Introduction to R for Psychologists

R is a software environment for statistical analysis: "the lingua franca of computational statistics" De Leeuw & Mair (2007). It is free, open source and cross-platform (running on Mac OS X, Unix, Linux and Windows PCs). It has excellent basic statistics functions powerful and versatile graphics hundreds of user-contributed packages and a large community of users.

R basics

Working with objects:

R is object-oriented and allows you to set up very many different types of objects. This can be a bit overwhelming at first and it helps to start with a few basics: the common objects used for most of your work in R.

Some common R objects:

- *characters* (text strings e.g., 'a' or '1')[1]
- *numbers* (numbers e.g., 2 or 1e+3)
- *vectors* (a one dimensional set of numbers or other elements)
- *data frames* (like vectors organized in columns)
- *matrices* (an *r* by *c* array of numbers or strings)
- *lists* (objects that contain other objects)

To list the objects presently available in R type:

```
> ls()
```

Note that the > character is the command line prompt in R (you don't need to type it). Just type `ls()` and hit return.

Assignment:

Assignment copies the contents of one object to another. This can be done in several ways, but the most common is via `<-` (the assignment operator)
e.g.,

```
> vector1 <- 6
```

[1] These are not the technical terms that R uses. In fact, elements that I have identified as characters or numbers are special types of vectors of length 1 (others include logical values such as TRUE or FALSE).

Using functions:

Most of your work in R will involve calling functions that act on objects to generate output. Standard arithmetic operators (+, -, *, /, ^) can be used (though they are not strictly functions) as well as hundreds of built-in (primitive) and user-defined functions of varying complexity.

Calling a function involves the function name followed by arguments in parentheses:

```
> vector2 <- c(2, 6, 4, 0, 6)
> mean(vector2)
[1] 3.6
```

Functions often take several arguments, some mandatory and some optional. The arguments are matched in order of entry. However, it easy to get the order mixed up and so it makes sense to name extra or optional arguments in complex functions. This will also help you work out what they do. For instance compare these two calls:

```
> mean(vector2, .2)

> mean(vector2, trim=.2)
```

Both should give the output:

```
[1] 4
```

This is because the `mean()` function is a generic trimmed mean function with the default proportion of trimming set to zero in each tail. The call `mean(vector2, trim=.2)` thus returns a trimmed mean that removes 20% of the sample from each tail (i.e., it calculates the mean of 2, 4 and 6).

Help

If you get stuck the R help function will provide basic information about the function. This can be difficult to interpret (as it is quite technical) but it should always include a list of arguments including defaults and details (including names) of objects in the output (often a list if the function is complex one). Most help files also give examples. Statistical functions also often include key references.

```
> help(mean)
> ?mean
```

Further help can be obtained from `example().` You can also see of there are any vignettes in the particular R packages with the `vignette()` function.

```
> example(mean)
```

Exercise 1 (R basics)

Try out the following bits of R code. It should be obvious what they do, but if not just ask or use R help). You may want to make notes.

I. Combine and assign

```
> v1 <- c(2, 5, 6, 8, 3, 4, 7)
> v1
> length(v1)
> is.vector(v1)
> is.numeric(v1)
```

II. Basic statistics

```
> mean(v1)
> sd(v1)
> var(v1)
> ss <- sum((v1 - mean(v1))^2)
> df <- length(v1) - 1
> ss/df
> (ss/df)^0.5
> sqrt(ss/df)
```

III. More statistics (and a few R tricks)

```
> x <- rnorm(n=20, mean=10, sd=2)
> y <- rnorm(20, 5, 2) + 0.5 * x
> cor(x, y)
> cov(x, y)
> cov(x, y) / prod(sd(x), sd(y))
> cor.test(x, y)
> cor.test(x, y, conf.level = .99, method='spearman',
  alternative='two.sided')
> t.test(x)
> t.test(y)
> t.test(x, y)
> t.test(x, y, var.equal=TRUE)
> t.test(x, y, paired=TRUE)
```

Functions like `t.test()` and `cor.test()` are very powerful, but may not work quite as you expect. Also bear in mind that output from these functions can be called from other functions:

```
> test.out <- t.test(x, y, var.equal=FALSE)
> is.list(test.out)
```

```
> test.out$statistic
> test.out$confint
> t.test(x, y, var.equal=F)$conf.int[1:2]
```

IV. Loading data from .csv files or spss .sav files

Note: either change the working directory from the menus, put the data files in the working directory files or specify the full path names. Mac OS X also allows you to drag the full path name into R.

```
> expenses <- read.csv('expenses.csv')
> is.data.frame(expenses)
> summary(expenses)

> library(foreign)
> h05 <- read.spss('Hayden_2005.sav')
> is.data.frame(h05)
> is.list(h05)
> summary(h05)
```

The default input from a .csv file is a data frame (effectively a bundle of vectors/variables joined together as named columns). The default input from an SPSS .sav file is a list (but this can be changed).

```
> h05.df <- read.spss('Hayden_2005.sav', to.data.frame=TRUE)
> is.data.frame(h05.df)
```

You can also write data to an output file. If the data you want to write is a matrix or data frame this is easy to do with `write.csv()`.

```
> write.csv(h05.df, 'temp.csv')
```

V. Working with data frames

Data objects that contain other objects (e.g., vectors, numbers or strings) can be difficult to get used to (but are incredibly powerful tools for complex analyses). To start with you need to be clear how to get at the data inside. The columns in a data frame are named vectors and so you can access these with the `$` symbol:

```
> names(h05.df)

> h05.df$days
> mean(h05.df$days)
> is.vector(h05.df$days)
> days <- h05.df$days
> mean(days)
```

However, each row and column in a data frame or matrix is also indexed (i.e., numbered) and you can get at a particular column, row or cell without its name by specifying the correct combinations of rows or cells.

Just specifying a single index number of a data frame gives the column:

```
> h05.df[2]
> mean(h05.df[2])
```

Specifying both a row and column index gives a cell. The rows are specified first and the columns separated by a comma come second. This gives the first observation in column 2:

```
> h05.df[1,2]
```

Note you sometimes get results that seem odd. Compare:

```
> is.vector(h05.df[2])
> is.vector(h05.df[,2])
```

This discrepancy occurs because a data frame has properties similar to a list and a matrix. It acts like a list of columns and also like a matrix of data. Given a single index number, R will interpret it as requesting the second column of the data frame. Given two index numbers it assumes they refer to different dimensions (rows and columns) and extracts the raw contents. This happens even if the first index is implicit (in this case implicitly selecting all the rows). The named column is in fact a data frame:

```
> is.data.frame(h05.df[2])
```

A neat trick for unwrapping the contents of complex objects is to use two sets of square brackets. This gets at the vector in column 2:

```
> h05.df[[2]]
```

You can also get at the contents of a vector directly or indirectly:

```
> h05.df$days[1]
> h05.df$days[1:9]
```

Other functions such as `subset()` or `stack()` are designed to make working with data frames and similar objects easier.

VI. Working with matrices

A matrix is an incredibly important form of data object for advanced statistical modeling. R has built-in functions for working with matrices (e.g., for matrix algebra). We won't go into these here (but just ask if you want to see some basic matrix algebra functions).

A matrix is also a great way to set out a contingency table. These data are from Haynes et al. (2008):

```
> cells <- c(3677,56,3924,31)
```

```
> cat.names <- list(c("Before", "After"), c("Alive", "Dead"))
> checklist <- matrix(cells, 2, 2, byrow=TRUE, dimnames=cat.names)
> checklist

> chisq.test(checklist)
```

R assumes matrix input to this function is for a chi-square test of independence (and has some complex defaults – for example employing a continuity correction for a 2 × 2 table but not for larger tables).

The matrix command enters data into the cells by column from left to right. The matrix command takes the form:

```
> matrix(data=NA, nrow=1, ncol=1, byrow=FALSE, dimnames=NULL)
```

The values after = are the defaults.

(Note: the value `NA` is a special object in R that is interpreted as a missing observation or argument. It is very useful if you are working with missing data. Here it just means that the default is to set up a matrix containing no data.).

```
> matrix(1:4, 2, 2)
```

specifies the cells as the numbers 1 to 4 and build a matrix with 2 rows and 2 columns, filling up the columns first.

VII. Coercion and creating new objects

Sometimes you need to get data from one format to another. Usually there is a function that will do this for you. Try out the following

```
> data.frame(x,y)

> xy <- cbind(x, y)
> is.matrix(xy)
> xy.df <- as.data.frame(cbind(x, y))
> is.data.frame(xy)
```

As a rule if an `is.object()` style function exists, you can coerce the object into a different type by `as.object()`.

R graphics

R has excellent exploratory graphics and the basic plot functions can easily be adapted to produce publication quality plots. User-contributed package add many different types of specialist plotting functions (and some packages devoted to graphical data analysis).

The main plotting function is called `plot()` and works by calling different functions depending on the object being plotted. Often the function called is `plot.default()` which is a generic scatter plot function.

An important plotting function is the graphical parameter function `par()` – this allows you to tweak all sorts of parameters of a plot or set of plots.

Some plot types:

`barplot()` `stem()` `boxplot()`

Exercise 2 Basic R graphics

I. A scatter plot

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

II. A histogram

```
> hist(h05.df$days)
```

… or a true (density) histogram:

```
> hist(h05.df$days, freq=FALSE)
```

Distribution functions in R

Nicknames	`binom` (binomial)	`chisq` (χ^2)	`f` (*F*)
	`lnorm` (lognormal)	`norm` (normal)	`pois` (Poisson)
	`t` (*t*)	`plogis` (logistic)	`unif` (uniform)

Add a prefix to the nickname to get the distribution function you need.

prefix	*function*
`d`	distribution (*pmf* or *pdf*)
`p`	cumulative probability (*cdf* or inverse quantile)
`q`	quantile
`r`	simulates random draws from the distribution

Examples

`dbinom(0, 5, .5)` is the probability of 0 successes from 10 binomial trials with *P* = .5.

`pnorm(-1.96)` gives the lower tail probability from a standard normal distribution when z = -1.96 (i.e., the proportion of the distribution below -1.96).

`qt(.975, 29)` is the quantile of the *t* distribution that is equal or greater than 97.5% of the distribution.

`rnorm(100)` simulates 100 random draws from a standard normal distribition (because the default is `mean=0` and `sd=1`)

III. Plotting distributions and functions

Probability mass function for a binomial distribution with 5 trials and $P = 1/6$.

```
> par(mar=c(4,4,1,1), cex=1.2)
> n <- 5
> probs <- dbinom(0:n, n, 1/6)
> plot(0:n, probs, pch = NA, xlab = 'Number of successes', ylab =
  'Probability')
> segments(0:n, rep(0,n+1), 0:n, probs, lwd=1.2, col='dark green')
```

Normal distribution with $\mu = 10$ and $s = 2$

```
> curve(dnorm(x, mean=10, sd=2), xlim=c(2,18))
```

Standard normal and *t* distribution with 29 *df*.

```
> curve(dnorm(x), xlim=c(-4,4))
> curve(dt(x, 29), xlim=c(-4,4), lty=3, col='red', add=TRUE)
```

Panel plot:

```
> par(mfrow=c(1,2), pty='s')
> curve(dnorm(x), xlim=c(-4,4))
> curve(dt(x, 1), xlim=c(-4,4), lty=3, col='red', add=TRUE)
> curve(dnorm(x), xlim=c(-4,4))
> curve(dt(x, 29), xlim=c(-4,4), lty=3, col='red', add=TRUE)
```

IV. Superimposing a distribution on a histogram

```
> days.mu <- mean(h05.df$days)
> days.sig <- sd(h05.df$days)

> par(mar=c(4,4,1,1), mfrow+c(1,1), pty= 'm')
> hist(h05.df$days, freq=FALSE, xlim=c(-100, 5000))
> curve(dnorm(x, days.mu, days.sig), lty=3, col='blue', add=TRUE)
```

V. Plotting kernel density estimates

A histogram is a form of density estimation that produces a discrete outcome. Kernel density estimation is a way to get a continuous outcome from density estimation.

```
> density.estimates <- density(h05.df$days)
> plot(density.estimates)
```

VI. Building up a slightly more complex plot (OPTIONAL)

This code plots a serial position curve based on data by Postman & Phillips (1965). In fact it plots three curves (for a no delay, 15 second delay and 30 second delay) of percentage recall of items in a 20-item list.

The idea here is to start with an empty plot and then add points for each condition using the `points()` function. This gives lots of control over the detail of the plot.

```
> pp65 <- read.csv("PP65.csv")

> plot(pp65[,2],pch=NA,ylim=c(0,80),xlab= "Serial position", ylab=
  "Mean percentage recall")

> points(pp65[,2],pch=21,bg='dark red', cex=1.2)
> lines(pp65[,2],lty=3)

> points(pp65[,3],pch=24,bg='dark green', cex=1.2)

> lines(pp65[,3],lty=2)
> points(pp65[,4],pch=22,bg='dark blue', cex=1.2)

> lines(pp65[,4],lty=5)

> legend(3,80, legend=c("No delay","15 second delay","30 second
  delay"),lty=c(3,2,5), pch=c(21,24,22), pt.bg=c('dark red', 'dark
  green', 'dark blue'))
```

Linear models in R

The linear models functions in R are called by a formula.

e.g., $Y = b_0 + b_1X_1 + b_2X_2 + e_i$ could be called by:

```
> lm(Y ~ X1 + X2, data=data.set1)
```

Note that the constant (intercept) is implied by this model (and does not need to be specified directly). The same model could be represented explicitly as:

```
> lm(Y ~ 1 + X1 + X2, data=data.set1)
```

To fit a no-intercept regression you can use either of these methods:

```
> lm(Y ~ 0 + X1 + X2, data=data.set1)
> lm(Y ~ X1 + X2 - 1, data=data.set1)
```

You can also fit an intercept-only regression:

```
> lm(Y ~ 1, data=data.set1)
```

Whatever approach is used, it produces a model object that can be interrogated and called by many other functions:

```
> lm.out <- lm(Y ~ X1 + X2, data = data.set1)

> summary(lm.out)
```

The two main modeling functions are `lm()` and `glm()` – the former is a general linear model (multiple regression) function and the latter is a generalized linear model function (e.g., logistic and Poisson regression).

```
> glm(Y ~ X1 + X2, family='gaussian', data=data.set1)
> summary(glm(Y ~ X1 + X2, family='gaussian', data=data.set1))
```

The examples below use the expenses.csv data set. This was compiled by blogger Mark Thompson[2] and contains a binary DV (named *problem*) for each of the 646 MPs in the UK (during the last parliament). The outcome *problem* is coded 1 if an MP was reported as having a problem with their expenses (i.e., an unproved, but published allegation of expenses abuses) and 0 if not. The question of interest is whether the expenses problems are more likely for MPs in safe seats (with large parliamentary majorities).

Exercise 3 (Linear models)

I. The t test as a linear model

The independent *t* test is a simple linear model:

```
> expenses <- read.csv('expenses.csv')
> t.test(majority ~ problem, data=expenses)
```

Running it as a regression has some advantages:

```
> lin.mod <- lm(majority ~ problem, data=expenses)
> lin.mod
> summary(lin.mod)
```

Now we can get raw residuals, standardized residuals, fitted, values, influence statistics and much more. For instance:

```
> qqnorm(rstudent(lin.mod))
```

This is a normal probability plot of the studentized residuals. You can get a better plot – with confidence bands from the `car` package (Fox, 2002) if it is installed.

[2] http://markreckons.blogspot.com/

```
> install.packages('car')     NOTE: may not work during workshop
> library(car)
> qq.plot(rstudent(lin.mod))
```

Some useful commands for most linear models are:

```
> residuals(lin.mod)
> confint(lin.mod)
> cooks.distance(lin.mod)
> predict(lin.mod)
> AIC(lin.mod)
> AIC(lin.mod, k=log(646))
> anova(lin.mod)
> drop1(lin.mod)
```

II. Linear regression with the Anscombe data

The Anscombe data consist of four sets of *X* and *Y* values that give identical regression lines. First look at the data and then try fitting a linear regression for a couple of data sets and then plotting them.

[`anscombe` is a data set that included in the base installation `datasets` package. See `?anscombe`].

```
> anscombe
> lm(y1 ~ x1, data=anscombe)
> lm(y2 ~ x2, data=anscombe)

> attach(anscombe)

> plot(x1, y1)
> ansc1 <- lm(y1 ~ x1, data=anscombe)
> abline(ansc1)

> plot(x2, y2)
> ansc2 <- lm(y1 ~ x1, data=anscombe)
> abline(ansc2)
```

If you wish, try a quadratic fit (two ways shown here):

```
> quad.fit <- lm(y2 ~ x2 + I(x2^2))
> quad.fit <- lm(y2 ~ poly(x2, degree=2, raw=TRUE))
> plot(x2, y2)
> quad.fit
> curve(-5.9957 + 2.7808*x - 0.1267*x^2, add = TRUE)

> detach(anscombe)
```

Attaching a data frame or list makes the objects inside it available by name. Remember to detach afterwards because otherwise you can easily 'mask' other objects if the names are the same.

III. Comparing linear and generalized linear models (OPTIONAL)

These analyses make *problem* the DV and *majority* the predictor.

```
> lm(problem ~ majority, data = expenses)
```

Because majority goes roughly from 0 to 20000 the coefficient for majority is tiny (and hard to interpret). It helps to rescale majority in units of 10,000:

```
> lm(problem ~ I(majority/10000), data = expenses)
```

The "as is" or `I()` function tells R to treat the operations inside the parentheses as ordinary arithmetic and not part of the formula. The same think could be done in steps:

```
> maj.10k <- expenses$majority/10000
> mod1.lm <- lm(problem ~ maj.10k, data=expenses)

> mod1.lm
> summary(mod1.lm)
```

The fit of the regression line can be plotted like this:

```
> plot(maj.10k, expenses$problem)
> abline(mod1.lm)
```

However, in this case we've got a discrete outcome so this doesn't work too well. We'd be better off fitting a logistic regression model:

```
> mod1.glm <- glm(problem ~ maj.10k, family='binomial',
  data=expenses)
> mod1.glm
> summary(mod1.glm)
```

The significance tests are similar in this case (because the sample is large and mean of the outcome problem is between 0.2 and 0.8, the normal distribution is a good approximation to the binomial distribution for the purposes of testing).

The advantage of the logistic regression is in terms of prediction. To get the predictions just use:

```
> predict(mod1.glm)
```

These are on the log odds (logit) scale used by the logistic regression. To get them in terms of odds you just transform them with the exponential function:

```
> exp(predict(mod1.glm))
```

This is especially useful for the confidence intervals as it gives the 95% CI for the odds ratios of the predictor effects:

```
> exp(confint(mod1.glm))
```

The default in R is to use profile likelihood confidence intervals (which are more accurate than the Wald intervals reported by many packages).[3]

To get them on a probability scale (i.e., to predict the probability of an expenses problem based on the size of the majority) either use the fitted values of the model or calculate them from the predictions,

```
> mod1.glm$fitted
> exp(predict(mod1.glm))/(1+exp(predict(mod1.glm)))
```

Having got the fitted values it is simple plot them:

```
> plot(expenses$majority, mod1.glm$fitted,
  ylab='Probability(Expenses problem)', xlab='Majority')
```

In this case the function is pretty linear within the range of the predictor (and explains why the simple linear regression is adequate).

It is also possible to add approximate (Wald) confidence bands to the plot:

```
> alpha <- .05
> maj <- data.frame(maj.10k = seq(0, 2, 1/10000))
> moe <- predict(mod1.glm, newdata = maj, se.fit=TRUE)[[2]]*qnorm(1-
  alpha/2)
> ub <- predict(mod1.glm, newdata = maj, se.fit=FALSE)+moe
> lb <- predict(mod1.glm, newdata = maj, se.fit=FALSE)-moe
> lines(c(0:20000), exp(lb)/(1+exp(lb)), col = 'gray')
> lines(c(0:20000), exp(ub)/(1+exp(ub)), col = 'gray')
```

This works by creating predictions for a data frame containing 10,000 new values of majority (not in the original data) ranging from low to high. The `predict()` function then generates a standard error for each of the 10,000 predictions and multiplies them by the appropriate quantile. This gives a margin of error for each prediction on the log odds scale. Adding or subtracting the margin of error from the predictions gives the lower and upper bounds on the log odds scale. The lines function then plots the transformed bounds as grey lines.

Confidence bands for a linear regression are not as tricky.

```
> alpha <- .05
> plot(x, y)
> temp.mod <- lm(y~x)
> new.data <- data.frame(x = seq(0, 20, 1/100))
```

[3] This only applies to generalized linear models – for standard linear regression models the usual CIs and tests using the *t* distribution are accurate.

```
> moe <- predict(temp.mod, newdata=new.data,
  se.fit=TRUE)[[2]]*qnorm(1-alpha/2)
> ub <- predict(temp.mod, newdata=new.data, se.fit=FALSE)+moe
> lb <- predict(temp.mod, newdata=new.data, se.fit=FALSE)-moe
> lines(seq(0, 20, 1/100), lb, col = 'gray')
> lines(seq(0, 20, 1/100), ub, col = 'gray')
```

ANOVA, ANCOVA and contrasts

Although ANOVA is a linear model, there are several functions that are important in fitting ANOVA models. It helps to review a few basic ideas:

- ANOVA is a linear model (e.g., a regression) in which all predictors are categorical
- These predictors are termed factors
- ANOVA output can be mimicked in regression software in various ways but, most commonly by fitting a model with effect coding for all categorical predictors
- a classic ANOVA model is orthogonal in its predictors (i.e., the factors are uncorrelated by design) and is balanced (i.e., has equal cell sizes)
- running ANOVA as a regression provides a method to deal with imbalance in factorial designs

R fits a model with categorical predictors and what amounts to dummy coding by default, though you may not notice this (depending on how you request output). It is also trivial to switch to other parameterizations (e.g., the cell means model).

A crucial starting point is to get R to understand that a categorical predictor is a factor. Data loaded into R as a data frame is assumed to be continuous if it contains only numbers. If it contains text should be recognized as a factor by default.

A factor is a class of object in R. So it can also be checked using `is.factor()`, coerced using `as.factor()` and defined using `factor()`. The `gl()` function also generates a factor with a particular pattern of levels. If the factor is ordered, you may want to create it as an ordered factor using `ordered()` or `as.ordered()`, but it safest to start with ordinary factors.

Exercise 4 (ANOVA, ANCOVA and contrasts)

The data set here is from a small study looking at the effect of diagrams on learning from text. There are four conditions: text only, text plus picture, text plus full diagram, and text plus segmented diagram. The DV is description quality (*descript*) measured by the number of propositions recalled by participants when asked to write an explanation of the text. Also recorded is the time (in seconds) taken to read the text.

I. Working with factors

```
> factor1 <- gl(10,4)
> factor 1
```

```
> library(foreign)
> diag.data <- read.spss("diagram.sav", to.data.frame=TRUE)
> summary(diag.data)
> is.factor(diag.data$group)
```

II. One-way ANOVA

```
> diag.fit <- lm(descript ~ group, data=diag.data)
> diag.fit
> summary(diag.fit)
```

This gives a linear model output with the equivalent of dummy coding. To get ANOVA style output you can use the `aov()` function which is a 'wrapper' for the `lm()` function.

```
> aov(diag.fit)
```

A full ANOVA table comes from either of these …

```
> summary(aov(diag.fit))
> anova(diag.fit)
```

III. Descriptives

```
> tapply(diag.data$descript, diag.data$group, mean)
> tapply(diag.data$descript, diag.data$group, sd)
```

A cell means fit of the ANOVA model also gives the means as parameter estimates:

```
> lm(descript ~ 0 + group, data=diag.data)
```

Also see the `describ()` and `describ.by()` functions in the `psych` package.

IV. ANOVAs are still linear models

Standard linear model functions can be used with ANOVA models:

```
> AIC(diag.fit)
> qqnorm(rstudent(diag.fit))
```

V. ANCOVA

ANCOVA is straightforward. However, it is good practice to center the covariate (as an aid to interpretation). This can be done with the `scale()` function or just by subtracting the mean. The former can be used within a formula.

```
> lm(descript ~ group + time, diag.data)
> lm(descript ~ group + scale(time, scale=FALSE), diag.data)
```

```
> diag.ancov <- lm(descript ~ group + scale(time, scale=FALSE),
  diag.data)
```

Use the cell means model as a quick way to get the adjusted means at the average value of the covariate:

```
> lm(descript ~ 0 + group + scale(time, scale=FALSE), diag.data)
```

Note the adjusted means are just regression predictions for the group and so can also be obtained (with standard errors, if required) from the `predict()` function.

VI. Compare models

You can compare linear models using `anova()`:

```
> anova(diag.fit, diag.ancov)
```

Or via AIC:

```
> AIC(diag.fit, diag.anov)
```

Or via BIC:

```
> AIC(diag.fit, diag.anov, k=log(40))
```

You can compare using AIC_C using the `bbmle` package which has a separate `BIC()` function.

The basic ANOVA tests use sequential (Type I) SS. This is sensitive to order of entry and therefore gives unique (Type III) tests only for the term added last to the model. However, Type III tests are not generally a good idea. A better option is hierarchical Type II tests. These are identical to Type III tests in balanced, orthogonal designs. In unbalanced designs the Type II tests generally give more sensitive/powerful tests of interaction effects (as well obeying the marginality principle). Type II tests are implemented by the `drop1()` function.

```
> drop1(diag.ancov, test = 'F')
```

VII. Factorial and repeated measures models.

A factorial design is specified in the formula in this way:

```
> lm(DV ~ factor1 + factor2 + factor1:factor2)
> lm(DV ~ factor1*factor2)
```

`factor1:factor2` is an interaction, while `factor1*factor2` tells R to fit all main effects and interactions between `factor1` and `factor2`.

Repeated measures design are trickier. I would usually use multilevel (linear mixed model) packages because they are more flexible than repeated measures ANOVA or the `ez` package `ezANOVA()` function.

A mixed ANOVA using basic R functions is:

```
> pride.long <- read.csv("pride_long.csv")

> pride.mixed <- aov(accuracy ~ emotion*condition +
  Error(factor(participant)/emotion), pride.long)

> summary(pride.mixed)
```

The tricky bit is defining the error term using `Error()`. This can be avoided by using multilevel (linear mixed model) approaches which can mimic repeated measures ANOVA.

```
> library(nlme)
> ml.mod <- lme(accuracy ~ emotion*condition, random= ~
  1|participant/emotion, pride.long)
> anova(ml.mod)
> summary(ml.mod)
```

This model assumes sphericity. The assumption can be relaxed by specifying an unstructured covariance matrix:

```
> unstruc.mod <- lme(accuracy ~emotion*condition, random=~ 0 +
  emotion|participant, pride.long)
> summary(unstruc.mod)
> anova(unstruc.mod)
```

VIII. Contrasts (OPTIONAL)

Note: this example performs contrasts and multiple comparisons (e.g., post hoc tests) for adjusted means in ANCOVA as well as for a one-way ANOVA.

You can change the default contrasts in R either for all analyses or just for a factor. First check the settings in the R options:.

```
> options('contrasts')
```

Change them to effect coding (note, strictly these are *sum to zero* or *sigma-restricted* contrasts implemented in a general linear model rather than effect coding per se). Re-run the linear regression to see what has changed:

```
> options(contrasts=c('contr.sum', 'contr.poly'))
> lm(descript ~ group + scale(time, scale=F), diag.data)
```

Change the options back afterward (back to the default dummy coding or treatment contrasts):

```
> options(contrasts=c('contr.treatment', 'contr.poly'))
```

Now just change the contrasts for a single factor (inside a data frame):

```
> contrasts(diag.data$group)
> contrasts(diag.data$group) <- c('contr.sum', 'contr.poly')
> contrasts(diag.data$group)
> lm(descript ~ group + scale(time, scale=F), diag.data)
```

You can use this to run a contrast. This example uses one-way ANOVA (but could be adapted for ANCOVA). First set up the coefficients in a matrix. Then change the contrasts for the factor of interest. Then re-run the regression:

```
> cont <- matrix(c(-1,-1,1,1), 4, 1, dimnames=list(NULL,
  "contrast"))
> contrasts(diag.data$group) <- cont
> diag.cont <- lm(descript ~ group, diag.data)
> summary(diag.cont)
```

This contrast compares the diagram with non-diagram groups.

You can also run pairwise (*a priori* and *post hoc*) multiple comparisons. Here the code runs and plots Tukey's HSD and then various adjusted *p* value procedures (Bonferroni, Holm, Hochberg, false discovery rate).

```
> TukeyHSD(aov(diag.fit))
> plot(TukeyHSD(aov(diag.fit)))

> pairwise.t.test(diag.data$descript, diag.data$group,
  p.adjust.method='bonferroni')

> pairwise.t.test(diag.data$descript, diag.data$group,
  p.adjust.method='holm')

> pairwise.t.test(diag.data$descript, diag.data$group,
  p.adjust.method='hochberg')

> pairwise.t.test(diag.data$descript, diag.data$group,
  p.adjust.method='fdr')
```

Much more powerful procedures are available in `multcomp`. This package also allows you to run contrasts and (if required) adjust *p* values for multiple testing (for any regression effect). Here is a quick example of how to run the Schaffer and Westfall procedures (both generally more powerful than standard *post hoc* procedures; the Westfall is also typically more powerful than Hochberg or, in small samples, false discovery rate). It also runs the Tukey HSD test (with or without robust standard errors).

This example does comparisons of adjusted means in ANCOVA (but would also work for ANOVA). The crucial step is to set up a matrix expressing the contrast as a general linear hypothesis. The easiest way to do this is by first fitting a cell means model (because this gets rid of the intercept and allows you to express the contrast as a straightforward function of the cell means; if there are other predictors in the model, set them to zero in the contrast matrix). Note that it is worth checking the orders of the parameters in the model to get the contrast weights correct.

Here is the ANCOVA contrast:

```
> C1 <- matrix(c(-1,-1,1,1,0), 1, 5, dimnames=list("C1", NULL))

> diag.ancov.cmm <- lm(descript ~ 0 + group + scale(time,
  scale=FALSE), diag.data)
> summary(diag.ancov.cmm)

> library(multcomp)
> summary(glht(diag.ancov.cmm, linfct=C1), test=adjusted('none'))
```

For a confidence interval the contrast should be rescaled so that the absolute value of the weights sums to 2. As they sum to 4 here, the linear function divides them by 2 to get the correct scale:

```
> confint(glht(diag.ancov.cmm, linfct=C1/2), test=adjusted('none'))
```

Here are *post hoc* corrections for all pairwise tests of adjusted means using the Westfall procedure. It looks complicated, but most of the work is setting up the dimension names and filling out the matrix. Because accuracy is crucial we suggest that you fill the contrast matrix row-by-row:

```
> cnames <- c('Text', 'Picture', 'Full', 'Segmented', 'Time')
> rnames <- c('S-F', 'S-P', 'S-T', 'F-P','F-T', 'P-T')

> C.mat <- matrix(,6,5, dimnames=list(rnames, cnames))
> C.mat[1,] <- c(0, 0, -1, 1, 0)
> C.mat[2,] <- c(0, -1, 0, 1, 0)
> C.mat[3,] <- c(-1, 0, 0, 1, 0)
> C.mat[4,] <- c(0, -1, 1, 0, 0)
> C.mat[5,] <- c(-1, 0, 1, 0, 0)
> C.mat[6,] <- c(-1, 1, 0, 0, 0)

> diag.ancova <- aov(formula = descript ~ 0 + group + time, data =
diag.data)

> summary(glht(diag.ancova, linfct=C.mat),
test=adjusted("Westfall"))

> summary(glht(diag.ancova, linfct=C.mat), test=adjusted("Shaffer"))
```

Writing functions

Writing a function is not difficult once you have worked how `function()` works. It creates a function object by assignment to a function name, and by defining the call to include the arguments of the function and any defaults (e.g., below the default for `n.2` is to be identical to `n.1`, so `n.2` can be left out of the call if the sample sizes are equal). If an argument is optional it can be specified as `NULL`, but will probably force you to call.

Function for calculating a pooled standard deviation:

```
sd.pool <- function(sd.1, n.1, sd.2, n.2 = n.1) {
num <- (n.1-1)*sd.1^2 + (n.2-1)*sd.2^2
denom <- n.1 + n.2 - 2
output <- sqrt(num/denom)
output
}

> sd.pool(6.1, 20, 5.3, 25)
[1] 5.66743
```

Exercise 5 (Writing a simple function)

Try and write a simple function. Possible suggestions:

(1) calculate a sample mean
(2) plot a normal probability plot for a random sample from a normal distribution of size *n*
(3) plot a scatter plot with regression line for a simple linear regression
(4) center a predictor in a regression
(5) approximate confidence interval for a standardized man difference
(6) approximate confidence interval for a correlation
(7) likelihood ratio for the evidence provided by one linear model relative to another

Simulation and bootstrapping

The main functions to use here are `sample()`, `replicate()` and the various random number distribution functions.

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

Setting `replace=FALSE` to `replace=TRUE` allows sampling with replacement from object `x`.

```
replicate(n, expr, simplify=TRUE)
```

Here the expression can be any function.

Exercise 6 (bootstrapping)

I. Simple bootstrap for a mean

```
> set.seed(112)
> dat1 <- round(rcauchy(20)*10)
> hist(dat1)
> mean(dat1)
> mean(dat1, 20)

> B <- 1000
> sims <- replicate(B, mean(sample(dat1,20,replace=TRUE), 20))
> mean(sims)
```

```
> bias <- mean(sims) - mean(dat1)
> bias

> quantile(sims, c(.025, .975))
> hist(sims)
```

In this case the distribution of the bootstrap samples is fairly symmetrical so the bootstrap should be OK. A better version of the percentile bootstrap is obtained from the `boot` package.

II. Using the boot package

```
> library(boot)

> B <- 1000
> medboot <- boot(dat1, function(x,i) median(x[i]), R = B)
> boot.ci(medboot, conf=.95)

> hist(medboot$t)
```

The bias corrected and accelerated bootstrap is usually better than the percentile bootstrap, though here the percentile bootstrap appears to work better (possibly because a median is being used – see Wilcox, 2003).

Advanced statistical modeling in R

There are dozens of specialist packages for statistical modeling. The best known in psychology include `nlme` and `lme4` for multilevel modeling.

The task view for R collects packages by theme:

http://cran.r-project.org/web/views/

A particularly useful one is the psychometric task view:

http://cran.r-project.org/web/views/Psychometrics.html

I won't say much about this because I'm not a psychometrician, but it is incredibly useful for various sophisticated psychometric models such as extensions of the Rasch model or item-response theory.

There are also good packages for meta-analysis and Bayesian modeling (e.g., `MCMCglmm` for Bayesian multilevel models using MCMC estimation and packages to interface with Matlab, BUGS and so forth).

The example here focuses on meta-analysis using the `metafor` package.

Exercise 7 (advanced modeling: meta-analysis)

I. Meta-analysis of raw mean differences using metafor

The package `metafor` can run analyses using a wide variety of effect sizes. As I'm interested in meta-analysis of unstandardized effect size I'll illustrate how the package works with this example. The analysis described here is a random effects model described by Knapp and Hartung (and also Bond et al.). Baguley (in prep.) describes R code for fixed effects and robust fixed effects models.

These data are for the effect of hypnosis and psychotherapy on weight gain (comparing psychotherapy with psychotherapy plus hypnosis). The data are from Kirsch (and replicate a reanalysis by Bond et al.).

```
> m.e <- c(10.7, 16.24, 10.03, 3.65, 5.73)
> n.e <- c(31, 57, 9, 17, 10)
> m.c <- c(2.87, 6.83, 7.18, 4.65, 7.47)
> n.c <- c(14, 52, 9, 18, 10)

> sd.pooled <- c(7.88, 9.84, 8.67, 6.34, 5.74)
> diff <- m.c - m.e

> se.diffs <- sd.pooled * sqrt(1/n.e + 1/n.c)

> install.packages('metafor')
> library(metafor)

> meta.out <- rma(yi=diff, sei=se.diffs, method = 'HS', knha=TRUE)
> meta.out

> meta.re <- rma(yi=diff, sei=se.diffs, method = 'HS', knha=TRUE)
> meta.re

> funnel(meta.re)
```

The fixed effects analysis here isn't optimal (Bond et al. describe a better model – implemented in R by Baguley). However it is useful here to show a trim and fill sensitivity analysis (for publication bias) and associated trim and fill funnel plot.

```
> meta.fe <- rma(yi=diff, sei=se.diffs, method = 'FE')
> funnel(trimfill(meta.fe))
```

And some more plots:

```
> forest(meta.re)

> qqnorm(meta.re)
```