A few illustrative examples in R

This "tutorial" is by no means unique! There are many, many other resources for learning R available on the web, beginning with the docs on the R web page itself: www.r-project.org.

Alternatively, just try googling "R tutorial" or something similar and see how many relevant links appear!

This web page consists mostly of excerpts from a series of R tutorials presented as companions to subject-oriented lectures during a week-long introductory course in statistics for astronomers at roughly the post-doctorate level. Several versions of the full tutorials are available at http://www.stat.psu.edu/~dhunter/astrostatistics/

Very briefly, some positive features of R are

1. It is easily extendible; there are over a thousand user-contributed packages available at www.r-project.org, and this does not include all of what is available (e.g., see http://bioconductor.org/)
2. It is written and maintained by a very strong group that includes some of the world's top computational statisticians.
3. It is an interpreted, command-line language, which makes it useful for exploratoty data analysis.
4. It is open-source and written (mostly) in C.

Several of these features are shared by other languages, such as MATLAB or Python (each of which has its own advantages), but as a tool for research-level computational statistical purposes, it is hard to beat R.

Downloading R; installing packages

Go to www.r-project.org, click on "CRAN" under "Download" in the list on the left side, choose a mirror, then look at the "Download and Install R" section.

Packages, which extend the capabilities of R, may also be downloaded from the CRAN mirrors. Alternatively, one may use the install.packages command, as in this example:

```
install.packages("mixtools")
library(mixtools)
example(npEMindrep)
```

But now we are getting ahead of ourselves!

Reading data into R

Enter R by typing "R" (LINUX) or double-clicking to execute Rgui.exe (Windows) or R.app (Mac).

When this tutorial is used online, the indented lines in typewriter font

```
# like this one
```

are meant to be copied and pasted directly into R at the command prompt.

We start by extracting some system and user information and the R.version you are using. citation tells how to cite R in publications. R is released under the GNU Public Licence, as indicated by copyright. Typing a question mark in front of a command opens the help file for that command.

```
Sys.info()
R.version
citation()
?copyright
```

The various capitalizations above are important as R is case-sensitive. When using R interactively, it is very helpful to know that the up-arrow key can retrieve previous commands, which may be edited using the left- and right-arrow keys and the delete key.

The last command above, ?copyright, is equivalent to help(copyright) or help("copyright"). However, to use this command you have to know that the function called "copyright" exists. Suppose that you knew only that there was a function in R that returned copyright information but you could not remember what it was called. In this case, the help.search function provides a handy reference tool:

```
help.search("copyright")
```

Last but certainly not least, a vast array of documentation and reference materials may be accessed via a simple command:

```
help.start()
```

The initial working directory in R is set by default or by the directory from which R is invoked (if it is invoked on the command line). It is possible to read and set this working directory using the getwd or setwd commands. A list of the files in the current working directory is given by dir, which has a variety of useful options and is only one of several utilities interfacing to the computer's files. In the setwd command, note that in Windows, path (directory) names are not case-sensitive and may contain either forward slashes or backward slashes; in the latter case, a backward slash must be written as "\\" when enclosed in quotation marks.

```
getwd  # oops!
getwd()
dir() # what's in this directory?
```

We wish to read an ASCII data file into an R object using the read.table command or one of its variants. Let's begin with a cleaned-up version of the Hipparcos dataset described above, a description of which is given at http://astrostatistics.psu.edu/datasets/HIP_star.html. There are two distinct lines below that read the dataset and create an object named hip. The first (currently commented out) may be used whenever one has access to the internet; the second assumes that the HIP_star.dat file has been saved into the current working directory.

```
#   hip  <-  read.table("http://astrostatistics.psu.edu/datasets/HIP_star.dat",
#      header=T,fill=T) # T is short for TRUE
   hip  <-  read.table("HIP_star.dat", header=T,fill=T)
```

The "<-", which is actually "less than" followed by "minus", is the R assignment operator. Admittedly, this is a bit hard to type repeatedly, so fortunately R also allows the use of a single equals sign (=) for assignment. It is *almost* always true that "=" is equivalent to "<-".

Summarizing the dataset

The following R commands list the dimensions of the dataset and print the variable names (from the single-line header). Then we list the first row, the first 20 rows for the 7th column, and the sum of the 3rd column.

```
dim(hip)
names(hip)
hip[1,]
hip[1:20,7]
sum(hip[,3])
```

Note that vectors, matrices, and arrays are indexed using the square brackets and that "1:20" is shorthand for the vector containing integers 1 through 20, inclusive. Even punctuation marks such as the colon have help entries, which may be accessed using help(":").

Next, list the maximum, minimum, median, and median absolute deviation (similar to standard deviation) of each column. First we do this using a for-loop, which is a slow process in R. Inside the loop, c is a generic R function that combines its arguments into a vector and print is a generic R command that prints the contents of an object. After the inefficient but intuitively clear approach using a for-loop, we then do the same job in a more efficient fashion using the apply command. Here the "2" refers to columns in the x array; a "1" would refer to rows.

```
for(i in 1:ncol(hip)) {
   print(c(max(hip[,i]), min(hip[,i]), median(hip[,i]), mad(hip[,i])))
}
apply(hip, 2, max)
apply(hip, 2, min)
apply(hip, 2, median)
```

```
apply(hip, 2, mad)
```

The curly braces {} in the for loop above are optional because there is only a single object (command) inside. Notice that the output gives only NA for the last column's statistics. This is because a few values in this column are missing. We can tell how many are missing and which rows they come from as follows:

```
sum(is.na(hip[,9]))
which(is.na(hip[,9]))
```

There are a couple of ways to deal with the NA problem. One is to repeat all of the above calculations on a new R object consisting of only those rows containing no NAs:

```
y  <-  na.omit(hip)
for(i in 1:ncol(y)) {
   print(c(max(y[,i]), min(y[,i]), median(y[,i]), mad(y[,i])))
}
```

Another possibility is to use the na.rm (remove NA) option of the summary functions. This solution gives slightly different answers from the the solution above; can you see why?

```
for(i in 1:ncol(hip)) {
   print(c(max(hip[,i],na.rm=T), min(hip[,i],na.rm=T), median(hip[,i],na.rm=T), mad(hip[,i],
}
```

A vector can be sorted using the Shellsort or Quicksort algorithms; rank returns the order of values in a numeric vector; and order returns a vector of indices that will sort a vector. The last of these functions, order, is often the most useful of the three, because it allows one to reorder all of the rows of a matrix according to one of the columns:

```
sort(hip[1:10,3])
hip[order(hip[1:10,3]),]
```

Each of the above lines gives the sorted values of the first ten entries of the third column, but the second line reorders *each* of the ten rows in this order. Note that neither of these commands actually alters the value of x, but we could reassign x to equal its sorted values if desired.

More R syntax

Arithmetic in R is straightforward. Some common operators are: + for addition, - for subtraction, * for multiplication, / for division, %/% for integer division, %% for modular arithmetic, ^ for exponentiation. The help page for these operators may accessed by typing, say,

```
?'+'
```

Some common built-in functions are exp for the exponential function, sqrt for square root, log10 for base-10 logarithms, and cos for cosine. The syntax resembles "sqrt(z)". Comparisons are made using < (less than), <= (less than or equal), == (equal to) with the syntax "a >= b". To test whether a and b are exactly equal and return a TRUE/FALSE value (for instance, in an "if" statement), use the command identical(a,b) rather a==b. Compare the following two ways of comparing the vectors a and b:

```
a <- c(1,2);b <- c(1,3)
a==b
identical(a,b)
```

Also note that in the above example, 'all(a==b)' is equivalent to 'identical(a,b)'.

R also has other logical operators such as & (AND), | (OR), ! (NOT). There is also an xor (exclusive or) function. Each of these four functions performs elementwise comparisons in much the same way as arithmetic operators:

```
a <- c(TRUE,TRUE,FALSE,FALSE);b <- c(TRUE,FALSE,TRUE,FALSE)
!a
a & b
a | b
xor(a,b)
```

However, when 'and' and 'or' are used in programming, say in 'if' statements, generally the '&&' and '||' forms are preferable. These longer forms of 'and' and 'or' evaluate left to right, examining only the first element of each vector, and evaluation terminates when a result is determined. Some other operators are listed here.

The expression "y == x^2" evaluates as TRUE or FALSE, depending upon whether y equals x squared, and performs no assignment (if either y or x does not currently exist as an R object, an error results).

Let's continue with simple characterization of the dataset: find the row number of the object with the smallest value of the 4th column using which.min. A longer, but instructive, way to accomplish this task creates a long vector of logical constants (tmp), mostly FALSE with one TRUE, then pick out the row with "TRUE".

```
which.min(hip[,4])
tmp <- (hip[,4]==min(hip[,4]))
(1:nrow(hip))[tmp] # or equivalently,
which(tmp)
```

Simple bivariate analysis: Boxplots

Earlier, we considered boxplots, a one-dimensional plotting technique. We may perform a slightly more sophisticated analysis using boxplots to get a glimpse at some bivariate structure. Let us examine the values of Vmag, with objects broken into categories according to the B minus V variable:

```
attach(hip)
boxplot(Vmag~cut(B.V,breaks=(-1:6)/2),
   notch=T, varwidth=T, las=1, tcl=.5,
   xlab=expression(B - V),
   ylab=expression(V[mag]),
   main="Can you find the red giants?",
   cex=1, cex.lab=1.4, cex.axis=.8, cex.main=1)
axis(2, labels=F, at=0:12, tcl=-.25)
axis(4, at=3*(0:4))
```

The notches in the boxes, produced using "notch=T", can be used to test for differences in the medians (see boxplot.stats for details). With "varwidth=T", the box widths are proportional to the square roots of the sample sizes. The "cex" options all give scaling factors, relative to default: "cex" is for plotting text and symbols, "cex.axis" is for axis annotation, "cex.lab" is for the x and y labels, and "cex.main" is for main titles. The two axis commands are used to add an axis to the current plot. The first such command above adds smaller tick marks at all integers, whereas the second one adds the axis on the right.

Nearly all aspects of the plot may be controlled, and thus publication-quality plots may be produced in R. Various mathematical symbols may also be added to plots; see the help file for plotmath for details. Here is a quick demo of some of the capabilities:

```
example(plotmath)
```

Scatterplots

The boxplots in the plot above are telling us something about the bivariate relationship between the two variables. Yet it is probably easier to grasp this relationship by producing a scatter plot.

```
plot(Vmag,B.V)
```

The above plot looks too busy because of the default plotting character, set let's use a different one:

```
plot(Vmag,B.V,pch=".")
```

Let's now use exploratory scatterplots to locate the Hyades stars. This open cluster should be concentrated both in the sky coordinates RA and DE, and also in the proper motion variables pm_RA and pm_DE. We start by noticing a concentration of stars in the RA distribution:

```
plot(RA,DE,pch=".")
```

See the cluster of stars with RA between 50 and 100 and with DE between 0 and 25?

```
rect(50,0,100,25,border=2)
```

Let's construct a logical (TRUE/FALSE) variable that will select only those stars in the appropriate rectangle:

```
filter1 <-  (RA>50 & RA<100 & DE>0 & DE<25)
```

Next, we select in the proper motions. (As our cuts through the data are parallel to the axes, this variable-by-variable classification approach is sometimes called Classification and Regression Trees or CART, a very common multivariate classification procedure.)

```
plot(pmRA[filter1],pmDE[filter1],pch=20)
rect(0,-150,200,50,border=2)
```

Let's replot after zooming in on the rectangle shown in red.

```
plot(pmRA[filter1],pmDE[filter1],pch=20, xlim=c(0,200),ylim=c(-150,50))
rect(90,-60,130,-10,border=2)
filter2 <- (pmRA>90 & pmRA<130 & pmDE>-60 & pmDE< -10) # Space in 'pmDE< -10' is necessary!
filter  <-  filter1 & filter2
```

Let's have a final look at the stars we have identified using the pairs command to produce all bivariate plots for pairs of variables. We'll exclude the first and fifth columns (the HIP identifying number and the parallax, which is known to lie in a narrow band by construction).

```
pairs(hip[filter,-c(1,5)],pch=20)
```

Notice that indexing a matrix or vector using negative integers has the effect of *excluding* the corresponding entries.

We see that there is one outlying star in the e_Plx variable, indicating that its measurements are not reliable. We exclude this point:

```
filter <- filter & (e_Plx<5)
pairs(hip[filter,-c(1,5)],pch=20)
```

How many stars have we identified? The filter variable, a vector of TRUE and FALSE, may be summed to reveal the number of TRUE's (summation causes R to coerce the logical values to 0's and 1's).

```
sum(filter)
```

As a final look at these data, let's consider the HR plot of Vmag versus B.V but make the 92 Hyades stars we just identified look bigger (pch=20 instead of 46) and color them red (col=2 instead of 1). This shows the Zero Age Main Sequence, plus four red giants, with great precision.

```
plot(Vmag,B.V,pch=c(46,20)[1+filter], col=1+filter,
   xlim=range(Vmag[filter]), ylim=range(B.V[filter]))
```

Linear and polynomial regression

Here is a quick example of linear regression relating BminusV to logL, where logL is the luminosity, defined to be (15 - Vmag - 5 log(Plx)) / 2.5. However, we'll use only the main-sequence Hyades to fit this model:

```
mainseqhyades <- filter & (Vmag>4 | B.V<0.2)
logL <- (15 - Vmag - 5 * log10(Plx)) / 2.5
x <- logL[mainseqhyades]
y <- B.V[mainseqhyades]
plot(x, y)
regline <- lm(y~x)
abline(regline, lwd=2, col=2)
summary(regline)
```

Note that the regression line passes exactly through the point (xbar, ybar):

```
points(mean(x), mean(y), col=3, pch=20, cex=3)
```

Here is a regression of y on exp(-x/4):

```
newx  <-  exp(-x/4)
regline2 <- lm(y~newx)
xseq  <-  seq(min(x), max(x), len=250)
lines(xseq, regline2$coef %*% rbind(1, exp(-xseq/4)), lwd=2, col=3)
```

Let's now switch to a new dataset, one that comes from NASA's Swift satellite. The statistical problem at hand is modeling the X-ray afterglow of gamma-ray bursts. First, read in the dataset:

```
#   grb <- read.table ("http://astrostatistics.psu.edu/datasets/GRB_afterglow.dat",
#      header=T, skip=1)
   grb <- read.table ("GRB_afterglow.dat", header=T, skip=1)
```

The skip=1 option in the previous statement tells R to ignore the first row in the data file. You will see why this is necessary if you look at the file. Let's focus on the first two columns, which are times and X-ray fluxes:

```
plot(grb[,1:2],xlab="time",ylab="flux")
```

This plot is very hard to interpret because of the scales, so let's take the log of each variable:

```
x <- log(grb[,1])
y <- log(grb[,2])
plot(x,y,xlab="log time",ylab="log flux")
```

The relationship looks roughly linear, so let's try a linear model (lm in R):

```
model1 <- lm(y~x)
abline(model1, col=2, lwd=2)
```

The "response ~ predictor(s)" format seen above is used for model formulas in functions like lm .

The model1 object just created is an object of class "lm". The class of an object in R can help to determine how it is treated by functions such as print and summary.

```
model1 # same as print(model1)
summary(model1)
```

Notice the sigma-hat, the R-squared and adjusted R-squared, and the standard errors of the beta-hats in the output from the summary function.

There is a lot of information contained in model1 that is not displayed by print or summary:

```
names(model1)
```

For instance, we will use the model1$fitted.values and model1$residuals information later when we look at some residuals plots.

Notice that the coefficient estimates are listed in a regression table, which is standard regression output for any software package. This table gives not only the estimates but their standard errors as well, which enables us to determine whether the estimates are very different from zero. It is possible to give individual confidence intervals for both the intercept parameter and the slope parameter based on this information, but remember that a line really requires both a slope **and** an intercept. Since our goal is really to estimate a line here, maybe it would be better if we could somehow obtain a confidence "interval" for the lines themselves.

This may in fact be accomplished. By viewing a line as a single two-dimensional point in (intercept, slope) space, we set up a one-to-one correspondence between all (nonvertical) lines and all points in two-dimensional space. It is possible to obtain a two-dimensional confidence *ellipse* for the (intercept,slope) points, which may then be mapped back into the set of lines to see what it looks like.

Performing all the calculations necessary to do this is somewhat tedious, but fortunately, someone else has already done it and made it available to all R users through CRAN, the Comprehensive R Archive Network. The necessary functions are part of the "car" (companion to applied regression) package. There are several ways to install the car package, but perhaps the most straightforward is by using the install.packages function. Once the car package is installed, its contents can be loaded into the current R session using the library function:

```
library(car)
```

If all has gone well, there is now a new set of functions, along with relevant documentation. Here is a 95% confidence ellipse for the (intercept,slope) pairs:

```
confidence.ellipse(model1)
```

Remember that each point on the boundary or in the interior of this ellipse represents a line. If we were to plot all of these lines on the original scatterplot, the region they described would be a 95% confidence band for the true

regression line. A graduate student, Derek Young, and I wrote a simple function to draw the borders of this band on a scatterplot. You can see this function at www.stat.psu.edu/~dhunter/R/confidence.band.r; to read it into R, use the source function:

```
source("confidence.band.r")
confidence.band(model1)
```

In this dataset, the confidence band is so narrow that it's hard to see. However, the borders of the band are not straight. You can see the curvature much better when there are fewer points or more variation, as in:

```
tmpx <- 1:10
tmpy <- 1:10+rnorm(10) # Add random Gaussian noise
confidence.band(lm(tmpy~tmpx))
```

Also note that increasing the sample size increases the precision of the estimated line, thus narrowing the confidence band. Compare the previous plot with the one obtained by replicating tmpx and tmpy 25 times each:

```
tmpx25 <- rep(tmpx,25)
tmpy25 <- rep(tmpy,25)
confidence.band(lm(tmpy25~tmpx25))
```

A related phenomenon is illustrated if we are given a value of the predictor and asked to predict the response. Two types of intervals are commonly reported in this case: A *prediction* interval for an individual observation with that predictor value, and a *confidence* interval for the mean of all individuals with that predictor value. The former is always wider than the latter because it accounts for not only the uncertainty in estimating the true line but also the individual variation around the true line. This phenomenon may be illustrated as follows. Again, we use a toy data set here because the effect is harder to observe on our astronomical dataset. As usual, 95% is the default confidence level.

```
confidence.band(lm(tmpy~tmpx))
predict(lm(tmpy~tmpx), data.frame(tmpx=7), interval="prediction")
text(c(7,7,7), .Last.value, "P",col=4)
predict(lm(tmpy~tmpx), data.frame(tmpx=7), interval="conf")
text(c(7,7,7), .Last.value, "C",col=5)
```

Polynomial curve-fitting: Still linear regression!

Because there appears to be a bit of a bend in the scatterplot, let's try fitting a quadratic curve instead of a linear curve. **Note: Fitting a quadratic curve is still considered linear regression.** This may seem strange, but the reason is that the quadratic regression model assumes that the response y is a *linear* combination of 1, x, and x^2. Notice the special form of the lm command when we implement quadratic regression. The I function means "as is" and it resolves any ambiguity in the model formula:

```
model2 <- lm(y~x+I(x^2))
summary(model2)
```

Here is how to find the estimates of beta using the closed-form solution:

```
X <- cbind(1, x, x^2) # Create nx3 X matrix
solve(t(X) %*% X) %*% t(X) %*% y # Compare to the coefficients above
```

Plotting the quadratic curve is not a simple matter of using the abline function. To obtain the plot, we'll first create a sequence of x values, then apply the linear combination implied by the regression model using matrix multiplication:

```
xx <- seq(min(x),max(x),len=200)
yy <- model2$coef %*% rbind(1,xx,xx^2)
lines(xx,yy,lwd=2,col=3)
```

Other methods of curve-fitting

Let's try a nonparametric fit, given by loess or lowess. First we plot the linear (red) and quadratic (green) fits, then we overlay the lowess fit in blue:

```
plot(x,y,xlab="log time",ylab="log flux")
```

```
abline(model1, lwd=2, col=2)
lines(xx, yy, lwd=3, col=3)
npmodel1 <- lowess(y~x)
lines(npmodel1, col=4, lwd=2)
```

It is hard to see the pattern of the lowess curve in the plot. Let's replot it with no other distractions. Notice that the "type=n" option to plot function causes the axes to be plotted but not the points.

```
plot(x,y,xlab="log time",ylab="log flux", type="n")
lines(npmodel1, col=4, lwd=2)
```

This appears to be a piecewise linear curve. An analysis that assumes a piecewise linear curve will be carried out on these data later in the week.

In the case of non-polynomial (but still parametric) curve-fitting, we can use nls. If we replace the response y by the original (nonlogged) flux values, we might posit a parametric model of the form flux = exp(a+b*x), where x=log(time) as before. Compare a nonlinear approach (in red) with a nonparametric approach (in green) for this case:

```
flux <- grb[,2]
nlsmodel1 <- nls(flux ~ exp(a+b*x), start=list(a=0,b=0))
npmodel2 <- lowess(flux~x)
plot(x, flux, xlab="log time", ylab="flux")
lines(xx, exp(9.4602-.9674*xx), col=2, lwd=2)
lines(npmodel2, col=3, lwd=2)
```

Interestingly, the coefficients of the nonlinear least squares fit are different than the coefficients of the original linear model fit on the logged data, even though these coefficients have exactly the same interpretation: If flux = exp(a + b*x), then shouldn't log(flux) = a + b*x? The difference arises because these two fitting methods calculate (and subsequently minimize) the residuals on different scales. Try plotting exp(a + b*xx) on the scatterplot of x vs. flux for both (a,b) solutions to see what happens. Next, try plotting a + b*xx on the scatterplot of x vs. log(flux) to see what happens.

If outliers appear to have too large an influence over the least-squares solution, we can also try resistant regression, using the lqs function in the MASS package. The basic idea behind lqs is that the largest residuals (presumably corresponding to "bad" outliers) are ignored. The results for our log(flux) vs. log(time) example look terrible but are very revealing. Can you understand why the output from lqs looks so very different from the least-squares output?

```
library(MASS)
lqsmodel1 <- lqs(y~x, method="lts")
plot(x,y,xlab="log time",ylab="log flux")
abline(model1,col=2)
abline(lqsmodel1,col=3)
```

Let us now consider least absolute deviation regression, which may be considered a milder form of resistant regression than lqs. In least absolute deviation regression, even large residuals have an influence on the regression line (unlike in lqs), but this influence is less than in least squares regression. To implement it, we'll use a function called rq (regression quantiles) in the "quantreg" package. Like the "car" package, this package is not part of the standard distribution of R, so we'll need to download it. In order to do this, we must tell R where to store the installed library using the install.packages function.

```
install.packages("quantreg",lib="V:/") # lib=... is not always necessary!
library(quantreg, lib.loc="V:/")
```

Assuming the quantreg package is loaded, we may now compare the least-squares fit (red) with the least absolute deviations fit (green). In this example, the two fits are nearly identical:

```
rqmodel1 <- rq(y~x)
plot(x,y,xlab="log time",ylab="log flux")
abline(model1,col=2)
abline(rqmodel1,col=3)
```

We conclude by mentioning some well-studied and increasingly popular methods of penalized least squares. These include ridge regression (in which the penalty is proportional to the L_2 norm of the parameter vector) and LASSO (which uses an L_1 norm instead of an L_2 norm). There are also other penalized least sqaures methods, perhaps most notably the SCAD (smoothly clipped absolute deviation) penalty. For an implementation of ridge regression, see the lm.ridge function in the MASS package. To use this function, you must first type library(MASS). For a package that implements LASSO, check out, e.g., the lasso2 package on CRAN.