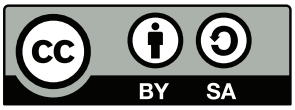
R for Experiment Psychologists

Andy J. Wills

March 20, 2015



Chapter 1

Data input and output

This chapter shows how to get data in and out of R. For reproducibility and flexibility, it is important to start with the most detailed set of raw data you have, and then aggregate that data into shorter summaries for particular analyses. In lab-based testing, you will normally have access to the stimulus, response and reaction time on every trial for every participant. For example, E-prime EDAT files provide this information. Make that your data set, and then aggregate from there.

Most commands in `R` expect your data to be in `long` format. This is a format where every data point has its own row, unlike `wide` format used in SPSS, where every participant has their own row. Long format is clearer and easier to use for analysis in cases where you have more than one within-subject variable.

```
  stim blk subj       acc
1   AB   1    2 0.6666667
2   AC   1    2 0.6666667
3   FD   1    2 0.3333333
4   GE   1    2 0.0000000
5   AB   2    2 0.8333333
6   AC   2    2 0.3333333
7   ...
```

Input from text files

Text files typically use a tab character to separate columns; CSV files are text files that use a comma to separate columns. Text or CSV files are a good choice for archival research data as they are simple, non-proprietary formats. Either can easily be read using the `read.table` function:

```
trstim <- read.table("output.txt",stringsAsFactors = FALSE, sep = "\t",
        header=FALSE,skip=135,nrows=60)
```

The first argument of `read.table` is the name of the file you wish to read. `stringsAsFactors` I typically set to FALSE in order to prevent R automatically recoding any non-numeric columns as factors, but it's up to you. `sep` is the character that separates the columns, the above example specifies a tab. `header` states whether the first line read gives the column labels. `skip` allows you to ignore the first N lines of the file, and `nrows` states how many rows you want to read after the first one. These last two commands are very handy for reading files where the format is not constant (i.e. different columns in different parts of the file) – you can read the file in different chunks.

If the file format is of variable length, you may need to write code to cope with that; knowing the number of rows in the file is then very useful:

```
flines <- length(readLines('data.txt'))
```

Input from other formats

If possible, avoid reading in data from SPSS or Excel files; they are complex proprietary formats whose format can change at the whim of the companies selling this format. Instead, export or request the data in a standard open format, such as a tab-delimited text file or a CSV file, and use the instructions above. Writing R packages to read these closed formats is difficult, time-consuming, and could largely be avoided if researchers provided their data in standard open cross-platform formats. If it is not possible to obtain a text file, the following may work:

SPSS

```
library(memisc)
cdta<- as.data.frame(as.data.set(spss.system.file('filename.sav')))
```

Excel

```
library(gdata)
ref2014 <- read.xls('REF2014 Results.xlsx', pattern = "Institution name")
```

The `pattern` option allows you to skip the first rows of a file by specifying a string on the first line you wish to read. `read.xls` is fairly reliable but a bit slow.

Removing and renaming columns

Some raw data files contain much more information than you want, and have sub-optimal column headings. You can select only the columns you need (the others are deleted, although of course still remain in the file you loaded). You can then rename columns:

```
alleg <- subset(alleg,select=c('Trial','Trial Type','Stim 1','Stim 2','Response'))
colnames(alleg) <- c('trial','stimtype','stim1','stim2','resp')
```

Dealing with header rows

Some raw data files give certain information that is true for all trials with a file (e.g. subject number) just once. Long format requires this information is replicated for each trial. This is easily achieved using cbind:

```
subj <- 1
alleg <- cbind(subj,alleg)
```

You can use a similar method to add columns that do vary but which were not present in the raw data file. For example, a file might use a non-standard row to indicate a block break. This is not long format, so we remove those rows, and then add a column indicating block number:

```
block <- rep(1:6,times=1,each=36)
alleg <- cbind(block,alleg)
```

Combining files

Raw data often comes in the form of a separate file for each participant. This is non-optimal for dissemination and analysis; ideally we want data for the entire experiment in one place. You can combine data files from multiple participants automatically, as long as the file names are rational. For example, where files are `data1.txt, data2.txt` etc:

```
big.alleg <- NULL
for(subjno in 1:2) {
fnam <- paste0('data',subjno,'.txt')
# Insert code to read data from file into 'alleg' here.
big.alleg <- rbind(big.alleg,alleg)
}
```

Converting to long format

Some packages, like SPSS, use a `wide` data format, and it can be a pain converting from one to the other by hand. Fortunately, the `melt` command in the `reshape2` package can do the conversion automatically. For example, with a data set like this:

```
head(dtaw)
  Participant           LearningType   Category Training1 Test1 Training2
1           1 Observational Learning Rule-based  98.33000 58.33  95.00000
2           2        Trial-and-error Rule-based  78.33333 55.00  73.33333
3           3 Observational Learning Rule-based  66.67000 41.67  83.33000
4           7        Trial-and-error Rule-based  58.33000 58.33  58.33000
5           9 Observational Learning Rule-based  98.33000 75.00 100.00000
6          10        Trial-and-error Rule-based  71.67000 71.67  78.33000
     Test2 Training3 Test3 Training4    Test4 Training5    Test5
1 50.00000  91.67000 66.67  90.00000 61.67000     91.67 56.67000
2 73.33333  83.33333 80.00  86.66667 88.33333     85.00 73.33333
3 80.00000  91.67000 56.67  78.33000 45.00000     85.00 60.00000
4 66.67000  76.67000 75.00  65.00000 73.33000     80.00 61.67000
5 81.67000 100.00000 76.67 100.00000 76.67000    100.00 80.00000
6 73.33000  68.33000 90.00  80.00000 95.00000     76.67 75.00000
```

The code to turn it into long format is fairly easy:

```
dtal <- melt(dtaw,id.vars=c('Participant','LearningType','Category')
             ,measure.vars=c('Training1','Test1','Training2','Test2'
             ,'Training3','Test3','Training4','Test4','Training5','Test5')
             ,value.name='acc'
             ,variable.name='block')

head(dtal)
  Participant           LearningType   Category     block      acc
1           1 Observational Learning Rule-based Training1 98.33000
2           2        Trial-and-error Rule-based Training1 78.33333
3           3 Observational Learning Rule-based Training1 66.67000
4           7        Trial-and-error Rule-based Training1 58.33000
5           9 Observational Learning Rule-based Training1 98.33000
6          10        Trial-and-error Rule-based Training1 71.67000
```

Data output

If you have to do substantial pre-processing of your raw data files to get them into a standard long format, you may wish to save out the pre-processed files as text files. This means you don't have

to keep on repeating the preprocessing, and also gives you more comprehensible files to share with the community:

```
write.table(dta,file="ply50data.txt",sep="\t",row.names=FALSE)
```

Chapter 2

Preprocessing

This chapter shows how to prepare your trial-level raw data for analysis.

Ordering

The order of the rows in your data file is irrelevant in long data format, but ordering can sometimes by helpful (e.g. if can improve readability). In the example below, a data frame is ordered by the variable presorder and then by the variable assoc.

```
ratesl <- ratesl[order(ratesl$presorder,ratesl$assoc),]
```

Subsetting

With all your raw data in one data frame, you will probably need to extract sections of that data to perform the analyses you are interested in. The base function `subset` is useful in this regard. In the example below, blocks 1 and 20 are extracted from `p1` and put into `p1fl`.

```
p1fl <- subset(p1,blk == 1 | blk == 20)
```

Aggregating data sets

Aggregation involves averaging over one or more variables in your raw data file to produce the level of summary required for your analysis. One useful function for aggregation is `aggregate` in the `stats` package. In the example below, the accuracy DV in raw data set p1 is averaged over trials, leaving stimulus, block and subject as factors. This is usually the least-aggregated data set one needs to inferential statistics (e.g. ANOVA). The second line names the columns of the aggregated file, for clarity in later analyses.

```
p1ag <- aggregate(p1$acc,list(p1$stim,p1$blk,p1$subj),mean)
colnames(p1ag) <- c('stim','blk','subj','acc')
```

Multiple factors from a single trial type variable

In a multi-factor design, most analyses will require each factor to be defined in its own column. In contrast, many raw data files contain a single column that defines group. So, for example, if you have high and low frequency abstract and concrete words, the raw data file may contain a column with contents such as 'high-abstract', which needs to be turned into two columns, one specifying frequency, another concreteness.

An easy way to convert is as follows:

```
p1anal$freq[p1anal$stim=='AB'] <- 'common'
p1anal$freq[p1anal$stim=='AC'] <- 'rare'
p1anal$freq[p1anal$stim=='FD'] <- 'common'
p1anal$freq[p1anal$stim=='GE'] <- 'rare'

p1anal$cond[p1anal$stim=='AB'] <- 'expt'
p1anal$cond[p1anal$stim=='AC'] <- 'expt'
p1anal$cond[p1anal$stim=='FD'] <- 'ctrl'
p1anal$cond[p1anal$stim=='GE'] <- 'ctrl'
p1anal <- subset(p1anal, select = c(subj,freq,cond,acc))
```

In this particular example, there are four stimulus codes (AB,AC,FD,GE), which are being converted into two factors (freq: common or rare and cond: expt or ctrl). The last line reorders the rows and excludes the now-redundant 'stim' column.

Recoding variables

Sometimes, your raw data is entirely numeric, and would be more readable if you used characters instead. This is one of various situations were you want to recode the entries in your data table. It can be done one entry a time, as in the above section, but if there are a lot of values to change this is tedious, error prone and hard to read. In these cases, use the `match` function instead. In the example below, stimulus codes 1 to 24 in a data file are replaced with letter codes that are more interpretable (if you know the experimental design):

```
letcodes2 <- factor(c('A','B','AB','C','D','CD','E','F','EF','G','H','GH',
                      'I','J','KL','M','N','OP',
                      'IJ','K','L','MN','O','P'))
data.rats$stimtype <- letcodes2[ match(data.rats$stimtype, 1:24)]
```

Converting from long to wide format

Some functions, particularly graph-drawing functions, require data in wide format. The `reshape` command in the `stats` package can do the conversion automatically. In the example below, `timevar` provides the column names to be used in wide format for each row of the data frame, whilst `idvar` sets the variables that should have their own lines. Theoretically, `reshape` can take more than one variable (e.g. stim and blk), but I've found it hard to make this work well. I recommend using a single `timevar` variable to specify multiple within-subject DVs.

```
p1anal$wcode <- c('AB.1', 'AC.1', 'FD.1', 'GE.1', 'AB.2', 'AC.2'...)

reshape(p1anal,direction="wide",timevar="wcode",idvar="subj")

subj AB.1      AC.1      FD.1      GE.1      ...
2    0.9083333 0.2666667 0.8250000 0.4500000 ...
3    0.8750000 0.6333333 0.9583333 0.8666667 ...
4    0.8750000 0.7500000 0.9333333 0.8166667 ...
...
```

Chapter 3

Descriptive statistics

Most of the commands in this section require your data to be in `long` format - see the pre-processing chapter.

Basic descriptives

The `describeBy` command in the `psych` package provides most standard descriptive statistics. In the example below, accuracy of response across different stimulus types is calculated from the long-format raw data file (the `head` command shows the first n lines of a data file). The option `mat=TRUE` sets the output type to a matrix, which is more compact than the alternative `mat=FALSE`.

`describeBy` produces 12 different summary statistics(n, mean, sd, median, trimmed, mad, min, max, range, skew, kurtosis, se), so we put the output of `describeBy` into variable `desc` and then use `subset` to pick the columns that are most commonly needed.

```
> head(p2anal, n=13)
   stim subj       acc
1     A    2 0.7500000
2    AB    2 0.8958333
3    AC    2 0.3750000
4     B    2 0.9583333
5    BC    2 0.3750000
6     C    2 0.3333333
7     D    2 0.9166667
8    DE    2 0.9166667
9     E    2 0.5833333
10   FD    2 1.0000000
11   GE    2 0.7083333
12    A    3 0.8750000
13   AB    3 0.9166667

> desc <- describeBy(p2anal$acc,p2anal$stim,mat=TRUE)
> ?subset
> subset(desc,select=c(group1,n,mean,sd))
    group1  n      mean         sd
11       A 16 0.6848958 0.15363553
12      AB 16 0.9010417 0.09176131
13      AC 16 0.7447917 0.20909583
14       B 16 0.8776042 0.09885777
15      BC 16 0.6406250 0.25132749
```

```
16       C 16 0.6666667 0.22922979
17       D 16 0.8645833 0.15701203
18      DE 16 0.9479167 0.06180165
19       E 16 0.5559896 0.24232654
110     FD 16 0.9596354 0.05457578
111     GE 16 0.8125000 0.17873009
```

Chapter 4

Publication-quality graphs

R, like SPSS, has many functions that allow you to produce quick graphs for data visualization. However, R also allows you to produce publication-quality graphs in a journal-friendly format. Producing such graphs is topic of this chapter. Most of the examples in this section require your data to be in `wide` format, and already aggregated (i.e. one number in the matrix for each plot point, not raw data) - see the pre-processing chapter for hints on how to do this.

By far the best way to produce journal-friendly publication-quality graphs is to produce a separate PDF file for each Figure (traditionally, EPS format was used, but PDF is a newer format that is now widely accepted). The three main advantages of PDFs, relative to image formats (e.g. TIFF, PNG), are:

- defined physical size - most journals have specific sizes they want figures to be in the published article. For example Elsevier want their figures to be either 90mm wide (single-column figures) or 190mm wide (full page width figures). It is much easier to get your figure looking great in the final article if you design it at the intended physical size.
- excellent quality - PDFs can contain vector-based graphics, which do not pixellate however much they are enlarged.
- small file size - vector-based graphics result in much smaller files (e.g. 7K in PDF versus 7MB as an uncompressed TIFF).

The way to produce a graph as a PDF file is shown in the following example. The first command, `pdf` creates a PDF file with the filename specified by `file`. The width and height of the figure, in inches, are given by the corresponding terms. `pointsize` gives the default size of text, in points. `paper` defines the paper size; A4 paper in this example.

The second command, `myplot(z)` is a user-defined function that defines what your plot will look like. It makes sense to use a function for this, because you can quickly prototype the figure in the RStudio Plots window by just calling the function from the console before you start producing and tweaking PDFs. Don't forget though that the Plot window produces only an *approximation* to the final figure. In particular, the Plot window graphic can look a bit rough and ragged. Also, if the aspect ratio of your Plot window (height:width) is different to that of your final figure, things can look quite different (adjust the Plot window size so that the aspect ratio is about right).

The final command, `dev.off()` completes the PDF file.

```
pdf(file='trainfig.pdf'
, width= 3.5
, height = 3
, pointsize = 10
, title = ''
, paper = 'a4')
myplot(z)
dev.off()
```

Line plot

In this example, we're producing a line graph showing accuracy on each of four stimulus types (AB,AC,FD,GE) for each of 10 training blocks.

The core command is `plot` in the `graphics` package. This command has a very large number of options, so we'll deal with them in categories.

To add further lines to the same plot, we use the `lines` command.

Note that these examples use the option `axes = FALSE`. This allows finer control over the appearance of the axes using the `axis` and `box` commands.

Limits and labels

- `xlab = 'Block'` - Give the label for the x-axis. Similarly, `ylab`
- `ylim = c(0.25,1)` - Set the minimum value on the y axis as 0.3 and the maximum as 1. There is a corresponding command `xlim` which is not used in this example.
- `mgp=c(2,1,0)` - Without going into details the first figure (2) defines how far the axes labels are from the axes. Larger figures means further from the axes.

Plot types

- `type = 'o'` - Lines with plot symbols.
- `type = 'b'` - Lines with plot symbols, alternate form. I think this looks more attractive, but 'o' is more common.
- `type = 'l'` - lines, but no plot symbols.
- `type = 'p'` - Plot symbols, but no lines.

Plot symbols

Another key option is `pch`, which defines which plot symbol to use. Here are the 15 most useful options:

	Open	Filled	Filled+Border
Squares	0	15	22
Circles	1	16	21
Triangle(up)	2	17	24
Triangle(down)	6	-	25
Diamond	5	-	23
Plus	3	-	-
Cross	4	-	-

Line types

You can set various line styles using `lty`. There are five line styles, here are the three most useful:

- `lty = 'solid'`
- `lty = 'dashed'`
- `lty = 'dotted'`

Box

Set the type of box enclosing your graph using the `box` command. The two common options are `box(bty='l')` for just the axes, and `box(bty='o')` for a square box.

Axes

Set how the axes look using the `axis` command. The first parameter defines which axis you are setting (1 = x-axis at bottom, 2 = y-axis at left, ... other positions are possible). Then there are a number of options, the most useful being `at`, which sets tick marks. For example at=c(1,5,9) places numbered ticks at 1, 5 and 9 only.

If you want the tick marks to have different labels, use the `labels` command. In the following example, each tick is given a custom label:

```
axis(1,at=1:10,labels=c('1-2','3-4','5-6','7-8','9-10'
,'11-12','13-14','15-16','17-18','19-20'))
```

Legend

Inclusion of the `legend` command provides a key to the plot symbols. The first option, `'bottomright'` specifies where to place the legend. Other useful options are `bottom, bottomleft, left, topleft, top, topright, right, center`. The next option specifies the labels for the plot points. The third option specifies the plot symbols. `inset` determines how far in from the specified location the legend appears, as a proportion of the size of the plot. `cex` can be used to enlarge or shrink the legend, relative to its standard size of 1.

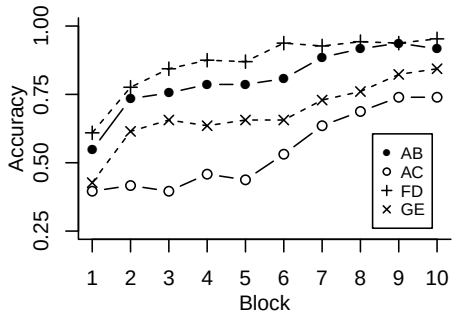
```
legend("bottomright"
, c('AB','AC','FD','GE')
, pch = c(16,1,3,4)
, inset=0.05
, cex=0.75)
```

Worked example

```
# Example data set
> z
              AB        AC        FD        GE
 [1,] 0.5468750 0.3958333 0.6093750 0.4270833
 [2,] 0.7343750 0.4166667 0.7760417 0.6145833
 [3,] 0.7552083 0.3958333 0.8437500 0.6562500
 [4,] 0.7864583 0.4583333 0.8750000 0.6354167
 [5,] 0.7864583 0.4375000 0.8697917 0.6562500
 [6,] 0.8072917 0.5312500 0.9375000 0.6562500
 [7,] 0.8854167 0.6354167 0.9270833 0.7291667
 [8,] 0.9166667 0.6875000 0.9427083 0.7604167
 [9,] 0.9375000 0.7395833 0.9375000 0.8229167
[10,] 0.9166667 0.7395833 0.9531250 0.8437500

# Define the graph we want as a function

trainplot2 <- function(z) {
  plot(
    z[,"AB"]
    , axes = FALSE
    , ylim = c(0.25,1)
    , xlab = 'Block'
    , ylab ='Accuracy'
    , bty = 'l'
```

```
    , mgp = c(2,1,0)
    , type = 'b'
    , pch = 16
    , lty = 'solid'
  )
  box(bty='l')
  axis(1,at=1:10)
  axis(2,at=c(0.25,0.5,0.75,1))
  lines(z[,"AC"],type = 'b', pch=1)
  lines(z[,"FD"],type = 'o', pch=3,lty=2)
  lines(z[,"GE"],type = 'o', pch=4,lty=2)
  legend("bottomright",c('AB','AC','FD','GE'), pch = c(16,1,3,4),inset=0.05,cex=0.75)
}

trainplot2(z) # Check in RStudio

# Produce the graph as a stand-alone PDF.

pdf(file='trainfigf2.pdf'
, width= 3.5
, height = 3
, pointsize = 10
, title = 'Training Phase'
, paper = 'a4')

trainplot2(z)

dev.off()
```

And here's the publication-ready figure:


Accuracy
0.25
0.50
0.75
1.00
1
2
3
4
5
6
7
8
9
10
Block
AB
AC
FD
GE

Adding error bars

Journals often request that graphs include error bars. The least error-prone way to do this is to calculate them from your data and then draw them directly on to the plot. In order to do the drawing, you will need the upper and lower value of each error bar. In the example below, there are 27 plot points at x values 1 to 27. The vector `ab` contains the y-axis value for each of these 27 points. The first two lines of code produce the upper and lower error bars for each point (in this example, simply +1 and -1 from the mean). The `arrows` function then draws the error bars (despite its name, arrows produces standard-looking error bars). The option `length` is the width of the crossbars, in inches. The options `angle` and `code` should be left as they are for standard uses of y-axis error bars.

```
eup <- ab + 1
edn <- ab - 1
arrows(1:27,eup,1:27,edn,angle=90,length=0.1,code=3)
```

Chapter 5

t-tests

Most of the examples in this section require your data to be in `long` format - see the pre-processing chapter.

One-sample t-test

In the example below, a one-sample t-test is conducted to test whether mean accuracy differs significantly from 0.5. Note that, prior to this test, the data set `tdta` was extracted from a larger dataset containing multiple stimulus types. To do this, use the `subset` command, described in the pre-processing chapter.

```
> tdta
    stim subj       acc
5     BC    2 0.3750000
16    BC    3 1.0000000
27    BC    4 0.4583333
38    BC    5 0.6666667
49    BC    6 0.7083333
60    BC    7 0.7916667
71    BC    8 0.7083333
82    BC    9 0.8333333
93    BC   10 0.4166667
104   BC   11 0.4583333
115   BC   12 0.7916667
126   BC   13 0.9583333
137   BC   15 0.2500000
148   BC   16 0.2500000
159   BC   17 1.0000000
170   BC   18 0.5833333
> store <- t.test(tdta$acc,mu=.5)
> store

One Sample t-test

data:  tdta$acc
t = 2.2381, df = 15, p-value = 0.04081
alternative hypothesis: true mean is not equal to 0.5
95 percent confidence interval:
 0.506702 0.774548
sample estimates:
```

```
mean of x
 0.640625
```

One of the great things about R is that you can extract what you want from the output directly. For example, here's how you would get R to report the following test in APA format:

```
> paste('t('
,store$parameter
,') = '
,round(store$statistic,2)
,', p = '
,round(store$p.value,5)
,sep='')

[1] "t(15) = 4.81, p = 0.00023"
```

This is particularly useful in dynamically-produced reports (see later chapter).

Paired and unpaired t-tests

Again, use the `t.test` command. Ensure your grouping variable has exactly two levels (subset your data if necessary, see the preprocessing chapter).

```
t.test(acc ~ stim, data = sleep,paired=FALSE)
```

This will work for both paired and unpaired two-sample t-tests, just set `paired` to TRUE or FALSE as appropriate.

Chapter 6

ANOVA

There are a number of different ways of performing ANOVA in R. A good option for psychologists is the `ez` package, on which this tutorial is based. As is normal for `R`, your data must be in `long` format. Although `ez` will aggregate data sets for you, this comes with various warnings, and I'd advise aggregating the data yourself using the `aggregate` command (see pre-processing chapter) so you can be sure your data for analysis is correct. What you are aiming for is a data set that has one row for each combination of factors for each participant. For example:

```
   subj   freq cond       acc
1     2 common expt 0.9083333
2     2   rare expt 0.2666667
3     2 common ctrl 0.8250000
4     2   rare ctrl 0.4500000
5     3 common expt 0.8750000
...
```

Note that the `ez` package works best with factors that are character (e.g. "expt, ctrl") rather than numeric (e.g. 1, 2). If your factors are truly ordered (e.g. blocks 1...20), then plain ANOVA is the wrong analysis anyway, as ANOVA is a method for unordered factors. Character factors have the advantage of being easier to interpret when you come back to your analysis after a long break. If you really want to use ANOVA despite having a numeric factor, it is possible to force ezANOVA to do this by changing the data type: e.g.

```
p1anal$block <- as.character(p1anal$block)
```

2 x 2 repeated measures

The command `ezANOVA` performs analysis of variance and places a summary table in a data frame of your choice (`p1out3` here). You can then view or manipulate the summary table; to view, just type the name of the data frame, as in the example below.

```
p1out3 <- ezANOVA( data = p1anal
, dv = acc
, wid = subj
, within = c('freq','cond')
, type = 3)

p1out3
```

The variables you need to set are:

`data` - Your data, in the appropriate format (see above).
`dv` - The column containing your dependent variable.
`wid` - The column containing your subject numbers.
`within` - The columns containing your factors. Note the standard R syntax of concatenation.
`type` - The type of sum-of-squares used. This only matters when groups are of unequal sizes. Type 3 is the standard adopted by SPSS, so we'll stick with it here. There are reasons why the SPSS standard may not be ideal, but these are beyond this tutorial.

`ezANOVA` will produce warnings if your factor variables are not actually defined as factors in your data frame. Do not worry about this, ezANOVA will sort this out itself, and your results remain valid. If any other warnings are produced, take heed and don't trust your analysis until you understand what has created the warning.

The output looks like this:

```
> p1out3
$ANOVA
     Effect DFn DFd         F            p p<.05       ges
2      freq   1  15 38.270682 0.0000174159     * 0.4409964
3      cond   1  15 14.562497 0.0016868476     * 0.1303251
4 freq:cond   1  15  2.427196 0.1400909708       0.0222761
```

This is very similar to SPSS output. The first two lines are the main effects of the two within-subjects variables. The third line is the interaction term. DFn, DFd, F and p give you the F-ratio and associated p-value; in APA format, you have $F(1, 15) = 2.43, p = 0.14$, for the third line. The $p < .05$ column should be self-explanatory; it marks with an asterix those tests that reach conventional levels of significance.

GES is an effect size measure, generalized eta squared. Journals are beginning to expect effect size measures. GES is superior to eta-squared as it permits comparisons of effect size across between- and within-subject designs (Bakerman, 2005). By convention, .02 is a small effect, .13 is a medium effect, .26 is a large effect.

Of course, ANOVA is meaningless unless combined with the descriptive statistics. The `ez` package includes simple commands that allow you to show these as a table, or as a graph.

To produce a table of descriptives, just replace 'ezANOVA' with 'ezStats',leaving the rest of the command the same:

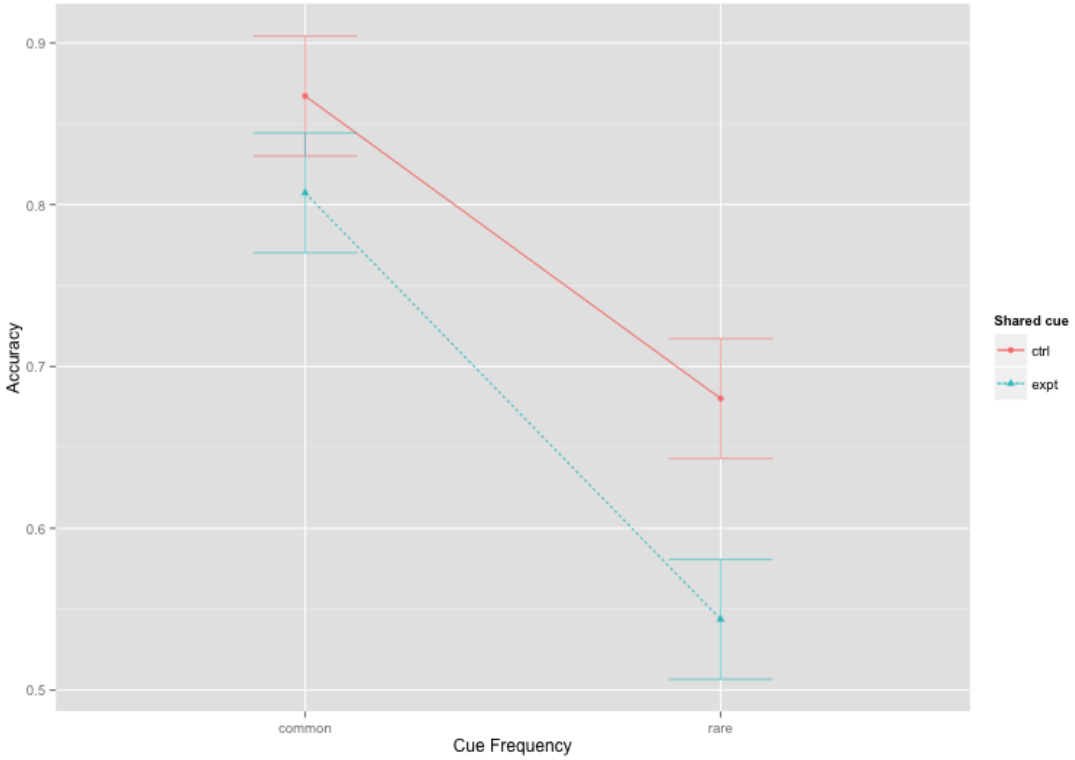
```
ezStats(data = p1anal, dv = acc, wid = subj, within = c('freq','cond'), type = 3)
```

```
    freq cond  N      Mean         SD       FLSD
1 common ctrl 16 0.8671875 0.06676859 0.07406683
2 common expt 16 0.8072917 0.09793587 0.07406683
3   rare ctrl 16 0.6802083 0.16944558 0.07406683
4   rare expt 16 0.5437500 0.16077071 0.07406683
```

The table should be largely self-explanatory, giving sample size, mean, and standard deviation. The last column, FLSD, is Fisher's least-significant difference - useful for post-hoc tests, but that's beyond the scope of this tutorial.

Sometimes, it can help to visualize data. To produce a quick graph, replace 'ezANOVA' with 'ezStats', and add some more information to the command. `x` defines the independent variable that will be shown on the x axis, `split` defines the independent variable that will be illustrated by different plot symbols, the remaining three arguments are optional labels for your graph.

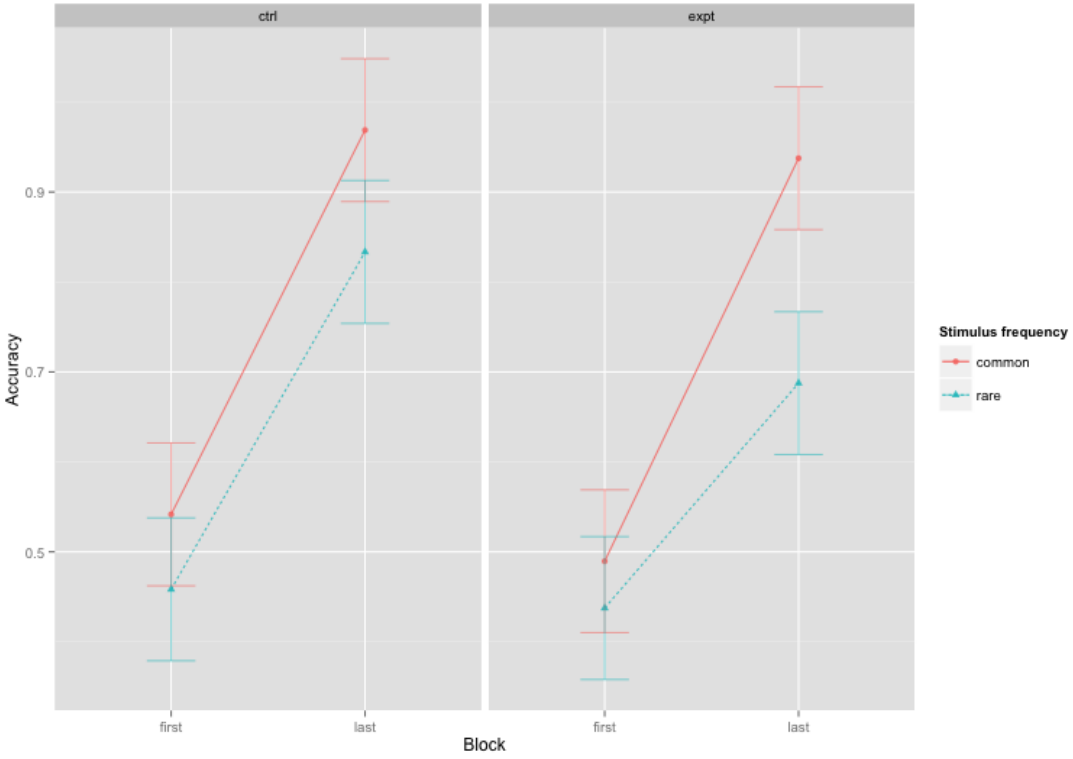
```
ezPlot(data = p1anal
       , dv = acc
       , wid = subj
       , within = c('freq','cond')
       , type = 3
       , x = 'freq'
       , split = 'cond'
       ,x_lab = 'Cue Frequency'
       ,y_lab = 'Accuracy'
       ,split_lab = 'Shared cue')
```



2 x 2 x 2 repeated measures

This is a simple extension of the 2x2 repeated measures technique, simply insert an additional factor. When plotting, a good option is to use the `col` option to produce two-panel graphs.

```
ezPlot(data = p1anal
       , dv = acc
       , wid = subj
       , within = c('freq','cond','blk')
       , type = 3
       , x = 'blk'
       , split = 'freq'
       , col = 'cond'
       ,x_lab = 'Block'
       ,y_lab = 'Accuracy'
       ,split_lab = 'Stimulus frequency')
```



2 x 2 between-subjects design

Between-subjects designs are also easy to implement. In the example below, the two between-subject factors are `cond` and `model`:

```
eza <- ezANOVA( data = bigmfnoid
                , dv = consist
                , wid = subj
                , between = c('cond','model')
                , type = 3)
```

2 x 2 mixed model

It is also easy to combine between-subject and within-subjects factors. In the following example, the repeated measure is `diff` and the between-subjects factor is `cond` :

```
accout <- ezANOVA( data = accag
                   , dv = acc
                   , wid = subj
                   , within = c('diff')
                   , between = cond
                   , type = 3)
```

Chapter 7

Writing concise, efficient, clear analyses

Data

Avoid number codes wherever possible, they make it needlessly hard for others to interpret your data and analyses. R supports `factor`s and `character` variables. So, if your experimental conditions are "load" and "no load", call them that, not 1 and 2. Use R's functions to recode variables if your raw data uses number codes (see Preprocessing chapter).

Code

Avoid loops

If, like me, your first encounter with programming was a language like BASIC or C, using loops is standard practice. Loops are slow in R; your code will run much faster if you can avoid them. Mostly, but not always, one can replace a loop with vectorized code, i.e. coding that operates on the whole array in one step. Vectorized code is also shorter and often easier to understand. Here's some example code, using a loop, to record the subject number of participants who did not meet a learning criterion:

```
excludelist <- NULL
for(i in 1:nrow(tally)) {
  if(tally$tally[i] < 0) {
    excludelist[i] <- tally$subj[i]
  } else {
    excludelist[i] <- NA
  }
}
excludelist <- excludelist[!is.na(excludelist)]
```

The same can be achieved with a single line of vectorized code:

```
excludelist <- tally$subj[tally$tally < 0]
```

At first sight, it's not clear how one extends vectorization where the criterion is a set of values, rather than the single value (0) used in the above example. For example, if I now have a vector of subject numbers for those who failed, how do I remove ALL of those subjects, without resorting to a loop? Answer: use `%in%`:

```
data.train <- data.train[!(data.train$subj %in% fail$subj),]
```

Another type of task where it might not be immediately obvious how to vetorize — where you want to create a smaller number of levels in an analysis by averaging across some of the existing levels. In the following example, we reduce the four stimulus types K, L, O, and P, to two stimulus categories, K/L and O/P by adding a categorizing column to the data frame, and then using `aggregate` to average within a category:

```
klop$klop[klop$stimtype == 'K'] <- 'K/L'
klop$klop[klop$stimtype == 'L'] <- 'K/L'
klop$klop[klop$stimtype == 'O'] <- 'O/P'
klop$klop[klop$stimtype == 'P'] <- 'O/P'
klop <- aggregate(klop$rating,list(klop$subj,klop$group,klop$klop),mean)
colnames(klop) <- c('subj','group','stimtype','rating')
```

Another approach where the calculation is a little more complex is to put each subset into its own data frame, and then use vector arithmetic. In the following example, we wish to calculate $MN - IJ + K/L - O/P$, where e.g. "K/L" is the mean of K and L already calculated above

```
ij <- ijmn[ijmn$stimtype == 'IJ',]
mn <- ijmn[ijmn$stimtype == 'MN',]
knl <- klop[klop$stimtype == 'K/L',]
onp <- klop[klop$stimtype == 'O/P',]
rfi <- mn$rating - ij$rating + knl$rating - onp$rating
```

Make formula format your friend

Where a function has a formula format, use it; it will make your code shorter and easier to understand. For example, the following code for a t-test

```
hinttally <- subset(tally,group==3)
nohinttally <- subset(tally,group==2)
tallyRESULT <- t.test(hinttally$tally,nohinttally$tally,paired=FALSE)
```

can be replaced by the single line

```
tallyRESULT <- t.test(tally~group,data=tally,paired=FALSE)
```

Distinguish between visible and invisible comments

Although I recommend using *RStudio* to develop and test analysis scripts, due to the ease with which scripts can be run interactively in this environment, it's worth bearing in mind how your finalized script will be used by others. In particular, it's good practice to write your analysis script such that running it through the `source` command produces the results of your analyses with sufficient comments to make this meaningful. For outputs to be produced when running a script this way, you need to either `print` them or `cat` them. Reserve hash-character comments for comments that only someone wishing to understand how your script works would need to read.

Specify required packages

Don't assume you (let alone anyone else) will recall which add-on packages are needed for which functions. Explicitly load the packages you need within your script e.g. `library(ez)`.

Chapter 8

Data and analysis archives

This chapter is a brief guide to preparing DAUs (Data and Analysis Units) with R. It specifies the requirements for `willslab.co.uk` archives, but I hope it is also of more general utility.

Archiving raw data

What is raw data? For the purposes of experimental psychology, raw data is the trial-level data for each participant. At a minimum, raw data records the physical stimulus presented on each trial, along with the participant's response, and (if measured) a reaction time. Some procedures will have relevant data at a sub-trial level (e.g. eye movements).

How should raw data be stored? There should be one raw data file per experimental task, containing all trials of all phases for all participants. It should be in LONG format, not WIDE format. It should be in plain-text format (`.TXT or .CSV`), **not** in complex closed formats such as Excel, SPSS, etc. The first row should contain human-readable column headings; all subsequent rows should contain data, without blank lines, blank columns, or blank cells. The same column headings should be used throughout the file. There is normally one "experimental task" per experiment, but some complex experiments use multiple asynchronous tasks, so in those cases it is likely one would need one file per task, rather than one file per experiment.

Archiving analyses

For each experiment, use a single R script to perform all operations necessary to replicate the published analyses. The script should directly read the raw data archive, do the necessary pre-processing, and output the results of the analyses to the screen in a human-readable form and in the order they are reported in the published paper. The R script should be entirely automated (i.e. will work if called through the `source` statement with no further user input). It should contain sufficient comment lines such that an independent researcher can work out what your code is doing. If your script requires packages, make sure the script calls them using `library`.

Archive index

Each experiment has its own web page, with a unique identifier (e.g. `www.willslab.co.uk/plym1`). All files are stored in a single directory with the identifier as its name (e.g. `plym1`). The directory must contain the file `index.html` and this must be an HTML document. The document must specify the following, in order: The unique identifier, the heading "description", a one-paragraph of the contents of the archive, the heading "citation", the way you wish the archive to be cited, the heading "Resources", and (optionally) the heading "Further resources".

Resources section This should contain, at a minimum, a link to the raw data file and the analysis script. The link to the raw data must be accompanied by a **full description of the meaning of each of the column headings of the raw data file**. Ideally, it should also contain the stimulus materials, the experimental program, the instructions, and anything else relevant to the running of the study. These materials must be in open cross-platform formats (e.g. PNG, WAV, TXT, CSV). Do not include anything that requires closed-source software to use (e.g. do not include anything that needs MATLAB, SPSS, E-prime, etc). When designing new studies, try to avoid using closed-source software (some alternatives, MATLAB: R or OCTAVE, SPSS: R, E-prime: PsychoPy).

Further resources This is the place to put things that others might find useful but which are not in open cross-platform formats.