

Exploratory data analysis and graphics

©2007 Ben Bolker

June 30, 2009

This lab will cover many if not all of the details you actually need to know about R to read in data and produce the figures shown in Chapter 2, and more. The exercises, which will be considerably more difficult than those in Lab 1, will typically involve variations on the figures shown in the text. You will work through reading in the different data sets and constructing the figures shown, or variants of them. It would be even better to work through reading in and making exploratory plots of your own data.

1 Cleaning, reshaping, and reading in data

1.1 Finding files

Note: the next few commands are examples! Nothing real until we get to reading in duncan_10m.csv ...

The basic commands for reading data into R are `read.table` and `read.csv`. If your data are already in the format R likes (long format, space- or comma-separated) then importing your data may be as easy as `read.table("mydata.txt",header=TRUE)`.

However, if R responds to your command with an error like

```
Error in file(file, "r") : unable to open connection
In addition: Warning message: cannot open file 'mydata.txt'
```

it means it can't find your file, probably because it isn't looking in the right place. By default, R's *working directory* is the directory in which the R program starts up, which is (again by default) something like `C:/Program Files/R/R-x.y.z/bin`. (R uses `/` as the [operating-system-independent] separator between directories in a file path.)

To let R know where your data files are located, you have a few choices:

- spell out the *path*, or file location, explicitly. (Use a single forward slash to separate folders (e.g. "c:/Users/bolker/My Documents/R/script.R"): this works on all platforms.)
- use `filename=file.choose()`, which will bring up a dialog box to let you choose the file and set `filename`. (This is only useful on Windows and MacOS).
- Use menu commands to change your working directory to wherever the files are located: **File/Change dir** (Windows) or **Misc/Change Working Directory** (or Apple-D) (Mac).
- Change your working directory to wherever the file(s) are located using the `setwd` (**set working directory**) function, e.g. `setwd("c:/temp")`

Changing your working directory is more efficient in the long run, if you save all the script and data files for a particular project in the same directory and switch to that directory when you start work.

Using the “change directory” commands from the menus is the simplest way to change the working directory for the duration of your R session. While you could just throw everything on your desktop, it’s good to get in the habit of setting up a separate working directory for different projects, so that your data files, metadata files, R script files, and so forth, are all in the same place. If you’re working on a shared machine it may be useful to put your working directory on a USB or network drive.

Depending on how you have gotten your data files onto your system (e.g. by downloading them from the web), Windows will sometimes hide or otherwise screw up the extension of your file (e.g. adding `.txt` to a file called `mydata.dat`). R needs to know the full name of the file, including the extension.

1.2 Checking the number of fields

The next potential problem is that R needs every line of your data file to have the same number of fields (variables) [there are ways to read irregular data into R, but it’s a bit trickier]. You may get an error like:

```
Error in read.table(file = file, header = header,
  sep = sep, quote = quote, :
  more columns than column names
```

or

```
Error in scan(file = file, what = what,
             sep = sep, quote = quote, dec = dec, :
line 1 did not have 5 elements
```

If you need to check on the number of fields that R thinks you have on each line, use

```
> count.fields("myfile.dat", sep = ",")
```

(you can omit the `sep=","` argument if you have whitespace- rather than comma-delimited data). If you are checking a long data file you can try

```
> cf = count.fields("myfile.dat", sep = ",")
> which(cf != cf[1])
```

to get the line numbers with numbers of fields different from the first line.

By default R will try to fill in what it sees as missing fields with NA (“not available”) values; this can be useful but can also hide errors. You can try

```
> mydata <- read.csv("myfile.dat", fill = FALSE)
```

to turn off this behavior; if you don’t have any missing fields at the end of lines in your data this should work.

1.3 Checking data

The quickest way to check that all your variables have been classified correctly:

```
> sapply(data, class)
```

(this applies the `class` command, which identifies the type of a variable, to each column in your data).

Non-numeric missing-variable strings (such as a star, `*`) will also make R misclassify. Use `na.strings` in your `read.table` command:

```
> mydata <- read.table("mydata.dat", na.strings = "*")
```

(you can specify more than one value with (e.g.) `na.strings=c("","***","bad","-9999")`).

The seed removal data were originally stored in two separate Excel files, one for the 10 m transect and one for the 25 m transect: After a couple of preliminary errors I decided to include `na.strings="?"` (to turn question marks into NAs).

```
> dat_10 = read.csv("duncan_10m.csv",na.strings="?",check.names=FALSE)
> dat_25 = read.csv("duncan_25m.csv",na.strings="?",check.names=FALSE)
```

R normally doesn't like column names that begin with numbers and adds an X in front of them, as well as replacing spaces, dashes, and other characters that don't belong in variable names, with dots. I have used `check.names=FALSE` to turn off this behavior, but some things (such as accessing columns with `$`) won't work.

1.4 Accessing data

To access individual variables within your data set use `mydata$varname` or `mydata[,n]` or `mydata[, "varname"]` where `n` is the column number and `varname` is the variable name you want. You can also use `attach(mydata)` to set things up so that you can refer to the variable names alone (e.g. `varname` rather than `mydata$varname`). However, **beware**: if you then modify a variable, you can end up with two copies of it: one (modified) is a local variable called `varname`, the other (original) is a column in the data frame called `varname`: it's probably better not to `attach` a data set until after you've finished cleaning and modifying it. Furthermore, if you have already created a variable called `varname`, R will find it before it finds the version of `varname` that is part of your data set. Attaching multiple copies of a data set is a good way to get confused: try to remember to `detach(mydata)` when you're done.

To access a data set called `dataset` that is built in to R or included in an R package, say

```
> data(dataset)
```

(`data()` by itself will list all available data sets.)

1.5 Packages (reminder)

The `sizeplot` function I used for Figure 2 in the chapter requires an add-on *package* (unfortunately the command for loading a package is `library!`). To use an additional package it must be (i) *installed* on your machine (with `install.packages`) or through the menu system and (ii) *loaded* in your current R session (with `library`).

```
> install.packages("plotrix")
```

```
> library(plotrix)
```

You must both install and load a package before you can use or get help on its functions, although `help.search` will list functions in packages that are installed but not yet loaded. You only have to install the package once on a particular machine, but you have to load the package again in every R session where you want to use it.

For this session, you will need the packages `chron`, `gdata`, `gplots`, `gtools`, `plotrix`, `reshape`, `rgl`, and `scatterplot3d` installed. (If you have the `emdbook` package installed (and loaded using `library(emdbook)`) and have a network connection, you should be able to type `get.emdbook.packages()` to install all of these packages, and a few more, automatically.)

1.6 Checking and cleaning up data

The `summary` and `str` commands are useful for looking at the structure of data sets. They originally showed that I had some extra columns and rows: row 160 of `dat_10`, and columns 40–44 of `dat_25`, were junk. I could have gotten rid of them this way:

```
> dat_10 = dat_10[1:159, ]
> dat_25 = dat_25[, 1:39]
```

(I could also have used *negative* indices to drop specific rows/columns: `dat_10[-160,]` and `dat_25[-(40:44),]` would have the same effect). Instead, I went back and edited the input files.

Exercise 1.1: Try out `sapply(dat_10, class)`, `head`, `summary` and `str` on these data; make sure you understand the results. If you are using Windows, try `View` as well (`View` is likely to crash R in Linux and MacOS with R versions < 2.9.1.)

Note: The data are `integer` rather than the more general `numeric`; this distinction rarely makes much difference.

1.7 Reshaping data

The data are in the “wrong” (wide) format. We reshape them, specifying `id.var=1:2` to preserve the first two columns, station and species, as identifier variables (you will need the `reshape` package installed).

First “melt” the data to long format:

```
> library(reshape)
> dat_10_melt = melt(dat_10, id.var = 1:2)
```

Convert the third column to a date, using `paste` to append 1999 to each date (`sep="-"` separates the two pasted strings with a dash):

```
> date_10 = paste(dat_10_melt[, 3], "1999", sep = "-")
```

Then use `as.Date` to convert the string to a date (`%d` means day, `%b` means three-letter month abbreviation, and `%Y` means four-digit year; check `?strptime` for more date format details).

```
> dat_10_melt[, 3] = as.Date(date_10, format = "%d-%b-%Y")
```

Finally, rename the columns.

```
> names(dat_10_melt) = c("station", "species", "date", "seeds")
```

Do the same for the 25-m transect data:

```
> dat_25_melt = melt(dat_25, id.var = 1:2)
> date_25 = paste(dat_25_melt[, 3], "1999", sep = "-")
> dat_25_melt[, 3] = as.Date(date_25, format = "%d-%b-%Y")
> names(dat_25_melt) = c("station", "species", "date", "seeds")
```

1.8 More on data types

While you can usually get by coding data in not quite the right way — for example, coding dates as numeric values or categorical variables as strings — R tries to “do the right thing” with your data, and it is more likely to do the right thing the more it knows about how your data are structured.

Strings instead of factors Sometimes R’s default of assigning factors is not what you want: if your strings are unique identifiers (e.g. if you have a code for observations that combines the date and location of sampling, and each location combination is only sampled once on a given date) then R’s strategy of coding unique levels as integers and then associating a label with integers will waste space and add confusion. If all of your non-numeric variables should be treated as character strings rather than factors, you can just specify `as.is=TRUE`; if you want specific columns to be left “as is” you can specify them by number or column name. For example, these two commands have the same result:

```
> tmpdata = read.csv("duncan_10m.csv", comment="",
  as.is="Seed.Species")
> tmpdata = read.csv("duncan_10m.csv", comment="", as.is=2)
> sapply(tmpdata, class)[1:2]
```

```
X10m.Station.. Seed.Species
      "integer"   "character"
```

While the first line of the data file uses the name “Seed Species”, R automatically converts the space to a dot to get the column name. Use `c` — e.g. `c("name1", "name2")` or `c(1,3)` — to specify more than one column. You can also use the `colClasses="character"` argument to `read.table` to specify that a particular column should be converted to type `character` —

```
> tmpdata = read.csv("duncan_10m.csv", comment = "", na.strings = "?",
  colClasses = c(rep("character", 2), rep("numeric", 37)))
```

brings in the first two columns as characters and the last 37 as numeric.

To convert factors back to strings *after* you have read them into R, use `as.character`.

```
> tmpdata = read.csv("duncan_10m.csv", comment = "", na.strings = "?")
> tmpdata$Seed.Species = as.character(tmpdata$Seed.Species)
```

Factors instead of numeric values In contrast, sometimes you have numeric labels for data that are really categorical values — for example, in the seed data set the stations are listed by integer codes (often data sets will contain redundant information in them, e.g. both a species name and a species code number). It’s best to specify appropriate data types, so use `colClasses` to force R to treat the data as a factor. For example:

```
> tmpdata = read.csv("duncan_10m.csv", comment = "", na.strings = "?",
  colClasses = c(rep("factor", 2), rep("numeric", 37)))
```

n.b.: by default, R sets the order of the factor levels alphabetically. You can find out the levels and their order in a factor `f` with `levels(f)`. If you want your levels ordered in some other way (e.g. site names in order along some transect), you need to specify this explicitly. Most confusingly, R will sort strings in alphabetic order too, even if they represent numbers. This is OK:

```
> f = factor(1:10)
> levels(f)
```

```
[1] "1" "2" "3" "4" "5" "6" "7" "8" "9" "10"
```

but the following is not, since we explicitly tell R to treat the numbers as characters (this can happen by accident in some contexts):

```
> f = factor(as.character(1:10))
> levels(f)
```

```
[1] "1" "10" "2" "3" "4" "5" "6" "7" "8" "9"
```

In a list of numbers from 1 to 10, “10” comes after “1” but before “2”!

You can fix the levels by using the `levels` argument in `factor` to tell R explicitly what you want it to do, e.g.:

```
> f = factor(as.character(1:10), levels = 1:10)
> x = c("far_north", "north", "middle", "south")
> f = factor(x, levels = c("far_north", "north", "middle", "south"))
```

so that the levels come out ordered geographically rather than alphabetically.

To put factors in the order in which they appear first in the data:

```
> f = factor(f, levels = unique(as.character(f)))
```

Sometimes your data contain a subset of integer values in a range, but you want to make sure the levels of the factor you construct include all of the values in the range, not just the ones in your data. Use `levels` again:

```
> f = factor(c(3, 3, 5, 6, 7, 8, 10), levels = 3:10)
```

Exercise 1.2: Explore the effects of the `levels` command by plotting the factor `f=factor(c(3,3,5,6,7,8,10))` as created with and without intermediate levels (if `f` is a factor, `plot(f)` automatically tabulates and plots a barplot showing the number of occurrences of each level).

Finally, you may want to get rid of levels that were included in a previous factor but are no longer relevant:

```
> f = factor(c("a", "b", "c", "d"))
> f2 = f[1:2]
> levels(f2)
```

```
[1] "a" "b" "c" "d"
```

```
> f2 = factor(as.character(f2))
> levels(f2)
```

```
[1] "a" "b"
```

For more complicated operations with `factor`, use the `recode` function in the `car` package.

Dates and times Dates and times can be tricky in R, but you can (and should) handle your dates as type `Date` within R rather than messing around with Julian days (i.e., days since the beginning of the year) or maintaining separate variables for day/month/year.

You can use `colClasses="Date"` within `read.table` to read in dates directly from a file, but only if your dates are in four-digit-year/month/day (e.g. 2005/08/16 or 2005-08-16) format; otherwise R will either butcher your dates or complain

```
Error in fromchar(x) : character string is not
  in a standard unambiguous format
```

If your dates are in another format in a single column, read them in as character strings (`colClasses="character"` or using `as.is`) and then use `as.Date`, which uses a very flexible `format` argument to convert character formats to dates:

```
> as.Date(c("1jan1960", "2jan1960", "31mar1960", "30jul1960"),
          format="%d%b%Y")

[1] "1960-01-01" "1960-01-02" "1960-03-31" "1960-07-30"

> as.Date(c("02/27/92", "02/27/92", "01/14/92",
          "02/28/92", "02/01/92"),
          format="%m/%d/%y")

[1] "1992-02-27" "1992-02-27" "1992-01-14" "1992-02-28" "1992-02-01"
```

The most useful format codes are `%m` for month number, `%d` for day of month, `%j` for Julian date (day of year), `%y` for two-digit year (dangerous for dates before 1970!) and `%Y` for four-digit year; see `?strftime` for many more details.

If you have your dates as separate (numeric) day, month, and year columns, you actually have to squash them together into a character format (with `paste`, using `sep="/"` to specify that the values should be separated by a slash) and then convert them to dates:

```
> year = c(2004, 2004, 2004, 2005)
> month = c(10, 11, 12, 1)
> day = c(20, 18, 28, 17)
> datestr = paste(year, month, day, sep = "/")
> date = as.Date(datestr)
> date
```

```
[1] "2004-10-20" "2004-11-18" "2004-12-28" "2005-01-17"
```

Although R prints the dates out so they look like a vector of character strings, they are really dates: `class(date)` will give you the answer "Date".

Times work similarly (you have to convert them after you read in your data), but you will need to install and attach the `chron` package to use them.

```
> install.packages(chron)
```

```
> library(chron)
```

Basic time conversion is pretty easy:

```
> timevec1 = c("11:00:00", "11:25:30", "15:30:20")
```

```
> (times1 = times(timevec1))
```

```
[1] 11:00:00 11:25:30 15:30:20
```

(`times1` looks identical to `timevec1`, but is in a different – and more useful – format. For example, differences (`diff`) work as they should. If you translate them to numeric values, they correspond to fractions of a day.

```
> d1 = diff(times1)
```

```
> d1
```

```
[1] 00:25:30 04:04:50
```

```
> as.numeric(d1)
```

```
[1] 0.01770833 0.17002315
```

If you have times without seconds, you have to use `paste` to append `:00` first:

```
> timevec2 = c("11:00", "11:25", "15:30")
```

```
> timevec2 = paste(timevec2, ":00", sep = "")
```

```
> times(timevec2)
```

```
[1] 11:00:00 11:25:00 15:30:00
```

Other traps:

- quotation marks in character variables: if you have character strings in your data set with apostrophes or quotation marks embedded in them, you have to get R to ignore them. I used a data set recently that contained lines like this:

```
Western Canyon|valley|Santa Cruz|313120N|1103145W0'Donnell Canyon
```

I used

```
> data = read.table("datafile", sep = "|", quote = "")
```

to tell R that | was the separator between fields and that it should ignore all apostrophes/single quotations/double quotations in the data set and just read them as part of a string.

- Hash/pound (#) symbols in your data set that are *not* intended to be comments. For example, I recently loaded a data set from Excel where some entries were #VALUE! (Excel worksheet cells involving a division by zero). R interpreted the # as signifying the start of a comment, and all subsequent variables in those rows were lost. (`read.csv` eliminates comment characters by using `comment.char=""` by default, but for `read.table` you will need to specify it explicitly.)

1.9 Augmenting the data

We've finished cleaning up and reformatting the data. Now we would like to calculate some derived quantities: specifically, `tcum` (elapsed time from the first sample), `tint` (time since previous sample), `taken` (number removed since previous sample), and `available` (number available at previous sample).

For each station, we want to calculate the cumulative (elapsed) time for each observation by subtracting the first date from all the dates; the time interval by taking the difference of successive dates (with `diff`) and putting an NA at the beginning; the number of seeds lost by taking the *negative* of the difference of successive numbers of seeds; and the number of seeds available at the previous time by prepending NA and dropping the last element. Then put the new derived variables together with the original data and re-assign it.

We will *write our own function* now, to take advantage of a package called `plyr` (this is an English pun). Later on we will be writing lots of our own functions. Here's how it works:

```
> tmpf = function(x) {
  tcum = as.numeric(x$date - x$date[1])
  tint = as.numeric(c(NA, diff(x$date)))
  taken = c(NA, -diff(x$seeds))
  available = c(NA, x$seeds[-nrow(x)])
  data.frame(x, tcum, tint, taken, available)
}
```

When you define an R function, it doesn't do anything immediately, just creates the function for later use. Any variables that get defined inside the function are purely temporary, and disappear when the function finishes. The function *returns* the last expression (in this case a data frame containing the original variables plus the derived variables).

Now we use the `ddply` function to split the data up by station, run the `tmpf` function on the data from each station, and put it back together. (`plyr` is required by the `reshape` package, so we've already loaded it.)

```
> dat_10 = ddply(dat_10_melt, "station", tmpf)
```

This trick is useful whenever you have individuals or stations that have data recorded only for the first observation of the individual. In some cases you can also do these manipulations by working with the data in wide format.

Do the same for the 25-m data:

```
> dat_25 = ddply(dat_10_melt, "station", tmpf)
```

Create new data frames with an extra column that gives the distance from the forest (`rep` is the R command to **repeat** values); then stick them together.

```
> dat_10 = data.frame(dat_10, dist = rep(10, nrow(dat_10)))
> dat_25 = data.frame(dat_25, dist = rep(25, nrow(dat_25)))
> SeedPred = rbind(dat_10, dat_25)
```

Convert station and distance from numeric to factors:

```
> SeedPred$station = factor(SeedPred$station)
> SeedPred$dist = factor(SeedPred$dist)
```

Reorder columns:

```
> SeedPred = SeedPred[, c("station", "dist", "species", "date",
  "seeds", "tcum", "tint", "taken", "available")]
```

Clean up temporary variables we have created:

```
> rm("d1", "date", "year")
```

2 Exploratory graphics: seed data

2.1 Mean number remaining with time

Attach the seed removal (predation) data:

```
> attach(SeedPred)
```

Using `attach` can make your code easier to read, since you don't have to put `SeedPred$` in front of the column names, but it's important to realize that attaching a data frame makes a local copy of the variables. Changes that you make to these variables are *not* saved in the original data frame, which can be very confusing. Therefore, it's best to use `attach` only after you've finished modifying your data. `attach` can also be confusing if you have columns with the same name in two different attached data frames: use `search` to see where R is looking for variables. It's best to attach just one data frame at a time — and make sure to `detach` it when you finish.

Separate out the 10 m and 25 m transect data from the full seed removal data set:

```
> SeedPred_10 = subset(SeedPred, dist == 10)
> SeedPred_25 = subset(SeedPred, dist == 25)
```

The `tapply` (for **table apply**, pronounced “t apply”) function splits a vector into groups according to the list of factors provided, then *applies* a function (e.g. `mean` or `sd`) to each group. To split the data on numbers of seeds present by `date` and `species` and take the mean (`na.rm=TRUE` says to drop NA values):

```
> s10_means = with(SeedPred_10,
  tapply(seeds, list(date, species), mean, na.rm=TRUE))
> s25_means = with(SeedPred_25,
  tapply(seeds, list(date, species), mean, na.rm=TRUE))
```

`matplot` (“**matrix plot**”) plots the columns of a matrix together against a single x variable. Use it to plot the 10 m data on a log scale (`log="y"`) with both lines and points (`type="b"`), in black (`col=1`), with plotting characters 1 through 8, with solid lines (`lty=1`). Use `matlines` (“**matrix lines**”) to add the 25 m data in gray. (`lines` and `points` are the base graphics commands to add lines and points to an existing graph.)

```
> matplot(s10_means, log = "y", type = "b", col = 1, pch = 1:8,
  lty = 1)
> matlines(s25_means, type = "b", col = "gray", pch = 1:8, lty = 1)
```

2.2 Number taken vs. number available

2.2.1 Jittered plot

Jittered plot:

```
> plot(jitter(available), jitter(taken))
```

2.2.2 Bubble plot

The following graph differs from the figure in Chapter 2, because I don't exclude cases where there are no seeds available. (I use `xlim` and `ylim` to extend the axes slightly.) `scale` and `pow` can be tweaked to change the size and scaling of the symbols.

To plot the numbers in each category, I use `text`, `row` to get row numbers, and `col` to get column numbers; I subtract 1 from the row and column numbers to plot values starting at zero.

I used

```
> t1 = table(available, taken)
```

to cross-tabulate the data, and then used the `text` command to add the numbers to the plot. There's a little bit more trickery involved in putting the numbers in the right place on the plot. `row(x)` gives a matrix with the row numbers corresponding to the elements of `x`; `col(x)` does the same for column numbers. Subtracting 1 (`col(x)-1`) accounts for the fact that columns 1 through 6 of our table refer to 0 through 5 seeds actually taken. When R plots, it simply matches up each of the x values, each of the y values, and each of the text values (which in this case are the numbers in the table) and plots them, even though the numbers are arranged in matrices rather than vectors. I also limit the plotting to positive values (using `[t1>0]`), although this is just cosmetic.

```
> library(plotrix)
> sizeplot(available, taken, scale = 0.5, pow = 0.5, xlim = c(-2,
  6), ylim = c(-2, 5))
> t1 = table(available, taken)
> r = row(t1) - 1
> c = col(t1) - 1
> text(r, c, t1)
```

Or you can use `balloonplot` from the `gplots` package:

```
> library(gplots)
> balloonplot(t1)
```

Finally, the default mosaic plot, either using the default `plot` command on the existing tabulation

```
> plot(t1)
```

or using `mosaicplot` with a formula based on the columns of `SeedPred`:

```
> mosaicplot(~available + taken, data = SeedPred)
```

The command to produce the barplot (Figure 3) was:

```
> barplot(t(log10(t1 + 1)), beside = TRUE, legend = TRUE, xlab = "Available",
          ylab = "log10(1+# observations)")
> op = par(xpd = TRUE)
> text(34.5, 3.05, "Number taken")
> par(op)
```

You could also use

```
> barplot(t(t1 + 1), log = "y", beside = TRUE, xlab = "Available",
          ylab = "1+# observations")
```

As mentioned in the text, $\log_{10}(t1+1)$ finds $\log(x + 1)$, a reasonable transformation to compress the range of discrete data; `t` transposes the table so we can plot groups by number available. The `beside=TRUE` argument plots grouped rather than stacked bars; `legend=TRUE` plots a legend; and `xlab` and `ylab` set labels. The statement `par(xpd=TRUE)` allows text and lines to be plotted outside the edge of the plot; the `op=par(...)` and `par(op)` are a way to set parameters and then restore the original settings (I could have called `op` anything I wanted, but in this case it stands for **old** parameters).

You can use `barchart` in the `lattice` package to produce these graphics, although the bars are horizontal rather than vertical by default. Try the following (`stack=FALSE` is equivalent to `beside=TRUE` for `barplot`):

```
> library(lattice)
> barchart(log10(1+table(available,taken)),
           stack=FALSE,
           auto.key=TRUE)
```

More impressively, the `lattice` package can automatically plot a barplot of a three-way cross-tabulation, in small multiples (I had to experiment a bit to get the factors in the right order in the `table` command): try

```
> barchart(log10(1+table(available,species,taken)),
           stack=FALSE,auto.key=TRUE)
```

Exercise 2.1*: Restricting your analysis to only the observations with 5 seeds available, create a barplot showing the distribution of number of seeds taken broken down by species. *Hints*: you can create a new data set that includes only the appropriate rows by using row indexing, then `attach` it.

2.3 Mean fraction taken: barplot with error bars

Computing the fraction taken:

```
> frac_taken = taken/available
```

Computing the mean fraction taken for each number of seeds available, using the `tapply` function: `tapply` (“table **apply**”, pronounced “t apply”), is an extension of the `table` function; it splits a specified vector into groups according to the factors provided, then *applies* a function (e.g. `mean` or `sd`) to each group. This idea of applying a function to a set of objects is a very general, very powerful idea in data manipulation with R; in due course we’ll learn about `apply` (apply a function to rows and columns of matrices), `lapply` (apply a function to lists), `sapply` (apply a function to lists and simplify), and `mapply` (apply a function to multiple lists). For the present, though,

```
> mean_frac_by_avail = tapply(frac_taken, available, mean)
```

computes the mean of `frac_taken` for each group defined by a different value of `available` (R automatically converts `available` into a factor temporarily for this purpose).

If you want to compute the mean by group for more than one variable in a data set, use `aggregate`.

We can also calculate the standard errors, σ/\sqrt{n} :

```
> n_by_avail = table(available)
> se_by_avail = tapply(frac_taken,available,
                      sd,na.rm=TRUE)/
                      sqrt(n_by_avail)
```

I'll actually use a variant of `barplot`, `barplot2` (from the `gplots` package, which you may need to install, along with the `gtools` and `gdata` packages) to plot these values with standard errors. (I am mildly embarrassed that R does not supply error-bar plotting as a built-in function, but you can use the `barplot2` in the `gplots` package or the `plotCI` function (the `gplots` and `plotrix` packages have slightly different versions).

```
> library(gplots)
> lower_lim = mean_frac_by_avail-se_by_avail
> upper_lim = mean_frac_by_avail+se_by_avail
> b = barplot2(mean_frac_by_avail,plot.ci=TRUE,
  ci.l=lower_lim,ci.u=upper_lim,
  xlab="Number available",
  ylab="Mean number taken")
```

I specified that I wanted error bars plotted (`plot.ci=TRUE`) and the lower (`ci.l`) and upper (`ci.u`) limits.

Bar plot of mean fraction taken *by species* — in this case we use `barplot`, saving the x locations of the bars in a variable `b`, and then add the confidence intervals with `plotCI`.

```
> library(plotrix)
> frac.taken = SeedPred$taken/SeedPred$available
> mean.frac.by.avail.by.species =
  tapply(frac.taken,list(available,species),mean,na.rm=TRUE)
> n.by.avail.by.species = table(available,species)
> se.by.avail.by.species = tapply(frac.taken,list(available,species),
  sd,na.rm=TRUE)/sqrt(n.by.avail.by.species)
> b = barplot(mean.frac.by.avail.by.species,beside=TRUE)
> plotCI(b,mean.frac.by.avail.by.species,
  se.by.avail.by.species,add=TRUE,pch=".",gap=FALSE)
```

3D plots: using `t1` from above, define the x , y , and z variables for the plot:

```
> avail = row(t1)[t1 > 0]
> taken = col(t1)[t1 > 0] - 1
> freq = log10(t1[t1 > 0])
```

The `scatterplot3d` package is a little bit simpler to use, but less interactive — once the plot is drawn you can't change the viewpoint. Plot `-avail` and `-taken` to reverse the order of the axes and use `type="h"` (originally named for a “high density” plot in R's 2D graphics) to draw lollipop:

```
> library(scatterplot3d)
> scatterplot3d(-avail, -taken, freq, type = "h", angle = 50, pch = 16)
```

With the `rgl` package: first plot spheres (`type="s"`) hanging in space:

```
> library(rgl)
> plot3d(avail, taken, freq, lit = TRUE, col.pt = "gray", type = "s",
        size = 0.5, zlim = c(0, 4))
```

Then add stems and grids to the plot:

```
> plot3d(avail, taken, freq, add = TRUE, type = "h", size = 4,
        col = gray(0.2))
> grid3d(c("x+", "y-", "z"))
```

Use the mouse to move the viewpoint until you like the result.

2.4 Histograms/small multiples

All I had to do to get the `lattice` package to plot the histogram by species was:

```
> histogram(~frac_taken | species, xlab = "Fraction taken")
```

or with base graphics:

```
> op = par(mfrow=c(3,3))
> for (i in 1:length(levels(species))) {
  hist(frac.taken[species==levels(species)[i]],
      xlab="Fraction taken",main="",
      col="gray")
}
```

```
> par(op)
```

`op` stands for “old parameters”. Saving the old parameters in this way and using `par(op)` at the end of the plot restores the original graphical parameters.

Clean up:

```
> detach(SeedPred)
```

Plots in this section: scatterplot (plot or xyplot) bubble plot (sizeplot), barplot (barplot or barchart or barplot2), histogram (hist or histogram).

Data manipulation: reshape, stack/unstack, table, split, lapply, sapply

Exercise 2.2*: generate three new plots based on one of the data sets in this lab (or elsewhere in Chapter 2), or on your own data.

3 Tadpole data

As mentioned in the text, reading in the data was fairly easy in this case: `read.table(...,header=TRUE)` and `read.csv` worked without any tricks. I take a shortcut, therefore, to load these datasets from the `emdbook` library:

```
> library(emdbook)
> data(ReedfrogPred)
> data(ReedfrogFuncresp)
> data(ReedfrogSizepred)
```

3.1 Boxplot of factorial experiment

The boxplot is fairly easy:

```
> graycols = rep(rep(gray(c(0.4, 0.7, 0.9)), each = 2), 2)
> boxplot(propsurv ~ size * density * pred, data = ReedfrogPred,
          col = graycols)
```

Play around with the order of the factors to see how useful the different plots are.

`graycols` specifies the colors of the bars to mark the different density treatments. `gray(c(0.4,0.7,0.9))` produces a vector of colors; `rep(gray(c(0.4,0.7,0.9)),each=2)` repeats each color twice (for the big and small treatments within each density treatment; and `rep(rep(gray(c(0.4,0.7,0.9)),each=2),2)` repeats the whole sequence twice (for the no-predator and predator treatments).

3.2 Functional response values

I'll attach the functional response data (`warn=FALSE` says not to warn about any variables that are masked by the newly attached data):

```
> attach(ReedfrogFuncresp, warn = FALSE)
```

A simple x - y plot, with an extended x axis and some axis labels:

```
> plot(Initial, Killed, xlim = c(0, 100), ylab = "Number killed",
      xlab = "Initial density")
```

Adding the `lowess` fit (`lines` is the general command for adding lines to a plot: `points` is handy too):

```
> lines(lowess(Initial, Killed))
```

Calculate mean values and corresponding initial densities, add to the plot with a different line type:

```
> meanvals = tapply(Killed, Initial, mean)
> densvals = unique(Initial)
> lines(densvals, meanvals, lty = 3)
```

Fit a spline to the data using the `smooth.spline` command:

```
> lms = smooth.spline(Initial, Killed, df = 5)
```

To add the spline curve to the plot, I have to use `predict` to calculate the predicted values for a range of initial densities, then add the results to the plot:

```
> ps = predict(lms, x = 0:100)
> lines(ps, lty = 2)
```

Equivalently, I could use the `lm` function with `ns` (natural spline), which is a bit more complicated to use in this case but has more general uses:

```
> library(splines)
> lm1 = lm(Killed ~ ns(Initial, df = 5), data = ReedfrogSizepred)
> p1 = predict(lm1, newdata = data.frame(Initial = 1:100))
> lines(p1, lty = 2)
```

Finally, I could do linear or quadratic regression (I need to use `I(Initial^2)` to tell R I really want to fit the square of the initial density); adding the lines to the plot would follow the procedure above.

```
> lm2 = lm(Killed ~ Initial, data = ReedfrogSizepred)
> lmq = lm(Killed ~ Initial + I(Initial^2), data = ReedfrogSizepred)
```

Clean up:

```
> detach(ReedfrogFuncresp)
```

The (tadpole size) vs. (number killed) plot follows similar lines, although I did use `sizeplot` because there were overlapping points.

4 Damsel fish data

4.1 Survivors as a function of density

Load and attach data:

```
> data(DamselRecruitment)
> data(DamselRecruitment_sum)
> attach(DamselRecruitment)
> attach(DamselRecruitment_sum)
```

Plot surviving vs. initial density; use `plotCI` to add the summary data by target density; and add a `lowess`-smoothed curve to the plot:

```
> init.dens = init/area*1000
> surv.dens = surv/area*1000
> plot(init.dens, surv.dens, log="x")
> plotCI(settler.den, surv.den, SE,
         add=TRUE, pch=16, col="darkgray", gap=0)
> lines(lowess(init.dens, surv.dens))
```

Clean up:

```
> detach(DamselRecruitment)
> detach(DamselRecruitment_sum)
```

4.2 Distribution of settlement density

Plot the histogram (normally one would specify `freq=FALSE` to plot probabilities rather than counts, but the uneven `breaks` argument makes this happen automatically).

```
> data(DamselSettlement)
> attach(DamselSettlement)
> hist(density[density < 200], breaks = c(0, seq(1, 201, by = 4)),
      col = "gray", xlab = "", ylab = "Prob. density")
> lines(density(density[density < 200], from = 0), col = 2, lwd = 2)
```

Some alternatives to try:

```
> hist(log(1 + density))
> hist(density[density > 0], breaks = 50)
```

(you can use `breaks` to specify particular breakpoints, or to give the total number of bins to use).

If you really want to lump all the large values together:

```
> h1 = hist(density, breaks = c(0, seq(1, 201, by = 4), 500), plot = FALSE)
> b = barplot(h1$counts, space = 0)
> axis(side = 1, at = b, labels = h1$mids)
```

(use `hist` to calculate the number of counts in each bin, but don't plot anything; use `barplot` to plot the values (ignoring the uneven width of the bins!), with `space=0` to squeeze them together).

Box and whisker plots showing density across different recruitment pulses at different sites:

```
> bwplot(log10(1 + density) ~ pulse | site, data = DamselSettlement,
         horizontal = FALSE)
```

Other variations to try:

```
> ## density distributions plotted in a single frame
> densityplot(~density, groups=site, data=DamselSettlement, xlim=c(0,100))
> ## box-and-whiskers of overall settlement by site
> bwplot(density~site, horizontal=FALSE, data=DamselSettlement)
> ## density rather than log10(1+density)
> bwplot(density~site|factor(pulse), horizontal=FALSE, data=DamselSettlement)
> ## violin plots
> bwplot(log10(1+density)~site|factor(pulse), data=DamselSettlement,
         panel=panel.violin,
         horizontal=FALSE)
> ## all site-pulse combinations, in base graphics (ugly)
> boxplot(density~site*pulse)

> detach(DamselSettlement)
```