```
> repeat expr
```

statement and the

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

statement.

The `break` statement can be used to terminate any loop, possibly abnormally. For `repeat` loops, this is the only way to terminate them.

The `next` can be used to discontinue one particular cycle and skip to the “next”.

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(`***arg***$_1$`, `***arg***$_2$`, ...)` ***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(`***expr***$_1$`, `***expr***$_2$`, ...)` and may occur anywhere a function call is legitimate.

9.1 Simple examples

As a first example, consider a function to calculate the two sample $t-$statistic, showing "all the steps". This is an artificial example, of course, since there are other, simpler ways of achieving the same end.

The function is defined as follows:

```
> twosam <- function(y1, y2) {
    n1  <- length(y1); n2  <- length(y2)
    yb1 <- mean(y1);   yb2 <- mean(y2)
    s1  <- var(y1);    s2  <- var(y2)
    s <- ((n1-1)*s1 + (n2-1)*s2)/(n1+n2-2)
    tst <- (yb1 - yb2)/sqrt(s2*(1/n1 + 1/n2))
    tst
  }
```

With this function defined, you could perform two sample $t-$tests using a call such as

```
> tstat <- twosam(datamale, datafemale); tstat
```

As a second 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. (This is ordinarily called the least squares estimates of the regression coefficients). This would ordinarily be done with the `qr()` function; however this is sometimes a bit tricky to use directly and it pays to have a simple function such as the following to use it safely.

Thus given a vector $y^{n\times 1}$ and a matrix $X^{n\times p}$ then

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

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

```
> bslash <- 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 <- bslash(Xmat, 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.

9.2 Defining new binary operators.

Had we given the `bslash()` 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.

9.3 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) {...[8]}
```

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

[8]This ellipsis is used as a customary typographical device to mean an abridgment. This is not the case in the latter part of this section.

```
> fun1 <- function(data, data.frame, graph=T, limit=20) {...8}
```

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.

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. (See §11.4.1 for more details on the `par()` function.) 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.

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

          [omitted statements]

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

          [more omissions]
      }
```

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

Note here that the ellipses, "`...`" are literal S-PLUS, not a typographical device.

9.4 Assignments within functions are local. Frames.

Note that ***any ordinary assignments done within the function are local and temporary and are 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.

9.5 More advanced examples

9.5.1 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 factors, 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.

```
> 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

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 canonical contrasts, since sometimes these give additional useful qualitative information.

9.5.2 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
```

```
> 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 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.

```
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

9.5.3 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 farthest 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.

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

9.7 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()`

```
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

```
> .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

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,
```

```
> .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

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

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

In matrix terms this would be written

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

where the $\boldsymbol{y}$ is the response vector, $\mathbf{X}$ is the ***model matrix*** or ***design matrix*** and has columns $\boldsymbol{x}_0$, $\boldsymbol{x}_1$, ..., $\boldsymbol{x}_p$, the determining variables. Very often $\boldsymbol{x}_0$ 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`, `x0`, `x1`, `x2`, ... are numeric variables, `X is a matrix` and `A`, `B`, `C`, ... are factors. The following formulæ on the left side below specify statistical models as described on the right.

`y ~ x` `y ~ 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 ~ -1 + x` `y ~ x - 1`	Simple linear regression of y on x through the origin, (that is, without an intercept term).
`log(y) ~ x1 + x2`	Multiple regression of the transformed variable, $\log(y)$, on x_1 and x_2 (with an implicit intercept term).
`y ~ poly(x,2)` `y ~ 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 ~ X + poly(x,2)`	Multiple regression y with model matrix consisting of the matrix X as well as polynomial terms in x to degree 2.
`y ~ A`	Single classification analysis of variance model of y, with classes determined by A.
`y ~ A + x`	Single classification analysis of covariance model of y, with classes determined by A, and with covariate x.

`y ~ A*B`
`y ~ A + B + A:B`
`y ~ B %in% A`
`y ~ 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 ~ (A + B + C)^2` `y ~ 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 ~ A * x`
`y ~ A/x`
`y ~ 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 ~ A*B + 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 `~` is used to define a ***model formula*** in S-PLUS. The form, for an ordinary linear model, is

$$\textbf{\textit{response}} \sim [\pm]\ \textbf{\textit{term}}_1 \pm \textbf{\textit{term}}_2 \pm \textbf{\textit{term}}_3 \pm \cdots$$

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.

The ***formula operators*** are similar in effect to the Wilkinson and Rogers notation used by such programs a 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

Form	Meaning
Y ~ 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^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

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
`anova(`***object***$_1$`,` ***object***$_2$`)`	Compare a submodel with an outer model and product an analysis of variance table.
`coefficients(`***object***`)`	Extract the regression coefficient (matrix). Short form: `coef(`***object***`)`.
`deviance(`***object***`)`	Residual sum of squares, weighted if appropriate.
`formula(`***object***`)`	Extract the model formula.
`plot(`***object***`)`	Product two plots, one of the observations against the fitted values, the other of the absolute residuals against the fitted values.

`predict(`***object***`,`
`newdata=`***data.frame***`)`
`predict.gam(`***object***`,`
`newdata=`***data.frame***`)`
	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 ***data.frame***.
	`predict.gam()` is a safe alternative to `predict()` that can be used for `lm`, `glm` and `gam` 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.

`print(`***object***`)`	Print a concise version of the object. Most often used implicitly.
`residuals(`***object***`)`	Extract the (matrix of) residuals, weighted as appropriate. Short form: `resid(`***object***`)`.
`summary(`***object***`)`	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. The model formula

response `~` ***mean.formula*** `+ Error(`***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, ... 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

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

  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

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

  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:

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

  and this inverse function, $\ell(.)$ 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.

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.

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

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 instrument by which the family is described. 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. Although this may seem a little complicated at first sight, 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

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

where for the probit case, $F(z) = (z)$ 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

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

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

Since the logit link is the default the parameter may be omitted on the second call. To see the results of each fit we could use

```
> summary(fmp)
> summary(fml)
```

Both models fit (all too) well. To find the LD50 estimate we can use a simple function:

```
> ld50 <- function(b) -b[1]/b[2]
> ldp <- ld50(coef(fmp)); ldl <- ld50(coef(fmp)); c(ldp, ldl)
```

The actual estimates from this data are 43.663 years and 43.601 years respectively.

Poisson models

With the poisson family the default link is the `log`, and in practice the major use of this family is to fit surrogate poisson log-linear models to frequency data, whose actual distribution is often multinomial. This is a large and important subject we will not discuss further here. It even forms a major part of the use of non-gaussian generalized models overall.

Occasionally genuinely poisson data arises in practice and in the past it was often analysed as gaussian data after either a log or a square-root transformation. As a graceful alternative to the latter, a poisson generalized linear model may be fitted as in the following example:

```
> fmod <- glm(y ~ A+B + x, family=poisson(link=sqrt), data=worm.counts)
```

Quasi-likelihood models

For all families the variance of the response will depend on the mean and will have the scale parameter as a multiplier. The form of dependence of the variance on the mean is a characteristic of the response distribution; for example for the poisson distribution Var$[y] = \mu$.

For quasi-likelihood estimation and inference the precise response distribution is not specified, but rather only a link function and the form of the variance function as it depends on the mean. Since quasi-likelihood estimation uses formally identical techniques to those for the gaussian distribution, this family provides a way of fitting gaussian models with non-standard link functions or variance functions, incidently.

For example, consider fitting the non-linear regression

$$y = \frac{\theta_1 z_1}{z_2 - \theta_2} + e \tag{1}$$

this may be written alternatively as

$$y = \frac{1}{\beta_1 x_1 + \beta_2 x_2} + e$$

where $x_1 = z_2/z_1$, $x_2 = -1/x_1$, $\beta_1 = 1/\theta_1$ and $\beta_2 = \theta_2/\theta_1$. Supposing a suitable data frame to be set up we could fit this non-linear regression as

```
> nlfit <- glm(y~x1+x2-1,family=
     quasi(link=inverse,variance=constant), data=biochem)
```

The reader is referred to the manual and the help document for further information, as needed.

10.7 Nonlinear regression models; parametrized data frames

S-PLUS provides two functions to fit nonlinear models that do not conform even to the partially linear paradigm of generalized linear models. These are `ms()` for arbitrary minimization problems where the objective functions is a sum of similar terms, and `nls()` for conventional nonlinear least squares estimation of normal nonlinear regression models.

In this brief introduction we only consider the nonlinear regression function `nls()` and leave `ms()` for the reader to pursue as needed.

10.7.1 Changes to the form of the model formula

In specifying a linear, or generalized linear model we could allow the regression parameters to be defined implicitly, and to be given names by transference from the column of the model matrix that they multiply.

In arbitrary nonlinear models no such simplicity applies and we have to specify the model as an ordinary expression that includes both determining variables and parameters together. For example to specify a model for a nonlinear regression such as 1 above, we would use

```
y ~ t1*x1/(x2 - t2)
```

where `y` is the response variable, `x1` and `x2` are determining variables and `t1` and `t2` are scalar parameters.

In such model formulæ all operators have their usual arithmetic expression meaning, and the useful facility of expanding factors and forming cross and nested structures is no longer available. All parameters must be explicitly defined in the formula, even if they come from a linear part of the model.

10.7.2 Specifying the parameters

Since the model formula now contains both determining variables and parameters, there has to be some mechanism for specifying which are which. But of course once the parameters have been specified the remaining variates in the model formula must be variables.

As well as specifying which are the parameters, it is also necessary in this case to specify an initial approximation for each with which to start the iterative estimation procedure.

There are two ways of specifying this information:

- If the call to `nls()` has a `start=` parameter specified, its value must be a list of named components. The names of the list specify the names of the parameters and the values specify the starting values.
- If the data is held in a data frame, the parameters may similarly be defined as a ***parameters attribute*** of the data frame.

Since our policy is generally to work with data frames as much as follows, we show the second possibility in the next example.

Example

Consider again a nonlinear regression of the form 1. An easy way to find initial estimates for the parameters is to regress $x_2 y$ on x_1 and x_2:

```
> fm0 <- lm(x2*y ~ x1 + x2 - 1, data=biochem)
> th <- coef(fm0)
```

To name the parameters and associate them with the `biochem` is done as follows:

```
> parameters(biochem) <- list(t1=th[1], t2=th[2])
```

Now to fit the nonlinear regression model:

```
> fm <- nls(y ~ t1*x1/(x2 - t2), data=biochem)
```

At this point we could use the `summary()` function and most of the other generics to investigate the model and display information. To extract the coefficients we could now use, for example

```
> th <- coef(fm)
```

and to make these least squares estimates the new values of the parameters associated with `biochem` we could simply repeat the step

```
> parameters(biochem) <- list(t1=th[1], t2=th[2])
```

Note that the function `parameters()` may either be used as an expression, in which case it extracts the list of parameters from a data frame, or it may be used as the target for an assignment, in which case it accepts a parameter list for a specified data frame. In this respect it is very similar to the `attributes()` function. There is also a function `param()` analogous to `attr()`, which handles one parameter at a time under a character string name.

10.8 Some non-standard models

We conclude this section with just a brief mention of some of the other facilities available in S-PLUS for special regression and data analysis problems.

Local approximating regressions. The `loess()` function fits a nonparametric regression by using a locally weighted regression. Such regressions are useful for highlighting a trend in messy data or for data reduction to give some insight into a large data set.

Robust regression There are several functions available for fitting regression models in a way resistant to the influence of extreme outliers in the data. The most sophisticated of these is `rreg()`, but others include `lmsfit()` for least median squares regression and `l1fit()` for regression using the L_1-norm. However these do not as yet have the facility of using formulæ to specify the model function, for example, and conform to an older protocol, which makes them sometimes rather tedious to use. There is also a `robust()` facility to change a `glm` family object into a robust version for use with the `glm()` model fitting function.

Generalized additive models. This technique aims to construct a regression function from smooth additive functions of the determining variables, usually one for each determining variable. The function `gam()` is in many ways similar to the other model fitting functions outlined above. In addition there are other model fitting functions that do a similar job. These include `avas()` and `ace()`. On the other hand `ppreg()` is available for projection pursuit regression, but this technique is still very much in need of a complete theoretical treatment and further practical experience. These latter functions are again conforming to an older protocol for model fitting functions and lack the convenience of the newer functions.

Tree based models Rather than seek an explicit global linear model for prediction or interpretation, tree based models seek to bifurcate the data, recursively, at critical points of the determining variables in order to partition the data ultimately into groups that are as homogeneous as possible within, and as heterogeneous as possible between. The results often lead to insights that other data analysis methods tend not to yield.

Models are again specified in the ordinary linear model form. The model fitting function is `tree()`, but many other generic functions such as `plot()` and `text()` are well adapted to displaying the results of a tree-based model fit in a graphical way.

11 Graphical procedures

The graphical facilities are an important and extremely versatile component of the S-PLUS environment. Best results are obtained when S-PLUS is used with a high quality graphics system such as X–windows, although even a simple ASCII terminal can be quite effective for some purposes.

Before the graphical facilities of S-PLUS may be used, it is necessary to inform S-PLUS what type of device is being used by starting a *device driver.* In an X–windows environment, the command to do this may be

```
> X11()
```

(which creates a separate window in which high-quality graphical output will appear,) or for a simpler graphics terminal the command

```
> tek4014()
```

may be appropriate.

Once a device driver is running S-PLUS plotting commands can be used to construct and display graphical objects. Plotting commands are divided into three basic groups:

High-level plotting functions create a new plot on the graphics device, possibly with axes, labels, titles and so on.

Low-level plotting functions add more information to an existing plot, such as extra points, lines and labels.

Interactive graphics functions allow you interactively add information to, or extract information from, an existing plot, using a pointing device such as a mouse.

Furthermore, S-PLUS maintains a list of *graphical parameters* which allow you to customise your plots.

11.1 High-level plotting commands

High-level plotting functions are designed to generate a complete plot of the data passed as arguments to the function. Where appropriate, axes, labels and titles are automatically generated (unless you request otherwise.) High-level plotting commands always start a new plot, erasing the current plot if necessary.

11.1.1 The `plot()` function

One of the most frequently used plotting functions in S-PLUS is the `plot()` function. This is a *generic* function: the type of plot produced is dependent on the type or *class* of the first argument.

`plot(x,y)` `plot(xy)`	If `x` and `y` are vectors, `plot(x,y)` produces a scatterplot of `x` against `y`. The same effect can be produced by supplying one argument (second form) as either a list containing two elements `x` and `y` or a two-column matrix.
`plot(x)`	Produces a time series plot if `x` is a numeric vector or time series object, or an Argand diagram if `x` is a complex vector.

`plot(f)` `plot(f,y)`	`f` is a factor object, `y` is a numeric vector. The first form generates a bar plot of `f`; the second form produces boxplots of `y` for each level of `f`.

`plot(df)`
`plot(~ expr)`
`plot(y ~ expr)`
	`df` is a data frame, `y` is any object, `expr` is a list of object names separated by '+' (e.g. `a + b + c`). The first two forms produce distributional plots of the variables in a data frame (first form) or of a number of named objects (second form). The third form plots `y` against every object named in `expr`.

11.1.2 Displaying multivariate data

S-PLUS provides two very useful functions for representing multivariate data. If `X` is a numeric matrix or data frame, the command

```
> pairs(X)
```

produces a pairwise scatterplot matrix of the variables defined by the columns of `X`, that is, every column of `X` is plotted against every other column of `X` and the resulting $n(n-1)$ plots are arranged in a matrix with plot scales constant over the rows and columns of the matrix.

When three or four variables are involved a *coplot* may be more enlightening. If `a` and `b` are numeric vectors and `c` is a numeric vector or factor object (all of the same length), then the command

```
> coplot(a ~ b | c)
```

produces a number of scatterplots of `a` against `b` for given values of `c`. If `c` is a factor, this simply means that `a` is plotted against `b` for every level of `c`. When `c` is numeric, it is divided into a number of *conditioning intervals* and for each interval `a` is plotted against `b` for values of `c` within the interval. The number and position of intervals can be controlled with `given.values=` argument to `coplot()` — the function `co.intervals()` is useful for selecting intervals. You can also use two ***given*** variables with a command like

```
> coplot(a ~ b | c + d)
```

which produces scatterplots of `a` against `b` for every joint conditioning interval of `c` and `d`.

The `coplot()` and `pairs()` function both take an argument `panel=` which can be used to customise the type of plot which appears in each panel. The default is `points()` to produce a scatterplot but by supplying some other low-level graphics function of two vectors `x` and `y` as the value of `panel=` you can produce any type of plot you wish. An example panel function useful for coplots is `panel.smooth()`.

11.1.3 Display graphics

Other high-level graphics functions produce different types of plots. Some examples are:

`tsplot(x1,x2,...)`	Plots any number of time series on the same scale. This automatic simultaneous scaling feature is also useful when the x_i's are ordinary numeric vectors, in which case they are plotted against the numbers $1, 2, 3, \ldots$.
`qqnorm(x)` `qqplot(x,y)`	Distribution-comparison plots. The first form plots the numeric vector `x` against the expected Normal order scores (a normal scores plot.) The second form plots the quantiles of `x` against those of `y` to compare their respective distributions.

`hist(x)`
`hist(x,nclass=`n`)`
`hist(x, breaks=...)`
	Produces a histogram of the numeric vector `x`. A sensible number of classes is usually chosen, but a recommendation can be given with the `nclass=` argument. Alternatively, the breakpoints can be specified exactly with the `breaks=` argument. If the `probability=T` argument is given, the bars represent relative frequencies instead of counts.

`dotchart(x,...)`	Construct a dotchart of the data in `x`. In a dotchart the $y-$axis gives a labelling of the data in `x` and the $x-$axis gives its value. For example it allows easy visual selection of all data entries with values lying in specified ranges.
`pie(slices, names, explode=...)`	Make a pie diagram, including the possibility of some pieces displaced or ***exploded*** out from the centre. (Pie diagrams are especially good for showing to administrators and bosses, but not much else, in my opinion.)

11.1.4 Arguments to high-level plotting functions

There are a number of arguments which may be passed to high-level graphics functions, as follows:

`add=T`	Forces the function to act as a low-level graphics function, superimposing the plot on the current plot (some functions only).
`axes=F`	Suppresses generation of axes — useful for adding your own custom axes with the `axis()` function. The default, `axes=T`, means include axes.

`log="x"`
`log="y"`
`log="xy"`
	Causes the x, y or both axes to be logarithmic. Only works for scatterplots (and variants).

`type=`	The `type=` argument controls the type of plot produced, as follows:
`type="p"`	Plot individual points (the default)
`type="l"`	Plot lines
`type="b"`	Plot points connected by lines (***both***)
`type="o"`	Plot points overlaid by lines

`type="h"`	Plot vertical lines from points to the zero axis (***high-density***)
`type="s"` `type="S"`	Step-function plots. In the first form, the top of the vertical defines the point; in the second, the bottom.
`type="n"`	No plotting at all. However axes are still drawn (by default) and the coordinate system is set up according to the data. Ideal for creating plots with subsequent low-level graphics functions.
`xlab="string"` `ylab="string"`	Axis labels for the x and y axes. Use these arguments to change the default labels, usually the names of the objects used in the call to the high-level plotting function.
`main="string"`	Figure title, placed at the top of the plot in a large font.
`sub="string"`	Sub-title, placed just below the x-axis in a smaller font.

11.2 Low-level plotting commands

Sometimes the high-level plotting functions don't produce exactly the kind of plot you desire. In this case, low-level plotting commands can be used to add extra information (such as points, lines or text) to the current plot.

Some of the more useful low-level plotting functions are:

`points(x,y)` `lines(x,y)`	Adds points or connected lines to the current plot. `plot()`'s `type=` argument can also be passed to these functions (and defaults to `"p"` for `points()` and `"l"` for `lines()`.)
`text(x, y, labels, ...)`	Add text to a plot at points given by `x, y`. Normally `labels` is an integer or character vector in which case `labels[i]` is plotted at point `(x[i], y[i])`. The default is `1:length(x)`. Note: This function is often used in the sequence `> plot(x, y, type="n"); text(x, y, names)` The graphics parameter `type="n"` suppresses the points but sets up the axes, and the `text()` function supplies special characters, as specified by the character vector `names` for the points.

`abline(a, b)`
`abline(h=y)`
`abline(v=x)`
`abline(`***lm.obj***`)`
	Adds a line of slope `b` and intercept `a` to the current plot. `h=y` may be used to specify `y`-coordinates for the heights of horizontal lines to go across a plot, and `v=x` similarly for the `x`-coordinates for vertical lines. Also ***lm.obj*** may be list with a `$coefficients` component of length 2 (such as the result of model-fitting functions,) which are taken as an intercept and slope, in that order.

`polygon(x, y, ...)`	Draws a polygon defined by the ordered vertices in (x,y). and (optionally) shade it in with hatch lines, or fill it if the graphics device allows the filling of figures.
`legend(x,y, legend,...)`	Adds a legend to the current plot at the specified position. Plotting characters, line styles, colours etc. are <identified with the labels in the character vector `legend`. At least one other argument `v` (a vector the same length as `legend`) with the corresponding values of the plotting unit must also be given, as follows:

`legend( ,angle=v)`	Shading angles
`legend( , density=v)`	Shading densities
`legend( ,fill=v)`	Colours for filled boxes
`legend( ,col=v)`	Colours in which points or lines will be drawn
`legend( ,lty=v)`	Line styles
`legend( ,pch=v)`	Plotting characters (character vector)
`legend( ,marks=v)`	Plotting symbols, as obtained when using a numeric argument to `pch=` (numeric vector).
`title(main,sub)`	Adds a title `main` to the top of the current plot in a large font and (optionally) a sub-title `sub` at the bottom in a smaller font.
`axis(side,...)`	Adds an axis to the current plot on the side given by the first argument (1 to 4, counting clockwise from the bottom.) Other arguments control the positioning of the axis within or beside the plot, and tick positions and labels. Useful for adding custom axes after calling `plot()` with the `axes=F` argument.

Low-level plotting functions usually require some positioning information (e.g. x and y coordinates) to determine where to place the new plot elements. Coordinates are given in terms of *user coordinates* which are defined by the previous high-level graphics command and are chosen based on the supplied data.

Where `x` and `y` arguments are required, it is also sufficient to supply a single argument being a list with elements named `x` and `y`. Similarly a matrix with two columns is also valid input. In this way functions such as `locator()` (see below) may be used to specify positions on a plot interactively.

11.3 Interactive graphics functions

S-PLUS also provides functions which allow users to extract or add information to a plot using a mouse. The simplest of these is the `locator()` function:

`locator(n,type)`	Waits for the user to select locations on the current plot using the left mouse button. This continues until `n` (default 500) points have been selected, or the middle mouse button is pressed. The `type` argument allows for plotting at the selected points and has the same effect as for high-level graphics commands; the default is no plotting. `locator()` returns the locations of the points selected as a list with two components `x` and `y`.

`locator()` is usually called with no arguments. It is particularly useful for interactively selecting positions for graphic elements such as legends or labels when it is difficult to calculate in advance where the graphic should be placed. For example, to place some informative text near an outlying point, the command

```
> text(locator(1), "Outlier", adj=0)
```

may be useful. `locator()` will still work if the current device does not support a mouse; in this case the user will be prompted for x and y coordinates.

`identify(x,y, labels)`	Allow the user to highlight any of the points defined by `x` and `y` (using the left mouse button) by plotting the corresponding component of `labels` nearby (or the index number of the point if `labels` is absent). Returns the indices of the selected points when the middle button is pressed.

Sometimes we want to identify particular *points* on a plot, rather than their positions. For example, we may wish the user to select some observation of interest from a graphical display and then manipulate that observation in some way. Given a number of (x, y) coordinates in two numeric vectors `x` and `y`, we could use the `identify()` function as follows:

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

The `identify()` functions performs no plotting itself, but simply allows the user to move the mouse pointer and click the left mouse button near a point. The point nearest the mouse pointer will be highlighted with its index number (that is, its position in the `x`/`y` vectors) plotted nearby. Alternatively, you could use some informative string (such as a case name) as a highlight by using the `labels` argument to `identify()`, or disable highlighting altogether with the `plot=F` argument. When the middle button is pressed, `identify()` returns the indices of the selected points; you can use these indices to extract the selected points from the original vectors `x` and `y`.

11.4 Using graphics parameters

When creating graphics, particularly for presentation or publication purposes, S-PLUS does not always produce exactly that which is required. You can, however, customise almost every aspect of the display using *graphics parameters.* S-PLUS maintains a list of a large number of graphics parameters which control things such as line style, colours, figure arrangement and text justification among many others. Every graphics parameter has a name (such as '`col`', which controls colours,) and a value (a colour number, for example.)

A separate list of graphics parameters is maintained for each active device, and each device has a default set of parameters when initialised. Graphics parameters can be set in two ways: either permanently, affecting all graphics functions which access the current device; or temporarily, affecting only a single graphics function call.

11.4.1 Permanent changes: The `par()` function

The `par()` function is used to access and modify the list of graphics parameters for the current graphics device.

`par()`	Without arguments, returns a list of all graphics parameters and their values for the current device.
`par(c("col", "lty"))`	With a character vector argument, returns only the named graphics parameters (again, as a list.)
`par(col=4,lty=2)`	With named arguments (or a single list argument) , sets the values of the named graphics parameters, and returns the original values of the parameters as a list.

Setting graphics parameters with the `par()` function changes the value of the parameters *permanently*, in the sense that all future calls to graphics functions (on the current device) will be affected by the new value. You can think of setting graphics parameters in this way as setting 'default' values for the parameters, which will be used by all graphics functions unless an alternative value is given.

Note that calls to `par()` *always* affect the global values of graphics parameters, even when `par()` is called from within a function. This is often undesirable behaviour — usually we want to set some graphics parameters, do some plotting, and then restore the original values so as not to affect the user's S-PLUS session. You can restore the initial values by saving the result of `par()` when making changes, and restoring the initial values when plotting is complete.

```
> oldpar <- par(col=4,lty=2)
```

. . . plotting commands . . .

```
> par(oldpar)
```

11.4.2 Temporary changes: Arguments to graphics functions

Graphics parameters may also be passed to (almost) any graphics function as named arguments. This has the same effect as passing the arguments to the `par()` function, except that the changes only last for the duration of the function call. For example:

```
> plot(x,y,pch="+")
```

produces a scatterplot using a plus sign as the plotting character, without changing the default plotting character for future plots.

11.5 Graphics parameters list

The following sections detail many of the commonly-used graphical parameters. The S-PLUS help documentation for the `par()` function provides a more concise summary; this is provided as a somewhat more detailed alternative.

Graphics parameters will be presented in the following form:

`name=value`	A description of the parameter's effect. `name` is the name of the parameter, that is, the argument name to use in calls to `par()` or a graphics function. `value` is a typical value you might use when setting the parameter.

11.5.1 Graphical elements

S-PLUS plots are made up of points, lines, text and polygons (filled regions.) Graphical parameters exist which control how these *graphical elements* are drawn, as follows:

`pch="+"`	Character to be used for plotting points. The default varies with graphics drivers, but it is usually '*' for terminals or window devices, and '•' for PostScript devices. Plotted points tend to appear slightly above or below the appropriate position unless you use `"."` as the plotting character, which produces centred points.
`pch=4`	When `pch` is given as an integer between 0 and 18 inclusive, a specialised plotting symbol is produced. To see what the symbols are, use the command `> legend(locator(1),as.character(0:18),marks=0:18)`
`lty=2`	Line types. Alternative line styles are not supported on all graphics devices (and vary on those that do) but line type 1 is always a solid line, and line types 2 and onwards are dotted or dashed lines, or some combination of both.
`lwd=2`	Line widths. Desired width of lines, in multiples of the 'standard' line width. Affects axis lines as well as lines drawn with `lines()`, etc.
`col=2`	Colours to be used for points, lines, text, filled regions and images. Each of these graphic elements has a list of possible colours, and the value of this parameter is an index to that list. Obviously, this parameter applies only to a limited range of devices.
`font=2`	Font to use for text. The appropriate value of this parameter is dependent on the graphics device being used; for the `postscript()` device this is an index to the system dataset `ps.fonts`.
`adj=-0.1`	Justification of text relative to the plotting position. 0 means left justify, 1 means right justify and 0.5 means to centre horizontally about the plotting position. The actual value is the proportion of text that appears to the left of the plotting position, so a value of -0.1 leaves a gap of 10% of the text width between the text and the plotting position.
`cex=1.5`	Character expansion. The value is the desired size of text characters (including plotting characters) relative to the default text size.

11.5.2 Axes and tick marks

Many of S-PLUS's high-level plots have axes, and you can construct axes yourself with the low-level `axis()` graphics function. Axes have three main components: the *axis line* (line style controlled by the `lty` graphics parameter), the *tick marks* (which mark off unit divisions along the axis line) and the *tick labels* (which mark the units.) These components can be customised with the following graphics parameters.

`lab=c(5,7,12)`	The first two numbers are the desired number of tick intervals on the x and y axes respectively. The third number is the desired length of axis labels, in characters (including the decimal point.) Choosing a too-small value for this parameter may result in all tick labels being rounded to the same number!
`las=1`	Orientation of axis labels. 0 means always parallel to axis, 1 means always horizontal, and 2 mean always perpendicular to the axis.
`mgp=c(3,1,0)`	Positions of axis components. The first component is the distance from the axis label to the axis position, in text lines. The second component is the distance to the tick labels, and the final component is the distance from the axis position to the axis line (usually zero). Positive numbers measure outside the plot region, negative numbers inside.
`tck=0.01`	Length of tick marks, as a fraction of the size of the plotting region. When `tck` is small (less than 0.5) the tick marks on the x and y axes are forced to be the same size. A value of 1 gives grid lines. Negative values give tick marks outside the plotting region. Use `tck=0.01` and `mgp=c(1,-1.5,0)` for internal tick marks.
`xaxs="s"` `yaxs="d"`	Axis styles for the x and y axes, respectively. With styles `"s"` (standard) and `"e"` (extended) the smallest and largest tick marks always lie outside the range of the data. Extended axes may be widened slightly if any points are very near the edge. This style of axis can sometimes leave large blank gaps near the edges. With styles `"i"` (internal) and `"r"` (the default) tick marks always fall within the range of the data, however style `"r"` leaves a small amount of space at the edges. Setting this parameter to `"d"` (direct axis) ***locks in*** the current axis and uses it for all future plots (or until the parameter is set to one of the other values above, at least.) Useful for generating series of fixed-scale plots.

11.5.3 Figure margins

A single plot in S-PLUS is known as a `figure` and comprises a *plot region* surrounded by margins (possibly containing axis labels, titles, etc.) and (usually) bounded by the axes themselves. A typical figure appears in Figure 10. Graphics parameters controlling figure layout include:

`mai=c(1,0.5,0.5,0)`	Widths of the bottom, left, top and right margins, respectively, measured in inches.
`mar=c(4,2,2,1)`	Similar to `mai`, except the measurement unit is text lines.

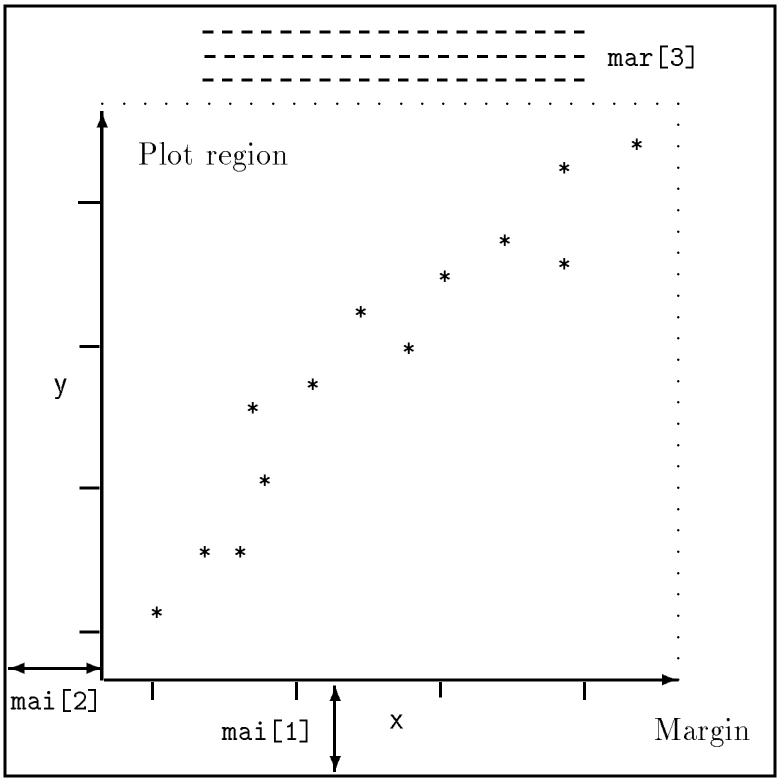

Figure 10: Anatomy of an S figure

`mar` and `mai` are equivalent in the sense that setting one changes the value of the other. The default values chosen for this parameter are often too large; the right-hand margin is rarely needed, and neither is the top margin if no title is being used. The bottom and left margins must be large enough to accommodate the axis and tick labels. Furthermore, the default is chosen without regard to the size of the device surface: for example, using the `postscript()` driver with the `height=4` argument will result in a plot which is about 50% margin unless `mar` or `mai` are set explicitly. When multiple figures are in use (see below) the margins are reduced by half, however this may not be enough when many figures share the same page.

11.5.4 Multiple figure environment

S-PLUS allows you to create an $n \times m$ array of figures on a single page. Each figure has its own margins, and the array of figures is optionally surrounded by an *outer margin* as shown in Figure 11.

The graphical parameters relating to multiple figures are as follows:

`mfcol=c(3,2)`
`mfrow=c(2,4)` Set size of multiple figure array. The first value is the number of rows; the second is the number of columns. The only difference between these two parameters is that setting `mfcol` causes figures to be filled by column; `mfrow` fills by rows. The arrangement in Figure 11 would have been created by setting `mfrow=c(3,2)`; the figure shows the page after four plots have been drawn.

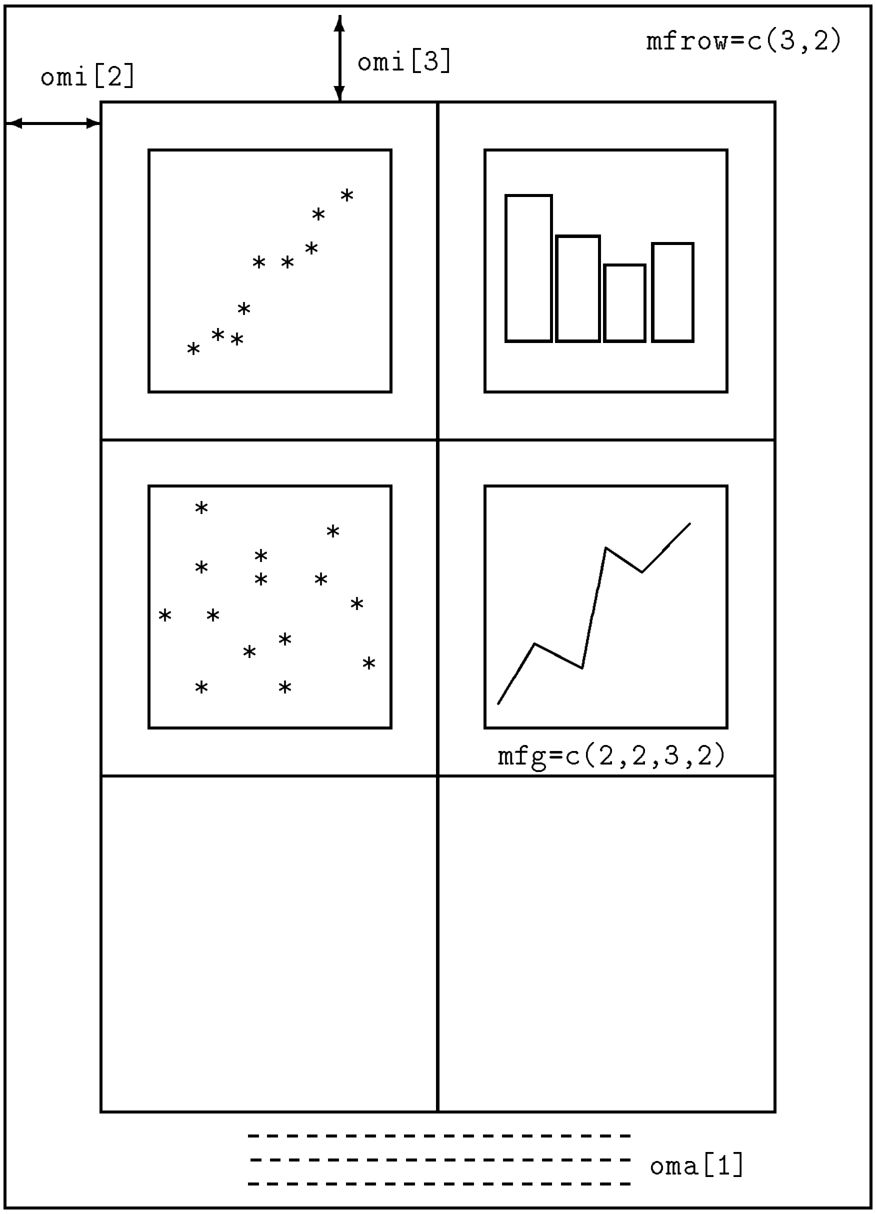

Figure 11: Page layout in multiple figure mode

`mfg=c(2,2,3,2)` Position of current figure in a multiple figure environment. The first two numbers are the row and column of the current figure; the last two are the number of rows and columns in the multiple figure array. Set this parameter to jump between figures in the array. You can even use different values for the last two numbers than the ***true*** values for unequally-sized figures on the same page.

`fig=c(4,9,1,4)/10` Position of the current figure on the page. Values are the positions of the left, right, bottom and top edges respectively, as a percentage of the page measured from the bottom left corner. The example value would be for a figure in the bottom right of the page. Set this parameter for arbitrary positioning of figures within a page.

`oma=c(2,0,3,0)` `omi=c(0,0,0.8,0)`	Size of outer margins. Like `mar` and `mai`, the first measures in text lines and the second in inches, starting with the bottom margin and working clockwise.

Outer margins are particularly useful for page-wise titles, etc. Text can be added to the outer margins with the `mtext()` function with argument `outer=T`. There are no outer margins by default, however, so you must create them explicitly using `oma` or `omi`.

11.6 Device drivers

S-PLUS can generate graphics (of varying levels of quality) on almost any type of display or printing device. Before this can begin, however, S-PLUS needs to be informed what type of device it is dealing with. This is done by starting a *device driver.* The purpose of a device driver is to convert graphical instructions from S-PLUS ('draw a line,' for example) into a form that the particular device can understand.

Device drivers are started by calling a device driver function. There is one such function for every device driver: type `help(Devices)` for a list of them all. For example, issuing the command

```
> postscript()
```

causes all future graphics output to be sent to the printer in PostScript format. Some commonly-used device drivers are:

`motif()` `openlook()` `X11()`	For use with the X11 or Open Windows window systems.
`suntools()`	For use with the SunView windowing system.
`postscript()`	For printing on PostScript printers, or creating PostScript graphics files.
`printer()` `crt()`	For terminals with little or no graphics capabilities. ASCII-based graphics are generated.

When you have finished with a device, be sure to terminate the device driver by issuing the command

```
> dev.off()
```

This ensures that the device finishes cleanly; for example in the case of hardcopy devices this ensures that every page is completed and has been sent to the printer.

11.6.1 PostScript diagrams for typeset documents.

By passing the `file` argument to the `postscript()` device driver function, you may store the graphics in PostScript format in a file of your choice. The plot will be in portrait orientation unless the `horizontal=T` argument is given, and you can control the size of the graphic with the `width` and `height` arguments (the plot will be scaled as appropriate to fit these dimensions.) For example, the command

```
> postscript("file.ps", height=4)
```

will produce a file containing PostScript code for a figure four inches high, perhaps for inclusion in a document.[9] It is important to note that if the file named in the command already exists, it will be overwritten. This is the case even if the file was only created earlier in the same S-PLUS session.

11.6.2 Multiple graphics devices

In advanced use of S-PLUS it is often useful to have several graphics devices in use at the same time. Of course only one graphics device can accept graphics commands at any one time, and this is known as the ***current device***. When multiple devices are open, they form a numbered sequence with names giving the kind of device at any position.

The main commands used for operating with multiple devices, and their meanings are as follows:

`motif()`
`postscript()`
`...`
	Each new call to a device driver function opens a new graphics device, thus extending by one the device list. This device becomes the current device, to which graphics output will be sent.

`dev.list()`	returns the number and name of all active devices. The device at position 1 on the list is always the ***null device*** which does not accept graphics commands at all.
`dev.next()` `dev.prev()`	returns the number and name of the graphics device next to, or previous to the current device, respectively.
`dev.set(which=k)`	can be used to change the current graphics device to the one at position `k` of the device list. Returns the number and label of the device.
`dev.off(k)`	Terminate the graphics device at point `k` of the device list. For some devices, such as `postscript` devices, this will either print the file immediately or correctly complete the file for later printing, depending on how the device was initiated.
`dev.copy(device, ..., which=k)` `dev.print(device, ..., which=k)`	Make a copy of the device `k`. Here `device` is a device function, such as `postscript`, with extra arguments, if needed, specified by `...`. `dev.print` is similar, but the copied device is immediately closed, so that end actions, such as printing hardcopies, are immediately performed. (See also `printgraph()`).
`graphics.off()`	Terminate all graphics devices on the list, except the null device.

[9]**Warning:** The PostScript code produced by the `postscript()` device driver is *not* Encapsulated PostScript, and thus including it in a document electronically (as opposed to physical cut-and-paste) can be rather problematic. For this type of application, a better solution is to use the `fig()` driver (available from statlib) and use a conversion program, such as `fig2dev`, to convert the resultant fig code to Encapsulated PostScript.