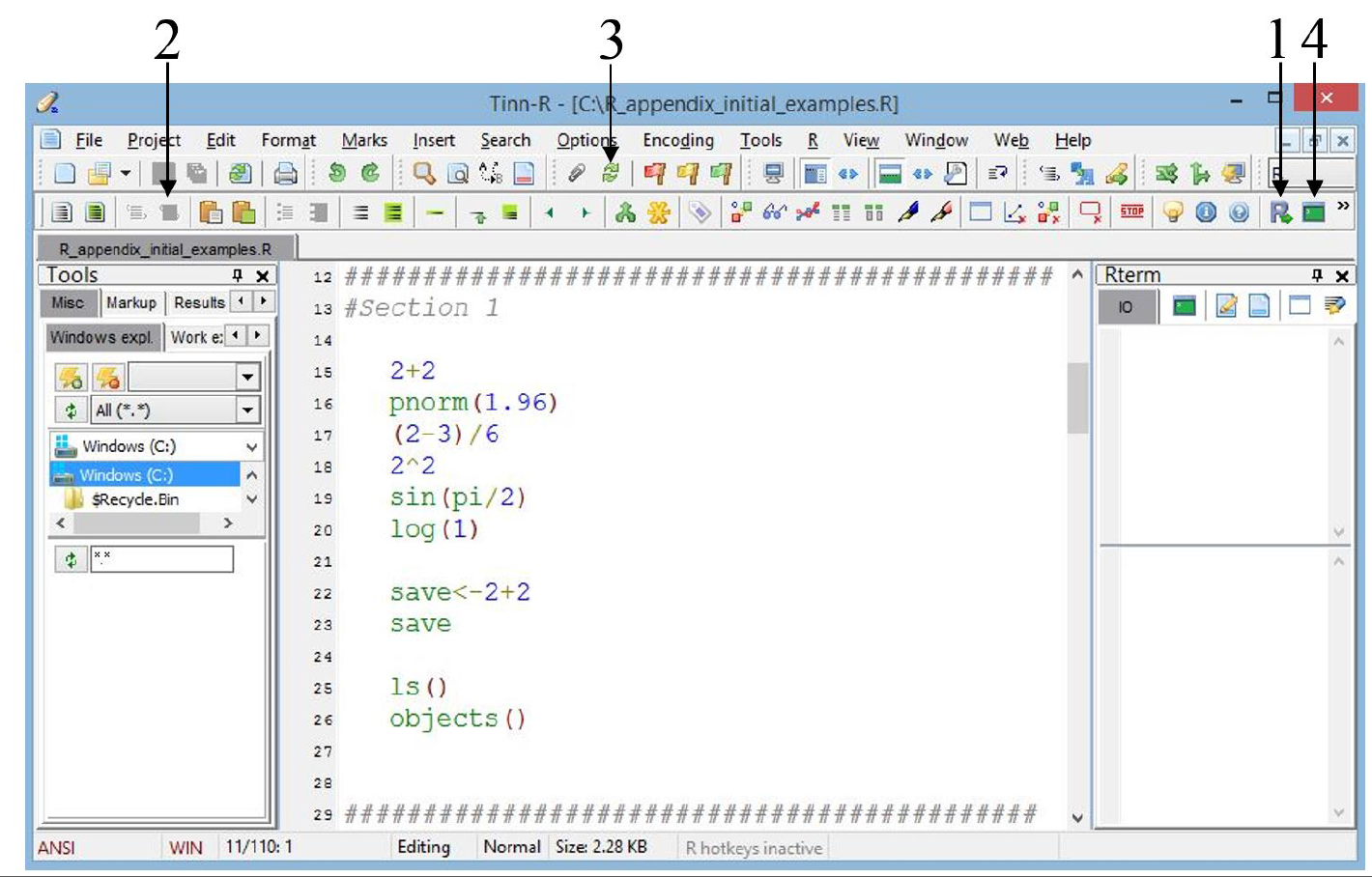
To run all of the program, select EDIT > RUN ALL. To save your code as a program outside of R, select FILE > SAVE and make sure to use a .R extension on the file name. To open a program, select FILE > OPEN SCRIPT. Note that you can have more than one program open at the same time.

There are MUCH BETTER program editors! Each of the editors described next have color coding of the program code which makes reading programs MUCH easier! **I recommend using one of these editors.**

Tinn-R

Tinn-R (<http://nbcgib.uesc.br/lec/software/editores/tinn-r/en>) is a free, Windows-based program editor that is a separate software package outside of R. This editor is much more advanced than the R editor. Note that a program needs to be saved with the .R extension for syntax highlighting to appear by default.

Below is a screen capture of what version 3.0.2.5 looks like.



In order to run code from the editor, R's GUI needs to be open. This can be opened by selecting the “R control: gui (start/close)” icon from the R toolbar (see #1).

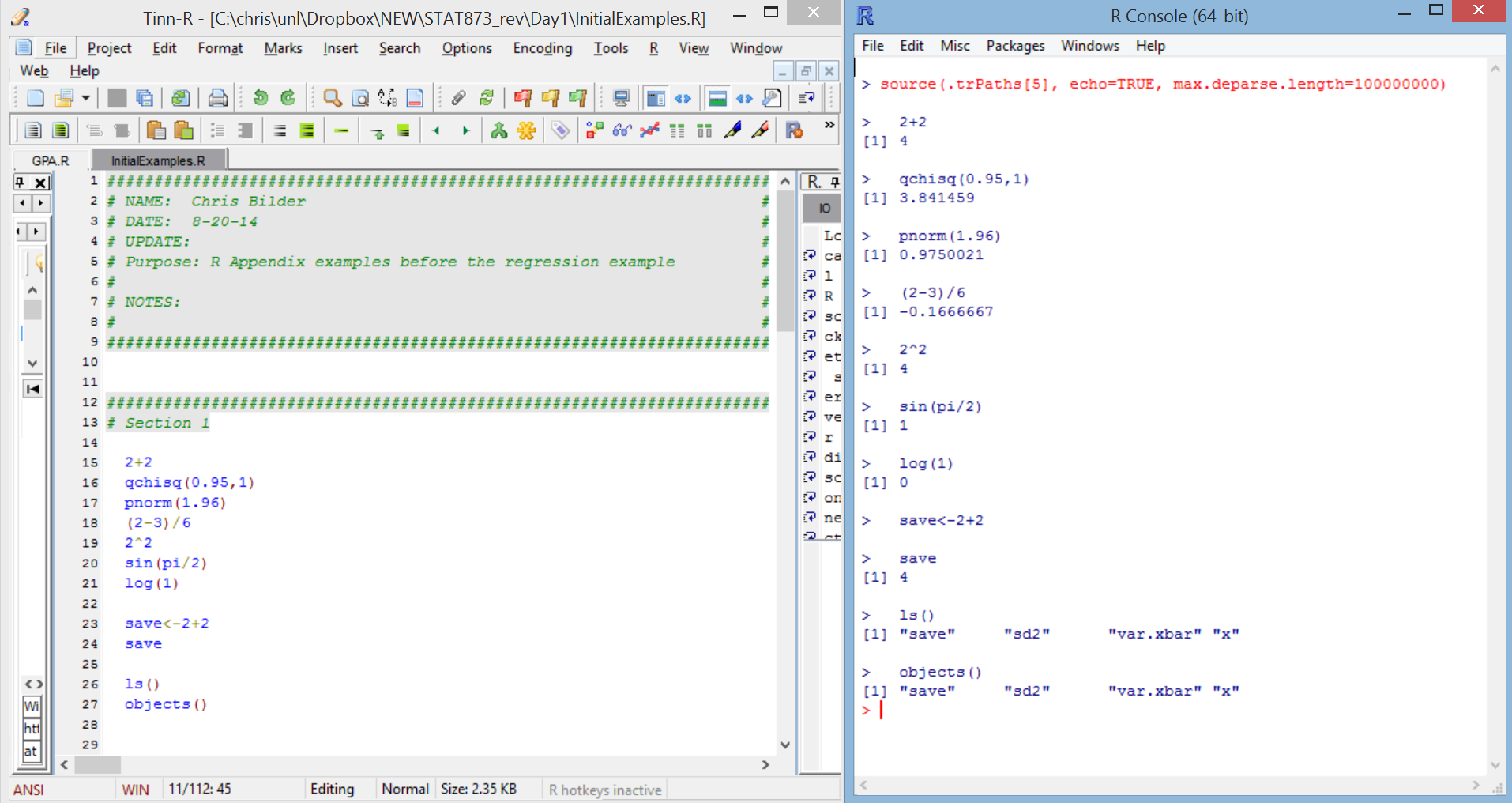
Tinn-R subsequently opens R in its SDI (single-document interface), which is a little different from R's MDI (multiple-document interface) that we have been using so far. The difference between the two interfaces is simply that the MDI uses the R GUI to contain all windows that R opens (like a graphics window – shown later in the notes) and the SDI does not.

Once R is open in its SDI, program code in Tinn-R can be transferred to R by selecting specific icons on Tinn-R's R toolbar. For example, a highlighted portion of code can be transferred to and then run in R by selecting the “R send: selection (echo = TRUE)” icon (see #2). Note that the transfer of code from Tinn-R to R does not work in the MDI.

Below are some additional important comments and tips for using Tinn-R:

* Upon Tinn-R's first use with R's SDI, the TinnRcom package is automatically installed within R to allow for the communication between the two softwares. This package is subsequently always loaded for later uses.
* When R code is sent from Tinn-R to R, the default behavior is for Tinn-R to return as the window of focus (i.e., the window location of the cursor) after R completes running the code. If Tinn-R and R are sharing the same location on a monitor, this prevents the user from immediately seeing the results in R due to it being hidden behind the Tinn-R window. In order to change this behavior, select Options > Application > R > Rgui and uncheck the Return to Tinn-R box. Alternatively, select the “Options: return focus after send/control Rgui” icon on the Misc toolbar (see #3).
* By default, the line containing the cursor is highlighted in yellow. To turn this option off, select Options > Highlighters (settings) and uncheck the Active line (choice) box.
* Long lines of code are wrapped to a new line by default. This behavior can be changed by selecting Options > Application > Editor and then selecting the No radio button for Line wrapping.
* Syntax highlighting can be maintained with code that is copied and pasted into a word processing program. After highlighting the desired code to copy, select Edit > Copy formatted (to export) > RTF. The subsequently pasted code will retain its color.
* When more than a few lines of code are transferred to R, you will notice that much of the code is not displayed in R to save space. This behavior can be changed by selecting Options > Application > R > BASIc and then changing the “option (max.deparse.length (echo=TRUE))” value to a very large number. I use a value of 100000000. Note that ALL R code and output always needs to be shown in projects turned in!
* Tinn-R can run R within its interface by using a link to a terminal version of R rather than R's GUI. To direct code to the Rterm window (located on the right side of figure), select the “R control: term (start/close)” icon on the R toolbar (see #4). One benefit from using R in this manner is that the syntax highlighting in the program editor is maintained in the R terminal window.

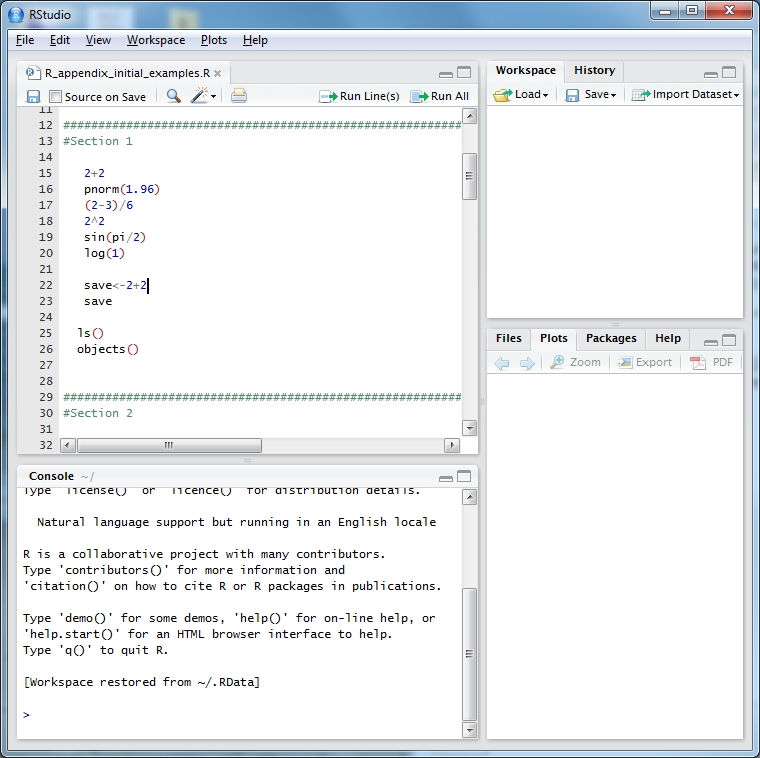
When using Tinn-R and R's GUI, it can be more efficient to use them in a multiple monitor environment. This allows for both to be viewable in different monitors at the same time. Code and output can be side-by-side in large windows without needing to switch back-and-forth between overlaying windows.



The same type of environment is achievable with a large, wide-screen monitor as well.

RStudio

While still fairly new in comparison to other editors, RStudio’s DesktopRStudio (http://www.rstudio.com; hereafter just referred to as “RStudio”) is likely the most used among all editors. This software is actually more than an editor because it integrates a program editor with the R Console window, graphics window, R-help window, and other items within one overall window environment. Thus, RStudio is an integrated development environment (IDE) for constructing and running R programs. The software is available for free, and it runs on Linux, Mac, and Windows operating systems. Below is a screen capture of it in version 0.92.23.



You can start a new program by selecting FILE > NEW > R SCRIPT or open an existing program by selecting FILE > OPEN FILE (without a program open, you will not see the program editor). To run a segment of code, you can highlight it and then select the “Run” icon in the program editor window.

Also, the editor can suggest function or package names from any loaded package if <Tab> is pressed at the end of any text string. For example, typing “pn” and pressing <Tab> yields a pop-up window suggesting functions pnbinom(), png(), and pnorm(). Pressing <Tab> where an argument could be given within a function (e.g., after a function name and its opening parenthesis or after a comma following another argument) gives a list of possible arguments for that function.

The windows available on the right side of the screen provide some additional useful information. In the upper right corner, you can view the list of objects in R’s database (similar to using ls() or objects() in the R Console). In the bottom right corner, all graphs will be sent to the PLOTS tab and help is immediately available through the HELP tab. Also, in the bottom right corner window, packages can be installed via the PACKAGES tab.

Other editors

I have often used WinEdt with the RWinEdt add-on in the past on my Windows-based computers. Also, the Emacs editor (<http://www.gnu.org/software/emacs>) with the Emacs Speaks Statistics (<http://ess.r-project.org/>) add-on are popular for Linux users.

**Regression Example**

Example: GPA data (GPA.R, gpa.txt, gpa.csv)

The independent variable is high school GPA (HS.GPA) and the dependent variable is College GPA (College.GPA). The purpose of this example is to fit a simple linear regression model and produce a scatter plot with the model plotted upon it.

Below is part of the code as it appears after being run in R. Note that I often need to fix the formatting to make it look “pretty” here. You are expected to do the same for your projects!

> #########################################################

> # NAME: Chris Bilder #

> # DATE: 8-12-14 #

> # PURPOSE: Regression model for GPA data #

> # #

> # NOTES: 1) #

> # #

> ########################################################

>

> #Read in the data

> gpa<-read.table(file = "C:\\data\\gpa.txt", header=TRUE,

sep = "")

> #Print data set

> gpa

HS.GPA College.GPA

1 3.04 3.10

2 2.35 2.30

3 2.70 3.00

4 2.55 2.45

5 2.83 2.50

6 4.32 3.70

7 3.39 3.40

8 2.32 2.60

9 2.69 2.80

10 2.83 3.60

11 2.39 2.00

12 3.65 2.90

13 2.85 3.30

14 3.83 3.20

15 2.22 2.80

16 1.98 2.40

17 2.88 2.60

18 4.00 3.80

19 2.28 2.20

20 2.88 2.60

> #Summary statistics for variables

> summary(gpa)

HS.GPA College.GPA

Min. :1.980 Min. :2.000

1st Qu.:2.380 1st Qu.:2.487

Median :2.830 Median :2.800

Mean :2.899 Mean :2.862

3rd Qu.:3.127 3rd Qu.:3.225

Max. :4.320 Max. :3.800

> #Simple plot

> plot(x = gpa$HS.GPA, y = gpa$College.GPA)



> #Better plot

> plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab = "HS

GPA", ylab = "College GPA", main = "College GPA vs. HS

GPA", xlim = c(0,4.5), ylim = c(0,4.5), col = "red",

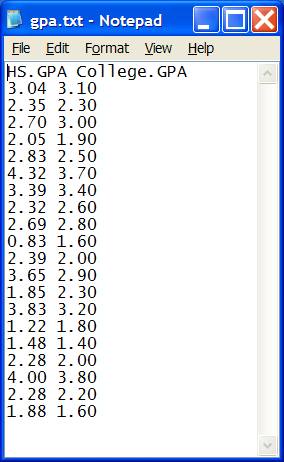
pch = 1, cex = 1.0, panel.first = grid(col = gray", lty

= "dotted"))



Notes:

* The # denotes a comment line in R. At the top of every program you should have some information about the author, date, and purpose of the program.
* The gpa.txt file is an ASCII text file that looks like:

  
  
The read.table() function reads in the data and puts it into an object called gpa here. Notice the use of the “\\” between folder names. This needs to be used instead of “\”. Also, you can use “/” too. Since the variable names are at the top of the file, the header = TRUEoption is given. The sep = "" option specifies white space (spaces, tabs, …) is used to separate variable values. One can use sep = "," for comma delimited files with read.table() or the function read.csv() without the sep or header arguments.

* There are a few different ways to read in Excel files into R. One way is to use the RODBCpackage. Below is the code that I used to read in an Excel version of gpa.txt.

library(RODBC)

z<-odbcConnectExcel("C:\\data\\gpa.xls")

gpa.excel<-sqlFetch(z, "sheet1")

close(z)

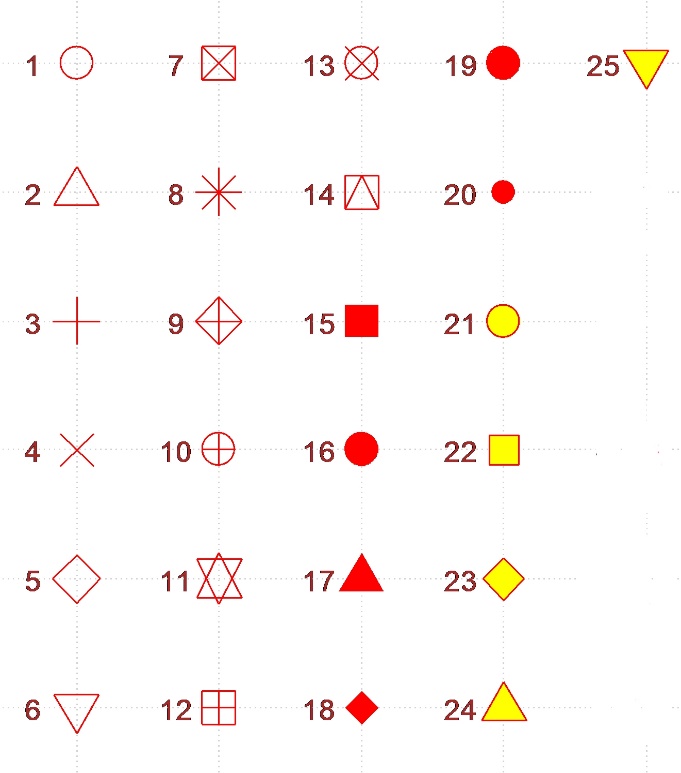
The data is stored in sheet1 of the Excel file gpa.xls. R puts the data into the object, gpa.excel, with slightly modified variable names. Due to communication issues with 32-bit and 64-bit versions of Excel and R, I recommend not using Excel files.

* You can save data to a file outside of R by using the write.table() function. Below is the code used to create a comma delimited file:

write.table(x = gpa.excel, file = "C:\\chris\\UNL\\  
 STAT875\\gpa-out.csv", quote = FALSE, row.names =

FALSE, sep =",")

* The gpa object is an object type called a *data frame*.
* The summary() function summarizes the information stored within an object. Different object types will produce different types of summaries. An example will be given soon where thesummary()function did produce a different type of summary.
* The plot()function creates a two dimensional plot of data. Here are descriptions of its options:
  + x = specifies what is plotted for the x-axis.
  + y = specifies what is plotted for the y-axis.
  + xlab = and ylab = specify the x-axis and y-axis labels, respectively.
  + main = specifies the main title of the plot.
  + xlim =andylim =specifythe x-axis and y-axis limits, respectively. Notice the use of the c() function.
  + col = specifies the color of the plotting points. Run the colors() function to see what possible colors can be used. Also, you can see [http://research.stowers-institute.org/efg/R/  
    Color/Chart/index.htm](http://research.stowers-institute.org/efg/R/Color/Chart/index.htm) for the colors from colors().
  + pch = specifies the plotting characters. Below is a list of possible characters.



* + cex = specifies the height of the plotting characters. The value 1.0 is the default.
  + panel.first = grid()specifies grid lines will be plotted. The line types can be specified as follows: 1=solid, 2=dashed, 3=dotted, 4=dotdash, 5=longdash, 6=twodash or as one of the character strings "blank", "solid", "dashed", "dotted", "dotdash", "longdash", or "twodash". These line type specifications can be used in other functions.
  + The par()function’s Help contains more information about the different plotting options!
* The plot can be brought into Word easily. In R, make sure the plot window is the current window and then select FILE > COPY TO THE CLIPBOARD > AS A METAFILE. Select the PASTE button in Word to paste it.

> #########################################################

> # Find estimated simple linear regression model

> #Fit the simple linear regression model and save the

results in mod.fit

> mod.fit<-lm(formula = College.GPA ~ HS.GPA, data = gpa)

> #A very brief look of what is inside of mod.fit - see

the summary function for a better way

> mod.fit

Call:

lm(formula = College.GPA ~ HS.GPA, data = gpa)

Coefficients:

(Intercept) HS.GPA

1.0869 0.6125

> #See the names of all of the object components

> names(mod.fit)

[1] "coefficients" "residuals" "effects" "rank"

"fitted.values"

[6] "assign" "qr" "df.residual"

"xlevels" "call"

[11] "terms" "model"

> mod.fit$coefficients

(Intercept) HS.GPA

1.0868795 0.6124941

> round(mod.fit$residuals[1:5],2)

1 2 3 4 5

0.15 -0.23 0.26 -0.20 -0.32

> #Put some of the components into a data.frame object

> save.fit<-data.frame(gpa, C.GPA.hat =

round(mod.fit$fitted.values,2), residuals =

round(mod.fit$residuals,2))

> #Print contents of save.fit

> save.fit

HS.GPA College.GPA C.GPA.hat residuals

1 3.04 3.10 2.95 0.15

2 2.35 2.30 2.53 -0.23

3 2.70 3.00 2.74 0.26

4 2.55 2.45 2.65 -0.20

5 2.83 2.50 2.82 -0.32

6 4.32 3.70 3.73 -0.03

7 3.39 3.40 3.16 0.24

8 2.32 2.60 2.51 0.09

9 2.69 2.80 2.73 0.07

10 2.83 3.60 2.82 0.78

11 2.39 2.00 2.55 -0.55

12 3.65 2.90 3.32 -0.42

13 2.85 3.30 2.83 0.47

14 3.83 3.20 3.43 -0.23

15 2.22 2.80 2.45 0.35

16 1.98 2.40 2.30 0.10

17 2.88 2.60 2.85 -0.25

18 4.00 3.80 3.54 0.26

19 2.28 2.20 2.48 -0.28

20 2.88 2.60 2.85 -0.25

> head(save.fit)

HS.GPA College.GPA C.GPA.hat residuals

1 3.04 3.10 2.95 0.15

2 2.35 2.30 2.53 -0.23

3 2.70 3.00 2.74 0.26

4 2.55 2.45 2.65 -0.20

5 2.83 2.50 2.82 -0.32

6 4.32 3.70 3.73 -0.03

> #Summarize the information stored in mod.fit

> summary(object = mod.fit)

Call:

lm(formula = College.GPA ~ HS.GPA, data = gpa)

Residuals:

Min 1Q Median 3Q Max

-0.55074 -0.25086 0.01633 0.24242 0.77976

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) 1.0869 0.3666 2.965 0.008299 \*\*

HS.GPA 0.6125 0.1237 4.953 0.000103 \*\*\*

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.3437 on 18 degrees of freedom

Multiple R-squared: 0.5768, Adjusted R-squared: 0.5533

F-statistic: 24.54 on 1 and 18 DF, p-value: 0.0001027

Notes:

* The lm() function fits the simple linear regression model. The results are stored in an object called mod.fit. Notice the use of the ~ to separate the dependent and independent variables. If there were multiple independent variables, the + sign can be used to separate them.
* By entering the mod.fit object name only, R prints a little information about what is inside of it.
* To see a list of the components inside of mod.fit, use the names()function. To access a component of an object, use the $ sign with the name of the component. Although it is not shown here, this will also work with data frames. For example, gpa$HS.GPA is the HS.GPA column of data inside of gpa!
* To create your own data frame, you can use the data.frame() function. Notice how you can specify the names of columns inside of the data.frame().
* The summary()function can be used with themod.fit object to summarize the contents of mod.fit. Notice the different results from what we had before with summary()! Without going into too much detail, each object has an “attribute” called a *class*. You can see this by specifying the attributes() or class()functions. For example,

> class(mod.fit)

[1] "lm"

says that mod.fit is of class type lm. The summary()function is what R calls a generic function. This is because it can be used with many different types of classes. When summary(mod.fit) is specified, R first checks the class type and then looks for a function called summary.lm(). Notice the “lm” part was the class type of mod.fit. There is a special function called summary.lm() that is used to summarize objects of type lm! You can see information about the function in the R help. When the summary(gpa)was used earlier, the summary function went to summary.data.frame() since the gpa object had a class that was named data.frame. If there was no corresponding function for the class of an object, summary.default() would be run.

* The estimated regression model is



> #########################################################

> #Put regression line on plot

> #Open a new graphics window (can use x11() for not

windows-based computers)

> win.graph(width = 8, height = 6, pointsize = 10)

> # 1 row and 2 columns of plots

> par(mfrow = c(1,2))

> #Same scatter plot as before

> plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab = "HS

GPA", ylab = "College GPA", main = "College GPA vs.

HS GPA", xlim = c(0,4.5), ylim = c(0,4.5), col =

"red", pch = 1, cex = 1.0, panel.first=grid(col =

"gray", lty = "dotted"))

> #Puts the line y = a + bx on the plot

> abline(a = mod.fit$coefficients[1], b =

mod.fit$coefficients[2], lty = “solid”, col =

"blue", lwd = 2)

> #Same scatter plot as before

> plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab = "HS

GPA", ylab = "College GPA", main = "College GPA vs. HS

GPA", xlim = c(0,4.5), ylim = c(0,4.0),

panel.first=grid(col = "gray", lty = "dotted"))

> #Add line

> curve(expr = mod.fit$coefficients[1] +

mod.fit$coefficients[2]\*x, xlim =

c(min(gpa$HS.GPA),max(gpa$HS.GPA)), col= "blue", add =

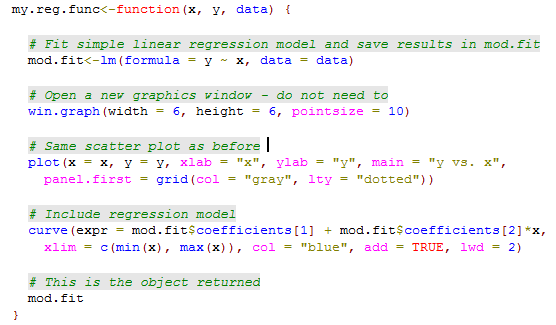
TRUE, lwd = 2)



Notes:

* The win.graph() function can be used to open a new plotting window.
* Theabline() function can be used to draw straight lines on a plot. In the format used here, the line y = a + bx was drawn where a was the  (intercept) and b was the  (slope).
* In the second plot, the curve()function was used to draw the line on the plot. This was done to have the line within the range of the high school GPA values.

Below is a function written to help automate the analysis.



> save.it<-my.reg.func(x = gpa$HS.GPA, y = gpa$College.GPA,

data = gpa)



> names(save.it)

[1] "coefficients" "residuals" "effects" "rank"

"fitted.values"

[6] "assign" "qr" "df.residual"

"xlevels" "call"

[11] "terms" "model"

> summary(save.it)

Call:

lm(formula = y ~ x, data = data)

Residuals:

Min 1Q Median 3Q Max

-0.55074 -0.25086 0.01633 0.24242 0.77976

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) 1.0869 0.3666 2.965 0.008299 \*\*

x 0.6125 0.1237 4.953 0.000103 \*\*\*

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.3437 on 18 degrees of freedom

Multiple R-squared: 0.5768, Adjusted R-squared: 0.5533

F-statistic: 24.54 on 1 and 18 DF, p-value: 0.0001027

Final notes:

* In addition to accessing a column of data in the gpa data frameby using the $ argument, you can also specify a particular column number. For example, gpa[,1] will return the first column of data. Also, gpa[1:3,] will return the first three rows.
* Typing a function name only, like lm, will show you the actual code that is used by the function to do the calculations! This can be useful when you want to know more about how a function works or if you want to create your own function by modifying the original version. Often, you will see code within a function like .C or .Fortran. These are calls outside of R to a C or Fortran program.
* Please remember to always use the Help if you do not understand a particular function!
* Mathematical expressions can be put on plots using the expression()function. For example,

> plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab = "HS

GPA", ylab = "College GPA", main = expression(hat(Y) ==

hat(beta)[0] + hat(beta)[1]\*x), xlim = c(0,4.5), ylim =

c(0,4.5), col = "red", pch = 1, cex = 1.0, panel.first

= grid(col = "gray", lty = "dotted"))

> curve(expr = mod.fit$coefficients[1] +

mod.fit$coefficients[2]\*x, xlim = c(min(gpa$HS.GPA),

max(gpa$HS.GPA)), col= "blue", add = TRUE, lwd = 2)



Run demo(plotmath) at an R Console prompt for help on the code.

* To get specific x-axis or y-axis tick marks on a plot, use the axis()function. For example,

#Note that xaxt = "n" tells R to not give any labels on the

# x-axis (yaxt = "n" works for y-axis)

plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab = "HS GPA",

ylab = "College GPA", main = "College GPA vs. HS GPA",

xaxt = "n", xlim = c(0, 4.5), ylim = c(0, 4.5), col =

"red", pch = 1)

#Major tick marks

axis(side = 1, at = seq(from = 0, to = 4.5, by = 0.5))

#Minor tick marks

axis(side = 1, at = seq(from = 0, to = 4.5, by = 0.1), tck

= 0.01, labels = FALSE)

* R is often referred to as an objected oriented language. This is because generic functions, like summary(), can lead to different results due to the class of the object used with it. As discussed earlier, R first checks an object's class type when a generic function is run. R will then look for a *method* function with the name format <generic function>.<class name>. For example, summary(mod.fit) finds the function summary.lm() that summarizes fits from regression models.   
    
  The purpose of generic functions is to use a familiar language set with any object. For example, we frequently want to summarize data or a model (summary()), plot data (plot()), and find predictions (predict()), so it is convenient to use the same language set no matter the application. The object class type determines the function action. Understanding generic functions may be one of the most difficult topics for new R users. One of the most important points students need to know now is where to find help for these functions. For example, if you want help on the results from summary(mod.fit), examine the help for summary.lm() rather than the help for summary() itself.

To see a list of all method functions associated with a class, use methods(class = <class name>), where the appropriate class name is substituted for <class name>. For our regression example, the method functions associated with the lm class are:

> methods(class = lm)

[1] add1.lm\* alias.lm\* anova.lm

case.names.lm\* confint.lm\*

cooks.distance.lm\*

<OUTPUT EDITED>

[31] rstudent.lm simulate.lm\* summary.lm   
 variable.names.lm\* vcov.lm\*

Non-visible functions are asterisked

To see a list of all method functions for a generic function, use methods(generic.function = <generic function name>) where the appropriate generic function name is substituted for <generic function name>. Below are the method functions associated with summary():

> methods(generic.function = summary)

[1] summary.aov summary.aovlist

summary.aspell\* summary.connection

summary.data.frame

<OUTPUT EDITED>

[26] summary.stepfun summary.stl\*

summary.table summary.tukeysmooth\*

Non-visible functions are asterisked

* When you become more advanced with your R coding, check out [http://google-styleguide.googlecode.com/svn  
  /trunk/google-r-style.html](http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html) for a style guide to it. By having one standard style to writing code, it becomes easier for everyone to read it.
* The open-source environment has led to the development of a large community for R users to go to when in need of help. In particular, there is an active listserv <http://www.r-project.org/mail.html>. One of the specific search engines for the listserv is available at <http://dir.gmane.org/gmane.comp.lang.r.general>. Other locations to go to for help include:
  + Stack Overflow at [http://stackoverflow.com/  
    questions/tagged/r](http://stackoverflow.com/questions/tagged/r) which provides an alternative format than a traditional listserv. This website follows a question and answer format where multiple answers can be submitted and ranked by the community in terms of the answer's usefulness. Also, Stack Overflow's web page at <http://stackoverflow.com/tags/r/info> provides a number of links to find help on R in other locations.
  + Another search engine, <http://rseek.org>, can return listings from various privately maintained R help sites and blogs.
  + There are a large number of blogs devoted to the use of R. The R-bloggers website at <http://www.r-bloggers.com> serves as an aggregator for many of these blogs.My favorite R blog is <http://blog.revolutionanalytics.com>, which is published by Revolution Analytics (a company that “sells” its own version of R).
* Are there any point-and-click ways to produce plots or output? Yes, the Rcmdr (short for “R Commander”) package can for many statistical methods. This package does not come with the initial installation of R, so you will need to install it. Once the package is installed, use library(package = Rcmdr) to start it. The first time that you run this code, R will ask if you want to install a number of other packages. Select yes, but note that this may take some time to install all of them. Once a data set is loaded into R Commander (select the box next to “Data set:”), various drop down window options can be chosen to complete the desired analysis.

**Rough timeline for R**

1970s

* 1976: A preliminary version of S was created by Bell Labs as a programming language like C, but for statistics. John Chambers is one of the primary inventors. A nice video about the early days of S is available at: <https://www.youtube.com/watch?v=jk9S3RTAl38> and <http://blog.revolutionanalytics.com/2014/01/john-chambers-recounts-the-history-of-s-and-r.html>.

1980s

* 1988: Statistical Sciences Inc. creates the S-Plus software package and sells the software to users.

1990s

* 1993: Statistical Sciences merges with Mathsoft and buys the exclusive license to market S.
* 1996: A paper by Ross Ihaka and Robert Gentleman is published in the *Journal of Computational and Graphical Statistics* on an early version of R.
* 1999: John Chambers receives the Association for Computing Machinery (ACM) Award for S.

2000s

* 2000: A first full release of R is made on February 29. It is based on the S syntax, but it is not exactly S. R is often referred to as “not unlike S”.
* 2001: I first hear about R early this year!
* 2004: The first UseR! conference is held in Austria and the R Foundation is formed.
* 2004: During a Joint Statistical Meetings (JSM) session that I attended, a SPSS executive says his company and other statistical software companies have felt R’s impact and they are changing their business model.
* 2007: The first USA-based UseR! conference is held at Iowa State University.
* 2008: The editor for the *Journal of the American Statistical Association* says during a JSM presentation that R has become the de facto statistical software package for researchers.
* 2009: A New York Times article [“Data Analysts Captivated by R’s Power”](http://www.nytimes.com/2009/01/07/technology/business-computing/07program.html?_r=1&emc=eta1) is published on 1-6-09. The article contains the now famous quote by Anne Milley of SAS:

I think it addresses a niche market for high-end data analysts that want free, readily available code. We have customers who build engines for aircraft. I am happy they are not using freeware when I get on a jet.

Milley later says that she should not have made the airplane comment. SAS starts promoting its ability to call R programs from PROC IML later this same year.

* 2009: The first issue of the R Journal (<http://journal.r-project.org>) is published in May 2009, and it replaces “R News”.

2010s

* 2010: The second USA-based UseR! Conference is held at the National Institutes of Standards.
* 2011: R 2.14.1 is released in December – The new release has some significant speed improvements (see <http://blog.revolutionanalytics.com/2011/08/with-byte-compiler-r-214-will-be-even-faster.html>).
* 2012: The third USA-based UseR! Conference is held at Vanderbilt University.
* 2013: Version 3.0.0 was released in April.
* 2014: The fourth USA-based UseR! conference is held at UCLA (<http://user2014.stat.ucla.edu>)

Chambers (2010) and Venables and Ripley (2000) provided much of the pre-2000 history.