2 Exploratory Data Analysis and Graphics

This chapter covers both the practical details and the broader philosophy of (1) reading data into R and (2) doing exploratory data analysis, in particular, graphical analysis. To get the most out of the chapter you should already have some basic knowledge of R's syntax and commands (see the R supplement of the previous chapter).

2.1 Introduction

One of the basic tensions in all data analysis and modeling is how much you have all your questions framed before you begin to look at your data. In the classical statistical framework, you're supposed to lay out all your hypotheses before you start, run your experiments, come back to your office and test those (and only those) hypotheses. Allowing your data to suggest new statistical tests raises the risk of "fishing expeditions" or "data-dredging"—indiscriminately scanning your data for patterns.* Data-dredging is a serious problem. Humans are notoriously good at detecting apparent patterns even when they don't exist. Strictly speaking, interesting patterns that you find in your data after the fact should not be treated statistically, only used as input for the next round of observations and experiments.† Most statisticians are leery of procedures like stepwise regression that search for the best predictors or combinations of predictors from among a large range of options, even though some have elaborate safeguards to avoid overestimating the significance of observed patterns (Whittingham et al., 2006). The worst aspect of such techniques is that in order to use them you must be conservative and discard real patterns, patterns that you originally had in mind, because you are screening your data indiscriminately (Nakagawa, 2004).

* "Bible codes," where people find hidden messages in the Bible, illustrate an extreme form of data-dredging. Critics have pointed out that similar procedures will also detect hidden messages in *War and Peace* or *Moby Dick* (McKay et al., 1999).

† Or you should apply a post hoc procedure [see `?TukeyHSD` and the `multcomp` package in R] that corrects for the fact that you are testing a pattern that was not suggested in advance—however, even these procedures apply corrections only for a specific set of possible comparisons, not for all possible patterns that you could have found in your data.

But these injunctions may be too strict for ecologists. Unexpected patterns in the data can inspire you to ask new questions, and it is foolish not to explore your hard-earned data. *Exploratory data analysis* (EDA; Tukey, 1977; Cleveland, 1993; Hoaglin et al., 2000, 2006) is a set of graphical techniques for finding interesting patterns in data. EDA was developed in the late 1970s when computer graphics first became widely available. It emphasizes *robust* and *nonparametric* methods, which make fewer assumptions about the shapes of curves and the distributions of the data and hence are less sensitive to nonlinearity and outliers. Most of the rest of this book will focus on models that, in contrast to EDA, are parametric (i.e., they specify particular distributions and curve shapes) and mechanistic. These methods are more powerful and give more ecologically meaningful answers, but they are also susceptible to being misled by unusual patterns in the data.

The big advantages of EDA are that it gets you looking at and thinking about your data (whereas stepwise approaches are often substitutes for thought), and that it may reveal patterns that standard statistical tests would overlook because of their emphasis on specific models. However, EDA isn't a magic formula for interpreting your data without the risk of data dredging. Only common sense and caution can keep you in the zone between ignoring interesting patterns and overinterpreting them. It's useful to write down a list of the ecological patterns you're looking for and how they relate your ecological questions *before* you start to explore your data, so that you can distinguish among (1) patterns you were initially looking for, (2) unanticipated patterns that answer the same questions in different ways, and (3) interesting (but possibly spurious) patterns that suggest new questions.

The rest of this chapter describes how to get your data into R and how to make some basic graphs in order to search for expected and unexpected patterns. The text covers both philosophy and some nitty-gritty details. The supplement at the end of the chapter gives a sample session and more technical details.

2.2 Getting Data into R

2.2.1 Preliminaries

ELECTRONIC FORMAT

Before you can analyze your data you have to get them into R. Data come in a variety of formats—in ecology, most are either plaintext files (space- or comma-delimited) or Excel files.* R prefers plaintext files with "white space" (arbitrary numbers of tabs or spaces) or commas between columns. Text files are less structured and may take up more disk space than more specialized formats, but they are the lowest common denominator of file formats and so can be read by almost anything (and, if necessary, examined and adjusted in any text editor). Since a wide variety of text editors can read plaintext formats, they are unlikely to be made obsolete by changes

* Your computer may be set up to open comma-delimited (`.csv`) files in Excel, but underneath they are just text files.

in technology (you could say they're already obsolete), and less likely to be made unusable by corruption of a few bits of the file; only hard copy is better.*

R is platform-agnostic. While text files do have very slightly different formats on Unix, Microsoft Windows, and Macintosh operating systems, R handles these differences. If you later save data sets or functions in R's own format (using `save` to save and `load` to load them), you will be able to exchange them freely across platforms.

Many ecologists keep their data in Excel spreadsheets. The `read.xls` function in the `gdata` package allows R to read Excel files directly, but the best thing to do with an Excel file (if you have access to a copy of Excel, or if you can open it in an alternative spreadsheet program) is to save the worksheet you want as a `.csv` (comma-separated values) file. Saving as a `.csv` file will also force you to go into the worksheet and clean up any random cells that are outside of the main data table—R won't like these. If your data are in some more exotic form (e.g., within a GIS or database system), you'll have to figure out how to extract them from that particular system into a text file. There are ways of connecting R directly with databases or GIS systems, but they're beyond the scope of this book. If you have trouble exporting data or you expect to have large quantities of data (e.g., more than tens of thousands of observations) in one of these exotic forms, read the R *Data Import/Export Manual*, which is accessible through Help in the R menus.

METADATA

Metadata is the information that describes the properties of a data set: the names of the variables, the units they were measured in, when and where the data were collected, etc. R does not have a structured system for maintaining metadata, but it does allow you to include a good deal of this metadata within your data file, and it is good practice to keep as much of this information as possible associated with the data file. Some tips on metadata in R:

- Column names are the first row of the data set. Choose names that compromise between convenience (you will be typing these names a lot) and clarity; `larval_density` or `larvdens` is better than either `x` or `larval_density_per_m3_in_ponds`. Use underscores or dots to separate words in variable names, not spaces. Begin names with a letter, not a number.
- R will ignore any information on a line following a `#`. I usually use this comment character to include general metadata at the beginning of my data file, such as the data source, units, and so forth—anything that can't easily be encoded in the variable names. I also use comments before, or at the ends of, particular lines in the data set that might need annotation, such as the circumstances surrounding questionable data points. You can't use `#` to make a comment in the middle of a line: use a comment like `# pH calibration failed` at the end of the line to indicate that a particular field in that line is suspect.

* Unless your data are truly voluminous, you should also save a hard-copy, archival version of your data (Gotelli and Ellison, 2004).

- If you have other metadata that can't easily be represented in plaintext format (such as a map), you'll have to keep it separately. You can reference the file in your comments, keep a separate file that lists the location of data and metadata, or use a system like Morpho (from `ecoinformatics.org`) to organize it.

Whatever you do, make sure that you have some workable system for maintaining your metadata. Eventually, your R scripts—which document how you read in your data, transformed it, and drew conclusions from it—will also become a part of your metadata. As mentioned in Chapter 1, this is one of the advantages of R over (say) Excel: after you've done your analysis, *if you were careful to document your work sufficiently as you went along*, you will be left with a set of scripts that will allow you to verify what you did; make minor modifications and rerun the analysis; and apply the same or similar analyses to future data sets.

SHAPE

Just as important as electronic or paper format is the organization or *shape* of your data. Most of the time, R prefers that your data have a single *record* (typically a line of data values) for each individual observation. This basically means that your data should usually be in "long" (or "indexed") format. For example, the first few lines of the seed removal data set look like this, with a line giving the number of seeds present for each station/date combination:

	station	date	dist	species	seeds
1	1	1999-03-23	25	psd	5
2	1	1999-03-27	25	psd	5
3	1	1999-04-03	25	psd	5
4	2	1999-03-23	25	uva	5
5	2	1999-03-27	25	uva	5
6	2	1999-04-03	25	uva	5

Because each station has seeds of only one species and can be at only a single distance from the forest, these values are repeated for every date. During the first two weeks of the experiment no seeds of `psd` or `uva` were taken by predators, so the number of seeds remained at the initial value of 5.

Alternatively, you will often come across data sets in "wide" format, like this:

	station	species	dist	seeds.1999-03-23	seeds.1999-03-27
1	1	psd	25	5	5
2	2	uva	25	5	5
3	3	pol	25	5	4
4	4	dio	25	5	5
5	5	cor	25	5	4
6	6	abz	25	5	5

(I kept only the first two date columns in order to make this example narrow enough to fit on the page.)

Long format takes up more room, especially if you have data (such as `dist` above, the distance of the station from the edge of the forest) that apply to each

station independent of sample date or species (which therefore have to be repeated many times in the data set). However, you'll find that this format is typically what statistical packages request for analysis.

You can read data into R in wide format and then convert it to long format. R has several different functions—`reshape` and `stack`/`unstack` in the base package, and `melt`/`cast`/`recast` in the `reshape` package*—that will let you switch data back and forth between wide and long formats. Because there are so many different ways to structure data, and so many different ways you might want to aggregate or rearrange them, software tools designed to reshape arbitrary data are necessarily complicated (Excel's pivot tables, which are also designed to restructure data, are as complicated as `reshape`).

- `stack` and `unstack` are simple but basic functions—`stack` converts from wide to long format and `unstack` from long to wide; they aren't very flexible.
- `reshape` is very flexible and preserves more information than `stack`/`unstack`, but its syntax is tricky: if `long` and `wide` are variables holding the data in the examples above, then

  ```
  > reshape(wide, direction = "long", timevar = "date",
  +     varying = 4:5)
  > reshape(long, direction = "wide", timevar = "date",
  +     idvar = c("station", "dist", "species"))
  ```

  convert back and forth between them. In the first case (wide to long) we specify that the time variable in the new long-format data set should be `date` and that columns 4–5 are the variables to collapse. In the second case (long to wide) we specify that `date` is the variable to expand and that `station`, `dist`, and `species` should be kept fixed as the identifiers for an observation.
- The `reshape` package contains the `melt`, `cast`, and `recast` functions, which are similar to `reshape` but sometimes easier to use, e.g.,

  ```
  > library(reshape)
  > recast(wide, formula = ... ~ ., id.var = c("station",
  +     "dist", "species"))
  > recast(long, formula = station + dist + species ~
  +     ..., id.var = c("station", "dist", "species",
  +     "date"))
  ```

  in the formulas above, `...` denotes "all other variables" and `.` denotes "nothing," so the formula `...~.` means "separate out by all variables" (long format) and `station+dist+species~...` means "separate out by station, distance, and species, put the values for each date on one line."

In general you will have to look carefully at the examples in the documentation and play around with subsets of your data until you get it reshaped exactly the way you want. Alternatively, you can manipulate your data in Excel, either with pivot tables or by brute force (cutting and pasting). In the long run, learning to reshape data will pay off, but for a single project it may be quicker to use brute force.

* If you don't know what a package is, go back and read about them in the R supplement for Chapter 1.

2.2.2 Reading in Data

BASIC R COMMANDS

The basic R commands for reading in a data set, once you have it in a long-format text file, are `read.table` for space-separated data and `read.csv` for comma-separated data. If there are no complications in your data, you should be simply be able to say (e.g.)

```
> data = read.table("mydata.dat", header = TRUE)
```

(if your file is actually called `mydata.dat` and includes a first row with the column names) to read your data in (as a *data frame*; see p. 35) and assign it to the variable `data`.

Reading in files presents several potential complications, which are more fully covered in the R supplement: (1) telling R where to look for data files on your computer system; (2) checking that every line in the file has the same number of variables, or *fields*—R won't read it otherwise; and (3) making sure that R reads all your variables as the right data types (discussed in the next section).

2.3 Data Types

When you read data into a computer, the computer stores those data as some particular data *type*. This is partly for efficiency—it's more efficient to store numbers as strings of bits rather than as human-readable character strings—but its main purpose is to maintain a sort of metadata about variables, so the computer knows what to do with them. Some operations make sense only with particular types—what should you get when you try to compute `2+"A"? "2A"?` If you try to do something like this in Excel, you get an error code—`#VALUE!`; if you do it in R, you get the message `Error non-numeric argument to binary operator`.*

Computer packages vary in how they deal with data. Some lower-level languages like C are *strongly typed*; they insist that you specify exactly what type every variable should be and require you to convert variables between types (say integer and real, or floating-point) explicitly. Languages or packages like R or Excel are looser; they try to guess what you have in mind and convert variables between types (*coerce*) automatically as appropriate. For example, if you enter `3/25` into Excel, it automatically converts the value to a date—March 25 of the current year.

R makes similar guesses as it reads in your data. By default, if every entry in a column is a valid number (e.g., `234`, `-127.45`, `1.238e3` [computerese for 1.238×10^3]), then R guesses the variable is numeric. Otherwise, it makes it a *factor*—an indexed list of values used to represent categorical variables, which I will describe in more detail shortly. Thus, any error in a numeric variable (extra decimal point, included letter, etc.) will lead R to classify that variable as a factor rather than a number. R also has a detailed set of rules for dealing with missing values (internally

* The + symbol is called a "binary operator" because it is used to combine two values.

represented as NA, for Not Available). If you use missing-value codes (such as * or -9999) in your data set, you have to tell R about it or it will read them naively as strings or numbers.

While R's standard rules for guessing about input data are pretty simple and allow you only two options (numeric or factor), there are a variety of ways for specifying more detail either as R reads in your data or after it has read them in; these are covered in more detail in the accompanying material.

2.3.1 Basic Data Types

R's basic (or *atomic*) data types are integer, numeric (real numbers), logical (TRUE or FALSE), and character (alphanumeric strings). (There are a few more, such as complex numbers, that you probably won't need.) At the most basic level, R organizes data into *vectors* of one of these types, which are just ordered sets of data. Here are a couple of simple (numeric and character) vectors:

```
> 1:5
[1] 1 2 3 4 5
> c("yes", "no", "maybe")
[1] "yes"   "no"    "maybe"
```

More complicated data types include dates (Date) and factors (factor). Factors are R's way of dealing with categorical variables. A factor's underlying structure is a set of (integer) levels along with a set of the labels associated with each level.

One advantage of using these more complex types, rather than converting your categorical variables to numeric codes, is that it's much easier to remember the meaning of the levels as you analyze your data, for example, north and south rather than 0 and 1. Also, R can often do the right things with your data automatically if it knows what types they are (this is an example of crude-versus-sophisticated where a little more sophistication may be useful). Much of R's built-in statistical modeling software depends on these types to do the right analyses. For example, the command lm(y~x) (meaning "fit a linear model of y as a function of x," analogous to SAS's PROC GLM) will do an ANOVA if x is categorical (i.e., stored as a factor) or a linear regression if x is numeric. If you want to analyze variation in population density among sites designated with integer codes (e.g., 101, 227, 359) and haven't specified that R should interpret the codes as categorical rather than numeric values, R will try to fit a linear regression rather than doing an ANOVA. Many of R's plotting functions will also do different things depending on what type of data you give them. For example, R can automatically plot date axes with appropriate labels. To repeat, data types are a form of metadata; the more information about the meaning of your data that you can retain in your analysis, the better.

2.3.2 Data Frames and Matrices

R can organize data at a higher level than simple vectors. A *data frame* is a table of data that combines vectors (columns) of different types (e.g., character, factor, and

numeric data). Data frames are a hybrid of two simpler data structures: *lists*, which can mix arbitrary types of data but have no other structure, and *matrices*, which are structured by rows and columns but usually contain only one data type (typically numeric). When treating the data frame as a list, you can extract columns of data from the data frame in a variety of different ways:

```
> SeedPred[[2]]
> SeedPred[["species"]]
> SeedPred{$}species
```

all extract the second column (a factor containing species abbreviations) from the data frame `SeedPred`. You can also treat the data frame as a matrix and use square brackets `[]` to extract the second column:

```
> SeedPred[, 2]
> SeedPred[, "species"]
```

or rows 1 through 10

```
> SeedPred[1:10, ]
```

(`SeedPred[i,j]` extracts the matrix element in row(s) `i` and column(s) `j`; leaving the columns or rows specification blank, as in `SeedPred[i,]` or `SeedPred[,j]`, takes row `i` (all columns) or column `j` (all rows) respectively.) A few operations, such as transposing or calculating a variance-covariance matrix, work only with matrices (not with data frames); R will usually convert (*coerce*) data frames to matrices automatically when it makes sense to, but you may sometimes have to use `as.matrix` to manually convert a data frame to a matrix.*

2.3.3 Checking Data

Now suppose you've decided on appropriate types for all your data and told R about it. Are the data you've read in actually correct, or are there still typographical or other errors?

SUMMARY

First check the results of `summary`. For a numeric variable `summary` will list the minimum, first quartile, median, mean, third quartile, and maximum. For a factor it will list the numbers of observations with each of the first six factor levels, then the number of remaining observations. (Use `table` on a factor to see the numbers of observations at all levels.) It will list the number of `NA`s for all types.

* Matrices and data frames can appear identical but behave differently. If `x` is a data frame, either `colnames(x)` or `names(x)` will tell you the column names. If `x` has a column called `a`, either `x$a` or `x[["a"]]` or `x[,"a"]` will retrieve it. If `x` is a matrix, you must use `colnames(x)` to get the column names and `x[,"a"]` to retrieve a column (the other commands will give errors). Use `is.data.frame` or `class` to tell matrices and data frames apart.

For example:

```
> summary(SeedPred[, 1:4])
```

```
    station          dist           species              date
1      :      74   10:5883      abz      :1480   Min.    :1999-03-23
2      :      74   25:5920      cd       :1480   1st Qu. :1999-05-23
3      :      74                cor      :1480   Median  :1999-07-24
4      :      74                dio      :1480   Mean    :1999-07-25
5      :      74                pol      :1480   3rd Qu. :1999-09-28
6      :      74                psd      :1480   Max.    :1999-11-28
(Other):11359                 (Other)    :2923
```

(To keep the output short, I'm looking at the first four columns of the data frame only: `summary(SeedPred)` would summarize the whole thing.)

Check the following points:

- Is the total number of observations right? For factors, is the right number of observations in each level?
- Do the summaries of the numeric variables—mean, median, etc.—look reasonable? Are the minimum and maximum values about what you expected?
- Are the numbers of `NA`s in each column reasonable? If not (especially if you have extra mostly `NA` columns), you may want to go back a few steps and use `count.fields` to identify rows with extra fields.

STR

The `str` command tells you about the **str**ucture of an R variable: it is slightly less useful than `summary` for dealing with data, but it may come in handy later for figuring out more complicated R variables. Applied to a data frame, it tells you the total number of observations (rows) and variables (columns) and prints out the names and classes of each variable along with the first few observations in each variable.

```
> str(SeedPred)
```

```
'data.frame':   11803 obs. of  9 variables:
 $ station  : Factor w/ 160 levels "1","2","3","4",..: 1 1 1 1 1 1 1
              1 1 1 ...
 $ dist     : Factor w/ 2 levels "10","25": 1 1 1 1 1 1 1 1 1 1 ...
 $ species  : Factor w/ 8 levels "abz","cd","cor",..: 7 7 7 7 7 7 7
              7 7 7 ...
 $ date     : Class 'Date'  num [1:11803] 10675 10678 10685 10692
              10699 ...
 $ seeds    : int  5 5 5 5 0 0 0 0 0 0 ...
 $ tcum     : num  0 3 10 17 24 31 39 46 53 60 ...
 $ tint     : num  NA 3 7 7 7 7 8 7 7 7 ...
 $ taken    : int  NA 0 0 0 5 0 0 0 0 0 ...
 $ available: int  NA 5 5 5 5 0 0 0 0 0 ...
```

CLASS

The `class` command prints out the class (`numeric`, `factor`, `Date`, `logical`, etc.) of a variable. `class(SeedPred)` gives `"data.frame"`; `sapply(SeedPred,class)` applies `class` to each column of the data individually.

```
> class(SeedPred)

[1] "data.frame"

> sapply(SeedPred, class)

 station     dist        species     date      seeds       tcum
 "factor"    "factor"    "factor"    "Date"    "integer"   "numeric"
 tint        taken       available
 "numeric"   "integer"   "integer"
```

HEAD

The `head` command just prints out the beginning of a data frame; by default it prints the first six rows, but `head(data,10)` (e.g.) will print out the first 10 rows.

```
> head(SeedPred)

  station dist species     date seeds tcum tint taken available
1       1   10     psd 1999-03-25     5    0   NA    NA        NA
2       1   10     psd 1999-03-28     5    3    3     0         5
3       1   10     psd 1999-04-04     5   10    7     0         5
4       1   10     psd 1999-04-11     5   17    7     0         5
5       1   10     psd 1999-04-18     0   24    7     5         5
6       1   10     psd 1999-04-25     0   31    7     0         0
```

The `tail` command prints out the end of a data frame.

TABLE

`table` is R's command for cross-tabulation; you can use it to check that you have appropriate numbers of observations in different factor combinations.

```
> table(SeedPred$station, SeedPred$species)

    abz cd cor dio mmu pol psd uva
1     0  0   0   0   0   0  74   0
2     0  0   0   0   0   0   0  74
3     0  0   0   0   0  74   0   0
4     0  0   0  74   0   0   0   0
5     0  0  74   0   0   0   0   0
6    74  0   0   0   0   0   0   0
```

(just the first six lines are shown): apparently, each station has seeds of only a single species. The `$` extracts variables from the data frame `SeedPred`, and `table` says we

want to count the number of instances of each combination of station and species; we could also do this with a single factor or with more than two.

DEALING WITH NAs

Missing values are a nuisance, but a fact of life. Throwing out or ignoring missing values is tempting, but it can be dangerous. Ignoring missing values can bias your analyses, especially if the pattern of missing values is not completely random. R is conservative by default and assumes that, for example, `2+NA` equals `NA`—if you don't know what the missing value is, then the sum of it and any other number is also unknown. Almost any calculation you make in R will be contaminated by `NA`s, which is logical but annoying. Perhaps most difficult is that you can't just do what comes naturally and say (e.g.) `x=x[x!=NA]` to remove values that are `NA` from a variable, because even comparisons to `NA` result in `NA`!

- You can use the special function `is.na` to count the number of `NA` values (`sum(is.na(x))`) or to throw out the `NA` values in a vector (`x=x[!is.na(x)]`).
- Functions such as `mean`, `var`, `sd`, `sum` (and some others) have an optional `na.rm` argument: `na.rm=TRUE` drops `NA` values before doing the calculation. Otherwise if `x` contains any `NA`s, `mean(x)` will result in `NA` and `sd(x)` will give an error about missing observations.
- To convert `NA` values to a particular value, use `x[is.na(x)]=value`; e.g., to set `NA`s to zero `x[is.na(x)]=0`, or to set `NA`s to the mean value `x[is.na(x)]=mean(x,na.rm=TRUE)`. *Don't do this unless you have a very good, and defensible, reason.*
- `na.omit` will drop `NA`s from a vector (`na.omit(x)`), but it is also smart enough to do the right thing if `x` is a data frame instead, and throw out all the *cases* (rows) where *any* variable is `NA`; however, this may be too stringent if you are analyzing a subset of the variables. For example, you might have a really unreliable soil moisture meter that produces lots of `NA`s, but you don't need to throw away all of these data points while you're analyzing the relationship between light and growth. (`complete.cases` returns a logical vector that says which rows have no `NA`s; if `x` is a data frame, `na.omit(x)` is equivalent to `x[complete.cases(x),]`.)
- Calculations of covariance and correlation (`cov` and `cor`) have more complicated options: `use="all.obs"`, `use="complete.obs"`, or `use="pairwise.complete.obs"`. `all.obs` uses all of the data (but the answer will contain `NA`s every time either variable contains one); `complete.obs` uses only the observations for which *none* of the variables are `NA` (but may thus leave out a lot of data); and `pairwise.complete.obs` computes the pairwise covariance/correlations using the observations where both of each particular pair of variables are non-NA (but may lead in some cases to incorrect estimates).

As you discover errors in your data, you may have to go back to your original data set to correct errors and then reenter them into R (using the commands you have saved, of course). Or you can change a few values in R, e.g.,

```
> SeedPred[24, "species"] = "mmu"
```

to change the species in the 24th observation from `psd` to `mmu`. Whatever you do, document this process as you go along, and always maintain your original data set in its original, archival, form, even including data you think are errors (this is easier to remember if your original data set is in the form of field notebooks). Keep a log of what you modify so conflicting versions of your data don't confuse you.

2.4 Exploratory Data Analysis and Graphics

The next step in checking your data is to graph them, which leads on naturally to exploring patterns. Graphing is the best way to understand not only data, but also the models that you fit to data; as you develop models you should graph the results frequently to make sure you understand how the model is working.

R gives you complete control of all aspects of graphics (Figure 1.7) and lets you save graphics in a wide range of formats. The only major nuisance of doing graphics in R is that R constructs graphics as though it were drawing on a static page, not by adding objects to a dynamic scene. You generally specify the positions of all graphics on the command line, not with the mouse (although the `locator` and `identify` functions can be useful). Once you tell R to draw a point, line, or piece of text there is no way to erase or move it. The advantage of this procedure, like logging your data manipulations, is that you have a complete record of what you did and can easily recreate the picture with new data.

R actually has two different coexisting graphics systems. The base graphics system is cruder and simpler, while the lattice graphics system (in the `lattice` package) is more sophisticated and complex. Both can create scatterplots, box-and-whisker plots, histograms, and other standard graphical displays. Lattice graphics do more automatic processing of your data and produce prettier graphs, but the commands are harder to understand and customize. In the realm of 3D graphics, there are several more options, at different stages of development. Base graphics and lattice graphics both have some 3D capabilities (`persp` in base, `wireframe` and `cloud` in lattice); the `scatterplot3d` package builds on base to draw 3D point clouds; the `rgl` package (still under development) allows you to rotate and zoom the 3D coordinate system with the mouse; and the `ggobi` package is an interface to a system for visualizing multidimensional point data.

2.4.1 Seed Removal Data: Discrete Numeric Predictors, Discrete Numeric Responses

As described in Chapter 1, the seed removal data set from Duncan and Duncan (2000) gives information on the rate at which seeds were removed from experimental stations set up in a Ugandan grassland. Seeds of eight species were set out at stations along two transects different distances from the forest and monitored every few days for more than eight months. We have already seen a subset of these data in a brief example, but we haven't really examined the details of the data set. There are a total of 11,803 observations, each containing information on the station number (`station`), distance in meters from the forest edge (`dist`), the species