Notes on the use of R for psychology experiments and questionnaires

Jonathan Baron
Department of Psychology, University of Pennsylvania

Yuelin Li
Center for Outcomes Research, Children's Hospital of Philadelphia*

August 20, 2003

Contents

We thank Andrew Hochman, Rashid Nassar, Christophe Pallier, and Hans-Rudolf Roth for helpful comments.

1 Introduction

This is a set of notes and annotated examples of the use of the statistical package R. It is "for psychology experiments and questionnaires" because we cover the main statistical methods used by psychologists who do research on human subjects, but of course it this is also relevant to researchers in others fields that do similar kinds of research.

R, like S–Plus, is based on the S language invented at Bell Labs. Most of this should also work with S–Plus. Because R is open-source (hence also free), it has benefitted from the work of many contributors and bug finders. R is a complete package. You can do with it whatever you can do with Systat, SPSS, Stata, or SAS, including graphics. Contributed packages are added or updated almost weekly; in some cases these are at the cutting edge of statistical practice.

Some things are more difficult with R, especially if you are used to using menus. With R, it helps to have a list of commands in front of you. There are lists in the on-line help and in the index of *An introduction to R* by the R Core Development Team, and in the reference cards listed in `http://finzi.psych.upenn.edu/`.

Some things turn out to be easier in R. Although there are no menus, the on-line help files are very easy to use, and quite complete. The elegance of the language helps too, particularly those tasks involving the manipulation of data.

The purpose of this document is to reduce the difficulty of the things that are more difficult at first. We assume that you have read the relevant parts of *An introduction to R*, but we do not assume that you have mastered its contents. We assume that you have gotten to the point of installing R and trying a couple of examples.

2 A few useful concepts and commands

2.1 Concepts

In R, most commands are functions. That is, the command is written as the name of the function, followed by parentheses, with the arguments of the function in parentheses, separated by commas when there is more than one, e.g., `plot(mydata1)`. When there is no argument, the parentheses are still needed, e.g., `q()` to exit the program.

In this document, we use names such as `x1` or `file1`, that is, names containing both letters and a digit, to indicate variable names that the user makes up. Really these can be of any form. We use the number simply to clarify the distinction between a made up name and a key word with a pre-determined meaning in R. R is case sensitive.

Although most commands are functions with the arguments in parentheses, some arguments require specification of a key word with an equal sign and a value for that key word, such as `source("myfile1.R",echo=T)`, which means read in `myfile1.R` and echo the commands on the screen. Key words can be abbreviated (e.g., `e=T`).

In addition to the idea of a function, R has *objects* and *modes*. Objects are anything that you can give a name. There are many different *classes* of objects. The main classes of interest here are *vector*, *matrix*, *factor*, *list*, and *data frame*. The mode of an object tells what kind of things are in it. The main modes of interest here are `logical, numeric,` and `character`.

We sometimes indicate the class of object (vector, matrix, factor, etc.) by using `v1` for a vector, `m1` for a matrix, and so on. Most R functions, however, will either accept more than one type of object or will "coerce" a type into the form that it needs.

The most interesting object is a data frame. It is useful to think about data frames in terms of rows and columns. The rows are subjects or observations. The columns are variables, but a matrix can be a column too. The variables can be of different classes.

The behavior of any given function, such as `plot(), aov()` (analysis of variance) or `summary()` depends on the object class and mode to which it is applied. A nice thing about R is that you almost don't need to know this, because the default behavior of functions is usually what you want. One way to use R is just to ignore completely the distinction among classes and modes, but *check* every step (by typing the name of the object it creates or modifies). If you proceed this way, you will also get error messages, which you must learn to interpret. Most of the time, again, you can find the problem by looking at the objects involved, one by one, typing the name of each object.

Sometimes, however, you must know the distinctions. For example, a factor is treated differently from an ordinary vector in an analysis of variance or regression. A factor is what is often called a categorical variable. Even if numbers are used to represent categories, they are not treated as ordered. If you use a vector and think you are using a factor, you can be misled.

2.2 Commands

As a reminder, here is a list of some of the useful commands that you should be familiar with, and some more advanced ones that are worth knowing about. We discuss graphics in a later section.

2.2.1 Getting help

`help.start()` starts the browser version of the help files. (But you can use `help()` without it.) With a fast computer and a good browser, it is often simpler to open the html documents in a browser while you work and just use the browser's capabilities.

`help(command1)` prints the help available about `command1`. `help.search("keyword1")` searches keywords for help on this topic.

`apropos(topic1)` or `apropos("topic1")` finds commands relevant to topic1, whatever it is.

`example(command1)` prints an example of the use of the command. This is especially useful for graphics commands. Try, for example, `example(contour)`, `example(dotchart)`, `example(image)`, and `example(persp)`.

2.2.2 Installing packages

`install.packages(c("package1","package2"))` will install these two packages from CRAN (the main archive), if your computer is connected to the Internet. You don't need the `c()` if you just want one package. You should, at some point, make sure that you are using the CRAN mirror page that is closest to you. For example, if you live in the U.S., you should have a `.Rprofile` file with `options(CRAN = "http://cran.us.r-project.org")` in it. (It may work slightly differently on Windows.)

`CRAN.packages(), installed.packages(), and update.packages()` are also useful. The first tells you what is available. The second tells you what is installed. The third updates the packages that you have installed, to their latest version.

To install packages from the Bioconductor set, see the instructions in
`http://www.bioconductor.org/reposToolsDesc.html`.

When packages are not on CRAN, you can download them and use `R CMD INSTALL package1.tar.gz` from a Unix/Linux command line. (Again, this may be different on Windows.)

2.2.3 Assignment, logic, and arithmetic

`<-` assigns what is on the right of the arrow to what is on the left. (If you use ESS, the _ key will produce this arrow with spaces, a great convenience.)

Typing the name of the object prints the object. For example, if you say:

```
t1 <- c(1,2,3,4,5)
t1
```

you will see `1 2 3 4 5`.

Logical objects can be true or false. Some functions and operators return TRUE or FALSE. For example, `1==1`, is TRUE because 1 does equal 1. Likewise, `1==2` is FALSE, and `1<2` is TRUE. Use `all()`, `any()`, `|`, `||`, `&`, and `&&` to combine logical expressions, and use `!` to negate them. The difference between the `|` and `||` form is that the shorter form, when applied to vectors, etc., returns a vector, while the longer form stops when the result is determined and returns a single TRUE or FALSE.

Set functions operate on the elements of vectors: `union(v1,v2)`, `intersect(v1,v2)`, `setdiff(v1,v2)`, `setequal(v1,v2)`, `is.element(element1,v1)` (or, `element1 %in% v1`).

Arithmetic works. For example, `-t1` yields `-1 -2 -3 -4 -5`. It works on matrices and data frames too. For example, suppose `m1` is the matrix

```
1 2 3
4 5 6
```

Then `m1 * 2` is

```
2  4  6
8 10 12
```

Matrix multiplication works too. Suppose `m2` is the matrix

```
1 2
1 2
1 2
```

then `m1 %*% m2` is

```
 6 12
15 30
```

and `m2 %*% m1` is

```
 9   12   15
 9   12   15
 9   12   15
```

You can also multiply a matrix by a vector using matrix multiplication, vectors are aligned vertically when they come after the `%*%` sign and horizontally when they come before it. This is a good way to find weighted sums, as we shall explain.

For ordinary multiplication of a matrix times a vector, the vector is vertical and is repeated as many times as needed. For example `m2 * 1:2` yields

```
1 4
2 2
1 4
```

Ordinarily, you would multiply a matrix by a vector when the length of the vector is equal to the number of rows in the matrix.

2.2.4 Vectors, matrices, lists, arrays, and data frames

`:` is a way to abbreviate a sequence of numbers, e.g., `1:5` is equivalent to 1,2,3,4,5.

`c(number.list1)` makes the list of numbers (separated by commas) into a vector object. For example, `c(1,2,3,4,5)` (but `1:5` is already a vector, so you do not need to say `c(1:5)`).

`rep(v1,n1)` repeats the vector `v1 n1` times. For example, `rep(c(1:5),2)` is `1,2,3,4,5,1,2,3,4,5`.

`rep(v1,v2)` repeats each element of the vector `v1` a number of times indicated by the corresponding element of the vector `v2`. The vectors `v1` and `v2` must have the same length. For example, `rep(c(1,2,3),c(2,2,2))` is `1,1,2,2,3,3`. Notice that this can also be written as `rep(c(1,2,3),rep(2,3))`. (See also the function `gl()` for generating factors according to a pattern.)

`cbind(v1,v2,v3)` puts vectors `v1, v2,` and `v3` (all of the same length) together as columns of a matrix. You can of course give this a name, such as `mat1 <- cbind(v1,v2,v2)`.

`matrix(v1,rows1,colmuns1)` makes the vector `v1` into a matrix with the given number of rows and columns. You don't need to specify both rows and columns, but you do need to put in both commas. You can also use key words instead of using position to indicate which argument is which, and then you do not need the commas. For example, `matrix(1:10, ncol=5)` represents the matrix

```
1  3  5  7  9
2  4  6  8 10
```

Notice that the matrix is filled column by column.

`data.frame(vector.list1)` takes a list of vectors, all of the same length (error message if they aren't) and makes them into a data frame. It can also include factors as well as vectors.

`dim(obj1)` prints the dimensions of a matrix, array or data frame.

`length(vector1)` prints the length of `vector1`.

You can refer to parts of objects. `m1[,3]` is the third column of matrix `m1`. `m1[,-3]` is all the columns except the third. `m1[m1[,1]>3,]` is all the rows for which the first column is greater than 3. `v1[2]` is the second element of vector `v1`. If `df1` is a data frame with columns `a, b`, and `c`, you can refer to the third column as `df1$c`.

Most functions return lists. You can see the elements of a list with `unlist()`. For example, try `unlist(t.test(1:5))` to see what the `t.test()` function returns. This is also listed in the section of help pages called "Value."

`array()` seems very complicated at first, but it is extremely useful when you have a three-way classification, e.g., subjects, cases, and questions, with each question asked about each case. We give an example later.

`outer(m1,m2,"fun1")` applies `fun1`, a function of two variables, to each combination of `m1` and `m2`. The default is to multiply them.

`mapply("fun1",o1,o2)`, another very powerful function, applies `fun1` to the elements of `o1` and `o2`. For example, if these are data frames, and `fun1` is `"t.test"`, you will get a list of t tests comparing the first column of `o1` with the first column of `o2`, the second with the second, and so on. This is because the basic elements of a data frame are the columns.

2.2.5 String functions

R is not intended as a language for manipulating text, but it is surprisingly powerful. If you know R, you might not need to learn Perl. Strings are character variables that consist of letters, numbers, and symbols.

`strsplit()` splits a string, and `paste()` puts a string together out of components.

`grep(), sub(), gsub()`, and `regexpr()` allow you to search for, and replace, parts of strings.

The set functions such as `union(), intersect(), setdiff()`, and `%in%` are also useful for dealing with databases that consist of strings such as names and email addresses.

You can even use these functions to write new R commands as strings, so that R can program itself! Just to see an example of how this works, try `eval(parse(text="t.test(1:5)"))`. The `parse()` function turns the text into an expression, and `eval()` evaluates the expression. So this is equivalent to `t.test(1:5)`. But you could replace `t.test(1:5)` with any string constructed by R itself.

2.2.6 Loading and saving

`library(xx1)` loads the extra library. A useful library for psychology is and `mva` (multivariate analysis). To find the contents of a library such as `mva` before you load it, say `library(help=mva)`. The `ctest` library is already loaded when you start R.

`source("file1")` runs the commands in `file1`.

`sink("file1")` diverts output to `file1` until you say `sink()`.

`save(x1,file="file1")` saves object `x1` to file `file1`. To read in the file, use `load("file1")`.

`q()` quits the program. `q("yes")` saves everything.

`write(object, "file1")` writes a matrix or some other object to `file1`.

`write.table(object1,"file1")` writes a table and has an option to make it comma delimited, so that (for example) Excel can read it. See the help file, but to make it comma delimited, say `write.table(object1,"file1",sep=",")`.

`round()` produces output rounded off, which is useful when you are cutting and pasting R output into a manuscript. For example, `round(t.test(v1)$statistic,2)` rounds off the value of `t` to two places. Other useful functions are `format` and `formatC`. For example, if we assign `t1 <- t.test(v1`, then the following command prints out a nicely formatted result, suitable for dumping into a paper:

```
print(paste("(t_{",t1[[2]],"}=",formatC(t1[[1]],format="f",digits=2),
            ", p=",formatC(t1[[3]],format="f"),")",sep=""),quote=FALSE)
```

This works because `t1` is actually a list, and the numbers in the double brackets refer to the elements of the list.

`read.table("file1")` reads in data from a file. The first line of the file can (but need not) contain the names of the variables in each column.

2.2.7 Dealing with objects

`ls()` lists all the active objects.

`rm(object1)` removes object1. To remove all objects, say `rm(list=ls())`.

`attach(data.frame1)` makes the variables in `data.frame1` active and available generally.

`names(obj1)` prints the names, e.g., of a matrix or data frame.

`typeof()`, `mode())`, and `class()` tell you about the properties of an object.

2.2.8 Summaries and calculations by row, column, or group

`summary(x1)` prints statistics for the variables (columns) in `x1`, which may be a vector, matrix, or data frame. See also the `str()` function, which is similar, and `aggregate()`, which summarizes by groups.

`table(x1)` prints a table of the number of times each value occurs in `x1`. `table(x1,y1)` prints a cross-tabulation of the two variables. The `table` function can do a lot more. Use `prop.table()` when you want proportions rather than counts.

`ave(v1,v2)` yields averages of vector `v1` grouped by the factor `v2`.

`cumsum(v1)` is the cumulative sum of vector `v1`.

You can do calculations on rows or columns of a matrix and get the result as a vector. `apply(x1,2,mean)` yields just the means of the columns. Use `apply(x1,1,mean)` for the rows. You can use other functions aside from `mean`, such as `sd`, `max`, `min` or `sum`. To ignore missing data, use `apply(x1,2,mean,na.rm=T)`, etc. For sums and means, it is easier to use `rowSums(), colSums(), rowMeans()`, and `colMeans` instead of `apply()`. Note that you can use apply with a function, e.g., `apply(x1,1,function(x) exp(sum(log(x)))` (which is a roundabout way to write `apply(x1,1,prod)`). The same thing can be written in two steps, e.g.:

```
newprod <- function(x) {exp(sum(log(x)))
apply(x1,1,newprod)
```

You can refer to a subset of an object in many other ways. One way is to use a square bracket at the end, e.g., `matrix1[,1:5]` refers to columns 1 through 5 of the matrix. You can also use this method for new objects, e.g., `(matrix1+matrix2)[,1:5]`, which refers to the first five columns of the sum of the two matrices. Another important method is the use of `by()` or `aggregate()` to compute statistics for subgroups defined by vectors or factors. You can also use `split()` to get a list of subgroups. Finally, many functions allow you to use a `subset` argument.

2.2.9 Functions and debugging

`function()` allows you to write your own functions.

Several functions are useful for debugging your own functions or scripts: `traceback(), debug(), browser(), recover()`.

3 Basic method

The following basic method is assumed here. You have a command file and then submit it, for each data set. Thus, for each experiment or study, you have two files. One consists of the data. Call it `exp1.data`. The other is a list of commands to be executed by R, `exp1.R`. (Any suffixes will do, although ESS recognizes the R suffix and loads special features for editing.) The advantage of this approach is that you have a complete record of what your transformed variables mean. If your data set is small relative to the speed of your computer, it is a good idea to revise exp1.R and re-run it each time you make a change that you want to keep. So you could have exp1.R open in the window of an editor while you have R in another window.[1]

[1]If you want, you can put your data in the same fi le as the commands. The simplest way to do this is to put commas between the numbers and then use a command like `t1 <- c(1,2,3, ...26,90)`, possibly over several lines, where the numbers are your data.

To analyze a data set, you start R in the directory where the data and command file are. Then, at the R prompt, you type

```
source("exp1.R")
```

and the command file runs. The first line of the command file usually reads in the data. You may include statistics and graphics commands in the source file. You will not see the output if you say `source("data1.R")`, although they will still run. If you want to see the output, say

```
source("data1.R",echo=T)
```

Command files can and should be annotated. R ignores everything after a #. In this document, the examples are not meant to be run.

We have mentioned ESS, which stands for "Emacs Speaks Statistics." This is an add-on for the Emacs editor, making Emacs more useful with several different statistical programs, including R, S–Plus, and SAS. [2] If you use ESS, then you will want to run R as a process in Emacs, so, to start R, say `emacs -f R`. You will want `exp1.R` in another window, so also say `emacs exp1.R`. With ESS, you can easily cut and paste blocks (or lines) of commands from one window to another.

Here are some tips for debugging:

- If you use the `source("exp1.R")` method described here, use `source("exp1.R",echo=T)` to echo the input and see how far the commands get before they "bomb."
- Use `ls()` to see which objects have been created.
- Often the problem is with a particular function, often because it has been applied to the wrong type or size of object. Check the sizes of objects with `dim()` or (for vectors) `length()`.
- Look at the `help()` for the function in question. (If you use `help.start()` at the beginning, the output will appear in your browser. The main advantage of this is that you can follow links to related functions very easily.)
- Type the names of the objects to make sure they are what you think they are.
- If the help is not helpful enough, make up a little example and try it. For example, you can get a matrix by saying `m1 <- matrix(1:12,,3)`.
- For debugging functions, try `debug()`, `browser()`, and `traceback()`. (See their help pages. We do very little with functions here.)

4 Reading and transforming data

4.1 Data layout

R, like Splus and S, represents an entire conceptual system for thinking about data. You may need to learn some new ways of thinking. One way that is new for users of Systat in particular (but perhaps more familiar to users of SAS) concerns two different ways of laying out a data set. In the Systat way, each subject is a row (which may be continued

[2] ESS is wonderful, but Emacs will cause trouble for you if you use a word processor like Word, and if you are used to shortcut keys such as ctrl-x for cutting text. The shortcut keys in Emacs are all different, and this leads to serious mind-boggle. One solution, adopted by the first author, is to give up word processors and editors that use the same shortcuts, such as Winedt. A side effect of this solution is that you have very little reason to use Microsoft Windows.

on the next row if too long, but still conceptually a row) and each variable is a column. You can do this in R too, and most of the time it is sufficient.

But some the features of R will not work with this kind of representation, in particular, repeated-measures analysis of variance. So you need a second way of representing data, which is that each row represents a single datum, e.g., one subject's answer to one question. The row also contains an identifier for all the relevant classifications, such as the question number, the subscale that the question is part of, AND the subject. Thus, "subject" becomes a category with no special status, technically a factor (and remember to make sure it is a factor, lest you find yourself studying the effect of the subject's number).

4.2 A simple questionnaire example

Let us start with an example of the old-fashioned way. In the file `ctest3.data`, each subject is a row, and there are 134 columns. The first four are age, sex, student status, and time to complete the study. The rest are the responses to four questions about each of 32 cases. Each group of four is preceded by the trial order, but this is ignored for now.

```
c0 <- read.table("ctest3.data")
```

The data file has no labels, so we can read it with `read.table`.

```
age1 <- c0[,1]
sex1 <- c0[,2]
student1 <- c0[,3]
time1 <- c0[,4]
nsub1 <- nrow(c0)
```

We can refer to elements of `c0` by `c0[row,column]`. For example, `c0[1,2]` is the sex of the first subject. We can leave one part blank and get all of it, e.g., `c0[,2]` is a vector (column of numbers) representing the sex of all the subjects. The last line defines `nsub1` as the number of subjects.

```
c1 <- as.matrix(c0[,4+1:128])
```

Now `c1` is the main part of the data, the matrix of responses. The expression 1:128 is a vector, which expands to 1 2 3 . . . 128. By adding 4, it becomes 5 6 7 . . . 132.

4.2.1 Extracting subsets of data

```
rsp1 <- c1[,4*c(1:32)-2]
rsp2 <- c1[,4*c(1:32)-1]
```

The above two lines illustrate the extraction of sub-matrices representing answers to two of the four questions making up each item. The matrix `rsp1` has 32 columns, corresponding to columns 2 6 10 ... 126 of the original 128-column matrix `c1`. The matrix `rsp2` corresponds to 3 7 11 ... 127.

Another way to do this is to use an array. We could say `a1 <- array(c1,c(ns,4,32))`. Then `a1[,1,]` is the equivalent of `rsp1`, and `a1[20,1,]` is `rsp1` for subject 20. To see how arrays print out, try the following:

```
m1 <- matrix(1:60,5,)
a1 <- array(m1,c(5,2,6))
m1
a1
```

You will see that the rows of each table are the first index and the columns are the second index. Arrays seem difficult at first, but they are very useful for this sort of analysis.

4.2.2 Finding means (or other things) of sets of variables

```
r1mean <- apply(rsp1,1,mean)
r2mean <- apply(rsp2,1,mean)
```

The above lines illustrate the use of `apply` for getting means of subscales. In particular, `abrmean` is the mean of the subscale consisting of the answers to the second question in each group. The `apply` function works on the data in its first argument, then applies the function in its third argument, which, in this case, is `mean`. (It can be `max` or `min` or any defined function.) The second argument is 1 for rows, 2 for columns (and so on, for arrays). We want the function applied to rows.

```
r4mean <- apply(c1[,4*c(1:32)],1,mean)
```

The expression here represents the matrix for the last item in each group of four. The first argument can be any matrix or data frame. (The output for a data frame will be labeled with row or column names.) For example, suppose you have a list of variables such as `q1, q2, q3`, etc. Each is a vector, whose length is the number of subjects. The average of the first three variables for each subject is, `apply(cbind(q1,q2,q3),1,mean)`. (This is the equivalent of the Systat expression `avg(q1,q2,q3)`. A little more verbose, to be sure, but much more flexible.)

You can use apply is to tabulate the values of each column of a matrix `m1`: `apply(m1,2,table)`. Or, to find column means, `apply(m1,2,mean)`.

There are many other ways to make tables. Some of the relevant functions are `table`, `tapply`, `sapply`, `ave`, and `by`. Here is an illustration of the use of `by`. Suppose you have a matrix `m1` like this:

```
1 2 3 4
4 4 5 5
5 6 4 5
```

The columns represent the combination of two variables, `y1` is `0 0 1 1`, for the four columns, respectively, and `y2` is `0 1 0 1`. To get the means of the columns for the two values of `y1`, say `by(t(m1), y1, mean)`. You get 3.67 and 4.33 (labeled appropriately by values of y1). You need to use `t(m1)` because `by` works by rows. If you say `by(t(m1), data.frame(y1,y2), mean)`, you get a cross tabulation of the means by both factors. (This is, of course, the means of the four columns of the original matrix.)

Of course, you can also use `by` to classify rows; in the usual examples, this would be groups of subjects rather than classifications of variables.

4.2.3 One row per observation

The next subsection shows how to transform the data from a layout from "one row per subject" to "one row per observation." We're going to use the matrix `rsp1`, which has 32 columns and one row per subject. Here are five subjects:

```
1 1 2 2 1 2 3 5 2 3 2 4 2 5 7 7 6 6 7 5 7 8 7 9 8 8 9 9 8 9 9 9
1 2 3 2 1 3 2 3 2 3 2 3 2 3 2 4 1 2 4 5 4 5 5 6 5 6 6 7 6 7 7 8
1 1 2 3 1 2 3 4 2 3 3 4 2 4 3 4 4 4 5 5 5 6 6 7 6 7 7 8 7 7 8 8
1 2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5 6 6 6 6 7 7 7 7 8 8 8 8 9 9 9
1 1 1 2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5 6 6 6 6 7 7 7 7 8 8 8 8 9
```

We'll create a matrix with one row per observation. The first column will contain the observations, one variable at a time, and the remaining columns will contain numbers representing the subject and the level of the observation on each variable of interest. There are two such variables here, `r2` and `r1`. The variable `r2` has four levels, `1 2 3 4`, and it cycles through the 32 columns as `1 2 3 4 1 2 3 4 ...` The variable `r1` has the values (for successive columns) `1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4`. These levels are ordered. They are not just arbitrary labels. (For that, we would need the `factor` function.)

```
r2 <- rep(1:4,8)
r1 <- rep(rep(1:4,rep(4,4)),2)
```

The above two lines create vectors representing the levels of each variable for each subject. The `rep` command for r2 says to repeat the sequence `1 2 3 4`, 8 times. The `rep` command for `r1` says take the sequence `1 2 3 4`, then repeat the first element 4 times, the second element 4 times, etc. It does this by using a vector as its second argument. That vector is `rep(4,4)`, which means repeat the number 4, 4 times. So `rep(4,4)` is equivalent to `c(4 4 4 4)`. The last argument, 2, in the command for `r1` means that the whole sequence is repeated twice. Notice that `r1` and `r2` are the codes for one row of the matrix `rsp1`.

```
nsub1 <- nrow(rsp1)
subj1 <- as.factor(rep(1:nsub1,32))
```

`nsub1` is just the number of subjects (5 in the example), the number of rows in the matrix `rsp1`. The vector `subj1` is what we will need to assign a subject number to each observation. It consists of the sequence `1 2 3 4 5`, repeated 32 times. It corresponds to the columns of `rsp1`.

```
abr1 <- data.frame(ab1=as.vector(rsp1),sub1=subj1,
 dcost1=rep(r1,rep(nsub1,32)),abcost1=rep(r2,rep(nsub1,32)))
```

The `data.frame` function puts together several vectors into a data.frame, which has rows and columns like a matrix.[3] Each vector becomes a column. The `as.vector` function reads down by columns, that is, the first column, then the second, and so on. So `ab` is now a vector in which the first `nsub1` elements are the same as the first column of `rsp1`, that is, `1 1 1 1 1`. The first 15 elements of ab are: `1 1 1 1 1 1 2 1 2 1 2 3 2 2 1`. Notice how we can define names within the arguments to the `data.frame` function. Of course, `sub` now represents the subject number of each observation. The first 10 elements of `sub1` are `1 2 3 4 5 1 2 3 4 5`. The variable `abcost` now refers to the value of `r2`. Notice that each of the 32 elements of `r2` is repeated `nsub` times. Thus the first 15 values of `abcost1` are `1 1 1 1 1 2 2 2 2 2 3 3 3 3 3`. Here are the first 10 rows of `abr1`:

```
    ab1 sub1 dcost1 abcost1
1     1    1      1       1
2     1    2      1       1
3     1    3      1       1
4     1    4      1       1
5     1    5      1       1
6     1    1      1       2
7     2    2      1       2
8     1    3      1       2
9     2    4      1       2
10    1    5      1       2
```

The following line makes a table of the means of `abr1`, according to the values of `dcost1` (rows) and `abcost1` (columns).

[3]The `cbind` function does the same thing but makes a matrix instead of a data frame.

```
ctab1 <- tapply(abr1[,1],list(abr1[,3],abr1[,4]),mean)
```

It uses the function `tapply`, which is like the `apply` function except that the output is a table. The first argument is the vector of data to be used. The second argument is a list supplying the classification in the table. This list has two columns corresponding to the columns of abr representing the classification. The third argument is the function to be applied to each grouping, which in this case is the mean. Here is the resulting table:

```
    1   2   3   4
1 2.6 3.0 3.7 3.8
2 3.5 4.4 4.4 5.4
3 4.5 5.2 5.1 5.9
4 5.1 6.1 6.2 6.8
```

The following line provides a plot corresponding to the table.

```
matplot(ctab1,type="l")
```

Type `l` means lines. Each line plots the four points in a column of the table. If you want it to go by rows, use `t(ctab1)` instead of `ctab1`. The function `t()` transposes rows and columns.

Finally, the following line does a regression of the response on the two classifiers, actually an analysis of variance.

```
summary(aov(ab1 ~ dcost1 + abcost1 + Error(sub1/(dcost1 + abcost1)),data=abr))
```

The function `aov`, like `lm`, fits a linear model, because `dcost1` and `abcost1` are numerical variables, not factors (although `sub1` is a factor). The model is defined by its first argument (to the left of the comma), where ˜ separates the dependent variable from the predictors. The second element defines the data frame to be used. The `summary` function prints a summary of the regression. (The `lm` and `aov` objects themselves contains other things, such as residuals, many of which are not automatically printed.) We explain the `Error` term later, but the point of it is to make sure that we test against random variation due to subjects, that is, test “across subjects.” Here is some of the output, which shows significant effects of both predictors:

```
Error: sub1
          Df Sum Sq Mean Sq F value Pr(>F)
Residuals  4 52.975  13.244

Error: sub1:dcost1
          Df  Sum Sq Mean Sq F value    Pr(>F)
dcost1     1 164.711 164.711  233.63 0.0001069 ***
Residuals  4   2.820   0.705
---

Error: sub1:abcost1
          Df Sum Sq Mean Sq F value   Pr(>F)
abcost1    1 46.561  46.561    41.9 0.002935 **
Residuals  4  4.445   1.111
---

Error: Within
           Df Sum Sq Mean Sq F value Pr(>F)
Residuals 145 665.93    4.59
```

4.3 Other ways to read in data

First example. Here is another example of creating a matrix with one row per observation.

```
symp1 <- read.table("symp1.data",header=T)
sy1 <- as.matrix(symp1[,c(1:17)])
```

The first 17 columns of `symp1` are of interest. The file `symp1.data` contains the names of the variables in its first line. The `header=T` (an abbreviation for `header=TRUE`) makes sure that the names are used; otherwise the variables will be names `V1, V2,` etc.

```
gr1 <- factor(symp1$group1)
```

The variable `group1`, which is in the original data, is a factor that is unordered.

The next four lines create the new matrix, defining identifiers for subjects and items in a questionnaire.

```
syv1 <- as.vector(sy1)
subj1 <- factor(rep(1:nrow(sy1),ncol(sy1)))
item <- factor(rep(1:ncol(sy1),rep(nrow(sy1),ncol(sy1))))
grp <- rep(gr1,ncol(sy1))
cgrp <- ((grp==2) | (grp==3))+0
```

The variable `cgrp` is a code for being in `grp` 2 or 3. The reason for adding `0` is to make the logical vector of `T` and `F` into a numeric vector of 1 and 0.

The following three lines create a table from the new matrix, plot the results, and report the results of an analysis of variance.

```
sytab <- tapply(syv,list(item,grp),mean)
matplot(sytab,type="l")
svlm <- aov(syv ~ item + grp + item*grp)
```

Second example. In the next example, the data file has labels. We want to refer to the labels as if they were variables we had defined, so we use the `attach` function.

```
t9 <- read.table("tax9.data",header=T)
attach(t9)
```

Third example. In the next example, the data file has no labels, so we can read it with `scan`. The `scan` function just reads in the numbers and makes them into a vector, that is, a single column of numbers.

```
abh1 <- matrix(scan("abh1.data"),,224,byrow=T))
```

We then apply the `matrix` command to make it into a matrix. (There are many other ways to do this.) We know that the matrix should have 224 columns, the number of variables, so we should specify the number of columns. If you say help(matrix) you will see that the matrix command requires several arguments, separated by commas. The first is the vector that is to be made into a matrix, which in this case is `scan("abh1.data")`. We could have given this vector a name, and then used its name, but there is no point. The second and third arguments are the number of rows and the number of columns. We can leave the number of rows blank. (That way, if we add or delete subjects, we don't need to change anything.) The number of columns is 224. By default, the matrix command fills the matrix by columns, so we need to say `byrow=TRUE` or `byrow=T` to get it to fill by rows, which is what we want. (Otherwise, we could just leave that field blank.)

We can refer to elements of abh1 by `abh1[row,column]`. For example, `abh[1,2]` is the sex of the first subject. We can leave one part blank and get all of it, e.g., `abh1[,2]` is a vector (column of numbers) representing the sex of all the subjects.

4.4 Other ways to transform variables

4.4.1 Contrasts

Suppose you have a matrix `t1` with 4 columns. Each row is a subject. You want to contrast the mean of columns 1 and 3 with the mean of columns 2 and 4. A t-test would be fine. (Otherwise, this is the equivalent of the `cmatrix` command in Systat.) Here are three ways to do it. The first way calculates the mean of the columns 1 and 3 and subtracts the mean of columns 2 and 4. The result is a vector. When we apply `t.test()` to a vector, it tests whether the mean of the values is different from 0.

```
t.test(apply(t1[c(1,3),],2,mean)-apply(t1[c(2,4),],2,mean))
```

The second way multiplies the matrix by a vector representing the contrast weights, `1, -1, 1, -1`. Ordinary multiplication of a matrix by a vector multiplies the rows, but we want the columns, so we must apply `t()` to transform the matrix, and then transform it back.

```
t.test(t(t(t1)*c(1,-1,1,-1)))
```

or

```
contr1 <- c(1,-1,1,-1)
t.test(t(t(t1)*contr1))
```

The third way is the most elegant. It uses matrix multiplication to accomplish the same thing.

```
contr1 <- c(1,-1,1,-1)
t.test(t1 %*% contr1)
```

4.4.2 Averaging items in a within-subject design

Suppose we have a matrix `t2`, with 32 columns. Each row is a subject. The 32 columns represent a 8x4 design. The first 8 columns represent 8 different levels of the first variable, at the first level of the second variable. The next 8 columns are the second level of the second variable, etc. Suppose we want a matrix in which the columns represent the 8 different levels of the first variable, averaged across the second variable.

First method: loop. One way to do it — inelegantly but effectively — is with a loop. First, we set up the resulting matrix. (We can't put anything in it this way if it doesn't exist yet.)

```
m2 <- t2[,c(1:8)]*0
```

The idea here is just to make sure that the matrix has the right number of rows, and all 0's. Now here is the loop:

```
for (i in 1:8) m2[,i] <- apply(t2[,i+c(8*0:3)],1,mean)
```

Here, the index `i` is stepped through the columns of `m2`, filling each one with the mean of four columns of `t2`. For example, the first column of `m2` is the mean of columns 1, 9, 17, and 25 of `t2`. This is because the vector `c(8*0:3)` is 0, 8, 16, 24. The `apply` function uses `1` as its second argument, which means to apply the function `mean` across *rows*.

Second method: matrix multiplication. Now here is a more elegant way, but one that requires an auxiliary matrix, which may use memory if that is a problem. This time we want the means according to the second variable, which has four levels, so we want a matrix with four columns. We will multiply the matrix `t2` by an auxiliary matrix `c0`.

The matrix `c0` has 32 rows and four columns. The first column is 1,1,1,1,1,1,1,1 followed by 24 0's. This is the result of `rep(c(1,0,0,0),rep(8,4))`, which repeats each of the elements of 1,0,0,0 eight times (since `rep(8,4)` means 8,8,8,8). The second column is 8 0's, 8 1's, and 16 0's.

```
c0 <- cbind(rep(c(1,0,0,0),rep(8,4)),rep(c(0,1,0,0),rep(8,4)),
 rep(c(0,0,1,0),rep(8,4)),rep(c(0,0,0,1),rep(8,4)))
c2 <- t2 %*% c0
```

The last line above uses matrix multiplication to create the matrix `c2`, which has 4 columns and one row per subject. Note that the order here is important; switching `t2` and `c0` will not work.

4.4.3 Selecting cases or variables

There are several other ways for defining new matrices or data frames as subsets of other matrices or data frames.

One very useful function is `which()`, which yields the indices for which its argument is true. For example, the output of `which(3:10 > 4)` is the vector `3 4 5 6 7 8`, because the vector `3:10` has a length of 8, and the first two places in it do not meet the criterion that their value is greater than 4. With `which()`, you can use a vector to select rows or columns from a matrix (or data frame). For example, suppose you have nine variables in a matrix `m9` and you want to select three sub-matrices, one consisting of variables 1, 4, 7, another with 2, 5, 8, and another with 3, 6, 9. Define `mvec` so that it is the vector `1 2 3 1 2 3 1 2 3`.

```
mvec9 <- rep(1:3,3)
m9a <- m9[,which(mvec9 == 1)]
m9b <- m9[,which(mvec9 == 2)]
m9c <- m9[,which(mvec9 == 3)]
```

You can use the same method to select subjects by any criterion, putting the `which()` expression before the comma rather than after it, so that it indicates rows.

4.4.4 Recoding and replacing numbers

Suppose you have `m1` a matrix of data in which 99 represents missing data, and you want to replace each 99 with `NA`. Simply say `m1[m1==99] <- NA`. Note that this will work only if `m1` is a matrix (or vector), not a data frame (which could result from a `read.table()` command). You might need to use the `as.matrix()` function first.

Sometimes you want to recode a variable, e.g., a column in a matrix. If `q1[,3]` is a 7-point scale and you want to reverse it, you can say

```
q1[,3] <- 8 - q1[,3]
```

Here is a more complicated example. This time `q2[,c(2,4)]` are two columns that must be recoded by switching 1 and 2 but leaving responses of 3 or more intact. To do this, say

```
q2[,c(2,4)] <- (q2[,c(2,4)] < 3) * (3 - q2[,c(2,4)]) +
 (q2[,c(2,4)] >= 3) * q2[,c(2,4)]
```

Here the expression `q2[,c(2,4)] < 3` is a two-column matrix full of `TRUE` and `FALSE`. By putting it in parentheses, you can multiply it by numbers, and `TRUE` and `FALSE` are treated as 1 and 0, respectively. Thus, `(q2[,c(2,4)] < 3) * (3 - q2[,c(2,4)])` switches 1 and 2, for all entries less than 3. The expression `(q2[,c(2,4)] >= 3) * q2[,c(2,4)]` replaces all the other values, those greater than or equal to 3, with themselves.

Finally, here is an example that will switch 1 and 3, 2 and 4, but leave 5 unchanged, for columns 7 and 9

```
q3[,c(7,9)] <- (q3[,c(7,9)]==1)*3 + (q3[,c(7,9)]==2)*4 +
 (q3[,c(7,9)]==3)*1 + (q3[,c(7,9)]==4)*2 + (q3[,c(7,9)]==5)*5
```

Notice that this works because everything on the right of `<-` is computed on the values in `q3` before any of these values are replaced.

4.4.5 Replacing characters with numbers

Sometimes you have questionnaire data in which the responses are represented as (for example) “y” and “n” (for yes and no). Suppose you want to convert these to numbers so that you can average them. The following command does this for a matrix `q1`, whose entries are `y`, `n`, or some other character for “unsure.” It converts `y` to 1 and `n` to -1, leaving 0 for the “unsure” category.

```
q1 <- (q1[,]=="y") - (q1[,]=="n")
```

In essence, this works by creating two new matrices and then subtracting one from the other, element by element.

4.5 Using R to compute course grades

Here is an example that might be useful as well as instructive. Suppose you have a set of grades including a midterm with two parts `m1` and `m2`, a final with two parts, and two assignments. You told the students that you would standardize the midterm scores, the final scores, and each of the assignment scores, then compute a weighted sum to determine the grade. Here, with comments, is an R file that does this. The critical line is the one that standardizes and computes a weighted sum, all in one command.

```
g1 <- read.csv("grades.csv",header=F) # get the list of scores
a1 <- as.vector(g1[,4])
m1 <- as.vector(g1[,5])
m2 <- as.vector(g1[,6])
a2 <- as.vector(g1[,7])
f1 <- as.vector(g1[,8])
f2 <- as.vector(g1[,9])
a1[a1=="NA"] <- 0 # missing assignment 1 gets a 0
m <- 2*m1+m2 # compute midterm score from the parts
f <- f1+f2
gdf <- data.frame(a1,a2,m,f)
gr <- apply(t(scale(gdf))*c(.10,.10,.30,.50),2,sum)
```

```
 # The last line standardizes the scores and computes their weighted sum
 # The weights are .10, .10, .30, and .50 for a1, a2, m, and f
gcut <- c(-2,-1.7,-1.4,-1.1,-.80,-.62,-.35,-.08,.16,.40,.72,1.1,2)
 # The last line defines cutoffs for letter grades.
glabels <- c("f","d","d+","c-","c","c+","b-","b","b+","a-","a","a+")
gletter <- cut(gr,gcut,glabels) # creates a vector of letter grades
grd <- cbind(g1[,1:2],round(gr,digits=4),gletter) # makes a matrix
 # gl[,1:2] are students’ names
grd[order(gr),] # sorts the matrix in rank order and prints it
round(table(gletter)/.83,1) # prints, with rounding
 # the .83 is because there are 83 students, also gets percent
gcum <- as.vector(round(cumsum(table(gletter)/.83),1))
names(gcum) <- glabels
gcum # prints cumulative sum of students with different grades
```

5 Graphics

R can do presentation-quality and publication-quality graphics. These often require some trial-and-error manipulation of labels, line styles, axes, fonts, etc., and you should consult the help pages and the *Introduction* for more details. The emphasis in this section is on the use of graphics for data exploration, but we provide some leads into the more advanced uses.

One trick with graphics is to know how each of the various graphics commands responds (or fails to respond) to each kind of data object: `data.frame, matrix,` and `vector`. Often, you can be surprised.

5.1 Default behavior of basic commands

Here is the default behavior for each object for each of some of the plotting commands, e.g., `plot(x1)` where `x1` is a vector, matrix, or data frame.

	vector	matrix	data.frame
`plot`	values as function of position	2nd column as function of 1st	plots of each column as function of others
`boxplot`	one box for whole vector	one box for all values in matrix	one box for each column (variable)
`barplot`	one bar for each position, height is value	one bar for each column, summing successive values in colors	error
`matplot`	one labeled point for each position, height is value	X axis is row, Y is value, label is column	X axis is row, Y is value, label is column

The `barplot` of a matrix is an interesting display worth studying. Each bar is stack of smaller bars in different colors. Each smaller bar is a single entry in the matrix. The colors represent the row. Adjacent negative and positive values are combined. (It is easier to understand this plot if all values have the same sign.)

5.2 Other graphics

To get a bar plot of the column means in a data frame `df1`, you need to say
`barplot(height=apply(df1),2,mean))`.

To get a nice parallel coordinate display like that in Systat, use `matplot` but transform the matrix and use lines instead of points, that is: `matplot(t(mat1),type="l")`. You can abbreviate `type` with `t`.

`matplot(v1, m1, type="l")` also plots the columns of the matrix `m1` on one graph, with `v1` as the horizontal axis. This is a good way to get plots of two functions on one graph.

To get scatterplots of the columns of a matrix against each other, use `pairs(x1)`, where `x1` is a matrix or data frame. (This is like "splom" in Systat, which is the default graph for correlation matrices.)

Suppose you have a measure `y1` that takes several different values, and you want to plot histograms of `y1` for different values of `x1`, next to each other for easy comparison. The variable `x1` has only two or three values. A good plot is `stripchart(y1 ~ x1, method='stack')`. When `y1` is more continuous, try `stripchart(y1 ~ x1, method='jitter')`.

Here are some other commands in their basic form. There are several others, and each of these has several variants. You need to consult the help pages for details.

`plot(v1,v2)` makes a scatterplot of `v2` as a function of `v1`. If `v1` and `v2` take only a small number of values, so that the plot has many points plotted on top of each other, try `plot(jitter(v1),jitter(v2))`.

`hist(x1)` gives a histogram of vector `x1`.

`coplot(y1 ~ x1 | z1)` makes several plots of `y1` as a function of `x1`, each for a different range of values of `z1`.

`interaction.plot(factor1,factor2,v1)` shows how `v1` depends on the interaction of the two factors.

Many wonderful graphics functions are available in the Grid and Lattice packages. Many of these are illustrated and explained in Venables and Ripley (1999).

5.3 Saving graphics

To save a graph as a `png` file, say `png("file1.png")`. Then run the command to draw the graph, such as `plot(x1,y1)`. Then say `dev.off()`. You can change the width and height with arguments to the function. There are many other formats aside from png, such as pdf, and postscript. See `help(Devices)`.

There are also some functions for saving graphics already made, which you can use after the graphic is plotted: `dev.copy2eps("file1.eps")` and `dev2bitmap()`.

5.4 Multiple figures on one screen

The `par()` function sets graphics parameters. One type of parameter specifies the number and layout of multiple figures on a page or screen. This has two versions, `mfrow` and `mfcol`. The command `par(mfrow=c(3,2))`, sets the display for 3 rows and 2 columns, filled one row at a time. The command f`par(mfcol=c(3,2))` also specifies 3 rows and 2 columns, but they are filled one column at a time as figures are plotted by other commands.

Here is an example in which three histograms are printed one above the other, with the same horizontal and vertical axes and the same bar widths. The breaks are every 10 units. The `freq=FALSE` command means that densities are specified rather than frequencies. The `ylim` commands set the range of the vertical axis. The `dev.print` line prints the result to a file. The next three lines print out the histogram as numbers rather than a plot; this is accomplished with `print=FALSE`. These are then saved to `hfile1`.

```
par(mfrow=c(3,1))
hist(vector1,breaks=10*1:10,freq=FALSE,ylim=c(0,.1))
hist(vector2,breaks=10*1:10,freq=FALSE,ylim=c(0,.1))
hist(vector3,breaks=10*1:10,freq=FALSE,ylim=c(0,.1))
dev.print(png,file="file1.png",width=480,height=640)
h1 <- hist(vector1,breaks=10*1:10,freq=FALSE,ylim=c(0,.1),plot=FALSE)
h2 <- hist(vector2,breaks=10*1:10,freq=FALSE,ylim=c(0,.1),plot=FALSE)
```

```
h3 <- hist(vector3,breaks=10*1:10,freq=FALSE,ylim=c(0,.1),plot=FALSE)
sink("hfile1")
h1
h2
h3
sink()
```

For simple over-plotting, use `par(new=T)`. Of course, this will also plot axis labels, etc. To avoid that, you might say `par(new=T,ann=F)`. (Apparent undocumented feature: this setting conveniently disappears after it is used once.) To plot several graphs of the same type, you can also use `points()`, `lines()`, or `matplot()`.

5.5 Other graphics tricks

When you use `plot()` with course data (e.g., integers), it often happens that points fall on top of each other. There are at least three ways to deal with this. One is to use `stripchart()` (see above). Another is to apply `jitter()` to one or both of the vectors plotted against each other, e.g., `plot(jitter(v1),v2)`. A third is to use `sunflowerplot(v1,v2)`, which uses symbols that indicated how many points fall in the same location.

Use `identify()` to find the location and index of a point in a scatterplot made with `plot()`. Indicate the point you want by clicking the mouse on it. The function `locator()` just gives the coordinates of the point. This is useful for figuring out where you want to add things to a plot, such as a legend.

`text()` uses a vector of strings instead of points in a plot. If you want a scatterplot with just these name, first make an empty plot (with `type="n"`) to get the size of the plot correct) and then use the text command, e.g.:

```
x <- 1:5
plot(x,x^2,type="n")
text(x,x^2,labels=c("one","two","three","four","five"),col=x)
```

In this case, the `col=x` argument plots each word in a different color.

To put a legend on a plot, you can use the "`legend=`" argument of the plotting function, or the `legend()` function, e.g., `legend(3,4,legend=c("Self","Trust"),fill=c("gray25","gray75"))`. This example illustrates the use of gray colors indicated by number, which is convenient for making graphics for publication. (For presentation or data exploration, the default colors are usually excellent.)

Several functions will draw various things on graphs. `polygon()` and `segments()` draw lines. They differ in the kind of input they want, and the first one closes the polygon it draws.

6 Statistics

This section is not a summary of all statistics but, rather, a set of notes on procedures that are more useful to the kind of studies that psychology researchers do.

6.1 Very basic statistics

Here is a list of some of the commands psychologists use all the time:

`t.test(a1,b1)` — Test the difference between the means of vectors `a1` and `b1`.

`t.test(a1,b1,paired=TRUE)` or `t.test(a1,b1,p=T)`, or, even simpler, `t.test(b1-a1)` — Test the difference between means of vectors `a1` and `b1` when these represent paired measures, e.g., variables for the same subject. The vectors can be parts of matrices or data.frames. A good plot to look at is

```
plot(a1,b1)
abline(0,1)
```

This plots `b1` as a function of `a1` and then draws a diagonal line with an intercept of 0 and a slope of 1. Another plot is `matplot(t(cbind(a1,b1)),type="l")`, which shows one line for each pair.

Sometimes you want to do a t-test comparing two groups represented in the same vector, such as males and females. For example, you have a vector called `age1` and a vector called `sex1`, both of the same length. Subject `i1`'s age and sex are `age1[i1]` and `sex1[i1]`. Then a test to see if the sexes differ in age is `t.test(age1[sex1==0],age1[sex1==1])` (or perhaps `t.test(age1[sex1==0],age1[sex1==1],var.equal=T)` for the assumption of equal variance). A good plot to do with this sort of test is
`stripchart(age1 sex1,method='jitter')` (or `stripchart(age1 sex1,method='stack')` if there are only a few ages represented).

The binomial test (sign test) for asking whether heads are more likely than tails (for example) uses `prop.test(h1,n1)`, where `h1` is the number of heads and `n1` is the number of coin tosses. Suppose you have two vectors `a1` and `b1` of the same length, representing pair of observations on the same subjects, and you want to find whether `a1` is higher than `b1` more often than the reverse. Then you can say `prop.test(sum(a1>b1), sum(a1>b1)+sum(a1<b1))`. The use of the sum of sums as the second argument excludes those cases where `a1==b1`. `prop.test` can also be used to compare several different proportions. (See the help file.)

`chisq.test(x1)` does a chi-square test on a matrix `x1`, where the cells represent counts in a classification table. There are several other ways of using this function. One other useful one is `chisq.test(a1,b1)`, where `a1` and `b1` are vectors or factors of the same length, representing the levels on which observations are classified. For example,

```
a1 <- c(1,1,1,1,1,1,1,0,0,0,0,0,0)
b1 <- c(1,1,1,1,1,1,0,0,0,0,0,0,1)
chisq.test(a1,b1)
```

will yield an almost significant result, since the two vectors match except for two cases. If you want a Fisher exact test instead of chi-square (for a 2x2 table), you just use `fisher.test()` instead of `chisq.test()`.

A related test is `mantelhaen.test`. It, and many useful nonparametric tests, are in the `ctest` package (which is loaded automatically but has its own help listing separate from `base`). Some of these tests can be exact. More generally, see the `loglin` function, which requires no special package, and the `loglm` function in the `MASS` package, which allows models to be specified in a form like linear models.

`cor(a1)` — Show the correlations of the columns for a matrix or data frame `a1`.

`cor.test(a1,b1)` — Show the correlation between vectors `a1` and `b1` and its significance.

Partial correlation is the correlation of the residuals, and this is one way to compute it. Thus, the partial correlation of `x1` and `y1`, partialling `z1`, is `cor(lm(x1~z1)$resid,lm(y1~z1)$resid)`. As we shall explain, `lm` is the function to fit a linear model, and `resid` is one of the elements that it returns (not usually printed, but available). The significance of the partial correlation is the same as the significance of the regression coefficient, so (again, as we shall explain) use `summary(lm(x1 ~ y1 + z1))` to get it.

For factor analysis, you need the `mva` library, so say `library(mva)` to load it. The main command is `factanal(m1,factors=3)`. (The number of factors can be varied.) Varimax is the default rotation, but you can also say, for example, `factanal(m1,factors=4,rota`

Principal components analysis is also in the `mva` library. The most useful commands are `print(prcomp(x1))` to get the components of matrix or data.frame `x1`, and `plot(prcomp(x1))` to see a scree plot of the variances accounted for by the components. The eigenvalues are the squares of the `sdev` values that are part of the output of `print(prcomp(x1))`.

The same library also does cluster analysis, e.g., `kmeans(x1,3)` where 3 is the number of clusters. If you want to define a factor identifying the cluster of each subject, you can say
`f1 <- as.factor(kmeans(x1,3)$cluster)`, or, if you just want the numbers and don't care about whether you

have a factor, `v1 <- kmeans(x1,3)$cluster`. To see a plot of the variable means for the 3 clusters, say `matplot(t(v1),t="l")`. (The last part abbreviates `type="lines"`.

Multiple tests are well handled by the `multtest` package, with is part of the Bioconductor packages. (See Section 2.2.2.) The documentation in that package provides a good introduction, with citations. The classic Bonferroni method is often unnecessarily conservative. On the other hand, multiple tests can sometimes be avoided by clear statement of a hypothesis and an effort to find the single best test of it.

6.2 Linear regression and analysis of variance (anova)

If you want to find whether `y1` depends on `x1` and `x2`, the basic thing you need is

```
lm(y1 ~ x1 + x2)
```

If these variables are part of a data frame called df1, then you can sayl `lm(y1 ~ x1 + x2, data=df1)`, or you can say `attach(df1)` before you run the analysis.

Note that `lm()` by itself doesn't do much that is useful. If you want a summary table, one way to get it is to say

```
summary(lm(y ~ x1 + x2))
```

The coefficients are unstandardized. If you want standardized coefficients, use `summary(lm(scale(y)  scale(x1) + scale(x2)))`. The `scale()` function standardizes vectors by default (and it does many other things, which you can see from `help(scale)`).

Another way to get a summary is with `anova()`. The `anova()` command is most useful when you want to compare two models. For example, suppose that you want to ask whether `x3` and `x4` together account for additional variance after x1 and x2 are already included in the regression. You cannot tell this from the summary table that you get from

```
summary(lm(y1 ~ x1 + x2 + x3 + x4))
```

That is because you get a test for each coefficient, but not the two together. So, you can do the following sequence:

```
model1 <- lm(y1 ~ x1 + x2)
model2 <- lm(y1 ~ x1 + x2 + x3 + x4)
anova(model1,model2)
```

As you might imagine, this is an extremely flexible mechanism, which allows you to compare two nested models, one with many predictors not contained in the other. Note that `anova` reports sums of squares sequentially, building up by adding the models successively. It is thus different from the usual report of a multiple regression, `summary(lm(...))`. Note also that you can add and drop variables from a model without retyping the model, using the functions `add()` and `drop()`.

6.3 Reliability of a test

Suppose `v1` is a matrix in which the rows are subjects and each column is a test item. You want to calculate the coefficient alpha, a measure of the reliability of the test (Lord & Novick, 1968, p. 88):

$$\alpha = \frac{n}{n-1}\left[1 - \frac{\sum \sigma^2(Y_i)}{\sigma_X^2}\right]$$

Here is the expression of this in R. The expression `nv` is the number of variables. (If you know this, you can just put it in the formula instead of `nv`).

```
nv1 <- ncol(v1)
(nv1/(nv1-1))*(1 - sum(apply(v1,2,var))/var(apply(v1,1,sum)))
```

Crucial here is the use of the `apply` function to rows and columns. The first use of `apply` finds the variance of each column of the matrix. The 2 indicates columns. Then we take the sum of these. The second application finds the total score for each subject by applying the function `sum` to each row. We then find the variance of this sum.

Another way to compute alpha involves the variance-covariance matrix of the items.

```
tvar1 <- var(v1, na.rm = FALSE)  # missing data aborts var()
```

The sum of values along the main diagonal of the variance-covariance matrix `(tvar1)` equals the numerator of the right-hand term in the formula, and the sum of all elements in `tvar1` equals the denominator, so alpha is:

```
(nv1/(nv1-1)) * (1 - sum(diag(tvar1))/sum(tvar1))
```

This approach is better for large data sets, because the `apply` function uses a lot of memory.

Now suppose that you want to see what happens if you delete item 3. You can do this by deleting variables in each formula (remembering in each case to change the value of `nv1`):

```
(nv1/(nv1-1)*(1 - sum(apply(v1[,-3],2,var))/var(apply(v1[-3],1,sum)))

(nv1/(nv1-2)) * (1 - sum(diag(tvar[-3,-3]))/sum(tvar[-3,-3]))
```

The advantage of using the variance-covariance matrix is that the effect of deleting certain items can be determined easily.

6.4 Goodman-Kruskal gamma

The Goodman-Kruskal gamma statistic (also known as γ) is like tau (τ) except for the denominator. It is an easily interpreted measure of rank correlation between two vectors `v1` and `v2`. The idea is to consider all pairs of observations in each vector: observations 1 and 2, 1 and 3, and so on. Count the number of times their ordering agrees, and call this `S+`. Count the number of times their ordering disagrees and call this `S-`. If one variable is tied, this does not count either way. Then $\gamma = fracS + -S - S + +S-$. In other words, it is the number of agreements minus the number of disagreements, all divided by the total of agreements and disagreements. A γ of 1 means that the correlation is as high as it can be, given the ties.

Gamma is available in the Hmisc package, in the `rcorr.cens()` function. To compute it for `v1` and `v1`, say rcorr.cens(v1,v2,outx=T). The `outx` argument concerns whether ties are ignored (as they should be). The `Dxy` value is what you want.

An instructive, but slower, function to compute gamma is:

```
goodman <- function(x,y){
  Rx <- outer(x,x,function(u,v) sign(u-v))
  Ry <- outer(y,y,function(u,v) sign(u-v))
  S1 <- Rx*Ry
  return(sum(S1)/sum(abs(S1)))}
```

To compute gamma for `v1` and `v2`, say `goodman(v1,v2)`.

6.5 Inter-rater agreement

An interesting statistical question came up when we started thinking about measuring the agreement between two people coding video-tapped interviews. This section discusses two such measures. One is the percentage agreement among the raters, the other is the kappa statistic commonly used for assessing inter-rater reliability (not to be confused with the R function called `kappa`). We will first summarize how either of them is derived, then we will use an example to show that kappa is better than percentage agreement.

Our rating task is as follows. Two raters, LN and GF, viewed the video-tapped interviews of 10 families. The raters judged the interviews on a check list of 8 items. The items were about the parent's attitude and goals. The rater marks a 'yes' on an item if the parents expressed feelings or attitudes that fit the item and 'no' otherwise. A yes is coded as 1 and a no 0.

The next table shows how the two raters classified the 10 families on Items 2 and 4.

					Family					
Item 2										
	A	B	C	D	E	F	G	H	I	J
LN	0	0	0	0	1	1	1	1	1	1
GF	0	1	1	1	1	1	1	1	1	1
Item 4										
LN	0	1	0	0	1	1	0	0	0	0
GF	0	1	0	0	1	0	0	1	0	1

Note that in both items, the two raters agreed on the classifications of 7 out of 10 families. However, In Item 2, rater LN gave more no's and GF gave about equal yeses and no's. In Item 4, rater GF gave 9 yeses and only 1 no. It turns out that this tendency to say yes or no affects the raters' agreement adjusted for chance. We will get to that in a moment.

Suppose that Item 2 was whether or not our interviewees thought that "learning sign language will mitigate the development of speech for a child who is deaf or hard of hearing". We want to know how much LN and GF agreed. The agreement is what we call an inter-rater reliability. They might agree positively (both LN and GF agreed that the parents thought so) or negatively (i.e., a no - no pair).

Our first measure, the percentage of agreement, is the proportion of families that the raters made the same classifications. We get a perfect agreement (100%) if the two raters make the same classification for every family. A zero percent means complete disagreement. This is straightforward and intuitive. People are familiar with a 0% to 100% scale.

One problem with percent agreement is that it does not adjust for chance agreement, the chance that the raters happen to agree on a particular family. Suppose, for example, that after the raters have forgotten what they did the first time, we ask them to view the videotape of family A again. Pure chance may lead to a disagreement this time; or perhaps even an agreement in the opposite direction.

That is where the κ statistic comes in. Statistics like kappa adjust for chance agreement by subtracting them out:

$$\kappa = \frac{\Pr(\text{observed agreement}) - \Pr(\text{chance agreement})}{\Pr(\text{maximum possible agreement}) - \Pr(\text{chance agreement})},$$

where the chance agreement depends on the marginal classifications. The marginal classifications, in our case, refer to each rater's propensity to say "yes" or "no". The chance agreement depends in part on how extreme the raters are. If, for example, rater 1 gave 6 yeses and 4 no's and rater 2 gave 9 yeses and only 1 no, then there is a higher chance for the raters to agree on yes-yes rather than no-no; and a disagreement is more likely to occur when rater 2 says yes and rater 1 says no.

Therefore, for the same proportion of agreement, the chance-adjusted kappa may be different. Although we do not usually expect a lot of difference. We can use the following example to understand how it works.

The numbers in the following table are the number of families who were classified by the raters. In both items, raters LN and GF agreed on the classification of 7 families and disagreed on 3 families. Note that they had very different marginal classifications.

If we only look at the percentage of agreement, then LN and GF have the same 70% agreement on Items 2 and 4. However, the κ agreement is 0.29 for Item 2 and 0.35 for Item 4.

		Item 2			Item 4		
		rater GF			rater GF		
		yes	no	marginal	yes	no	marginal
rater	yes	6	0	6	2	1	3
LN	no	3	1	4	2	5	7
		9	1	10	4	6	10

Why? We can follow the formula to find out. In both items, the observed agreement, when expressed as counts, is the sum of the numbers along the diagonal. For Item 2 it is $(6+1)=7$. Divide that by 10 you get 70% agreement. The maximum possible number of agreement is therefore $10/10$.

The chance agreement for Item 2 is $(6/10)\times(9/10)+(4/10)\times(1/10)$. That is the probability of both raters said 'yes' plus both said 'no'. Rater LN gave 6 yeses and GF gave 9 yeses. There is a $\frac{6}{10}$ probability for LN to say yes and a $\frac{9}{10}$ probability for GF to say yes. Therefore, the joint probability, i.e., the chance for us to get a yes-yes classification, is $\frac{6}{10}\times\frac{9}{10}$. Similarly, the probability of a no-no classification is $\frac{4}{10}\times\frac{1}{10}$.

For Item 2, we have $\kappa = \frac{7}{10}-\frac{58}{100}/\frac{10}{10}-\frac{58}{100} = 0.29$. The κ for Item 4 is $\frac{7}{10}-\frac{54}{100}/\frac{10}{10}-\frac{54}{100} = 0.35$. The kappa statistics are different between Items 2 and 4 because their chance agreements are different. One is $\frac{58}{100}$ and the other is $\frac{54}{100}$. The marginals of the two tables show us that the two raters made more yes judgments in one instance and more no judgments in the other. That in itself is OK, the raters make the classifications according to what they observe. There is no reason for them to make equal amount of yeses and no's. The shift in the propensity to make a particular classification inevitably affects getting an agreement by chance. This correction for chance may lead to complications when the raters are predominantly positive or negative. The paper by Guggenmoos-Holzmann (1996) has a good discussion.[4]

The same principle applies to two raters making multiple classifications such as 'aggressive', 'compulsive', and 'neurotic', or some other kinds of judgments. An important thing to remember is that we are only using kappa to compare classifications that bear no rank-ordering information. Here a 'yes' classification is not better or worse than a 'no'. There are other ways to check agreement, for example, between two teachers giving letter grades to homework assignments. An 'A+' grade is better than an 'A-'. But that is a separate story.

Kappa is available in the `e1071` package as `classAgreement()`, which requires a contingency table as input.

The following function, which is included for instructional purposes (given that R already has a function for kappa), also computes the kappa agreement between two vectors of classifications. Suppose we want to calculate the agreement between LN and GF on Item 2, across 10 interviews. The vector `r1` contains the classifications from LN, which is `c(0, 0, 0, 0, 1, 1, 1, 1, 1, 1)`; and `r2` contains GF's classifications, `c(0, 1, 1, 1, 1, 1, 1, 1, 1, 1)`. The `kappaFor2` function returns the overall κ statistic and the standard error. The test statistic is based on a z test, and the two-tailed p-value for the null hypothesis that $\kappa = 0$ is also returned.

```
kappaFor2 <- function(r1, r2, na.method = "na.rm")
{
    if (na.method == "na.rm")
        na.rm <- T
    else na.rm <- F
    ttab <- table(r1, r2)
```

[4]Guggenmoos-Holzmann, I. (1996). The meaning of kappa: probabilistic concepts of reliability and validity revisited. Journal of Clinical Epidemiology, 49(7), 775–782.

```
    tsum <- sum(ttab, na.rm = na.rm)
    #
    # change the counts into proportions
    #
    ttab <- ttab/tsum
    #
    # find the marginals
    #
    tm1 <- apply(ttab, 1, sum, na.rm = na.rm)
    tm2 <- apply(ttab, 2, sum, na.rm = na.rm)
    #
    agreeP <- sum(diag(ttab), na.rm = na.rm)
    chanceP <- sum(tm1 * tm2, na.rm = na.rm)
    kappa2 <- (agreeP - chanceP)/(1 - chanceP)
    kappaSE <- 1/((1 - chanceP) * sqrt(tsum)) * sqrt(chanceP +
      chanceP^2 - sum(tm1 * tm2 * (tm1 + tm2), na.rm = na.rm))
    # browser()
    kz <- kappa2/kappaSE
    kp <- 2 * (1 - pnorm(kz))
    ans <- c(kappa2, kappaSE, kz, kp)
    names(ans) <- c("kappa", "S.E.", "z.stat", "p.value")
    return(ans)
}
```

Sometimes a function requires further editing. Suppose the `kappaFor2` function is entered at R's system prompt (i.e., the `>` character), and there was an error. Then we can type the command `kappaFor2 <- emacs(kappaFor2)` to edit its contents with the emacs editor; or use `vi()` for the visual editor. In the above function there is a `browser()` command, commented out by a # character. The `browser()` command is used to debug a function. Each time R runs the `kappaFor2` function, it stops at the point where a working `browser()` command was set, and we can debug the function by examining the variables inside the function.

We run the function `kappaFor2` and the results show that the agreement on Item 2 is not reliably greater than 0, with a two-tail *p*-value of about 0.20.

```
> kappaFor2(r1 = c(0, 0, 0, 0, 1, 1, 1, 1, 1, 1),
            r2 = c(0, 1, 1, 1, 1, 1, 1, 1, 1, 1))
    kappa      S.E.    z.stat   p.value
0.2857143 0.2213133 1.2909944 0.1967056
```

6.6 Generating random data for testing

Suppose you want to test that the last two formulas yield the same result, but you don't have any data handy. You can generate a sample of 10 Likert-type, 5-point scale responses from 100 Ss as follows:

```
v1 <- matrix(sample(c(1, 2, 3, 4, 5), size=1000, replace=T), ncol=10)
```

6.7 Within-subject correlations and regressions

Suppose you have a set of 8 items with 2 measures for each, and each subject answers both questions about each item. You want to find out how the answers to the two questions correlate *within* each subject, across the 8 items. Then you want to find the average, across subjects, of these within-subject correlations. The matrices `m1` and `m2` contain the data for the questions. Each matrix has 8 columns, corresponding to the 8 items, and one row per subject. The following will do it:

```
nsub1 <- nrow(m1)
cors1 <- rep(NA,nsub1)
for (i in 1:nsub1) cors1[i] <- cor(m1[i,],m2[i,])
```

The first line finds the number of subjects, `nsub1`. The second line creates an vector of `nsub1 NA`'s to hold the correlations, one for each subject. The third line fills this vector with the correlations using a loop, which uses `i` to index the subject. Now, if we want to do a `t` test to see if the correlations are positive, for example, we can say `t.test(cors1)`.

Similarly, you can store within-subject regressions in a matrix, as in the following example.

```
# first set up a matrix to hold the results, including the intercept
reg1 <- matrix(NA,4,nsub1) # nsub1 is the number of subjects
for (x in 1:ns) reg1[x] <- lm(y1[x,]~x1[x,]+x2[x,]+x3[x,])$coefficients
t.test(reg1[,1]) # is the mean intercept positive?
t.test(reg1[,2]) # is the mean first coefficient positive?
```

This works because `lm()` produces a list, and element `coefficients` of that list is the coefficients. This element itself may be decomposed into the intercept, the first coefficient, and so on.

6.8 Advanced analysis of variance examples

We now turn to repeated-measure analysis of variance using the `aov()` function in R.[5] The `aov()` function is used to produce a Univariate ANOVA table similar to the one produced by SAS, SPSS, and Systat. The SAS syntax of an identical analysis is also listed in example 2 for comparison.[6][7]

6.8.1 Example 1: Mixed effects model (Hays, 1988, Table 13.21.2, p. 518)

The following example shows you how to carry out repeated-measure analysis of variance. Repeated-measure designs are common in experimental psychology. We use the data in Hays (1988), but we change the story behind it to make it easier to understand. Imagine a psychologist is asked to conduct a study to help design the control panel of a machine that delivers medicine by intravenous infusion. The main purpose of the study is to find the best shape and color of the buttons on the control panel to improve efficiency and prevent potential errors. The psychologist wants to know how quickly users (physicians) respond to buttons of different colors and shapes. To simplify the example, suppose that the psychologist hypothesizes that bright colors are easier to see than dark colors so the users respond to them faster. In addition, she thinks that users can spot circles faster than squares. Thus she has two effects to test, one for color (a bright color compared to a dark color) and the other for shape (round vs. square), and she wants to know if the physician respondents show shorter reaction time with a particular color and shape combination.

[5] We drop our convention here of using numbers in made-up variable names, in order to be consistent with the original names in the examples we cite.

[6] The following examples are inspired by the examples of Gabriel Baud-Bovy, `<baudbovy@fpshp1.unige.ch>` contributed to S-News `<http://www.stat.cmu.edu/s-news/>`, entitled "ANOVAs and MANOVAs for repeated measures (solved examples)," dated 2/17/1998.

[7] The statistics theory behind the syntax can be found in the references so detailed explanations are not provided here. The examples are:
1) Hays, Table 13.21.2, p. 518 (1 dependent variable, 2 independent variables: 0 between, 2 within)
2) Maxwell and Delaney, p. 497 (1 dependent variable, 2 independent variables: 0 between, 2 within)
3) Stevens, Ch. 13.2, p. 442 (1 dependent variable, 1 independent variable: 0 between, 1 within)
4) Stevens, Ch. 13.12, p. 468 (1 dependent variable, 3 independent variables: 1 between, 2 within)

The psychologists knows that she will be able to recruit only some physicians to run the test apparatus. Thus she wants to collect as many test results as possible from a single respondent. Each physician is then given four trials, one with a test apparatus of round red buttons, one with square red buttons, one with round gray buttons, and one with square gray buttons. Here the users only try each arrangement once, but in real life the psychologist could ask the users to repeat the tests several times in random order to get a more stable response time.

An experimental design like this is called a “repeated measure” design because each respondent is measured repeatedly. In social sciences it is often referred to as a within-subject design because the measurements are made repeatedly within individual subjects. The variables `shape` and `color` are therefore called within-subject variables. It is possible to do the experiment between subjects, that is, each reaction time data point comes from a different subject. A completely between-subject experiment is also called a randomized design. If done between-subject, the experimenter would need to recruit four times as many subjects. This is not a very efficient way of collecting data

This example has 2 within-subject variables and no between subject variables:

- one dependent variable: time required to solve the puzzles
- one random effect: subject (see Hays for reasons why)
- 2 within-subject fixed effects: shape (2 levels), color (2 levels)

We first enter the reaction time data into a vector `data1`. Then we will transform the data into appropriate format for the repeated analysis of variance using `aov()`.

```
data1<-c(
49,47,46,47,48,47,41,46,43,47,46,45,
48,46,47,45,49,44,44,45,42,45,45,40,
49,46,47,45,49,45,41,43,44,46,45,40,
45,43,44,45,48,46,40,45,40,45,47,40) # across subjects then conditions
```

We can take a look at the data in a layout that is easier to read. Each subject takes up a row in the data matrix.

```
> matrix(data1, ncol= 4, dimnames =
+ list(paste("subj", 1:12), c("Shape1.Color1", "Shape2.Color1",
+ "Shape1.Color2", "Shape2.Color2")))

        Shape1.Color1 Shape2.Color1 Shape1.Color2 Shape2.Color2
subj 1             49            48            49            45
subj 2             47            46            46            43
subj 3             46            47            47            44
subj 4             47            45            45            45
subj 5             48            49            49            48
subj 6             47            44            45            46
subj 7             41            44            41            40
subj 8             46            45            43            45
subj 9             43            42            44            40
subj 10            47            45            46            45
subj 11            46            45            45            47
subj 12            45            40            40            40
```

Next we use the `data.frame()` function to create a data frame `Hays.df` that is appropriate for the `aov()` function.

```
Hays.df <- data.frame(rt = data1,
subj = factor(rep(paste("subj", 1:12, sep=""), 4)),
shape = factor(rep(rep(c("shape1", "shape2"), c(12, 12)), 2)),
color = factor(rep(c("color1", "color2"), c(24, 24))))
```

The experimenter is interested in knowing if the shape (`shape`) and the color (`color`) of the buttons affect the reaction time (`rt`). The syntax is:

```
aov(rt ~ shape * color + Error(subj/(shape * color)), data=Hays.df)
```

We provide the `aov()` function with a formula, `rt ~ shape * color`. The asterisk is a shorthand for `shape + color + shape:color`. The `Error(subj/(shape * color))` is very important for getting the appropriate statistical tests. We will first explain what the syntax means, then we will explain why we do it this way.

The `Error(subj/(shape * color))` statement is used to break down the sums of squares into several pieces (called error strata). The statement is equivalent to `Error(subj + subj:shape + subj:color + subj:shape:color)`, meaning that we want to separate the following error terms: one for subject, one for subject by shape interaction, one for subject by color interaction, and one for subject by shape by color interaction.

This syntax generates the appropriate tests for the within-subject variables `shape` and `color`. When you run a `summary()` after an analysis of variance model, you get

```
> summary(aov(rt ~ shape * color + Error(subj/(shape*color)), data=Hays.df))

Error: subj
          Df  Sum Sq Mean Sq F value Pr(>F)
Residuals 11 226.500  20.591

Error: subj:shape
          Df  Sum Sq Mean Sq F value  Pr(>F)
shape      1 12.0000 12.0000  7.5429 0.01901 *
Residuals 11 17.5000  1.5909
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Error: subj:color
          Df  Sum Sq Mean Sq F value   Pr(>F)
color      1 12.0000 12.0000  13.895 0.003338 **
Residuals 11  9.5000  0.8636
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Error: subj:shape:color
            Df    Sum Sq   Mean Sq   F value Pr(>F)
shape:color  1 1.200e-27 1.200e-27 4.327e-28      1
Residuals   11   30.5000    2.7727
```

Note that the shape of the button is tested against the subject by shape interaction, shown in the `subj:shape` error stratum. Similarly, color is tested against subject by color interaction. The last error stratum, the `Error: subj:shape:color` piece, shows that the two-way interaction `shape:color` is tested against the sum of square of `subj:shape:color`.

Without the `Error(subj/(shape * color))` formula, you get the wrong statistical tests:

```
summary(aov(rt ~ shape * color, data=Hays.df))
```

```
            Df    Sum Sq   Mean Sq   F value Pr(>F)
shape        1    12.000    12.000    1.8592 0.1797
color        1    12.000    12.000    1.8592 0.1797
shape:color  1 1.342e-27 1.342e-27 2.080e-28 1.0000
Residuals   44   284.000     6.455
```

All the variables are tested against a common entry called “Residuals”. The “Residuals” entry is associated with 44 degrees of freedom. This common Residuals entry is the sum of all the pieces of residuals in the previous output of `Error(subj/(shape * color))`, with 11 degrees of freedom in each of the four error strata.

Hays (1988) provides explanations on why we need a special function like `Error()`. In this experiment the psychologist only wants to compare the reaction time differences between round and square buttons. She is not concerned about generalizing the effect to buttons of other shapes. We say that the reaction time difference between round and square buttons a “fixed” effect. The variable `shape` is a “fixed” factor, meaning that in this case the number of possible shapes is fixed to two—round and square. The reaction time differences between the two conditions do not generalize beyond these two shapes. Similarly, the variable `color` is also considered fixed (again the effect not generalizable to colors other than red and gray).

However, the experimenter is concerned about generalizing the findings to other potential test subjects. The 12 subjects reported here belong to a random sample of numerous other potential users of the device. The study would not be very useful without this generalizability because the results of the experiments would only apply to these particular 12 test subjects. Thus the effect associated with the variable `subject` is considered “random.”

In a repeated-measure design where the within-subject factors are considered fixed effects and the only random effect comes from subjects, the within-subject factors are tested against their interaction with the random subject effect. The appropriate `F` tests are the following:

F(shape in subj) = MS(shape) / MS(shape : subj) = 13.895

F(color in subj) = MS(color) / MS(color : subj) = 7.543

What goes inside the `Error()` statement, and the order in which they are arranged, are very important in ensuring correct statistical tests. Suppose you only ask for the errors associated with `subj`, `subj:shape`, and `subj:color`, without the final `subj:shape:color` piece, you use a different formula: `Error(subj/(shape + color))`. You get the following output instead:

```
> summary(aov(rt ~ shape * color + Error(subj/(shape*color)),
  data=Hays.df))

[identical output as before ... snipped ]

Error: Within
            Df    Sum Sq   Mean Sq   F value Pr(>F)
shape:color  1 1.185e-27 1.185e-27 4.272e-28      1
Residuals   11   30.5000    2.7727
```

Note that `Error()` lumps the `shape:color` and `subj:shape:color` sums of squares into a “Within” error stratum. The “Residuals” in the Within stratum is actually the last piece of sum of square in the previous output (`subj:shape:color`).

By using the plus signs instead of the asterisk in the formula, you only get `Error(subj + subj:shape + subj:color)`. The `Error()` function labels the last stratum as “Within”. Everything else remains the same. This difference is important, especially when you have more than two within-subject variables. We will return to this point later.

The `Error()` statement gives us the correct statistical tests. Here we show you two examples of common errors. The first mistakenly computes the repeated measure design as if it was a randomized, between-subject design. The second only separates the subject error stratum. The `aov()` function will not prevent you from fitting these models because they are legitimate. But the tests are not what we want.

```
summary(aov(rt ~ (shape * color) * subj, data=Hays.df))
summary(aov(rt ~ shape * color + Error(subj), data=Hays.df))
```

In a repeated-measure design, there is the between-subject variability (e.g., response time fluctuations due to individual differences) and the within-subject variability (an individual may sometimes respond faster or slower across different questions). `Error()` inside an `aov()` is used to handle these multiple sources of variabilities.

The `Error(subj/(shape * color))` statement says that the shape and color of the buttons are actually *nested* within individual subjects. That is, the changes in response time due to shape and color should be considered within the subjects.

An analogy helps in understanding why repeated measurements are equivalent to designs with variables nested within subjects. In an agricultural experiment Federer (1955, p. 274; cited in Chambers and Hastie, 1993, pp. 157-159) tested the effect of chemical treatments on the rate of germination for seeds. The seeds were planted in different greenhouse flats. Due to differences in light, moisture, and temperature, seeds planted in different flats are likely to grow at a different rate. These differences have nothing to do with the treatment, so Federer separated the effect of different flats in the analysis.

Similarly, buttons on a control panel are given to different subjects. The time it takes each subject to perform the tests is likely to vary considerably. In `aov()` we use the syntax `shape %in% subj` to represent that the effect of the `shape` variable is actually nested within the `subj` variable. Also, we want to separate the effect due to individual subjects. We use `subj / shape` to mean that we want to model the effects of subjects, plus the effect of shape within subjects. R expands the forward slash into `( subj + ( shape %in% subj) )`.

6.8.2 Example 2: Maxwell and Delaney, p. 497

It is the same design as in the previous example, with two within and a subject effect. We repeat the same R syntax, then we include the SAS GLM syntax for the same analysis. Here we have:

one dependent variable: reaction time

two independent variables: visual stimuli are tilted at 0, 4, and 8 degrees; with noise absent or present. Each subject responded to 3 tilt x 2 noise = 6 trials.

The data are entered slightly differently; their format is like what you would usually do with SAS, SPSS, and Systat.

```
MD497.dat <- matrix(c(
420, 420, 480, 480, 600, 780,
420, 480, 480, 360, 480, 600,
480, 480, 540, 660, 780, 780,
420, 540, 540, 480, 780, 900,
540, 660, 540, 480, 660, 720,
360, 420, 360, 360, 480, 540,
480, 480, 600, 540, 720, 840,
480, 600, 660, 540, 720, 900,
540, 600, 540, 480, 720, 780,
480, 420, 540, 540, 660, 780),
ncol = 6, byrow = T)  # byrow=T so the matrix's layout is exactly like this
```

Next we transform the data matrix into a data frame. Note that we use very simple names for the variables. You can actually use very elaborated names for your variables. For example, you can use a combination of upper- and lower-case words to name the `rt` variable `VisualStiReactionTime`. But usually it is a good idea to use simple and mnemonic variable names. That's way we call this data frame MD497.df (page 497 in the Maxwell and Delaney book).

```
MD497.df <- data.frame(
rt    = as.vector(MD497.dat),
subj  = factor(rep(paste("s", 1:10, sep=""), 6)),
deg   = factor(rep(rep(c(0,4,8), c(10, 10, 10)), 2)),
noise = factor(rep(c("no.noise", "noise"), c(30, 30))))
```

Then we test the main effects and the interaction in one aov() model. The syntax is the same as in the Hays example:

```
taov <- aov(rt ~ deg * noise + Error(subj / (deg + noise)), data=MD497.df)
summary(taov)
```

The following SAS GLM does the same univariate analysis, plus some multivariate tests. Maxwell and Delaney summarized the conditions under which one wants to trust the multivariate results more than the univariate results. In SAS, each row contains the data from one subject, across 3 degrees of tilt and two levels of noise. The GLM syntax has a `class` option where the between-subject factors are listed (if any).

```
data rt1;
input deg0NA deg4NA deg8NA deg0NP deg4NP deg8NP;
cards;
420 420 480 480 600 780
420 480 480 360 480 600
480 480 540 660 780 780
420 540 540 480 780 900
540 660 540 480 660 720
360 420 360 360 480 540
480 480 600 540 720 840
480 600 660 540 720 900
540 600 540 480 720 780
480 420 540 540 660 780
;

proc glm data=rt1;
model deg0NA deg4NA deg8NA deg0NP deg4NP deg8NP = ;
repeated noise 2 (0 1), degree 3 (0 4 8) / summary ;
run;
```

6.8.3 Example 3: More Than Two Within-Subject Variables

Earlier we noted that `Error(subj/(shape * color))`, which uses an asterisk to connect `shape` and `color`, produces detailed breakdown of the variance components. The `Error(subj/(shape + color))` statement prints out what you specifically ask for and lumps the remainder into a "Within" stratum. It appears as if you can use the two formulae interchangeably, as long as you are careful in interpreting the almost identical results. This is not true if you have more than two within-subject fixed effects and one random effect associated with the subjects.

The next hypothetical example [8] shows that `aov(a * b * c + Error(subj/(a*b*c)))` gives you all the appropriate statistical tests for interactions in `a:b`, `b:c`, and `a:c`; but `aov(a * b * c + Error(subj/(a+b+c)))` does not. The problem with the latter is because the second part of its syntax, `Error(subj/(a+b+c))`, is inconsistent with the first part, `aov(a * b * c)`. As a result `Error()` lumps everything other than `Error: subj:a`, `Error: subj:b`, and `Error: subj:c` into a common entry of residuals.

For beginners it is helpful to keep the two parts consistent. It is easier to remember too. However, it is very important to know that there are other cases where consistent syntax is not the only rule of thumb. For example, when some of your experimental conditions should be considered "random." In the example of designing a control panel of a medical device, you may wish to generalize the findings to other potential design specifications. Another situation is when the stimuli you present to your subjects are a random sample of numerous other possible stimuli. A good example is the "language-as-fixed-effect fallacy" paper by Clark (1973). Clark showed that many studies in linguistics had a mistake in treating the effect associated with words as a fixed effect. The studies he cited typically used for stimuli a sample of English words (somewhat randomly selected). However, these studies typically analyzed the effect associated with words as a fixed effect (like what we are doing with `shape` and `color`). Many statistically significant findings disappeared when Clark treated them appropriately as random effects.

```
subj <- gl(10, 32, 320) # 10 subjects, each tested 32 times, total length 320
  a  <- gl(2,  16, 320) # first 16 trials with a1 then next 16 with a2
  b  <- gl(2,   8, 320) # first 8 triasl with b1, then next 8 with b2, etc.
  c  <- gl(2,   4, 320)
  x  <- rnorm(320)
  d1 <- data.frame(subj, a, b, c, x)
  d2 <- aggregate(x, list(a = a, b = b, c = c, subj = subj), mean)
  summary(a1 <- aov(x ~ a * b * c + Error(subj/(a*b*c)), d2))
  summary(a2 <- aov(x ~ a * b * c + Error(subj/(a+b+c)), d2))
  summary(a3 <- aov(x ~ a * b * c + Error(subj/(a*b*c)), d1))
```

6.8.4 Example 4: Stevens, 13.2, p.442; a simpler design with only one within variable

one random effect (subject)

1 fixed effect (drug)

```
data <- c(
30,14,24,38,26,
28,18,20,34,28,
16,10,18,20,14,
34,22,30,44,30)
Stv.df <- data.frame(rt=data,
subj = factor(rep(paste("subj", 1:5, sep=""), 4)),
drug = factor(rep(paste("drug", 1:4, sep=""), c(5,5,5,5))))
```

We only have one within-subject variable (`drug`) so that the syntax is simply drug nested within subject.

```
summary(aov(rt ~ drug + Error(subj/drug), data = Stv.df))
```

Again, F is incorrect if the mean square of the `drug` effect is not compared with the correct error mean square (in this case it should be the mean square of `subj:drug`). Without the `Error(subj/drug)` term, R treats it like a completely randomized design.

[8] contributed by Christophe Pallier

```
summary(aov(rt ~ drug, data = Stv.df))
```

6.8.5 Example 5: Stevens pp. 468 – 474 (one between, two within)

The original data came from Elashoff (1981).[9] It is a test of drug treatment effect by one between-subject factor: `group` (two groups of 8 subjects each) and two within-subject factors: `drug` (2 drugs) and `dose` (3 doses).

```
Ela.mat <-matrix(c(
19,22,28,16,26,22,
11,19,30,12,18,28,
20,24,24,24,22,29,
21,25,25,15,10,26,
18,24,29,19,26,28,
17,23,28,15,23,22,
20,23,23,26,21,28,
14,20,29,25,29,29,
16,20,24,30,34,36,
26,26,26,24,30,32,
22,27,23,33,36,45,
16,18,29,27,26,34,
19,21,20,22,22,21,
20,25,25,29,29,33,
21,22,23,27,26,35,
17,20,22,23,26,28), nrow = 16, byrow = T)
```

We first put them in a multivariate format, using the `cbind.data.frame()` function.

```
Ela.mul <- cbind.data.frame(subj=1:16, gp=factor(rep(1:2,rep(8,2))), Ela.mat)
dimnames(Ela.mul)[[2]] <-
c("subj","gp","d11","d12","d13","d21","d22","d23") # d12 = drug 1, dose 2
```

Here is the command for transferring it to the univariate format.

```
Ela.uni <- data.frame(effect = as.vector(Ela.mat),
subj = factor(paste("s", rep(1:16, 6), sep="")),
gp = factor(paste("gp", rep(rep(c(1, 2), c(8,8)), 6), sep="")),
drug = factor(paste("dr", rep(c(1, 2), c(48, 48)), sep="")),
dose=factor(paste("do", rep(rep(c(1,2,3), rep(16, 3)), 2), sep="")),
row.names = NULL)
```

As we discussed earlier, we can use the `tapply()` function to calculate the means across various conditions. We can think of it as using one statement to run the `mean()` function 12 times! The output matrix is very useful for plotting.

```
tapply(Ela.uni$effect, IND = list(Ela.uni$gp, Ela.uni$drug, Ela.uni$dose),
FUN = mean)
```

[9] "Data for the panel session in software for repeated measures analysis of variance." Proceedings of the Statistical Computing Section of the American Statistical Association.

We can also easily custom design a function `se()` to calculate the standard error for the means. R does not have a built-in function for that purpose, but there is really no need because the standard error is just the square root (R has the `sqrt()` function) of the variance (`var()`), divided by the number of observations (`length()`). We can use one line of `tapply()` to get all standard errors. The `se()` makes it easy to find the confidence intervals for those means. Later we will demonstrate how to use the means and standard errors we got from `tapply()` to plot the data.

```
se <- function(x)
      {
        y <- x[!is.na(x)] # remove the missing values
        sqrt(var(as.vector(y))/length(y))
}
```

Without further delay, here is our repeated ANOVA with one between effect and two within effects:

1. Two error strata, one **Within** and one between (subject)

   (a) The following syntax is only accurate for the between-subject, gp effect alone. The tests in the **Within** table are incorrect because they are all lumped together in the entry labeled “Residual SS”.

   ```
   summary(aov(effect ~ gp * drug * dose + Error(subj), data=Ela.uni))
   ```

   (b) The next command performs the correct tests of all effects. We use `Error(subj + subj:drug + subj:dose)` to test `gp`, `drug`, `dose` and their interactions. Worth noting is that R knows that the `gp` effect goes with the subject error stratum, although we did not mention `gp` in the `Error()` statement.

   ```
   summary(aov(effect ~ gp * drug * dose + Error(subj/(dose+drug)), data=Ela.uni))
   ```

2. In R, we not only can use the built-in functions such as `aov()` to do the analyses, we can also take advantage of R’s flexibility and do many analyses by hand. The following examples demonstrate that some of the ANOVA tests we did earlier with the `aov()` function can also be done manually with contrasts.

   (a) We can use the following contrast to test the group effect. On the left hand side of the `aov()` model, we use matrix multiplication (`%*%`) to apply the contrast (`contr`) to each person’s 6 data points. As a result, each person’s 6 data points become one number that is actually the person’s total score summed across all conditions. The matrix multiplication is the same as doing `1 * d11 + 1 * d12 + 1 * d13 + 1 * d21 + 1 * d22 + 1 * d23` for each person.

   Then we use the `aov()` function to compare the total scores across the two groups. We can verify that in this output the F statistic for the `gp` marginal effect is exactly the same as the one in the the previous `aov(...  Error())` output, although the sums of squares are different because the contrast is not scaled to length 1.

   ```
   contr <- matrix(c(
    1,
    1,
    1,
    1,
    1,
    1), ncol = 1)

   taov <- aov(cbind(d11,d12,d13,d21,d22,d23) %*% contr ~ gp, data = Ela.mul)
   summary(taov, intercept = T)
   ```

(b) The following contrast, when combined with the `aov()` function, will test the drug main effect and drug:group interaction. The contrast `c(1, 1, 1, -1, -1, -1)` applies positive 1's to columns 1:3 and negative 1's to columns 4:6. Columns 1:3 contain the data for drug 1 and 4:6 for drug 2, respectively. So the contrast and the matrix multiplication generates a difference score between drugs 1 and 2. When we use `aov()` to compare this difference among two groups, we actually test the `drug:gp` interaction.

```
contr <- matrix(c(
 1,
 1,
 1,
-1,
-1,
-1), ncol = 1)

tmp<-aov(cbind(d11,d12,d13,d21,d22,d23) %*% contr ~ gp, Ela.mul)
summary(tmp,intercept= T)
```

(c) The next contrast, when combined with the `manova()` function in R-1.2.0 or later, tests the dose main effect and the dose:group interaction. The first contrast `c(1, 0, -1, 1, 0, -1)` tests if the difference between dose 1 and dose 3 are statistically significant across groups; and the second contrast `c(0, 1, -1, 0, 1, -1)` tests the difference between dose 2 and dose 3 across two groups. When tested simultaneously with `manova()`, we get

```
contr <- matrix(c(
 1, 0,
 0, 1,
-1,-1,
 1, 0,
 0, 1,
-1,-1), nrow = 6, byrow = T)

tmp <- manova(cbind(d11,d12,d13,d21,d22,d23) %*% contr ~ gp, Ela.mul)
summary(tmp, test="Wilks", intercept = T)
```

(d) Another `manova()` contrast, which tests `drug:dose` interaction and `drug:dose:group` interaction.

```
contr <- matrix(c(
 1,-1,
 0, 2,
-1,-1,
-1, 1,
 0,-2,
 1, 1), nrow = 6, byrow = T)

tmp<-manova(cbind(d11,d12,d13,d21,d22,d23) %*% contr ~ gp, Ela.mul)
summary(tmp, test="Wilks", intercept = T)
```

6.8.6 Graphics with error bars

Next we will demonstrate how to use R's powerful graphics functions to add error bars to a plot. The example uses the Elashoff example discussed earlier. In this example we will briefly show how visual representations compliment the statistical tests. We use R's `jpg()` graphics driver to generate a graph that can be viewed by a web browser. The command syntax may appear intimidating for beginners, but it is worth the increased efficiency in the long run.

You can also use the `postscript()` graphics driver to generate presentation-quality plots. PostScript files can be transformed into PDF format so that nowadays the graphs generated by R can be viewed and printed by virtually anyone.[10]

Typically the graphs are first generated interactively with drivers like `X11()`, then the commands are saved and edited into a script file. A command syntax script eliminates the need to save bulky graphic files.

First we start the graphics driver `jpg()` and name the file where the graph(s) will be saved.

```
attach(Ela.uni)
jpeg(file = "ElasBar.jpg")
```

Then we find the means, the standard errors, and the 95% confidence bounds of the means.

```
tmean <- tapply(effect, IND = list(gp, drug, dose), mean)
tse   <- tapply(effect, IND = list(gp, drug, dose), se)
tbarHeight <- matrix(tmean, ncol=3)
dimnames(tbarHeight) <- list(c("gp1dr1","gp2dr1","gp1dr2","gp2dr2"),
          c("dose1","dose2" ,"dose3"))
tn <- tapply(effect, IND = list(gp, drug, dose), length)
tu <- tmean + qt(.975, df=tn-1) * tse    # upper bound of 95% confidence int.
tl <- tmean + qt(.025, df=tn-1) * tse    # lower bound
tcol <- c("blue", "darkblue", "yellow", "orange")  # color of the bars
```

After all the numbers are computed, we start building the barplot. First we plot the bars without the confidence intervals, axes, labels, and tick marks. Note that the `barplot()` function returns the x-axis values at where the center of the bars are plotted. Later we will use the values in `tbars` to add additional pieces.

```
tbars <- barplot(height=tbarHeight, beside=T, space=c(0, 0.5),
                 ylab="", xlab="", axes=F, names.arg=NULL, ylim=c(-15, 40),
                 col=tcol)
```

Then we add the 95% confidence intervals of the means to the bars.

```
segments(x0=tbars, x1=tbars, y0=tl, y1=tu)
segments(x0=tbars-.1, x1=tbars+0.1, y0=tl, y1=tl)    # lower whiskers
segments(x0=tbars-.1, x1=tbars+0.1, y0=tu, y1=tu)    # upper whiskers
```

The axes labels are added.

```
axis(2, at = seq(0, 40, by=10), labels = rep("", 5), las=1)
tx <- apply(tbars, 2, mean)   # center positions for 3 clusters of bars
```

We plot the horizontal axis manually so that we can ask R to put things at exactly where we want them.

```
segments(x0=0, x1=max(tbars)+1.0, y0=0, y1=0, lty=1, lwd = 2)
text(c("Dose 1", "Dose 2", "Dose 3"), x = tx, y = -1.5, cex =1.5)
mtext(text=seq(0,40,by=10), side = 2, at = seq(0,40,by=10),
      line = 1.5, cex =1.5, las=1)
mtext(text="Drug Effectiveness", side = 2, line = 2.5, at = 20, cex =1.8)
```

[10]One converter is `ps2pdf`, or try GhostScript.

Finally we want to plot the legend of the graph manually. R also has a `legend()` function, although less flexible.

```
tx1 <- c(0, 1, 1, 0)
ty1 <- c(-15, -15, -13, -13)
polygon(x=tx1, y=ty1, col=tcol[1])
polygon(x=tx1, y=ty1 + 2.5, col=tcol[2])  # 2nd, moved 2.5 points up
polygon(x=tx1, y=ty1 + 5, col=tcol[3])    # 3rd
polygon(x=tx1, y=ty1 + 7.5, col=tcol[4])  # last
```

Finally, we add the legend labels for the filled rectangles.

```
text(x = 2.0, y = -14, labels="group 1, drug 1",  cex = 1.5, adj = 0)
text(x = 2.0, y = -11.5, labels="group 2, drug 1",  cex = 1.5, adj = 0)
text(x = 2.0, y = -9, labels="group 1, drug 2",  cex = 1.5, adj = 0)
text(x = 2.0, y = -6.5, labels="group 2, drug 2",  cex = 1.5, adj = 0)
```

The greatest effectiveness is attained by subjects in group 2 when drug 2 is given. This suggests a group by drug interaction, which is confirmed by the `aov()` results outlined earlier. It also indicates an increasing effectiveness from dose 1 to 3, which is also confirmed by the statistics.

6.9 Use `Error()` for repeated-measure ANOVA

In this section we give an intuitive explanation to the use of the `Error()` statement for repeated-measure analysis of variance. These explanations are different than what are typically covered in advanced textbooks. The conventional method focuses on deriving the appropriate error terms for specific statistical tests. We use an intuitive method, which will show that using `Error()` inside an `aov()` function is actually the same as performing t-tests using contrasts.

The conventional explanation is computationally and theoretically equivalent to what we are about to summarize. Detailed theoretical explanations can be found in most advanced textbooks, including the book by Hoaglin, Mosteller, and Tukey (1991). Explanations of the technical details can be found in the book by Chambers and Hastie (1992).

We first review Analysis of Variance using a method commonly seen in most introductory textbooks. This method uses an ANOVA table to describes how much of the total variability is accounted for by all the related variables. An ANOVA table is exactly what `aov()` does for you. We first apply this method to the `Hays.df` data described earlier (but repeated here), then we use the ANOVA table to explain why we must add the `Error()` statement in an `aov()` command in order to get the appropriate significance tests. Finally we draw a connection between `Error()` and specific t-tests tailored for repeated-measure data.

6.9.1 Basic ANOVA table with `aov()`

The `aov()` function generates a basic ANOVA table if `Error()` is not inserted. Applying a simple `aov()` to the `Hays.df` data, you get an ANOVA table like the following:

```
summary(aov(rt ~ subj * color * shape, data = Hays.df))
                 Df    Sum Sq   Mean Sq
subj             11   226.500    20.591
color             1    12.000    12.000
shape             1    12.000    12.000
subj:color       11     9.500     0.864
subj:shape       11    17.500     1.591
color:shape       1 1.493e-27 1.493e-27
subj:color:shape 11    30.500     2.773
```

R analyzes how reaction time differs depending on the subjects, color and the shape of the stimuli. Also, you can have R tell you how they interact with one another. A simple plot of the data may suggest an interaction between color and shape. A `color:shape` interaction occurs if, for example, the color yellow is easier to recognize than red when it comes in a particular shape. The subjects may recognize yellow squares much faster than any other color and shape combinations. Therefore the effect of color on reaction time is not the same for all shapes. We call this an interaction.

The above `aov()` statement divides the total sum of squares in the reaction time into pieces. By looking at the size of the sums of squares (`Sum Sq` in the table), you can get a rough idea that there is a lot of variability among subjects and negligible in the `color:shape` interaction.

So we are pretty sure that the effect of color does not depend on what shape it is. The sum of square for `color:shape` is negligible. Additionally, the `subj` variable has very high variability, although this is not very interesting because this happens all the time. We always know for sure that some subjects respond faster than others.

Obviously we want to know if different colors or shapes make a difference in the response time. One might naturally think that we do not need the `subj` variable in the `aov()` statement. Unfortunately doing so in a repeated design can cause misleading results:

```
summary(aov(rt ~ color * shape, data = Hays.df))
            Df    Sum Sq   Mean Sq   F value Pr(>F)
color        1    12.000    12.000    1.8592 0.1797
shape        1    12.000    12.000    1.8592 0.1797
color:shape  1 1.246e-27 1.246e-27 1.931e-28 1.0000
Residuals   44   284.000     6.455
```

This output can easily deceive you into thinking that there is nothing statistically significant! This is where `Error()` is needed to give you the appropriate test statistics.

6.9.2 Using `Error()` within `aov()`

It is important to remember that `summary()` generates incorrect results if you give it the wrong model. Note that in the statement above the `summary()` function automatically compares each sum of square with the residual sum of square and prints out the `F` statistics accordingly. In addition, because the `aov()` function does not contain the `subj` variable, `aov()` lumps every sum of squares related to the `subj` variable into this big `Residuals` sum of squares. You can verify this by adding up those entries in our basic ANOVA table ($226.5+9.5+17.5+1.49E-27+30=284$).

R does not complain about the above syntax, which assumes that you want to test each effect against the sum of residual errors related to the subjects. This leads to incorrect `F` statistics. The residual error related to the subjects is not the correct error term for all. Next we will explain how to find the correct error terms using the `Error()` statement. We will then use a simple t-test to show you why we want to do that.

6.9.3 The Appropriate Error Terms

In a repeated-measure design like that in Hays, the appropriate error term for the `color` effect is the `subj:color` sum of squares. Also the error term for the other within-subject, `shape` effect is the `subj:shape` sum of squares. The error term for the `color:shape` interaction is then the `subj:color:shape` sum of squares. A general discussion can be found in Hoaglin's book. In the next section we will examine in some detail the test of the `color` effect.

For now we will focus on the appropriate analyses using `Error()`. We must add an `Error(subj/(shape + color))` statement within `aov()`. This repeats an earlier analysis.

```
summary(aov(rt ~ color * shape + Error(subj/(color + shape)), data = Hays.df))
```

```
Error: subj
          Df  Sum Sq Mean Sq F value Pr(>F)
Residuals 11 226.500  20.591

Error: subj:color
          Df  Sum Sq Mean Sq F value   Pr(>F)
color      1 12.0000 12.0000  13.895 0.003338 **
Residuals 11  9.5000  0.8636
---
Signif. codes:  0  ‘***’  0.001  ‘**’  0.01  ‘*’  0.05  ‘.’  0.1  ‘ ’  1

Error: subj:shape
          Df  Sum Sq Mean Sq F value  Pr(>F)
shape      1 12.0000 12.0000  7.5429 0.01901 *
Residuals 11 17.5000  1.5909
---
Signif. codes:  0  ‘***’  0.001  ‘**’  0.01  ‘*’  0.05  ‘.’  0.1  ‘ ’  1

Error: Within
            Df    Sum Sq   Mean Sq   F value Pr(>F)
color:shape  1 1.139e-27 1.139e-27 4.108e-28      1
Residuals   11   30.5000    2.7727
```

As we mentioned before, the `Error(subj/(color + shape))` statement is the short hand for dividing all the residual sums of squares—in this case all subject-related sums of squares—into three error strata. The remaining sums of squares are lumped into a `Within` stratum.

The `Error()` statement says that we want three error terms separated in the ANOVA table: one for `subj`, `subj:color`, and `subj:shape`, respectively. The `summary()` and `aov()` functions are smart enough to do the rest for you. The effects are arranged according to where they belong. In the output the `color` effect is tested against the correct error term `subj:color`, etc. If you add up all the `Residuals` entries in the table, you will find that it is exactly 284, the sum of all subject-related sums of squares.

6.9.4 Sources of the Appropriate Error Terms

In this section we use simple examples of t-tests to demonstrate the need of the appropriate error terms. Rigorous explanations can be found in Edwards (1985) and Hoaglin, Mosteller, and Tukey (1991). We will demonstrate that the appropriate error term for an effect in a repeated ANOVA is exactly identical to the standard error in a t statistic for testing the same effect.

Let's use the data in Hays (1988), which we show here again as `hays.mat` (See earlier example for how to read in the data).

```
hays.mat
        Shape1.Color1 Shape2.Color1 Shape1.Color2 Shape2.Color2
subj 1             49            48            49            45
subj 2             47            46            46            43
subj 3             46            47            47            44
subj 4             47            45            45            45
subj 5             48            49            49            48
subj 6             47            44            45            46
subj 7             41            44            41            40
subj 8             46            45            43            45
subj 9             43            42            44            40
subj 10            47            45            46            45
subj 11            46            45            45            47
subj 12            45            40            40            40
```

In a repeated-measure experiment the four measurements of reaction time are correlated by design because they are from the same subject. A subject who responds quickly in one condition is likely to respond quickly in other conditions as well.

To take into consideration these differences, the comparisons of reaction time should be tested with differences across conditions. When we take the differences, we use each subject as his/her own control. So the difference in reaction time has the subject's baseline speed subtracted out. In the `hays.mat` data we test the `color` effect by a simple t-test comparing the differences between the columns of "Color1" and "Color2."

Using the `t.test()` function, this is done by

```
t.test(x = hays.mat[, 1] + hays.mat[, 2], y = hays.mat[, 3] + hays.mat[, 4],
+ paired = T)

         Paired t-test

data:  hays.mat[, 1] + hays.mat[, 2] and hays.mat[, 3] + hays.mat[, 4]
t = 3.7276, df = 11, p-value = 0.003338
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.819076 3.180924
sample estimates:
mean of the differences
                      2
```

An alternative is to test if a contrast is equal to zero, we talked about this in earlier sections:

```
t.test(hays.mat %*% c(1, 1, -1, -1))

         One Sample t-test

data:  hays.mat %*% c(1, 1, -1, -1)
t = 3.7276, df = 11, p-value = 0.003338
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 0.819076 3.180924
sample estimates:
mean of x
        2
```

This `c(1, 1, -1, -1)` contrast is identical to the first t-test. The matrix multiplication (the `%*%` operand) takes care of the arithmetic. It multiplies the first column by a constant 1, add column 2, then subtract from that columns 3 and 4. This tests the `color` effect. Note that the p-value of this t test is the same as the p-values for the first t test and the earlier `F` test.

It can be proven algebraically that the square of a t-statistic is identical to the `F` test for the same effect. So this fact can be used to double check the results. The square of our t-statistic for `color` is $3.7276^2 = 13.895$, which is identical to the `F` statistic for `color`.

Now we are ready to draw the connection between a t-statistic for the contrast and the F-statistic in an ANOVA table for repeated-measure aov(). The t statistic is a ratio between the effect size to be tested and the standard error of that effect. The larger the ratio, the stronger the effect size. The formula can be described as follows:

$$t = \frac{\bar{x}_1 - \bar{x}_2}{s/\sqrt{n}}, \tag{1}$$

where the numerator is the observed differences and the denominator can be interpreted as the expected differences due to chance. If the actual difference is substantially larger than what you would expect, then you tend to think that the difference is not due to random chance.

Similarly, an F test contrasts the observed variability with the expected variability. In a repeated design we must find an appropriate denominator by adding the the Error() statement inside an aov() function.

The next two commands show that the error sum of squares of the contrast is exactly identical to the Residual sum of squares for the subj:color error stratum.

```
tvec <- hays.mat %*% c(1, 1, -1, -1)/2
sum((tvec - mean(tvec))^2)
[1] 9.5
```

The sum of squares of the contrast is exactly 9.5, identical to the residual sum of squares for the correct F test. The scaling factor $1/2$ is critical because it provides correct scaling for the numbers. By definition a statistical contrast should have a vector length of 1. This is done by dividing each element of the contrast vector by 2, turning it to c(1/2, 1/2, -1/2, -1/2). The scaling does not affect the t-statistics. But it becomes important when we draw a connection between a t-test and an F test.

You get the standard error of the t-statistic if you do the following:

```
sqrt(sum((tvec - mean(tvec))^2 / 11) / 12)
[1] 0.2682717
```

The first division of 11 is for calculating the variance; then you divide the variance by the sample size of 12, take the square root, you have the standard error for the t-test. You can verify it by running se(hays.mat %*% c(1, 1, -1, -1)/2).

6.9.5 Verify the Calculations Manually

All the above calculations by aov() can be verified manually. This section summarizes some of the technical details. This also gives you a flavor of how Analysis Of Variance can be done by matrix algebra. First we re-arrange the raw data into a three-dimensional array. Each element of the array is one data point, and the three dimensions are for the subject, the shape, and the color, respectively.

```
hays.A <- array(hays.dat, dim=c(12, 2, 2))
dimnames(hays.A) <- list(paste("subj", 1:12, sep = ""),
+ c("Shape1", "Shape2"), c("Color1", "Color2"))
```

Because at this point we want to solve for the effect of color, we use the apply() function to average the reaction time over the two shapes.

```
Ss.color <- apply(hays.A, c(1, 3), mean)  # Ss x color: average across shape
```

Next we test a t-test contrast for the color effect, which is the same as `t.test(Ss.color %*% c(1, -1))`. Also note that the square of the t statistic is exactly the same as the `F` test.

```
Contr <- c(1, -1)
Ss.color.Contr <- Ss.color %*% Contr
mean(Ss.color.Contr) / (sqrt(var(Ss.color.Contr) / length(Ss.color.Contr)))
         [,1]
[1,] 3.727564
```

The above t-test compares the mean of the contrast against the standard error of the contrast, which is `sqrt(var(Ss.color.Contr) / length(Ss.color.Contr))`

Now we can verify that the sum of square of the contrast is exactly the same as the error term when we use `aov()` with the `Error(subj:color)` stratum.

```
sum((Ss.color.Contr - mean(Ss.color.Contr))ˆ2)
[1] 9.5
```

6.10 Logistic regression

Multiple regression is usually not appropriate when we regress a dichotomous (yes-no) variable on continuous predictors. The assumptions of normally distributed error are typically violated. So we usually use logistic regression instead. That is, we assume that the probability of a “yes” is certain function of a weighted sum of the predictors, the inverse logit. In other words, if Y is the probability of a “yes” for a given set of predictor values $X_1, X_2, \ldots$, the model says that

$$\log \frac{Y}{1-Y} = b_0 + b_1 X_1 + b_2 X_2 + \ldots + \text{error}$$

The function $\log \frac{Y}{1-Y}$ is the logit function. This is the “link function” in logistic regression. Other link functions are possible in R. If we represent the right side of this equation as X, then the inverse function is

$$Y = \frac{e^X}{1+e^X}$$

In R, when using such transformations as this one, we use `glm` (the generalized linear model) instead of `lm`. We specify the “family” of the model to get the right distribution. Here the family is called `binomial`. Suppose the variable `y` has a value of 0 or 1 for each subject, and the predictors are `x1, x2`, and `x3`. We can thus say

```
summary(glm(y ˜ x1 + x2 + x3, family=binomial))
```

to get the basic analysis, including `p` values for each predictor. Psychologists often like to ask whether the overall regression is significant before looking at the individual predictors. Unfortunately, R does not report the overall significance as part of the `summary` command. To get a test of overall significance, you must compare two models. One way to do this is:

```
glm1 <- glm(y ˜ x1 + x2 + x3, family=binomial)
glm0 <- glm(y ˜ 1, family=binomial)
anova(glm0,glm1,test="Chisq")
```

6.11 Log-linear models

Another use of `glm` is log-linear analysis, where the family is `poisson` rather than `binomial`. Suppose we have a table called `t1.data` like the following (which you could generate with the help of `expand.grid()`). Each row represents the levels of the variables of interest. The last column represents the number of subjects with that combination of levels. The dependent measure is actually expens vs. notexpens. The classification of subjects into these categories depended on whether the subject chose the expensive treatment or not. The variable "cancer" has three values (cervic, colon, breast) corresponding to the three scenarios, so R makes two dummy variables, "cancercervic" and "cancercolon". The variable "cost" has the levels "expens" and "notexp". The variable "real" is "real" vs. "hyp" (hypothetical).

```
cancer cost real count
colon notexp real 37
colon expens real 20
colon notexp hyp  31
colon expens hyp  15
cervic notexp real 27
cervic expens real 28
cervic notexp hyp  52
cervic expens hyp   6
breast notexp real 22
breast expens real 32
breast notexp hyp  25
breast expens hyp  27
```

The following sequence of commands does one analysis:

```
t1 <- read.table("t1.data",header=T)
summary(glm(count ~ cancer + cost + real + cost*real,
 family=poisson(), data=t1)
```

This analysis asks whether "cost" and "real" interact in determining "count," that is, whether the response is affected by "real." See the chapter on Generalized Linear Models in Venables and Ripley (1999) for more discussion of how this works.

6.12 Conjoint analysis

In true conjoint analysis, we present a subject with stimuli made by crossing at least two different variables. For example, in one study, Baron and Andrea Gurmankin presented each subject with 56 items in a random order. The items consisted of each of each of eight medical conditions (ranging from a wart to immediate death) at each of seven probability levels, and the subjects provided a badness rating for each of the 56 items. The assumption of conjoint analysis is that the subject behaves as if he represents the disutility of each condition as a number, and the probability as another number, adds the two numbers, and transforms the result monotonically into the response scale provided. The representation of probability is a monotonic function of the given probability.

When we analyze the data, we try to recover the three transformations so as to get the best fit assuming that this model is true. The three transformations are the assignment of numbers to the probabilities, the assignment of numbers to the conditions, and the function relating the result to the response scale. (In this case, if the subject followed expected-utility theory, the probability would be transformed logarithmically, so that the additive representation corresponded to multiplication.)

R does not have a conjoint analysis package as such. But the Acepack package contains a function called `ace()`, for "alternating conditional expectations," which does essentially what we want. It maximizes the variance in the dependent variable (the response) that is explained by the predictors (probability and condition), by using an iterative process. Here is an example in which the response is called `bad`, which is a matrix in which the rows are subjects, and within each row the probabilities are in groups of 8, with the conditions repeated in each group.

```
probs <- rep(1:7,rep(8,7))
conds <- gl(8,1,56)
cnames <- c("wart","toe","deaf1","leg1","leg2","blind","bbdd","death")
pnames <- c(".001",".0032",".01",".032",".1",".32","1")

c.wt.ace <- matrix(0,ns,8) # resulting numbers for conditions
p.wt.ace <- matrix(0,ns,7) # resulting transformed probabilities
bad.ace <- matrix(0,ns,56) # transformed responses

for (i in 1:ns) # fit the model for each subject
  {av1 <- ace(cbind(probs,conds),bad[i,],lin=1)
   bad.ace[i,] <- av1$ty
   p.wt.ace[i,] <- av1$tx[8*0:6+1,1]
   c.wt.ace[i,] <- av1$tx[1:8,2]
}
```

In the end, the matrices `p.wt.ace, c.wt.ace` and `bad.ace` should have the transformed numbers, one row per subject.

6.13 Imputation of missing data

Schafer and Graham (2002) provide a good review of methods for dealing with missing data. R provides many of the methods that they discuss. One method is multiple imputation, which is found in the Hmisc package. Each missing datum is inferred repeated from different samples of other variables, and the repeated inferences are used to determine the error. It turns out that this method works best with the `ols()` function from the Design package rather than with (the otherwise equivalent) `lm()` function. Here is an example, using the data set `t1`.

```
library(Hmisc)
f <- aregImpute(~v1+v2+v3+v4, n.impute=20,
     fweighted=.2, defaultLinear=T, data=t1)
library(Design)
fmp <- fit.mult.impute(v1~v2+v3, ols, f, data=t1)
summary(fmp)
```

The first command (`f`) imputes missing values in all four variables, using the other three for each variable. The second command (`fmp`) estimates a regression model in which `v1` is predicted from two of the remaining variables. A variable can be used (and should be used, if it is useful) for imputation even when it is not used in the model.

7 References

Chambers, J. M., & Hastie, T. J. (1992). *Statistical models in S.* Pacific Grove, CA: Wadsworth & Brooks Cole Advanced Books and Software.

Clark, H. H. (1973). The language-as-fixed-effect fallacy: A critique of language statistics in psychological research. *Journal of Verbal Learning & Verbal Behavior, 12,* 335–359.

Hays, W. L. (1988, 4th ed.) *Statistics.* New York: Holt, Rinehart and Winston.

Hoaglin, D. C., Mosteller, F., & Tukey, J. W. (Eds.) (1983). *Understanding robust and exploratory data analysis.* New York: Wiley.

Lord, F. M., & Novick, M. R. (1968). *Statistical theories of mental test scores*. Reading, MA: Addison-Wesley.

Maxwell, S. E. & Delaney, H. D. (1990) *Designing Experiments and Analyzing Data: A model comparison perspective.* Pacific Grove, CA: Brooks/Cole.

Schafer, J. L., & Graham, J. W. (2002). Missing data: Our view of the state of the art. `Psychological Methods, 7`, 147–177.

Stevens, J. (1992, 2nd ed) *Applied Multivariate Statistics for the Social Sciences.* Hillsdale, NJ: Erlbaum.

Venables, W. N., & Ripley, B. D. (1999). *Modern applied statistics with S–PLUS* (3rd Ed.). New York: Springer.