Notes on **S-PLUS**:

A Programming Environment for Data Analysis and Graphics

Bill Venables and Dave Smith
Department of Statistics
The University of Adelaide

Email: `venables@stats.adelaide.edu.au`

Version 2.2, September, 1992

Preface

These notes were originally intended only for local consumption at the University of Adelaide, South Australia. After some encouraging comments from students, the author decided to release them to a larger readership in the hope that in some small way they promote good data analysis. S (or S-PLUS) is no panacea, of course, but in my view it represents the single complete environment most conducive to good data analysis so far available.

The authors are indebted to many people for useful contributions, but in particular Lucien W. Van Elsen, who did the basic TeX to LaTeX conversion and Rick Becker who offered an authoritative and extended critique on an earlier version. Others to offer useful comments and corrections include Doug Bates, Ron Baxter, Ray Brownrigg, Sue Clancy, Kathy Haskard, Martin Maechler, James Pearce, Andreas Ruckstuhl and Catarina Savi.

Responsibility for this version, however, remains entirely with the authors, and the notes continue to enjoy a fully unofficial and unencumbered status.

Comments and corrections are always welcome. Please address email correspondence to the first author at `venables@stats.adelaide.edu.au`.

Suggestions to the reader

Most S-PLUS novices will start with the introductory session in Appendix A. This should give some familiarity with the style of S-PLUS sessions and more importantly some instant feedback on what actually happens.

Many users will come to S-PLUS mainly for its graphical facilities. In this case section 11 on the graphics facilities can be read at almost any time and need not wait until all the preceding sections have been digested.

Bill Venables and David Smith,
University of Adelaide,
6th September, 1992.

Contents

1 Introduction and Preliminaries

1.1 The S environment

S is an integrated suite of software facilities for data manipulation, calculation and graphical display. Among other things it has

- an effective data handling and storage facility,
- a suite of operators for calculations on arrays, in particular matrices,
- a large, coherent, integrated collection of intermediate tools for data analysis,
- graphical facilities for data analysis and display either at a workstation or on hardcopy, and
- a well developed, simple and effective programming language which includes conditionals, loops, user defined recursive functions and input and output facilities. (Indeed most of the system supplied functions are themselves written in the S language.)

The term “environment” is intended to characterize it as a fully planned and coherent system, rather than an incremental accretion of very specific and inflexible tools, as is frequently the case with other data analysis software.

S is very much a vehicle for newly developing methods of interactive data analysis. As such it is very dynamic, and new releases have not always been fully upwardly compatible with previous releases. Some users welcome the changes because of the bonus of new technology and new methods that come with new releases; others seem to be more worried by the fact that old code no longer works. Although S is intended as a programming language, in my view one should regard programs written in S as essentially ephemeral.

The name S as with many names within the UNIX world, is not explained, but left as a cryptic puzzle, and probably a weak pun. However its authors insist it does ***not*** stand for “Statistics”!

1.2 The S-PLUS environment

These notes will be mainly concerned with S-PLUS, an enhanced version of S distributed by Statistical Sciences, Inc., Seattle, Washington. Most of what we will have to say, however, applies interchangably to S and S-PLUS

1.3 Reference manuals

The basic reference is ***The New* S *Language: A Programming Environment for Data Analysis and Graphics*** by Richard A. Becker, John M. Chambers and Allan R. Wilks. The new features of the August 1991 release of S are covered in ***Statistical Models in*** S Edited by John M. Chambers and Trevor J. Hastie. In addition there are specifically S-PLUS reference books: S-PLUS ***User's Manual*** (Volumes 1 & 2) and S-PLUS ***Reference Manual*** (in two volumes, ***A–K*** and ***L–Z***).

It is not the intention of these notes to replace these manuals. Rather these notes are intended as a brief introduction to the S-PLUS programming language and a minor amplification of some important points. Ultimately the user of S-PLUS will need to consult this reference manual, probably frequently.

1.4 S-PLUS and X–windows

The most convenient way to use S-PLUS is at a high quality graphics workstation running a windowing system. Since these are becoming more readily available, these notes are aimed at users who have this facility. In particular we will occasionally refer to the use of S-PLUS on an X–window system, and even with the motif window manager, although the vast bulk of what is said applies generally to any implementation of the S-PLUS environment.

Setting up a workstation to take full advantage of the customizable features of S-PLUS is a straightforward if somewhat tedious procedure, and will not be considered further here. Users in difficulty should seek local expert help.

1.5 Using S-PLUS interactively

When you use the S-PLUS program it issues a prompt when it expects input commands. The default prompt is ">", which is sometimes the same as the shell prompt, and so it may appear that nothing is happening. However, as we shall see, it is easy to change to a different S-PLUS prompt if you wish. In these notes we will assume that the shell prompt is "`$ `".

In using S-PLUS the suggested procedure for the first occasion is as follows:

1. Create a separate sub-directory, say `work`, to hold data files on which you will use S-PLUS for this problem. This will be the working directory whenever you use S-PLUS for this particular problem.

   ```
   $ mkdir work
   $ cd work
   ```

2. Place any data files you wish to use with S-PLUS in `work`.
3. Create a sub-directory of `work` called `.Data` for use by S-PLUS.

   ```
   $ mkdir .Data
   ```

4. Start the S-PLUS program editor with the command

   ```
   $ Splus -e
   ```

   (The `-e` flag is optional, but it allows the inbuilt line editor to be used, which is very handy for correcting typing errors.)
5. At this point S-PLUS commands may be issued (see later).
6. To quit the S-PLUS program the command is

   ```
   > q()
   $
   ```

The procedure is simpler using S-PLUS after the first time:

Make `work` the working directory and start the program as before:

```
$ cd work
$ Splus -e
```

Use the S-PLUS program, terminating with the `q()` command at the end of the session.

1.6 An introductory session

Readers wishing to get a feel for S-PLUS at a workstation (or terminal) before proceeding are strongly advised to work through the model introductory session given in Appendix A, starting on page 65.

1.7 S-PLUS and UNIX

S-PLUS allows escape to the operating system at any time in the session. If a command, on a new line, begins with an exclamation mark then the rest of the line is interpreted as a UNIX command. So for example to look through a data file without leaving S-PLUS you could use

```
> !more curious.dat
```

When you finish paging the file the S-PLUS session is resumed.

In fact the integration of S-PLUS into UNIX is very complete. For example, there is a command, `unix(...)`, that executes any unix command, (specified as a character string argument), and passes on any output from the command as a character string to the program. Essentially the full power of the operating system remains easily available to the user of the S-PLUS program during any session.

The only non-UNIX implementation of S-PLUS so far is one for the DOS operating system. Users should consult the appropriate user guides for more information.

1.8 Getting help with functions and features

S-PLUS has an inbuilt help facility similar to the `man` facility of UNIX. To get more information on any specific named function, for example `solve` the command is

```
> help(solve)
```

An alternative is

```
> ?solve
```

For a feature specified by special characters, the argument must be enclosed in double or single quotes, making it a ‘character string’:

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

Either form of quote mark may be used to escape the other, as in the string `"It's important"`. Our convention in these notes is to use double quote marks for preference.

A much more comprehensive help facility is available with the X windows version of S-PLUS The command

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

causes a “help window” to appear (with the “motif” graphical user interface). It is at this point possible to select items interactively from a series of menus, and the selection process again causes other windows to appear with the help information. This may be either scanned at the screen and dismissed, or sent to a printer for hardcopy, or both.

1.9 S-PLUS commands. Case sensitivity.

Technically S-PLUS is a ***function language*** with a very simple syntax. It is ***case sensitive*** as are most UNIX based packages, 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 lost. An assignment also evaluates an expression and passes the value to a variable but the result is not automatically printed.

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-PLUS will give a different prompt, for example

```
+
```

on second and subsequent lines and continue to read input until the command is syntactically complete. This prompt may also be changed if the user wishes. In these notes we will generally omit the continuation prompt and indicate continuation by simple indenting.

1.10 Recall and correction of previous commands

1.10.1 S-PLUS

S-PLUS (but not plain S) provides a mechanism for recall and correction of previous commands. For interactive use this is a vital facility and greatly increases the productive output of most people. To invoke S-PLUS with the command recall facility enabled use the `-e` flag:

```
> Splus -e
```

Within the session, command recall is available using either emacs-style or vi-style commands. The former is very similar to command recall with an interactive shell such as tcsh. Details are given in Appendix B of these notes, or they may be found in the reference manual or the online help documents.

1.10.2 S

With S no built-in mechanism is available, but there are two common ways of obtaining command recall for interactive sessions.

- Run the S session under emacs using S–mode, a major mode designed to support S. This is probably more convenient than even the inbuilt editor of S-PLUS in the long term, however it does require a good deal of preliminary effort for persons not familiar with the emacs editor. It also often requires a dedicated workstation with a good deal of memory and other resources.
- Run the S session under some front end processor, such as the public domain fep program, available from the public sources archives. This provides essentially the same service as the inbuilt S-PLUS editor, but with somewhat more overhead, (but a great deal less overhead than emacs requires.)

1.11 Executing commands from, or diverting output to, a file

If commands are stored on an external file, say `commands.S` in the working directory `work`, they may be executed at any time in an S-PLUS session with the command

```
> source("commands.S")
```

Similarly

```
> sink("record.lis")
```

will divert all subsequent output from the terminal to an external file, `record.lis`. The command

```
> sink()
```

restores it to the terminal once again.

1.12 Data directories. Permanency. Removing objects.

The entities that S-PLUS creates and manipulates are known as ***objects***. These may be variables, arrays of numbers, character strings, functions, or more general structures built from such components.

All objects created during your S-PLUS sessions are stored as ***files***, in a special form, in the `.Data` sub-directory of your working directory `work`, say.

Each object is held as a separate file of the same name and so may be manipulated by the usual UNIX commands such as `rm`, `cp` and `mv`. This means that if you resume your S-PLUS session at a later time, objects created in previous sessions are still available, which is a highly convenient feature.

This also explains why it is recommended that you should use separate working directories for different jobs. Common names for objects are single letter names like `x`, `y` and so on, and if two problems share the same `.Data` sub-directory the objects will become mixed up and you may overwrite one with another.

There is, however, another method of partitioning variables within the same `.Data` directory using ***data frames***. These are discussed further in §6.4.

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

```
> objects()
```

whose result is a vector of character strings giving the names.

When S-PLUS looks for an object, it searches in turn through a sequence of places 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 places currently on the search list are displayed by the function

```
> search()
```

The names of the objects held in any place in the search list can be displayed by giving the `objects()` function an argument. For example

```
> objects(2)
```

lists the contents of the entity at position 2 of the search list. The search list can contain either data frames and allies, which are themselves internal S-PLUS objects, as well as directories of files which are UNIX objects.

Extra entities can be added to this list with the `attach()` function and removed with the `detach()` function, details of which can be found in the manual or the `help` facility.

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

```
> rm(x, y, z, ink, junk, temp, foo, bar)
```

The function `remove()` can be used to remove objects with non-standard names. Also the ordinary UNIX facility, `rm`, may be used to remove the appropriate files in the `.Data` directory, as mentioned above.

2 Simple manipulations; numbers and vectors

2.1 Vectors and Assignments

S-PLUS 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-PLUS command

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

This is an ***assignment*** statement using the ***function*** c() which in this context can take an arbitrary number of vector ***arguments*** and whose value is a vector got by concatenating its arguments end to end.[1]

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

Notice that the assignment operator is **not** the usual = operator, which is reserved for another purpose. It consists of the two characters < ('less than') and - ('minus') occurring strictly side-by-side and it 'points' to the object receiving the value of the expression.[2]

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 at the terminal (and the value of x, of course, unchanged).

The further assignment

```
> y <- c(x, 0, x)
```

would create a vector y with 11 entries consisting of two copies of x with a zero in the middle place.

2.2 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.

[1] With other than vector types of argument, such as list mode arguments, the action of c() is rather different. See §6.2.1.

[2] The underscore character, "_" is an allowable synonym for the left pointing assignment operator "<-", however we discourage this option, as it can easily lead to much less readible code.

The elementary arithmetic operators are the usual `+`, `-`, `*`, `/` and `^` for raising to a power. In addition all of the common arithmetic functions are available. `log`, `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))`. `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 calculates the sample mean, which is the same as `sum(x)/length(x)`, and `var(x)` which gives

```
sum((x-mean(x))^2)/(length(x)-1)
```

or sample variance. If the argument to `var()` is an $n \times p$ matrix the value is a $p \times p$ sample covariance matrix got by 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; however there are other more flexible sorting facilities available (see `order()` or `sort.list()` which produce a permutation to do the sorting).

`rnorm(x)` is a function which generates a vector (or more generally an array) of pseudo-random standard normal deviates, of the same size as `x`.

2.3 Generating regular sequences

S-PLUS 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 sequence backwards.

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-PLUS 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 an object 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`.

2.4 Logical vectors

As well as numerical vectors, S-PLUS allows manipulation of logical quantities. The elements of a logical vectors have just two possible values, represented formally as `F` (for 'false') and `T` (for 'true').

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 & c2` is their intersection, `c1 | c2` is their union 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, for example see the next subsection.

2.5 Missing values

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)
```

Notice that the logical expression `x == NA` is quite different from `is.na(x)` since `NA` is not really a value but a marker for a quantity that is not available. Thus `x == NA` is a vector of the same length as `x` ***all*** of whose values are `NA` as the logical expression itself is incomplete and hence undecidable.

2.6 Character vectors

Character quantities and character vectors are used frequently in S-PLUS, for example as plot labels. Where needed they are denoted by a sequence of characters delimited by the double quote character. E. g. `"x-values"`, `"New iteration results"`.

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

The `paste()` function takes an arbitrary number of arguments and concatenates them into a single character string. Any numbers given among the arguments are coerced into character strings in the evident way, that is, 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` into the character vector

```
("X1", "Y2", "X3", "Y4", "X5", "Y6", "X7", "Y8", "X9", "Y10")
```

Note particularly that recycling of short lists takes place here too; thus `c("X", "Y")` is repeated 5 times to match the sequence `1:10`.

2.7 Index vectors. Selecting and modifying subsets of a data set

Subsets of the elements of a vector may be selected by appending to the name of the vector an ***index vector*** in square brackets. More generally any expression that evaluates to a vector may have subsets of its elements similarly selected by appending an index vector in square brackets immediately after the expression.

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.** Such an 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. above.

   ```
   > fruit <- c(5, 10, 1, 20)
   ```

```
> names(fruit) <- c("orange", "banana", "apple", "peach")
```

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

The advantage is that alphanumeric ***names*** are often easier to remember than ***numeric indices***. This option is particularly useful in connection with data frames, as we shall see later.

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 does not make much sense here.

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)
```
[3]

[3]Note that `abs()` does not work as expected with complex arguments. The appropriate function for the complex modulus is `Mod()`.

3 Objects, their modes and attributes

3.1 Intrinsic attributes: *mode* and *length*

The entities S-PLUS operates on are technically known as ***objects***. Examples are vectors of numeric (real) or complex values, vectors of logical values and vectors of character strings. These are known as 'atomic' structures since their components are all of the same type, or ***mode***, namely ***numeric***[4], ***complex***, ***logical*** and ***character*** respectively.

Vectors must have their values ***all of the same mode***. Thus any given vector must be unambiguously either ***logical***, ***numeric***, ***complex*** or ***character***. The only mild exception to this rule is the special "value" listed as `NA` for quantities not available. Note that a vector can be empty and still have a mode. For example the empty character string vector is listed as `character(0)` and the empty numeric vector as `numeric(0)`.

S-PLUS also operates on objects called ***lists***, which are of mode ***list***. These are ordered sequences of objects which individually can be of any mode. ***lists*** are known as 'recursive' rather than atomic structures since their components can themselves be lists in their own right.

The other recursive structures are those of mode ***function*** and ***expression***. ***Functions*** are the functions that form part of the S-PLUS system along with similar user written functions, which we discuss in some detail later in these notes. ***Expressions*** as objects form an advanced part of S-PLUS which will not be discussed in these notes, except indirectly when we discuss ***formulæ*** used with modelling in S-PLUS.

By the ***mode*** of an object we mean the basic type of its fundamental constituents. This is a special case of an ***attribute*** of an object. The ***attributes*** of an object provide specific information about the object itself. Another attribute of every object is its ***length***. The functions `mode(`***object***`)` and `length(`***object***`)` can be used to find out the mode and length of any defined structure.

For example, if `z` is a complex vector of length 100, then in an expression `mode(z)` is the character string `"complex"` and `length(z)` is `100`.

S-PLUS caters for changes of mode almost anywhere it could be considered sensible to do so, (and a few where it might not be). For example with

```
> z <- 0:9
```

we could put

```
> digits <- as.character(z)
```

after which `digits` is the character vector `("0", "1", "2", ..., "9")`. A further ***coercion***, or change of mode, reconstructs the numerical vector again:

```
> d <- as.numeric(digits)
```

Now `d` and `z` are the same.[5] There is a large collection of functions of the form `as.something()` for either coercion from one mode to another, or for investing an object with some other attribute it may not already possess. The reader should consult the help file to become familiar with them.

[4] ***numeric*** mode is actually an amalgam of three distinct modes, namely ***integer***, ***single*** precision and ***double*** precision, as explained in the manual.

[5] In general coercion from numeric to character and back again will not be exactly reversible, because of roundoff errors in the character representation.

3.2 Changing the length of an object

An "empty" object may still have a mode. For example

```
> e <- numeric()
```

makes `e` an empty vector structure of mode numeric. Similarly `character()` is a empty character vector, and so on. Once an object of any size has been created, new components may be added to it simply by giving it an index value outside its previous range. Thus

```
> e[3] <- 17
```

now makes `e` a vector of length 3, (the first two components of which are at this point both `NA`). This applies to any structure at all, provided the mode of the additional component(s) agrees with the mode of the object in the first place.

This automatic adjustment of lengths of an object is used often, for example in the `scan()` function for input. (See §7.2.)

Conversely to truncate the size of an object requires only an assignment to do so. Hence if `alpha` is an object of length 10, then

```
> alpha <- alpha[2 * 1:5]
```

makes it an object of length 5 consisting of just the former components with even index. The old indices are not retained, of course.

3.3 `attributes()` and `attr()`

The function `attributes(`***object***`)` gives a list of all the non-intrinsic attributes currently defined for that object. The function `attr(`***object, name***`)` can be used to select a specific attribute. These functions are rarely used, except in rather special circumstances when some new attribute is being created for some particular purpose, for example to associate a creation date or an operator with an S-PLUS object. The concept, however, is very important.

3.4 The *class* of an object

A special attribute known as the ***class*** of the object has been introduced in the August 1991 release of S and S-PLUS to allow for an object oriented style of programming in S-PLUS.

For example if an object has class `data.frame`, it will be printed in a certain way, the `plot()` function will display it graphically in a certain way, and other generic functions such as `summary()` will react to it as an argument in a way sensitive to its class.

To remove temporarily the effects of class, use the function `unclass()`. For example if `winter` has the class `data.frame` then

```
> winter
```

will print it in data frame form, which is rather like a matrix, whereas

```
> unclass(winter)
```

will print it as an ordinary list. Only in rather special situations do you need to use this facility, but one is when you are learning to come to terms with the idea of class and generic functions.

Generic functions and classes will be discussed further in §9.7, but only briefly.

4 Categories and factors

A ***category*** is a vector object used to specify a discrete classification of the components of other vectors of the same length. A ***factor*** is similar, but has the ***class*** `factor`, which means that it is adapted to the generic function mechanism. Whereas a category can also be used as a plain numeric vector, for example, a factor generally cannot.

4.1 A specific example

Suppose, for example, we have a sample of 30 tax accountants from the all states and territories[6] and their individual state of origin is specified by a character vector of state mnemonics as

```
> state <- c("tas", "sa",  "qld", "nsw", "nsw", "nt",  "wa",  "wa",
             "qld", "vic", "nsw", "vic", "qld", "qld", "sa",  "tas",
             "sa",  "nt",  "wa",  "vic", "qld", "nsw", "nsw", "wa",
             "sa",  "act", "nsw", "vic", "vic", "act")
```

For some purposes it is convenient to represent such information by a numeric vector with the distinct values in the original (in this case the state labels) represented by a small integer. Such a numeric vector is called a ***category***. However at the same time it is important to preserve the correspondence of these new integer labels with the originals. This is done via the ***levels*** attribute of the category.

More formally, when a category is formed from such a vector the ***sorted unique values*** in the vector form the ***levels*** attribute of the category, and the values in the category are in the set `1, 2, ..., k` where `k` is the number of unique values. The value at position `j` in the factor is `i` if the `i`th sorted unique value occurred at position `j` of the original vector.

Hence the assignment

```
> stcode <- category(state)
```

creates a category with values and attributes as follows

```
> stcode
 [1] 6 5 4 2 2 3 8 8 4 7 2 7 4 4 5 6 5 3 8 7 4 2 2 8 5 1 2 7 7 1
 attr(, "levels"):
 [1] "act" "nsw" "nt"  "qld" "sa"  "tas" "vic" "wa"
```

Notice that in the case of a character vector, “sorted” means sorted in alphabetical order.

A ***factor*** is similarly created using the `factor()` function:

```
> statef <- factor(state)
```

The `print()` function now handles the factor object slightly differently:

```
> statef
 [1] tas sa  qld nsw nsw nt  wa  wa  qld vic nsw vic qld qld sa
[16] tas sa  nt  wa vic qld nsw nsw wa  sa  act nsw vic vic act
```

If we remove the factor class, however, using the function `unclass()`, it becomes virtually identical to the category:

[6]Foreign readers should note that there are eight states and territories in Australia, namely the Australian Capital Territory, New South Wales, the Northern Territory, Queensland, South Australia, Tasmania, Victoria and Western Australia.

```
> unclass(statef)
 [1] 6 5 4 2 2 3 8 8 4 7 2 7 4 4 5 6 5 3 8 7 4 2 2 8 5 1 2 7 7 1
attr(, "levels"):
[1] "act" "nsw" "nt"  "qld" "sa"  "tas" "vic" "wa"
```

4.2 The function `tapply()` and ragged arrays

To continue the previous example, suppose we have the incomes of the same tax accountants in another vector (in suitably large units of money)

```
> incomes <- c(60, 49, 40, 61, 64, 60, 59, 54, 62, 69, 70, 42, 56,
               61, 61, 61, 58, 51, 48, 65, 49, 49, 41, 48, 52, 46,
               59, 46, 58, 43)
```

To calculate the sample mean income for each state we can now use the special function `tapply()`:

```
> incmeans <- tapply(incomes, statef, mean)
```

giving a means vector with the components labelled by the levels

```
> incmeans
  act    nsw   nt  qld sa  tas vic    wa
 44.5 57.333 55.5 53.6 55 60.5  56 52.25
```

The function `tapply()` is used to apply a function, here `mean()`, to each group of components of the first argument, here `incomes`, defined by the levels of the second component, here `statef`, as if they were separate vector structures. The result is a structure of the same length as the levels attribute of the factor containing the results. The reader should consult the help document for more details.

Suppose further we needed to calculate the standard errors of the state income means. To do this we need to write an S-PLUS function to calculate the standard error for any given vector. We discuss functions more fully later in these notes, but since there is an in built 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))
```

(Writing functions will be considered later in §9.) After this assignment, the standard errors are calculated by

```
> incster <- tapply(incomes, statef, stderr)
```

and the values calculated are then

```
> incster
 act    nsw  nt    qld     sa tas   vic     wa
 1.5 4.3102 4.5 4.1061 2.7386 0.5 5.244 2.6575
```

As an exercise you may care to find the usual 95% confidence limits for the state mean incomes. To do this you could use `tapply()` once more with the `length()` function to find the sample sizes, and the `qt()` function to find the percentage points of the appropriate $t-$distributions.

The function `tapply()` can be used to handle more complicated indexing of a vector by multiple categories. For example, we might wish to split the tax accountants by both state and sex. However in this simple instance what happens can be thought of as follows. The values in the vector are collected into groups corresponding to the distinct entries in the category. The function is then applied to each of these groups individually. The value is a vector of function results, labelled by the levels attribute of the category.

The combination of a vector and a labelling factor or category 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, as we see in the next section.

5 Arrays and matrices

5.1 Arrays

An array can be considered as a multiply subscripted collection of data entries, for example numeric. S-PLUS allows simple facilities for creating and handling arrays, and in particular the special case of matrices.

A dimension vector is 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.

A vector can be used by S-PLUS as an array only if it has a dimension vector as its ***dim*** attribute. Suppose, for example, `z` is a vector of 1500 elements. The assignment

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

gives it the ***dim*** attribute that allows it 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, as we shall see in §5.4.

The values in the data vector give the values in the array in the same order as they would occur in Fortran, that is 'column major order', 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]`.

5.2 Array indexing. Subsections of an array

Individual elements of an array may be referenced, as above, 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.*

Continuing the previous example, `a[2,,]` is a 4×2 array with dimension vector `c(4,2)` and data vector containing the values

```
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.

For any array, say `Z`, the dimension vector may be referenced explicitly as `dim(Z)` (on either side of an assignment).

Also, if an array name is given with just ***one subscript or index vector***, then the corresponding values of the data vector only are used; in this case the dimension vector is ignored. This is not the case, however, if the single index is not a vector but itself an array, as we next discuss.

5.3 Index arrays

As well as an index vector in any subscript position, an array may be used with a single ***index array*** in order either to assign a vector of quantities to an irregular collection of elements in the array, or to extract an irregular collection as a vector.

A matrix example makes the process clear. In the case of a doubly indexed array, an index matrix may be given consisting of two columns and as many rows as desired. The entries in the index matrix are the row and column indices for the doubly indexed array. Suppose for example we have a 4×5 array `X` and we wish to do the following:

- Extract elements `X[1,3]`, `X[2,2]` and `X[3,1]` as a vector structure, and
- Replace these entries in the array `X` by 0s.

In this case we need a 3×2 subscript array, as in the example given in Figure 1.

```
> x <- array(1:20,dim=c(4,5))   # Generate a 4 x 5 array.
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
> i <- array(c(1:3,3:1),dim=c(3,2))
> i                             # i is a 3 x 2 index array.
     [,1] [,2]
[1,]    1    3
[2,]    2    2
[3,]    3    1
> x[i]                          # Extract those elements
[1] 9 6 3
> x[i] <- 0                     # Replace those elements by zeros.
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    0   13   17
[2,]    2    0   10   14   18
[3,]    0    7   11   15   19
[4,]    4    8   12   16   20
>
```

Figure 1: Using an index array

As a less trivial example, suppose we wish to generate an (unreduced) design matrix for a block design defined by factors `blocks` (`b` levels) and `varieties`, (`v` levels). Further suppose there are `n` plots in the experiment. We could proceed as follows:

```
> Xb <- matrix(0, n, b)
> Xv <- matrix(0, n, v)
> ib <- cbind(1:n, blocks)
> iv <- cbind(1:n, varieties)
> Xb[ib] <- 1
> Xv[iv] <- 1
> X <- cbind(Xb, Xv)
```

Further, to construct the incidence matrix, `N` say, we could use

```
> N <- crossprod(Xb, Xv)
```

However a simpler direct way of producing this matrix is to use `table()`:

```
> N <- table(blocks, varieties)
```

5.4 The `array()` function

As well as giving a vector structure a `dim` attribute, arrays can be constructed from vectors by the `array` function, which has the form

> `> Z <- array(`***data_vector, dim_vector***`)`

For example, if the vector `h` contains 24, or fewer, numbers then the command

```
> Z <- array(h, dim=c(3,4,2))
```

would use `h` to set up $3 \times 4 \times 2$ array in `Z`. If the size of `h` is exactly 24 the result is the same as

```
> dim(Z) <- c(3,4,2)
```

However if `h` is shorter than 24, its values recycled from the beginning again to make it up to size 24. See §5.4.1 below. As an extreme but common example

```
> Z <- array(0, c(3,4,2))
```

makes Z an array of all zeros.

At this point `dim(Z)` stands for the dimension vector `c(3,4,2)`, and `Z[1:24]` stands for the data vector as it was in `h`, and `Z[]` with an empty subscript or `Z` with no subscript stands for the entire array as an array.

Arrays may be used in arithmetic expressions and the result is an array formed by element by element operations on the data vector. The `dim` attributes 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. However the precise rule concerning mixed array and vector calculations has to be considered a little more carefully.

5.4.1 Mixed vector and array arithmetic. The recycling rule

The precise rule affecting element by element mixed calculations with vectors and arrays is somewhat quirky and hard to find in the references. From experience I have found the following to be a reliable guide.

- The expression is scanned from left to right.
- Any short vector operands are extended by recycling their values until they match the size of any previous (or subsequent) operands.
- As long as short vectors and arrays, only, are encountered, the arrays must all have the same `dim` attribute or an error results.
- Any vector operand longer than some previous array immediately converts the calculation to one in which all operands are coerced to vectors. A diagnostic message is issued if the size of the long vector is not a multiple of the (common) size of all previous arrays.

- If array structures are present and no error or coercion to vector has been precipitated, the result is an array structure with the common `dim` attribute of its array operands.

5.5 The outer product of two arrays

An important operation on arrays is the ***outer product***. If `a` and `b` are two numeric arrays, their outer product is an array whose dimension vector is got by concatenating their two dimension vectors, (order is important), and whose data vector is got by forming all possible products of elements of the data vector of `a` with those of `b`. The outer product is formed by the special operator `%o%`:

```
> ab <- a %o% b
```

An alternative is

```
> ab <- outer(a, b, '*')
```

The multiplication function can be replaced by an arbitrary function of two variables. For example if we wished to evaluate the function

$$f(x, y) = \frac{\cos(y)}{1 + x^2}$$

over a regular grid of values with $x-$ and $y-$coordinates defined by the S-PLUS vectors `x` and `y` respectively, we could proceed as follows:

```
> f <- function(x,y) cos(y)/(1 + x^2)
> z <- outer(x, y, f)
```

In particular the outer product of two ordinary vectors is a doubly subscripted array (that is a matrix, of rank at most 1). Notice that the outer product operator is of course non-commutative. Defining your own S-PLUS functions will be considered further in Chapter 9.

5.5.1 An example: Determinants of 2×2 digit matrices

As an artificial but cute example, consider the determinants of 2×2 matrices $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ where each entry is a non-negative integer in the range $0, 1, \ldots, 9$, that is a digit.

The problem is to find the determinants, $ad - bc$, of all possible matrices of this form and represent the frequency with which each value occurs as a ***high density*** plot. This amounts to finding the probability distribution of the determinant if each digit is chosen independently and uniformly at random.

A neat way of doing this uses the `outer()` function twice:

```
> d <- outer(0:9, 0:9)
> fr <- table(outer(d, d, "-"))
> plot(as.numeric(names(fr)), fr, type="h",
        xlab="Determinant", ylab="Frequency")
```

Notice the coercion of the `names` attribute of the frequency table to numeric in order to recover the range of the determinant values. The "obvious" way of doing this problem with `for`–loops, to be discussed in §8.2, is so inefficient as to be impractical.

It is also perhaps surprising that about 1 in 20 such matrices is singular.

5.6 Generalized transpose of an array

The function `aperm(a, perm)` may be used to permute an array, `a`. The argument `perm` must be a permutation of the integers {`1, 2, ..., k`}, where `k` is the number of subscripts in `a`. The result of the function is an array of the same size as `a` but with old dimension given by `perm[j]` becoming the new `j`th dimension. The easiest way to think of this operation is as a generalization of transposition for matrices. Indeed if `A` is a matrix, (that is, a doubly subscripted array) then `B` given by

```
> B <- aperm(A, c(2,1))
```

is just the transpose of `A`. For this special case a simpler function `t()` is available, so we could have used `B <- t(A)`.

5.7 Matrix facilities. Multiplication, inversion and solving linear equations.

As noted above, a matrix is just an array with two subscripts. However it is such an important special case it needs a separate discussion. S-PLUS contains many operators and functions that are available only for matrices. For example `t(X)` is the matrix transpose function, as noted above. The functions `nrow(A)` and `ncol(A)` give the number of rows and columns in the matrix `A` respectively.

The operator `%*%` is used for matrix multiplication. An $n \times 1$ or $1 \times n$ matrix may of course be used as an $n-$vector if in the context such is appropriate. Conversely vectors which occur in matrix multiplication expressions are automatically promoted either to row or column vectors, whichever is multiplicatively coherent, if possible, (although this is not always unambiguously possible, as we see later).

If, for example, `A` and `B` are square matrices of the same size, then

```
> A * B
```

is the matrix of element by element products and

```
> A %*% B
```

is the matrix product. If `x` is a vector, then

```
> x %*% A %*% x
```

is a quadratic form.[7]

The function `crossprod()` forms "crossproducts", meaning that

`> crossprod(X, y)` is the same as `t(X) %*% y`

but the operation is more efficient. If the second argument to `crossprod()` is omitted it is taken to be the same as the first.

Other important matrix functions include `solve(A, b)` for solving equations, `solve(A)` for the matrix inverse, `svd()` for the singular value decomposition, `qr()` for QR decomposition and `eigen()` for eigenvalues and eigenvectors of symmetric matrices.

The meaning of `diag()` depends on its argument. `diag(vector)` gives a diagonal matrix with elements of the vector as the diagonal entries. On the other hand `diag(matrix)` gives the vector of main diagonal entries of `matrix`. This is the same

[7] Note that `x %*% x` is ambiguous, as it could mean either $\mathbf{x}'\mathbf{x}$ or $\mathbf{x}\mathbf{x}'$, where $\mathbf{x}$ is the column form. In such cases the smaller matrix seems implicitly to be the interpretation adopted, so the scalar $\mathbf{x}'\mathbf{x}$ is in this case the result. The matrix $\mathbf{x}\mathbf{x}'$ may be calculated either by `cbind(x) %*% x` or `x %*% rbind(x)` since the result of `rbind()` or `cbind()` is always a matrix.

convention as that used for `diag()` in MATLAB. Also, somewhat confusingly, if `k` is a single numeric value then `diag(k)` is the $k \times k$ identity matrix!

A surprising omission from the suite of matrix facilities is a function for the determinant of a square matrix, however the absolute value of the determinant is easy to calculate for example as the product of the singular values. (See later.)

5.8 Forming partitioned matrices. `cbind()` and `rbind()`.

As we have already seen informally, matrices can be built up from other vectors and matrices by the functions `cbind()` and `rbind()`. Roughly `cbind()` forms matrices by binding together matrices horizontally, or column-wise, and `rbind()` vertically, or row-wise.

In the assignment

```
> X <- cbind(arg1, arg2, arg3, ...)
```

the arguments to `cbind()` must be either vectors of any length, or matrices with the same column size, that is the same number of rows. The result is a matrix with the concatenated arguments arg_1, arg_2, ... forming the columns.

If some of the arguments to `cbind()` are vectors they may be shorter than the column size of any matrices present, in which case they are cyclically extended to match the matrix column size (or the length of the longest vector if no matrices are given).

The function `rbind()` does the corresponding operation for rows. In this case any vector argument, possibly cyclically extended, are of course taken as row vectors.

Suppose `X1` and `X2` have the same number of rows. To combine these by columns into a matrix `X`, together with an initial column of `1`s we can use

```
> X <- cbind(1, X1, X2)
```

The result of `rbind()` or `cbind()` always has matrix status. Hence `cbind(x)` and `rbind(x)` are possibly the simplest ways explicitly to allow the vector `x` to be treated as a column or row matrix respectively.

5.9 The concatenation function, `c()`, with arrays.

It should be noted that whereas `cbind()` and `rbind()` are concatenation functions that respect `dim` attributes, the basic `c()` function does not, but rather clears numeric objects of all `dim` and `dimnames` attributes. This is occasionally useful in its own right.

The official way to coerce an array back to a simple vector object is to use `as.vector()`

```
> vec <- as.vector(X)
```

However a similar result can be achieved by using `c()` with just one argument, simply for this side-effect:

```
> vec <- c(X)
```

There are slight differences between the two, but ultimately the choice between them is largely a matter of style (with the former being preferable).

5.10 Frequency tables from factors. The `table()` function

Recall that a factor defines a partition into groups. Similarly a pair of factors defines a two way cross classification, and so on. The function `table()` allows frequency tables to be calculated from equal length factors. If there are k category arguments, the result is a $k-$way array of frequencies.

Suppose, for example, that `statef` is a factor giving the state code for each entry in a data vector. The assignment

```
> statefr <- table(statef)
```

gives in `statefr` a table of frequencies of each state in the sample. The frequencies are ordered and labelled by the levels attribute of the category. This simple case is equivalent to, but more convenient than,

```
> statefr <- tapply(statef, statef, length)
```

Further suppose that `incomef` is a category giving a suitably defined "income class" for each entry in the data vector, for example with the `cut()` function:

```
> factor(cut(incomes,breaks=35+10*(0:7))) -> incomef
```

Then to calculate a two-way table of frequencies:

```
> table(incomef,statef)
               act nsw nt qld sa tas vic wa
    35+ thru 45   1   1  0   1  0   0   1  0
    45+ thru 55   1   1  1   1  2   0   1  3
    55+ thru 65   0   3  1   3  2   2   2  1
    65+ thru 75   0   1  0   0  0   0   1  0
```

Extension to higher way frequency tables is immediate.

6 Lists, data frames, and their uses

6.1 Lists

An S-PLUS ***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 complex vector, a character array, a function, and so on. Here is a simple example of how to make a list:

```
> Lst <- list(name="Fred", wife="Mary", no.children=3, child.ages=c(4,7,9))
```

Components are always ***numbered*** and may always be referred to as such. Thus if `Lst` is the name of a list with four components, these may be individually referred to as `Lst[[1]]`, `Lst[[2]]`, `Lst[[3]]` and `Lst[[4]]`. If, further, `Lst[[4]]` is a vector subscripted array then `Lst[[4]][1]` is its first entry.

If `Lst` is a list, then the function `length(Lst)` gives the number of (top level) components it has.

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.

So in the simple example given above:

`Lst$name` is the same as `Lst[[1]]` and is the string `"Fred"`,

`Lst$wife` is the same as `Lst[[2]]` and is the string `"Mary"`,

`Lst$child.ages[1]` is the same as `Lst[[4]][1]` and is the number `4`.

It is very important to distinguish `Lst[[1]]` from `Lst[1]`. "`[[...]]`" is the operator used to select a single element, whereas "`[...]`" is a general subscripting operator. Thus the former is the ***first object in the list*** `Lst`, and if it is a named list the name is ***not*** included. The latter is a ***sublist of the list*** `Lst` ***consisting of the first entry only. If it is a named list, the name is transferred to the sublist.***

The names of components may be abbreviated down to the minimum number of letters needed to identify them uniquely. Thus `Lst$coefficients` may be minimally specified as `Lst$coe` and `Lst$covariance` as `Lst$cov`.

The vector of names is in fact simply an attribute of the list like any other and may be handled as such. Other structures besides lists may, of course, similarly be given a ***names*** attribute also.

6.2 Constructing and modifying lists

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

> `Lst <- list(`***name***$_1$=***object***$_1$, ***name***$_2$=***object***$_2$, ..., ***name***$_m$=***object***$_m$)

sets up a list `Lst` of m components using ***object***$_1$, ..., ***object***$_m$ for the components and giving them names as specified by the argument names, (which can be freely chosen). If these names are omitted, the components are numbered only. The components used to form the list are ***copied*** when forming the new list and the originals are not affected.

Lists, like any subscripted object, can be extended by specifying additional components. For example

```
> Lst[5] <- list(matrix=Mat)
```

6.2.1 Concatenating lists

When the concatenation function `c()` is given list arguments, the result is an object of mode list also, whose components are those of the argument lists joined together in sequence.

```
> list.ABC <- c(list.A, list.B, list.C)
```

Recall that with vector objects as arguments the concatenation function similarly joined together all arguments into a single vector structure. In this case all other attributes, such as `dim` attributes, are discarded.

6.3 Some functions returning a list result

Functions and expressions in S-PLUS must return a single object as their result; in cases where the result has several component parts, the usual form is that of a list with named components.

6.3.1 Eigenvalues and eigenvectors

The function `eigen(Sm)` calculates the eigenvalues and eigenvectors of a symmetric matrix `Sm`. The result of this function is a list of two components named `values` and `vectors`. The assignment

```
> ev <- eigen(Sm)
```

will assign this list to `ev`. Then `ev$val` is the vector of eigenvalues of `Sm` and `ev$vec` is the matrix of corresponding eigenvectors. Had we only needed the eigenvalues we could have used the assignment:

```
> evals <- eigen(Sm)$values
```

`evals` now holds the vector of eigenvalues and the second component is discarded. If the expression

```
> eigen(Sm)
```

is used by itself as a command the two components are printed, with their names, at the terminal.

6.3.2 Singular value decomposition and determinants

The function `svd(M)` takes an arbitrary matrix argument, `M`, and calculates the singular value decomposition of `M`. This consists of a matrix of orthonormal columns `U` with the same column space as `M`, a second matrix of orthonormal columns `V` whose column space is the row space of `M` and a diagonal matrix of positive entries `D` such that `M = U %*% D %*% t(V)`. `D` is actually returned as a vector of the diagonal elements. The result of `svd(M)` is actually a list of three components named `d`, `u` and `v`, with evident meanings.

If `M` is in fact square, then, it is not hard to see that

```
> absdetM <- prod(svd(M)$d)
```

calculates the absolute value of the determinant of `M`. If this calculation were needed often with a variety of matrices it could be defined as an S-PLUS function

```
> absdet <- function(M) prod(svd(M)$d)
```

after which we could use `absdet()` as just another S-PLUS function. As a further trivial but potentially useful example, you might like to consider writing a function, say `tr()`, to calculate the trace of a square matrix. [Hint: You will not need to use an explicit loop. Look again at the `diag()` function.]

Functions will be discussed formally later in these notes.

6.3.3 Least squares fitting and the QR decomposition

The function `lsfit()` returns a list giving results of a least squares fitting procedure. An assignment such as

```
> ans <- lsfit(X, y)
```

gives the results of a least squares fit where `y` is the vector of observations and `X` is the design matrix. See the help facility for more details, and also for the follow-up function `ls.diag()` for, among other things, regression diagnostics. Note that a grand mean term is automatically included and need not be included explicitly as a column of `X`.

Another closely related function is `qr()` and its allies. Consider the following assignments

```
> Xplus <- qr(X)
> b <- qr.coef(Xplus, y)
> fit <- qr.fitted(Xplus, y)
> res <- qr.resid(Xplus, y)
```

These compute the orthogonal projection of `y` onto the range of `X` in `fit`, the projection onto the orthogonal complement in `res` and the coefficient vector for the projection in `b`, that is, `b` is essentially the result of the `MATLAB` 'backslash' operator.

It is not assumed that `X` has full column rank. Redundancies will be discovered and removed as they are found.

This alternative is the older, low level way to perform least squares calculations. Although still useful in some contexts, it would now generally be replaced by the statistical models features, as will be discussed in §10.

6.4 Data frames

A ***data frame*** is a list with class `data.frame`. There are restrictions on lists that may be made into data frames, namely

- The components must be vectors (numeric, character, or logical), factors, numeric matrices, lists, or other data frames.

- Matrices, lists, and data frames provide as many variables to the new data frame as they have columns, elements, or variables, respectively.
- Numeric vectors and factors are included as is, and non-numeric vectors are coerced to be factors, whose levels are the unique values appearing in the vector.
- Vector structures appearing as variables of the data frame must all have the ***same length***, and matrix structures must all have the same ***row size***.

Data frames may in many ways be regarded as a matrix with columns possibly of differing modes and attributes. It may be displayed in matrix form, and its rows and columns extracted using matrix indexing conventions.

6.4.1 Making data frames

Objects satisfying the restrictions placed on the columns (components) of a data frame may be used to form one using the function `data.frame`:

```
> accountants <- data.frame(home=statef,loot=income, shot=incomef)
```

A list whose components conform to the restrictions of a data frame may be ***coerced*** into a data frame using the function `as.data.frame()`

The simplest way to construct a data frame from scratch is to use the `read.table()` function to read an entire data frame from an external file. This is discussed further in §7.

6.4.2 `attach()` and `detach()`

The `$` notation, such as `accountants$statef`, for list components is not always very convenient. A useful facility would be somehow to make the components of a list or data frame temporarily visible as variables under their component name, without the need to quote the list name explicitly each time.

The `attach()` function, as well as having a directory name as its argument, may also have a data frame. Thus suppose `lentils` is a data frame with three variables `lentils$u`, `lentils$v`, `lentils$w`. The attach

```
> attach(lentils)
```

places the data frame in the search list at position 2, and provided there are no variables `u`, `v` or `w` in position 1, `u`, `v` and `w` are available as variables from the data frame in their own right. At this point an assignment such as

```
> u <- v+w
```

does not replace the component `u` of the data frame, but rather masks it with another variable `u` in the working directory at position 1 on the search list. To make a permanent change to the data frame itself, the simplest way is to resort once again to the `$` notation:

```
> lentils$u <- v+w
```

However the new value of component `u` is not visible until the data frame is detached and attached again.

To detach a data frame, use the function

```
> detach()
```

More precisely, this statement detaches from the search list the entity currently at position 2. Thus in the present context the variables `u`, `v` and `w` would be no longer visible, except under the list notation as `lentils$u` and so on.

NOTE: With the current release of S-PLUS the search list can contain at most 20 items. Avoid attaching the same data frame more than once. Always detach the data frame as soon as you have finished using its variables.

6.4.3 Working with data frames

A useful convention that allows you to work with many different problems comfortably together in the same working directory is

- gather together all variables for any well defined and separate problem in a data frame under a suitably informative name;
- when working with a problem attach the appropriate data frame at position 2, and use the working directory at level 1 for operational quantities and temporary variables;
- before leaving a problem, add any variables you wish to keep for future reference to the data frame using the `$` form of assignment, and then `detach()`;
- finally remove all unwanted variables from the working directory and keep it as clean of left-over temporary variables as possible.

In this way it is quite simple to work with many problems in the same directory, all of which have variables named `x`, `y` and `z`, for example.

6.4.4 Attaching arbitrary lists

`attach()` is a generic function that allows not only directories and data frames to be attached to the search list, but other classes of object as well. In particular any object of mode `list` may be attached in the same way:

```
> attach(any.old.list)
```

It is also possible to attach objects of class `pframe`, to so-called *parametrized data frames*, needed for nonlinear regression and elsewhere.

Being a generic function it is also possible to add methods for attaching yet more classes of object should the need arise.

7 Reading data from files

Large data objects will usually be read as values from external files rather than entered during an S-PLUS session at the keyboard. S-PLUS input facilities are simple and their requirements are fairly strict and even rather inflexible. There is a clear presumption by the designers of S-PLUS that you will be able to modify your input files using other tools, such file editors and the UNIX utilities `sed` and `awk` to fit in with the requirements of S-PLUS. Generally this is very simple.

There is, however, a function `make.fields()` that can be used to convert a file with fixed width, non separated, input fields into a file with separated fields. There is also a facility `count.fields()` that will count the number of fields on each line of such a file. Occasionally for very simple conversion and checking problems these may be adequate to the task, but in most cases it is better to do the preliminary spade work before the S-PLUS session begins.

If variables are to be held mainly in data frames, as we strongly suggest, they should be, an entire data frame can be read directly with the `read.table()` function. There is also an older, more primitive input function, `scan()`, that is still occasionally useful in special circumstances.

7.1 The `read.table()` function

To read an entire data frame directly, the external file will normally have a special form.

- The first line of the file should have a ***name*** for each variable in the data frame.
- Each additional line of the file has its first item a ***row label*** and the values for each variable.

If the file has one fewer item in its first line than in its second, this arrangement is presumed to be in force. So the first few lines of a file to be read as a data frame might look as in Figure 2. By default numeric items (except row labels) are read as numeric

```
     Price    Floor     Area   Rooms     Age  Cent.heat
01   52.00    111.0      830     5       6.2      no
02   54.75    128.0      710     5       7.5      no
03   57.50    101.0     1000     5       4.2      no
04   57.50    131.0      690     6       8.8      no
05   59.75     93.0      900     5       1.9     yes
...
```

Figure 2: Input file form with names and row labels

variables and non-numeric variables, such as `Cent.heat` in the example, as factors. This can be changed if necessary.

The function `read.table()` can then be used to read the data frame directly

```
> HousePrice <- read.table("houses.data")
```

Often you will want to omit including the row labels directly and use the default labels. In this case the file may omit the row label column as in Figure 3. The data frame may then be read as

```
> HousePrice <- read.table("houses.data", header=T)
```

```
   Price    Floor     Area   Rooms     Age  Cent.heat
   52.00    111.0      830     5       6.2      no
   54.75    128.0      710     5       7.5      no
   57.50    101.0     1000     5       4.2      no
   57.50    131.0      690     6       8.8      no
   59.75     93.0      900     5       1.9     yes
...
```

Figure 3: Input file form without row labels

where the `heading=T` option specifies that the first line is a line of headings, and hence, by implication from the form of the file, that no explicit row labels are given.

7.2 The `scan()` function

Suppose the data vectors are of equal length and are to be read in in parallel. Further suppose that there are three vectors, the first of mode character and the remaining two of mode numeric, and the file is `input.dat`. The first step is to use `scan()` to read in the three vectors as a list, as follows

```
> in <- scan("input.dat", list("",0,0))
```

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 components are the three vectors read in. To separate the data items into three separate vectors, use assignments like

```
> label <- in[[1]]; x <- in[[2]]; y <- in[[3]]
```

More conveniently, the dummy list can have named components, in which case the names can be used to access the vectors read in. For example

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

If you wish to access the variables separately they may either be re-assigned to variables in the working frame:

```
> label <- in$id; x <- in$x; y <- in$y
```

or the list may be attached at position 2 of the search list, (see §6.4.4).

If the second argument is a single value and not a list, a single vector is read in, all components of which must be of the same mode as the dummy value.

```
> X <- matrix(scan("light.dat", 0), ncol=5, byrow=T)
```

There are more elaborate input facilities available and these are detailed in the manual.

7.3 Other facilities; editing data

Once a data set has been read, there is an X–window based facility in S-PLUS for making small changes. The command

```
> xnew <- data.ed(xold)
```

will allow you to edit your data set `xold` using a spreadsheet-like environment in a separate editing window, and on completion the changed object is assigned to `xnew`. `xold`, and hence `xnew`, can be any matrix, vector, data frame, or atomic data object.

8 More language features. Loops and conditional execution

8.1 Grouped expressions

S-PLUS is an expression language in the sense that its only command type is a function or expression which returns a result. Even an assignment is an expression whose result is the value assigned, and it may be used wherever any expression may be used; in particular multiple assignments are possible.

Commands may be grouped together in braces, `{expr`$_1$`; expr`$_2$`;...; expr`$_m$`}`, in which case 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 a part of an even larger expression, and so on.

8.2 Control statements

8.2.1 Conditional execution: `if` statements

The language has available 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.

8.2.2 Repetitive execution: `for` loops, `repeat` and `while`

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

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

where ***name*** is the loop variable. `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. One possibility here is to use `coplot()` to be discussed later, which will produce an array of plots corresponding to each level of the factor. Another way to do this, now putting all plots on the one display, is as follows:

```
> 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 category. This is a useful function, mostly used in connection with boxplots. See the `help` facility for further details.)

WARNING: `for()` loops are very slow in the present version of S-PLUS and should be avoided if possible. Many functions, such as `apply()`, `tapply()`, `sapply()` and others, are written primarily to avoid using explicit `for()` loops.

Other looping facilities include the