Tutorial 1: An Introduction to R, Central Tendency, and Graphics

Goal: To provide an introduction to the R computing environment and expose you to some basic features of statistics and graphics. Work through all of the following Steps to gain experience and then solve the problem requested.

Note: Assignments will use two typefaces. All text in the Arial font is instruction or explanation. All text in `Courier font` is input or output from R.

Step-1: The R Interface

I assume that you have already installed R for the appropriate platform that you are using and have already launched the program. R works essentially as a command line program that employs a question & answer approach. At the command line (depicted by a > symbol), you ask R to do something and press Enter (↵). The program does what it is told, prints the result if relevant, and/or asks for more input. Note: if you make a mistake on input (i.e., ask it to do something it doesn't recognize, it will return an error message).

For a quick impression of what R can do, try typing the following at the command prompt:

```
> plot(rnorm(1000))
```

This command draws 1000 numbers at random from the normal distribution (rnorm = random normal) and plots them in a pop-up graphics window. This example provides a quick and simple depiction of the R interface.

Step-2: R Calculator

In its simplest form, R can be used as a calculator. You can enter an arithmetic expression and receive a result. Fore example:

```
> 2+2
[1] 4
```

R knows that 2+2=4! Not particularly impressive, but you get the general idea. The [1] in front of the result is R's way of printing numbers as part of vectors. It is not particularly relevant in our first example; but, consider the situation where you ask R to produce 15 random numbers from a normal distribution:

```
> rnorm(15)
 [1]  0.19868590 -0.14433200 -0.82711721  1.65502927 -1.72037092
 [6]  0.15928215 -2.22537532 -0.33032950 -1.01694861 -0.25599696
[11]  0.21400560  1.34505977  0.67708519 -0.08029174 -0.80276094
```

In this case, the [6] indicates that 0.15928215 is the 6th element in a vector of 15 values. Note: if you are following along in R, your numbers will be different from these because random number generation is involved.

Some other examples to try:

```
> 3^2
[1] 9
```

Note the order of operations and use of parentheses (as on a calculator):

```
> 10+2/3
[1] 10.66667

> (10+2)/3
[1] 4
```

Expressions can also be used:

```
> pi
[1] 3.141593

> exp(-2)
[1] 0.1353353
```

Step-3: Variable Names

Just as you would use a calculator, it is usually necessary to store intermediate results, so they do not have to be keyed in repeatedly. R, like all other computer languages, allows you to assign values to symbolic variables.

To assign the value 2 to the variable x:

> x<-2

The two characters <- should be read as a single symbol. This is known as the assignment operator. Note that spacing is generally disregarded by R. However, in this case, if you put a space between the < and -, R would interpret it as “less than” followed by a “minus” sign!

Now that x=2 we can employ x in any subsequent calculation. For example:

```
> x+x
[1] 4

> x/2
[1] 1
```

If at anytime you forget what you have assigned x to be, just type “x”:

```
> x
[1] 2
```

Names of variables can be chosen quite freely in R. They can be built from almost any series of letters, digits, or dot-symbology.

Caveats: (1) You can not start with a digit or a dot followed by a digit. You might wish to indicate the height of plants after one year as “height.1yr” which would be a typical use of dot symbology and commonly used in R. (2) While R is relatively insensitive to spacing is quite sensitive to case. $X \neq x$, these are two different variable assignments.

Recommendation: try to use intuitive names for variables. Using an “X” for one variable and an “x” for another variable will rapidly result in miscalculations.

Step-4: Vectorized Arithmetic

Not much stats will get done with single numbers. One of the strengths of R is that it can handle entire data vectors as single objects. An example follows:

```
> weight <- c(60,72,57,90,95,72)
> weight
[1] 60 72 57 90 95 72
```

Here we create a variable called “weight”. The <- is the assignment operator. The construct c(…) is used to define the vector. Thus, the variable is being assigned six values (which might be the weight in kg of men in the sample).

These conventions for vectorized calculations make it very easy to specify statistical calculations. Let’s suppose we wished to determine the mean of the weight variable:

We know that the formula for the mean is: $x = \frac{\sum_{i=1}^{n} x}{n}$

Therefore in R:

```
> sum(weight)
[1] 446
> sum(weight)/length(weight)
[1] 74.33333
```

Note that “length” is referring to the length of the vector (6 objects) is equivalent to n.

Suppose also that we wished to get a measure of variability around the mean, say the standard deviation. We know that the formula for the standard deviation is:

$$SD = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{(n-1)}}$$

So, in R we can proceed in the various steps to see what happens. Store the mean in a new variable called xbar and then determine the individual deviations tha make up the numerator:

```
> xbar <- sum(weight)/(length(weight))
> weight-xbar
[1] -14.333333  -2.333333 -17.333333  15.666667  20.666667  -2.333333
```

We now need to get the squared deviations:

```
> (weight - xbar)^2
[1] 205.444444   5.444444 300.444444 245.444444 427.111111   5.444444
```

To produce the numerator, these values need to be summed:

```
> sum((weight - xbar)^2)
[1] 1189.333
```

And to generate the SD, construct the denominator, take square root, & finalize calculation:

```
> sqrt(sum((weight - xbar)^2)/length(weight) -1)
[1] 14.04358
```

This exercise was a bit belabored for a pedagogic reason, and that was to show you the power of R as a calculator. These are the same sets of calculations you would perform on a calculator to determine the mean and the SD. But, R is also a statistical program and has many predefined calculations built in. The same results could have been generated in the following fashion:

```
> mean(weight)
[1] 74.33333

> sd(weight)
[1] 15.42293
```

We could also generate a whole host of additional statistics, for example:

```
> median(weight)
[1] 72
```

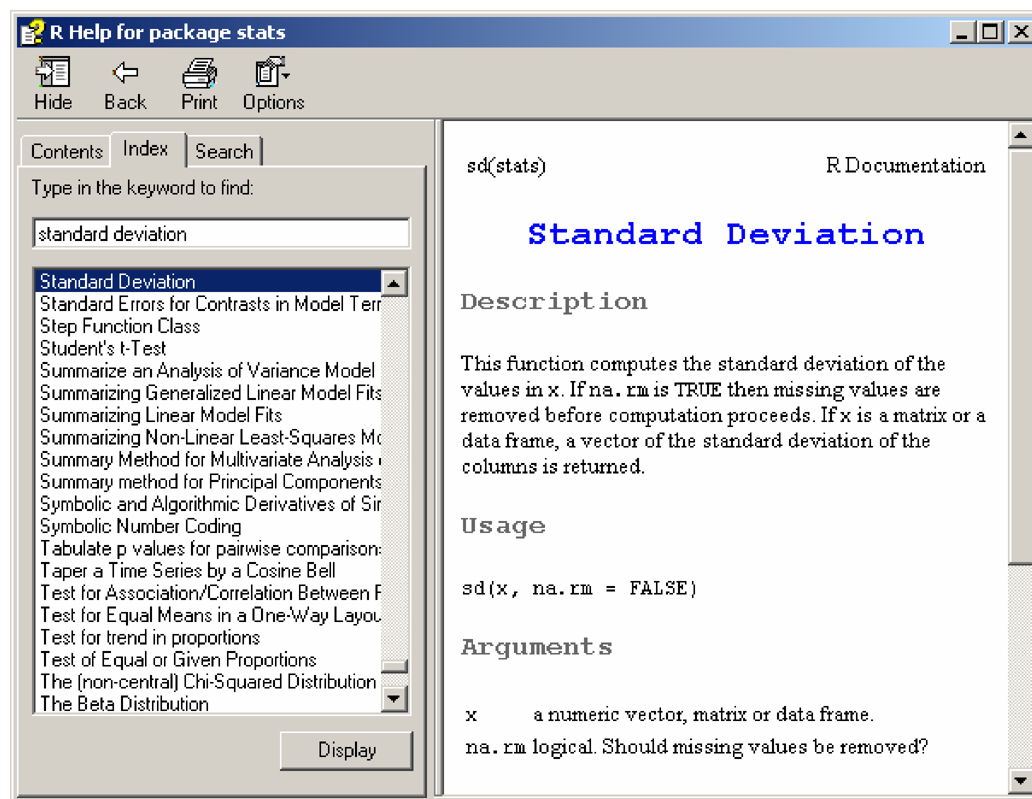
```
> var(weight)
[1] 237.8667

> range(weight)
[1] 57 95
```

Step-5: Getting Help

One of the really nice features of R is the enormous wealth of online HTML help available. Suppose in the previous example that you knew you wanted to get the mean and SD of a set of values but didn't know what the correct syntax was? Simply type "help(stats)" at he command prompt:

```
> help(stats)
```

Note that an entire help menu opens up as a separate window in the R-GUI. Click on the Index tab and type in "standard deviation" and then select standard deviation from the list and the documentation appears in the right-hand window along with the correct syntax for usage.



The use of help can be employed anywhere in R and has very strong capabilities (i.e., it really is "helpful" help).

Step-6: Basic Graphics

One of the most important features of R is a strong graphics capability. This is a critical component to statistical data analysis. In fact, many argue that the presentation and analysis of data *relies* on the generation of appropriate graphics. Much of statistics is oriented towards getting a feel for or insight into your data.

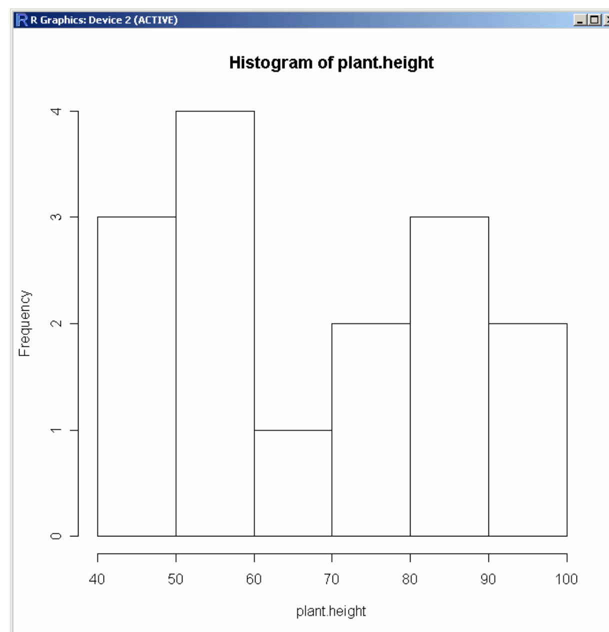
Let's create a new variable called "plant.height" (in cm) with a larger vector (more values):

```
> plant.height <- c(50,52,53,67,89,99,43,48,84,60,72,57,90,95,72)
> plant.height
 [1] 50 52 53 67 89 99 43 48 84 60 72 57 90 95 72
```

One of the first things we usually ask ourselves in parametric statistics is whether or not our data are normally distributed or not. The first way to examine this is to construct a histogram or bin plot:

```
> hist(plant.height)
```

Which results in a separate graphics panel that looks like this:



Obviously, these data do not look much like a normal or bell-shaped distribution. In fact, the plot is suggestive of a bimodal distribution.

The graphics component of R is actually an entirely separate subsystem. There are default parameters for every plot (like the histogram you just constructed), but usually you will want to change these for various reasons. The methods for doing so may at times seem tedious, but in the end it is a very flexible and powerful approach. A good way to start thinking about what your options are is to do a demonstration (many of the sub-routines of R provide this option):

```
> Demo(graphics)
```

Keep hitting the Enter key as directed and you will see a series of windows demonstrating some of the capabilities of R. Next ask for help:

```
> Help(graphics)
```

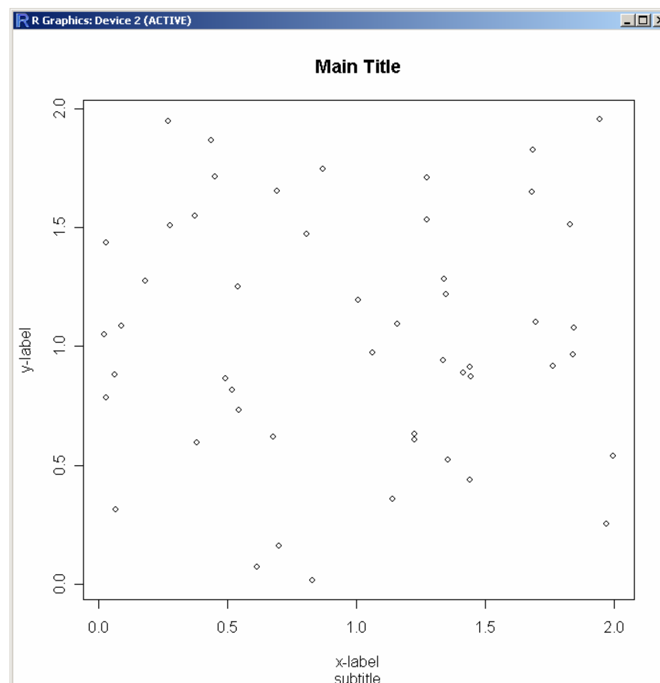
Help will provide you with a detailed set of options for any particular type of graph that you are interested in (for example the histogram that we just constructed).

For now, let's try to understand the graphics model that R employs. There is a figure region containing a centralized plotting region surrounded by margins. Coordinates inside the plotting region are specified in data units. Coordinates in the margins are specified in lines of text as you move in a direction perpendicular to a side of the plotting region, but in data units as you move along the side. This is useful because you often wish to place text in the margins of a plot.

A standard x-y plot scatter plot has an x and a y title label generated from the expressions being plotted. You may, however, override these labels and also add two further titles, a main title above the plot and a subtitle at the very bottom, in the plot call.

```
> x<-runif(50,0,2)
> y<-runif(50,0,2)
> plot (x,y, main="Main Title", sub="subtitle", xlab="x-label",
ylab="y-label")
```

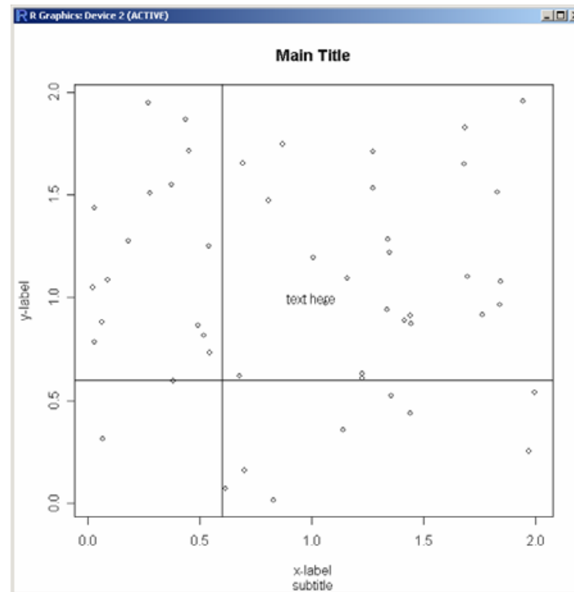
Which results in the following:



Inside the plotting region you can place points and lines that are either specified in the plot call or added later with points and lines. By way of example, you can also place text (at 1.0, 1.0) and add a vertical and horizontal line (abline function) to the plot (at 0.6, 0.6):

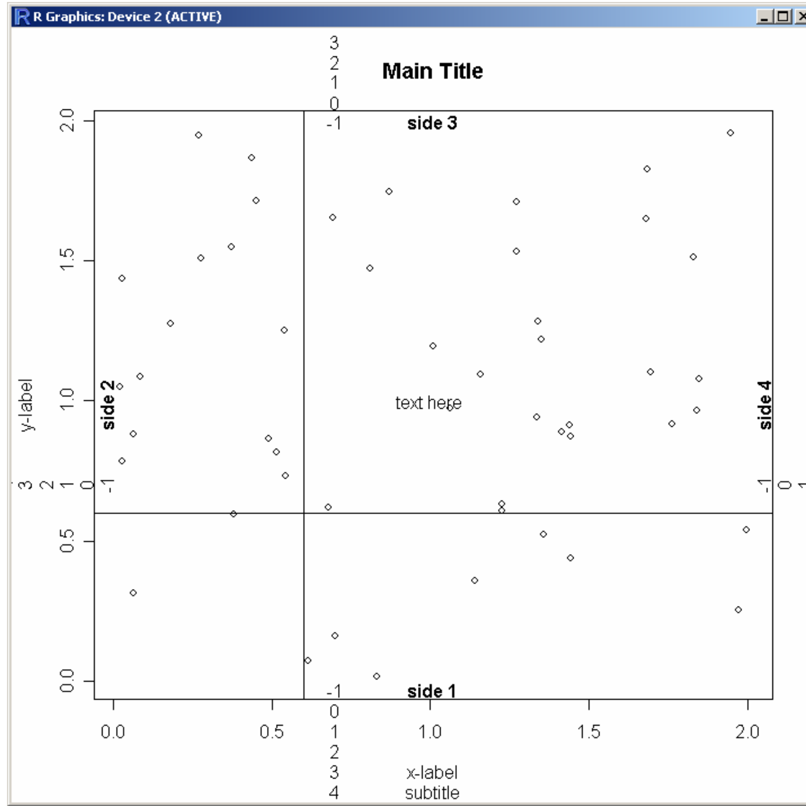
```
> text(1.0,1.0, "text here")
> abline(h=0.6,v=0.6)
```

Which results in:



The margin coordinates are used by the mtext function and can be demonstrated as follows:

```
> for(side in 1:4) mtext(-1:4,side=side,at=0.7,line=-1:4)
> mtext(paste("side",1:4), side=1:4, line=-1, font=2)
```



The for loop places the numbers -1 to 4 on corresponding lines in each of the four margins, at an off-center position of 0.7 measured in user coordinates. The subsequent call places a label on each side, giving the side number. The argument font=2 means that a boldface font is used. Notice from the figure that not all of the margins are wide enough to fit lines of text (there is some cropping that occurs).

Step 7: Building Complex Plots

High level plots, that is those that are used to communicate final results your scientific peers, are usually composed of elements, each of which can be drawn separately. The separate commands often allow finer control over the elements.

The `par` function allows incredibly fine control over the details of a plot, although it can be quite confusing when you first get started. The best way to learn it is to experiment and change parameters one at a time so you can see the response.

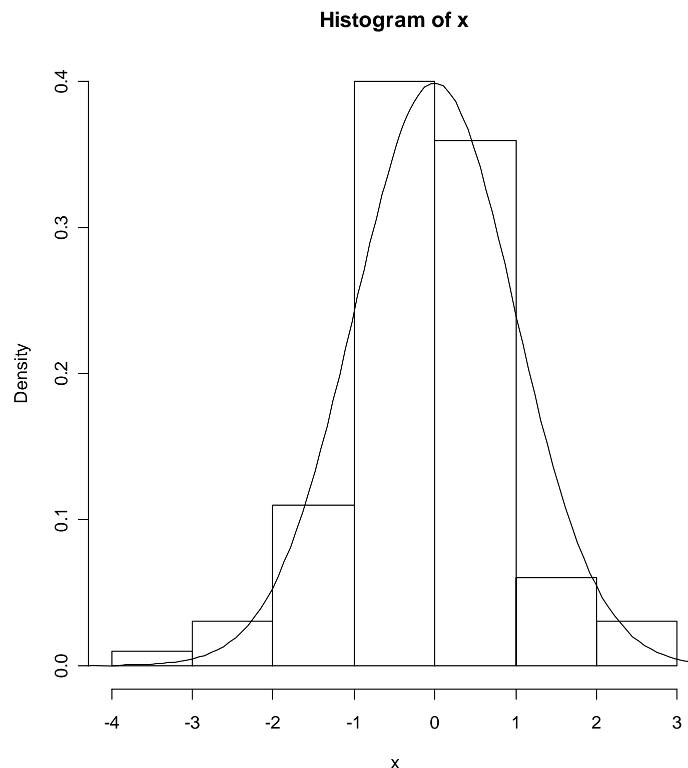
The par settings allow you to control line widths and types, character sizes and fonts, colors, style of axis, size of the plot, clipping, etc. One large figure can be simply created using `mfrow` and `mfcol` functions to create separate panels.

We will explore par as the need arises. It would be dizzying to jump into it now.

It is not uncommon that one wishes to combine several elements together into the same plot. Consider overlaying a histogram (like we created above) with a normal density line function. This would permit one to examine observed data relative to expected data for a normal distribution. Consider:

```
> x<-rnorm(100)
> hist(x,freq=F)
> curve(dnorm(x),add=T)
```

The `freq=F` argument to the `hist` ensures that the histogram is in densities (as opposed to the default which is frequencies). The `curve` function graphs an expression (in terms of x) and its `add=T` allows it to overplot an existing plot. The result is as follows:



Problem: Box Example 3.1 (Zar 1999, p.22)

Using R, enter the data as a vector. Determine N, ΣXs, mean, median, and mode. Construct a histogram. Overlay the histogram with a normal probability curve as above (hint: the example is based on random numbers and a theoretical normal distribution, you will need to figure out how to center the curve on the observed data).