Lab 1 **Statistics for Laboratory Scientists**
Due: 8 Feb 2006

Introduction

This is the first of three computer labs for the course *Statistics for Laboratory Scientists*. The aim of these labs is to assist the student in learning to use the R software and to carry out basic statistical analyses.

The final page of each lab will contain several questions. Students are to carry out the analyses in the lab and submit answers to these questions. We encourage students to work together, but expect each student to write down his/her answers independently.

Note that the majority of the R code for this lab may be obtained as a file (`lab1.R`) at the following. Save this file to your computer and then open it in R using, from the menu bar, File → Display file. You can then copy and paste from the file into R, to save a bit of typing.

```
http://www.biostat.jhsph.edu/~kbroman/teaching/labstat/third/labs.html
```

Alternatively, you can type

```
url.show("http://www.biostat.jhsph.edu/~kbroman/teaching/labstat/third/lab1.R")
```

That URL is a bit long; you could find the file on the web, and the copy-and-paste the URL.

We cannot deny that R can be difficult to learn, but we feel strongly that it is a worthwhile venture. We recommend the following five rules for learning a computer language:

1. Experiment.
2. Think about what you're doing.
3. Read the manuals
4. Ask questions.
5. Use it routinely.

The aim of this first lab is to get a basic understanding of the syntax of R, and to learn some of the basic graphical and statistical features of R. It is sure to be rather boring, and even frustrating. It can be hard to learn new computer languages—there are so many details to remember.

Learning R

See the "Notes on R for Windows" webpage for information about installing R in Windows and getting starting with the language. The page includes a list of resources on learning and using R.

```
http://www.biostat.jhsph.edu/~kbroman/Rintro/Rwin.html
```

If you become committed to using R regularly, we recommend two books: *Introductory statistics with R* by Dalgaard and *Modern applied statistics with S* by Venables and Ripley.

Reading data into R

1. Download the following Excel file:
   `http://www.biostat.jhsph.edu/~kbroman/teaching/data/mathura.xls`
2. Open the file in Excel and save it as a comma-delimited (CSV) file.
3. Start R (e.g., double-click on the R icon on the desktop).
4. Type the following at the R prompt, replacing `"c:/mathura.csv"` with the appropriate location/name of the file. Note that in R you must use forward-slashes in place of the backslashes that are usually used in Windows.
   ```
   mat <- read.csv("c:/mathura.csv")
   ```

`read.csv` is a *function* that reads in data from comma-delimited files. The parentheses indicate that it is a function and contain its *arguments* (here just one argument—the file to be read).

The symbol `<-` is the "assignment" operator. (One can also use the symbol `=`, but I prefer `<-`.) The output of the function `read.csv` (i.e., the contents of the file) are assigned to a data object called `mat`.

One could instead use the more generic function `read.table` to read in these data, as follows.

```
mat <- read.table("c:/mathura.csv", sep=",", header=TRUE)
```

Here we specified two additional arguments. `sep=","` indicates that the file is comma-delimited, while `header=TRUE` indicates that the first line of file is a header row (containing the field names). The `=` sign here is different from the `<-` sign; it is for assigning values to optional arguments in a function.

If you type, at the R prompt, the name of a function without the parentheses, the code defining the function will be printed. Type `read.csv` to see its definition. You'll see arguments that may be specified and their default values; you'll further see that this function simply calls the more generic function `read.table`.

Type `?read.csv` or `help(read.csv)` to view the help file for this function (describing and giving examples of its use). What comes up is the help file for `read.table`, which also describes `read.csv` and several similar functions.

Note: You can use `read.csv` to download and load data from a file directly from the web. And so, as an alternative to what we did above, you could type the following into R:

```
mat <- read.csv("http://www.biostat.jhsph.edu/~kbroman/teaching/data/mathura.csv")
```

A few quick things

Before we delve into tedious details, let's look at a few commands to explore the above data.

After reading the data from `mathura.csv`, the object `mat` should now be in your workspace. Type `ls()` or `objects()` to list the objects in your workspace. (Note that these are both functions; another function is `q()`, which is used to exit R.)

If you type `mat` at the R prompt, it will print the contents of the object. These data are from Mathura et al., J Appl Physiol 91:74–78, 2001, and consist of red blood cell (RBC) velocity (in mm/s) and capillary diameter (in μm), at rest and during venous occlusion, as measured by capillaroscopy and OPS imaging.

First let's get a quick summary and plot of the data.

```
summary(mat)
plot(mat)
```

The data has five variables: the measurement method, the RBC velocity at rest and during venous occlusion, and the capillary diameter at rest and during venous occlusion. The function `summary` gives the mean and the quartiles of each numeric variable and the frequency distribution for the categorical variable. The function `plot` gives a plot of each variable against each other variable. This may also be obtained with the function `pairs`. The function `abline` is used below to plot a line with intercept 0 and slope 1 (i.e., the line "y = x").

```
plot(velo.vo ~ velo.rest, data=mat)
points(velo.vo ~ velo.rest, data=mat,
       subset=(method=="capillaroscopy"), col="red")
abline(0,1)
```

Use of the function `plot` results in a scatter plot of the measurements of the RBC velocity during venous occlusion against that at rest, for all measurements. Let's worry about the syntax of the command later. The second command highlights in red the points corresponding to measurements by capillaroscopy. (Points corresponding to measurements by OPS imaging remain in black.) We can do the same for the capillary diameter measurements, as follows.

```
plot(diam.vo ~ diam.rest, data=mat)
points(diam.vo ~ diam.rest, data=mat,
       subset=(method=="capillaroscopy"), col="red")
abline(0,1)
```

Data objects

The following is likely to be rather boring, so beware.

In R, data can have several possible *modes*, including numeric (numbers), character (text), logical (TRUE or FALSE), and factor (categorical). There are several different types of data objects in R, including vectors, matrices, lists, and "data frames." A vector is an ordered set whose elements all have the same mode. A matrix is a rectangular set whose elements are all of the same mode. A list is an ordered set of other data objects (the components of which may be themselves vectors, matrices, lists, or whatever). A data frame is probably the most important data type; it can be viewed as either a matrix whose columns are allowed to be of different modes, or as a list of vectors, each of the same length.

The names of objects are case-sensitive (e.g., `mat` is different than `Mat`). It's best to avoid using names that have already been taken by standard R functions (e.g., `c` or `data`).

The object `mat` is an example of a data frame. The first column is a factor (the measuring method used), while the other columns are numeric (RBC velocity or capillary diameter, at rest or under venuous occlusion).

R is distributed with a good amount of example data. You can obtain a list of these data sets by typing `data()`. To get access to the dataset `PlantGrowth`, you must first type `library(datasets)` and then `data(PlantGrowth)`. Then type `ls()` and you'll see that it is in your workspace. Type `?PlantGrowth` or `help(PlantGrowth)` to view a description of these data.

Creating simple data objects

Here we describe four extremely important functions for creating simple vectors. The most important is the function `c`, which combines a set of numbers or character strings into a vector. Type the following.

```
x <- c(1, 3.5, -28.4, 10)
x
animals <- c("cat", "dog", "mouse", "monkey")
avector <- c(TRUE, TRUE, TRUE, FALSE, FALSE)
```

The operator `:` can be used to create a vector of numbers incremented by 1. Type the following:

```
1:10
3:8
-3:8
8:2
10:10
5.2:20
```

Of course, if you want to use these vectors, you need to *assign* them to something (e.g., `v <- 5.2:20`).

The function `seq` is somewhat more general than `:`. Consider the following:

```
seq(1, 10, by=1)
seq(3, 9, by=3)
seq(3, 9, length=10)
seq(2, by=0.2, length=8)
```

The function `rep` repeats stuff to create a vector. (The first argument gives a vector to be repeated; the second argument gives the number of times to repeat each element of the first argument.)

```
rep(2, 10)
rep(c(1,2,3), 5)
rep(c(1,2,3), c(2,4,5))
rep(1:3, 4)
rep(1:3, rep(4,3))
```

Subsetting vectors

One may refer to individual elements of a vector using square brackets, `[]`. For example, `x[3]` refers to the third element of the vector `x`. Use vectors of positive integers to refer to multiple elements of the vector, or *negative* integers to refer to all elements *except* those indicated.

```
x <- seq(2, 40, by=2)
length(x)
x[5]
x[c(1,3,9)]
x[-(1:10)]
x[-5]
```

One reason to do this is to replace certain elements of the vector with something new.

```
z <- c(1, 3, 5, 9)
z
z[2] <- -3
z
```

You may also index a vector using a *logical* vector with the same length as the vector under consideration. A logical vector is a vector of TRUE's and FALSE's.

```
y <- c(rep(TRUE,4), rep(FALSE,14), TRUE, TRUE)
length(y)
x[y]
```

The purpose of this is likely not immediately clear, so let us explain. The logical and other *operators* are useful for pulling out elements which meet certain criteria. Consider first the logical operators.

!	not
&	and (element-wise)
|	or (element-wise)

Try out these examples, and play around a bit.

```
a <- c(rep(c(TRUE, FALSE), 2), NA)
b <- c(rep(c(TRUE, FALSE), c(2,2)), FALSE)
a
b
!a
a & b
a | b
!(a | b)
!a | b
```

Now consider the following, even more important operators.

`==`	equal to
`!=`	not equal to
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to
`is.na()`	is “missing” (`NA`)

Here are some examples.

```
x <- c(1,5,3,NA,9,11,2,3)
x<=5
x>3 & x<11
is.na(x)
x[!is.na(x)]
x[!is.na(x) & x<5]
```

Subsetting matrices

You can also refer to portions of a matrix or data frame using square brackets by including a comma; indices before the comma refer to rows while indices after the comma refer to columns.

Let’s do some similar things with the example data `PlantGrowth`.

```
data(PlantGrowth)
summary(PlantGrowth)
PlantGrowth[PlantGrowth[,2]=="ctrl",]
summary(PlantGrowth[PlantGrowth[,2]=="ctrl",])
summary(PlantGrowth[PlantGrowth[,2]=="trt1",])
summary(PlantGrowth[PlantGrowth[,2]=="trt2",])
```

Hopefully your workspace still contains the object `mat`; if not, please read it into R again, using `read.csv`. Note that missing data (`NA`’s) are almost always a bit of a pain—in statistics generally, and in fiddling with data in R in particular.

```
mat[1:5,]
mat[,2]
mat[11:20, c(1,4:5)]
mat[is.na(mat[,2]), ]
mat[is.na(mat[,2]) & is.na(mat[,3]), ]
mat[is.na(mat[,2]) | is.na(mat[,3]), ]
mat[mat[,1]=="OPS imaging", ]
```

R as a calculator

You can use R as a fancy calculator. Most functions may be used on vectors or matrices, in which case they act on each element of the vector or matrix.

```
2 + 3 - 2^3
(2 + 3 - 2^3)*4
(2 + 3 - 2^3)/4
3^4
2 + 3^4
2 + (1:4)^4
sin(0.5)
log(seq(1, 2, length=11))
```

```
log10(seq(1, 100, length=11))
log2(c(1, 2, 4, 8, 16, 32))
x <- c(1, 5, 10, NA, 15)
sum(x)
sum(x, na.rm=TRUE)
prod(x, na.rm=TRUE)
```

Note that the function `log` calculates the natural logarithm. The functions `log10` and `log2` are used to calculate logarithms base 10 and base 2, respectively.

The functions `sum` and `prod` calculate the sum and product, respectively, of the elements of a vector.

Summary statistics

Of course, the basic summary statistics are available in R: `mean`, `median`, `sd`, `quantile`, `range`.

```
data(PlantGrowth)
mean(PlantGrowth[PlantGrowth[,2]=="ctrl",1])
mean(PlantGrowth[PlantGrowth[,2]=="trt1",1])
mean(PlantGrowth[PlantGrowth[,2]=="trt2",1])

z <- PlantGrowth[PlantGrowth[,2]=="ctrl", 1]
median(z)
sd(z)
x <- quantile(z, c(0.25, 0.75))
x
diff(x)
range(z)
diff(range(z))
```

All of these functions accept an argument `na.rm` for dealing with missing data (`NA`'s). By default, `na.rm=FALSE`, and these functions return `NA` if the input has any missing data. If one uses `na.rm=TRUE`, any missing values are removed prior to the calculations.

```
x <- mat[mat[,1]=="capillaroscopy",2]
x
sum(is.na(x))
mean(x)
mean(x, na.rm=TRUE)
median(x, na.rm=TRUE)
sd(x, na.rm=TRUE)
diff(quantile(x, c(0.25,0.75), na.rm=TRUE))
diff(range(x,na.rm=TRUE))
```

Loops

Here we give the briefest glimpse of programming in R. The function `for` may be used for repeating a task a number of times. Consider the following code.

```
me <- 1:4
for(z in 1:4) me[z] <- mean(mat[,z+1], na.rm=TRUE)
me
```

The function `for` is used to calculate the mean of columns 2, 3, 4, and 5 of the data set `mat` and save them in the vector `me`.

Such "loops" may be *nested*. This allows us to calculate the averages of each of the four numeric columns of `mat` after they have been split into two groups, according to the method for measurement. (We use the function `levels` to obtain the different categories of the factor `mat[,1]`.)

```
me <- matrix(nrow=2,ncol=4)
le <- levels(mat[,1])
for(i in 1:2)
  for(j in 1:4)
    me[i,j] <- mean(mat[mat[,1]==le[i], j+1], na.rm=TRUE)
me
```

Note that `for` loops may contain multiple commands, if those commands are enclosed in curly braces, `{}`.

```
s <- me <- matrix(nrow=2,ncol=4)
le <- levels(mat[,1])
for(i in 1:2) {
  for(j in 1:4) {
    me[i,j] <- mean(mat[mat[,1]==le[i], j+1], na.rm=TRUE)
    s[i,j] <- sd(mat[mat[,1]==le[i], j+1], na.rm=TRUE)
  }
}
me
```

The apply functions

Three functions which can make some tasks quite efficient in R, but are commonly found rather confusing, are `apply`, `sapply` and `tapply`. We attempt to describe these here.

The function `apply` is used to "apply" another function to each column (or row) of a matrix or data frame. For example, suppose we wish to obtain the average of each column (except the first one) of `mat[,-1]`. We showed above how to use a `for` loop to do this. An alternative is the following.

```
me <- apply(mat[,-1], 2, mean, na.rm=TRUE)
```

The first argument to `apply` is the input matrix or data frame. The third argument is the function to "apply." The second argument is taken to be 2 if one wishes to apply the function to each column, (or 1 if one wishes to apply the function to each row). Any arguments after the third are passed to the function being applied. Here, we use `na.rm=TRUE` so that any missing data is discarded.

As an alternative, we could have used the function `sapply`, which, for data frames, "applies" a function to each column of the data frame. By using `sapply`, we can get away without the "2."

```
me <- sapply(mat[,-1], mean, na.rm=TRUE)
```

We can use this to get the mean and SD of each numeric column, restricting attention to the measurements made by capillaroscopy.

```
x <- mat[mat[,1]=="capillaroscopy",]
me <- sapply(x[,-1], mean, na.rm=TRUE)
sd <- sapply(x[,-1], sd, na.rm=TRUE)
```

The function `tapply` is used to split a vector (say `a`) into groups defined by some other vector (say `b`) and then apply some function to each group. For example, consider the data `PlantGrowth`. The first column is a measure of growth; the second column is a factor with levels `"ctrl"` (control), `"trt1"` (treatment one), and `"trt2"` (treatment two). The following calculates the group-specific means and SDs of the plant growth.

```
data(PlantGrowth)
tapply(PlantGrowth[,1], PlantGrowth[,2], mean)
tapply(PlantGrowth[,1], PlantGrowth[,2], sd)
```

We can use the functions `sapply` and `tapply` together to get column means (and SDs) for each measurement method for the data `mat`. This allows us to avoid the nested pair of `for` loops that we used above.

```
sapply(mat[,-1], tapply, mat[,1], mean, na.rm=TRUE)
sapply(mat[,-1], tapply, mat[,1], sd, na.rm=TRUE)
```

What are these commands doing? Let's look at the first one.

1. `mat[,-1]` is the data frame `mat` with the first column (the measurement method) dropped.
2. The function `sapply` passes each column of `mat[,-1]`, one at a time, to the function `tapply`, along with the remaining arguments (`mean, na.rm=TRUE`).
3. The function `tapply` splits a column of `mat[,-1]` into the two groups defined by the factor `mat[,1]`, and passes each group to the function `mean`, along with the remaining argument, `na.rm=TRUE`.
4. Finally, the function `mean` calculates the group-specific mean, after first dropping any missing values.

Simple graphics

Let's look at how to make dotplots, boxplots and histograms. Let's start with dotplots, though there is no built-in function to create dotplots the way I like them. Let's use the `mat` data, and make a dotplot of the RBC velocities at rest, as measured by capillaroscopy and OPS imaging. First, we create a vector `x` containing these velocities and a vector `y` that is 1 if the measurement method is capillaroscopy and 2 otherwise.

```
x <- mat[,2]
y <- rep(2, length(x))
y[ mat[,1]=="capillaroscopy" ] <- 1
```

Now we can make the plot, using the function `plot`.

```
plot(x, y)
```

We may wish to "jitter" the values in `y`, add a better label to the x-axis, change the limits of the y-axis, and suspend plotting of the y-axis and y-axis label. The function `runif` returns a specified number of random numbers, uniformly distributed between specified limits.

```
u <- runif(length(y), -0.1, 0.1)
plot(x, y+u, xlab="RBC velocity (at rest)", ylim=c(0.5, 2.5),
     yaxt="n", ylab="")
```

We can also add some horizontal lines at 1 and 2. The function `abline` adds lines to a plot. The argument `h` is used to get horizontal lines. `lty=2` makes them dashed lines, and `col="gray"` makes them gray.

```
abline(h=c(1,2), lty=2, col="gray")
```

Boxplots can be easier.

```
boxplot(velo.rest ~ method, data=mat)
```

We can also do histograms.

```
par(mfrow=c(2,1))
hist(mat[ mat[,1]=="capillaroscopy", 2], main="Capillaroscopy",
     xlab="RBC velocity (at rest)")
hist(mat[ mat[,1]=="OPS imaging", 2], main="OPS imaging",
     xlab="RBC velocity (at rest)")
```

The function `par` is used for detailed control of graphics in R. The argument `mfrow` is used to make multiple plots in one plotting window. `mfrow=c(2,1)` is used to get two rows in one column of plots.

R commands discussed in this lab

`<-`	`read.csv`	`read.table`
`?`	`help`	`ls`
`objects`	`q`	`summary`
`plot`	`points`	`data`
`c`	`:`	`seq`
`rep`	`sin`	`log`
`log10`	`log2`	`sum`
`prod`	`mean`	`median`
`sd`	`quantile`	`range`
`diff`	`for`	`levels`
`apply`	`sapply`	`tapply`
`abline`	`boxplot`	`par`
`hist`		

Lab 1 **Statistics for Laboratory Scientists**
Due: 8 Feb 2006

1. Specify R code, using the function `rep`, to create the vector
   (1, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 4, 4, 5).
2. Specify R code for pulling out the rows of the mat object for which the fourth column is *not missing* and is less than 9.
3. Use R to calculate the following sum. Please give the code that you used.

$$\log_{10} 2 + \log_{10} 4 + \log_{10} 6 + \log_{10} 8 + \ldots \log_{10} 1000$$

4. Specify R code for converting 50, 65, 80, and 95 degrees Fahrenheit to the corresponding temperatures in celsius using the formula $C = 5(F - 32)/9$.
5. Fill in [a], [b], [c], and [d] in the following table for the data set `mat`.

		mean (SD) of	
method	rest/v.o.	RBC velocity (mm/s)	capillary diameter (μm)
capillaroscopy	rest	0.77 (0.24)	[c]
	v.o.	[a]	12.0 (1.6)
OPS imaging	rest	[b]	11.2 (2.0)
	v.o.	0.15 (0.09)	[d]