# Chapter 2
# R Tutorial for Oceanographers

**Abstract** R comes with an excellent tutorial that, like many fine tutorials, tends to be ignored by people with little patience for material presented in a general manner. This is why the present chapter uses oceanographic examples to explain R concepts, and why code makes up so much of the text. The early examples are designed to encourage readers to become comfortable whilst navigating the R documentation, because this skill can be the key to moving from simple examples to real-world applications. The main concepts of R data types and language features are illustrated here in practical terms, with many of the explanations involving graphical representation. Since experienced R users are unlikely to study this chapter in great depth, specialized methods of oceanographic analysis are mainly deferred to succeeding chapters.

## 2.1 Introduction

R can be deceptive at first, because it handles simple tasks so well that newcomers might wonder if it has the power for advanced work. They need not worry, for R balances simplicity and power in ways both subtle and varied. Some users notice this first in the thoughtful system of default function arguments, through which R achieves simplicity without loss of flexibility. Others will focus on how R uses object orientation methods to generalize tasks, letting users think about science instead of syntax. Those with programming experience will see the benefits of the functional basis of R, and its innovative rules for the scope of variables and the evaluation of expressions. And those working on computationally demanding tasks will appreciate the R interfaces to C, C++ and Fortran, and its handling of multiple-processor systems.

R is a practical language that owes some of its strength to its lineage. Many of its best characteristics can be traced to the S and S-plus languages upon which it was patterned. These earlier systems were well designed at the outset, and were honed by use in advanced research settings (Becker and Chambers 1984; Becker et al. 1988; Chambers and Hastie 1992). R was also born in a research setting, which may

© Springer Science+Business Media, LLC, part of Springer Nature 2018
D. E. Kelley, *Oceanographic Analysis with R*,
https://doi.org/10.1007/978-1-4939-8844-0_2

explain why it has innovations that take it beyond the earlier languages (Ihaka and Gentleman 1996; Chambers 2008).

The important book by Venables and Ripley (1999) contains a wide-ranging and authoritative overview of R and its use, and it can be recommended to any reader. Dalgaard (2002) is a good companion, especially for beginners. The Chambers (2008) and Wickham (2014) treatments of technical aspects should prove useful to advanced users, especially those developing R packages. There are also many books about specialized topics, e.g. graphical display (Murrell 2006; Wickham 2009), time series analysis (Shumway and Stoffer 2006), neural networks (Ripley 1996), Bayesian methods (Albert 2009), numerical ecology (Borcard et al. 2011), etc. Readers should have little difficulty finding books on a specialized applications of R; for example, the present book is part of a Springer "UseR!" series that has dozens of titles.

In addition to texts, the extensive features of R are covered in detail in the official documentation (R Core Team 2017). For beginners, the most important part of this is the essay entitled "An introduction to R." Other essays deal with R as a language, with writing packages to extend the system, etc.; these are recommended to readers who already use R frequently. There is also a full reference manual that, spanning thousands of pages, is best consulted a little at a time.

A sensible way to learn R is to work through a tutorial. The "Introduction to R" can be used in this fashion, and few readers would be disappointed with its pacing, coverage or clarity. The present chapter is not as deep, nor as broad, but it does have two advantages: (a) it is cast in oceanographic terms, which may hold readers' interest better than general material and (b) it contains many exercises that should speed up the learning process.

By the end of this chapter, readers should be able to accomplish simple tasks in R, and understand code for more complicated tasks. This will set the stage for the upcoming chapters, in which the focus shifts more directly to oceanographic analysis. But, before any of this can be done, we must acknowledge the "elephant in the room", Matlab.

Over the past few decades, Matlab has become so popular in oceanography that many regard it as a *lingua franca* for data analysis. Figure 2.1 (and Appendix A) illustrates that mapping from Matlab to R is not difficult, especially when viewed in stages. For example, the first Matlab line

```
load xy.dat
```

causes a file named `xy.dat` to be read, with the numerical values being stored in a matrix named `xy`. Although the syntax is simple, it hides a great deal. Upon encountering the `load` token, Matlab interprets the next token as a file name, and constructs a variable with an analogous name, into which to store the contents of the file. By contrast, the equivalent line in R

```
xy <- read.table("xy.dat")
```

is more complicated, but also more direct. The "`<-`" token indicates assignment. To its left is the name of a variable to store the result of an expression to its right. In this case, the expression is the value returned by a function named `read.table()`

```
load xy.dat
plot(xy(:,1),xy(:,2))
quit
```

```
xy <- read.table("xy.dat")
plot(xy[,1], xy[,2])
q()
```

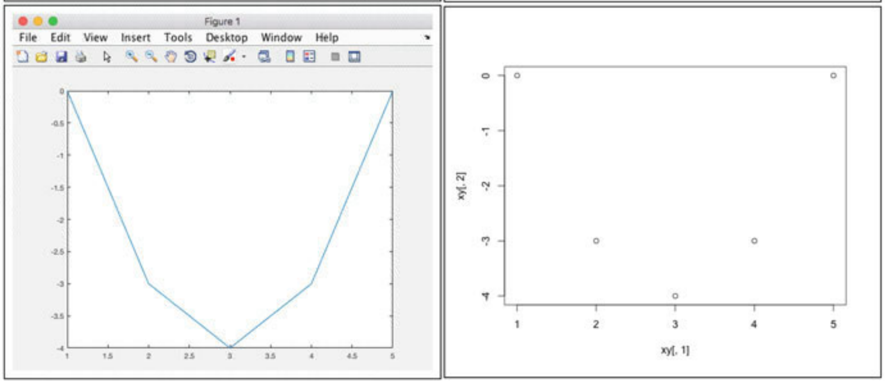

**Fig. 2.1** Comparison of Matlab and R for file input and graphics

that reads tabular data and returns a so-called "data frame" (discussed at length later) representing information within the named file.

Next, the Matlab line

```
plot(xy(:,1),xy(:,2))
```

constructs a line graph with the first column of `xy` taken as the *x* coordinate and the second as the *y* coordinate. The "`:`" means to use all rows. Note the use of parentheses in two very different ways here, to indicate arguments to a function and to indicate indices of a matrix. The R version

```
plot(xy[,1], xy[,2])
```

differs in several ways. First, it creates a scatter graph by default, although this can be changed to a line graph easily (see Exercise 2.1). Second, the axis labels indicate the names of the plotted items, which is very helpful in exploratory analysis because it reduces the need to change axis titles if variables are changed. Third, R uses square brackets for indexing and does not require the "`:`" place-holder.

Finally, a user exits Matlab with

```
quit
```

while the equivalent in R is

```
q()
```

which calls a function named `q()`. Matlab users may find it odd to exit a program by calling a function, but this fits the theme of R, which is a function-oriented language. Indeed, many things that one might think of as commands or operators in Matlab take the form of functions in R, and this has subtle and helpful effects that will become clearer in the remainder of this book.

**Exercise 2.1** Type `help(plot)` in a console, and use the results to see how to draw a line graph instead of a scatter plot. (See page 187 for a solution.)

**Exercise 2.2** Consult the documentation for `read.table()`, to see how to indicate that the first line of the file contains a line with the names of the columns. (See page 187 for a solution.)

**Exercise 2.3** Use the `mfrow` argument of `par()` to draw multi-panel plots in R, emulating the Matlab `subplot` command. (See page 188 for a solution.)

**Exercise 2.4** Use `outer()` to emulate the Matlab function `meshgrid`. (See page 188 for a solution.)

## 2.2 First Steps with R

### *2.2.1 License*

R is subject to a "GNU General Public License". This has three practical benefits. First, it means that R can be included in linux systems, which ensures distribution within a community of technically minded users who tend to help other users by contributing to online forums and sharing code. Second, R is an open source application, so that users can examine its internal workings, in case they want to evaluate the methods or extend them. And third, R is available free of charge, which has obvious benefits to students, researchers and consultants.

### *2.2.2 Installation*

On linux systems, R is installed in the same way as other software, either using GUI operations or by typing commands in a terminal. On other systems, installation is a simple matter of visiting the R website[1] and installing the appropriate pre-compiled version. Archived versions are also available there.

### *2.2.3 R Packages*

R benefits greatly from a scheme for combining code and documentation into so-called packages that are distributed in the Comprehensive R Archive Network, CRAN, which is available at the R website. As shown in Fig. 2.2, the number of

[1] http://cran.r-project.org/.

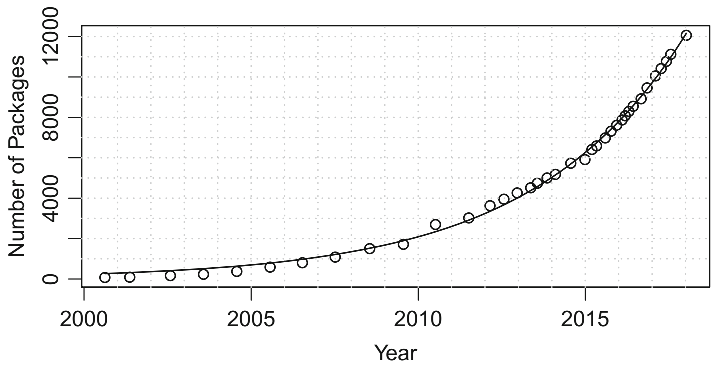

**Fig. 2.2** Number of R packages on the CRAN archive site. The curve results from using `nls()` to perform a nonlinear regression with model $n = n_0 \exp((t - t_0)/\tau)$, where $n$ is package count, $t - t_0$ is the time since the first point shown, and $n_0$ and $\tau$ are free parameters. This yields a doubling time of $3.2 \pm 0.1$ years (95% confidence interval)

packages[2] has increased dramatically since R was released, with over ten thousand being available at the time of writing. The coverage of these packages is broad, reflecting the broad popularity of R. Since packages are usually developed by experts in the sub-field of application, they tend to address relevant problems in up-to-date ways. Code quality tends to be high, given both the expertise of the developers and automated code-quality checks that are built into the packaging process.

The R documentation explains the process of package development in clear terms, and so it is not difficult for users to develop packages for their own work. Since all R users benefit from the work of others, there is a natural tendency to share. This sharing often starts within work groups, but as code becomes robust, it can make sense to share more broadly on CRAN.

Some of packages used in this book are unlikely to be installed by default in the version of R on the reader's computer. To see whether a package is already installed, open an R console and type, e.g.

```
library(oce)
```

If an error message results, then it will be necessary to install `oce`, either with a menu item in a GUI version of R or by writing

```
install.packages("oce")
```

in an R console. Development versions of packages may be installed from source files.

[2]The package count was inferred from web archives at http://wayback.archive.org/.

For packages that are developed on the GitHub social coding website[3], the building process is, e.g.

```
library(devtools)
install_github("oce", "dankelley", "development")
```

It is important to note that many of the examples in this book deal with the `oce` package, but the

```
library(oce)
```

that should precede the use of `oce` has been removed from the code examples in the text, some of which are only a few lines long. However, a `library()` call will be provided for all other packages used in examples.

Code presented in this book uses packages named `boot`, `bvpSolve`, `changepoint`, `DBI`, `deSolve`, `devtools`, `doMC`, `doParallel`, `fields`, `geonames`, `gsl`, `hexbin`, `imputeTS`, `jpeg`, `KernSmooth`, `lattice`, `lmodel2`, `lubridate`, `magrittr`, `MASS`, `microbenchmark`, `mixtools`, `ncdf4`, `oce`, `ocedata`, `party`, `plyr`, `propagate`, `R.matlab`, `ReacTran`, `rgdal`, `rootSolve`, `RSQLite`, `segmented`, `signal`, `smatr`, `tiff`, `vioplot`, `WaveletComp`, and `XML`, plus the packages upon which these depend.

### *2.2.4 Running R*

Starting an `R` console is system-dependent. Many systems provide an icon that can be clicked to launch a GUI-based interface to R (see Appendix B), in addition to a command-line tool that can be used within a terminal.

The startup behaviour of R can be customized with a startup file, e.g. the author has

```
options(digits=7, digits.secs=3)
options(editor="mvim")
options(oceEOS="gsw")
```

in a file called `.Rprofile` in his home directory, to control the number of digits in numbers and times, to set his preferred text editor, and to default to the "Gibbs Seawater" formulation of the seawater equation of state, as opposed to the older UNESCO formulation (see Sect. 5.2.1 and Appendix D). Information on R startup is provided by `help(Startup)`.

There are GUI wrappers that simplify some R operations, including the popular `Rstudio` system, but R is still a computing language at its core, with detailed actions being controlled by textual instructions. These instructions may be typed in an `R` console or entered in a file that is processed by R. Many users combine the two methods, using the console to test provisional approaches, and gradually copying working code into an editor window. Power users are inclined to prefer working in

[3] github.com.

editors that are external to GUI applications, and this is easy with `Rstudio`. The most popular standalone text editors are Emacs (for which the ess mode handles R) and Vim (for which the `vim-r` plugin handles R); each allows transferral of individual lines or blocks of code to a running R session.

Except for trivial work, it is a mistake to use R in a solely interactive mode. Saving code in files is necessary to achieve reproducible results. These files should be self-contained, so that they can be used directly by another researcher with access to the data. There are several ways to use such a file. Within an R session, such a file (called `work.R`, say) can be executed with

```
source("work.R")
```

in an R console, by clicking an icon in an `Rstudio` window, or with

```
R --no-save < work.R
```

or

```
Rscript work.R
```

within a unix-like operating system.

It is easy to automate R analysis considerably on Unix. For example, if the following[4] is put into a file called `Makefile`, then typing `make` in a terminal will run R on all the files with name ending in `.R`, generating a `.out` for each.

```
R = $(wildcard *.R)
OUT = $(R:.R=.out)
%.out: %.R
	R --no-save < $< > $@
all: $(OUT)
clean:
	rm -f *.out
```

Typing `make` again does not call R, because the `.out` files are older than their `.R` sources. If any of the `.R` files are changed, then typing `make` again processes just those files. Typing `make clean` removes the `.out` files, resetting the process. Using `make` speeds up complex multi-staged work, with the extra benefit (compared with interactive analysis) of documenting the entire processing procedure, a key component of reproducible research.

### 2.2.5 Getting Help

Readers who have followed along with the worked exercises will already be comfortable accessing the R help system in a basic way, e.g.

```
help(read.table)
```

explains `read.table()`. As a convenience, this can also be written

```
?read.table
```

[4]The indented lines of the `Makefile` must start with a tab character, not spaces.

If the function (or dataset) name is not known, it may still be found by searching the documentation, e.g.

```
help.search("trig")
```

yields information on a variety of functions relating to trigonometry and

```
help.search("angle", package="oce")
```

reveals `oce` functions or datasets whose documentation contains the word `angle`. The search string must be enclosed in quotes for `help.search()`, but quotes are optional for `help()`.

An overview of the functions in a package is easily obtained, e.g.

```
help(package="oce")
```

yields the documentation at a broad level, and

```
package?oce
```

yields a more specific entry about the package.

The `example()` function is a companion to `help()` that runs any examples that are provided in the documentation for a named function, e.g. `example(plot)` provides a few examples of plotting. This can be very useful, because many documentation pages have pertinent examples.

**Exercise 2.5** Use `help.find()` to find an R package that accesses the www.geonames.org website, and thus locate Halifax, Nova Scotia. (See page 189 for a solution.)

## 2.3 Syntax

### *2.3.1 Expressions*

R can be used as a calculator, simply by typing expressions, and e.g.

```
377 / 120
| [1] 3.141667
```

(Ptolemy's approximation of $\pi$) demonstrates a call-and-response typographic convention used throughout this book. Importantly, the response displayed here was created by R itself, using a system called "sweave" (Leisch 2002). Thus, readers can rest assured that the numerical and graphical examples of the book will work as indicated, apart from formatting nuances.[5]

A confusing aspect is the string "`[1]`" preceding the result of the calculation. This is a counter indicating that the first number on the line is the first in a sequence. These counters are helpful for long sequences. For example, the colon operator (`:`) forms a sequence of values (a "vector", in R parlance) in a stated range, e.g. `1:7`

[5]R displays a prompt before the input, but this is omitted throughout this book.

yields the integers from 1 to 7 and their inverse cubes can be calculated with the power operator (`^`)

```
1 / (1:7)^3
|[1] 1.000000000 0.125000000 0.037037037 0.015625000
|[5] 0.008000000 0.004629630 0.002915452
```

Even this short vector may reveal the usefulness of output counters.

Readers who skipped the parentheses in the previous expression will see a display of hundreds of numbers. This is because the exponential operator is acted upon before the sequence-forming operator. It is said that "`^`" takes precedence over "`:`". By contrast, "`:`" takes precedence over multiplication, as the reader can verify by entering "`1 / 1:7 * 3`" in a console. The precedence of operators is well documented, but it is also easy to forget, so parentheses are recommended to avoid confusion.

On a more aesthetic note, it is good idea to use white space to clarify the notation, especially in complicated expressions. This is somewhat a matter of preference, with some R users inserting a single space before and after every operator, and others inserting space only for certain operators, or in certain circumstances, depending on whether the expression is being entered interactively or being inserted in a document to be shared widely.

Much more could be said about the syntax of R, but it is preferable to continue on with practical examples. For any but the simplest of work, it is certain that users will need to make use of R functions. Of course, given its focus on data analysis, R provides many functions for statistical and numerical work. Most functions can work with single values, or collections of values such as vectors. This latter fact will be a commonplace for Matlab programmers, but those coming to R from C-like languages will find that it greatly simplifies coding, eliminating loops. For example, the Taylor series approximation

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} \tag{2.1}$$

for $x = 1/10$ can be evaluated to five terms with

```
sum(0.1^(0:4) / factorial(0:4))
|[1] 1.105171
```

where `factorial()` calculates $n!$ and `sum()` adds elements.

In addition to such numerical-analysis functions, a conventional set of functions for programming is available. For example, the minimum of a vector is calculated with `min()`, while the index of the smallest element is given by `which.min()`. The corresponding functions for maximum are `max()` and `which.max()`. The range of values in a vector (or any other collection of numerical values) is calculated with `range()`. The boolean function `all()` examines a vector of boolean values (or expressions) and returns `TRUE` if each of these values is `TRUE`. Similarly, `any()` indicates whether any of the values is `TRUE`. All of these functions can handle missing values (see Sect. 2.3.3.6).

Comments may be added to R expressions by placing a # character on the line, so long as that character is not in a quoted character string. Since an empty line is a valid R expression, this also provides a scheme for standalone comments.

```
# Note that atan() returns an angle in radians
4 * atan(1)                               # pi
|[1] 3.141593
```

**Exercise 2.6** Use `cumsum()` to monitor the convergence of the Taylor series for `exp()`. (See page 189 for a solution.)

## 2.3.2 *Variables*

### 2.3.2.1 Variable Assignment

As noted previously, R denotes assignment with the operator "`<-`", e.g.

```
phi <- (1 + sqrt(5)) / 2
```

stores the golden ratio in a variable named `phi`. It is also possible to assign to variables indirectly using `assign()`, e.g.

```
assign("phi", (1 + sqrt(5)) / 2)
```

This second approach is helpful when the variable of interest only becomes known as the R code is being run, as is common in reading data files.

### 2.3.2.2 Variable Evaluation

Writing the name of a variable in an expression that is evaluated causes R to look up the value and use it appropriately,[6] e.g.

```
1 / phi
|[1] 0.618034
```

It is also possible to find a variable by name, using `get()`, e.g.

```
get("phi") - 1
|[1] 0.618034
```

(Aside: comparison of the previous two results reveals a key property of the golden ratio.)

[6] A subtle point is that R does not always look up the values of variables until they are needed. This is related to R concepts of "lazy evaluation" and "promises".

#### 2.3.2.3 Variable Names

Readers with programming experience will find that R accepts common conventions for variable names. For example, a name may contain letters and numbers, the case of letters is significant, etc. However, there are important differences between R variable names and those in other languages.

In some languages, a period in a variable name has a special meaning, e.g. to refer to part of a multi-component object. This is not so in R, where the purpose of a period is both more varied and more confusing (see, e.g., Bååth 2012). In some cases, R treats a period in a variable name just like any other character, e.g. in

```
one.half <- 1 / 2
```

it provides a visual cue that two words are linked. However, in some circumstances, periods have a special meaning relating to the generic functions, a somewhat complex topic to be dealt with in Sect. 2.3.11.6. In the interest of clarity, many R users avoid periods in names, except for generic functions. That raises a question of how best to provide a visual cue of word linkage. A style that is popular in some other languages is to use underlines, e.g.

```
three_quarters <- 3 / 4
```

However, underline was once used to indicate assignment in R (a convention borrowed from S), and some text editors automatically expand this character to `<-`. To avoid confusion, the author uses "camel case" notation, in which a case switch is used to separate words, e.g.

```
fourFifths <- 4 / 5
```

This notation is used in the `oce` package and in other packages, including the popular Bioconductor system for genomic analysis.

*Caution* R provides great freedom in variable names. For example, `assign()` permits a blank character as a variable name, and `get()` recovers its value without complaint. Needless to say, such tricks are best avoided.

*Caution* R does not have read-only variables. For example, even though `pi` is automatically defined by R to be $\pi$, it is valid to write `pi <- 3.41`, and doing so can lead to errors that may be difficult to find later.

#### 2.3.2.4 Variable Scope

Variables in R come into existence when they are assigned a value. Those values are accessible only within what is called the variable "scope". For example, variables created within a function are destroyed when the function returns. This is important, because it means that calling an R function tends not to have side effects on any variables in the enclosing scope. This lack of side effects is an aspect of so-called functional programming, in which a function affects its environment only through its return value.

Sometimes, however, it seems that side effects are the best solution to a coding problem, and the `<<-` operator may be used then, to widen the scope of assignment. See, e.g., Gentleman and Ihaka (2000) and Wickham (2014) for further discussion.

### 2.3.3 *Basic Storage Types*

#### 2.3.3.1 Numerical Types

Integer numbers are denoted with suffix "`L`" (for "long", the size of integer storage in R), e.g.

```
five <- 5L
```

which is revealed as an integer by `storage.mode()`

```
storage.mode(five)
| [1] "integer"
```

As noted previously, sequences of numbers can be generated with "`:`"

```
1:3
| [1] 1 2 3
```

and more general increments are handled with `seq()`, e.g.

```
byTwo <- seq(10L, 20L, 2L)
```

produces the sequence 10, 12, ..., 20 as integers (but the returned vector would be of the "double" storage mode if the third argument were not an integer, or if other argument values required floating-point representation). Integers that map to the contents of the sequence can be produced with `seq_along()`

```
seq_along(byTwo)
| [1] 1 2 3 4 5 6
```

and such mappings are often helpful in working through datasets, such as the stations with an oceanographic section, or the levels within a given station.

Floating-point numbers are indicated with decimal points or exponents

```
twoPi <- 8 * atan2(1, 1)
avogadro <- 6.02e23
```

Complex numbers are denoted with suffix "`i`" on the imaginary part, e.g.

```
x <- 1 + 2i
```

The real and imaginary parts of a complex number are recovered with `Re()` and `Im()`, and related functions perform other requisite tasks. Many R functions accept complex numbers as arguments, e.g.

```
sqrt(1i)
| [1] 0.7071068+0.7071068i
```

```
exp(pi * 1i)
| [1] -1+0i
```

Unfortunately, R does not provide small storage types, such as the 2-byte integers that are used to save space in several acoustic Doppler and satellite data formats. If memory is sufficient, it may be sensible to promote such data to 8-byte integers or floating-point values in R. However, for large datasets it is better to glue together pairs of single-byte elements, as is done in the `oce` package (Chap. 3).

#### 2.3.3.2 Logical Type

R uses `TRUE` and `FALSE` to denote logical values, e.g.

```
waterIsWet <- TRUE
```

Logical negation is achieved by putting `!` to the left of a logical quantity. Writing `|` between two logical quantities yields "or", while `&` yields "and." Note that these operators produce vectors when applied to vectors, while the related operators `||` and `&&` each produce single-valued results.

A common way to construct logical values is through comparison, e.g.

```
x <- seq(-3, 3)
x < 0
|[1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
```

shows how the less-than operator is used; similar operators include `<=`, `>`, `>=`, `==`, and `!=`. (See Exercise 2.7 for notes regarding the use of `==`.) A common use of logical values is for subsetting, e.g.

```
mean(x[x > 0]) # mean of positive values
|[1] 2
```

(This is equivalent to `mean(subset(x, x > 0))`, which is a clearer expression to some users.)

The indexing method also works for assignment, e.g.

```
x[x <= 0] <- NA
mean(x, na.rm=TRUE)
|[1] 2
```

where `mean()` has been given an argument to ignore missing values.

*Caution* Although `TRUE` and `FALSE` can be abbreviated `T` and `F`, this is a poor idea because it leads to confusion, especially in oceanographic work, where "`T`" is a common abbreviation for a temperature.

**Exercise 2.7** Use `==` to find your computer's precision, i.e. the smallest resolvable difference between floating-point values. (See page 190 for a solution.)

**Exercise 2.8** Explain why `all.equal()` is good way to compare floating-point values. (See page 190 for a solution.)

#### 2.3.3.3 Textual (Character) Type

Text strings may be enclosed in single or double quotation marks, and one nested in the other is taken as a literal, e.g.

```
cat("Henry ('Hank') Stommel was a smart man.")
|Henry ('Hank') Stommel was a smart man.
```

Strings may be pasted together with `paste()`

```
paste("Stommel", "(1948)", "is a classic.")
| [1] "Stommel (1948) is a classic."
```

Substrings may be extracted with `substr()`

```
filename <- "atlantic.dat"
substr(filename, 1, 8)
| [1] "atlantic"
```

and `nchar()` gives the number of characters in a string

```
substr(filename, nchar(filename)-2, nchar(filename))
| [1] "dat"
```

Strings may be split into components with `strsplit()`

```
strsplit("Stommel-Arons-Faller", split="-")
| [[1]]
| [1] "Stommel" "Arons"   "Faller"
```

the result of which is a "list," discussed in Sect. 2.3.6. (Splitting is a very useful operation, for many tasks.)

R provides a variety of functions for altering strings, including `sub()`, which replaces the first matched substring

```
sub("a", "A", "atlantic")
| [1] "Atlantic"
```

and `gsub()`, which (by default) replaces all occurrences

```
gsub("a", "A", "atlantic")
| [1] "AtlAntic"
```

Various text encoding schemes may be used for strings, including ASCII, UTF-8, and "byte" forms. People requiring accents in strings probably know how to enter them with key combinations, e.g. the "ä" in "Väisälä" may be obtained with `option-u a` on some systems. Such characters may also be entered with a coding system known as ISO/IEC 8859-1. In this, the code for "ä" is the hexadecimal sequence `E4`. Such sequences may be entered into R strings by prefacing them with the two-character code `\x` and setting the encoding with `Encoding()`, e.g.

```
N <- "V\xE4is\xE4l\xE4"
Encoding(N) <- "latin1"
N
| [1] "Väisälä"
```

Setting the encoding to `"bytes"` can guide the construction of regular expressions, but this can be a tricky business, depending on the encoding used to input data, etc.

It can be wise to stick with one encoding scheme. In `oce`, that scheme is (usually) UTF-8. To see how this works, consider reading CTD files created by Seabird software. If the software is set up to save $\sigma_\theta$ in CNV files, then there will be a column named `sigma-é00`. Then the relevant header line can be isolated in UTF-8 and latin-1 formats with, e.g.

```
f <- system.file("extdata", "d201211_0011.cnv",
package="oce")
readLines(f, encoding="latin1")[54]
```
```
[1] "# name 22 = sigma-é00: Density [sigma-theta,
Kg/m^3]"
```
```
readLines(f, encoding="UTF-8")[54]
```
```
[1] "# name 22 = sigma-\xe900: Density [sigma-theta,
Kg/m^3]"
```

Although the first form has the advantage of displaying the line as a text editor might, the UTF-8 form may be more convenient for programming, e.g. `read.ctd.cnv()` finds such entries in headers with

```
n <- gsub("^# name [0-9][0-9]* = (.*):.*$",
          "\\1", h, ignore.case=TRUE, useBytes=TRUE)
if (1==length(grep("^sigma-\xe9[0-9]{2}$", n,
useBytes=TRUE)))
```

Readers with programming experience will have no trouble reading regular expressions in the previous lines of code. Others might benefit from a brief sketch (consulting the R help system for details). Consider the strings `"Atlantic"` and `"Pacific"`, which may be joined into a vector with `c()`. To see which contains the letter `"t"`, write

```
grep("t", c("Atlantic", "Pacific"))
```
```
[1] 1
```

If no element had contained the pattern, `grep()` would have returned a zero-length vector, so the expression `0 < length(grep(p, x))` tests whether any element of vector `x` contains pattern `p`.

Special characters can be inserted into patterns, achieving useful outcomes. The character `"^"` stands for the start of a string, and `"$"` stands for the end of the string, as in the CTD example above. A period can stand for any character. An asterisk after a pattern indicates that the pattern may appear zero or more times. Alternative characters are enclosed square brackets, and a dash can be used to create a sequence. Substrings enclosed within parentheses can be reused later, as is done in the `gsub()` example, where `n` is assigned the value of the first (and only) marked substring.

Special difficulties arise with human-entered text, because typos are common, and users in different locales might spell differently. R helps in such cases with fuzzy string matching provided by `agrep()` and `adist()`.

**Exercise 2.9** A directory contains Biosonics echosounder files, with names indicating start times, with four digits for year, two for month and two for day, followed by an underline and then two digits for hour, two for minute, and two for second, ending with `.dt4`. Use `grep()` to isolate data starting between 1100 h and 1500 h on June 28th, 2008. (See page 190 for a solution.)

### 2.3.3.4 Binary (Raw) Type

R has a byte-level type that it designates as "raw." Positive integers in the range from 0 to 255 can be converted to this form with `as.raw()`, e.g.

```
as.raw(1:16)
| [1] 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10
```

and `as.raw()` also handles floating-point values in the allowed range, by first coercing its input to integer form. Raw constants follow a C-like notation, e.g. `0x0f` or `0x0F` for $0000\ 1111_2$, and C notation is also used for binary operations, with the conventional operators `|`, `&` and `!`.

A good way to handle binary formats is to use `readBin()` to read the whole file into a `raw` vector, and then to work though that vector with `readBin()` on smaller chunks of data. Readers who have struggled with files in which the endianness shifts from entry to entry will appreciate the fact that `readBin()` has an argument specifying the endian nature of the item being decoded.

### 2.3.3.5 Time Types

A simple approach to dealing with time in R is to treat it as a numerical value that has meaning to the user, but not to R. For example, a numerical model might output time in days or years, with the expectation being that an analyst will take the unit into account when making calculations and plotting results. However, a more systematic approach is required for data.

Decoding times recorded in notebooks can require answering questions such as whether 2018/2/10 is early in the year or late in it, whether 8 o'clock is in the morning or the evening, and when (or whether) daylight-savings time commenced in a given year, in a given jurisdiction ... and realizing that such answers might not hold across pages, even those written by a single person.

Some details are more universal, and R handles them well by itself. These include leap years and leap seconds, along with the thornier matter of timezones. (The IANA timezone database[7] spans over 20,000 lines.) Readers interested in such things should consult Ripley and Hornik (2001) for an introduction to how R handles dates and times.

R has two schemes for representing time types. In one, time is represented by a numerical value representing the year, another representing the month, etc. This scheme is used by `as.POSIXlt()`, which converts text strings into times. In the second scheme, used by the analogous function `as.POSIXct()`, time is represented by a single numerical value that measures the interval since a reference time. Both schemes have additional information on timezone, etc.

Conversion between these two schemes is trivial, so there is little need to discuss both. The single-number scheme will be the focus here. Time objects created by

[7] www.iana.org/time-zones.

`as.POSIXct()` store the number of seconds since the start of the year 1970.[8] Prior times have negative values, and fractional seconds are handled by the use of floating-point storage.

For example, one minute past the origin time is

```
t0 <- as.POSIXct("1970-01-01 00:01:00", tz="UTC")
```

This may be displayed as a text string or a numeric value

```
t0
|[1] "1970-01-01 00:01:00 UTC"
as.numeric(t0)
|[1] 60
```

Note the use of `tz` to set the timezone. If `tz` is not given, R will use a local time zone. This default can lead to highly undesirable results, such as an R program producing different results when run in different regions, so a `tz` value should always be supplied. However, even this is problematic, because of ambiguities in timezone notation. For example, AST means Atlantic Standard Time to the author, but it might mean Alaskan Standard Time, or Afghanistan Standard Time, to a reader. A good solution is to specify timezones by region and city, e.g.

```
as.numeric(as.POSIXct("1970-1-1 00:00:00",
                      tz="America/Halifax"))
|[1] 14400
```

shows that local standard time in the author's city is 4 h "behind" `UTC`. (Use `help(timezones)` to learn more about timezones.)

Handling alternative representations of times is simplified with the `format` argument, e.g.

```
as.POSIXct("Jan 1, 1970 00:01:00", tz="UTC",
           format="%b %d, %Y %H:%M:%S")
|[1] "1970-01-01 00:01:00 UTC"
```

handles a common format used in non-technical writing; note that `%b` is an abbreviated month name, `%d` is decimal day, `%Y` is year including century, `%H` is hour, `%M` is minute, and `%S` is second. See the documentation for `strptime()` for the details of the coding scheme, which includes some standardized forms, e.g.

```
as.POSIXct("1917-12-06 09:04:35", format="%F %T",
           tz="America/Halifax")
|[1] "1917-12-06 09:04:35 AST"
```

expresses the time of the Halifax explosion in ISO 8601 format.

Using `as.POSIXlt()` is very similar to this. Working with numerical values for year, month, etc., is also easy, with `ISOdatetime()`.

All of the above relates to the `base` package. The `lubridate` package (Grolemund and Wickham 2011) also has functions that parse common formats, and e.g.

[8] Alternative time origins may be specified to `as.POSIXlt()`, and this can be helpful in working with times represented in other systems such as SPSS and SAS.

```
library(lubridate)
ymd("1970 Jan 1")
| [1] "1970-01-01"
ymd("1970-01-01")
| [1] "1970-01-01"
```

demonstrates that it is quite adept at decoding formats.

Oceanographers use a wide variety of numerical schemes for time, so `oce` provides `numberAsPOSIXct()` for inferring times from the Unix, Matlab, SAS, SPSS, GPS and Argo numerical schemes.

The above has dealt with single values, but of course R also handles vectors of times. For example, sequences of times may be created with `seq()`

```
seq(t0, by="1 min", length.out=2)
| [1] "1970-01-01 00:01:00 UTC" "1970-01-01 00:02:00 UTC"
```

or simply by adding a sequence of numerical values to a time, e.g.

```
t0 + 60 * seq(0, 1)
| [1] "1970-01-01 00:01:00 UTC" "1970-01-01 00:02:00 UTC"
```

Despite the strong support for time types in R, there are many ways to get into trouble if care is not taken. A few hints may help.

1. Always set the timezone, ideally to `tz="UTC"`.
2. Matlab users should note that Julian days start at 0 in R, e.g.
   ```
   as.numeric(julian(as.POSIXct("1970-1-1", tz="UTC")))
   | [1] 0
   ```
3. Be careful when assembling multiple times with `c()`, e.g.
   ```
   as.POSIXct(c("1970-1-1", "1970-1-1 0:0:1"))
   | [1] "1970-01-01 AST" "1970-01-01 AST"
   as.POSIXct(c("1970-1-1 0:0:0", "1970-1-1 0:0:1"))
   | [1] "1970-01-01 00:00:00 AST" "1970-01-01 00:00:01 AST"
   ```
   reveals that R selects the printing format based on the first element.
4. Be aware that `format()` and `strftime()` handle unspecified timezones differently, e.g.
   ```
   t0 <- as.POSIXct("1970-01-01 00:01:00", tz="UTC")
   format(t0)
   | [1] "1970-01-01 00:01:00"
   strftime(t0)
   | [1] "1969-12-31 20:01:00"
   ```
   (Note that setting `tz="UTC"` in the `format()` and `strftime()` calls makes them yield identical results.)
5. Limit your choice of functions, and study the documentation well, to avoid surprises such as the one just mentioned.
6. Try using `attr()`, `attributes()` and `as.numeric()` to find the roots of any problems that may arise.

#### 2.3.3.6 Missing Values and Other Special Values

Conventional notation is used for numerically problematic values in R, e.g.

```
c(1/0, -1/0, asin(3))
│[1]  Inf -Inf  NaN
```

R has a special code for missing values, such as might result from instrument malfunction. These are indicated with `NA`, which could be read as "not appropriate." A missing value can take the place of most R items, e.g.

```
c(1, 2, NA, 4)
│[1]  1  2 NA  4
```

```
c("inshore", NA, "offshore")
│[1] "inshore"  NA         "offshore"
```

The provision of missing values in R reveals that it is a language that grew from the demands of research. Other systems such as Matlab reuse `NaN` or some other code for a missing value, blurring meaning. Many R functions detect missing values and offer ways to control how they are interpreted, e.g.

```
mean(c(1, NA, 2))
│[1] NA
```

```
mean(c(1, NA, 2), na.rm=TRUE)
│[1] 1.5
```

There are also functions for selecting data, e.g.

```
mean(na.omit(c(1, NA, 2)))
│[1] 1.5
```

In many cases, simply omitting missing data will be sufficient, but sometimes this is not an option, e.g. skipping data in a time series will yield problems with computing spectral properties. An entry to the general literature about handling missing data is provided by Horton and Kleinman (2007). Readers can also find guidance in their own branches of the oceanographic literature, with treatments often being keyed to instrument type, e.g. Sect. 5.9.2.2 presents ideas for dealing with acoustic-Doppler velocimeter (ADV) data.

R uses the symbol `NULL` to indicate an extant but empty quantity. This is sometimes used in function arguments. A common use in processing is in the item-by-item construction of vectors, as explained in Sect. 2.3.4.

R provides several functions for testing whether numbers are problematic, e.g. `is.nan()` tests for `NaN`, `is.infinite()` tests for `Inf`, and `is.null()` tests for `NULL` values. In many cases, the best test is `is.finite()`, which returns `TRUE` only if the argument is not `Inf`, not `NaN`, and not `NA`.

*Caution* Matlab data files tend to use `NaN` for missing value, so that Matlab files that are converted to R need an extra step, e.g.

```
x <- readMat("x.mat")
x[is.nan(x)] <- NA
```

## 2.3.4 Vectors

Vectors hold sequences of values, and e.g.

```
x <- 3
is.vector(x)
| [1] TRUE
```

reveals that even single values are vectors.

A vector can contain entries of any atomic mode, meaning `"logical"`, `"integer"`, `"numeric"`, `"complex"`, `"character"` or `"raw"`, or of either `"expression"` or `"list"` mode. However, a vector cannot contain an admixture of these modes. Applications requiring the grouping of items of dissimilar modes should use lists (Sect. 2.3.6), instead of vectors.

A few examples will suffice to show how to use vectors in practical work. A vector of numerical values may be constructed in a number of ways, e.g. with the colon operator "`:`" for integers

```
threeStooges <- 1:3
```

with a sequence function for more general values

```
thirds <- seq(0, 1, length.out=4)
```

with the repeat function

```
fourScore <- rep(20, 4)
```

and with the collection function

```
irrational <- c(pi, exp(1))
```

The last of these methods also works for strings

```
stoogeNames <- c("Larry", "Curly", "Moe")
```

Vector elements are accessed with a square-bracket "`[`" notation patterned on C, although R indexes the first element at `1`, not `0` as in C, e.g.

```
stoogeNames[2]
| [1] "Curly"
```

and multiple elements may be accessed at the same time, by providing a vector of indices within the square brackets, e.g.

```
stoogeNames[c(1, 3)]
| [1] "Larry" "Moe"
```

A boolean vector may also be used to access elements. For example, the most colourful of the seven seas of Medieval literature are

```
seas <- c("Mediterranean", "Adriatic", "Arabian",
          "Black", "Caspian", "Persian", "Red")
seas[c(FALSE, FALSE, FALSE, TRUE, FALSE, FALSE, TRUE)]
| [1] "Black" "Red"
```

In the above examples, the vector contents are specified in code, so R can set up storage before assigning values. However, in some cases, vector length is discovered by data inspection or user instruction. If the requisite vector length is known before determining contents, storage may be allocated with e.g.

```
depths <- vector("numeric", 100)
```

with values entered later. If length is not known in advance, a vector can be constructed incrementally (less efficiently), starting with an empty vector

```
lengths <- NULL
```

and then, as each new length `L` is found, append it to the list, e.g.

```
lengths <- c(lengths, L)
```

Another approach is to assign past the end of the vector, e.g.

```
lengths[2] <- 4
```

appends an additional item to the vector. Starting in 2017, R developed a new scheme for memory allocation beyond the end of a vector, yielding great improvements to the processing speed of this method, making it preferable to the `c()` method in many cases.

*Caution* Binary operations between vectors follow a recycling rule that permits combination even if the vectors are of unequal length. This works by cycling through the elements of the shorter vector as needed, e.g. in

```
1:6 + c(1, 0)
|[1] 2 2 4 4 6 6
```

the second vector is expanded to `c(0, 1, 0, 1, 0)` before the addition. This behaviour is very helpful, but can be surprising to programmers coming from languages that disallow operating on mismatched objects.

**Exercise 2.10** Use `floor()` to select even integers from a vector. (See page 190 for a solution.)

## 2.3.5 Arrays and Matrices

As with vectors, the concept of a matrix should be familiar to most readers. A simple interpretation is a grid of values, as one might construct by writing a number (or string, etc.) in the boxes on a square-ruled sheet of paper. An array is a more general item that can have higher dimensions. For example, a gridded sea-level field $\eta = \eta(x, y)$ might be stored in a matrix, while a 3D `array` would suit a gridded temperature field $T = T(x, y, z)$.

As with vectors, arrays are fairly flexible in terms of their contents, but any given array can contain only one type, and that type must be single-valued (i.e. it is not possible to store an array in the cell of another array).

To see if an item is a matrix, use `is.matrix()`, and to see whether it is an array, use `is.array()`. Coercion rules usually ensure that arrays alter themselves if the type of an element alters, e.g. if a floating-point number is inserted into a matrix of integers, the rest of the numbers are converted to floating point values.

The `matrix()` function can construct matrices from vectors, e.g.

```
m <- matrix(1:6, nrow=2)
m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

shows how to fill a matrix by columns; use `byrow=TRUE` to fill by rows. Matrices can also be created by combining columns with `cbind()` or rows with `rbind()`.

#### 2.3.5.1 Algebra

Matrix operations may be carried out in an element-by-element manner with operators such as `+`, `*`, etc. For example, each cell in `A*B` is the product of corresponding cells in `A` and `B`. (Note that a recycling rule applies to matrices, as for vectors.) Matrix multiplication is denoted by the `%*%` operator, and the usual linear-algebra rules control whether `A%*%B` can succeed, depending on the matrix dimensions. Those dimensions may be recovered or set with `dim()`. Other common matrix operations include: transpose with `t()`, determinant calculation with `det()`, singular value decomposition with `svd()`, QR decomposition with `qr()`, eigenanalysis with `eigen()`, inversion with `solve()`, generalized matrix inversion with `ginv()` from the `MASS` package, etc.

#### 2.3.5.2 Indexing and Subsetting

Subsets of matrices can be extracted with `[`, e.g. with `m` as defined previously,

```
m[1,]
[1] 1 3 5
```

shows the first row, and

```
m[-1,]
[1] 2 4 6
```

shows the results of deleting that row.

The labelling in the results shown above indicates that R has converted the column and row matrices into vectors. This behaviour can be disabled by supplying the `drop` argument, e.g.

```
m[1,,drop=FALSE]
     [,1] [,2] [,3]
[1,]    1    3    5
```

Note the need for two commas here, because the second argument to `[` is an index; see `help("[")` for more.

*Caution* The R conversion to vectors should be kept in mind when converting matrix-oriented code from other languages.

#### 2.3.5.3 Reshaping

In addition to reporting the dimensions of an item, `dim()` can be used on the left-hand side of an assignment expression, to set the dimension, e.g.

```
dim(m) <- c(1, 6)
m
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    2    3    4    5    6
```

This reveals that matrices and arrays are stored in column order, which proves convenient when connecting R to C, C++ or Fortran.

#### 2.3.5.4 Storage

R stores matrices and arrays as vectors, saving dimensions with attributes

```
attributes(m)
```

```
$dim
[1] 1 6
```

The dollar sign in the result is an indication that the attributes are stored in an item named `dim` within a list. (See Sect. 2.3.6 for more on lists.)

#### 2.3.5.5 Example: A Rotation Matrix

Suppose moorings are placed at locations drawn in Fig. 2.3 (left) with

```
E <- seq(0, 0.5, 0.1)
N <- seq(0, 0.5, 0.1)
plot(E, N, xlim=c(0,1), ylim=c(0,1), asp=1)
```

and that these locations are to be expressed in an $xy$ coordinate system rotated 45° anti-clockwise of geographic. This is expressed with rotation matrix

$$\mathbf{R} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \tag{2.2}$$

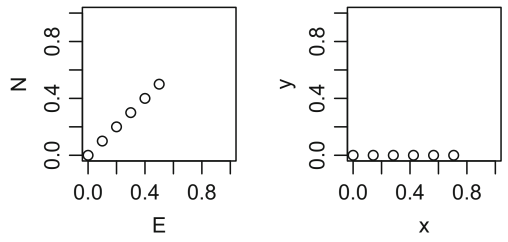

**Fig. 2.3** Rotation from $E$-$N$ coordinate system to $x$-$y$ coordinate system.

with $\theta = 45°$, which may be expressed as

```
theta <- 45 * pi / 180
S <- sin(theta)
C <- cos(theta)
R <- matrix(c(C, S, -S, C), byrow=TRUE, nrow=2)
```

so that matrix multiplication yields $x$ and $y$ in columns

```
xy <- R %*% rbind(E, N)
plot(xy[1,], xy[2,], xlim=c(0,1), ylim=c(0,1), asp=1,
     xlab="x", ylab="y")
```

as in the right panel of Fig. 2.3.

**Exercise 2.11** Write a function to find the indices of the maximal value of a matrix. (See page 191 for a solution.)

## 2.3.6 *Lists*

The restriction that any given vector, matrix or array can hold just one data type yields efficiencies, but some programming situations call for collections of disparate items. In R, such collections are called lists, and they may be constructed with `list()`, e.g.

```
hex <- list(name="six", value=6, constituents=c(2,3))
```

or with `as.list()`. If the list items have names, they may be retrieved with "`$`" extraction operator or "`[[`" (this second operator being overloaded to look within `oce` objects; see Sect. 3.3).

```
hex$name
| [1] "six"
```

```
hex[["value"]]
| [1] 6
```

Regardless of whether list items are named, they may be also recovered by numerical index, with single-bracket notation

```
hex[3]
| $constituents
| [1] 2 3
```

retaining item names and double-bracket notation

```
hex[[3]]
| [1] 2 3
```

discarding names.

As will be explained in Chap. 3, the `oce` package stores data as lists, because so many oceanographic data combine vectors with matrices, raw bytes with decimal numbers, etc.

**Exercise 2.12** Show how to access a list within a list. (See page 191 for a solution.)

### 2.3.7 *Factors*

Factors designate categorical data, and they have wide applications to oceanographic work, for such things as species names, etc. Factors simplify data analysis because many R functions handle them in useful ways. The ideas may be illustrated with air-sea drag data reported by Garratt (1977).

```
data(drag, package="ocedata")
```

The values of the drag coefficient $C_D$ stored as `Cd` within this dataset were inferred from either a “profile” method or an “eddy correlation” method. A factor is a natural way to store this information, as is revealed with

```
str(drag) # results not shown
```

The list of categories is provided by `levels()`

```
levels(drag$method)
```
```
[1] "profile" "eddy"
```

while `as.numeric()` reveals the method used for each datum

```
as.numeric(drag$method)
```
```
 [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2
     2 2 2 2 2
[29] 2 2 2
```

These numerical values are convenient for coding the data with symbols, as in the left panel of Fig. 2.4.

```
code <- as.numeric(drag$method)
levels <- levels(drag$method)
plot(drag$U, drag$Cd, pch=code,
     xlab="Wind Speed [m/s]", ylab=expression(C[D]))
legend("topleft", pch=1:length(levels), legend=levels)
```

As an example of R plotting functions that handle factors in special ways,

```
boxplot(Cd ~ method, data=drag, notch=TRUE)
```

yields the box plots shown in right panel of Fig. 2.4 (see Tukey (1977) for more on box plots and other methods of exploratory data analysis).

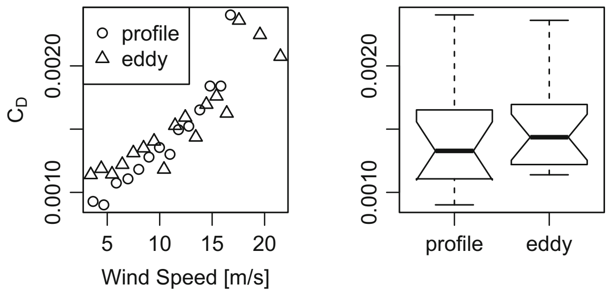

**Fig. 2.4** Left: Symbol coding with factors. Right: boxplot coded with factors

Factors are also given special consideration by several non-graphical functions, and this lowers the effort of some everyday tasks, such as regression and the analysis of variance (Sect. 2.5.6).

More generally, factors provide a simple way to subdivide datasets into components to be processed individually, somewhat akin to the map-reduce method (Lämmel 2008) popularized by Google. For example,

```
dragSplit <- split(drag, drag$method)
```

splits `drag` into a list with elements named `profile` and `eddy`, making it easy to study differences between the subtypes, e.g. a *t* test

```
t.test(dragSplit$profile$U, dragSplit$eddy$U)$p.value
[1] 0.4587837
```

might offer some insights.

For the `drag` data, the identification of `method` as a factor is straightforward. Factors can also be helpful with continuously varying data that can be subdivided meaningfully, as with the constructed $O_2$ profile[9] drawn as Fig. 2.5 with

```
p <- seq(0, 1000, 10)
O2 <- 280 - 1.7 * p * exp(-p / 200)
plot(O2, p, ylim=rev(range(p)), type="l")
```

Here, `p` is pressure in decibars and `O2` is oxygen concentration in μmol/kg. Figure 2.5 shows the profile constructed as follows.

If there were a need to find the thickness of the layer with $O_2$ concentration under 180 μmol/kg, a first step might be to create a factor

```
f<-factor(O2<180,levels=c(TRUE,FALSE),labels=c("Low",
"High"))
```

(There is no requirement to supply `levels` and `labels`, but doing so reduces the chance of errors of interpretation.) To find the thickness of the layer in question, split the data

```
psplit <- split(p, f)
```

creating a list with elements named `Low` and `High`.

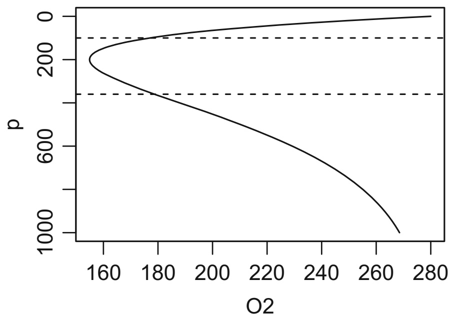

**Fig. 2.5** Oxygen profile, with minimum region indicated

[9]This profile results from a nonlinear regression (Sect. 2.5.5.2) of the oxygen profile at station 112 of the `section` dataset in the `ocedata` package.

The pressure limits of the low-oxygen zone can be added to Fig. 2.5 with

```
abline(h=range(psplit$Low), lty=2)
```

**Exercise 2.13** Use `factor()` and `split()` to identify the months in which Keeling $CO_2$ signal rises and falls. (See page 191 for a solution.)

### *2.3.8 Data Frames*

Data frames are a complement to matrices and lists, and they are very important in R. Indeed, `help(data.frame)` states that data frames are "used as the fundamental data structure by most of R's modeling software."

One way to think of data frames is as matrices that may contain columns of different storage types. This extension can be very useful for general data, e.g. in listing information on two leaders of oceanography

```
leaders <- data.frame(person=c("Munk", "Stommel"),
                      born=c(1917, 1920))
```

The `print()` function displays an entire data frame

```
print(leaders)
   person born
1    Munk 1917
2 Stommel 1920
```

while `str()` provides an overview

```
str(leaders)
'data.frame':	2 obs. of  2 variables:
 $ person: Factor w/ 2 levels "Munk","Stommel": 1 2
 $ born  : num  1917 1920
```

that reveals that R created the first column as a factor.

Data frame columns may be selected in various ways:

```
leaders[1]    # or leaders["person"]
   person
1    Munk
2 Stommel
```

```
leaders[[1]]  # or leaders[["person"]]
[1] Munk    Stommel
Levels: Munk Stommel
```

```
leaders$person
[1] Munk    Stommel
Levels: Munk Stommel
```

The `$` style is convenient for interactive work, but the others are required for programs that need to use `names()` to find names at run time.

It is easy to modify data frames, e.g.

```
leaders$born<-c(as.Date("1917-10-19"),as.Date
("1920-09-27"))
```

alters the birth dates and

```
leaders$institution <- c("SIO", "WHOI")
```

adds a new column for workplace.

**Exercise 2.14** Construct a data frame with column x containing numbers from 0 to $2\pi$, and y containing $\sin x$. (See page 192 for a solution.)

**Exercise 2.15** Append volume to the oceans dataset from the ocedata package. (See page 192 for a solution.)

**Exercise 2.16** Suppose a data frame contains CTD data for a series of stations, with columns for salinity, temperature, pressure, and station ID. Use split() and factor() to create a list with one element per station. (See page 193 for a solution.)

## 2.3.9 Contingency Tables

Tabulation may be accomplished with functions table() or tabulate(). The first of these returns an object of class "table", which is essentially an integer vector with names, while the second returns a integer without names.

A handy use of table() is to count missing values, e.g. the number of missing phosphate data in the section dataset from the ocedata package can be found as follows:

```
data(section, package="oce")
table(sapply(section[["phosphate"]], is.na))
```
```
FALSE  TRUE
 2817    24
```

where sapply() applies is.na() to all the $PO_4$ measurements in the section.

## 2.3.10 Conditional Evaluation

R provides conditional evaluation with if statements, e.g. the following checks whether the sun is above the horizon.

```
if (sunAzimuth > 0)
    cat("daytime\n")
```

In this case, a single action is to be performed. As in many other computing languages, multiple actions are enclosed with braces, e.g.

```
if (ocean == "Atlantic") {
    area <- 1e8 # km^3
    depth <- 4  # km
    cat("Ocean volume:", area * depth, "km^3\n")
}
```

It is also possible to specify a statement (or block of statements) to be executed if the tested condition is false, e.g.

```
if (depth < 100) cat("shelf\n") else cat("deep\n")
```

which also illustrates that blocks can be put on one line.

The `if` statement acts as an expression that returns a value, e.g.

```
sedimentType <- if (L < 62.5e-6) "mud" else "sand"
```

distinguishes between two sediment categories; nested `if`s might be used if samples might contain gravel, etc.

In the previous examples, only a single value was being tested. Multiple values can be handled with loops (Sect. 2.3.12), but it is usually better to use `ifelse()` to improve speed and clarity.

The first argument to `ifelse()` is a vector, matrix, etc., of logical values. The second is a corresponding vector (etc.) of values to be selected if a particular test value is `TRUE`, and the third contains values for `FALSE` conditions. For example, a matrix of depths

```
H <- matrix(c(10, 50, 90, 200), nrow=2)
```

can be categorized by domain with

```
ifelse(H < 100, "shelf", "deep")
```

```
     [,1]    [,2]   
[1,] "shelf" "shelf"
[2,] "shelf" "deep" 
```

### *2.3.11 Functions*

#### 2.3.11.1 Built-In Functions

R provides functions for common tasks related to plotting, statistics, and numerical analysis. Many specialized mathematical functions are provided as well, either in the base system or in packages. Some searching may be required to find the best package for a given task. For example, the formula for the perimeter of an ellipse involves the complete elliptic integral of the second kind, and this is available as `ellint_Ecomp()` in the `gsl` (GNU scientific library) package,[10] e.g. the perimeter of an ellipse with radii 1 and 2 is

[10] See http://www.gnu.org/software/gsl/ for more on GSL.

```
library(gsl)
a <- 2
b <- 1
4 * a * ellint_Ecomp(sqrt((a^2-b^2)/a^2))
|[1] 9.688448
```

#### 2.3.11.2 Defining Functions

The syntax for defining functions is similar to that for defining variables, using the assignment operator "<-". For example, ship speeds reported in knots can be converted to metres per second with a function defined as

```
knotToSI <- function(k) (1852/3600) * k
```

This assigns to the symbol `knotToSI` a function that takes a single argument named `k`. The expression following the list of arguments is the value to be returned by the function.

As with `if` statements, functions with more than one line need braces, e.g.

```
knotToSI <- function(k) {
    factor <- 1852/3600
    factor * k
}
```

The rule is that the last item evaluated in the function provides the return value. It is also possible to return a value specifically with `return()`.

Of course, the purpose of this function is to work with numbers. Calling this with a non-numeric argument will generate an error. Good functions will check for erroneous argument values, and they will be flexible enough to handle a range of conditions. For example, `knotToSI()` might be extended to handle marine-telegraph specifications such as `k="dead slow"`, by inserting a conditional that uses `is.numeric()` and adjusts `k` accordingly.

Function arguments may have default values, e.g.

```
knotToSI <- function(x, modern=TRUE)
    if (modern) x * (1852/3600) else x * (1853.248/3600)
```

permits the use of nautical miles as defined in the US prior to 1954, the year when the nation adopted the international standard.

Functions may check to see whether an argument had been supplied at call time, using `missing()`, e.g.

```
knotToSI <- function(x, modern=TRUE)
{
    if (missing(x))
        stop("must supply x")
    # rest of function as in previous examples
}
```

reports an error if nothing can be calculated; another choice might be

```
    if (missing(x))
        return(NA)
```

The argument list may contain ellipses (`...`) to indicate that there may be additional arguments that may be passed on to children, e.g.

```
indicateTheOcean <- function(ocean, ...)
    mtext(ocean, ...)
indicateTheOcean("Pacific", col="blue")
```

uses `mtext()` to draw text in a plot margin, using a blue colour. Importantly, `indicateTheOcean` does nothing related to colour; it merely provides the calling function with an opportunity to specify arguments such as `col` along to `mtext()`.

Many useful R functions have a large number of arguments, which would make it easy to make errors in calling the functions, but for the fact that R permits named arguments. This permits the skipping of arguments, and the specification of arguments out of order, e.g.

```
f <- function(x, y, z, u, v, w) ...
f(z=Z, w=W)
```

would make sense in situations not requiring $x$, $y$, $u$ and $v$.

**Exercise 2.17** Devise a function using `ifelse()` that returns the tangential velocity in a Rankine vortex. (See page 193 for a solution.)

#### 2.3.11.3 Recursive Functions

Functions may be recursive, meaning that they can call themselves. Some algorithms are defined elegantly in such terms. For example, the greatest common denominator of two integers can be computed with

```
gcd <- function(a,b) if (b == 0) a else gcd(b,a %% b)
```

where the `%%` operator computes the remainder of division of two integers. However, as in other languages that permit recursion, there can be a computational penalty for mimicking elegant mathematical recursion in code (see Appendix E for some notes on handling computationally demanding tasks).

#### 2.3.11.4 Functions as Arguments to Other Functions

Many important R functions take other functions as arguments. This scheme can be valuable, as may be illustrated with examples of three commonly used functions in this class: `uniroot()`, `optimize()` and `lapply()`.

First, consider the case of root-finding. Suppose the task is to find the temperature at which water of practical salinity 35 has density 1025 kg/m$^3$ at atmospheric pressure. To do this, define the function

```
densityMismatch <- function(T)
    swRho(rep(35, length(T)), T, 0, eos="unesco") - 1025
```

which should have a root at the desired temperature.[11] A search interval for the root may be found with, e.g.

```
T <- seq(0, 30, length.out=100)
plot(T, densityMismatch(T), type="l")
abline(h=0, col=2)
```

which produces a graph (not shown) verifying that this function indeed has a zero in the interval 0°C< $T$ < 30°C, so

```
uniroot(densityMismatch, interval=c(0, 30))$root
[1] 19.08284
```

provides the desired temperature.

Optimization provides a second example of functions calling functions. For the multivariate case, R provides `optim()`, plus related functions such as `nlm()` and `nls()`. There are subtle details to multivariate optimization, and so a better preliminary illustration is provided by the one-dimensional case, which is handled by `optimize()`. For example, suppose the goal is to find the temperature that yields maximum fresh-water density at sealevel pressure. With salinity pressure fixed, we may write a univariate function:

```
dens <- function(T)
    swRho(salinity=0,temperature=T,pressure=0,eos=
    "unesco")
```

so that the desired temperature may be computed with

```
optimize(dens, interval=c(0, 10), maximum=TRUE)$maximum
[1] 3.980739
```

where the `maximum` value overrides the default, which is to seek a minimum.

**Exercise 2.18** Use `uniroot()` and `coriolis()` from the `oce` package, to find the critical latitude at which the Coriolis parameter $f$ matches the M2 tidal frequency (12.4206 hour period). (See page 193 for a solution.)

**Exercise 2.19** Use `uniroot()` to create a function that calculates linear gravity wave speed as a function of period. (See page 193 for a solution.)

#### 2.3.11.5 Function Closures

Function closures provide a mechanism for binding functions with parameters, making it easy to create suites of related functions that are made distinct by those parameters. Although this methodology may be unfamiliar to some readers, it is worth learning because it can improve code simplicity and reusability. The scheme uses functions that create other functions, e.g.

[11]Note the use of the UNESCO equation of state here; with the GSW equation, longitude and latitude would also have to be supplied; see Sect. 5.2.1 and Appendix D.

```
exponent <- function(p)
    function(x) x^p
```

makes a function that creates functions for exponentiation:

```
cubeRoot <- exponent(1/3)
```

creates a cube-root function, called as `cubeRoot(8)`, for example.

**Exercise 2.20** Create a function closure for individualized calibration of Seabird thermistors. (See page 194 for a solution.)

#### 2.3.11.6 Generic Functions

Items in R have an attribute named `class`, which is revealed by `class()`

```
atl <- "Atlantic"
class(atl)
| [1] "character"
```

and changed with the same function

```
class(atl) <- "ocean"
class(atl)
| [1] "ocean"
```

A form of object orientation[12] is achieved in R via "generic" functions that are replaced by specialized functions according to the class of the first argument. The syntax is simple, with the specialized function being named as the desired generic function, followed by period and then the class name, e.g.

```
print.ocean <- function(x)
    cat("My favourite ocean is the", x, "\n")
```

defines a function to be used if `print()` is called with an `ocean` object.

```
print(atl)
|My favourite ocean is the Atlantic
```

Generic functions provide users with specialized functions without demanding that they know the individualized names of those functions, or even the classes of the objects under consideration. R provides specialized variants for such key functions as `print()`, `summary()` and `plot()`, and this greatly simplifies processing, as users tend to rely on the default of `plot(x)` doing something sensible, no matter what `x` may be.

A list of generic functions can be retrieved with, e.g., `methods(plot)`, and help on specialized functions is found by appending the class name, e.g. `help(plot.ts)` yields help on the time-series plot function.

[12]Readers who wish to learn more details of object orientation in R might start with Chambers (2008) or Wickham (2014).

### 2.3.11.7 Function Pipelines

In recent years, efforts have been made to implement an R analogy to Unix pipelines, with a notation for chaining function calls. For example, the mean sine of the integers from 1 to 10 may be computed conventionally in R with

```
mean(sin(1:10))
```

and the `magrittr` package lets this be written

```
library(magrittr)
1:10 %>% sin %>% mean
```

where `%>%` is a binary operator that takes the item on its left and supplies it as the first argument to the function on its right.

The scheme also works with user-generated functions, including anonymous function, which can use "`.`" as a place-holder for the passed value, e.g. $\sum_0^{10} (1/2)^n$ becomes

```
1:10 %>% {(1/2)^.} %>% sum
|[1] 0.9990234
```

Parentheses permit extra arguments to be supplied, e.g.

```
1:10 %>% sin %>% mean(trim=0.10)
```

uses a trimmed mean.

Some clarity is lent to complex operations by using line breaks, e.g.

```
1:10 %>%
    sin %>%
    mean(trim=0.10)
```

partly because of the space provided for comments on the individual steps.

### 2.3.11.8 Operators as Functions

In R, operators are functions. Thus, when the R parser encounters the division operator, it calls a function named "/", so that `1/2` is equivalent to

```
`/`(1, 2)
|[1] 0.5
```

Readers with programming experience will see how this relates to the previous section, and might start using it for wider purposes. However, some care is required, e.g. the following shows how to turn addition into subtraction

```
`+` <- `-`
3 + 2
|[1] 1
```

Note that the original meaning of + is recovered with

```
rm(`+`)
```

### 2.3.12 *Loops*

R has several styles of looping structures that provide for repeated calculation. The choice of structure is sometimes dictated by the problem at hand, and sometimes by personal style.

In a `for` loop, an index term is set to each value in a sequence, e.g.

```
for (i in 1:10)
    cat(2^i, ' ')
|2  4  8  16  32  64  128  256  512  1024
```

Note that loops comprising more than one line require braces, just like conditional blocks with more than one line. In many cases, the iteration will be over the indices of items in a list or a vector, and a good way to handle this is to use `seq_along()` (as in Sect. 2.3.3.1) to find the indices; also, note that, e.g.

```
for (i in seq_along(x))
    print(x[i])
```

will not execute the loop if `x` is empty, whereas

```
for (i in 1:length(x))
    print(x[i])
```

will try to execute the loop contents with `i=1` and then with `i=0`, perhaps surprising those who have not studied how "`:`" works. It pays to get in the habit of using `seq_along()` or its cousin, `seq_len()`. Also, if an index is not actually needed, it may be clearer to write, e.g.

```
for (n in seq(0, 5))
    cat(factorial(n), " ")
|1  1  2  6  24  120
```

Although `for` is a natural way to loop over discrete cases, some methods are better expressed with a `while` loop, which repeats while a condition remains `TRUE`. For example, Heron's method for estimating square roots is

```
x <- 4 # number whose square root is desired
r <- 1 # first guess
while (abs(r^2 - x) > 0.01)          # tolerance 0.01
    r <- 0.5 * (r + x / r)
r
|[1] 2.00061
```

A more basic loop is `repeat`, which never exits unless a `break` is executed.

```
x <- 0
repeat {
    cat(x, ' ')
    x <- x + 1
    if (x > 5)
        break
}
|0  1  2  3  4  5
```

Actually, `break` can be used to break out of any type of loop, not just `repeat` loops. A relative is `next`, which causes a short-circuit that returns to the top of the loop, e.g.

```
for (x in seq(-1, 1)) {
    if (x < 0)
        next
    print(x)
}
```
```
[1] 0
[1] 1
```

**Exercise 2.21** Write a loop that displays the values of items in the current workspace, using `ls()` and `get()`. (See page 194 for a solution.)

## *2.3.13 Alternative to Loops*

Loops are not always desirable, e.g. the addition of two vectors with a loop

```
nx <- length(x)
z <- vector("numeric", length=nx)
for (i in 1:nx)
    z[i] <- x[i] + y[i]
```

is slower[13] and more difficult to understand than the non-looping form

```
z <- x + y
```

For this example, the looping form is also less general than the second form, because the latter handles matrices and arrays, not just vectors.

In addition to arithmetic cases such as the above, R offers a variety of ways to avoid loops. For example, `apply()` uses a given function for the elements of matrices or arrays, with

```
m <- matrix(1:8, nrow=2, byrow=TRUE)
m
```
```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
```
```
apply(m, 1, sum)
```
```
[1] 10 26
```

showing how to sum along rows. (Use the second argument to `2` to sum across columns.) The third argument is any function with a single argument, e.g.

```
apply(m, 1, function(x) sum(x^2))
```
```
[1]  30 174
```

computes the sum of squares along rows.

[13]For more on performance issues, see Appendix E.

If the item under consideration is a list, then `lapply()` should be used, e.g.

```
l <- list(a=0:100, b=c(9,11))
lapply(l, mean)
```

(results not shown) computes the mean values of `a` and `b`.

There are several other functions in the "apply" family that are worth learning about. Readers familiar with Google's map-reduce method (Lämmel 2008) will see an analogy with the approach used in R. This topic is further discussed by Wickham (2011), both generally and in the context of his `plyr` and `dplyr` packages, which extend and simply the `apply` family of functions in base R.

**Exercise 2.22** Extract velocity from the `oce` dataset `adp`, and plot distance-averaged beam-1 velocity versus time. (See page 194 for a solution.)

**Exercise 2.23** Calculate and plot yearly average $CO_2$ data, using `lapply()`. (See page 195 for a solution.)

**Exercise 2.24** Use a function in the `plyr` package to find minima and maxima of the data stored in `ctd[["data"]]`, a CTD station provided by the `oce` package. (See page 196 for a solution.)

## 2.4 Graphics

A great strength of R as a research tool is the simplicity and power of its graphics system. The use of generic functions (Sect. 2.3.11.6) ensures that applying `plot()` to differing data types produces results tailored to those types. For example, supplied with two columns, `plot()` produces an $x$-$y$ plot. Supplied with a data frame, it creates a useful multi-panel graph that compares every column with every other column. Supplied with the results of a regression, `plot()` produces a set of deeply informative plots. In addition to such general things, many R packages provide plotting types, e.g. `oce` extends `plot()` to provide dozens of specialized oceanographic plots.

### *2.4.1 Scatter and Line Plots*

Several examples of scatter plots and line plots having already been shown, more complicated examples may be of interest at this point in the text.

The base R system does not provide polar plots, but these can be constructed without difficulty. Consider the hourly wind speed and direction measurements held in the `buoy` dataset of the `ocedata` package.

```
data(buoy, package="ocedata")
```

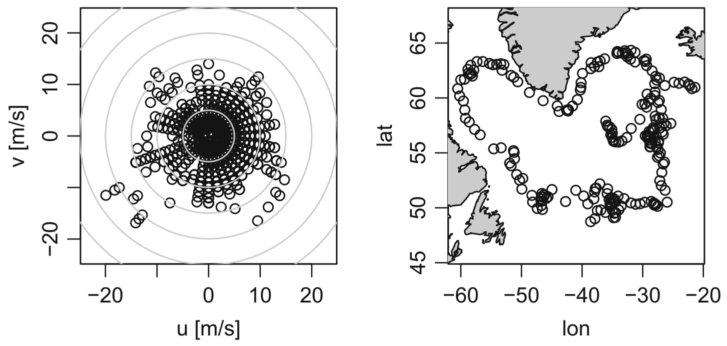

**Fig. 2.6** Left: winds near Halifax. Right: Argo float trajectory

The data frame stores direction from which the wind blows, measured clockwise from true North, so the air velocities ($u > 0$ meaning flow to east, etc.) are computed with

```
theta <- (90 - buoy$direction) * pi / 180
u <- -buoy$wind*cos(theta)
v <- -buoy$wind*sin(theta)
```

and the left panel of Fig. 2.6 is created with

```
s <- c(-1, 1) * max(buoy$wind, na.rm=TRUE)
plot(u, v, xlab="u [m/s]", ylab="v [m/s]",
     xlim=s, ylim=s, asp=1)
for (ring in seq(5, 30, 5))
    lines(ring*cos(seq(0, 2*pi, pi/32)),
          ring*sin(seq(0, 2*pi, pi/32)), col="gray")
```

Note the use of `xlim` and `ylim` to centre the diagram, `xlab` and `ylab` to control labels, and `asp` to set the aspect ratio; these are all optional arguments that would likely be skipped in a quick plot.

As a second example, consider the `argo` dataset, representing Argo float measurements made about once per ten days. This is an `oce` object, so the location data can be extracted with the accessor operator, `[[` (Sect. 3.3)

```
data(argo, package="oce")
lat <- argo[["latitude"]]
lon <- argo[["longitude"]]
```

after which it is a simple matter to plot the float trajectory[14] with a coastline[15]

```
plot(lon, lat, asp=1/cos(pi*mean(range(lat))/180))
data(coastlineWorldMedium, package="ocedata")
cwlon <- coastlineWorldMedium[["longitude"]]
```

[14] See Sect. 5.7 for more on Argo floats.

[15] Several coastline resolutions are provided in the `ocedata` and `oce` packages.

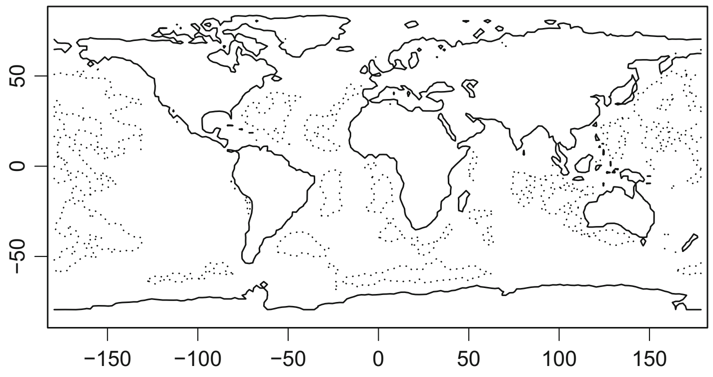

**Fig. 2.7** Contoured world topography, showing coastline and 5 km isobath

```
cwlat <- coastlineWorldMedium[["latitude"]]
polygon(cwlon, cwlat, col="gray")
```

### 2.4.2 *Contour Plots*

The `ocedata` package provides a 2-degree resolution topographic dataset called `topo2`, a simple matrix of elevation in metres above mean sea level. The matrix may be contoured simply, with `contour(topo2)` yielding a serviceable diagram with auto-selected contour intervals and axes running from 0 to 1, but the following yields a more informative result (Fig. 2.7); note the use of `xaxs` and `yaxs` to prevent distracting space around the plot.

```
data(topo2, package="ocedata")
lon <- seq(-179.5, 178.5, length.out=180)# see ?topo2
lat <- seq(-89.5, 88.5, length.out=90)
contour(lon, lat, topo2, drawlabels=FALSE,
        levels=c(0,-5000),
        lty=c(1, 3), xaxs="i", yaxs="i", asp=1)
```

*Caution* Users familiar with Matlab contouring may find R contouring confusing. If the `topo2` dataset is saved to a file with `write.table()` (with `row.names` and `col.names` both `FALSE`) and loaded into Matlab with `load topo2.dat`, then the matrix will have to be transposed before contouring in Matlab. A simple mnemonic should make the R approach clear: the mathematical form $z(x, y)$ translates to the code form `z[ix, iy]`, where `ix` and `iy` are indices for the `x` and `y` grid directions.

**Exercise 2.25** Reproduce Fig. 2.7 with axes labelled in geographical notation. (See page 196 for a solution.)

**Exercise 2.26** Devise a wrapper function to handle reversed `x` or `y` values in contouring. (See page 196 for a solution.)

**Exercise 2.27** Contour the formula for wind-chill temperature, $13.12 + 0.6215T - 11.37U^{0.16} + 0.3965TU^{0.16}$, as a function of air temperature, $T$, and wind speed, $U$. (See page 196 for a solution.)

### *2.4.3 Image Plots*

The R function `image()` produces basic images, e.g.

```
image(topo2)
```

yields the top panel of Fig. 2.8. There is no provision for a colour bar, but this can be created with `imagep()` in the `oce` package, e.g.

```
imagep(topo2)
```

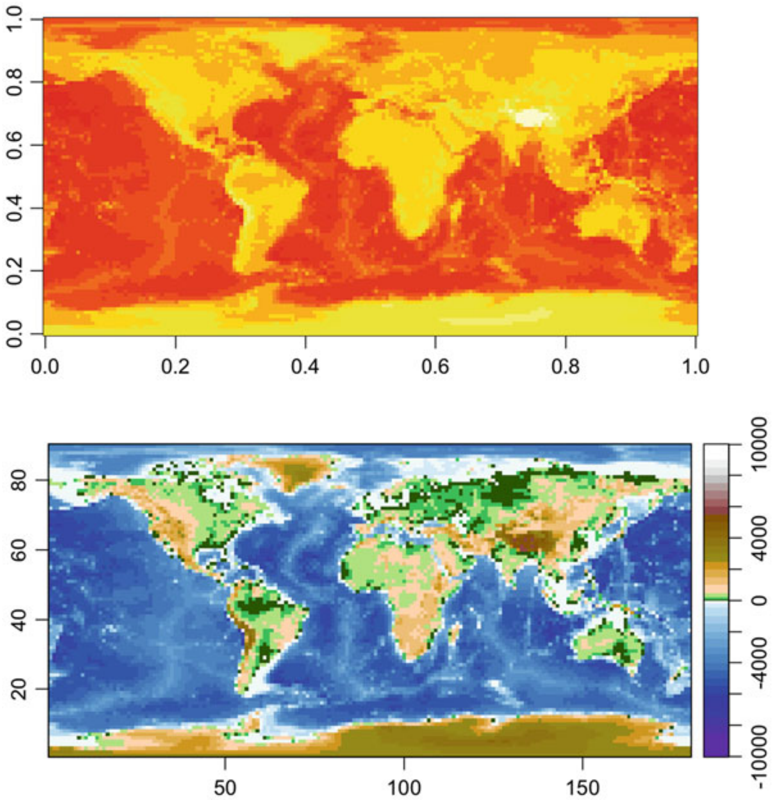

**Fig. 2.8** Images produced with `image()`, top, and `imagep()`, bottom

produces an image (not shown) that is similar, apart from the different default colour scheme and the axis labelling.

Since the data represent topography, it makes sense to set a colour scale used in the literature for such fields. For example, the colour scale named “globe” within the Generic Mapping Tools software (Wessel et al. 2013) may be specified with

```
imagep(topo2, colormap=colormap(name="gmt_globe"))
```

which creates the bottom panel of Fig. 2.8. The `imagep()` axis labelling indicates the matrix dimensions, which may be more useful than the unit range shown by `image()`. Another practical consideration is that `imagep()` handles several special cases for its first argument, e.g. if it is a `topo` object as defined by the `oce` package, then `imagep()` will extract longitude and latitude, and use these on the axes.

## 2.4.4 Hexagon Binning

R offers several ways to show data density in a two-dimensional space, including `smoothScatter()` in the base `graphics` package (Exercise 2.28), and more sophisticated variants in other packages. The hexagon bin scheme, in the `hexbin` package, is worth illustrating because it is also popular in other computing systems (Carr 1991).

As an example, the distribution of salinity and temperature values in the `papa` dataset (from Ocean Weather Station P) may be displayed as in Fig. 2.9 with

```
library(hexbin)
data(papa, package="ocedata")
S <- as.vector(papa$salinity)
T <- as.vector(papa$temperature)
plot(hexbin(S, T, xbins=20))
```

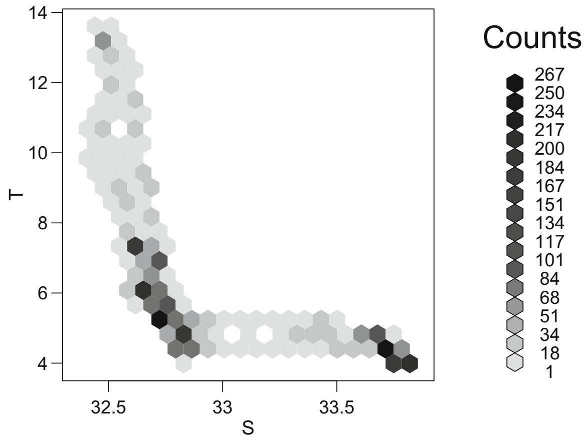

**Fig. 2.9** Hexagon bin representation of data density in the `papa` dataset

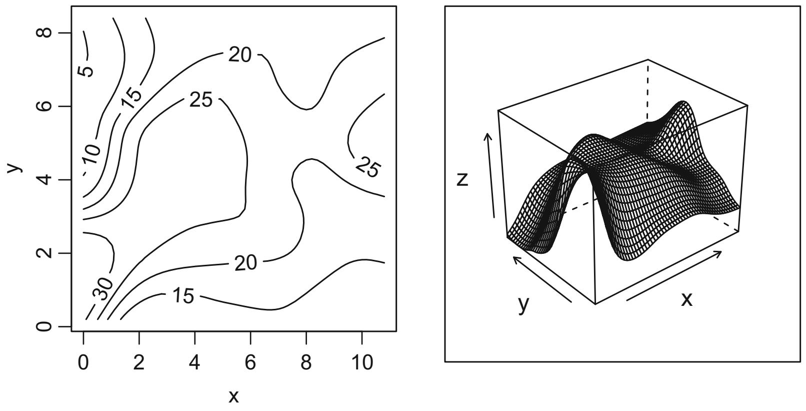

**Fig. 2.10** Three-dimensional data represented with contours and a wireframe

## 2.4.5 *Three-Dimensional Plots*

The `rggobi` package provides interactive 3D plots, while the `lattice` package provides static ones. Such graphs being best suited to smooth functions, an example will be constructed using `interpBarnes()` to smooth the `wind` dataset.

The contour diagram in the left panel of Fig. 2.10 is created with

```
library(lattice)
data(wind, package="oce")
g <- interpBarnes(wind$x, wind$y, wind$z)
contour(g$xg, g$yg, g$zg, xlab="x", ylab="y", labcex=1)
```

and the 3D wireframe plot is added as the right panel with

```
wireframe(g$zg, xlab="x", ylab="y", zlab="z", cex=5)
```

Adjusting the viewing angle of wireframe plots can sometimes reveal features of interest, but often at the expense of obscuring other features. Ragged fields are a particular problem, e.g. `wireframe(topo2)` yields an uninformatively dense blob. It can be best to avoid eye-catching 3D plots, relying instead on more quantitative formats such as contour diagrams and images.

## 2.4.6 *Time-Series Plots*

The `ts` class handles time series data that are sampled at a constant rate. Constructing and plotting such things is easy. For example, the `giss` dataset (Hansen et al. 2010) from the `ocedata` package can be converted to a `ts` object and plotted as in the left panel of Fig. 2.11 with

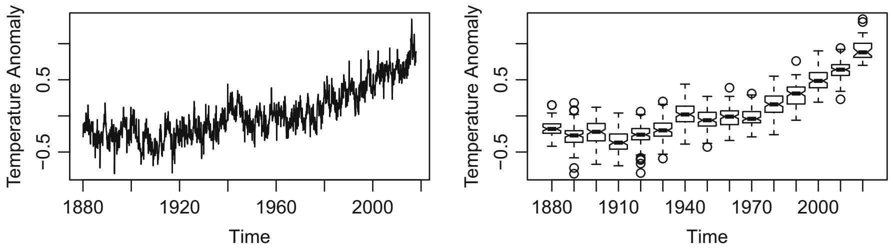

**Fig. 2.11** GISS surface temperature anomaly illustrated with time-series and box plots

```
data(giss, package="ocedata")
Ta <- ts(giss$index, start=giss$year[1],
         frequency=1 / mean(diff(giss$year)))
plot(Ta, ylab="Temperature Anomaly")
```

(The automatic naming of the horizontal axis is a sign that the plot function is decoding time information stored in `attributes(Ta)`; such a scheme also facilitates spectral analysis, discussed later.)

## 2.4.7 Box Plots

The monthly `giss` signal being fairly ragged, an analyst might wish to construct statistical descriptions over years or decades. A box plot can be a good starting point for this sort of analysis (see, e.g., Tukey 1977). The `round()` function may be used to round to powers ten so that

```
decade <- round(time(Ta), -1)
```

yields an indicator of decade, and then

```
boxplot(Ta ~ decade, notch=TRUE,
        xlab="Time", ylab="Temperature Anomaly")
```

gives a box plot as in the right panel of Fig. 2.11. The `notch` setting yields a display of a type of confidence interval on the median, which is desirable in some applications.

## 2.4.8 Lagged Autocorrelation Plots

Lagged autocorrelation analysis offers further insights on the `giss` temporal variability, e.g.

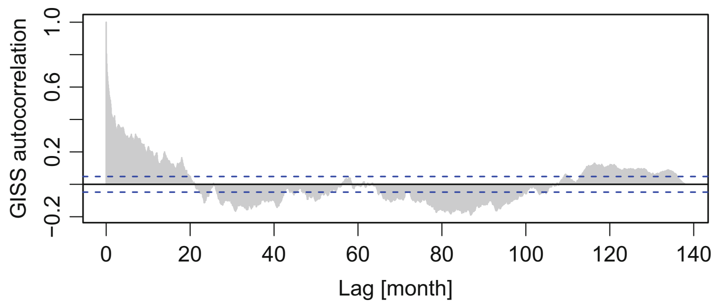

**Fig. 2.12** Autocorrelation analysis of GISS surface temperature anomaly

```
Tad <- Ta - predict(lm(giss$index ~ giss$year))
acf(Tad, lag.max=length(Tad), col="gray",
    xlab="Lag [month]", ylab="GISS autocorrelation")
```

yields Fig. 2.12 for a detrended version of the data. (The maximum lag has been set to the length of the time series, because the default is too small to show the interannual variation.)

### 2.4.9 Histogram Plots

Histograms may be created with `hist()`, e.g. histograms of the velocity components in the `adp` dataset may be constructed with

```
data(adp, package="oce")
velo <- adp[["v"]]
eastward <- velo[,,1]
northward <- velo[,,2]
upward <- velo[,,3]
hist(eastward, main="")
hist(northward, main="")
hist(upward, main="")
```

yielding Fig. 2.13. The `main` argument to `hist()` prevents an unaesthetic label at the top of each panel. Another argument that is commonly set is `breaks`, which controls the bins of the histogram. In some cases, it can help to extract the return value from `hist()`, to construct other types of plots. For example, a cumulative histogram could be constructed with

```
eh <- hist(eastward, plot=FALSE)
plot(eh$mids, cumsum(eh$counts), type="s")
```

where `plot=FALSE` prevents `hist()` from plotting. The cumulative histogram, not shown here, uses “staircase” type to reveal the break points.

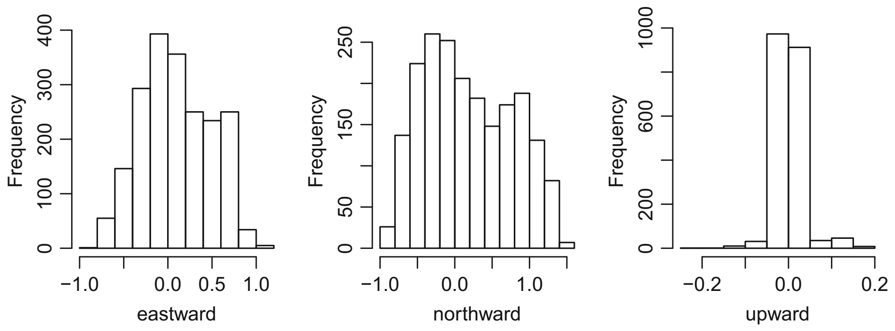

**Fig. 2.13** Histograms of velocity measured with an Acoustic Doppler Current Profiler

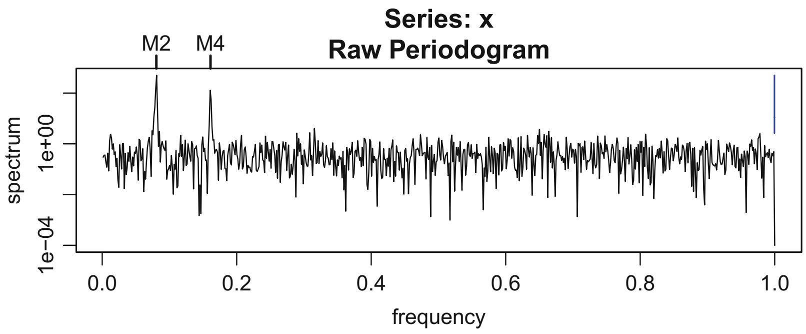

**Fig. 2.14** Spectrum of an artificial time-series. Note that the vertical axis is logarithmic

## 2.4.10 *Spectrum Plots*

Spectral analysis is a complicated subject that will be discussed in some detail in Sect. 5.9.4.5. Here, the goal is just to introduce plotting. For example,

```
t <- seq(0, 30 * 24, 1/2)
f <- list(M2=0.0805114007, M4=0.1610228013) # cph
x <- 2*sin(f$M2*2*pi*t) + sin(f$M4*2*pi*t) + rnorm(t)
```

simulates a month of half-hourly sampled tidal data, which may be converted to a time series with

```
xts <- ts(x, frequency=2)
```

after which the spectrum shown in Fig. 2.14 may be constructed with

```
spectrum(xts)
rug(c(f$M2, f$M4), side=3, tcl=-0.5, lwd=2)
mtext(c("M2", "M4"), side=3, line=0.5, at=c(f$M2, f$M4))
```

The vertical line above the highest frequency is actually a cross indicating uncertainty in spectral value and bandwidth, but it looks like a line because the bandwidth is very narrow, without spectral smoothing (see Sect. 5.9.4.5).

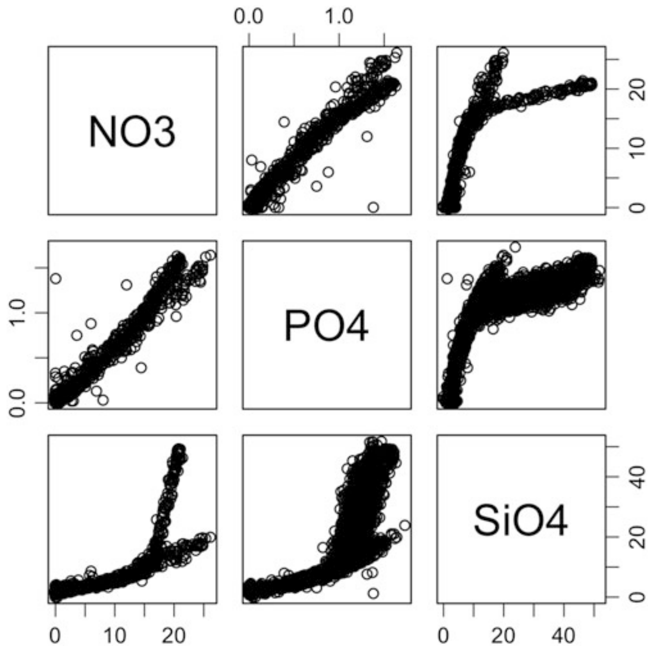

**Fig. 2.15** Pairs plot of hydrographic data in the `section` dataset

## 2.4.11 *Pairs Plots*

The R function `pairs()` makes it easy to "plot everything versus everything else," to quote advice the author received when he entered oceanography. This may be illustrated with the nutrient data in the WOCE oceanographic section provided in the `section` dataset, e.g.

```
data(section, package="oce")
d <- data.frame(NO3=section[["nitrate"]],
                PO4=section[["phosphate"]],
                SiO4=section[["silicate"]])
```

yields a dataset with $NO_3$, etc., and then

```
pairs(d)
```

yields Fig. 2.15. Actually, using `plot()` would have yielded a similar plot, because the first argument is a data frame, and so the generic function would have handed control to `pairs()`.

**Exercise 2.28** Use the `panels` argument to draw the panels as density diagrams, using `smoothScatter()`. (See page 197 for a solution.)

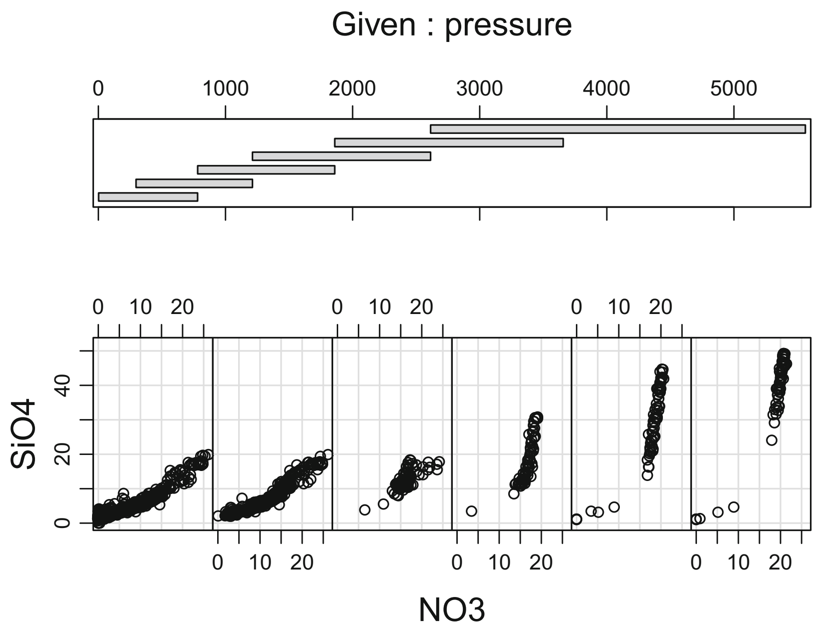

**Fig. 2.16** Conditioning plot showing the dependence of silicate on nitrate and depth (indicated by pressure), for the `section` dataset, spanning the Atlantic at 36N. The scatter-plot panels correspond to pressure ranges indicated by bars in the upper panel, revealing that the relationship between $SiO_4$ and $NO_3$ varies with depth in the water column

### *2.4.12 Conditioning Plots*

A conditioning plot, or coplot, represents multivariate dependencies by breaking data into categories of the independent variables. The dependence is expressed with a formula that is similar to that used for regression.

For example, Fig. 2.15 reveals a fairly tight fit between $NO_3$ and $PO_4$, but $SiO_4$ displays a more forked dependency. To investigate whether depth may be the hidden factor here, we might construct a conditioning plot of $SiO_4$ conditioned on both $NO_3$ and pressure. The first step is to extract the data

```
data(section, package="oce")
pressure <- section[["pressure"]]
NO3 <- section[["nitrate"]]
SiO4 <- section[["silicate"]]
```

after which Fig. 2.16 may be constructed with

```
coplot(SiO4 ~ NO3 | pressure, rows=1)
```

providing an indication that the nitrate-silicate relationship indeed varies with depth. Further investigation (e.g. of dependence on latitude) can be handled either by conditioning on another variable or by colour-coding symbols. (Other methods for studying watermass patterns are dealt with in Sect. 5.2.3.)

### 2.4.13 Function Plots

Given a need to graph the decaying oscillatory function $e^{-x} \cos 2\pi x$, one might start by defining the function

```
f <- function(x) exp(-x) * cos(2 * pi * x)
```

moving on to create vectors to be plotted

```
xx <- seq(0, 1, length.out=100)
yy <- f(xx)
```

and finally plotting

```
plot(xx, yy, type="l")
```

Although this is straightforward, it is certainly more tedious than using a function plot, e.g.

```
plot(function(x) exp(-x) * cos(2 * pi * x))
```

Readers who try this will notice that a default range of $x$ has been used, and that the vertical axis is automatically labelled with the formula. This second feature ensures that the label represents the function faithfully, which reduces the chance of erroneous results in interactive work involving trials with different functions. Small conveniences such as this help to explain why R is widely regarded as a comfortable tool for everyday analysis.

### 2.4.14 Aesthetic Control of R Graphics

R offers control over many aesthetic aspects of plots, through arguments to the plotting functions and through calls to `par()`. A few important examples are listed below.

The `type` argument of `plot()` indicates whether data on $x$-$y$ graphs should be indicated with symbols, connected with lines, etc. Symbol type is set with `pch` and symbol size with `cex`. Line type and width are set with `lty` and `lwd`, respectively. Colour is controlled with `col`. (For filled symbols, `col` applies to the perimeter, with `bg` being used for the fill colour.)

The default R plot margins are wider than those in typical publications, so the margins in most of the examples in this book are narrowed by adjusting the `mar` argument to `par()`. The related `mgp` argument, which controls the spacing of axis titles and labels, is also adjusted here. Readers may find

```
par(mar=c(3, 3, 1, 1), mgp=c(2, 0.7, 0))
```

to be a good starting point, if they wish to devote more space to the content and less to the axes.

Another practical consideration is the need for mathematical notation in plots, which is handled with expressions, e.g.

```
plot(1:10, (1:10)^2, xlab="x", ylab=expression(x^2))
```

An asterisk between symbols in expressions indicates multiplication, so that, for example, `expression(a*b)` is typeset as $ab$ (as in TEX and LATEX). Most other arithmetic operators are typeset as-is, e.g. `expression(a/b)` yields $a/b$. Indexing gives subscripts, so `expression(a[0])` yields $a_0$, while carat gives superscripts, `expression(a^2)` yielding $a^2$. Text and expressions may be combined with `paste()` or multiplication, e.g. both `expression(paste(x, " [m]"))` and `expression(x*" [m]")` yield $x$ [m]. A restriction worth noting is that expressions cannot contain certain keywords, so that, e.g. `expression(V[in])` is disallowed because `in` has a special meaning. However, `substitute()` solves this problem, with, e.g. `as.expression(substitute(V[x], list(x="in")))` yielding $V_{in}$.

In addition to aesthetic concerns, graphical designer should also take into account issues of accessibility. This is an area that is treated seriously in the R community, and an argument can be made that R outperforms other software in terms of accessibility to viewers with vision limitations (Cleveland and McGill 1984; Ihaka 2003; Zeileis et al. 2009).

### *2.4.15 Limitations of R Graphics*

The first impression of many Matlab users is that R is weak at interactive graphics. This is partly, but not entirely, true. For example, the `zoom` package provides a simple way to zoom in on subregions of plots. Similarly, `locator()` can be used to find plotted values based on mouse clicks. At a deeper level, `getGraphicsEvent()` provides access to mouse up/down events, keyboard events, etc., so it can be used for sophisticated interactive graphs. More widely, and arguably more importantly, the `shiny` package provides a way to set up flexible and powerful GUI systems (see Sect. 2.8). However, all of this requires extra programming, so it is not analogous to the scrolling/zooming behaviour that Matlab users may enjoy.

Matlab users who rely heavily on the GUI may find that they can derive benefits from changing their expectations and adopting new habits. For example, expanding a subregion of a graph in an interactive R session is no harder than using the up-arrow to recall the command that created the plot, adding or modifying the `xlim` or `ylim` arguments to that command, and pressing "enter" to repeat the `plot()` call. This is not as easy as using the mouse to select a region to enlarge, but neither is it especially arduous. Furthermore, R analysts tend to copy their interactive code into scripts, and this means that the details of the graphical setup are easily shared across research groups. This script-based way of working is employed by serious analysts in both Matlab and R, because it has clear advantages in terms of reproducible research. This is a matter of increasing concern in oceanographic research, with journals encouraging authors to supplement their papers with both data and methodological details.

Another complaint from some Matlab users is that multi-panel plots are handled awkwardly in R. Again, this is true, but it does not usually pose serious limitations in practice. Multi-panel graphs may be created in several ways in core R. The most common scheme may be to use `par()`, specifying `mfrow` or `mfcol` to lay out a uniform grid, or using `layout()` for a non-uniform grid. Unfortunately, these schemes do not permit alteration of previously drawn panels, a task that is accomplished easily in Matlab. The solution in R is simply to plan a bit more, or to cut/paste interactively entered commands into a script, and to make modifications there.

All of the above (and most of this book) is framed in reference to the system called "base graphics." This system is decades old, and has stood the test of time, despite some limitations. An recent alternative is the "grid graphics" system, developed by Paul Murrell, and described in his textbook (Murrell 2006) as well as in the documentation for the `grid` package. As a low-level system, `grid` imposes a high burden on the user. The best way to leverage its strengths may be with Hadley Wickham's `ggplot2` package, which produces elegant results for a broad range of graphical elements. While `ggplot2` is worth serious consideration for many applications, there are two reasons why it is not used in this book or in the companion `oce` package. First, the R documentation uses base graphics, so serious use of `ggplot2` requires that analysts become comfortable with two systems that differ widely in fundamental respects. Second, `ggplot2` is significantly slower than base graphics, which can prove to be problematic for the interactive analysis of oceanographic datasets, which are often quite large. For example, on the author's computer, plotting a histogram of $10^7$ points took 0.5 s with `hist(x)` in base graphics, and 7.5 s with `qplot(x, geom="histogram")` in `ggplot2`. The slowness of the `ggplot2` version is a problem for modern oceanographic instruments, some of which have on-board storage of 16 Gbyte.

## 2.5 Probability and Statistics

### *2.5.1 Probability*

R is used commonly in undergraduate teaching of probability and statistics, so nobody should not be surprised that it handles these topics elegantly and efficiently. Readers with textbook knowledge and limited coding experience are cautioned against trying to improve upon the R functions, because they tend to be more robust than naive solutions. The `choose()` function provides a case in point, with

```
choose(52, 5)
```

```
[1] 2598960
```

giving the number of possible 5-card poker hands in a 52-card deck. Mathematically, this is ${}^{52}C_5 = 52!/(47!\,5!)$, but expressing this directly, with

```
factorial(52) / (factorial(47) * factorial(5))
| [1] 2598960
```

is problematic. Although this works for a 52-card deck, it fails with much larger decks, for which `factorial()` produces values too large to represent in the computer. (Choosing from four decks illustrates this on a 64-bit machine.) Similar limitations exist for other "natural" expressions in probability. The lesson is that only experts should consider replacing built-in R functions.

**Exercise 2.29** The Rink Ratz® hockey card game has a 69-card deck with 2 desirable "miraculous save" cards. At the start of the game, 5 cards are discarded without being examined. What is the probability that there will be exactly 1 miraculous save card left in the deck? (See page 197 for a solution.)

### 2.5.2 *Statistics*

R provides various properties of statistical distributions, in a family of functions with names that might seem cryptic at first. The first letter of the name indicates the quantity to be calculated, with subsequent letters indicating the distribution, as outlined in Table 2.1. Using `d` for the first letter retrieves a probability density

**Table 2.1** Statistical distributions provided in R

| Name code | Distribution |
|---|---|
| `beta` | beta distribution |
| `binom` | Binomial distribution |
| `cauchy` | Cauchy distribution |
| `chisq` | $\chi$-squared distribution |
| `exp` | Exponential distribution |
| `f` | F distribution |
| `gamma` | Gamma distribution |
| `geom` | Geometric distribution |
| `hyper` | Hypergeometric distribution |
| `lnorm` | Log-normal distribution |
| `logis` | Logistic distribution |
| `nbinom` | Negative-binomial distribution |
| `norm` | Normal distribution |
| `pois` | Poisson distribution |
| `signrank` | Signed-rank distribution |
| `t` | t distribution |
| `unif` | Uniform distribution |
| `weibull` | Weibull distribution |
| `wilcox` | Wilcox distribution |

function, while p retrieves a cumulative probability density function, q a quantile function, and r a random number function. The examples of the following sections should clarify the scheme.

#### 2.5.2.1 Statistical Tables

Although statistical tables lose much of their value in a system that permits easy calculation of any desired quantity, readers might find it instructive to try reproducing some of the tables found in their textbooks, e.g.

```
t(outer(1:3, 1:5, function(nu1, nu2) qf(1-0.05, nu1, nu2)))
```

```
           [,1]       [,2]       [,3]
[1,] 161.447639 199.500000 215.707345
[2,]  18.512821  19.000000  19.164292
[3,]  10.127964   9.552094   9.276628
[4,]   7.708647   6.944272   6.591382
[5,]   6.607891   5.786135   5.409451
```

creates a table of $F$ values for $\alpha = 0.05$. Here, outer() has been used to create the table, with an anonymous function taking $\nu_1$ and $\nu_2$ as input.

**Exercise 2.30** Construct a graph comparing the normal distribution with the $t$ distribution with 2 degrees of freedom. (See page 197 for a solution.)

#### 2.5.2.2 Confidence Intervals of Means

A confidence interval on the mean of a vector $x$ of length $n$ drawn from the Student t distribution is given by $t_* s/\sqrt{n}$ where $t_*$ is the value of the t distribution for $n-1$ degrees of freedom and a specified confidence level, and the standard deviation $s$ may be computed with sd(). For example,

```
qt(0.975, df=length(x)-1) * sd(x) / sqrt(length(x))
```

illustrates the computation of a 95% confidence interval of the mean of x.

Other distributions are handled similarly, with qt() being replaced with qnorm() for the normal distribution, dchisq() for $\chi^2$, etc.

#### 2.5.2.3 Measurement Uncertainty and Error Bars

It is important to distinguish between the confidence intervals of means discussed in the previous section, and measurement uncertainties. The latter refer to the spread of data, and this is usually what is meant by with "±" in text and by error bars on plots.

As explained by Taylor and Kuyatt (1994, Sections 3 and 4), measurement uncertainties may be divided into Type A, related to the statistics of the data, and Type B, related to wider factors such as previous measurements, manufacturer specifications, calibrations, etc.

For Type A, Taylor and Kuyatt (1994, Section 2.3) recommend calculating measurement uncertainty as $ku_i$, where $k$ is a "coverage factor" calculated from the statistics of the data and $u_i$ is the "standard uncertainty," inferred as the standard deviation and thus calculated with `sd()`. This is a factor of $\sqrt{n}$ larger than the confidence interval on the mean, so analysts must be careful to report methods clearly to avoid confusion between two numbers that may be very different.[16]

Setting $k$ to 1.96 achieves 95% coverage probability for normally distributed data, perhaps explaining why some analysts use $k = 2$ for approximate 95% coverage probability. Different applications may call for different $k$ values, so analysts should provide enough information to let readers perform their own calculations with different parameters (Taylor and Kuyatt 1994, Section 7).

**Exercise 2.31** Write a function that computes measurement uncertainties assuming a t distribution. (See page 198 for a solution.)

**Exercise 2.32** Write a function that plots error bars. (See page 198 for a solution.)

#### 2.5.2.4 Random Number Generation

As noted above, random numbers may be generated from many different distributions. Two commonly used cases are `rnorm()` for the normal distribution and `runif()` for the uniform distribution. Another useful function is `set.seed()`, which sets the seed for a sequence of random numbers, which is handy for reproducible research.

### *2.5.3 Summaries and Overview Functions*

The `summary()` function produces useful summaries of data and objects. Being a generic function, it acts differently when supplied with different arguments. In the simple case of numerical arguments, it reports the data range, mean, and three quartiles, e.g.

```
summary(rnorm(100))
```

```
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
-2.48865 -0.51127 -0.02768 -0.04943  0.60007  2.55300
```

[16]Home electricity provides a dramatic illustration. Although voltage measurements may give a confidence interval on the mean that barely departs from 0V, the measurement uncertainty will indicate that any given measurement could easily be of order 100V. That is why electrical outlets must be covered up, in houses with young children.

Other data types are treated in sensible ways. Related functions are also useful, e.g. `quantile()` calculates quantiles, and `fivenum()` gives Tukey's five-number summary (lower limit, lower "hinge", median, upper hinge, and upper limit). A useful semi-graphical way to summarize data is with `stem()`, e.g. with[17]

```
set.seed(253)
x <- rnorm(n=30, mean=2, sd=0.1)
```

as data, the results

```
stem(x)
```

```
 The decimal point is 1 digit(s) to the left of the |

 18 | 9
 19 | 13344
 19 | 56778888
 20 | 12333
 20 | 666779
 21 | 111
 21 |
 22 | 1
 22 |
 23 | 3
```

indicate (in the first and second lines) that 1.89 and 1.91 are in $x$, along with two instances of 1.93 and 1.94, etc. Thus, `stem()` produces not just a textual histogram, but also a list of the (rounded) data.

There are several graphical representations that may be used, e.g. Fig. 2.17 shows histograms and box plots

```
hist(x, main="")
boxplot(x, horizontal=TRUE, notch=TRUE)
```

along with density and violin plots

```
plot(density(x), main="")
rug(x, side=1)
library(vioplot)
vioplot(x, horizontal=TRUE, col="gray")
```

Note that the data are shown along the lower axis of the density plot, by using `rug()`. This function is quite handy with relatively small datasets.

### 2.5.4 *Hypothesis Testing*

Hypothesis testing is supported with a suite of R functions. Explanations of the underlying ideas may be found in most textbooks on statistics, and e.g. Legendre and

[17]Note the use of `set.seed()` to let readers reconstruct the example.

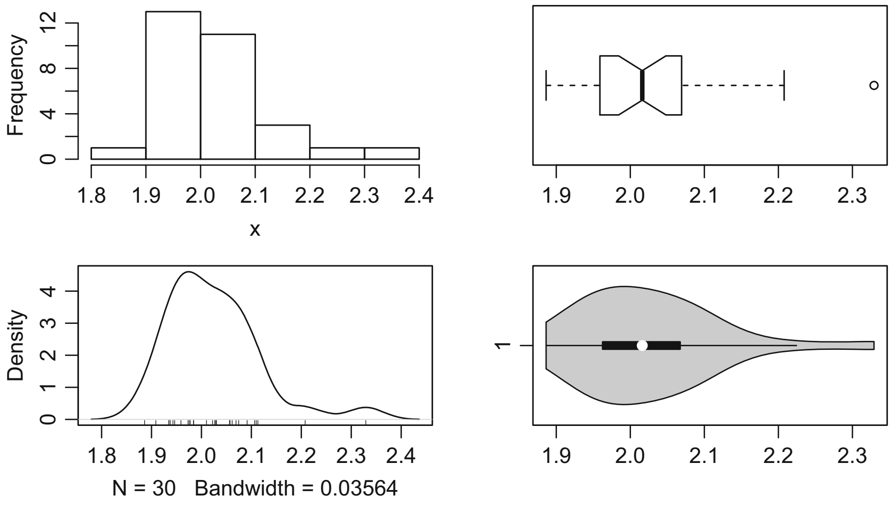

**Fig. 2.17** Data summaries from `hist()`, `boxplot()`, `density()` and `vioplot()`

Legendre (1998) and Borcard et al. (2011) provide treatments in the oceanographic context.[18]

The popular *t* test is handled with `t.test()`, which provides both one-sample or two-sample tests, the latter paired or unpaired. The `alternative` argument controls alternative hypotheses, while `mu` sets the mean, `conf.level` sets the confidence level, etc.

The data shown in Fig. 2.17 provide an example.

```
t.test(x)
```

```
        One Sample t-test

data:  x
t = 120.05, df = 29, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 1.989308 2.058263
sample estimates:
mean of x
 2.023786
```

[18]It is unwise to use hypothesis tests without considering their limitations. Some issues of misapplication are outlined by, e.g., Johnson and Omland (2004) and Hauer (2004), and deep concerns about the misuse of $p$ values are raised in a highly influential editorial in The American Statistician (Wasserstein and Lazar 2016).

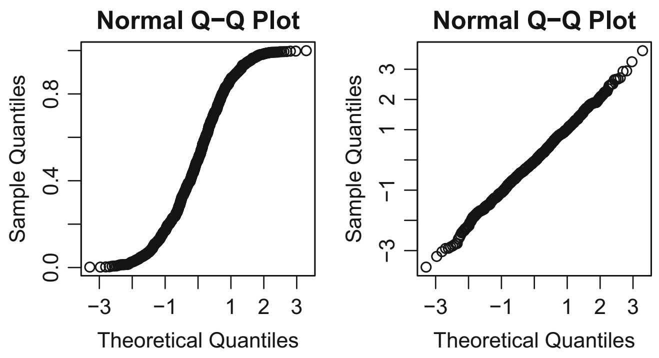

**Fig. 2.18** Q-Q plots for random numbers chosen from a uniform distribution (left) and normal distribution (right)

The $p$ value is small[19] compared with typical significance levels (often 0.05), which argues in favour of the alternative hypothesis (which is helpfully stated in the `t.test()` output) that the mean of $x$ is not equal to zero.

The documentation for the `stats` package is worth consulting for its discussion of the many other tests that complement the $t$ test. Just one more example will be given here: testing for normality.

A typical first step in testing a distribution for normality is to draw a Q-Q plot with `qqplot()`. For example, Fig. 2.18 shows such plots with random numbers drawn from uniform and normal distributions.

```
set.seed(254)
uniform <- runif(n=1e3)
normal <- rnorm(n=1e3)
par(mfcol=c(1,2))
qqnorm(uniform)
qqnorm(normal)
```

A linear Q-Q plot suggests a normal distribution, as in the right panel. This plot style is so useful that it is included in the graphs produced when `plot()` is called with the results of a regression model (see Sect. 2.5.5).

R provides a variety of formal tests as well, e.g. a Shapiro-Wilks test

```
shapiro.test(uniform)
        Shapiro-Wilk normality test

data:  uniform
W = 0.9495, p-value < 2.2e-16
```

yields a small $p$ value, suggesting (as expected) that the `uniform` data are not normally distributed. By contrast, the same test on the `normal` data yields a high $p$

[19]If $p < 2.2 \times 10^{-16}$, R regression summaries simply reports "`p-value: < 2.2e-16`".

value, consistent with a normal distribution. (Other tests of normality are provided in the `mvShapiroTest`, `nortest` and `fBasics` packages.)

**Exercise 2.33** Show how `split()` and `laply()` can be used to produce a monthly climatology of a signal, and illustrate using the results of Exercises 2.31 and 2.32. (See page 198 for a solution.)

### *2.5.5 Regression*

Since regression is used in many fields, there are many resources that explain the theory and provide practical advice. Accordingly, the present treatment is somewhat cursory, focussing mainly on R details; see also Chambers and Hastie (1992), Faraway (2005), and Faraway (2002).

The standard R system provides linear, generalized, and nonlinear least-squares regression, with `lm()`, `glm()`, and `nls()` respectively, Optional packages provide much more, e.g. `MASS` (Venables and Ripley 1999) provides robust regression with `rlm()` and ridge regression with `lm.ridge()`, `segmented` (Muggeo 2008) provides piecewise-linear regression, etc. Some methods are handled by multiple packages, e.g. ridge regression is provided by `mgcv` (Wood 2001) as well as by `MASS`. Readers with specific needs should study the documentation of specialized functions, but it is wise to start with the basics, the following discussion of which is divided into linear and nonlinear categories.

#### 2.5.5.1 Linear Regression

Linear regression involves the study of data modelled by

$$y = \beta_1 x_1 + \beta_2 x_2 + \cdots + \epsilon \tag{2.3}$$

where $y$ is a response vector that depends on vectors $x_i$, scalars $\beta_i$ are constants to be determined, and $\epsilon$ is a vector of error or misfit. An intercept is handled by setting $x_1 = 1$, and polynomial regression by expressing, e.g., $y = a + bx + cx^2$ with $x_1 = 1$, $x_2 = x$ and $x_3 = x^2$.

Synthetic data can be used to illustrate how regression (essentially, calculating $\beta_i$) works in R. For example,

```
set.seed(2551)                    # for reproducibility
x <- seq(0, 1, length.out=25)
y <- 1 + 2 * x + 4 * x^2 + rnorm(length(x), sd=0.1)
```

creates a test case that is not quite linear, so it can be used to illustrate how to work through a sequence of regression models.

A sensible first step is to plot the data, e.g.

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

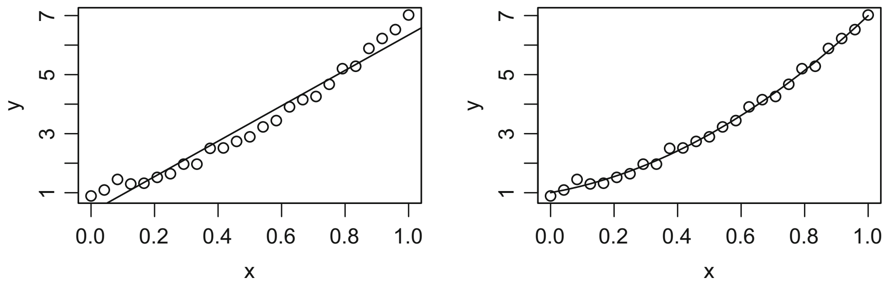

**Fig. 2.19** Use of `lm()` for linear and quadratic regression models

producing the symbols in Fig. 2.19. Although inspection reveals curvature, an analyst working with such data might still start with a linear model, in the interests of simplicity or to satisfy a theoretical or practical constraint.

The workhorse regression function for linear models is `lm()`, and

```
lm(y ~ x)
```

```
Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
     0.3469       5.9952
```

shows that it is easy to use. The "~" in the first argument indicates that it is a formula. It could be read as "y depends linearly on x", but emphatically not that the model is $y = \beta x$, because an intercept is implied; the zero-intercept case is explained later. A formula can specify interaction terms and other complications; see `help(formula)` and `help(lm)` for more on the notation.

The results of the regression may be saved to a variable, e.g.

```
linear <- lm(y ~ x)
```

and this is helpful because it facilitates further analysis, e.g.

```
abline(linear)
```

adds the regression line to the left panel of Fig. 2.19.

The right panel of Fig. 2.19 shows the result of a quadratic fit, resulting from using `poly()` in the regression formula

```
quadratic <- lm(y ~ poly(x, 2, raw=TRUE))
```

Here, `raw=TRUE` tells `poly()` to use conventional polynomials, not orthogonal polynomials. Another way to get conventional polynomials is with

```
quadratic <- lm(y ~ x + I(x^2))
```

where `I()` causes `lm()` to take "^" to mean exponentiation as opposed to interaction (see `help(formula)`). The prediction may be added with

```
plot(x, y)
lines(x, predict(quadratic))
```

where it should be noted that `predict()` is a generic function that works for all sorts of regressions.

The `summary()` function is useful for studying regression results, e.g.

```
summary(linear)
```

```
Call:
lm(formula = y ~ x)

Residuals:
    Min      1Q  Median      3Q     Max
-0.4512 -0.3282 -0.0916  0.2943  0.6782

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    0.347      0.140    2.47    0.021 *
x              5.995      0.240   24.94   <2e-16 ***
---
Signif. codes:
0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.361 on 23 degrees of freedom
Multiple R-squared:  0.964,       Adjusted R-squared:  0.963
F-statistic:  622 on 1 and 23 DF,  p-value: <2e-16
```

provides the information normally stated in publications, and much more. First, it repeats the regression formula, which is helpful for stored output. Then, it provides information about the residual deviations from the fit, which can be quite helpful if combined with information on measurement uncertainties. The model coefficients are presented next, along with standard errors and corresponding $t$ and $p$ values. Finally, the overall fit is described in terms of correlation coefficients, $F$ statistic and $p$ value.

In this case, the artificial data have been constructed with a random $y$ component of standard deviation 0.1, and the residual standard error of the linear regression is three times as large. Such a comparison (with the standard deviation of this simulation perhaps being replaced with an estimate of measurement uncertainty in a real study) might motivate further investigation of model formulation, even if Fig. 2.19 had not shown a systematic misfit.

The improved results of quadratic regression are revealed by

```
summary(quadratic)
```

an excerpt of which is

```
Residual standard error: 0.08655 on 47 degrees of freedom
```

indicating a residual comparable to the “noise,” a condition that might indicate success if this were a calibration study.

However, `summary()` is not the only tool R provides for checking on regression results. The `plot()` function is specialized to handle regression output, e.g.

```
par(mfrow=c(2, 2))
plot(linear)
```

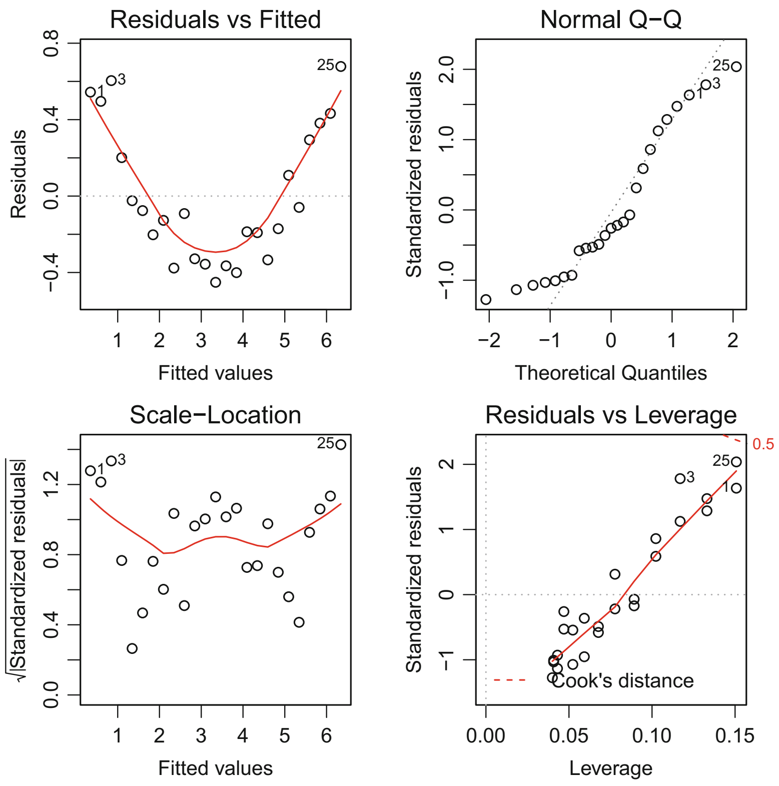

Fig. 2.20 Plot of the regression object for the linear fit shown in Fig. 2.19

yields Fig. 2.20. The top-left panel shows how the model-data misfit varies with fitted value (a format that also handles multiple independent variables), and the clear pattern of variation indicates the limitation of a linear model in this case. Furthermore, the Q-Q plot in the top-right panel reveals that the misfit is not normally distributed. The other two panels provide further diagnostics (see `help(plot.lm)`). The numbers in the panels are the indices of points that may deserve further consideration, and this can be helpful in revealing data outliers, as well as poor choices of regression model.

A few questions should be answered before discussing the nonlinear case.

- *Can the line be forced to go through the origin?* Yes, by appending `-1` to the formula, e.g. writing `lm(y ~ x - 1)` in place of `lm(y ~ x)`.
- *Can the line slope be specified?* Yes, by using `offset`. This can be useful in fitting power laws, e.g. for $y = Ax^{2/3}$ with $A = 10$ plus noise,

```
x <- 1:100
y <- rnorm(100, mean=10)*x^(2/3)
```

```
m <- lm(log10(y) ~ offset(log10(x^(2/3))))
10^confint(m)
                    2.5 %    97.5 %
 (Intercept) 9.757443 10.20824
```

where `confint()` gives confidence intervals on the regression coefficients. Another notation places known dependence on the left in the formula, e.g.

```
10^confint(lm(log10(y/x^(2/3)) ~ 1))
```

- *Can R do stepwise regression?* Yes; see `help(step)` and `help(stepAIC, package="MASS")`.
- *Can R draw confidence intervals on the fitted curve?* Yes, with `predict()`

```
prediction <- predict(lm(y~x),interval="confidence")
lines(x, prediction[,1]) # fit
lines(x, prediction[,2], lty="dashed") # lower bound
lines(x, prediction[,3], lty="dashed") # upper bound
```

- *Can R do multiple regression?* Yes, just add the independent variables to the right side of the formula, e.g.

```
data(ctd, package="oce")
sigthe <- ctd[["sigmaTheta"]]
sal <- ctd[["salinity"]]
temp <- ctd[["temperature"]]
m <- lm(sigthe ~ sal + temp)
```

- *How are regression coefficients retrieved?* With `m` as in above, use, e.g.

```
coef(m)
 (Intercept)          sal         temp
  -6.6899404    1.0215729   -0.1170753
```

- *Can R handle type-II regression, with errors in both x and y?* Yes, and several methods are available. Several R packages support type-II regression, including `smatr` (Warton et al. 2012) and `lmodel2` (Legendre 2014). See Warton et al. (2006) for general issues, and e.g. Ricker (1973), Marsden (1999), McArdle (2003), and Clarke and Van Gorder (2012) in the context of oceanography.
- *Can R handle robust regression?* Yes, using `rlm()` from the `MASS` package, e.g. the following simulates the effect of a missing-value code

```
x <- 1:10
y <- -10 + 5 * x + rnorm(10)
y[10] <- -99.99 # insert 'missing value' code
```

and Fig. 2.21, constructed with

```
plot(x, y)
abline(lm(y ~ x))
library(MASS)
abline(rlm(y ~ x), lwd=3)
```

shows that `rlm()` performs better than `lm()` in this case.
- *Can R handle piecewise linear regression?* Yes, with the `segmented` package, as the right panel of Fig. 2.21 shows for wave heights and wind speeds

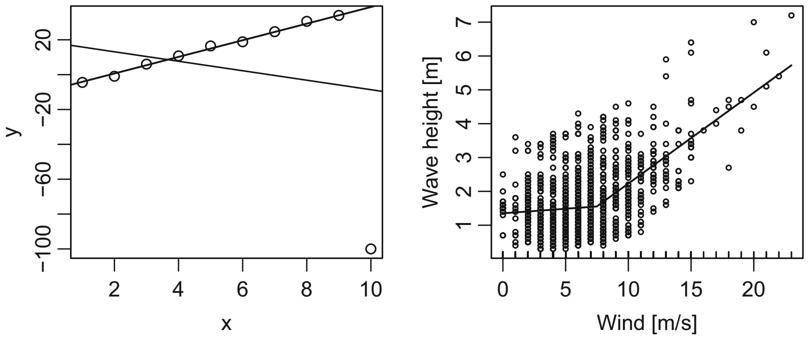

**Fig. 2.21** Left: conventional and robust regression of data with a spurious outlier, in thin and thick lines. Right: segmented regression of `buoy` data

```
data(buoy, package="ocedata")
plot(buoy$wind, buoy$height, cex=1/2,
     xlab="Wind [m/s]", ylab="Wave height [m]")
library(segmented)
s <- segmented(lm(height~wind, data=buoy),
               seg.Z=~wind, psi=c(10))
plot(s, add=TRUE, lwd=3)
```

**Exercise 2.34** Contrast the residual plots produced by `plot()` for `linear` and `quadratic`. (See page 199 for a solution.)

**Exercise 2.35** Use `eigen()` and `cov()` to draw a line that intersects the means of $x$ and $y$, and that has the same slope as the principal eigenvector of the covariance matrix. (See page 200 for a solution.)

#### 2.5.5.2 Nonlinear Regression

Nonlinear regression involves working with a model equation of the form

$$y = f(\mathbf{x}, \boldsymbol{\beta}) + \epsilon \tag{2.4}$$

where $f$ is a nonlinear function of independent variables $\mathbf{x} = (x_1, x_2, \ldots)$ and model parameters $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots)$ and $\epsilon$ represents model-data misfit. The concepts of nonlinear regression are explained in a variety of textbooks and other sources; in the R context, see, e.g., Chapter 8 of Venables and Ripley (1999) and the references in the documentation for `nls()`, the function used for nonlinear regression in R. Before getting into the details, it should be noted that `nls()` is harder to use than its linear cousin `lm()`, simply because nonlinear regression is a much more complicated task, and not one that can always be accomplished in practice (see, e.g., Gallant 1975).

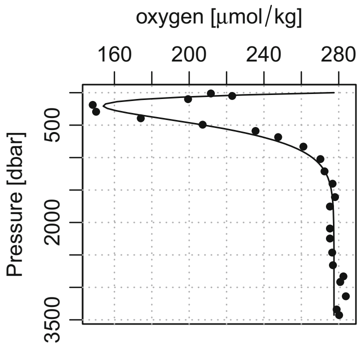

**Fig. 2.22** An oxygen concentration profile and the predictions of a nonlinear regression model

Typical procedures may be illustrated with the sample task[20] of fitting a model to an oxygen profile plotted in Fig. 2.22 with

```
data(section, package="oce")
stn <- section[["station", 112]]
plotProfile(stn, "oxygen", type="p", pch=20)
p <- stn[["pressure"]]
O2 <- stn[["oxygen"]]
```

This suggests a model function with a mid-depth minimum, e.g.

$$O_2 = A - Bpe^{-p/C} \tag{2.5}$$

where $A$, $B$ and $C$ are parameters to be determined. Since `nls()` requires starting values for its parameter search, the first step is to estimate reasonably sensible values of $A$, $B$ and $C$ (e.g. all should be positive, to get $O_2 > 0$ with a mid-depth minimum and a deep-water asymptote). Substituting $p = 0$ in (2.5) reveals that $A$ is the surface $O_2$ value, and differentiation indicates that the minimum $O_2 = A - BC/e$ occurs at $p = C$. Figure 2.22 might thus motivate starting values $A = 300\,\mu$mol/kg, $B = 2\,\mu$mol/(kg dbar), and $C = 200$ dbar, each perhaps to within a factor of 2. A test of whether `nls()` can find a solution without accurate starting values might be to set each of the three numerical values to 10.

Indeed, these starting values produce a converged solution

```
m <- nls(O2~A-B*p*exp(-p/C), start=list(A=10, B=10, C=10))
```

and the gist of this nonlinear regression is summarized as

```
m
Nonlinear regression model
  model: O2 ~ A - B * p * exp(-p/C)
   data: parent.frame()
      A       B       C
```

[20]See also the `NISTnls` package, which provides data and code for statistical test suites developed by researchers at the U.S. National Institute for Standards and Technology.

```
277.473   1.668 201.307
 residual sum-of-squares: 3501

Number of iterations to convergence: 11
Achieved convergence tolerance: 3.894e-06
```

The model prediction is added to Fig. 2.22 with

```
pp <- seq(0, max(p), length.out=100)
lines(predict(m, list(p=pp)), pp)
```

and the reasonable agreement with data (compared with $O_2$ scatter in deep water) might motivate further investigation. The next step would be to use `summary(m)`, after which using `profile(m)` and `plot(profile(m))` can shed light on the tightness of the fitted parameters. However, before taking any further steps, `nls()` must converge, which is not always the case. Problems fall into several classes.

The first requirement is that `start` must match the formula. The regression cannot succeed if `start` is missing parameters, with, e.g.

```
nls(y ~ a+b*x, start=list(a=1))
```

yielding the straightforward message

```
Error in nls(y ~ a + b * x, start = list(a = 1)) :
  parameters without starting value in 'data': b
```

Unfortunately, specifying too many `start` items

```
nls(y ~ a + b * x, start=list(a=1, b=1, c=1))
```

yields the more cryptic error message

```
Error in nlsModel(formula, mf, start, wts) :
  singular gradient matrix at initial parameter
  estimates
```

This is because the calculation involves partial derivatives of model misfit with respect to each `start` item, and the derivative with respect to $c$ is zero for this model, yielding a non-invertible matrix.

It is also a mistake for models to have commingled parameters, e.g. the $y = (\beta_1 + \beta_2)x$ cannot be solved for $\beta_1$ and $\beta_2$ independently, so

```
x <- 1:10
y <- 3*x
nls(y ~ (a+b)*x, start=list(a=1, b=2))
```

yields a singular-gradient matrix. In this case, the matrix is singular because two derivatives are identical, and matrices with identical rows or columns cannot be inverted.

Helpfully, `nls()` also checks for singularity during its search through parameter space. For example, $y = ax + \exp(bx)$ degenerates to the problematic form $y = (a + b)x$ if `nls()` selects a value of $b$ for which $|bx| \ll 1$ for the data being examined. The error message in such a condition is similar to that shown above, but without the "initial" phrase.

Problems can also arise when the data-model misfit function has a much stronger dependence on one parameter than on another. This is akin to the challenge of navigating to the lowest spot in a curvy valley that is long and thin. This sort of problem can be signalled by a variety of error messages from `nls()`, e.g.

```
number of iterations exceeded maximum ...
```

(where ... will be an integer) indicates that an excessive number of steps has been taken, with no end in sight. This might be solved with, e.g.

```
nls(..., control=list(maxiter=200)) # default is 50
```

but it may be better to reformulate the problem, e.g. by a change of variable to get O(1) variations. Similarly, the problem

```
step factor ... reduced below 'minFactor' of ...
```

may be alleviated by altering `minFactor` in `control`.

Sometimes, it helps to try alternative solution methods. The default algorithm employed by `nls()` is based on a Gauss-Newton procedure, but if this fails, it might help to use the `algorithm` argument to try other methods. The `"port"` algorithm is notable because it permits the specification of upper and lower limits for the parameters, which can help if `nls()` is straying into regions of parameter space that are unphysical (e.g. negative salinities) or uninteresting (e.g. angles outside the range 0 to $2\pi$).

If the `nls()` numerical differentiation is problematic, a function can be provided to calculate derivatives (Chambers and Hastie 1992), e.g. for (2.5)

$$\frac{\partial \mathrm{O}_2}{\partial A} = 1\,, \frac{\partial \mathrm{O}_2}{\partial B} = -p\,\mathrm{e}^{-p/c},\ \text{and}\ \frac{\partial \mathrm{O}_2}{\partial C} = -\frac{Bp^2}{C^2}\mathrm{e}^{-p/C} \tag{2.6}$$

which may be employed as follows:

```
O2Model <- function(A, B, C)
{
    E <- exp(-p / C)
    prediction <- A - B * p * E
    gA <- 1
    gB <- -p * E
    gC <- -B * p^2 / C^2 * E
    gradient <- cbind(gA, gB, gC)
    attr(prediction, "gradient") <- gradient
    prediction
}
mg <- nls(O2~O2Model(A, B, C), start=list(A=10, B=10,
C=10))
```

with results as already shown.

As a general matter, it can be helpful to call `nls()` with `trace=TRUE`, which prints the sequence of parameter values being tested. This can reveal a variety of problems, e.g. poor starting values can be signalled by rapid departures from the initial state. When a sequence of datasets are to be studied, it can be good to use `try()` to handle errors in `nls()` calls, whether to provide helpful information on the problematic test cases or to work through a sequence of trial model formulations.

**Exercise 2.36** Extract tritium Tu and pressure $p$ from the `ocedata` dataset `geosecs235`, and use `nls()` to fit the model

$$\mathrm{Tu} = A\exp(-(p-p_0)^2/D^2) + A\exp(-(p+p_0)^2/D^2)$$

where $A$, $p_0$ and $D$ are parameters to be inferred. (See page 201 for a solution.)

### 2.5.6 *Analysis of Variance*

The analysis of variance (ANOVA) is popular in some branches of oceanography, and barely used in others. Introductions to the method may be found in most Statistics textbooks (see, e.g., De Veaux et al. 2006). The R perspective is described briefly in the documentation of `aov()` and `anova()`, with much more detailed treatments of ANOVA and the related method of regression being provided in the texts by Chambers and Hastie (1992) and Faraway (2005), a version of the latter being freely available online as Faraway (2002). These are all general treatments. For marine examples, see texts by Legendre and Legendre (1998) or Borcard et al. (2011), the second of which uses R.

An example with constructed data is sufficient to reveal procedures. Suppose thermometers labelled $T1$, $T2$ and $T3$ each record temperatures in five isothermal water baths. If the instruments all have random errors of 0.001°C, $T2$ has a systematic offset of 0.010°C and $T3$ has a systematic offset of 0.001°C, then an artificial dataset can be created with

```
set.seed(256)
T <- data.frame(T1=10.000 + rnorm(n=5, sd=0.001),
                T2=10.010 + rnorm(n=5, sd=0.001),
                T3=10.001 + rnorm(n=5, sd=0.001))
```

A box plot is a good way to display the data, and

```
boxplot(T, horizontal=TRUE, xlab="Temperature")
```

creates the left panel of Fig. 2.23, which suggests that $T2$ is definitely different from the others, and that $T3$ is possibly offset from $T1$.

The work is simplified by using `stack()`, which returns columns for the dependent variable[21] in `values` and the dependent variable `ind`.

```
Ts <- stack(T)
```

The analysis of variance is carried out with `aov()`, using a formula notation like that used by `lm()`

```
a <- aov(values ~ ind, data=Ts)
```

[21] An alternative to `stack()` is `melt()`, from the `reshape2` package (Wickham 2007). If this is used, then `aov()` must use `value` for `values` and `variable` for `ind`.

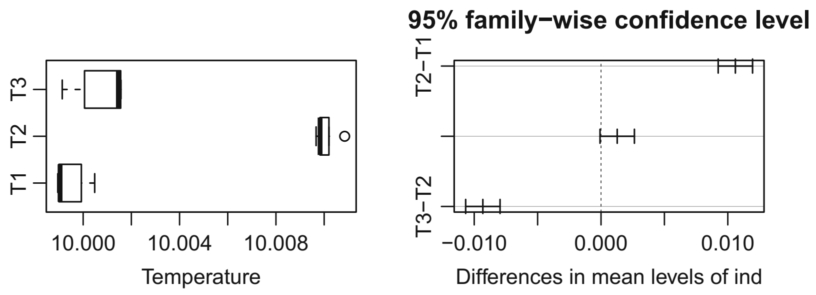

**Fig. 2.23** Results of analysis of variance of simulated data from three thermistors

As with linear regression, `summary()` gives an overview:

```
summary(a)
```

```
            Df    Sum Sq   Mean Sq F value    Pr(>F)
ind          2 0.0003346 1.673e-04   259.3 1.34e-10 ***
Residuals   12 0.0000077 6.500e-07
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

which suggests differences between the instruments. This is consistent with the boxplot of Fig. 2.23, but there is another way to see this graphically, with the Tukey honest significant difference, i.e.

```
plot(TukeyHSD(a))
```

producing the right panel of Fig. 2.23, which suggests that instruments $T1$ and $T3$ are similar, but that $T2$ is different. The procedure has thus flagged the $T2$ anomaly in the constructed data, but not the smaller $T3$ anomaly.

**Exercise 2.37** Increase the value of `n` until the TukeyHSD diagram indicates that $T1$ and $T3$ are producing different values. (See page 201 for a solution.)

## *2.5.7 Partitioning Decision Trees*

The term "regime shift" has been used to describe variations in physical or biological properties of the ocean that take the form of rapid transitions from one quasi-steady state to another. This is a topic of great interest and some controversy; see, e.g., Miller et al. (1994), Rudnick and Davis (2003), deYoung et al. (2008) and Lindegren et al. (2012).

One way to look for regime shifts is with the partitioning tree approach used by `ctree()` in the `party` package (Hothorn et al. 2006). Artificial data

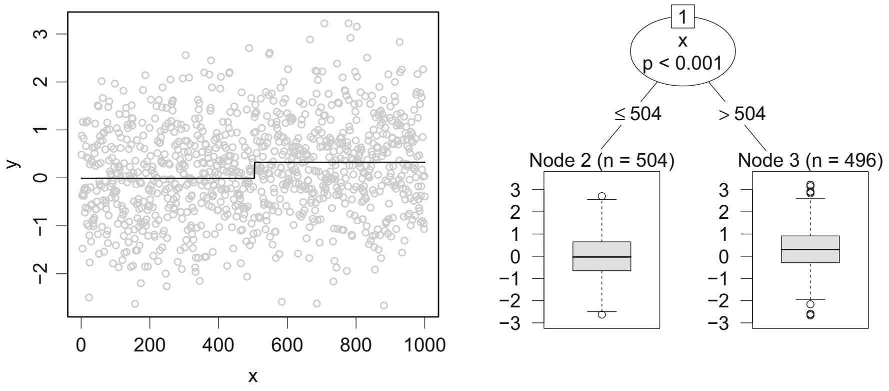

**Fig. 2.24** Analysis of a simulated regime shift. Top: random numbers that shift slightly at $x = 500$. Bottom: diagram produced by `ctree()` in the `party` package

```
set.seed(257)                        # for reproducibility
n <- 1000
x <- 1:n
y <- rnorm(n) + ifelse(x > 500, 1/3, 0)
plot(x, y, cex=0.75, col="gray")
```

as shown in the symbols of the left panel of Fig. 2.24 can illustrate procedures. It is difficult to discern the shift at $x = 500$, but

```
library(party)
p <- ctree(y ~ x)
lines(x, predict(p))
```

adds lines to the diagram that show that `ctree()` finds a shift at $x = 504$, reasonably close to the actual value. More about the fit is obtained with

```
plot(p)
```

which produces the right panel of Fig. 2.24.

Another approach is to use the `changepoint` package (Killick and Eckley 2014; Killick et al. 2016), e.g. a test for a shift in the mean

```
library(changepoint)
cpts(cpt.mean(y, penalty="SIC"))
| [1] 504
```

matches the `ctree()` result.

**Exercise 2.38** Use `ctree()` and `cpt.mean()` to examine the Southern Oscillation Index data (`soi` in the `oce` package) for regime shifts between 1967 and 1985. (See page 202 for a solution.)

## 2.6 Numerical Methods

### 2.6.1 *Sorting*

The `sort()` function returns a sorted vector, while `order()` returns indices that will yield a sorted vector. Thus, for example, `x[order(x)]` yields the same results as `sort(x)`. The `order()` method has the advantage of working with matrices, e.g. the ocean data

```
data(oceans, package="ocedata")
```

may be reordered according to average depth with

```
oceansOrdered <- oceans[order(oceans$AvgDepth), ]
```

and a reverse ordering can be found by supplying `decreasing=TRUE` to `order()`. Ranking can be achieved by a nested call to `order()`, e.g. a column of rank by average depth may be added with

```
oceans$rankByAvgDepth <- order(order(oceans$AvgDepth,
decreasing=TRUE))
```

### 2.6.2 *Root Finding*

As noted in Sect. 2.3.11.4, roots of univariate functions may be found with `uniroot()`. Roots of polynomials can be found with `polyroot()`, e.g. $(x-1)(x+1)$ may be written $a_1 + a_2x + a_3x^2$ with $a = (-1, 0, 1)$, yielding:

```
polyroot(c(-1, 0, 1))
|[1]  1+0i -1+0i
```

### 2.6.3 *Integration*

Numerical integration of a function of a single variable is handled with `integrate()`. For example, $\int_0^\pi \sin\theta \, d\theta$ is calculated (along with an error estimate) with

```
integrate(sin, 0, pi)
|2 with absolute error < 2.2e-14
```

Infinite limits may also be supplied, e.g. for the witch of Agnesi function

```
woa <- function(x, a=1)
    8 * a^3 / (x^2 + 4*a^2)
integrate(woa, -Inf, Inf)
|12.56637 with absolute error < 1.3e-09
```

the integral matches the theoretical value of $4\pi$ to within $2 \times 10^{-15}$.

**Exercise 2.39** Use `integrate()` to calculate the perimeter of an ellipse of major axis $a = 2$ and minor axis $b = 1$. (See page 202 for a solution.)

## *2.6.4 Piecewise Linear Interpolation*

Piecewise-linear interpolation is provided with `approx()`, which returns interpolated values, and `approxfun()`, which returns an interpolating function.

A common use of `approx()` is to interpolate values to a uniform one-dimensional grid. For example, the `ctd` dataset in the `oce` package holds hydrographic data sampled at unevenly spaced pressures.

```
data(ctd, package="oce")
p <- ctd[["pressure"]]
```

so that, e.g., salinity may be interpolated to pressures $(0, 0.5, \ldots)$ dbar with

```
S <- ctd[["salinity"]]
Sinterp <- approx(p, S, seq(0, max(p), 0.5))$y
```

The first two arguments to `approx()` are the independent and dependent variables, and the third is the interpolating grid. Values beyond the data range will be returned as `NA`, although the `rule` argument to `approx()` provides an alternative to that convention.

The use of `approxfun()` may be illustrated with the turbulence measurements of Grant et al. (1962). The dataset `turbulence` in the `ocedata` package contains wavenumber $k$ and one-dimensional spectrum function $\phi$, provided in non-SI units for comparison with this classic paper. A quantity of interest is $\epsilon$, the rate of viscous dissipation of turbulent kinetic energy per unit mass. Under certain conditions, this is $\epsilon = 15\nu \int_0^\infty k^2\phi\, dk$ where $\nu$ is the kinematic viscosity of seawater, motivating a plot of $k^2\phi$ versus $k$, with

```
data(turbulence, package="ocedata")
k <- turbulence$k
phi <- turbulence$phi
plot(k, k^2*phi, pch=20, ylim=c(0, 0.41),
     xlab=expression(k), ylab=expression(k^2*phi))
```

resulting in Fig. 2.25. Grant et al. (1962) reported that $\epsilon = 0.610\,\mathrm{cm}^2/\mathrm{s}^3$, and it may be of interest to see whether a similar value can be recovered by integrating under a function representing the data just plotted, e.g.

```
lfcn <- approxfun(k, k^2 * phi)
```

the results of which are added to the diagram with

```
kk <- seq(min(k), max(k), length.out=100)
lines(kk, lfcn(kk))
```

The integration is performed with

```
I <- integrate(lfcn, min(k), max(k))
```

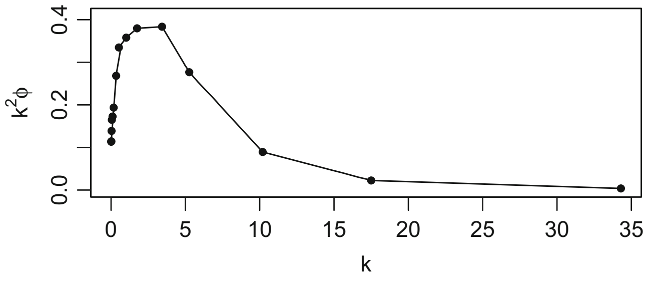

**Fig. 2.25** Turbulence data recorded by Grant et al. (1962), with $k$ wavenumber (1/cm) and $\phi$ one-dimensional velocity spectral function (cm$^3$/s$^2$)

after which calculating $\epsilon$ is a simple matter of extracting the numerical value of the integration and scaling to convert to the cgs units used in the 1960s.

```
nu <- 1e4 * swViscosity(35,10) / swRho(35,10,10,eos=
"unesco")
15 * nu * I$value
|[1] 0.6810874
```

(This $\epsilon$ estimate exceeds the Grant et al. (1962) value by 12%.)

### *2.6.5 Two-Dimensional Interpolation*

The two-dimensional case of interpolating on a rectangular grid is handled with `interp.surface()` from the `fields` package. This does local bilinear interpolation of $z = z(x, y)$ by applying

$$(1-x')(1-y')z_{00} + (1-x')y'z_{01} + x'(1-y')z_{10} + x'y'z_{11} \qquad (2.7)$$

where $(x', y')$ is the relative position of the point within the grid cell that bounds it, and $z_{00}$ is the value at $x' = y' = 0$, $z_{01}$ is that at $x' = 0$, $y' = 1$, etc.

**Exercise 2.40** Use `interp.surface` to find water depth $H$ under the mean Gulf Stream position as defined in the `gs` dataset of the `ocedata` package. Draw a map of the Gulf Stream location along with a graph of how $H$ varies with distance along the path. (See page 203 for a solution.)

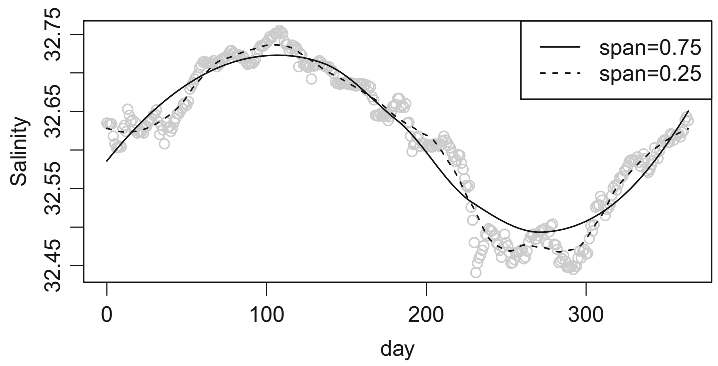

**Fig. 2.26** Using `loess()` for surface salinity at ocean weather station Papa

### 2.6.6 *Locally Weighted Polynomial Fitting*

R has two functions for locally weighted polynomial fitting, `lowess()` and `loess()`. The first is used by `panel.smooth()`, which in turn is used by `plot.lm()`, `coplot()`, etc., so it deserves understanding, but the newer `loess()` is the focus here.

Figure 2.26 shows variation of surface salinity at ocean weather station Papa, plotted with

```
data(papa, package="ocedata")
day <- as.numeric(papa$t - papa$t[1]) / 86400
salinity <- papa$salinity[,1]
plot(day, salinity, ylab="Salinity", col="gray")
```

where two `loess()` fits are shown, illustrating the use of the `span` argument.

```
l <- loess(salinity ~ day)
lines(day, predict(l))
ll <- loess(salinity ~ day, span=0.25)
lines(day, predict(ll), lty="dashed")
legend("topright",lty=1:2,legend=c("span=0.75",
"span=0.25"))
```

### 2.6.7 *Interpolating and Smoothing Splines*

R provides interpolating splines with `spline()` and `splinefun()` and smoothing splines with `smooth.spline()`. Such functions can be quite helpful in dealing with noisy oceanographic data. For example, a smoothing spline may be fitted to the `turbulence` data with

```
s <- smooth.spline(k, k^2*phi)
```

where default values are used for additional arguments that control the degree of smoothing, knot distribution, etc.

The predicted values are found with

```
spred <- predict(s, kk)
```

producing a list with `x` being the `k` supplied to `smooth.spline()` and `y` being the interpolated value, so that

```
lines(spred$x, spred$y, lty="dotted")
```

could add a spline curve to Fig. 2.25.

**Exercise 2.41** Contrast the predictions of interpolating and smoothing splines for the `turbulence` data. (See page 204 for a solution.)

**Exercise 2.42** Create a function returning the prediction of a smoothing spline, and use it to calculate $\epsilon$ as in Sect. 2.6.4. (See page 205 for a solution.)

## *2.6.8 Cluster Analysis*

Cluster analysis may be used to divide a set of data into subsets based on similarity of some property within groups and dissimilarity between them. Different applications call for different measures of similarity, perhaps explaining the diversity of approaches to cluster analysis (Estivill-Castro 2002).

In the popular k means clustering method (Hartigan and Wong 1979), the measure of similarity is Euclidean distance in property space, the square of which is given by

$$\sum_{i=1}^{n} (x_i - \hat{x}_i)^2 \tag{2.8}$$

where $x_1 \dots x_n$ and $\hat{x}_1 \dots \hat{x}_n$ are the coordinates of two points in $n$-dimensional property space. This has a simple interpretation if the coordinates are of an equal type, e.g. representing a geometric location, but other cases are more problematic.

For example, if $x_1$ is salinity and $x_2$ is temperature, as on a temperature-salinity diagram, then the two terms in the expanded sum within (2.8) have different units, so there can be no physical meaning to their addition. Other problems can arise even in a geometrical view, e.g. owing to the vastly different vertical and horizontal extents of oceans. For this reason, (2.8) might be better expressed in nondimensional form, as

$$\sum_{i=1}^{n} \frac{(x_i - \hat{x}_i)^2}{L_i^2} \tag{2.9}$$

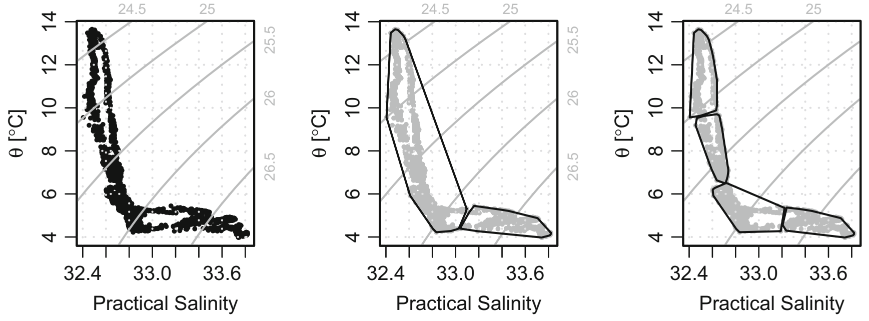

**Fig. 2.27** Demonstration of cluster analysis in temperature-salinity space

where $L_i$ is an appropriate scale in the same unit as $x_i$. In many practical problems, then, a core issue is the selection of the $L_i$ values.

The methodology can be illustrated with temperature and salinity in the `papa` dataset, drawn in the left panel of Fig. 2.27 with.[22]

```
data(papa, package="ocedata")
S <- as.vector(papa$salinity)
T <- as.vector(papa$temperature)
p <- rep(swPressure(-papa$z), each=dim(papa$salinity)[1])
ctd <- as.ctd(S, T, p, longitude=-145, latitude=50)
plotTS(ctd, pch=20, cex=1/2, eos="unesco")
```

Most readers might describe the pattern as having two "arms", one relatively isohaline, the other relatively isothermal. However, opinions might vary, given a request to identify 10 groupings. It helps to use extra oceanographic information in deciding how many clusters to seek. Thought should also be given to the choice of independent variables. For example, in the present case, some applications might call for a transformation to density-spiciness space, which could yield different clusters. Generally, careful attention to the setup can be the key to getting sensible results from cluster analysis.

For the purpose of illustration, temperature and salinity may be used as the coordinates, with `scale()` used to nondimensionalize the variables, after which `kmeans()` may be used to do cluster analysis with 2 and 4 suggested groupings, in the middle and right panels of Fig. 2.27, with

```
plotTSCluster <- function(ctd, k=4)
{
    theta <- swTheta(ctd)
    Stheta <- scale(cbind(S, theta), TRUE, TRUE)
    cl <- kmeans(Stheta, k, nstart=30)
```

[22]It is not strictly necessary to use `as.ctd()` to create a `"ctd"` object, but it makes it easier to create a standardized plot with isopycnals.

```
        plotTS(ctd, col="darkgray", pch=20,cex=0.5,
        eos="unesco")
        which <- cl$cluster
        for (i in 1:k) {
            x <- S[which==i]
            y <- theta[which==i]
            hull <- chull(x, y) # chull() computes
            complex hulls
            hull <- c(hull, hull[1])
            lines(x[hull], y[hull])
        }
    }
    set.seed(268)                   # for reproducibility
    plotTSCluster(ctd, 2)
    plotTSCluster(ctd, 4)
```

in which `chull()` has been used to find the convex hull polygons surrounding the clusters. Note also that `kmeans()` uses random numbers in its search for cluster centres, so repeating a calculation without using `set.seed()` can yield different answers in some cases.

### *2.6.9 Fast Fourier Transforms*

The `fft()` function, which provides forward and inverse fast Fourier transforms (FFT), is used by `convolve()` and `spectrum()`, and it is also useful by itself. It does not normalize in either direction, leaving this to the user, e.g.

```
    fftn <- function(z, inverse=FALSE)
        fft(z, inverse) / sqrt(length(z))
```

defines a 1-D wrapper with a common normalization (see Exercise 5.25 for an application to rotary spectra.)

With `fftn()` thus defined, a test of Parseval's Theorem might be, e.g.

```
    library(testthat)
    x <- rnorm(100)
    X <- fftn(x)
    xx <- fftn(X, TRUE)
    expect_equal(sum(x^2), sum(Mod(X)^2))
```

where the handy `expect_equal()` from the `testthat` package is used to check that the time-series variance matches the integral of the power spectrum. (If the test failed, it would print an error message.)

A check of the invertibility of the `fftn()` formulation might be

```
    expect_equal(x+0i, fftn(fftn(x), inverse=TRUE))
```

## 2.7 Input and Output

R has sufficient flexibility to handle any conceivable file format. Since this is achieved by a fairly long list of functions and arguments, the present discussion covers several pages, despite being a thin summary.

### *2.7.1 Reading from Text Files*

#### 2.7.1.1 Simple Tables

If a file named `table_eg_1.dat` starts with

```
x y
1 1.6180
2 2.7183
3 3.1416
```

then

```
d <- read.table("table_eg_1.dat", header=TRUE)
```

will read the data into a data frame (see Sect. 2.3.8) named `d`. Since `header` is `TRUE`, the column names are inferred from the first line of the file. If the `header` argument is dropped, the columns will be named `V1` and `V2`, unless the `col.names` argument is supplied. This convention applies also to relatives such as `read.csv()`, `read.fwf()` and `read.fortran()` in the `utils` package, and analogues in the `readr` package, which can be faster.[23]

#### 2.7.1.2 Complicated Tables

A Southern Oscillation Index (SOI) dataset[24] starts as follows:

```
1866 -1.2 -0.3 -1.0 -0.7  0.1 -0.9 -0.7  0.7 -0.4  0.1
       1.6 -0.3
1867  0.4 -0.0 -0.0  0.8  0.7 -0.5  0.6  0.5  0.1 -0.7
      -1.2 -1.7
```

One way to read such data would be to create a matrix

```
d <- as.matrix(read.table("../data/soi.dat", header=FALSE))
```

and then construct a decimal-year time vector

[23] A test with a 90 Mb file on the author's machine revealed `read_csv()` to be nearly 6 times faster than `read.csv()`.

[24] http://www.cgd.ucar.edu/cas/catalog/climind/SOI.signal.ascii.

```
y <- d[,1]
year <- seq(from=head(y,1), to=tail(y,1)+11/12, by=1/12)
```

The first step in isolating the SOI values is to drop the first column of `d`, after which the matrix should be transposed with `t()` before creating the vector:

```
soi <- as.vector(t(d[,-1]))
```

It is necessary to account for missing values, equal to `-99.9` in this dataset, but it is risky to check for numerical equality, so

```
missing <- soi < (-90) # parentheses prevents "<-" typo
```

may be preferable. Usually, the next step would be to set the missing values to the R coded value, with `soi[missing] <- NA`, but with this particular dataset, the missing values are all at the end,

```
year <- year[!missing]
soi <- soi[!missing]
```

might be used, if there were no need to retain data length.

#### 2.7.1.3 Line-Based Input

The `readLines()` function reads a file by lines, returning a vector of strings, one per line. To illustrate how this can be useful in decoding complicated data files, a mapping application was used to generate an automobile route from Dalhousie University to the Woods Hole Oceanographic Institution, resulting in an XML file. In this file, geographical locations are delimited by strings `<coordinates>` and `</coordinates>`. Within these blocks are comma-separated values of longitude, latitude and a third item.

The first step in inferring the route is to read the whole file as strings, and to find the portion containing the route

```
d <- readLines("../data/dalwhoi.kml")
start <- grep("^\\ s*<coordinates>\\ s*$", d)
end <- grep("^\\ s*</coordinates>\\ s*$", d)
```

Here, `^` stands for the string start, `\\s*` stands for whitespace, and `$` stands for the string end (see Sect. 2.3.3.3). Next,

```
pathIndices <- seq(start + 1, end - 1)
data <- read.csv(text=d[pathIndices], header=FALSE)
lon <- data$V1
lat <- data$V2
```

reads the locations. Figure 2.28 is then constructed with

```
data(coastlineWorldFine, package="ocedata")
mapPlot(coastlineWorldFine, projection="+proj=merc",
        col="lightgray", longitudelim=range(lon),
        latitudelim=range(lat))
mapLines(lon, lat, lwd=5) # thick line for the route
```

which makes use of a Mercator projection (see Chap. 3).

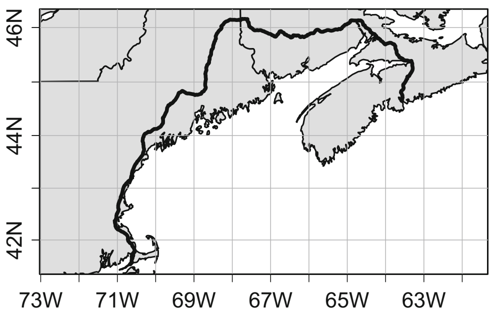

**Fig. 2.28** Road from Dalhousie University to Woods Hole Oceanographic Institution

**Exercise 2.43** Read the Dalhousie-WHOI route using the `XML` package. (See page 205 for a solution.)

**Exercise 2.44** Read the sample CTD file `ctd.cnv`, skipping the header and naming the columns. (See page 206 for a solution.)

#### 2.7.1.4 Reading by Words and Characters

The `scan()` function reads a file a "word" (defined as a character grouping separated by whitespace) at a time, returning the vector of items, e.g. if `../data/soi.dat` contains the SOI data,[25] then

```
tokens <- scan("../data/soi.dat")
length(tokens)
|[1] 1963
```

reveals that `scan()` reads the whole file, finding 1963 words.

At a finer level, `readChar()` can be used to read individual characters, e.g.

```
readChar("../data/soi.dat", 10)
|[1] " 1866  -1."
```

**Exercise 2.45** Read the SOI data with `scan()`. (See page 207 for a solution.)

#### 2.7.1.5 File and Text Connections

When supplied with a filename, most reading functions that are directed at files start afresh with each invocation, e.g.

```
readChar("../data/soi.dat", 30)
|[1] " 1866  -1.2  -0.3  -1.0  -0.7 "
readChar("../data/soi.dat", 30)
|[1] " 1866  -1.2  -0.3  -1.0  -0.7 "
```

[25] http://www.cgd.ucar.edu/cas/catalog/climind/SOI.signal.ascii.

shows that `readChar()` rereads the first 30 characters each time it is called. File connections provide a way to read sequentially through files. The value returned by `file()` is a file connection, and e.g.

```
soi <- file("../data/soi.dat", "r") # second arg
means read-only
readChar(soi, 15)
|[1] " 1866  -1.2  -0"
readChar(soi, 15)
|[1] ".3  -1.0  -0.7 "
```

demonstrates that `readChar()` retains a pointer to file location when its first argument is a file connection, as opposed to a file name.

Connections may also be made to strings, using `textConnection()`, e.g.

```
con <- textConnection("but not a drop to drink")
scan(con, "character", nmax=3)
|[1] "but" "not" "a"
scan(con, "character", nmax=3)
|[1] "drop"  "to"    "drink"
```

Readers with programming experience are likely to see that connections are the key to working with complex files such as are common in oceanography.

### *2.7.2 Reading Binary Data*

R has powerful and flexible tools for working with binary files. This is important for oceanographic work, because many instruments record in binary format. Working with binary data is a somewhat complicated business in any language, but readers with some skill (e.g. those who know the meaning of phrases such as “little endian” and “unsigned int”) should not have difficulty handling their data in R.

The first processing step is typically to read the entire file into a memory buffer, after which the buffer is examined in detail. For example, the `oce` package (Chap. 3) reads acoustic Doppler files with code of the form

```
file <- file(filename, "rb")
seek(file, 0, "end")
fileSize <- seek(file, 0, "start")
```

Here, `file()` is given argument `"rb"` to open the file read-only, in binary format. Then `seek()` is used to “point” to the end of the file. Calling `seek()` a second time moves the pointer to the start of the file and also returns the number of bytes in the file, saved here as `fileSize`. Now, the whole file may be read into a buffer with `readBin()`, in binary (“raw”) form.

```
buf <- readBin(file, "raw", fileSize)
```

Each element of `buf` corresponds to a byte. A common operation is to check those bytes for coded sequences that flag data sections. For example, ADCP files from RDI-Teledynestart with the byte `0x7f` repeated twice, so a test to see if the file might be of this type is

```
probablyRdiAdcp <- buf[1] == 0x7f && buf[2] == 0x7f
```

and this is one of many tests used by `oceMagic()` to infer file type, helping `read.oce()` to select a specialized reading routine (Sect. 3.2).

Another common operation is to match bytes for sequences that occur at arbitrary locations within files, not just at the start. In many files, data are provided in discrete chunks, with each one starting with a particular byte sequence. A single byte is not a good flag for this, because the probability of a random byte matching is 1/256, which is so high that typical files will have many false positives. This explains why the scheme is usually to have a multi-byte sequence, along with other clues, such as checksums, that identify data chunks.

For large or complex files, it can prove convenient to switch from R to C or C++, to gain processing speed and to simplify programming. For example, the calculation of checksums in acoustic Doppler files is relatively simple in C or C++, each of which is very well-suited to computation at the byte level (Fig. 1.1). The efficiencies gained by moving some core calculations to such compiled languages are dramatic. Since these are not languages known to all oceanographers, it is convenient that `oce` handles many important file types, freeing analysts from the need to go beyond R in day-to-day work.

### *2.7.3 Reading Databases*

R can read and write several types of database, e.g. mySQL and SQLite are handled with the `RMySQL` and `RSQLite` packages. The interfaces are similar enough that a single illustration should suffice.

In 2011, the author started a community educational project to monitor sky light using data loggers that measure light levels and feed the results to a collating computer. This collating computer uses an SQLite database named `skyview.db` that was originally created with

```
CREATE TABLE observations(id integer primary key,
                          time int, light_mean real);
```

where `time` is the observation time in Unix seconds and `light_mean` is the mean light level during a sampling interval.

The collating computer updates graphs on webpages on a regular interval, using code that starts with

```
library(RSQLite)
m <- dbDriver("SQLite")
d <- dbConnect(m, dbname="../data/skyview.db")
```

to load the SQLite driver and connect to the database. Then, it is a simple matter of querying the database to extract the columns from the table. For example, the `observations` table is recovered with

```
o <- dbReadTable(d, "observations")
```

which creates `o` as a data frame including all the data in the `observations` table. If the goal were to extract only certain columns, a query such as

```
o<-dbGetQuery(d, "select time, light_mean from
observations")
```

might be used. (The second argument to `dbGetQuery()` is written in SQL, which can be an advantage in working with a large database that can be pared down at a low level, before transmission of data to R.)

**Exercise 2.46** From https://rbr-global.com/support/matlab-tools, get `RSKtools` and extract time and temperature from the SQLite file named `sample.rsk`. (See page 207 for a solution.)

## *2.7.4 Reading NetCDF Files*

The NetCDF file format is popular in oceanography and meteorology, particularly for data that can be expressed as vectors or arrays. Of its several useful features, two that stand out are its use of an endian-independent binary format and its tight binding of data with metadata that indicate units, ranges, data-quality flags, etc.

An example is provided by the 5-degree resolution version of the World Ocean Atlas (see, e.g., Boyer et al. 2009), available from NODC.[26] This provides gridded values of salinity and temperature, etc. Accessing the data is easy with the `ncdf4` package, written by David W. Pierce. The first step is to open a file connection, e.g. with

```
library(ncdf4)
con <- nc_open("../data/woa13_decav_t00_5dv2.nc")
```

after which an overview of the contents is found by printing `con`. The results, which are detailed and much too long to show here, indicate that the file contains coordinate variables named `lon`, `lat` and `depth`, in addition to arrays holding temperature information in different forms. Data may be accessed with `ncvar_get()`, e.g. the atlas grid geometry is recovered with

```
lon <- ncvar_get(con, "lon")
lat <- ncvar_get(con, "lat")
depth <- ncvar_get(con, "depth")
```

and a 3D array containing analysed temperature is recovered with

```
t_mn <- ncvar_get(con, "t_mn")
```

With such NetCDF-specific work complete, it is easy to explore the data. For example, Fig. 2.29 shows latitudinal dependence of sea-surface temperature, created with

[26] https://www.nodc.noaa.gov/OC5/woa13/woa13data.html.

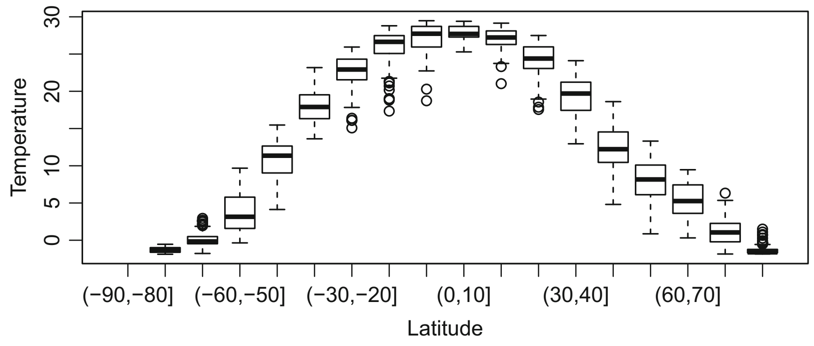

**Fig. 2.29** Variation of sea-surface temperature with latitude and longitude

```
SST <- t_mn[, , depth==0]       # third index is depth
vSST <- as.vector(SST)
vlat <- rep(lat, each=length(lon))
lat10 <- cut(vlat, breaks=seq(-90, 90, 10))
boxplot(vSST ~ lat10, xlab="Latitude",
ylab="Temperature")
```

where `cut()` has been used to break latitude up into 10-degree bands. Note the casting of the temperature matrix into vector form with `as.vector()`, and the creation of a corresponding latitude by replication with `rep()`.

**Exercise 2.47** Plot SST contours with a coastline. (See page 207 for a solution.)

## *2.7.5 Writing Files*

R provides a variety of methods for writing to files. This may be done in textual and binary ways. Conveniently, the names of functions for writing tend to be paired with those for reading, e.g. `writeChar()` is a sibling to `readChar()`.

The `R.matlab` package can be used to write in Matlab format, but the reader is cautioned (as of the writing of this book) that the package is somewhat limited, e.g. for arrays with more than 2 dimensions.

An important R function for writing is `save()`, which stores R objects to a file in a binary format that preserves their structure and contents, suitable for recovery with `load()`. Saving data in this way offers a convenient way to buffer results, avoiding repeating slow calculations across R sessions. Importantly, `save()` uses a binary format that retains full numerical resolution, alleviating any need to decide how many digits to use in a text-based file. Since `save()` and `load()` store endian information, there is no need to worry about spurious results in transferring data between machines of different architectures.

## 2.8 Creating R GUI Systems

As mentioned in Sect. 2.4.15, R can handle graphical input in several ways. A simple method is to use `locator()` to find the location of a user's mouse click on a plot, e.g. to take advantage of the fact that an analyst might be able to detect a data anomaly more easily by examining a graph than by devising a new algorithm.

Such approaches are helpful for simple tasks, but more sophisticated GUI elements are desirable for complex tasks, and that's where the `shiny` package comes in. This package makes it easy to set up GUI systems (or "apps") in R. It has tools for creating GUI elements such as sliders, menus, radio boxes, text boxes, etc., in addition to a powerful and flexible system for connecting such elements to general R code. For example, a `shiny` app can be set up so that radio buttons control whether to represent data with contours or images, with slider bars to control parameters used in smoothing data, etc. The `shiny` system also has good support for mouse clicks and drags within a plotting area, which can be helpful in selecting data to flag or subregions to be replotted at higher magnification. Importantly, the user's actions can be accessed throughout the app, which means that a user action in one panel of a plot can control the display in another panel, providing an escape from the focus limitation of multi-panel plots that was mentioned in Sect. 2.4.15.

There are many online tutorials dealing with `shiny`, those provided by `Rstudio` being noteworthy.[27] A good way to learn `shiny` is to download a tutorial app and start modifying it. For example, a common way to search for errors in CTD data is to look for outliers on a $T$–$S$ plot. A plot-interaction-exclude app provided on the `Rstudio` website[28] addresses a similar task, and so it provides a starting point for a CTD-editing app.

It can be helpful to divide app code into two files, one defining the user interface and the other defining the actions associated with the interface elements. For the former, saving

```
library(shiny)
verticalLayout(wellPanel(plotOutput("plot", brush="brush")),
               wellPanel(actionButton("toggle", "Toggle"),
                         actionButton("reset", "Reset"),
                         textOutput("excluded")))
```

to a file named `ui.R` sets up an interface with a plot area ("`wellPanel`" in `shiny` parlance) at the top of the window and an action area below, as illustrated in Fig. 2.30. The user is invited to drag the mouse to select ("brush") bad data, and then click "Toggle" to switch the status of the indicated points from good to bad (and vice-versa, for corrections). The other items of the second `wellPanel()` will be used to list the indices of suspicious data, and to provide a way to reset the analysis.

[27] https://www.rstudio.com/products/shiny.

[28] http://shiny.rstudio.com/gallery/plot-interaction-exclude.html.

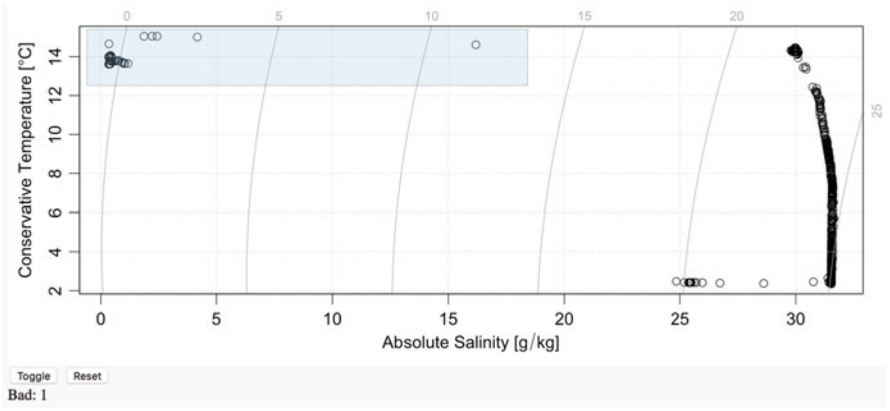

**Fig. 2.30** Using the CTD-edit `shiny` app. A point with spurious salinity of 99 has already been designated as bad, and eliminated from the plot. The mouse has been dragged over low-salinity values that are suspect, and these will disappear when "Toggle" is pressed

These actions are accomplished and demonstrated using a built-in dataset with the following code, saved in a file named `server.R`.

```
library(shiny)
library(oce)
data(ctdRaw)
ctd <- ctdRaw
shinyServer(function(input, output) {
  n <- length(ctd[["pressure"]])
  vals <- reactiveValues(keep=rep(TRUE, n))
  output$plot <- renderPlot({
    plotTS(subset(ctd, vals$keep), eos="gsw")
    discard <- subset(ctd, !vals$keep)
    points(discard[["SA"]], discard[["CT"]], pch=20,
    col="red")
  })
  output$excluded <- renderText({
    paste("Bad:", paste(which(!vals$keep),
    collapse=" "))
  })
  observeEvent(input$toggle, {
    df <- data.frame(SA=ctd[["SA"]], CT=ctd[["CT"]])
    res <- brushedPoints(df, input$brush,
                         "SA", "CT", allRows=TRUE)
    vals$keep <- xor(vals$keep, res$selected_)
  })
```

```
    observeEvent(input$reset, {
      vals$keep <- rep(TRUE, n)
    })
  })
```

In examining such code, the reader would benefit from reading the documentation for `reactiveValues()` and `observeEvent()`, for these are key tools used to control interactions throughout the app.

**Exercise 2.48** Extend the CTD-edit `shiny` app to read data from a file, set flags for bad data and save the result to a file. (See page 208 for a solution.)

## 2.9 Debugging

Debugging tools are an important part of any computing system, and a good topic with which to end this tutorial. R comes with a simple but effective set of debugging tools that should become familiar to any user who writes programs of significant complexity.

Debugging is best done in an interactive environment. The present discussion describes actions that can be taken in the R console or in an editor, but readers who are using the `Rstudio` system will find that there are GUI actions for the procedures outlined here.[29] If `file.R` produces an error when run from the (presumed Unix) operating system with

```
Rscript file.R
```

then the first step is to launch an interactive R environment, and execute

```
source("file.R")
```

to replicate the problem. Then, using

```
traceback()
```

will reveal the spot where the error occurred, along with an execution trace indicating how control arrived at that spot. If the error is obvious, `file.R` can be altered and re-sourced, to work through whatever solution occurs to the coder. It is also helpful to set the system up to trigger the debugger upon errors.

Sometimes an error will be reported long after something was done incorrectly. (For example, line 100 might contain an attempt to read a file that was incorrectly named in line 50.) For this reason, it can be helpful to add lines that print variables at strategic spots in the code. This can be time consuming, so a better approach can be to use `browser()` or `debug()`. For example, with error at line 200, a good approach might be to add

```
browser()
```

[29] RStudio has a variety of other helpful features, e.g. a code-completing editor and a code-analysis tool that can recommend alterations that may make code more robust.

at line 199. When R gets to that spot, the prompt will change, and the user will be able to examine any variable of interest. Actually, any R command can be executed, so the user is free to try to find the problem by plotting, etc. There are also some control commands that can be executed in a `browser()` session, the most useful of which are `n` (or an empty line), which advances to the next step of execution, and `Q`, which exits the browser. Although it is simple, `browser()` is tremendously helpful in finding errors of any kind, and it may be the most important debugging tool a R programmer can learn.

Sometimes it is not desired to edit the code, so inserting calls to `browser()` is not an option. For this purpose, `debug()` is handy. It takes as its first argument the name of a function. Then, when execution is started, everything proceeds normally until that function is encountered, whereupon control is handed over to `browser()`. It is also possible to instruct `Rstudio` to act as though a `break()` call had been been inserted at a specified spot in the code, without actually modifying the source file.

In the special case of the `oce` package, another debugging method is to set the `debug` argument to an integer higher than zero, so that functions print a record of some important aspects of their processing. This is just one of the many practical aspects of the `oce` package that are sketched in the next chapter.