Introductory Guide
to
S-Plus

Final Version

B.D. Ripley
Professor of Applied Statistics,
University of Oxford
e-mail: ripley@stats.ox.ac.uk

24 August 1994

Preface

This guide was originally written for graduate students in Statistics at the University of Oxford. The first versions were based closely on notes by Dr. Bill Venables of the Department of Statistics at the University of Adelaide, but have been updated to reflect later versions of S, the extensions of S-Plus and local facilities. Several sections, in particular 4, 6 and 11, remain close to Dr. Venables' original material. This guide will no longer be updated, following the publication of Venables & Ripley (1994). [See p. 1. Where that takes a significantly better approach than earlier editions of these notes, the material formerly here has been dropped.]

The guide is to S-Plus, but much of it will be relevant to users of the underlying S. Extensions which are only in S-Plus include dynamic graphics (§6.3, `brush` and `spin`) and the classical statistics functions (§9). The terminology of this guide is intended to be precise, only referring to S-Plus rather than S for features unique to S-Plus.

These notes were written for a particular environment, S-Plus 3.2 on Sun SparcStations running the Open Windows windowing system. You will find a number of differences depending on your local environment. It will help to have the library `ripley` available — it should be in the same source as these notes. It can be also be obtained by anonymous ftp from

`markov.stats.ox.ac.uk` (163.1.20.1)

in file `pub/S/ripley.sh.Z`. It is available from `statlib` (see Section A.2) as

```
send ripley from S
```

Alternatively, `library(MASS)` from Venables & Ripley (1994) can be used.

B.D. Ripley,
University of Oxford,
24th August, 1994.

Contents

1 Introduction

S is a statistical language developed at AT&T's Bell Laboratories. S-Plus is a binary distribution of S, with added functions, produced by the StatSci Division of MathSoft in Seattle. The S system was radically re-designed in the 1988 release and known as *'New S'*. In August 1991 a new release of what is once again called S consisted of a moderate revision of *'New S'* together with far-ranging extensions. S-Plus 3.0 was introduced in late 1991, based on that release of S, with numerous additional features. S-Plus 3.1 was released at the very end of 1992, and S-Plus 3.2 in very early 1994.

The main references are:

R.A. Becker, J.M. Chambers and A.R. Wilks (1988) *The NEW S language.* Wadsworth & Brooks/Cole.

J.M. Chambers and T.J. Hastie (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

It is not the intention of this guide to replace the books. Rather these notes are intended as a brief introduction to the capabilities of the S programming language and to how to perform some common statistical procedures within S. Users of S-Plus will need to consult both books, probably frequently. Both books contain some reference documentation, but the on-line versions (see §1.2) are later and definitive.

There also manuals for S-Plus itself, whose organization differs from release to release.

Other books include

W.N. Venables and B.D. Ripley (1994) *Modern Applied Statistics with S-Plus.* New York: Springer ISBN 0-387-94350-1

which goes far beyond the coverage of this guide, including many topics (such as robust statistics, non-linear regressions, modern regression, survival analysis, tree-based models, time series and spatial statistics) not covered here, as well as in greater depth on what is covered.

1.1 Starting and Finishing

To start S-Plus, type the command

```
machine% Splus
```

After a short while (and, the first time, an initialization message) you get the S-Plus prompt[1]:

```
>
```

This is waiting for input from you.

Technically S is a *function language* with a very simple syntax. Like most Unix based packages it is *case sensitive*, so `A` and `a` are different variables. Elementary commands consist of either *expressions* or *assignments*. If an expression is given as a command, it is evaluated, printed, and the value is discarded[2]. An assignment also evaluates an expression and passes the value

[1] which can be changed, but the default is assumed here

[2] In fact it is kept in the (hidden) variable `.Last.value` and so can be retrieved from the 'bin'.

to a variable but the result is not printed automatically. An expression can be as simple as `2 + 3` or a complex function call. Assignments are indicated by the *assignment operator* `<-` or `_`. (As the first needs two keystrokes, lazy typists use the second. However, the first is easier to read.) For example,

```
> 2+3
[1] 5
> mean(hstart)
[1] 137.9944
> m <- mean(hstart); v <- var(hstart)
> m/sqrt(v)
[1] 3.174021
```

The `[1]` states that the answer is starting at the first element of a vector.

Commands are separated either by a semi-colon, `;`, or by a newline. If a command is not complete at the end of a line, S will give a different prompt, namely

```
+
```

on second and subsequent lines and continue to read input until the command is syntactically complete.

S can be extended by writing new functions, which then can be used exactly as built-in functions (and can even replace them). How to write your own functions is covered in section 12.

1.2 Getting Help

S has an inbuilt help facility similar to the man facility of Unix. To get more information on any specific named function or dataset, for example `mean`, the command is

```
> help(mean)
```

For a feature specified by special characters, and in a few other cases (one is `"swiss"`), the argument must be enclosed in double quotes, making it a 'character string':

```
> help("[[")
```

Help uses a window which overlays your main window. The pager accepts a number of options, including `space` for the next page and `q` to quit. (Other useful options are `1G` to go to the top and `control-b` to go back a page.) If you prefer, a separate help window (which can be left up) can be obtained by the argument `window=T`. Another way to get help is by

```
> ?mean
```

Short help is given by the function `args`.

S-Plus also has a window-based help facility, started by

```
> help.start(gui="openlook")
```

Click with the left mouse button on items to select categories and items. The help window can be left up, or removed by

```
> help.off()
```

It is **not** advisable to quit S-Plus windows from the frame menu.

1.3 Hardcopy Output

Graphics are printed by holding down the right button on the `graph` menu in a `openlook()` window (see §6) and releasing over the print item. This will print on the nearest laser printer (or that selected by your `PRINTER` environment variable).

To record a session cut-and-paste to a `textedit` window, then remove your mistakes (if any) and save as a Unix file.

2 Datasets

Datasets are stored in a directory `~/.Data`. They are permanent, so all the objects you create are retained until explicitly deleted. (As the directory name `.Data` begins with `.` it will normally be hidden in file listings from Unix by `ls`.) If there is a `.Data` directory in the current directory when S is invoked, that directory is used rather than `~/.Data`. This provides one way to organize your S, using separate directories for each project.

In S, to get a list of names of the objects currently defined use the command

```
> objects()
```

Your own functions are also stored in `.Data`. To find out whether an object is a function or dataset, and what is in it, just type its name at the prompt, e.g.

```
> stack.x
```

```
> plot
```

This prints out the function, dataset, ... In the later versions of S it may print a *short summary* of the object. To get the full details, use

> `print.default(`*object*`)`

When S looks for an object, it searches in turn through a sequence of directories known as the *search list*. Usually the first entry in the search list is the `.Data` sub-directory of the current working directory. The names of the directories currently on the search list can be found by the function

```
> search()
```

The names of the objects held in any directory on the search list can be displayed by giving the `ls` function an argument. For example `objects(2)` lists the contents of the second directory in the search list. Normally the second, third and fourth directories are built-in functions, and the fifth, sixth and seventh contain standard datasets

Extra search directories can be added to this list with the `attach(...)` function and removed with the `detach(...)` function, details of which can be found in the manuals or the `help` facility. Note that attached directories are searched *after* the `.Data` directory in the order last attached to first attached.

To remove objects permanently the function `rm` is available:

```
> rm(x,y,z,in,junk,temp)
```

The function `remove(...)` can be used to remove objects with non-standard names.

Warning

Objects in your `.Data` *directory will take precedence over system objects of the same name.* This is a frequent cause of rather obscure errors, and can cause apparently correct behaviour but erroneous results. Avoid using names such as `c, s, t, glm, range, tree` for your own objects. If you get peculiar errors, clean up your `.Data` directory and try again!

S keeps a record of commands in the `.Audit` file in the `.Data` directory. This is a hidden file and can grow rather large. Use (from the Unix command line)

```
Splus TRUNC_AUDIT 0
```

occasionally to clean out the audit file entirely (or omit the `0` to keep the last 0.5Mb).

3 A First Session

The sample session given below is intended to show by example some of the capabilities of the system. Work through the session given by the commands on the left of the page. Some clues as to what is going on are given at the right hand side of the page.

`machine% Splus`	Start the session.
`> openlook()`	Open the graphics window.
`> library(ripley)`	Add a library of functions and datasets.
`> help(trees)`	use *q* to quit
`> trees`	Print out a *data frame* of the trees data
`> attach(trees)`	so that we can use names *diam* etc
`> hist(diam)`	Histogram as counts.
`> hist(diam, nclass=10, probability=T)`	as probability density
`> help(hist)`	
`> stem(diam)`	Stem-and-leaf plot.
`> plot(diam, volume)`	Scatter plot.
`> trees.lm <- lm(volume ~ diam)`	linear regression
`> summary(trees.lm)`	summary of fit
`> anova(trees.lm)`	analysis of variance table
`> abline(trees.lm)`	plot line on scatter plot
`> identify(diam, volume, height)`	Move mouse to plot and click with left button to see what height is. Click middle button to quit.
`> par(mfrow=c(1,2))`	set up 1 row, 2 cols for plots
`> plot(trees.lm)`	plots of fitted values and |residuals| vs fitted value.
`> par(mfrow=c(1,1))`	one plot again.
`> qqnorm(residuals(trees.lm))`	normal probability plot of residuals
`> qqnorm(studres(trees.lm))`	and of Studentized residuals
`> qqline(studres(trees.lm))`	line through quartiles
`> pairs(trees)`	all pair-wise scatter plots
`> brush(cbind(diam, height, volume))`	rotate points in 3D, select and de-select points. Click on `quit` to end
`> trees.lm2 <- lm(volume ~ diam + height)`	multiple regression. Try functions as before
`> trees.lm3 <- lm(log(volume) ~ log(diam) + log(height))`	
`> detach("trees")`	to avoid any confusion
`> help(road)`	
`> attach(road)`	
`> plot(drivers, deaths)`	
`> plot(drivers, deaths, log="xy")`	
`> state <- row.names(road)`	
`> identify(drivers, deaths, state)`	Find the ‘odd’ states.
`> plot(fuel, deaths, log="xy")`	
`> identify(fuel, deaths, state)`	
`> road.mat <- cbind(drivers, fuel, deaths)`	Set up a matrix
`> pairs(road.mat)`	Look at pattern of all three
`> brush(road.mat, rowlab=state, spin=F)`	Use mouse to highlight points and check their identity. Then click on `quit`
`> q()`	Finish session

4 Simple Data Manipulation

The basic data objects in S are *vectors*, *arrays*, *lists* and *data frames*.

4.1 Vectors

S operates on named *data structures*. The simplest such structure is the *vector*, which is a single entity consisting of an ordered collection of numbers. To set up a vector named `x`, say, consisting of five numbers, namely 10.4, 5.6, 3.1, 6.4 and 21.7, use the S command

```
> x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
```

This is an *assignment* statement using the *function* `c(...)` taking an arbitrary number of vector *arguments* and whose value is the vector of its arguments.

A number occurring by itself in an expression is taken as a vector of length one.

Assignments can also be made in the other direction, using the obvious change in the assignment operator. So the same assignment could be made using

```
> c(10.4, 5.6, 3.1, 6.4, 21.7) -> x
```

If an expression is used as a complete command, the value is printed *and lost*. So now if we were to use the command

```
> 1/x
```

the reciprocals of the five values would be printed (and, of course, the value of `x` would be unchanged).

4.2 Vector Arithmetic

Vectors can be used in arithmetic expressions, in which case the operations are performed element-by-element. Vectors occurring in the same expression need not all be of the same length. If they are not, the value of the expression is a vector with the same length as the longest vector which occurs in the expression. Shorter vectors in the expression are *recycled* as often as need be (perhaps fractionally) until they match the length of the longest vector. In particular a constant is simply repeated. So with the above assignments the command

```
> v <- 2*x + y + 1
```

generates a new vector `v` of length 11 constructed by adding together, element-by-element, `2*x` repeated 2.2 times, `y` repeated just once, and `1` repeated 11 times.

The elementary arithmetic operators are the usual `+`, `-`, `*`, `/` and `^` for raising to a power. In addition all of the common arithmetic functions are available. `log`, `log10`, `exp`, `sin`, `cos`, `tan`, `sqrt`, and so on, all have their usual meaning. `max` and `min` select the largest and smallest elements of an vector respectively. `range` is a function whose value is a vector of length two, namely `c(min(x),max(x))`. The element-by-element maximum and minimum of two or more vectors are given by `pmax` and `pmin`. `length(x)` is the number of elements in `x`, `sum(x)` gives the total of the elements in `x` and `prod(x)` their product.

Two statistical functions are `mean(x)`, which evaluates to `sum(x)/length(x)` and `var(x)`, which gives the value `sum((x-mean(x))^2)/(length(x)-1)`, the sample variance. If the argument to `var(...)` is an $n \times p$ matrix the value is a $p \times p$ sample covariance matrix obtained from regarding the rows as independent p-variate sample vectors.

`sort(x)` returns a vector of the same size as `x` with the elements arranged in increasing order. Other, more flexible, sorting facilities are available (see `order(...)` which produces a permutation to do the sorting, and `sort.list`).

4.3 Generating Regular Sequences of Numbers.

S has a number of facilities for generating commonly used sequences of numbers. For example `1:30` is the vector `c(1,2,...,29,30)`. The colon operator has highest priority within an expression, so, for example `2*1:15` is the vector `c(2,4,6,...,28,30)`. Put `n <- 10` and compare the sequences `1:n-1` and `1:(n-1)`.

The construction `30:1` may be used to generate a backwards sequence.

The function `seq(...)` is a more general facility for generating sequences. It has five arguments, only some of which may be specified in any one call. The first two arguments, if given, specify the beginning and end of the sequence, and if these are the only two arguments given the result is the same as the colon operator. That is, `seq(2,10)` is the same vector as `2:10`.

Parameters to `seq(...)`, and to many other S functions, can also be given in named form, in which case the order in which they appear is irrelevant. The first two parameters may be named `from=`*value* and `to=`*value*; thus `seq(1,30)`, `seq(from=1,to=30)` and `seq(to=30,from=1)` are all the same as `1:30`. The next two parameters to `seq(...)` may be named `by=`*value* and `length=`*value*, which specify a step size and a length for the sequence respectively. If neither of these is given, the default `by=1` is assumed.

For example

```
> seq(-5, 5, by=.2) -> s3
```

generates in `s3` the vector `c(-5.0,-4.8,-4.6,...,4.6,4.8,5.0)`. Similarly

```
> s4 <- seq(length=51, from=-5, by=.2)
```

generates the same vector in `s4`.

The fifth parameter may be named `along=`*vector*, which if used must be the only parameter, and creates a sequence `1, 2, ..., length(`*vector*`)`, or the empty sequence if the vector is empty (as it can be).

A related function is `rep(...)` which can be used for replicating a structure in various complicated ways. The simplest form is

```
> s5 <- rep(x, times=5)
```

which will put five copies of `x` end-to-end in `s5`.

4.4 Logical Vectors. Missing Values

As well as numerical vectors, S allows manipulation of logical quantities. The elements of a logical vector have just two possible values, represented formally as `F` (for 'false') and `T` (for 'true'). (`TRUE` and `FALSE` are also valid representations.)

Logical vectors are generated by *conditions*. For example

```
> temp <- x>13
```

sets `temp` as a vector of the same length as `x` with values `F` corresponding to elements of `x` where the condition is *not* met and `T` where it is.

The logical operators are `<`, `<=`, `>`, `>=`, `==` for exact equality and `!=` for inequality. In addition if `c1` and `c2` are logical expressions, then `c1 & c1` is their intersection (and), `c1 | c2` is their union (or) and `! c1` is the negation of `c1`.

Logical vectors may be used in ordinary arithmetic, in which case they are *coerced* into numeric vectors, `F` becoming `0` and `T` becoming `1`. However there are situations where logical vectors and their coerced numeric counterparts are not equivalent.

In some cases the components of a vector may not be completely known. When an element or value is "not available" or a "missing value" in the statistical sense, a place within a vector may be reserved for it by assigning it the special value `NA`. In general any operation on an `NA` becomes an `NA`. The motivation for this rule is simply that if the specification of an operation is incomplete, the result cannot be known and hence is not available.

The function `is.na(x)` gives a logical vector of the same size as `x` with value `T` if and only if the corresponding element in `x` is `NA`.

```
> ind <- is.na(z)
```

4.5 Character Vectors

Character quantities and character strings are used frequently in S, for example as plot labels. They are denoted by a sequence of characters delimited by the double quote character. E.g. `"x-values"`, `"New iteration results"`. Single quotes can also be used, in matching pairs.

Character strings may be collected into a vector by the `c(...)` function; examples of their use will emerge frequently.

The `paste(...)` function takes an arbitrary number of character string arguments and concatenates them into a single character string. Any numbers given among the arguments are coerced into character strings in the same way they would be if they were printed. The arguments are by default separated in the result by a single blank character, but this can be changed by the named parameter, `sep=`*string*, which changes it to *string*, possibly empty.

For example

```
> labs <- paste(c("X","Y"), 1:10, sep="")
```

makes `labs` the character vector `("X1", "Y2", "X3", ..., "X9", "Y10")`. Note in particular that recycling of short vectors takes place here too; thus `c("X", "Y")` is repeated 5 times to match the sequence.

The elements of a vector can be named (as well as numbered) by assigning a character vector to its `names` attribute, e.g.

```
> costs <- c(26, 45, 67, 33, 51)
> names(costs) <- c("banana", "apple", "orange", "fig", "kiwi")
> costs
 banana apple orange fig kiwi
     26    45     67  33   51
```

4.6 Index Vectors. Selecting and Modifying Subsets of a Data Set

Elements of a vector may be extracted by specifying the element in square brackets, e.g. `x[5]`. More generally, subsets of a vector (or any expression that evaluates to a vector) may be selected by appending to the name of the vector an *index vector* in square brackets. Such index vectors can be any of four distinct types:

1. **A logical vector.** In this case the index vector must be of the same length as the vector from which elements are to be selected. Values corresponding to `T` in the index vector are selected and those corresponding to `F` omitted. For example

   ```
   > y <- x[!is.na(x)]
   ```

   creates (or re-creates) an object `y` which will contain the non-missing values of `x`, in the same order. Note that if `x` has missing values, `y` will be shorter than `x`. Also

   ```
   > (x+1)[(!is.na(x)) & x>0] -> z
   ```

   creates an object `z` and places in it the values of the vector `x+1` for which the corresponding value in `x` was both non-missing and positive.

2. **A vector of positive integral quantities.** In this case the values in the index vector must lie in the the set $\{$`1, 2, ..., length(x)`$\}$. The corresponding elements of the vector are selected and concatenated, *in that order*, in the result. The index vector can be of any length and the result is of the same length as the index vector. For example `x[6]` is the sixth component of `x` and

   ```
   > x[1:10]
   ```

   selects the first 10 elements of `x` (assuming `length(x)` $\geq$ `10`). Also

   ```
   > c("x","y")[rep(c(1,2,2,1),times=4)]
   ```

   (an admittedly unlikely thing to do) produces a character vector of length 16 consisting of `"x", "y", "y", "x"` repeated four times.

3. **A vector of negative integral quantities.** In this case the index vector specifies the values to be *excluded* rather than included. Thus

   ```
   > y <- x[-(1:5)]
   ```

   gives `y` all but the first five elements of `x`.

4. A vector of character strings. This possibility only applies where an object has a `names` attribute to identify its components. In this case a subvector of the names vector may be used in the same way as the positive integral labels in 2.

```
> lunch <- fruit[c("apple","orange")]
```

This option is particularly useful in connection with data frames (see §4.9).

An indexed expression can also appear on the receiving end of an assignment, in which case the assignment operation is performed *only on those elements of the vector*. The expression must be of the form `vector[`*index_vector*`]` as having an arbitrary expression in place of the vector name would not make sense.

The vector assigned must match the length of the index vector, and in the case of a logical index vector it must again be the same length as the vector it is indexing.

For example

```
> x[is.na(x)] <- 0
```

replaces any missing values in `x` by zeros and

```
> y[y<0] <- -y[y<0]
```

has the same effect as

```
> y <- abs(y)
```

4.7 Arrays

An array can be considered as a multiply subscripted collection of data entries of the same type, for example numeric, logical or character string.

An array is defined by having a dimension vector, a vector of positive integers. If its length is `k` then the array is `k`–dimensional. The values in the dimension vector give the upper limits for each of the `k` subscripts. The lower limits are always 1. Suppose, for example, `z` is a vector of 1500 elements. The assignment

```
> dim(z) <- c(3,5,100)
```

allows `z` to be treated as a $3 \times 5 \times 100$ array.

Other functions such as `matrix(...)` and `array(...)` are available for simpler and more natural looking assignments in special cases, e.g.

```
> z <- array(z, c(3,5,100))
```

```
> z <- matrix(z, 3, 5))
```

The values in the data vector give the values in the array in the same order as they would occur in Fortran, that is, with the first subscript moving fastest and the last subscript slowest. For example if the dimension vector for an array, say `a`, is `c(3,4,2)` then there are $3 \times 4 \times 2 = 24$ entries in `a` and the data vector holds them in the order `a[1,1,1], a[2,1,1], ..., a[2,4,2], a[3,4,2]`. To make life easier, `matrix` has a `byrow=T` parameter for data presented by row rather than by column.

Individual elements of an array may be referenced by giving the name of the array followed by the subscripts in square brackets, separated by commas. More generally, subsections of an array may be specified by giving a sequence of *index vectors* in place of subscripts; however *if any index position is given an empty index vector, then the full range of that subscript is taken*. Thus `a[2,,]` is a 4×2 array with dimension vector `c(4,2)` and data vector

```
a[2,1,1],a[2,2,1],a[2,3,1],a[2,4,1],a[2,1,2],a[2,2,2],a[2,3,2],a[2,4,2],
```

in that order. `a[,,]` stands for the entire array, which is the same as omitting the subscripts entirely and using `a` alone.

Arrays may be used in arithmetic expressions and the result is an array formed by element-by-element operations on the data vector. The dimension vectors of operands generally need to be the same, and this becomes the dimension vector of the result. So if `A`, `B` and `C` are all similar arrays, then

```
> D <- 2*A*B + C + 1
```

makes `D` a similar array with data vector the result of the evident element-by-element operations. The matrix multiplication operator is `%*%` .

There are extensive matrix manipulation facilities, including transposes and eigenvalue, Cholesky, QR and singular-value decompositions. See help on `t`, `eigen`, `chol`, `qr` and `svd`.

Any dimension of an array can be given a set of names using `dimnames`, but is usually easier to use the facilities of data frames.

Matrices can be built up from given vectors and matrices by the functions `cbind(...)` and `rbind(...)`. Informally, `cbind(...)` forms matrices by binding together vectors or matrices horizontally, or column-wise, and `rbind(...)` vertically, or row-wise.

4.8 Lists

An S *list* is an object consisting of an ordered collection of objects known as its *components*. There is no particular need for the components to be of the same mode or type, and, for example, a list could consist of a numeric vector, a logical value, a matrix, a character array, a function, and so on.

Components are always *numbered* and may always be referred to as such. If `trees` is a list, then the function `length(trees)` gives the number of (top level) components it has, specified as `trees[[1]]`, `trees[[2]]` and so on.

Components of lists may also be *named,* and in this case the component may be referred to either by giving the component name as a character string in place of the number in double square brackets, or, more conveniently, by giving an expression of the form

> *name*`$`*component_name*

for the same thing. This is a very useful convention as it makes it easier to get the right component if you forget the number, and is strongly advised. You can find out the names of the components by

> `names(`*names*`)`

and this generates much less output that printing the object, which will achieve the same purpose.

The names of components may be abbreviated down to the minimum number of letters needed to identify them uniquely. Most of the datasets are in fact lists (or can be treated as lists), so we could refer to the component `diam` of the `trees` data as `trees$d`. Similarly, many S functions return lists of results.

It is important to distinguish `trees[[1]]` from `trees[1]`. "`[[...]]`" is the operator used to select a single element of a list, whereas "`[...]`" is a general subscripting operator for vectors. Fortunately, numbered components are needed very rarely.

New lists may be formed from existing objects by the function `list(...)`. An assignment of the form

```
> trees <- list(diam=tree.d, height=tree.h, volume=tree.v)
```

sets up a list `tree` of 3 components using the existing objects `tree.d`, `tree.h` and `tree.v` for the components and giving them names as specified by the argument names (which can be chosen freely). If these names are omitted, the components are numbered only.

Lists can be `attach`-ed as well as directories, and this allows their components to be accessed as if they were stand-alone entities. Thus in the `trees` example we could have

```
> attach(trees)

> mean(height)
```

It is wise to `detach("trees")` after use to avoid any nasty surprises.

4.9 Data Frames

Data frames were introduced in the August 1991 release of S, and can be thought of as closely-coupled lists of data vectors of the same length. Unlike matrices, the data vectors can be of different types, including character data. Both the rows and columns can be labelled. Consider the data frame `road` from `library(ripley)`:

```
> road
        deaths drivers  popden rural temp  fuel
Alabama    968     158    64.0  66.0   62 119.0
 Alaska     43      11     0.4   5.9   30   6.2
  ................
     Mo   1289     234    63.0 100.0   40 180.0
   Mont    259      38     4.6  72.0   29  31.0
```

which has both row and column labels. The columns can be treated as components of a list:

```
> road$rural
 [1]  66.0   5.9  33.0  73.0 118.0  73.0   5.1   3.4   0.0  57.0  83.0  40.0
[13] 102.0  89.0 100.0 124.0  65.0  40.0  19.0  29.0  17.0  95.0 110.0  59.0
[25] 100.0  72.0
```

and the structure can be treated as a two-dimensional array:

```
> road[2,4]
 Alaska
    5.9
> road["Mo", "temp"]
 Mo
 40
> road["Mo",]
   deaths drivers popden rural temp fuel
Mo   1289     234     63   100   40  180
```

Note how the row label is carried along.

Data frames can be `attach`-ed just as lists can, and this allows their columns to be accessed as if they were named vectors.

A data frame can be created from vectors and matrices by the `data.frame` function. For example:

```
> treeframe <- data.frame(diam=tree.d, height=tree.h, volume=tree.v)
```

If the columns are not named, they pick up the names of the vectors, so

```
> treeframe <- data.frame(tree.d,tree.h,tree.v)
```

gives

```
   tree.d tree.h tree.v
 1    8.3     70   10.3
 2    8.6     65   10.3
 3    8.8     63   10.2
 4   10.5     72   16.4
 5   10.7     81   18.8
 ......
```

Character vectors given to `data.frame` are automatically treated as factors (see §10), unless specified within a `I()` function.

5 Reading data into S

Data objects will usually be read as values from external files. This is done most conveniently with the `scan(...)` function. To read a vector from the keyboard we can use

```
> counts <- c(2,3,3,4,3,2,1,3,8,11,6,6,7,12,11,11,

+    117,121,47,22,85,98,43,20,119,209,68,43,67,99,46,33)
```

or

```
counts <- scan()
2 3 3 4 3 2 1 3 8 11 6 6 7 12 11 11
117 121 47 22 85 98 43 20 119 209 68 43 67 99 46 33
```

Input is terminated by a blank input line (from the terminal only, despite the documentation) or by EOF (ctrl-D in Unix). To read in a character vector we specify the vector type by the second argument:

```
> diet <- scan(,"")
D E A C B F C D F B A E
F A C E D B B C E A F D
E F B D C A A B D F E C
```

To read from a *file* specify its name as the first argument, for example

```
> counts <- scan("chd.dat")
```

Now suppose that multiple data vectors of equal length are to be read in in parallel. For example suppose that there are three vectors, the first of mode character and the remaining two of mode numeric, and the file is `input.dat`. Use `scan(...)` to read in the three vectors as a list, as follows

```
> in <- scan("input.dat",list(id="", x=0, y=0))
```

The second argument is a dummy list structure that establishes the mode of the three vectors to be read. The result, held in `in`, is a list whose (named) components are the three vectors read in.

Matrices are usually read by row, as follows

```
> X <- matrix(scan("light.dat"), ncol=5, byrow=T)
```

The argument `skip=` to `scan` can be used to skip header rows of files.

Data frames can be read from a file by the `read.table` function. The data file should be a table in one of a number of formats:

1. A file such as `rotifer.dat` (page 39) which has a first row naming the columns, followed by the table of numeric data can be read by

   ```
   > rotifer <- read.table("rotifer.dat", header=T)
   ```

2. A file laid out like the listing of a data frame. This has a first header line, and rows which contain the row label followed by the data for the columns, such as

```
        deaths  drivers popden  rural   temp    fuel
Alabama 968      158     64      66      62      119
Alaska   43      11      0.4     5.9     30      6.2
.............
```

   Note that the header has one less entry than subsequent rows. This format is read by

```
> road <- read.table("road.dat")
```

3. A table without any header. The row and column labels are then `1, ... ,` m and `V1`, ...`V`n. However, if there exists a character column without duplicates, the first such is taken as the row labels and removed as a column.

Sometimes it is necessary to read in character strings which contain spaces. This can be done by separating the fields in the file by, for example, tabs or commas:

```
> usroad <- scan("road.dat", sep="\t", list(state="", deaths=0,
+ drivers=0, popden=0, rural=0, jantemp=0, fuel=0))
```

where `\t` is the usual Unix abbreviation for a tab character. This device also applies to `read.table`.

5.1 Writing out data

There are amny ways to write out data from S, for example the `print`, `cat` and `format` commands. To write directly to a file, there are `cat`, `write` and, from S-Plus 3.2, `write.table` which is usually the simplest method. This can write a dataframe, matrix or vector, with syntax

```
> write.table(data, file="", sep=",")
```

and further arguments can be found in the help page. By default it writes out comma-separated items on rows, but the separator can be changed to space or tab (`"\t"` in Unix).

The function `write` writes a vector, with syntax

```
> write(data, file="data", ncolumns=5)
```

for numeric data, and in one column for character data. To write out a matrix `m`, use

```
> write(t(m), file="data", ncolumns=ncol(m))
```

The function `format` converts data to a line of characters, and can be used with `write` or `cat` to construct custom reports.

6 Graphics

The graphical facilities are central to S. The steps involved are as follows:

1. The type of terminal, or *device*, is declared to S at the beginning of the session:

   ```
   > openlook()
   ```

2. A command is issued to construct a plot from data. For example

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

   specifies a simple point plot where `x` and `y` are vectors giving the x- and y-coordinates of the points respectively. (The command includes a default automatic choice of axes, scales, titles and plotting characters, all of which can be overridden with additional *graphical parameters* that could be included as named arguments in the command.)

6.1 Graphical Parameters

Functions producing graphical output usually have optional additional named arguments that can be specified to override some default parameter settings and hence modify the characteristics of a plot. A short list of the main ones is as follows:

`axes=L`	If `FALSE` all axes are suppressed. Default `TRUE`, axes are automatically constructed.
`type="c"`	Type of plot desired. Values for `c` are: `p` for points only, (the default for function `plot`), `l` for lines only, `b` for both points and lines, (the lines miss the points), `s, S` for step functions (`s` specifies to change now, `S` to change just before the next point), `o` for overlaid points and lines, `h` for high density vertical line plotting, and `n` for no plotting (but axes are still found and set).
`xlab="string"` `ylab="string"`	Give labels for the `x`– and/or `y`–axes (default: the names, including suffices, of the `x` and `y` coordinate vectors).
`sub="string"` `main="string"`	`sub` specifies a title to appear under the `x`–axis label and `main` a title for the top of the plot in larger letters. (default: both empty).
`xlim=c(lo ,hi)` `ylim=c(lo, hi)`	Approximate minimum and maximum values for `x`– and/or `y`–axes settings. These values are automatically rounded to make them "pretty" for axis labelling.

Other graphical parameters control the background characteristics of all subsequent plots and are usually specified by a call to the function `par(...)`. There are a great number of these parameters and the command

```
> help(par)
```

gives a complete list of them and their meanings. Some of the more commonly adjusted ones are as follows:

`lty=n` — Line type is `n`. If lines are being plotted, a variety of line types is available; `n = 1` means a solid line, `n = 2, 3, ...` indicates a variety of broken line forms.

`pch="c"` — Specify the character to be used for plotting points (default: `*` for graphics terminals, • for PostScript).

`mfrow=c(m,n)`
`mfcol=c(m,n)` — multiple frames on the one plot. Instead of plotting just one graph per screen, each screen (or page) will contain an array of `m*n` graphs forming an $m \times n$ grid. If `mfrow` is used the screen is filled row-by-row and if `mfcol` is used it is filled column-by-column. Useful if many graphs are to be inspected simultaneously and high resolution is not necessary.

`pty="c"` — Specify the type of plotting region currently in effect. Possible values for `c` are
`s` to generate a square plotting region;
`m` (the default) to generate a maximal size plotting region.

6.2 Some Basic Plotting Functions

The elementary plotting functions are as follows:

`plot(x,y,...)` — Scatter plot of points with `x`– and `y`–coordinates given by the two main parameters. The pair `x,y` may be replaced by a single list with components labeled `x` and `y`, called a 'plot list'.
Graphical parameters are particularly useful.

`points(x,y,...)` — Add points to an existing plot (possibly using a different plotting character. Follows on from a `plot(...)` command.

`lines(x,y,...)` — Add lines to an existing plot. Similar to points.
Note

```
> plot(x,y); lines(spline(x,y))
```

will join the points of a plot by a cubic spline interpolation function. (See `help(spline)` for further information.)

`text(x,y, labels,...)` — Add text to a plot at points given by `x,y`. Normally `labels` is an integer or character vector in which case `labels[i]` is plotted at point `(x[i],y[i])`. The default is `1:length(x)`.
Note: This function is often used in the sequence

```
> plot(x,y,type="n"); text(x,y)
```

The graphics parameter `type="n"` suppresses the plotting of points but set up the axes, and the `text(...)` function supplies special characters (in this case just the integers by default) for the points.

`abline(a,b,...)`
`abline(h=c,...)`
`abline(v=c,...)`
`abline(`*lmobject*`,...)` — Draw a line in intercept and slope form, (`a`,`b`), across an existing plot. `h=c` may be used to specify `y`–coordinates for the heights of horizontal lines to go across a plot, and `v=c` similarly for the `x`–coordinates for vertical lines.

6.3 Interaction with Plots

S-Plus allows users to interact with plots, by identifying points and by adding information at places selected by mouse clicks.

`identify(x,y,labels)`	On a current plot of `x,y`, clicking the LEFT mouse button places the appropriate string from `label` near the point which has been clicked on. Click the MIDDLE mouse button to finish. If `label` is omitted uses index numbers, and always returns the indices of selected points.
`locator()`	Returns a list of vector coordinates of points clicked by the LEFT mouse button. Click the MIDDLE mouse button to finish.
`locator(,"p")`	ditto, but plots the points as in `plot`.
`legend(locator(),...)`	Add a legend box at a mouse-selected point (one LEFT click). See help page for the box contents and other options.

`locator()` is often used with `text` to add annotation to plots, e.g.

```
> text(locator(),"controls"); text(locator(),"cases")
```

6.4 Brush and Spin

These are S-Plus enhancements to allow dynamic manipulation of graphs. Spin allows three columns chosen from a matrix of data vectors to be rotated in space.

```
> help("state")
```

```
> spin(state.x77)
```

Use the left mouse button to select three of the variables, then use the cross-shaped pad to rotate the point cloud. Finally click on `quit`.

```
> brush(state.x77, hist=T)
```

includes `spin` and a `pairs` plot. Additionally one can ‘brush’ by selecting points with the left mouse button, and de-selecting them with the middle button. One can mark points in different ways, with the four symbols, and even label points if `label` is selected.

```
> brush(rbind(iris[,,1],iris[,,2],iris[,,3]))
```

Now select the first 50 points with one symbol and the last fifty with another. The intermediate nature of the middle 50 then stands out.

6.5 Equally-scaled plots

It is sometime necessary to make geometrically-square plots, for example so that distances can be assessed accurately. This is somewhat tricky, but done by the functions `eqscplot` in `library(ripley)`, which adjusts the axis scales to be equal within the current window shape.

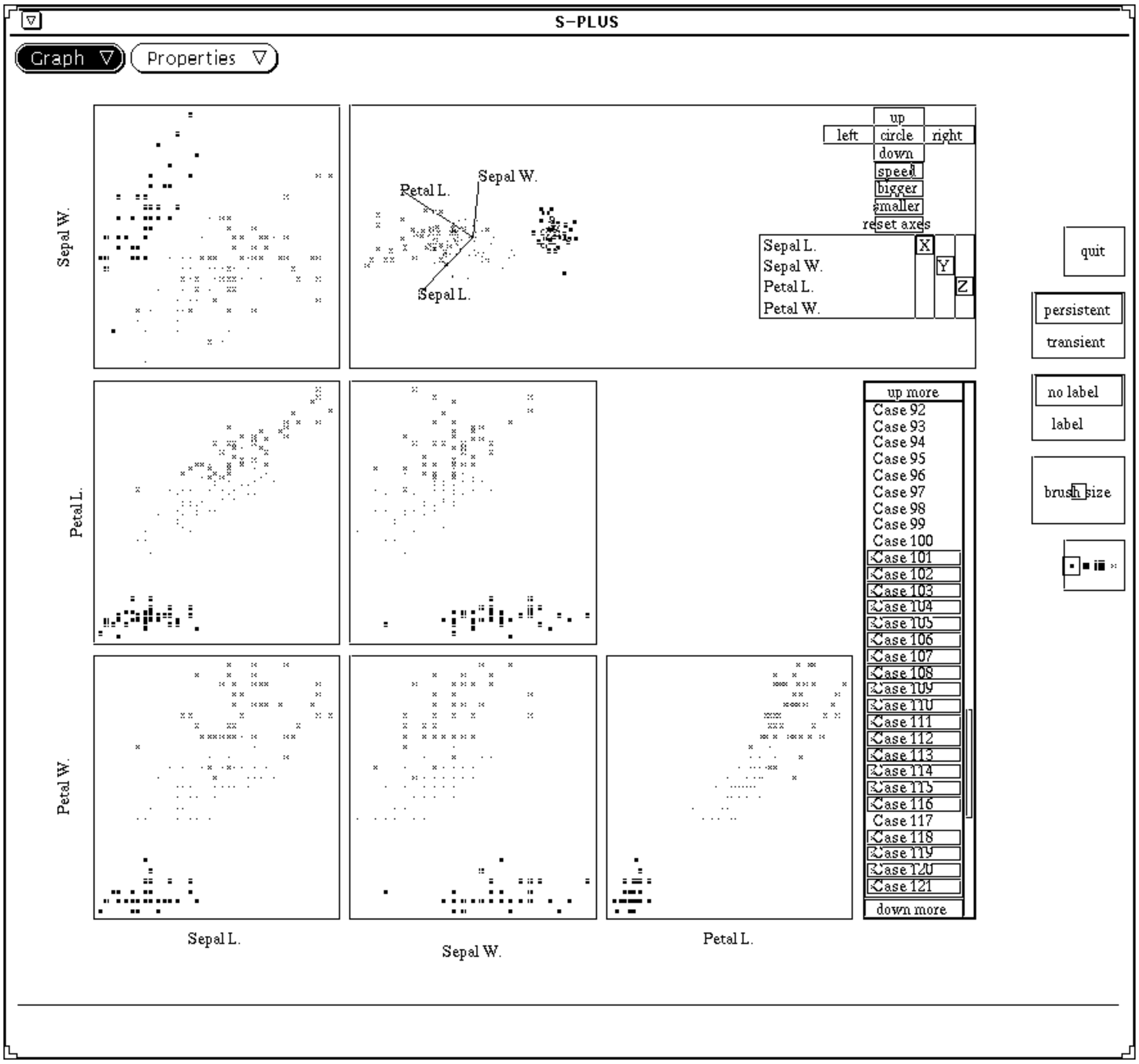

Figure 1: Screen dump of an `openlook()` window displaying `brush` on the `iris` data, with different highlights for the three groups.

7 Statistical Summaries

7.1 Arithmetical Summaries

Standard summaries such as `mean`, `median` and `var` are available. The `var` function will take a data *matrix* and give the variance-covariance matrix, and `cor` computes the correlation matrix, either from two vectors or a data matrix.

There are also standard functions `max`, `min`, `range` and `quantile`. The functions `mean` and `cor` will compute trimmed summaries. More sophisticated robust summaries are available, such as `location.m` and `scale.tau` as well as via the `robust` library.

7.2 Histograms and Stem-and-Leaf Plots

The standard histogram function is `hist(x, ...)` which plots a conventional histogram. More control is available via the extra parameters. The parameter `probability=T` gives a plot of unit area rather than cell counts, and `nclass` sets the number of bins.

Densities can be estimated via the function `density`:

```
hist(hstart, nclass=20, probability=T, ylim=c(0,0.02))
lines(density(hstart))
lines(density(hstart, width=20), lty = 3)
```

See figure 2.

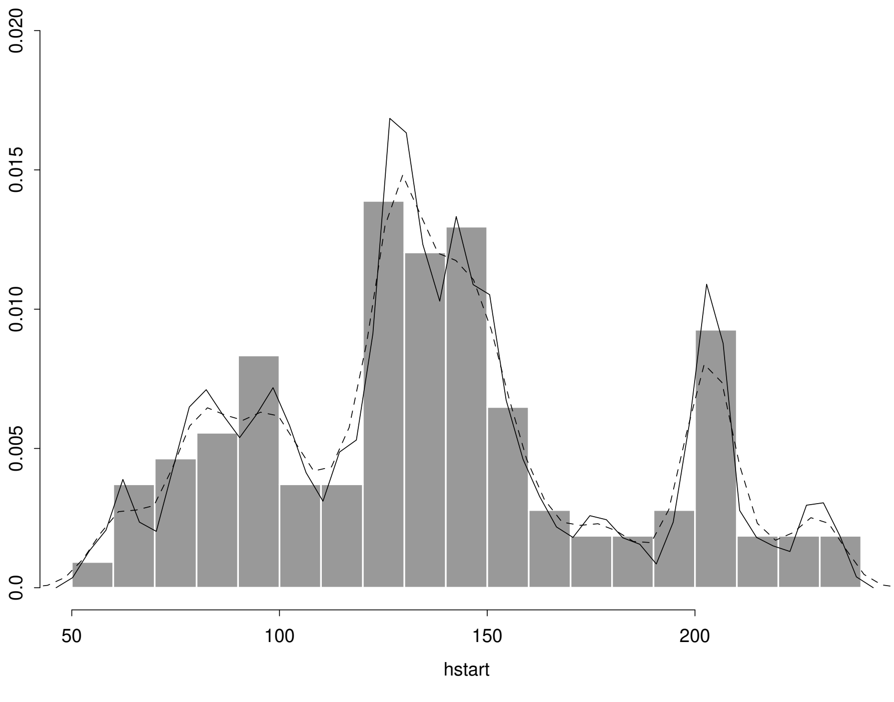

Figure 2: A histogram of `hstart` with two density estimates overlaid.

A *stem-and-leaf* plot is an enhanced histogram:

```
> stem(hstart)

N = 108   Median = 133.85
Quartiles = 105.2, 158.8

Decimal point is 1 place to the right of the colon

    5 : 5
    6 : 2239
    7 : 55799
    8 : 233567
    9 : 1235779
   10 : 00456
   11 : 04568
   12 : 023466667777899
   13 : 0112344456799
   14 : 1222333447999
   15 : 0013458
   16 : 0159
   17 : 66
   18 : 27
   19 : 77
   20 : 01333445667
   21 : 38
   22 : 68
   23 : 14
```

Apart from giving a visual picture of the data, this gives more detail. The actual data, in sorted order, is roughly `55, 62, 62, 63, 69, ...` and this can be read off the plot. Sometimes the pattern of numbers (all odd?) gives clues. Quantiles can be computed (roughly) from the plot.

7.3 Boxplots

A *boxplot* is a way to look at the overall shape of a set of data. The central box shows the data between the quartiles, with the median represented by a line. ‘Whiskers’ go out to the extremes of the data, and very extreme points are shown by themselves. It is also possible to plot boxplot for groups side-by-side:

```
> library(ripley)
> boxplot(split(nottem, cycle(nottem)), names=month.abb)
```

divides a time-series into months, and plots the boxplots for each month on one plot. See figure 3. Other styles of boxplot are available—see the help page.

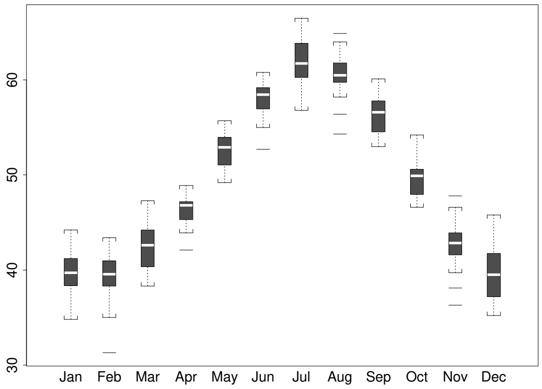

Figure 3: Boxplots for months of `nottem` data.

8 Distributions

S has functions built it to (approximate) the density, cumulative distribution function and quantile function (the inverse of the CDF) for many standard distributions. There are also function to simulate samples from these distributions. The first letter of the name indicates the function, e.g. `dnorm, pnorm, qnorm, rnorm` respectively.

Distributions available are:

Distribution	S name	parameters
beta	`beta`	`shape1, shape2`
binomial	`binom`	`size, prob`
Cauchy	`cauchy`	`location, scale`
chisquare	`chisq`	`df`
exponential	`exp`	`rate`
F	`f`	`df1,df2`
gamma	`gamma`	`shape`
geometric	`geom`	`prob`
hypergeometric	`hyper`	`m, n, k`
log-normal	`lnorm`	`meanlog, sdlog`
logistic	`logis`	`loc, scale`
negative binomial	`nbinom`	`size, prob`
normal	`norm`	`mean, sd`
normal range	`nrange`	`size, sd`
Poisson	`pois`	`lambda`
stable	`stab`	`index, skewness`
T	`t`	`df`
uniform	`unif`	`min, max`
Weibull	`weibull`	`shape, scale`
Wilcoxon	`wilcox`	`m, n`

The function `sample` re-samples from a data vector, with or without replacement.

8.1 Q-Q Plots

One of the best ways to compare the distribution of a sample `x` with a distribution is to use a Q-Q plot, of which the normal probability plot is the best-known example. Q-Q plots can also be used to compare two samples. For a sample `x` the quantile function is the inverse of the empirical CDF, that is

$$\text{quantile}(p) = \min(z \mid \text{proportion } p \text{ of the data } \leq z)$$

The function `qqplot(x, y, ...)` plots the quantile functions of two samples `x` and `y` against each other, and so compares two samples. The function `qqnorm(x)` replaces one of the samples by a sample at the quantiles of a standard normal distribution. This idea can be applied quite generally. For example, to test a sample against a t_9 distribution, we use

```
plot( qt(ppoints(x),9), sort(x) )
```

where `ppoints` computes the appropriate set of probabilities for the plot.

The function `qqline` helps assess how straight a `qqnorm` plot is by plotting a straight line through the upper and lower quartiles. (See the example in §3.)

9 Classical Statistics

S-Plus 3.1 has a section on classical statistics. The same functions are used to perform tests and to calculate confidence intervals.

The table shows the amount of wear in a shoe experiment with 10 boys, an experiment reported in Box, Hunter & Hunter (1977), *Statistics for Experimenters*. There were two materials (A and B) that were randomly assigned to the left or right shoe.

boy	A	B
1	13.2 (L)	14.0 (R)
2	8.2 (L)	8.8 (R)
3	10.9 (R)	11.2 (L)
4	14.3 (L)	14.2 (R)
5	10.7 (R)	11.8 (L)
6	6.6 (L)	6.4 (R)
7	9.5 (L)	9.8 (R)
8	10.8 (L)	11.3 (R)
9	8.8 (R)	9.3 (L)
10	13.3 (L)	13.6 (R)

We can use these data to illustrate one-sample and paired and unpaired two-sample tests. The rather voluminous output has been edited:

```
> shoes <- scan(,list(A=0, B=0))
1: 13.2 14.0
3: 8.2 8.8
5: 11.2 10.9
7: 14.3 14.2
9: 10.7 11.8
11: 6.6 6.4
13: 9.5 9.8
15: 10.8 11.3
17: 9.3 8.8
19: 13.3 13.6
21:
> attach(shoes)
> t.test(A, mu=10)

         One-sample t-Test

data:  A
t = 0.8127, df = 9, p-value = 0.4373
alternative hypothesis: true mean is not equal to 10
95 percent confidence interval:
  8.876427 12.383573
sample estimates:
 mean of x
```

```
     10.63

> t.test(A)$conf.int
[1]  8.876427 12.383573
attr(, "conf.level"):
[1] 0.95
> wilcox.test(A, mu=10)

         Exact Wilcoxon signed-rank test

data:  A
signed-rank statistic V = 34, n = 10, p-value = 0.5566
alternative hypothesis: true mu is not equal to 10

> t.test(A, B)

         Standard Two-Sample t-Test

data:  A and B
t = -0.3689, df = 18, p-value = 0.7165
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -2.744924  1.924924
sample estimates:
 mean of x mean of y
     10.63     11.04

t.test(A, B, var.equal=F)
>
         Welch Modified Two-Sample t-Test

data:  A and B
t = -0.3689, df = 17.987, p-value = 0.7165
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -2.745046  1.925046
sample estimates:
 mean of x mean of y
     10.63     11.04

> t.test(A, B, paired=T)

         Paired t-Test

data:  A and B
t = -3.3489, df = 9, p-value = 0.0085
alternative hypothesis: true mean of differences is not equal to 0
95 percent confidence interval:
 -0.6869539 -0.1330461
   ...
```

```
> wilcox.test(A, B, paired=T)

        Wilcoxon signed-rank test

data:  A and B
signed-rank normal statistic with correction Z = -2.4495, p-value = 0.0143
```

The sample size is rather small, and one might wonder about the validity of the t-distribution. An alternative for a randomized experiment such as this is to base inference on the permutation distribution of `d`. Figure 4 shows that the agreement is very good. (As the computation of this figure uses some subtle ideas in S, it is omitted: see Venables & Ripley (1994, Chapter 5).)

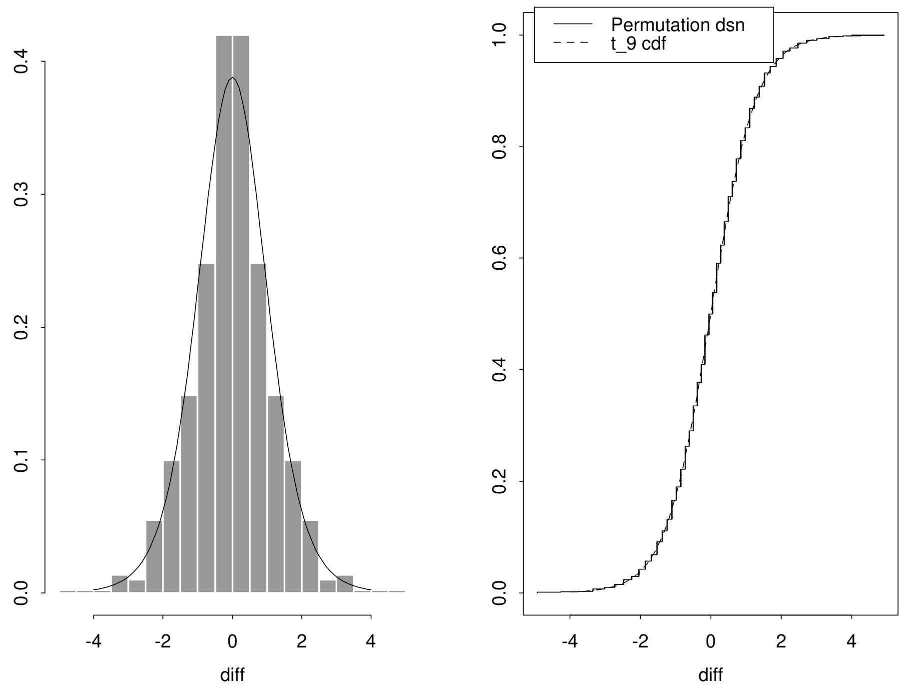

Figure 4: Histogram and empirical CDF of the permutation distribution of the t-test in the shoes example. The density and CDF of t_9 are shown overlaid.

The list of classical tests is:

`binom.test`	`chisq.test`	`cor.test`	`fisher.test`
`friedman.test`	`kruskal.test`	`mantelhaen.test`	`mcnemar.test`
`prop.test`	`t.test`	`var.test`	`wilcox.test`

Many of these have alternative methods – for `cor.test` there are methods `"pearson"`, `"kendall"` and `"spearman"`.

10 Handling Categorical Data

Consider a (fictitious) survey of shoppers in Britain. Amongst the variables collected for each person surveyed are sex, age, TV area[3], social class[4], transport used for this trip to the shops, and total spend at supermarkets. The possible values of these variables are

sex:	M, F
age:	–24, 25–44, 45–59, 60+
TV area:	1, . . . , 12
social:	A, B, C1, C2
transport:	car, bus, cycle, foot
spend:	*positive continuous*

This provides examples of each of S's types of categorical data structure. There are two main structures, *categories* and *factors*. The latter were introduced in the August 1991 release, and have almost entirely superseded the use of categories. A factor is regarded as a vector over the *set* of levels which have no implied order. Thus sex, TV area and transport are all factors. However, TV area is coded by number rather than by the names of the companies. These variables can be declared as

```
sex <- factor(sex.data)
TV.area <- factor(TV.data)
transport <- factor(transport.data)
```

Internally in S levels are numbered in alphabetical order, and when factors are used as treatments in designed experiments, the order of levels may matter. For example, if we want to contrast females with males (rather than *vice versa*) we need to specify the *levels* of the factor explicitly:

```
> sex <- factor(sex.data, levels=c("M","F"))
```

Social class is an *ordered factor* in that the classes are perceived as ordered, with "A" (professionals) regarded as highest. We can declare an order by

```
social <- ordered(factor(social.data))
levels(social) <- levels(social)[4:1]
age <- ordered(factor(age.data),
          levels=c("-24", "25-44", "45-59", "60+"))
```

The first line orders the levels by the default (alphabetical) order. The second shows how the set of levels may be changed, in this case by reversing the existing ordering. Age is an ordered category for which it is necessary to specify the levels explicitly. Had `age.data` been specified as a continuous variable, it could have been categorized using `cut` (whose help page gives other ways to produce the categories):

```
age.cdata <- cut(age.data, c(0, 25, 45, 60, 99))
age <- ordered(factor(age.cdata),
          levels=c("-24", "25-44", "45-59", "60+"))
```

[3]Britain is covered by 12 commercial TV companies, so this provides a simple geographical variable.
[4]Derived from occupation.

Some of the functions for statistical models treat ordered factors in appropriate special ways.

10.1 The Function `tapply(...)` and Ragged Arrays

To continue the previous example, suppose we have want to summarize spend by some of the factors To calculate the sample mean income for each age-group we can now use the special function `tapply(...)`:

```
> spend.means <- tapply(spend, age, mean)
```

giving a means vector with the components labeled by the levels

```
> spend.means
 -24   25-44   45-59  60+
27.20  35.53   33.42  17.65
```

Suppose further we needed to calculate the standard errors of the mean spends. To do this we need to write an S function to calculate the standard error for any given vector. We discuss functions more fully in §12, but since there is an inbuilt function `var(...)` to calculate the sample variance, such a function is a very simple one-liner, specified by the assignment:

```
> stderr <- function(x) sqrt(var(x)/length(x))
```

After this assignment, the standard errors are calculated by

```
> spend.stderr <- tapply(spend, age, stderr)
```

and the values calculated are then

```
spend.stderr
 -24   25-44   45-59  60+
3.70   2.33    4.55   2.70
```

The function `tapply(...)` can be used to handle more complicated indexing of a vector by multiple factors. For example, we might wish to split the spend by both age and sex:

```
> tapply(spend, list(age, sex), mean)
```

The combination of a vector and a labelling factor is an example of what is called a *ragged array*, since the subclass sizes are possibly irregular. When the subclass sizes are all the same the indexing may be done implicitly and much more efficiently by using arrays. The function `apply` is the analogue of `tapply` for arrays.

The pattern of our survey can be seen by the `table` function, which takes a listing of factors and returns the contingency table as an array, e.g.

```
> table(sex, age, TV.area, social, transport)
```

11 Loops and Conditional Execution

Commands may be grouped together in braces, `{expr`$_1$`; expr`$_2$`;...; expr`$_m$`}`. The value of the group is the result of the last expression in the group evaluated. Since such a group is also an expression it may, for example, be itself included in parentheses and used as part of an even larger expression, and so on. This facility is most often used with the control statements of this section.

The control statements are very close in spirit to those of the C programming language, and only a few are mentioned here. There is a conditional construction of the form

> `if (`*expr*$_1$`)` *expr*$_2$ `else` *expr*$_3$

where *expr*$_1$ must evaluate to a logical value and the result of the entire expression is then evident.

There is also a `for`–loop construction which has the form

> `for (`*name* `in` *expr*$_1$`)` *expr*$_2$

where *name* is a dummy, `expr`$_1$ is a vector expression (often a sequence like `1:20`), and `expr`$_2$ is often a grouped expression with its sub-expressions written in terms of the dummy *name*. `expr`$_2$ is repeatedly evaluated as *name* ranges through the values in the vector result of `expr`$_1$.

As an example, suppose `ind` is a vector of class indicators and we wish to produce separate plots of `y` versus `x` within classes. Use the `help` facility to understand the following:

```
> yc <- split(y,ind);  xc <- split(x, ind)
> for (i in 1:length(yc)){plot(xc[[i]],yc[[i]]);
+    abline(lsfit(xc[[i]],yc[[i]]))}
```

(Note the function `split(...)` which produces a list of vectors got by splitting a larger vector according to the classes specified by a factor.)

Other looping facilities include the

> `repeat` *expr*

statement and the

> `while (`*condition*`)` *expr*

statement. The `break` statement can be used to terminate any loop abnormally, and `next` can be used to discontinue one particular cycle.

Loops in S are often memory-hungry, and care may be needed not to use up all of your computer's memory. Expert advice is necessary on work-arounds.

12 Writing Your Own Functions

As we have seen informally in §10.1, the S language allows the user to create his or her own functions. These are true S functions that are stored in a special internal form and may be used in further expressions and so on. In the process the language gains enormously in power, convenience and elegance. Most of the functions supplied as part of the S system, such as `mean(...)` and `var(...)` and so on, are themselves written in S and thus do not differ materially from user written functions. (However, increasingly such functions are being re-written as internal functions to gain efficiency.) Listing these functions (by printing their name *without* parentheses) is a very fruitful way to gain hints for writing your own functions.

A function is defined by an assignment of the form

`> name <- function(`arg_1`, `arg_2`, ...)` *expression*

The *expression* is an S expression, (usually a grouped expression), that uses the arguments, arg_i, to calculate a value. The value of the expression is the value returned for the function. A call to the function then takes the form `name(`$expr_1$`, `$expr_2$`,...)` and may occur anywhere a function call is legitimate.

For example, the `IQR` function in `library(robust)` is defined as:

```
IQR <- function(y)
{
        r <- quantile(y, c(.25, .75))
        r[2] - r[1]
}
```

This first computes the quartiles, then returns the last value computed, their difference.

Note that *any ordinary assignments done within the function are temporary and lost after exit from the function.* Thus `r` is not left behind, and does not affect any other object `r`.

If global and permanent assignments are intended within a function, then the 'superassignment' operator, '`<<-`' can be used. See the `help` documentation for details, and see also the `synchronize()` function.

As a second example of a useful function, consider a function to evaluate the 'Huber proposal 2' robust estimator(s) of location and/or scale:

```
hubers <- function(y, k = 1.5, mu, s, initmu = median(y), tol = 1.0e-6)
{
    y <- y[!is.na(y)]
    n <- length(y)
    if(missing(mu)) {
        mu0 <- initmu
        n1 <- n-1
    } else {
        mu0 <- mu
        mu1 <- mu
        n1 <- n
    }
    if(missing(s)) {
```

```
        s0 <- mad(y)
    } else {
        s0 <- s
        s1 <- s
    }
    th <- 2 * pnorm(k) - 1
    beta <- th + k^2 * (1 - th) - 2 * k * dnorm(k)
    repeat {
        yy <- pmin(pmax(mu0 - k * s0, y), mu0 + k * s0)
        if(missing(mu))  mu1 <- sum(yy)/n
        if(missing(s)) {
            ss <- sum((yy - mu1)^2)/n1
            s1 <- sqrt(ss/beta)
        }
        if((abs(mu0 - mu1) < tol * s0) && abs(s0 - s1) < tol * s0)
            break
        mu0 <- mu1
        s0 <- s1
    }
    list(mu = mu0, s = s0)
}
```

This allows either of the location `mu` and scale `s` to be specified. Optional arguments are the parameter `k`, the initial value for `mu` and a convergence tolerance. The first line removes all missing values. The `missing()` function checks if a parameter is supplied. Two constants are then calculated as functions of `k`. The rest of the function is a loop. In general loops are inefficient in S and should be avoided if at all possible, but here we have no choice as the calculation is iterative. Finally the function returns two components, the location and scale.

It is sometimes useful to be able to time commands:

```
cputime <- function(x) sum(unix.time(x)[-3])
elapsed <- function(x) unix.time(x)[3]
```

which return the total cpu time and the elapsed time taken by a command or sequence of commands enclosed in `{...}`. Note: as these are functions, assignments inside them are in the frame of the function rather than permanent. Alternatively, use `proc.time()` before and after a group of commands.

13 Statistical Models

These facilities form the heart of the 1991 version of S. They are based on object-oriented extensions, so that generic functions such as `print` know what to do with the results of various models. The two most basic notions are a *data frame* (§4.9) and a *model formula*.

13.1 Model Formulas

A model formula couples a y-vector with a model expressed in a terminology very similar to that of GLIM and GENSTAT. The form is

```
> loss ~ hardness + tens
```

for the linear regression of `loss` on `hardness` and `tens`. Factors are replaced by a set of indicator variables for the regression, and can interact via the : operator (not . as this is a valid character in a variable name). Thus we can have all the following constructs:

`> time ~ poison + treatment + poison:treatment`	equivalent to
`> time ~ poison * treatment`	
`> strength ~ yarns/bobbins`	nested layout
`> gain ~ group + initial`	parallel lines
`> conc ~ -1 + reading`	line thorough the origin
`> conc ~ poly(reading, 2)`	quadratic polynomial
`> conc ~ ns(reading, 4, intercept=T)`	natural spline
`> conc ~ s(reading)`	smooth function, for `gam`

The syntax of a linear-model fit is

`lm(`*model formula* , *data frame*`)`

where the names in the model formula refer to columns of the data frame, which can be omitted if it has already been attached. For example

```
> library(ripley)
> attach(rubber)
> tyres.lm <- lm(loss ~ hard + tens)
> summary(tyres.lm)
> anova(tyres.lm)
> coefficients(tyres.lm)
> plot(fitted(tyres.lm), resid(tyres.lm))
```

This show how to extract information from a fit by the use of ancillary functions. There are no standard ancillary functions for standardized and Studentized residuals, but I have added them as `stdres()` and `studres()` in `library(ripley)`.

13.2 One-way Layouts

The analysis of one-way layout is best illustrated by an example. The table gives data on observed concentrations (*ng/ml*) of a chemical in groups of 10 patients after oral administration of almitrine bismesylate:

	drug dose (mg)			
subject	25	50	100	200
1	34	92	256	229
2	46	150	271	232
3	50	81	270	288
4	49	155	120	195
5	21	85	333	354
6	52	95	198	288
7	30	95	109	288
8	29	82	140	170
9	27	110	147	522
10	51	99	196	296

```
> stdev <- function(x) sqrt(var(x))                Function to compute st. dev.
> chemical <- scan("chemical.dat")
> dose <- rep(c(25, 50, 100, 200), 10)             Label the observations by dose
> group <- factor(dose)                            Make a factor from the doses
> boxplot(split(chemical, dose))
> tapply(chemical, dose, mean)
> tapply(chemical, dose, stdev)
> chems <- data.frame(group, chemical)             set up for AOV
> chems.aov <- aov(chemical ~ group, chems)
> summary(chems.aov)                               print out table
> coefficients(chems.aov)                          and the parameters
> chems.aov <- aov(log(chemical) ~ group, chems)  and on log scale
> summary(chems.aov)
> summary(aov(log(chemical) ~ log(dose)+group, chems))
                                                   test for linearity of response
```

which gives

```
> summary(chems.aov)
          Df Sum of Sq  Mean Sq  F Value        Pr(F)
group      3  356084.5 118694.8 28.91804 1.069219e-09
Residuals 36  147762.9   4104.5
> coefficients(chems.aov)
 (Intercept) group1   group2   group3
     158.375  32.75 44.11667 42.60833
> summary(chems.aov)
          Df Sum of Sq  Mean Sq F Value        Pr(F)
```

```
group      3  22.93801 7.646003 74.7226 1.554312e-15
Residuals 36   3.68371 0.102325
> summary(aov(log(chemical) ~ log(dose)+group, chems))
          Df Sum of Sq  Mean Sq  F Value      Pr(F)
log(dose)  1  21.87397 21.87397 213.7692 0.00000000
group      2   1.06404  0.53202   5.1993 0.01038375
Residuals 36   3.68371  0.10233
```

The parameterization of linear models for designed experiments is a little tricky. The usual parameterization is to impose a 'sum to zero' constraint on the parameters for a factor. GLIM sets the parameter for the first level to zero, so that parameters for the the other levels are differences between that level and the first. By default S uses the Helmert parameterization, which compares the second and subsequent levels to the *average* of lower levels. The usual parameterization can be gotten as default by setting

```
> options(contrasts=c("contr.sum", "contr.poly"))
```

and the GLIM parameterization by

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

Of course, the parameterization only affects the coefficients, not the fitted values, residuals, ... The contrasts for a particular term in a fit can be changed by the `C()` function, e.g. `C(group, sum)` or using `contrasts`.

There is a 'clever' way to test for linearity using a re-parameterization of the factor `group` as an *ordered* factor, for which the default parameterization is polynomial in {`1, ..., #(levels)`}. (This relies on `log(dose)` having levels in an arithmetic progression. One could always use `poly(log(dose),3)` in place of `ldose`.)

```
> ldose <- ordered(factor(log(dose)))
> summary.lm(aov(log(chemical) ~ ldose, chems))
```

(As far as I can see the use of `summary.lm` is necessary to get results for the individual coefficients.) This shows that the response can be regarded as quadratic in log(dose):

```
> summary.lm(aov(log(chemical) ~ ldose, chems))
Call: aov(formula = log(chemical) ~ ldose, data = chems)
Residuals:
     Min      1Q    Median     3Q    Max
 -0.5706 -0.2187 -0.001092 0.2806 0.6481

Coefficients:
               Value Std. Error  t value Pr(>|t|)
(Intercept)   4.7750   0.0506    94.4089   0.0000
    ldose.L   1.4790   0.1012    14.6208   0.0000
    ldose.Q  -0.3254   0.1012    -3.2168   0.0027
    ldose.C   0.0228   0.1012     0.2255   0.8228

Residual standard error: 0.3199 on 36 degrees of freedom
Multiple R-Squared: 0.8616
```

```
F-statistic: 74.72 on 3 and 36 degrees of freedom,
           the p-value is 1.554e-15

Correlation of Coefficients:
        (Intercept) ldose.L ldose.Q
ldose.L  0
ldose.Q  0           0
ldose.C  0           0       0
```

13.3 Designed Experiments

The central concept for designed experiments is a *factor*. Consider the famous Box-Cox poisons data (survival times (in hours) of animals with 3 poisons and 4 antidotes, from Box & Cox (1964), *J. Roy. Statist. Soc.* B26, 211–252 and Box, Hunter & Hunter (1977), *Statistics for Experimenters*). The function `fac.design` generates the rows, columns and so on – consult its help page for full details.

```
stimes <- scan("poison.dat")                                  data in hours
fnames <- list(treat=LETTERS[1:4], repl=1:4, poison=c("I","II","III"))
poisons<- data.frame(fac.design(c(4,4,3),fnames),stimes)
par(mfrow=c(3,2))
plot.design(poisons)                                          plot main effects
plot.design(poisons, fun=median)                              and using medians
attach(poisons)
plot.factor(stimes ~ treat + poison,data=poisons)             box plots
interaction.plot(treat, poison, stimes)
interaction.plot(treat, poison, stimes, fun=median)
poisons.aov <- aov(stimes ~ treat * poison)                   full fit
fits <- fitted(aov(stimes ~ treat + poison))                  additive fit for 1dofna
summary(poisons.aov)
par(mfrow=c(2,2))
hist(resid(poisons.aov))
qqnorm(resid(poisons.aov))
plot(fitted(poisons.aov),resid(poisons.aov))
summary(aov(stimes ~ treat + poison + fits^2 + treat:poison))
```

which gives

```
> summary(poisons.aov)
             Df Sum of Sq  Mean Sq  F Value     Pr(F)
treat         3   92.1206 30.70688 13.80558 0.0000038
poison        2  103.3012 51.65062 23.22174 0.0000003
treat:poison  6   25.0138  4.16896  1.87433 0.1122506
Residuals    36   80.0725  2.22424
```

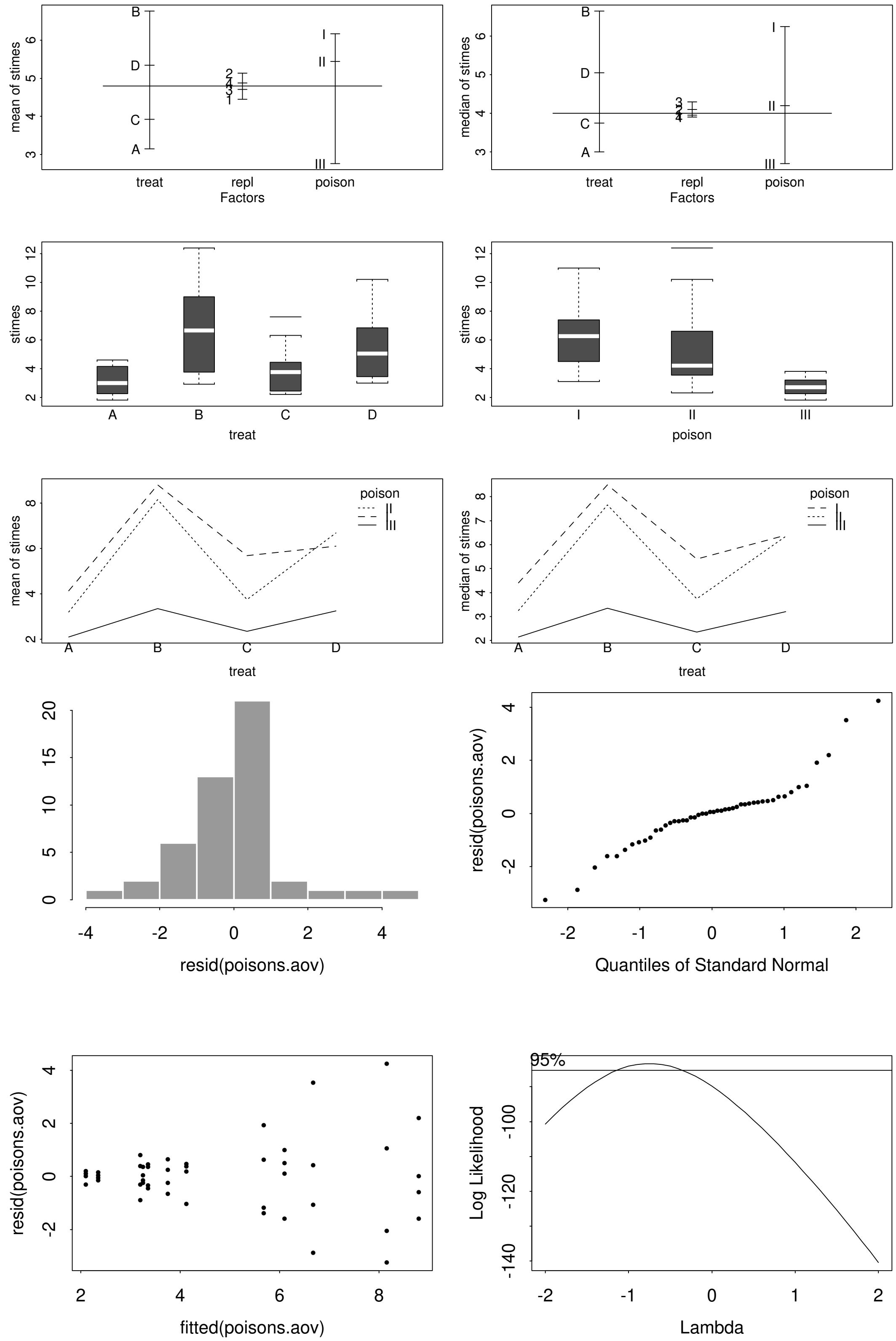

Figure 5: Plots for Poison data

```
> summary(aov(stimes ~ treat + poison + fits^2 + treat:poison))
             Df Sum of Sq  Mean Sq  F Value     Pr(F)
treat         3   92.1206 30.70688 13.80558 0.0000038
poison        2  103.3012 51.65062 23.22174 0.0000003
I(fits^2)     1   15.3724 15.37242  6.91132 0.0125158
treat:poison  5    9.6413  1.92827  0.86693 0.5127325
Residuals    36   80.0725  2.22424
```

indicating the need for transformation. The `I(...)` function protects the argument from expansion; `(treat+poison)^2` is equivalent to `treat+poison+treat:poison` and generally `(factors)^n` gives up to n-th order interactions.

There is no direct Box-Cox function, but we can do the operations by hand. They are quite slow (25 secs on a SparcStation IPC), due to the overhead of calling the `aov function`:

```
xl <- seq(-2,1,by=0.1)
loglik <- as.vector(xl)
n <- length(stimes)
nlngm <- log(prod(stimes))
for(i in 1:length(xl)){
    if(abs(xl[i]) > 0.01)
        {
        ss <- sum((aov(stimes^xl[i] ~ treat + poison)$resid)^2)
        loglik[i] <- n*log(abs(xl[i])) - n/2*log(ss) + (xl[i]-1)*nlngm
        }
    else
        {
        ss <- sum((aov(log(stimes) ~ treat + poison)$resid)^2)
        loglik[i] <- - n/2*log(ss) - nlngm
        }
    }
plot(xl, loglik, xlab = "Lambda", ylab = "Log Likelihood", type = "l")
lambdahat <- loglik[loglik == max(loglik)]
limit <- lambdahat - 0.5 * qchisq(0.95, 1)
abline(limit, 0)
scal <- (par("usr")[4] - par("usr")[3])/par("pin")[2]
text(c(xl[1]), limit + 0.1 * scal, " 95%")
```

A more efficient way (4 secs) is to use the function `BoxCox` in the library `ripley`:

```
> library(ripley)
> BoxCox(stimes ~ treat + poison)
```

Now consider a Latin square. Six litters of six piglets were ranked in order of birthweight, providing a 6×6 table, and each piglet given one of 6 dietary supplements in a Latin square. The weight gain (in *kg*) over 12 weeks is given in the table.

```
> diet <- scan(,"")
D E A C B F
C D F B A E
F A C E D B
B C E A F D
E F B D C A
A B D F E C

> wtgain <- scan()
5.74 7.21 5.92 6.58 7.24 4.83
5.92 5.74 7.88 6.53 6.73 6.05
8.18 5.87 5.41 7.49 7.44 6.43
3.58 5.21 5.61 4.39 7.63 5.44
6.05 8.16 6.27 5.84 6.71 5.77
4.95 6.35 5.56 7.50 7.04 6.22

> diet <- factor(diet)
> latin <- data.frame(fac.design(c(6,6), list(brank=1:6,litter=1:6)),
+    diet, wtgain)
> plot.design(latin)
> Diet <- C(diet, treatment)
> latin.aov <- aov(wtgain ~ brank + litter + Diet, latin)
> summary(latin.aov)
> summary.lm(latin.aov)
```

The last command gives t-values for the contrasts (diet ? – diet A).

```
> summary(latin.aov)
          Df Sum of Sq  Mean Sq  F Value      Pr(F)
brank      5   7.92405 1.584809 2.788687 0.04545633
litter     5   7.72041 1.544083 2.717023 0.04962477
Diet       5  11.61751 2.323503 4.088518 0.01015490
Residuals 20  11.36599 0.568299
> summary.lm(latin.aov)
Call: aov(formula = wtgain ~ brank + litter + Diet, data = latin)
Residuals:
    Min      1Q Median     3Q    Max
 -2.051 -0.2906 0.1211 0.3715 0.9061

Coefficients:
               Value Std. Error  t value Pr(>|t|)
(Intercept)   5.6050   0.3078    18.2122   0.0000
...............
      DietB   0.4617   0.4352     1.0607   0.3015
      DietC   0.4033   0.4352     0.9267   0.3651
      DietD   0.3550   0.4352     0.8156   0.4243
      DietE   0.9700   0.4352     2.2287   0.0375
      DietF   1.7583   0.4352     4.0399   0.0006
..............
```

13.4 Generalized Linear Models

The functions `lm` and `aov` have extensions `glm` which fits generalized linear models, and `gam` which further extends this to allow semi-parametric smooth functions in the explanatory variables. We can, for example, fit the poisons data by a gamma GLM:

```
attach(poisons)
poisons.glm <- glm(stimes ~ treat + poison, family=Gamma)
                                                    note the 'G'
summary(poisons.glm)
anova(poisons.glm)                                  analysis of deviance table
```

Once again there is a whole range of ancillary functions such as `deviance`, `predict` and `residuals`. The latter will produce a four types of residuals, but uses deviance residuals by default.

The `family` argument is also used to specify other aspects of the fit such as the link function. For example, one can have `family=binomial(link=probit))`. With the binomial the response can either be a factor (taken as first level *vs* the rest) or a matrix with two columns giving the number of successes and failures. There is a `quasi` family allowing user-defined models, and a `robust` family generator allowing robust fitting. The scope for ingenuity is unlimited!

Binary Data

The following example is taken from D. Collett (1991) *Modelling Binary Data*, page 217. Numbers of rotifers falling out of suspension for two species (*Polyartha major* and *Keratella cochlearis*) are given for different fluid densities in the table, as file `rotifer.dat`:

```
density pm.y pm.tot kc.y kc.tot
1.019  11  58  13 161
1.020   7  86  14 248
1.021  10  76  30 234
1.030  19  83  10 283
1.030   9  56  14 129
1.030  21  73  35 161
1.031  13  29  26 167
1.040  34  44  32 286
1.040  10  31  22 117
1.041  36  56  23 162
1.048  20  27   7  42
1.049  54  59  22  48
1.050  20  22   9  49
1.050   9  14  34 160
1.060  14  17  71  74
1.061  10  22  25  45
1.063  64  66  94 101
1.070  68  86  63  68
1.070 488 492 178 190
1.070  88  89 154 154
```

An annotated session follows. Several points need further explanation.

The parametrizations need careful consideration. By default S uses a linear-model parameterization, contrasting each level with the average of the previous levels. This is less useful for GLMs. The first way out below is to remove the overall mean (the `-1` term) which forces separate means for each species. We can also change to the GLIM parameterization by the `options` line.

There is a catch here. By default `factor()` numbers the factor levels in alphabetical order, so we have to force the order we want (see §10).

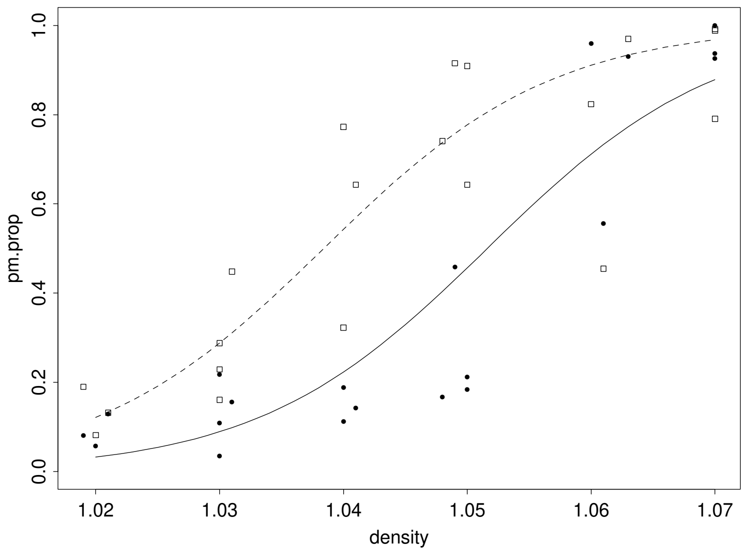

Figure 6: Plots for Rotifer data. The square symbols and dashed line indicate species *Polyartha major*.

```
rotifers <- read.table("rotifer.dat", header=T)
rotifers
```
list the data frame

```
attach(rotifers)
kc.prop <- kc.y/kc.tot
```
compute the proportions

```
pm.prop <- pm.y/pm.tot
```
and plot them

```
plot(density, pm.prop, type="n", ylim = c(0,1))
points(density, pm.prop, pch=0)
points(density, kc.prop)
```

fit separate models for each species

```
glm.pm <- glm(cbind(pm.y, pm.tot-pm.y) ~ density, binomial(logit))
glm.kc <- glm(cbind(kc.y, kc.tot-kc.y) ~ density, binomial(logit))
glm.pm; glm.kc
```
bare summaries

Now combine the two species

```
species <- factor(c(rep("pm",20), rep("kc",20)),
    levels= c("pm", "kc"))
rotifer2 <- data.frame(dens = c(density, density),
    yes = c(pm.y, kc.y), tot = c(pm.tot, kc.tot), species)
attach(rotifer2)
glm.rot <- glm(cbind(yes, tot-yes) ~ dens * species, binomial(logit))
```

```
glm.rot                                            Note the parameterization used
glm.rot <- glm(cbind(yes, tot-yes) ~ -1+dens * species, binomial(logit))
glm.rot                                            separate means for each species
options(contrasts=c("contr.treatment", "contr.poly"))
glm.rot <- glm(cbind(yes, tot-yes) ~ dens*species, binomial(logit))
glm.rot
summary(glm.rot)
anova(glm.rot)                                     over-dispersion, but a common slope
                                                   looks OK
glm.rot <- glm(cbind(yes, tot-yes) ~ dens + species, binomial(logit))
lines(density, fitted(glm.rot)[species=="kc"])
lines(density, fitted(glm.rot)[species=="pm"], lty=3)
                                                   these lines are rather crude, so try harder!
xden <- seq(1.02, 1.07, 0.001)
yden <- predict(glm.rot, data.frame(dens=rep(xden,2),
    species=factor(c(rep("pm", 51), rep("kc", 51)),
    levels= c("pm", "kc"))), type="response")
lines(xden, yden[1:51], lty=3)
lines(xden, yden[52:102], lty=3)
```

Poisson Data

We consider the log-linear analysis of a contingency table. As this has two ‘history’ factors and two levels of the the response, it could also be treated as binomial data. The response is the occurrence of coronary heart disease. The table is of the form:

		blood pressure			
chd	serum cholesterol	1	2	3	4
yes	1	2	3	3	4
	2	3	2	1	3
	3	8	11	6	6
	4	7	12	11	11
no	1	117	121	47	22
	2	85	98	43	20
	3	119	209	68	43
	4	67	99	46	33

with log-linear analysis:

```
num <- scan()
2 3 3 4 3 2 1 3 8 11 6 6 7 12 11 11
117 121 47 22 85 98 43 20 119 209 68 43 67 99 46 33

fnames <-list(press=1:4, serum=1:4, chd=c("y","n"))
kk <- data.frame(fac.design(c(4,4,2),fnames), num)
```

```
kk.glm <- glm(num ~ serum*press*chd, family=poisson, data=kk)
anova(kk.glm, test="Chi")
kk.glm1 <- update(kk.glm, . ~ .-serum:press:chd)
par(mfrow=c(1,2)); plot(kk.glm1)
```

The `anova` command gives an *analysis of deviance* for `glm` objects:

```
> anova(kk.glm, test="Chi")
Analysis of Deviance Table

Poisson model

Response: num

Terms added sequentially (first to last)
                Df Deviance Resid. Df Resid. Dev   Pr(Chi)
           NULL                    31   1644.227
          serum  3   77.370        28   1566.856 0.0000000
          press  3  318.287        25   1248.570 0.0000000
            chd  1 1169.609        24     78.960 0.0000000
    serum:press  9   24.449        15     54.511 0.0036454
      serum:chd  3   30.452        12     24.059 0.0000011
      press:chd  3   19.284         9      4.775 0.0002388
serum:press:chd  9    4.775         0      0.000 0.8534820
```

13.5 Updating and Selecting Models

There are number of facilities to update models. The `update` function takes a result of a previous fit and changes the model in some way.

`add1` and `drop1` show the (approximate) effects of adding and dropping single terms, and `step` runs a fairly general stepwise fitting procedure. (Note that S-Plus 3.x has a separate `stepwise` function for multiple regression.)

14 Multivariate Analysis

S-Plus is particularly rich is functions for exploratory multivariate analysis, such as `pairs`, `brush` and `spin`. There are also functions for classical multivariate analysis.

Clustering

The workhorses here are `dist` which computes distance matrices (also used in `cmdscale`) and `hclust` which computes a cluster tree by single-, average- or complete linkage.

`dist`	Distance matrix calculations
`hclust`	Hierarchical clustering
`cutree`	Create groups from a cluster tree
`plclust`	Plot a cluster tree
`labclust`	Label a cluster tree plot
`clorder`	Re-order leaves of a cluster tree
`subtree`	Extract part of a cluster tree
`mclust`	”model-based” clustering
`mclass`	auxiliary functions
`mreloc`	

Graphical Methods

This is a varied collection of functions for displaying multivariate data.

`cmdscale`	Classical multi-dimensional scaling
`faces`	Chernoff’s faces
`mstree`	Minimal spanning tree
`stars`	Star plots
`biplot`	Biplot (v 3.2)

Two analyses of socio-economic data on Swiss cantons:

```
library(ripley)
d <- dist(swiss.x)
x <- cmdscale(d)
c1 <- x[ ,1]; c2 <- x[ ,2]
eqscplot(c1, c2, type="n") # from library(ripley)
text(c1, c2, seq(c1))

h <- hclust(d)
plclust(h)
cutree(h, 3)
plclust(clorder(h, cutree(h, 3))) # re-order tree into three groups
```

Matrix Methods

The classical methods based on variance-covariance matrices.

`mahalanobis`	Mahalanobis distances
`cancor`	Canonical correlation analysis
`discr`	Discriminant analysis
`pcrcomp`	Principal components analysis
`princomp`	Principal components analysis (v 3.2)
`factanal`	Principal components analysis (v 3.2)

An example of discriminant analysis with Fisher's iris data:

```
iris.var <- rbind(iris[,,1], iris[,,2], iris[,,3])
species <- rep(1:3,rep(50,3))
iris.dis <- discr(iris.var, 3)
iris.dv <- iris.var %*% iris.dis$vars # find discriminant variables
brush(cbind(iris.dv, species))
iris.x <- iris.dv[,1] ; iris.y <- iris.dv[,2]
iris.lab <- c(rep("s", 50), rep("c", 50), rep("v", 50))
plot(iris.x, iris.y, type="n", xlab="first discriminant variable",
   ylab="second discriminant variable")
text(iris.x, iris.y, iris.lab, cex=0.7)
```

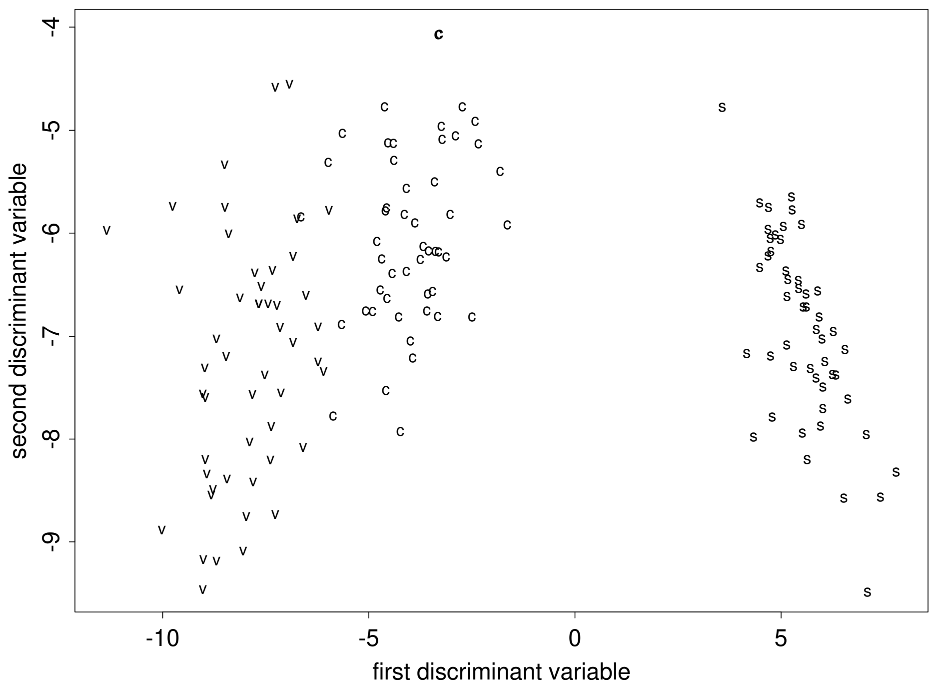

Figure 7: Discriminant analysis

A Libraries

Libraries are a mechanism to add 'packages' of extra objects (functions and datasets) to S. To find out which libraries are available type

```
> library()
```

which on one of my systems gave:

```
 The following sections are available in the library:

SECTION         BRIEF DESCRIPTION

chron           Functions to handle dates and times.
examples        Functions and objects from The New S Language.
external        Handle external (large) objects.
image           Display images.
maps            Display of maps with projections.
progdraw        Sdraw example from Programmer's Manual.
progexam        Examples from Programmer's Manual.
semantics       Functions from chapter 11 of The New S Language.

ripley          B.D. Ripley's teaching functions

For more information on each library section see the README file in
each library section directory or in S-PLUS run:
        library( help = <section_name> )

Library sections from Venables & Ripley (1994)
        'Modern Applied Statistics with S-Plus'

MASS            main library
nnet            neural networks
spatial         spatial statistics
```

To find out more about a section, use

```
> library(help=name)
```

e.g.

```
> library(help=robust)
functions for robust statistics

IQR(y)                  inter-quartile range
huber(y, k = 1.5)       Huber location with MAD scale
hubers(y, k = 1.5, mu, s)   Huber proposal 2 [with mu known, s known]
hreg(x, y, k = 1.5)     Huber robust regression

datasets

chem                    copper in wholemeal flour
abbey                   nickel in syenite rock
```

```
milk                    lead in milk powder
phones                  Belgian 'phone calls 1950-1973
```

To use the library, invoke it by

> library(*name*)

which attaches it as a data directory at the end of the search list. Thus libraries cannot over-ride standard functions nor your own functions. To make a library over-ride the system functions, use

> library(*name*, first=T)

which attaches it at position 2 (*after* the .Data directory).

A.1 Library ripley

This is a collection of useful functions and datasets for teaching at Oxford.

BoxCox(x,y)	Box-Cox plot for transformations.
gl(from,to,size,x)	replacement for GLIM %GL.
eqscplot	equally-scaled plot function.
stdres(object)	calculate standardized residuals from a fit.
studres(object)	calculate Studentized residuals from a fit.

Datasets in the library are:

accdeaths	US accidental deaths 1973-8
cement	dataset on heat evolved in setting cements
cpus	dataset on performance of cpus
deaths	time series on UK lung deaths 1974-9 from Diggle
mdeaths, fdeaths	as above, for males and females
gehan	remission times on leukaemia patients (censored)
forbes	Forbes' dataset on boiling points, from Atkinson
hills	dataset on times of Scottish hill races
leuk	(uncensored) survival times on leukaemia patients
lh	time series on luteinizing hormone from Diggle
mammals	body weight(kg) and brain weight (g) of mammals, from Weisberg
mcycle	motorcycle impact data – Silverman JRSS B 1985
motorette	accelerated life testing on motorettes
nottem	time-series of temperatures in Nottingham, 1920-1939
road	dataset on road deaths in the US
rock	dataset on relating permeability to physical measurements
rubber	dataset on rubber wear
ships	ship damage incidents, from McCullagh & Nelder
trees	Black Cherry trees heights, diameters and volumes

A.2 Sources of Libraries

Many S users have generously collected together their functions and datasets together into libraries and made them publically available. An archive of libraries is maintained at Carnegie-

Mellon as a service to the statistical profession by Mike Meyer. To obtain details of its contents by e-mail send a message to

```
statlib@lib.stat.cmu.edu
```

with body

```
send index
send index from S
```

Ftp to `lib.stat.cmu.edu` with user `statlib` is also available.