Chapter 6

Introduction to S

6.1 History and Terminology

Most major statistical packages are computer programs that have their own control language. The syntax goes with one computer program and just one. The SAS language controls the SAS software, and that's it. Minitab syntax controls Minitab, and that's it. S is a little different. Originally, S was both a program and a language; they were developed together at the former AT&T Bell Labs starting in the late 1970's. Like the unix operating system (also developed around the same time at Bell Labs, among other places), S was open-source and in the public domain. "Open-source" means that the actual program code (initially in Fortran, later in C) was public. It was free to anyone with the expertise to compile and install it.

Later, S was spun off into a private company that is now called Insightful Corporation. They incorporated both the S syntax and the core of the S software into a commercial product called S-Plus. S-Plus is *not* open-source. The "Plus" part of S-Plus is definitely proprietary. S-Plus uses the S language, but the S language is not owned by Insightful Corporation. It's in the public domain.

R also uses the S language. This is a unix joke. You know, like how the unix `less` command is an improved version of `more`. Get it? In fact unix itself is an improvement on an earlier operating system called Multics. Get it? Apparently, this joke is so funny that they keep making it at every opportunity.

R is produced by a team of volunteer programmers and statisticians, under the auspices of the Free Software Foundation. It is an official GNU project. What is GNU? GNU stands for "GNU's Not Unix." The recursive nature of this answer is another unix joke. Get it?

The GNU project was started by a group of programmers (led by the great Richard Stallman, author of `emacs`) who believed that software should be open-source and free for anyone to use, copy or modify. They were irritated by the fact that corporations could take unix, enhance it in a few minor (or major) ways, and copyright the result. Solaris, the version of unix used on most Sun workstations, is an example. An even more extreme example is Macintosh OS X, which is just a very elaborate graphical shell running on top of Berkeley Standard Distribution unix.

The GNU operating system was to look and act like unix, but to be rewritten from the ground up, and legally protected in such a way that it could not be incorporated into any piece of software that was proprietary. Anybody would be able to modify it and even sell the modified version – or the original. But any modified version, like the original, would have to be open-source, with no restrictions on copying or use of the software. The main GNU project has been successful; the result is called `linux`.

R is another successful GNU project. The R development team rewrote the S software from scratch without using any of the original code. It runs under the unix, linux, MS Windows and Macintosh operating systems. It is free, and easy to install. Go to `http://www.R-project.org` to obtain a copy of the software or more information. There are also links on the course home page.

While they were redoing S, the R development team quietly fixed an extremely serious problem. While the S *language* provides a beautiful environment for simulation and customized computer-intensive statistical methods, the S *software* did the job in a terribly inefficient way. The result was that big simulations ran very slowly, and long-running jobs often aborted or crashed the system unless special and very unpleasant measures were taken. S-Plus, because it is based on the original S code, inherits these problems. R is immune to them.

Anyway, S is a language, and R is a piece of software that is controlled by the S language. The discussion that follows will usually refer to S, but all the examples will use the R implementation of S — specifically, R version 2.0.0 running under unix on fisher. Mostly, what we do here will also work in S-Plus. Why would you ever want to use S-Plus? Well, it does have some capabilities that R does not have (yet), particularly in the areas of survival analysis and spatial statistics.

6.2 S as a Calculator

To start R, type "R" and return at the unix prompt. Then we get

```
R : Copyright 2004, The R Foundation for Statistical Computing
Version 2.0.0  (2004-10-04), ISBN 3-900051-07-0

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

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

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

>
```

S is built around *functions*. As you can see above, even asking for help and quitting are functions (with no arguments).

The primary mode of operation of S is line oriented and interactive. It is quite unix-like, with all the good and evil that implies. S gives you a prompt that looks like a "greater than" sign. You type a command, press Return (Enter), and the program does what you say. Its default behaviour is to return the value of what you type, often a numerical value. In the first example, we receive the ">" prompt, type "1+1" and then press the Enter key. S tells us that the answer is 2. Then we obtain $2^3 = 8$.

```
> 1+1
[1] 2
> 2^3 # Two to the power 3
[1] 8
```

What is this [1] business? It's clarified when we ask for the numbers from 1 to 30.

```
> 1:30
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30
```

S will give you an array of numbers in compact form, using a number in brackets to indicate the ordinal position of the first item on each line. When it answered "1+1" with `[1] 2`, it was telling us that the first item in this array (of one item) was 2.

S has an amazing variety of mathematical and statistical functions. For example, the gamma function is defined by $\Gamma(a) = \int_0^\infty e^{-t}t^{a-1}\,dt$, and with a bit of effort you can prove that $\Gamma(\frac{1}{2}) = \sqrt{\pi}$. Note that everything to the left of a # is a comment.

```
> gamma(.5)^2      # Gamma(1/2) = Sqrt(Pi)
[1] 3.141593
```

Assignment of values is carried out by a "less than" sign followed *immediately* by a minus sign; it looks like an arrow pointing to the left. The command `x <- 1` would be read "x gets 1."

```
> x <- 1           # Assigns the value 1 to x
> y <- 2
> x+y
[1] 3
> z <- x+y
> z
[1] 3
> x <- c(1,2,3,4,5,6)    #  Collect these numbers; x is now a vector
```

Originally, x was a single number. Now it's a vector (array) of 6 numbers. S operates naturally on vectors.

```
> y <- 1 + 2*x
> cbind(x,y)
     x  y
[1,] 1  3
[2,] 2  5
[3,] 3  7
```

```
[4,] 4  9
[5,] 5 11
[6,] 6 13
```

The cbind command binds the vectors x and y into columns. The result is a matrix whose value is returned (displayed on the screen), since it is not assigned to anything.

The bracket (subscript) notation for selecting elements of an array is very powerful. The following is just a simple example.

```
> z <- y[x>4]           # z gets y such that x > 4
> z
[1] 11 13
```

If you put an array of integers inside the brackets, you get those elements, in the order indicated.

```
> y[c(6,5,4,3,2,1)] # y in opposite order
[1] 13 11  9  7  5  3
> y[c(2,2,2,3,4)] # Repeats are okay
[1] 5 5 5 7 9
> y[7] # There is no seventh element.  NA is the missing value code
[1] NA
```

Most operations on arrays are performed element by element. If you take a function of an array, S applies the function to each element of the array and returns an array of function values.

```
> z <- x/y       # Most operations are performed element by element
> cbind(x,y,z)
     x  y         z
[1,] 1  3 0.3333333
[2,] 2  5 0.4000000
[3,] 3  7 0.4285714
[4,] 4  9 0.4444444
[5,] 5 11 0.4545455
[6,] 6 13 0.4615385
> x <- seq(from=0,to=3,by=.1)   # A sequence of numbers
> y <- sqrt(x)
```

S is a great environment for producing high-quality graphics, though we won't use it much for that. Here's just one example. We activate the `pdf` graphics device, so that all subsequent graphics in the session are written to a file that can be viewed with Adobe's *Acrobat Reader*. We then make a line plot of the function $y = \sqrt{x}$, and quit.

```
> pdf("testor.pdf")
> plot(x,y,type='l')       # That's a lower case L
> q()
```

Actually, graphics are a good reason to download and install R on your desktop or laptop computer. By default, you'll see nice graphics output on your screen. Under unix, it's a bit of a pain unless you're in an X-window environment (and we're assuming that you are not). You have to transfer that `pdf` file somewhere and view it with *Acrobat* or *Acrobat Reader*.

Continuing the session, a couple of interesting things happen when we quit. First, we are asked if we want to save the "workspace image." The responses are Yes, No and Cancel (don't quit yet). If you say Yes, R will write a file containing all the objects (x, y and z in the present case) that have been created in the session. Next time you start R, your work will be "restored" to where it was when you quit.

```
Save workspace image? [y/n/c]: y
credit.erin > ls
testor.pdf
```

Notice that when we type `ls`, to list the files, we see only `testor.pdf`, the pdf file containing the plot of $y = \sqrt{x}$. Where is the workspace image? It's an invisible file; type `ls -a` to see *all* the files.

```
credit.erin > ls -a
./                ../                .RData            testor.pdf
```

There it is: `.RData`. Files beginning with a period don't show up in output to the `ls` command unless you use the `-a` option. R puts `.RData` *in the (sub)directory from which R was invoked.* This means that if if you have a separate subdirectory for each project or assignment (not a bad way to organize your work), R will save the workspace from each job in a separate place, so that you can have variables with names like x in more than one place, containing different numbers. When we return to R,

```
credit.erin > R

R : Copyright 2004, The R Foundation for Statistical Computing
Version 2.0.0  (2004-10-04), ISBN 3-900051-07-0

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

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

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

[Previously saved workspace restored]

> ls()
[1] "x" "y" "z"

> max(x)
[1] 3
```

All the examples so far (and many of the examples to follow) are interactive, but for serious work, it's better to work with a command file. Put your commands in a file and execute them all at once. Suppose your commands are in a file called `commands.R`. At the S prompt, you'd execute them with `source("commands.R")`. From the unix prompt, you'd do it like this. The `--vanilla` option invokes a "plain vanilla" mode of operation suitable for this situation.

```
credit.erin > R --vanilla < commands.R > homework.out
```

For really big simulations, you may want to run the job in the background at a lower priority. The `&` suffix means run it in the background. `nohup` means don't hang up on me when I log out. `nice` means be nice to other users, and run it at a lower priority.

```
credit.erin > nohup nice R --vanilla < bvnorm.R > bvnorm.out &
```

6.3 S as a Stats Package

Here, we illustrate traditional multiple regression with S, testing the parallel slopes assumption for the metric cars data. Compare `mcars.sas` and `mcars.lst`. There are lots of comment statements that help explain what is going on. More detail will be given in lecture. In addition, the course home page has a link to a nice 100-page manual. If you plan to use R seriously, you should download this manual and read it. But if you come to lecture, you probably don't need to look at it for the purposes of this class.

Here is the "program" named `lesson2.R`.

```
#####################################################################
# lesson2.R: execute with    R --vanilla < lesson2.R > lesson2.out #
#####################################################################

datalist <-  scan("mcars.dat",list(id=0,country=0,kpl=0,weight=0,length=0))
# datalist is a linked list.
datalist
# There are other ways to read raw data. See help(read.table).
weight <- datalist$weight ; length <- datalist$length ; kpl <- datalist$kpl
country <- datalist$country
cor(cbind(weight,length,kpl))
# The table command gives a bare-bones frequency distribution
table(country)
# That was a matrix. The numbers 1 2 3 are labels.
# You can save it, and you can get at its contents
countrytable <- table(country)
countrytable[2]
# There is an "if" function that you could use to make dummy variables,
# but it's easier to use factor.
countryfac <- factor(country,levels=c(1,2,3),
                     label=c("US","Japanese","European"))
# This makes a FACTOR corresponding to country, like declaring it
# to be categorical. How are dummy variables being set up?
contrasts(countryfac)
# The first level specified is the reference category. You can get a
# different reference category by specifying the levels in a different order.
cntryfac <- factor(country,levels=c(2,1,3),
                     label=c("Japanese","US","European"))
contrasts(cntryfac)
# Test interaction. For comparison, with SAS we got F = 11.5127, p < .0001
# First fit (and save!) the reduced model. lm stands for linear model.
redmod <- lm(kpl ~ weight+cntryfac)
# The object redmod is a linked list, including lots of stuff like all the
# residuals. You don't want to look at the whole thing, at least not now.
summary(redmod)

# Full model is same stuff plus interaction. You COULD specify the whole thing.
```

```
fullmod <- update(redmod,. ~ . + weight*cntryfac)
anova(redmod,fullmod)
# The ANOVA summary table is a matrix. You can get at its (i,j)th element.
aovtab <- anova(redmod,fullmod)
aovtab[2,5] # The F statistic
aovtab[2,6] < .05       #    p < .05 -- True or false?
1>6 # Another example of an expression taking the logical value true or false.
```

Here is the output file `lesson2.out`. Note that it shows the commands. This would not happen if you used `source("lesson2.R")` from within R. I have added some blank lines to the output file to make it more readable.

```
> ####################################################################
> # lesson2.R: execute with    R --vanilla < lesson2.R > lesson2.out #
> ####################################################################
>
> datalist <-  scan("mcars.dat",list(id=0,country=0,kpl=0,weight=0,length=0))
Read 100 records
> # datalist is a linked list.
> datalist
$id
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
 [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
 [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
 [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
 [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
 [91]  91  92  93  94  95  96  97  98  99 100

$country
  [1] 1 2 1 1 1 1 3 1 3 1 2 1 1 3 2 1 1 1 3 2 1 1 1 3 2 1 1 1 1 1 2 1 2 1 1 1 3
 [38] 3 1 1 1 1 1 1 1 1 1 1 3 1 1 1 1 1 1 1 1 3 1 1 1 2 1 1 1 1 2 2 1 1 1 2 1 1
 [75] 1 2 2 3 1 1 1 1 3 1 1 1 1 1 1 1 1 1 1 1 3 3 3 1 1 1

$kpl
  [1]  5.04 10.08  9.24  7.98  7.98  7.98  9.66  7.56  5.88 10.92 12.60  8.40
 [13]  8.82 10.92  7.56 12.18  5.04  5.88  7.14 13.02  5.88 10.92  6.72 10.50
 [25]  8.82  5.88  6.72 11.76  9.24  7.56  7.56 11.76 10.50  5.88  9.24  7.98
 [37]  7.14 17.22  6.72  7.98  7.14  6.30  5.88  8.82  9.24  9.24  5.88  8.40
 [49] 10.50  9.24  7.56  7.56 12.60 12.60  7.98  7.56  8.40  9.66  7.56  6.30
 [61]  5.88  7.56 10.08  5.04  8.82 11.76 14.70 10.08  9.24 10.92 10.50  7.56
 [73]  8.82  7.56  7.14  7.56 10.08  8.82  5.88  8.82  8.82 10.08 17.22  6.72
 [85]  9.24  5.88  7.56 11.76  7.98  8.82  5.88  5.88  7.14  5.04 17.22 17.22
 [97]  7.14 10.50  6.72  7.56

$weight
  [1] 2178.0 1026.0 1188.0 1444.5 1485.0 1485.0  972.0 1665.0 1539.0 1003.5
 [11]  891.0 1273.5 1930.5  823.5 1084.5  949.5 2178.0 1755.0 1426.5  990.0
 [21] 1827.0 1134.0 1813.5 1192.5 1237.5 1858.5 1813.5 1062.0 1431.0 1651.5
 [31] 1201.5 1062.0 1008.0 1858.5 1318.5 1440.0 1273.5  918.0 1813.5 1530.0
 [41] 1683.0 1836.0 1723.5 1827.0 1449.0 1318.5 1858.5 1273.5  868.5 1318.5
 [51] 1665.0 1620.0  954.0  954.0 1516.5 1665.0 1462.5  972.0 1665.0 1674.0
 [61] 1755.0 1201.5 1237.5 2178.0 1930.5 1062.0  922.5 1026.0 1449.0 1134.0
 [71]  990.0 1084.5 1930.5 1516.5 1507.5 1084.5 1026.0  958.5 1858.5 1930.5
 [81] 1192.5 1237.5  918.0 1813.5 1449.0 1755.0 1561.5 1062.0 1489.5 1192.5
 [91] 1827.0 1755.0 1683.0 2178.0  918.0  918.0 1426.5  990.0 1660.5 1498.5
```

```
$length
  [1] 591.82 431.80 426.72 510.54 502.92 502.92 436.88 543.56 487.68 431.80
 [11] 391.16 495.30 518.16 360.68 441.96 414.02 591.82 518.16 490.22 419.10
 [21] 561.34 462.28 523.24 449.58 467.36 551.18 523.24 431.80 490.22 553.72
 [31] 444.50 431.80 436.88 551.18 472.44 505.46 480.06 393.70 523.24 508.00
 [41] 558.80 563.88 510.54 558.80 508.00 472.44 551.18 495.30 393.70 472.44
 [51] 543.56 523.24 414.02 414.02 508.00 543.56 497.84 436.88 543.56 538.48
 [61] 518.16 444.50 454.66 591.82 518.16 431.80 416.56 431.80 508.00 462.28
 [71] 419.10 441.96 518.16 502.92 439.42 441.96 431.80 408.94 551.18 518.16
 [81] 454.66 454.66 393.70 523.24 508.00 518.16 502.92 431.80 502.92 454.66
 [91] 561.34 518.16 558.80 591.82 393.70 393.70 490.22 419.10 538.48 510.54

> # There are other ways to read raw data. See help(read.table).

> weight <- datalist$weight ; length <- datalist$length ; kpl <- datalist$kpl
> country <- datalist$country
> cor(cbind(weight,length,kpl))
           weight     length        kpl
weight  1.0000000  0.9462018 -0.7704194
length  0.9462018  1.0000000 -0.7899859
kpl    -0.7704194 -0.7899859  1.0000000

> # The table command gives a bare-bones frequency distribution
> table(country)
country
 1  2  3
73 13 14

> # That was a matrix. The numbers 1 2 3 are labels.
> # You can save it, and you can get at its contents
> countrytable <- table(country)
> countrytable[2]
 2
13

> # There is an "if" function that you could use to make dummy variables,
> # but it's easier to use factor.
> countryfac <- factor(country,levels=c(1,2,3),
+                       label=c("US","Japanese","European"))
> # This makes a FACTOR corresponding to country, like declaring it
> # to be categorical. How are dummy variables being set up?

> contrasts(countryfac)
         Japanese European
US              0        0
Japanese        1        0
European        0        1

> # The first level specified is the reference category. You can get a
> # different reference category by specifying the levels in a different order.
> cntryfac <- factor(country,levels=c(2,1,3),
+                       label=c("Japanese","US","European"))

> contrasts(cntryfac)
         US European
Japanese  0        0
```

```
US        1        0
European  0        1



> # Test interaction. For comparison, with SAS we got F = 11.5127, p < .0001

> # First fit (and save!) the reduced model. lm stands for linear model.
> redmod <- lm(kpl ~ weight+cntryfac)

> # The object redmod is a linked list, including lots of stuff like all the
> # residuals. You don't want to look at the whole thing, at least not now.

> summary(redmod)

Call:
lm(formula = kpl ~ weight + cntryfac)

Residuals:
    Min      1Q  Median      3Q     Max 
-3.0759 -0.9810 -0.1919  0.4725  5.0795 

Coefficients:
                   Estimate Std. Error t value Pr(>|t|)    
(Intercept)      16.2263357  0.7631228  21.263   <2e-16 ***
weight           -0.0060407  0.0005708 -10.583   <2e-16 ***
cntryfacUS        1.2361472  0.5741299   2.153   0.0338 *  
cntryfacEuropean  1.4595914  0.6456563   2.261   0.0260 *  
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1 

Residual standard error: 1.676 on 96 degrees of freedom
Multiple R-Squared: 0.618,      Adjusted R-squared: 0.606 
F-statistic: 51.76 on 3 and 96 DF,  p-value:     0 

> 
> # Full model is same stuff plus interaction. You COULD specify the whole thing.

> fullmod <- update(redmod,. ~ . + weight*cntryfac)

> anova(redmod,fullmod)

Analysis of Variance Table

Model 1: kpl ~ weight + cntryfac
Model 2: kpl ~ weight + cntryfac + weight:cntryfac
  Res.Df     RSS Df Sum of Sq      F    Pr(>F)    
1     96 269.678                                  
2     94 216.617  2    53.061 11.513 3.372e-05 ***
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1 

> # The ANOVA summary table is a matrix. You can get at its (i,j)th element.
> aovtab <- anova(redmod,fullmod)
> aovtab[2,5] # The F statistic
[1] 11.51273
```

```
> aovtab[2,6] < .05       #    p < .05 -- True or false?
[1] TRUE

> 1>6 # Another example of an expression taking the logical value true or false.
[1] FALSE
$
```

6.4 Random Numbers and Simulation

S is a superb environment for simulation and customized computer-intensive statistical methods. That's really why it is being discussed. Simulation is an extremely general and powerful method for calculating probabilities that are difficult to figure out by other means. Well, technically it's a way of *estimating* those probabilities, based a sample of random numbers. Before proceeding, we need a couple of definitions.

We will use the term **statistical experiment** to refer to any procedure whose outcome is not known in advance with certainty. The most standard, and the most boring example of a statistical experiment is to toss a coin and observe whether it comes up heads or tails. We *model* statistical experiments by pretending that they obey the laws of probability.

When we carry out a statistical experiment, the things that can happen (the things we pay attention to) are called **outcomes**. Sets of outcomes are called **events**. For example, if you roll a die, the outcomes are the numbers 1 through 6, and "even" is an event consisting of the outcomes $\{2, 4, 6\}$.

The main principle we will use is called the **Law of Large Numbers**. There are quite a few versions of this law. Here's a verbal statement of the one we will use. *If a statistical experiment is carried out independently a very large number of times (trials) under identical conditions, the proportion of times an event occurs approaches the probability of the event, as the number of trials increases.* In elementary texts, this is sometimes used as the definition of probability. But in more sophisticated treatments, it's a theorem.

For example, suppose you are planning to test differences between means for an experimental versus a control group, and you have strong reason to believe that your data will have a chi-square distribution within groups. You are going to log-transform the data to take care of the positive skewness of the chi-square, and then use a common t-test.

Suppose data in the experimental group is chi-square with one degree of freedom (so the population mean is one and the variance is two), and the data in the control group is chi-square with two degree of freedom (so the population mean is two and the variance is four). What is the power of the t-test on the transformed data with $n = 20$ in each group?

Nobody can figure this out mathematically, but it's pretty easy with simulation. Here's how to do it.

1. Using the random number generator in some software package, generate 20 independent chi-square values with one degree of freedom, and 20 independent chi-square values with two degrees of freedom.

2. Log transform all the values.

3. Compute the t-test.

4. Check to see if $p < 0.05$.

Do this a large number of times. The proportion of times $p < 0.05$ is the power — or more precisely, a Monte Carlo estimate of the power.

The number of times a statistical experiment is repeated is called the **Monte Carlo sample size**. How big should the Monte Carlo sample size be? It depends on how much precision you need. We will produce confidence intervals for all our Monte Carlo estimates, to get a handle on the probable margin of error of the statements we make. Sometimes, Monte Carlo sample size can be chosen by a power analysis. More details will be given later.

The example below shows several simulations of taking a random sample of size 20 from a standard normal population ($\mu = 0$, $\sigma^2 = 1$). Now actually, computer-generated random numbers are not *really* random; they are completely determined by the execution of the computer program that generates them. The most common (and best) random number generators actually produce a *stream* of pseudo-random numbers that will eventually repeat. In the good ones (and R uses a good one), "eventually" means after the end of the universe. So the pseudo-random numbers that R produces really *act* random, even though they are not. It's safe to say that they come closer to satisfying the assumptions of significance tests than any real data.

If you don't instruct it otherwise, R will use the system clock to decide on *where* in the random number stream it should begin. But sometimes you want to be able to reproduce the results of a simulation exactly, say if you're debugging your program, or you have already spent a lot of time making a graph based on it. In this case you can control the starting place in the random number stream, by setting the "seed" of the random number generator. The seed is a big integer; I used 12345 just as an example.

```
> rnorm(20) # 20 standard normals
 [1]  0.24570675 -0.38857202  0.47642336  0.75657595  0.71355871 -0.74630629
 [7] -0.02485569  1.93346357  0.15663167  1.16734485  0.57486449  1.32309413
[13]  0.63712982  2.00473940  0.04221730  0.70896768  0.42128470 -0.12115292
[19]  1.42043470 -1.04957255

> set.seed(12345) # Be able to reproduce the stream of pseudo-random numbers.
> rnorm(20)
 [1]  0.77795979 -0.89072813  0.05552657  0.67813726  0.80453336 -0.35613672
 [7] -1.24182991 -1.05995791 -2.67914037 -0.01247257 -1.22422266  0.88672878
[13] -1.32824804 -2.73543539  0.40487757  0.41793236 -1.47520817  1.15351981
[19] -1.24888614  1.11605686
> rnorm(20)
 [1]  0.866507371  2.369884323  0.393094088 -0.970983967 -0.292948278
 [6]  0.867358962  0.495983546  0.331635970  0.702292771  2.514734599
[11]  0.522917841 -0.194668990 -0.089222053 -0.491125596 -0.452112445
[16] -0.515548826 -0.244409517 -0.008373764 -1.459415684 -1.433710170

> set.seed(12345)
> rnorm(20)
 [1]  0.77795979 -0.89072813  0.05552657  0.67813726  0.80453336 -0.35613672
 [7] -1.24182991 -1.05995791 -2.67914037 -0.01247257 -1.22422266  0.88672878
[13] -1.32824804 -2.73543539  0.40487757  0.41793236 -1.47520817  1.15351981
[19] -1.24888614  1.11605686
```

The `rnorm` function is probably the most important random number generator, because it is used so often to investigate the properties of statistical tests that assume a normal distribution. Here is some more detail about `rnorm`.

```
> help(rnorm)
Normal                package:base                R Documentation

The Normal Distribution

Description:

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

Usage:

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

```
Arguments:

     x,q: vector of quantiles.

       p: vector of probabilities.

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

    mean: vector of means.

      sd: vector of standard deviations.

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

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

Details:

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

     The normal distribution has density

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

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

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

Value:

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

References:

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

See Also:

     `runif' and `.Random.seed' about random number generation, and
     `dlnorm' for the Lognormal distribution.

Examples:

     dnorm(0) == 1/ sqrt(2*pi)
     dnorm(1) == exp(-1/2)/ sqrt(2*pi)
     dnorm(1) == 1/ sqrt(2*pi*exp(1))

     ## Using "log = TRUE" for an extended range :
```

```
par(mfrow=c(2,1))
plot(function(x)dnorm(x, log=TRUE), -60, 50, main = "log { Normal density }")
curve(log(dnorm(x)), add=TRUE, col="red",lwd=2)
mtext("dnorm(x, log=TRUE)", adj=0); mtext("log(dnorm(x))", col="red", adj=1)

plot(function(x)pnorm(x, log=TRUE), -50, 10, main = "log { Normal Cumulative }")
curve(log(pnorm(x)), add=TRUE, col="red",lwd=2)
mtext("pnorm(x, log=TRUE)", adj=0); mtext("log(pnorm(x))", col="red", adj=1)
```

After generating normal random numbers, the next most likely thing you might want to do is randomly scramble some existing data values. The `sample` function will select the elements of some array, either with replacement or without replacement. If you select all the numbers in a set without replacement, you've rearranged them in a random order. This is the basis of randomization tests. Sampling *with* replacement is the basis of the bootstrap.

```
> help(sample)
sample                package:base                R Documentation

Random Samples and Permutations

Description:

     `sample' takes a sample of the specified size from the elements of
     `x' using either with or without replacement.

Usage:

     sample(x, size, replace = FALSE, prob = NULL)

Arguments:

       x: Either a (numeric, complex, character or logical) vector of
          more than one element from which to choose, or a positive
          integer.

    size: A positive integer giving the number of items to choose.

 replace: Should sampling be with replacement?

    prob: A vector of probability weights for obtaining the elements of
          the vector being sampled.

Details:

     If `x' has length 1, sampling takes place from `1:x'.

     By default `size' is equal to `length(x)' so that `sample(x)'
     generates a random permutation of the elements of `x' (or `1:x').

     The optional `prob' argument can be used to give a vector of
     weights for obtaining the elements of the vector being sampled.
     They need not sum to one, but they should be nonnegative and not
     all zero.  If 'replace' is false, these probabilities are applied
     sequentially, that is the probability of choosing the next item is
     proportional to the probabilities amongst the remaining items. The
     number of nonzero weights must be at least 'size' in this case.

Examples:

     x <- 1:12
     # a random permutation
     sample(x)
     # bootstrap sampling
     sample(x,replace=TRUE)

     # 100 Bernoulli trials
     sample(c(0,1), 100, replace = TRUE)
```

6.4.1 Illustrating the Regression Artifact by Simulation

In the ordinary use of the English language, to "regress" means to go backward. In Psychiatry and Abnormal Psychology, the term "regression" is used when a person's behaviour changes to become more typical of an earlier stage of development — like when an older child starts wetting the bed, or an adult under extreme stress sucks his thumb. Isn't this a strange word to use for the fitting of hyperplanes by least-squares?

The term "regression" (as it is used in Statistics) was coined by Sir Francis Galton (1822-1911). For reasons that now seem to have a lot to do with class privilege and White racism, he was very interested in heredity. Galton was investigating the relationship between the heights of fathers and the heights of sons. What about the mothers? Apparently they had no height.

Anyway, Galton noticed that very tall fathers tended to have sons that were a bit shorter than they were, though still taller than average. On the other hand, very short fathers tended to have sons that were taller than they were, though still shorter than average. Galton was quite alarmed by this "regression toward mediocrity" or "regression toward the mean," particularly when he found it in a variety of species, for a variety of physical characteristics. See Galton's "Regression towards mediocrity in hereditary stature", *Journal of the Anthropological Institute* **15** (1886), 246-263. It even happens when you give a standardized test twice to the same people. The people who did the very best the first time tend to do a little worse the second time, and the people who did the very worst the first time tend to do a little better the second time.

Galton thought he had discovered a Law of Nature, though in fact the whole thing follows from the algebra of least squares. Here's a verbal alternative. Height is influenced by a variety of chance factors, many of which are *not* entirely shared by fathers and sons. These include the mother's height, environment and diet, and the vagaries of genetic recombination. You could say that the tallest fathers included some who "got lucky," if you think it's good to be tall (Galton did, of course). The sons of the tall fathers had some a genetic predisposition to be tall, but on average, they didn't get as lucky as their fathers in every respect. A similar argument applies to the short fathers and their sons.

This is the basis for the so-called **regression artifact**. Pre-post designs with extreme groups are doomed to be misleading. Programs for the disadvantaged "work" and programs for the gifted "hurt." This is a very serious methodological trap that has doomed quite a few evaluations of social programs, drug treatments – you name it.

Is this convincing? Well, the argument above may be enough for some people. But perhaps if it's illustrated by simulation, you'll be even more convinced. Let's find out.

Suppose an IQ test is administered to the same 10,000 students on two occasions. Call the scores `pre` and `post`. After the first test, the 100 individuals who did worst are selected for a special remedial program, but it does *nothing*. And, the 100 individuals who did best on the pre-test get a special program for the gifted, but it does *nothing*. We do a matched t-test on the students who got the remedial program, and a matched $t - test$ on the students who got the gifted program.

What should happen? If you followed the stuff about regression artifacts, you'd expect significant improvement from the students who got the remedial program, and significant deterioration from the students who got the gifted program – even though in fact, both programs are completely ineffective (and harmless). How will we simulate this?

According to classical psychometric theory, a test score is the sum of two independent pieces, the *True Score* and *measurement error*. If you measure an individual twice, she has the same True Score, but the measurement error component is different.

True Score and measurement error have population variances. Because they are independent, the variance of the observed score is the sum of the true score variance and the error variance. The proportion of the observed score variance that is True Score variance is called the test's *reliability*. Most "intelligence" tests have a mean of 100, a standard deviation of 15, and a reliability around 0.80.

So here's what we do. Making everything normally distributed and selecting parameter values so the means, standard deviations and reliability come out right, we

- Simulate 10,000 true scores.
- Simulate 10,000 measurement errors for the pre-test and an independent 10,000 measurement errors for the post-test.
- Calculate 10,000 pre-test scores by `pre = True + error1`.
- Calculate 10,000 post-test scores by `pre = True + error2`.
- Do matched t-tests on the individuals with the 100 worst and the 100 best pre-test scores.

This procedure is carried out *once* by the program `regart.R`. In addition, `regart.R` carries out a matched t-test on the entire set of 10,000 pairs, just to verify that there is no systematic change in "IQ" scores.

```
# regart.R    Demonstrate Regression Artifact
###################### Setup ######################
N <- 10000 ; n <- 100
truevar <- 180 ; errvar <- 45
truesd <- sqrt(truevar) ; errsd <- sqrt(errvar)
# set.seed(44444)
# Now define the function ttest, which does a matched t-test

ttest <- function(d) # Matched t-test. It operates on differences.
    {
    ttest <- numeric(4)
    names(ttest) <- c("Mean Difference"," t "," df "," p-value ")
    ave <- mean(d) ; nn <- length(d) ; sd <- sqrt(var(d)) ; df <- nn-1
    tstat <- ave*sqrt(nn)/sd
    pval <- 2*(1-pt(abs(tstat),df))
    ttest[1] <- ave ; ttest[2] <- tstat; ttest[3] <- df; ttest[4] <- pval
    ttest # Return the value of the function
    }
```

```
#####################################################

error1 <- rnorm(N,0,errsd) ; error2 <- rnorm(N,0,errsd)
truescor <- rnorm(N,100,truesd)
pre <- truescor+error1 ; rankpre <- rank(pre)

# Based on their rank on the pre-test, we take the n worst students and
# place them in a special remedial program, but it does NOTHING.

# Based on their rank on the pre-test, we take the n best students and
# place them in a special program for the gifted, but it does NOTHING.

post <- truescor+error2
diff <- post-pre # Diff represents "improvement."
                 # But of course diff = error2-error1 = noise

cat("\n") # Skip a line
cat("------------------------------------ \n")
dtest <- ttest(diff)
cat("Test on diff (all scores) \n") ; print(dtest) ; cat("\n")

remedial <- diff[rankpre<=n] ; rtest <- ttest(remedial)
cat("Test on Remedial \n") ; print(rtest) ; cat("\n")

gifted <- diff[rankpre>=(N-n+1)] ; gtest <- ttest(gifted)
cat("Test on Gifted \n") ; print(gtest) ; cat("\n")
cat("------------------------------------ \n")
```

The `ttest` function is a little unusual because it takes a whole vector of numbers (length unspecified) as input, and returns an array of 4 values. Often, functions take one or more numbers as input, and return a single value. We will see some more examples shortly. At the R prompt,

```
> source("regart.R")

------------------------------------
Test on diff (all scores)
Mean Difference              t              df         p-value
   1.872566e-02    1.974640e-01    9.999000e+03    8.434685e-01

Test on Remedial
Mean Difference              t              df         p-value
   7.192531e+00    8.102121e+00    9.900000e+01    1.449729e-12

Test on Gifted
Mean Difference              t              df         p-value
  -8.311569e+00   -9.259885e+00    9.900000e+01    4.440892e-15

------------------------------------
> source("regart.R")

------------------------------------
Test on diff (all scores)
Mean Difference              t              df         p-value
   2.523976e-02    2.659898e-01    9.999000e+03    7.902525e-01

Test on Remedial
Mean Difference              t              df         p-value
   5.510484e+00    5.891802e+00    9.900000e+01    5.280147e-08

Test on Gifted
Mean Difference              t              df         p-value
      -8.972938      -10.783356       99.000000        0.000000

------------------------------------
> source("regart.R")

------------------------------------
Test on diff (all scores)
Mean Difference              t              df         p-value
```

```
      0.0669827       0.7057641    9999.0000000       0.4803513

Test on Remedial
Mean Difference               t              df         p-value
   8.434609e+00    9.036847e+00    9.900000e+01    1.376677e-14

Test on Gifted
Mean Difference               t              df         p-value
      -8.371483      -10.215295       99.000000        0.000000

------------------------------------
```

The preceding simulation was unusual in that the phenomenon it illustrates happens virtually every time. In the next example, we need to use the Law of Large Numbers.

6.4.2 An Example of Power Analysis by Simulation

Suppose we want to test the effect of some experimental treatment on mean response, comparing an experimental group to a control. We are willing to assume normality, but *not* equal variances. We're ready to use an unequal-variances t-test, and we want to do a power analysis.

Unfortunately it's safe to say that nobody knows the exact non-central distribution of this monster. In fact, even the central distribution isn't exact; it's just a very good approximation. So, we have to resort to first principles. There are four parameters: $\theta = (\mu_1, \mu_2, \sigma_1^2, \sigma_2^2)$. For a given set of parameter values, we will simulate samples of size n_1 and n_2 from normal distributions, do the significance test, and see if it's significant. We'll do it over and over. By the Law of Large Numbers, the proportion of times the test is significant will approach the power as the Monte Carlo sample size (the number of data sets we simulate) increases.

The number we get, of course, will just be an *estimate* of the power. How accurate is the estimate? As promised earlier, we'll accompany every Monte Carlo estimate of a probability with a confidence interval. Here's the formula. For the record, it's based on the normal approximation to the binomial, not bothering with a continuity correction.

$$\widehat{P} \pm z_{1-\frac{\alpha}{2}} \sqrt{\frac{\widehat{P}(1-\widehat{P})}{m}} \tag{6.1}$$

This formula will be implemented in the S function `merror` for "margin of error."

```
merror <- function(phat,m,alpha) # (1-alpha)*100% merror for a proportion
    {
    z <- qnorm(1-alpha/2)
    merror <- z * sqrt(phat*(1-phat)/m)  # m is (Monte Carlo) sample size
    merror
    }
```

The Monte Carlo estimate of the probability is denoted by $\widehat{P}$, the quantity m is the Monte Carlo sample size, and $z_{1-\alpha/2}$ is the value with area $1 - \frac{\alpha}{2}$ to the left of it, under the standard normal curve. Typically, we will choose $\alpha = 0.01$ to get a 99% confidence interval, so $z_{1-\alpha/2} = 2.575829$.

How should we choose m? In other words, how many data sets should we simulate? It depends on how much accuracy we want.Since our policy is to accompany Monte Carlo estimates with confidence intervals, we will choose the Monte Carlo sample size to control the width of the confidence interval.

According to Equation (6.1), the confidence interval is an estimated probability, plus or minus a margin of error. The margin of error is $z_{1-\frac{\alpha}{2}}\sqrt{\frac{\widehat{P}(1-\widehat{P})}{m}}$, which may be viewed as an *estimate* of $z_{1-\frac{\alpha}{2}}\sqrt{\frac{P(1-P)}{m}}$. So, for any given probability we are trying to estimate, we can set the desired margin of error to some small value, and solve for m. Denoting the *criterion* margin of error by c, the general solution is

$$m = \frac{z^2_{1-\frac{\alpha}{2}}}{c^2} P(1-P), \tag{6.2}$$

which is implemented in the S function `mmargin`.

```
mmargin <- function(p,cc,alpha)
           # Choose m to get (1-alpha)*100% margin of error equal to cc
           {
           mmargin <- p*(1-p)*qnorm(1-alpha/2)^2/cc^2
           mmargin <- trunc(mmargin+1) # Round up to next integer
           mmargin
           } # End definition of function mmargin
```

Suppose we want a 99% confidence interval around a power of 0.80 to be accurate to plus or minus 0.01.

```
> mmargin(.8,.01,.01)
[1] 10616
```

The table below shows Monte Carlo sample sizes for estimating power with a 99% confidence interval.

Table 6.1: Monte Carlo Sample Size Required to Estimate Power with a Specified 99% Margin of Error

	Power Being Estimated					
Margin of Error	0.70	0.75	0.80	0.85	0.90	0.99
0.10	140	125	107	85	60	7
0.05	558	498	425	339	239	27
0.01	13,934	12,441	10,616	8,460	5,972	657
0.005	55,734	49,762	42,464	33,838	23,886	2,628
0.001	1,393,329	1,244,044	1,061,584	845,950	59,7141	65,686

It's somewhat informative to see how the rows of the table were obtained.

```
> wpow <- c(.7,.75,.8,.85,.9,.99)
> mmargin(wpow,.1,.01)
[1] 140 125 107  85  60   7
> mmargin(wpow,.05,.01)
[1] 558 498 425 339 239  27
> mmargin(wpow,.01,.01)
[1] 13934 12441 10616  8460  5972   657
> mmargin(wpow,.005,.01)
[1] 55734 49762 42464 33838 23886  2628
> mmargin(wpow,.001,.01)
[1] 1393329 1244044 1061584  845950  597141   65686
```

Equations (6.1) and (6.2) are general; they apply to the Monte Carlo estimation of *any* probability, and Table 6.1 applies to any Monte Carlo estimation of power. Let's return to the specific example at hand. Suppose we the population standard deviation of the Control Group is 2 and the standard deviation of the Experimental Group is 6. We'll let the population means be $\mu_1 = 1$ and $\mu_2 = 3$, so that the difference between population means is half the *average* within-group population standard deviation.

To select a good starting value of n, let's pretend that the standard deviations are equal to the average value, and we are planning an ordinary two-sample t-test. Referring to formula (4.4) for the non-centrality parameter of the non-central F-distribution, we'll let $q = \frac{1}{2}$; this is optimal when the variances are equal. Since $\delta = \frac{1}{2}$, we have $\phi = n\,q(1-q)\,\delta^2 = \frac{n}{16}$. Here's some S. It's short — and sweet. Well, maybe it's an acquired taste. It's also true that I know this problem pretty well, so I knew a good range of n values to try.

```
> n <- 125:135
> pow <- 1-pf(qf(.95,1,(n-2)),1,(n-2),(n/16))
> cbind(n,pow)
        n       pow
 [1,] 125 0.7919594
 [2,] 126 0.7951683
 [3,] 127 0.7983349
 [4,] 128 0.8014596
 [5,] 129 0.8045426
 [6,] 130 0.8075844
 [7,] 131 0.8105855
 [8,] 132 0.8135460
 [9,] 133 0.8164666
[10,] 134 0.8193475
[11,] 135 0.8221892
```

We will start the unequal variance search at $n = 128$. And, though we are interested in more accuracy, it makes sense to start with a target margin of error of 0.05. The idea is to start out with rough estimation, and get more accurate only once we think we are close to the right n.

```
> n1 <- 64 ; mu1 <- 1 ; sd1 <- 2 # Control Group
> n2 <- 64 ; mu2 <- 3 ; sd2 <- 6 # Experimental Group
>
> con <- rnorm(n1,mu1,sd1) ; exp <- rnorm(n2,mu2,sd2)

> help(t.test)
```

The output of help is omitted, but we learn that the default is a test assuming unequal variances – just what we want.

```
> t.test(con,exp)

Welch Two Sample t-test

data:  con and exp
t = -2.4462, df = 78.609, p-value = 0.01667
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -3.4632952 -0.3556207
sample estimates:
mean of x mean of y
 1.117435  3.026893

> t.test(con,exp)[1]
$statistic
        t
-2.446186

> t.test(con,exp)[3]
$p.value
[1] 0.01667109
>
> m <- 500 # Monte Carlo sample size (Number of simulations)
> numsig <- 0 # Initializing
> for(i in 1:m)
+     {
+     con <- rnorm(n1,mu1,sd1) ; exp <- rnorm(n2,mu2,sd2)
+     numsig <- numsig+(t.test(con,exp)[3]<.05)
+     }
> pow <- numsig/m
> cat ("Monte Carlo Power = ",pow,"\n") ; cat ("\n")
Monte Carlo Power =  0.708

> m <- 500 # Monte Carlo sample size (Number of simulations)
> numsig <- 0 # Initializing
> for(i in 1:m)
+     {
+     con <- rnorm(n1,mu1,sd1) ; exp <- rnorm(n2,mu2,sd2)
+     numsig <- numsig+(t.test(con,exp)[3]<.05)
+     }
> pow <- numsig/m
> cat ("Monte Carlo Power = ",pow,"\n") ; cat ("\n")
Monte Carlo Power =  0.698
```

Try it again.

```
>
> m <- 500 # Monte Carlo sample size (Number of simulations)
> numsig <- 0 # Initializing
> for(i in 1:m)
+     {
+     con <- rnorm(n1,mu1,sd1) ; exp <- rnorm(n2,mu2,sd2)
+     numsig <- numsig+(t.test(con,exp)[3]<.05)
+     }
> pow <- numsig/m
> cat ("Monte Carlo Power = ",pow,"\n") ; cat ("\n")
Monte Carlo Power =  0.702
```

Try a larger sample size.

```
> n1 <- 80 ; mu1 <- 1 ; sd1 <- 2 # Control Group
> n2 <- 80 ; mu2 <- 3 ; sd2 <- 6 # Experimental Group
> m <- 500 # Monte Carlo sample size (Number of simulations)
> numsig <- 0 # Initializing
> for(i in 1:m)
+     {
+     con <- rnorm(n1,mu1,sd1) ; exp <- rnorm(n2,mu2,sd2)
+     numsig <- numsig+(t.test(con,exp)[3]<.05)
+     }
> pow <- numsig/m
> cat ("Monte Carlo Power = ",pow,"\n") ; cat ("\n")
Monte Carlo Power =  0.812
```

Try it again.

```
> n1 <- 80 ; mu1 <- 1 ; sd1 <- 2 # Control Group
> n2 <- 80 ; mu2 <- 3 ; sd2 <- 6 # Experimental Group
> m <- 500 # Monte Carlo sample size (Number of simulations)
> numsig <- 0 # Initializing
> for(i in 1:m)
+     {
+     con <- rnorm(n1,mu1,sd1) ; exp <- rnorm(n2,mu2,sd2)
+     numsig <- numsig+(t.test(con,exp)[3]<.05)
+     }
> pow <- numsig/m
> cat ("Monte Carlo Power = ",pow,"\n") ; cat ("\n")
Monte Carlo Power =  0.792
```

It seems that was a remarkably lucky guess. Now seek margin of error around 0.01.

```
>
> m <- 10000 # Monte Carlo sample size (Number of simulations)
> numsig <- 0 # Initializing
> for(i in 1:m)
+     {
+     con <- rnorm(n1,mu1,sd1) ; exp <- rnorm(n2,mu2,sd2)
+     numsig <- numsig+(t.test(con,exp)[3]<.05)
+     }
> pow <- numsig/m
> cat ("Monte Carlo Power = ",pow,"\n") ; cat ("\n")
Monte Carlo Power =  0.8001

> merror <- function(phat,m,alpha) # (1-alpha)*100% merror for a proportion
+     {
+     z <- qnorm(1-alpha/2)
+     merror <- z * sqrt(phat*(1-phat)/m)  # m is (Monte Carlo) sample size
+     merror
+     }
> margin <- merror(.8001,10000,.01) ; margin
[1] 0.01030138
> cat("99% CI from ",(pow-margin)," to ",(pow+margin),"\n")
99% CI from  0.7897986  to  0.810
```

This is very nice, except that I can't believe equal sample sizes are optimal when the variances are unequal. Let's try sample sizes proportional to the standard deviations, so $n_1 = 40$ and $n_2 = 120$. The idea is that perhaps the two population means should be estimated with roughly the same precision, and we need a bigger sample size in the experimental condition to compensate for the larger variance. Well, actually I chose the relative sample sizes to minimize the standard deviation of the sampling distribution of the difference between means — the quantity that is estimated by the denominator of the t statistic.

```
> n1 <- 40 ; mu1 <- 1 ; sd1 <- 2 # Control Group
> n2 <- 120 ; mu2 <- 3 ; sd2 <- 6 # Experimental Group
> m <- 500 # Monte Carlo sample size (Number of simulations)
> numsig <- 0 # Initializing
> for(i in 1:m)
+     {
+     con <- rnorm(n1,mu1,sd1) ; exp <- rnorm(n2,mu2,sd2)
+     numsig <- numsig+(t.test(con,exp)[3]<.05)
+     }
> pow <- numsig/m
> cat ("Monte Carlo Power = ",pow,"\n") ; cat ("\n")
Monte Carlo Power =  0.89

> margin <- merror(pow,m,.01)
> cat("99% CI from ",(pow-margin)," to ",(pow+margin),"\n")
99% CI from  0.8539568  to  0.9260432
>
> # This is promising. Get some precision.
>
> n1 <- 40 ; mu1 <- 1 ; sd1 <- 2 # Control Group
> n2 <- 120 ; mu2 <- 3 ; sd2 <- 6 # Experimental Group
> m <- 10000 # Monte Carlo sample size (Number of simulations)
> numsig <- 0 # Initializing
> for(i in 1:m)
+     {
+     con <- rnorm(n1,mu1,sd1) ; exp <- rnorm(n2,mu2,sd2)
+     numsig <- numsig+(t.test(con,exp)[3]<.05)
+     }
> pow <- numsig/m
> cat ("Monte Carlo Power = ",pow,"\n") ; cat ("\n")
```

```
Monte Carlo Power =  0.8803

> margin <- merror(pow,m,.01)
> cat("99% CI from ",(pow-margin)," to ",(pow+margin),"\n")
99% CI from  0.8719386  to  0.8886614
```

So again we see that power depends on *design* as well as on effect size and sample size. It will be left as an exercise to find out how much sample size we could save (over the $n_1 = n_2 = 80$ solution) by taking this into account in the present case.

Finally, it should not be surprising that R has a t-test function (type `help(t.test)` for details), and the custom function `ttest` was unnecessary. To see the variety of specialized statistical functions that are available, type `library(help=stats)`.