

6.2 Constructing and modifying lists

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

```
> St <- list(name1=object1, name2=object2, ..., namem=objectm)
```

sets up a list `St` of m components using `comp1`, ..., `compm` 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

```
> St[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.

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 a 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. This is done most conveniently with the `scan()` function for simple data items, and the `read.table()` function for reading entire data frames directly.

7.1 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.

7.2 The `read.table()` function

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

- The first line of the file should have 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

	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

read as numeric 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

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

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

where the `header=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.

7.3 Other facilities; editing data

S-PLUS input facilities are simple and their requirements are fairly strict. There is a clear presumption that rather than use S-PLUS to accommodate a subtle variety of input, (and more so output), protocols 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 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.

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, `{expr1; expr2;...; exprm}`, 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

The language has available a conditional construction of the form

```
> if (expr1) expr2 else expr3
```

where `expr1` 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 expr1) expr2
```

where *name* is the loop variable. *expr1* is a vector expression, (often a sequence like `1:20`), and *expr2* is often a grouped expression with its sub-expressions written in terms of the dummy *name*. *expr2* is repeatedly evaluated as *name* ranges through the values in the vector result of *expr1*.

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(lsfilt(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.)

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.

Control statements are most often used in connection with *functions* which are discussed in §9, and where more examples will emerge.

9 Writing your own functions

As we have seen informally along the way, the S-PLUS language allows the user to create objects of mode *function*. These are true S-PLUS 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, and learning to write useful functions is one of the main ways to make your use of S-PLUS comfortable and productive.

It should be emphasized that most of the functions supplied as part of the S-PLUS system, such as `mean()`, `var()`, `postscript()` and so on, are themselves written in S-PLUS and thus do not differ materially from user written functions.

A function is defined by an assignment of the form

```
> name <- function(arg1, arg2, ...) expression
```

The *expression* is an S-PLUS 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 usually takes the form `name(expr1, expr2, ...)` and may occur anywhere a function call is legitimate.

For example, consider a function to emulate directly the `MATLAB` backslash command, which returns the coefficients of the orthogonal projection of the vector *y* onto the column space of the matrix, *X*. Thus given a vector $y^{n \times 1}$ and a matrix $X^{n \times p}$ then

$$X \backslash y = \text{def. } (X'X)^{-} X'y$$

where $(X'X)^{-}$ is a generalized inverse of $X'X$.

```
> bsplash <- function(X, y)
{
  X <- qr(X)
  qr.coef(X, y)
}
```

After this object is created it is permanent, like all objects, and may be used in statements such as

```
> regcoeff <- bsplash(Design, yvar)
```

and so on.

The classical S-PLUS function `lsfit()` does this job quite well, and more. It in turn uses the functions `qr()` and `qr.coef()` in the slightly counterintuitive way above to do this part of the calculation. Hence there is probably some value in having just this part isolated in a simple to use function if it is going to be in frequent use. If so, we may wish to make it a matrix binary operator for even more convenient use.

9.1 Defining new binary operators.

Had we given the `bsplash()` function a different name, namely one of the form

`%anything%`

it could have been used as a *binary operator* in expressions rather than in function form. Suppose, for example, we choose `!` for the internal character. The function definition would then start as

```
> "%!%" <- function(X, y) {... }
```

(Note the use of quote marks.) The function could then be used as `X %!% y`. (The backslash symbol itself is not a convenient choice as it presents special problems in this context.)

The matrix multiplication operator, `%*%`, and the outer product matrix operator `%o%` are other examples of binary operators defined in this way.

9.2 Named arguments and defaults. "..."

As first noted in §2.3 if arguments to called functions are given in the "*name=object*" form, they may be given in any order. Furthermore the argument sequence may begin in the unnamed, positional form, and specify named arguments after the positional arguments.

Thus if there is a function `fun1` defined by

```
> fun1 <- function(data, data.frame, graph, limit) {...7}
```

Then the function may be invoked in several ways, for example

```
> ans <- fun1(d, df, 20, T)
```

```
> ans <- fun1(d, df, graph=T, limit=20)
```

```
> ans <- fun1(data=d, limit=20, graph=T, data.frame=df)
```

are all equivalent.

In many cases arguments can be given commonly appropriate default values, in which case they may be omitted altogether from the call when the defaults are appropriate. For example, if `fun1` were defined as

```
> fun1 <- function(data, data.frame, graph=T, limit=20) {...7}
```

it could be called as

```
> ans <- fun1(d, df)
```

which is now equivalent to the three cases above, or as

```
> ans <- fun1(d, df, limit=10)
```

which changes one of the defaults.

It is important to note that defaults may be arbitrary expressions, even involving other arguments to the same function; they are not restricted to be constants as in our simple example here.

⁷This ellipsis is used as a customary typographical device to mean an abridgement. This is not the case in the latter part of section.

Another frequent requirement is to allow one function to pass on argument settings to another. For example many graphics functions use the function `par()` and functions like `plot()` allow the user to pass on graphical parameters to `par()` to control the graphical output. This can be done by including an extra argument, literally "...", of the function, which may then be passed on. An outline example is given in Figure 4. Note here that the ellipses, "..." are literal S-PLUS, not a typographical

```
fun1 <- function(data, data.frame, graph=T, limit=20, ...)
{
  [omitted statements]

  if (graph)
    par(pch="*", ...)

  [more omissions]
}
```

Figure 4: Use of the ellipsis argument, "..."

device.

9.3 Assignments within functions are local. Frames.

Note that *any ordinary assignments done within the function are local and temporary and lost after exit from the function.* Thus the assignment `X <- qr(X)` does not affect the value of the argument in the calling program.

To understand completely the rules governing the scope of S-PLUS assignments the reader needs to be familiar with the notion of an evaluation *frame*. This is a somewhat advanced, though hardly difficult, topic and is not covered further in these notes.

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

9.4 More advanced examples

Efficiency factors in block designs

As a more complete, if a little pedestrian, example of a function, consider finding the efficiency factors for a block design. (Some aspects of this problem have already been discussed in §5.3.)

A block design is defined by two factor, say `blocks` (`b` levels) and `varieties`, (`v` levels). If $R^{v \times v}$ and $K^{b \times b}$ are the *replications* and *block size* matrices, and $N^{b \times v}$ is the incidence matrix, then the efficiency factors are defined as the eigenvalues of the matrix

$$E = I_v - R^{-1/2} N' K^{-1} N R^{-1/2} = I_v - A' A$$

where $A = K^{-1/2} N R^{-1/2}$. One way to write the function is as in Figure 5.

It is numerically slightly better to work with the singular value decomposition on this occasion rather than the eigenvalue routines.

The result of the function is a list giving not only the efficiency factors as the first component, but also the block and variety cononical contrasts, since sometimes these give additional useful qualitative information.

Dropping all names in a printed array

For printing purposes with large matrices or arrays, it is often useful to print them in close block form without the array names or numbers. Removing the `dimnames` attribute will not achieve this effect, but rather the array must be given a `dimnames` attribute consisting of empty strings. For example to print a matrix, `X`

```
> temp <- X
```

```

> bdeff <- function(blocks, varieties) {
  blocks <- as.factor(blocks)           # minor safety move
  b <- length(levels(blocks))
  varieties <- as.factor(varieties)    # minor safety move
  v <- length(levels(varieties))
  K <- as.vector(table(blocks))        # remove dim attr
  R <- as.vector(table(varieties))     # remove dim attr
  N <- table(blocks, varieties)
  A <- 1/sqrt(K) * N * rep(1/sqrt(R), rep(b, v))
  sv <- svd(A)
  list(eff=1 - sv$d^2, blockcv=sv$u, varietycv=sv$v)
}

```

Figure 5: A function for block design efficiencies

```

> dimnames(temp) <- list(rep("", nrow(X)), rep("", ncol(X)))
> temp; rm(temp)

```

This can be much more conveniently done using a function, `no.dimnames()`, shown in Figure 6, as a “wrap around” to achieve the same result. It also illustrates how some effective and useful user functions can be quite short. With this function

```

no.dimnames <- function(a){
#
# Remove all dimension names from an array for compact printing.
#
  d <- list()
  l <- 0
  for(i in dim(a)) {
    d[[l <- l + 1]] <- rep("", i)
  }
  dimnames(a) <- d
  a
}

```

Figure 6: A function for printing arrays in compact form

defined, an array may be printed in close format using

```
> no.dimnames(X)
```

This is particularly useful for large integer arrays, where patterns are the real interest rather than the values.

Recursive numerical integration

Functions may be recursive, and may themselves define functions within themselves. Note, however, that such functions, or indeed variables, are not inherited by called functions in higher evaluation frames as they would be if they were on the search list.

The example in Figure 7 shows a naive way of performing one dimensional numerical integration. The integrand is evaluated at the end points of the range and in the middle. If the one-panel trapezium rule answer is close enough to the two panel, then the latter is returned as the value. Otherwise the same process is recursively applied to each panel. The result is an adaptive integration process that concentrates function evaluations in regions where the integrand is furthest from linear. There is, however, a heavy overhead, and the function is only competitive with other algorithms when the integrand is both smooth and very difficult to evaluate.

The example is also given partly as a little puzzle in S-PLUS programming.

```

area <- function(f, a, b, eps = 1.0e-06, lim = 10)
{
  fun1 <- function(f, a, b, fa, fb, a0, eps, lim, fun)
  {
    d <- (a + b)/2
    h <- (b - a)/4
    fd <- f(d)
    a1 <- h * (fa + fd)
    a2 <- h * (fd + fb)
    if(abs(a0 - a1 - a2) < eps || lim == 0)
      return(a1 + a2)
    else {
      return(fun(f, a, d, fa, fd, a1, eps, lim - 1, fun) +
             fun(f, d, b, fd, fb, a2, eps, lim - 1, fun))
    }
  }
  fa <- f(a)
  fb <- f(b)
  a0 <- ((fa + fb) * (b - a))/2
  fun1(f, a, b, fa, fb, a0, eps, lim, fun1)
}

```

Figure 7: A recursive function within a function

9.5 Customizing the environment. .First and .Last

Any function named `.First()` in the `.Data` directory has a special status. It is automatically performed at the beginning of an S-PLUS session and may be used to initialize the environment. For example, the definition in Figure 8 alters the prompt to `$` and sets up various other useful things that can then be taken for granted in the rest of the session. Similarly a function

```

> .First <- function() {
  options(prompt="$ ", continue="+\t")      # $ is the prompt
  options(digits=5, length=999)           # custom numbers and printout
  options(gui="motif")                    # default graphics user interface
  tek4014()                               # for terminal work
  par(pch = "+")                          # plotting character
  attach(paste(unix("echo $HOME"), "/.Data", sep = ""))
                                           # Home of my personal library
  library(examples)                       # attach also the system examples
}

```

Figure 8: An example of a `.First()` function

`.Last()`, if defined, is executed at the very end of the session. An example is given in Figure 9

```

> .Last <- function() {
  graphics.off()                          # a small safety measure.
  cat(paste(unix("date"), "\nAdios\n"))    # Is it time for lunch?
}

```

Figure 9: An example of a `.Last()` function

9.6 Classes, generic functions and object orientation

The class of an object determines how it will be treated by what are known as *generic* functions. Put the other way round, a generic function performs a task or action on its arguments *specific to the class of the argument itself*. If the argument lacks any class attribute, or has a class not catered for specifically by the generic function in question, there is always a *default action* provided.

An example makes things clearer. In a sense the `print()` function has always been generic, since its action is to adopt a style of output appropriate to its arguments. Thus a matrix appears as a matrix, a vector as a vector, and so on. (Note that the `print()` function can be called explicitly, or implicitly by giving an expression as a complete command.)

The August 1991 release of S-PLUS increases the number of such functions, alters the mechanism by which they are implemented and via the class mechanism offers the user the facility of designing and writing generic functions for special purposes. Among the other new, or newly generic functions are `plot()` for displaying objects graphically, `summary()` for summarising analyses of various types, and `anova()` for comparing statistical models.

The number of generic functions that can treat a class in a specific way can be quite large. For example, the functions that can accommodate in some fashion objects of class `data.frame` include

```
[,      [[<-,  dbdetach,  dimnames<-,  pairs,  signif,
[<-,  aperm,  dim,      formula,  plot,  summary,
[[,   atan,  dimnames,  ordered<-,  print,  t,
```

A currently complete list can be got by using the `methods()` function:

```
> methods(class="data.frame")
```

Conversely the number of classes a generic function can handle can also be quite large. For example the `plot()` function has variants for classes of object

```
data.frame,  default,  glm,  pregam,  surv.fit,
design,      factor,  lm,  preloess,  tree,
formula,    gam,     loess,  profile,  tree.sequence,
```

and perhaps more. A complete list can be got again by using the `methods()` function:

```
> methods(plot)
```

The reader is referred to the official references for a complete discussion of this mechanism.

10 Statistical models in S-PLUS

This section presumes the reader has some familiarity with statistical methodology, in particular with regression analysis and the analysis of variance. Later we make some rather more ambitious presumptions, namely that something is known about generalized linear models and nonlinear regression.

The requirements for fitting statistical models are sufficiently well defined to make it possible to construct general tools that apply in a broad spectrum of problems. Since the August 1991 release S-PLUS provides an interlocking suite of facilities that make fitting statistical models very simple. However these are not at the same high level as those in, say, Genstat, especially in the form of the output which in keeping with general S-PLUS policy is rather minimal.

10.1 Defining statistical models; formulæ

The template for a statistical model is a linear regression model with independent, homoscedastic errors

$$y_i = \sum_{j=0}^p \beta_j x_{ij} + e_i, \quad e_i \sim \text{NID}(0, \sigma^2), \quad i = 1, 2, \dots, n$$

In matrix terms this would be written

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \mathbf{e}$$

where the \mathbf{y} is the response vector, \mathbf{X} is the *model matrix* or *design matrix* and has columns $\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_p$, the determining variables. Very often \mathbf{x}_0 will be a column of 1s defining an *intercept* term.

Examples.

Before giving a formal specification, a few examples may usefully set the picture.

Suppose $y, x, x_0, x_1, x_2, \dots$ are numeric variables, \mathbf{X} is a matrix and A, B, C, \dots are factors. The following formulæ on the left side below specify statistical models as described on the right.

$y \sim x$ $y \sim 1 + x$	Both imply the same simple linear regression model of y on x . The first has an implicit intercept term, and the second an explicit one.
------------------------------	--

$y \sim -1 + x$ $y \sim x - 1$	Simple linear regression of y on x through the origin, (that is, without an intercept term).
-----------------------------------	--

$\log(y) \sim x_1 + x_2$	Multiple regression of the transformed variable, $\log(y)$, on x_1 and x_2 (with an implicit intercept term).
--------------------------	--

$y \sim \text{poly}(x, 2)$ $y \sim 1 + x + I(x^2)$	Polynomial regression of y on x of degree 2. The first form uses orthogonal polynomials, and the second uses explicit powers, as basis.
---	---

$y \sim \mathbf{X} + \text{poly}(x, 2)$	Multiple regression y with model matrix consisting of the matrix \mathbf{X} as well as polynomial terms in x to degree 2.
---	---

$y \sim A$	Single classification analysis of variance model of y , with classes determined by A .
------------	--

$y \sim A + x$	Single classification analysis of covariance model of y , with classes determined by A , and with covariate x .
----------------	---

$y \sim A*B$ $y \sim A + B + A:B$ $y \sim B \%in\% A$ $y \sim A/B$	Two factor non-additive model of y on A and B . The first two specify the same crossed classification and the second two specify the same nested classification. In abstract terms all four specify the same model subspace.
---	--

$y \sim (A + B + C)^2$ $y \sim A*B*C - A:B:C$	Three factor experiment but with a model containing main effects and two factor interactions only. Both formulæ specify the same model.
--	---

$y \sim A * x$ $y \sim A/x$ $y \sim A/(1 + x) - 1$	Separate simple linear regression models of y on x within the levels of A , with different codings. The last form produces explicit estimates of as many different intercepts and slopes as there are levels in A .
--	---

$y \sim A*B + \text{Error}(C)$	An experiment with two treatment factors, A and B , and error strata determined by factor C . For example a split plot experiment, with whole plots, (and hence also subplots), determined by factor C .
--------------------------------	--

The operator \sim is used to define a *model formula* in S-PLUS. The form, for an ordinary linear model, is

$$\text{response} \sim [\pm] \text{term}_1 \pm \text{term}_2 \pm \text{term}_3 \pm \dots$$

response is a vector or matrix, (or expression evaluating to a vector or matrix) defining the response variable(s).

\pm is an operator, either + or -, implying the inclusion or exclusion of a term in the model, (the first is optional).

term is either

- a vector or matrix expression, or 1,
- a factor, or
- a *formula expression* consisting of factors, vectors or matrices connected by *formula operators*.

In all cases each term defines a collection of columns either to be added to or removed from the model matrix. A 1 stands for an intercept column and is by default included in the model matrix unless explicitly removed.

Form	Meaning
$Y \sim M$	Y is modelled as M
$M_1 + M_2$	Include M_1 and M_2
$M_1 - M_2$	Include M_1 leaving out terms of M_2
$M_1 : M_2$	The tensor product of M_1 and M_2 . If both terms factors, then the “subclasses” factor.
$M_1 \%in\% M_2$	Similar to $M_1 : M_2$, but with a different coding.
$M_1 * M_2$	$M_1 + M_2 + M_1 : M_2$
M_1 / M_2	$M_1 + M_2 \%in\% M_1$
$M \sim n$	All terms in M together with “interactions” up to order n
$I(M)$	Insulate M . Inside M all operators have their normal arithmetic meaning, and that term appears in the model matrix.

Table 1: Summary of model operator semantics

The *formula operators* are similar in effect to the Wilkinson and Rogers notation used by such programs as Glim and Genstat. One inevitable change is that the operator “.” becomes “:” since the period is a valid name character in S-PLUS. The notation is summarised as in the Table 1 (based on Chambers & Hastie, p. 29).

Note that inside the parentheses that usually enclose function arguments all operators have their normal arithmetic meaning. The function $I()$ is an identity function used only to allow terms in model formulæ to be defined using arithmetic operators.

Note particularly that the model formulæ specify the *columns of the model matrix*, specification of the parameters is implicit. This is not the case in other contexts, for example in fitting nonlinear models

10.2 Regression models; fitted model objects

The basic function for fitting ordinary multiple models is $lm()$, and a streamlined version of the call is as follows:

```
> fitted.model <- lm(formula, data=data.frame)
```

For example

```
> fm2 <- lm(y ~ x1 + x2, data=production)
```

would fit a multiple regression model of y on x_1 and x_2 (with implicit intercept term).

The important but technically optional parameter `data=production` specifies that any variables needed to construct the model should come first from the *production data frame*. *This is the case regardless of whether data frame production has been attached to the search list or not.*

10.3 Generic functions for extracting information

The value of $lm()$ is fitted model object; technically a list of results of class `lm`. Information about the fitted model can then be displayed, extracted, plotted and so on by using generic functions that orient themselves to objects of class `lm`. A full list of these at the present time is

```
add1   coef       effects  kappa   predict  residuals
alias  deviance    family   labels  print    summary
anova  drop1       formula  plot    proj
```

A brief description of the most commonly used ones is given in Table 2.

Function	Value or Effect
<code>anova(object₁, object₂)</code>	Compare a submodel with an outer model and produce an analysis of variance table.
<code>coefficients(object)</code>	Extract the regression coefficient (matrix). Short form: <code>coef(object)</code> .
<code>deviance(object)</code>	Residual sum of squares, weighted if appropriate.
<code>formula(object)</code>	Extract the model formula.
<code>plot(object)</code>	Produce two plots, one of the observations against the fitted values, the other of the absolute residuals against the fitted values.
<code>predict(object, newdata=data.frame)</code> <code>predict.gam(object, newdata=data.frame)</code>	The data frame supplied must have variables specified with the same labels as the original. The value is a vector or matrix of predicted values corresponding to the determining variable values in <i>data.frame</i> . <code>predict.gam()</code> is a safe alternative to <code>predict()</code> that can be used for <code>lm</code> , <code>glm</code> and <code>gam</code> fitted objects. It must be used, for example, in cases where orthogonal polynomials are used as the original basis functions, and the addition of new data implies different basis functions to the original.
<code>print(object)</code>	Print a concise version of the object. Most often used implicitly.
<code>residuals(object)</code>	Extract the (matrix of) residuals, weighted as appropriate. Short form: <code>resid(object)</code> .
<code>summary(object)</code>	Print a comprehensive summary of the results of the regression analysis.

Table 2: Commonly used generic functions on class `lm` objects

10.4 Analysis of variance; comparing models

The model fitting function `aov(formula, data=data.frame)` operates at the simplest level in a very similar way to the function `lm()`, and most of the generic functions listed in Table 2 apply.

It should be noted that in addition `aov()` allows an analysis of models with multiple error strata such as split plot experiments, or balanced incomplete block designs with recovery of inter-block information envisaged. Model formula

$$\text{response} \sim \text{mean.formula} + \text{Error}(\text{strata.formula})$$

Specifies a multi-stratum experiment with error strata defined by the *strata.formula*. In the simplest case, *strata.formula* is simply a factor, when it defines a two strata experiment, namely between and within the levels of the factor.

For example, with all determining variables factors a model formula such as that in:

```
> fm <- aov(yield ~ v + n*p*k + Error(farms/blocks), data=farm.data)
```

would typically be used to describe an experiment with mean model `v + n*p*k` and three error strata, namely “between farms”, “within farms, between blocks” and “within blocks”.

10.4.1 ANOVA tables

Note also that the analysis of variance table (or tables) are for a sequence of fitted models. The sums of squares shown are the decrease in the residual sums of squares resulting from an inclusion of *that term* in the model at *that place* in the sequence. Hence only for orthogonal experiments will the order of inclusion be inconsequential.

For multistratum experiments the procedure is first to project the response onto the error strata, again in sequence, and to fit the mean model to each projection. For further details, see Chambers and Hastie, §5.

A more flexible alternative to the default full ANOVA table is to compare two or more models directly using the `anova()` function.

```
> anova(fitted.model.1, fitted.model.2, ...)
```

The display is then an ANOVA table showing the differences between the fitted models when fitted in sequence. The fitted models being compared would usually be an hierarchical sequence, of course. This does not give different information to the default, but rather makes it easier to comprehend and control.

10.5 Updating fitted models. The ditto name “.”

The `update()` function is largely a convenience function that allows a model to be fitted that differs from one previously fitted usually by just a few additional or removed terms. Its form is

```
> new.model <- update(old.model, new.formula)
```

In the *new.formula* the special name consisting of a period, “.”, only, can be used to stand for “the corresponding part of the old model formula”. For example

```
> fm05 <- lm(y ~ x1 + x2 + x3 + x4 + x5, data=production)
> fm6 <- update(fm05, . ~ . + x6)
> smf6 <- update(fm6, sqrt(.) ~ .)
```

would fit a five variate multiple regression with variables (presumably) from the data frame `production`, fit an additional model including a sixth regressor variable, and fit a variant on the model where the response had a square root transform applied.

Note especially that if the `data=` argument is specified on the original call to the model fitting function, this information is passed on through the fitted model object to `update()` and its allies.

The name “.” can also be used in other contexts, but with slightly different meaning. For example

```
> fmfull <- lm(y ~ . , data=production)
```

would fit a model with response `y` and regressor variables *all other variables in the data frame production*.

Other functions for exploring incremental sequences of models are `add1()`, `drop1()`, `step()` and `stepwise()`. The names of these give a good clue to their purpose, but for full details see the help document.

10.6 Generalized linear models; families

Generalized linear modelling is a development of linear models to accommodate both non-normal response distributions and transformations to linearity in a clean and straightforward way. A generalized linear model may be described in terms of the following sequence of assumptions:

- There is a response, y , of interest and stimulus variables x_1, x_2, \dots whose values influence the distribution of the response.
- The stimulus variables influence the distribution of y through *a single linear function, only*. This linear function is called the *linear predictor*, and is usually written

$$\eta = \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_p x_p$$

hence x_i has no influence on the distribution of y if and only if $\beta_i = 0$.

- The distribution of y is of the form

$$f_Y(y; \mu, \varphi) = \exp \left[\frac{A}{\varphi} \{y\lambda(\mu) - \gamma(\lambda(\mu))\} + \tau(y, \varphi) \right]$$

where φ is a *scale parameter*, (possibly known), and is constant for all observations, A represents a prior weight, assumed known but possibly varying with the observations, and μ is the mean of y . So it is assumed that the distribution of y is determined by its mean and possibly a scale parameter as well.

- The mean, μ , is a smooth invertible function of the linear predictor:

$$\mu = m(\eta), \quad \eta = m^{-1}(\mu) = \ell(\mu)$$

and this inverse function, $\ell(\cdot)$ is called the *link function*.

These assumptions are loose enough to encompass a wide class of models useful in statistical practice, but tight enough to allow the development of a unified methodology of estimation and inference, at least approximately. The reader is referred to any of the current reference works on the subject for full details, such as

Generalized linear models by Peter McCullagh and John A Nelder, 2nd edition, Chapman and Hall, 1989, or

An introduction to generalized linear models by Annette J Dobson, Chapman and Hall, 1990.

10.6.1 Families

The class of generalized linear models handled by facilities supplied in S-PLUS includes *gaussian*, *binomial*, *poisson*, *inverse gaussian* and *gamma* response distributions and also *quasi-likelihood* models where the response distribution is not explicitly specified. In the latter case the *variance function* must be specified as a function of the mean, but in other cases this function is implied by the response distribution.

Each response distribution admits a variety of link functions to connect the mean with the linear predictor. Those automatically available are as in Table 3.

Link Function	Family Name					
	binomial	gaussian	Gamma	inverse.gaussian	poisson	quasi
logit	*					*
probit	*					*
cloglog	*					*
identity		*	*		*	*
inverse			*			*
log			*		*	*
1/mu^2				*		*
sqrt					*	*

Table 3: Families and the link functions available to them

The combination of a response distribution, a link function and various other pieces of information that are needed to carry out the modelling exercise is called the *family* of the generalized linear model.

10.6.2 The glm() function

Since the distribution of the response depends on the stimulus variables through a single linear function *only*, the same mechanism as was used for linear models can still be used to specify the linear part of a generalized model. The family has to be specified in a different way.

The S-PLUS function to fit a generalized linear model is `glm()` which uses the form

```
> fitted.model <- glm(formula, family =family.generator, data=data.frame)
```

The only new feature is the *family.generator*, which is the way the family is described. Although it seems a little complicated at first sight, (it is the name of a function that generates a list of functions and expressions that together define and control the model and estimation process), its use is quite simple.

The names of the standard, supplied family generators are given under "Family Name" in Table 3. Where there is a choice of links, the name of the link may also be supplied with the family name, in parentheses as a parameter. In the case of the `quasi` family, the variance function may also be specified in this way.

Some examples make the process clear.

The gaussian family

A call such as

```
> fm <- glm(y ~ x1+x2, family=gaussian, data=sales)
```

achieves the same result as

```
> fm <- lm(y ~ x1+x2, data=sales)
```

but much less efficiently. Note how the gaussian family is not automatically provided with a choice of links, so no parameter is allowed. If a problem requires a gaussian family with a nonstandard link, this can usually be achieved through the quasi family, as we shall see later.

The binomial family

Consider a small, artificial example.

On the Greek island of Kalythos the male inhabitants suffer from a congenital eye disease, the effects of which become more marked with increasing age. Samples of islander males of various ages were tested for blindness and the results recorded. The data is shown in Table 4.

Age:	20	35	45	55	70
No. tested:	50	50	50	50	50
No. blind:	6	17	26	37	44

Table 4: The Kalythos blindness data

The problem we consider is to fit both logistic and probit models to this data, and to estimate for each model the LD50, that is the age at which the chance of blindness for a male inhabitant is 50%.

If y is the number of blind at age x and n the number tested, both models have the form

$$y \sim B(n, F(\beta_0 + \beta_1 x))$$

where for the probit case, $F(z) = \Phi(z)$ is the standard normal distribution function, and in the logit case, (the default), $F(z) = e^z / (1 + e^z)$. In both cases the LD50 is

$$\text{LD50} = -\beta_0 / \beta_1$$

that is, the point at which the argument of the distribution function is zero.

The first step is to set the data up as a data frame

```
> kalythos <- data.frame(x=c(20,35,45,55,70), n=rep(50,5),
  y=c(6,17,26,37,44))
```

To fit a binomial model using `glm()` there are two possibilities for the response:

- If the response is a *vector* it is assumed to hold *binary* data, and so must be a 0, 1 vector.
- If the response is a *two column matrix* it is assumed that the first column holds the number of successes for the trial and the second holds the number of failures.

Here we need the second of these conventions, so we add a matrix to our data frame:

```
> kalythos$Ymat <- cbind(kalythos$y, kalythos$n - kalythos$y)
```

To fit the models we use

```
> fmp <- glm(Ymat ~ x, family=binomial(link=probit), data=kalythos)
```