> 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)); ld1 <- ld50(coef(fml)); c(ldp, ld1)

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 the \( \text{Var}[y] = \mu \).

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

\[
y = \frac{\theta_1 z_1}{z_2 - \theta_2} + \epsilon
\]

(1)

this may be written alternatively as

\[
y = \frac{1}{\beta_1 x_1 + \beta_2 x_2} + \epsilon
\]

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 \texttt{ms()} for arbitrary minimization problems where the objective functions is a sum of similar terms, and \texttt{nls()} for conventional nonlinear least squares estimation of normal nonlinear regression models.

In this brief introduction we only consider the nonlinear regression function \texttt{nls()} and leave \texttt{ms()} for the reader to pursue as needed.
10.7 Nonlinear regression models; parametrized data frames

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 ?? above, we would use

\[ y \sim t_1 x_1 / (x_2 - t_2) \]

where \( y \) is the response variable, \( x_1 \) and \( x_2 \) are determining variables and \( t_1 \) and \( t_2 \) are scalar parameters.

In such model formulae 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 specifying 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 \( \text{nlm}() \) has a \texttt{nls()} 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 \texttt{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 ??.. An easy way to find initial estimates for the parameters is to regress \( x_2 y \) on \( x_1 \) and \( x_2 \):

\[
> \text{fm0} \leftarrow \text{lm}(x2\ast y \sim x1 + x2 - 1, \text{data} = \text{biochem})
\]

\[
> \text{th} \leftarrow \text{coef(fm0)}
\]

To name the parameters and associate them with the \texttt{biochem} is done as follows:

\[
> \text{parameters(biochem) \leftarrow list(t1=th[1], t2=th[2])}
\]

Now to fit the nonlinear regression model:

\[
> \text{fm} \leftarrow \text{nls}(y \sim t1\ast x1 / (x2 - t2), \text{data} = \text{biochem})
\]

At this point we could use the \texttt{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

\[
> \text{th} \leftarrow \text{coef(fm)}
\]

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

\[
> \text{parameters(biochem) \leftarrow list(t1=th[1], t2=th[2])}
\]

Note that the function \texttt{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 \texttt{attributes()} function. There is also a function \texttt{param()} analogous to \texttt{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 `ireg()`, but others include `lmfit()` 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 formulae 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 an 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 `avan()` and `ace()`. On the other hand `pprep()` 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 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 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 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.

<table>
<thead>
<tr>
<th>plot(x, y)</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>plot(xy)</td>
<td>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.</td>
</tr>
<tr>
<td>plot(f)</td>
<td>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.</td>
</tr>
<tr>
<td>plot(df)</td>
<td>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.</td>
</tr>
</tbody>
</table>

11.1.2 Displaying multivariate data

S 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, i.e. 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:
11.1 High-level plotting commands

\texttt{tsplot(x1,x2,...)}  \hspace{1cm} Plots any number of time series on the same scale. This automatic simultaneous scaling feature is also useful when the x,’s are ordinary numeric vectors, in which case they are plotted against the numbers 1, 2, 3, ...

\texttt{qqnorm(x)}  
\texttt{qqplot(x,y)}  \hspace{1cm} 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.

\texttt{hist(x)}  
\texttt{hist(x,nclass=n)}  
\texttt{hist(x,breaks=...)}  \hspace{1cm} Produces a histogram of the numeric vector \( x \). A sensible number of classes is usually chosen, but a recommendation can be given with the \texttt{nclass=} argument. Alternatively, the breakpoints can be specified exactly with the \texttt{breaks=} argument. If the \texttt{probability=T} argument is given, the bars represent relative frequencies instead of counts.

\texttt{dotchart(z,...)}  \hspace{1cm} 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.

\texttt{pie(slices, names, explode=...)}  \hspace{1cm} 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:

\texttt{add=T}  \hspace{1cm} Forces the function to act as a low-level graphics function, superimposing the plot on the current plot (some functions only).

\texttt{axes=F}  \hspace{1cm} Suppresses generation of axes — useful for adding your own custom axes with the \texttt{axis()} function. The default, \texttt{axes=T}, means include axes.

\texttt{log="x"}  
\texttt{log="y"}  
\texttt{log="xy"}  \hspace{1cm} Causes the \( x \), \( y \) or both axes to be logarithmic. Only works for scatterplots (and variants).

\texttt{type=}  \hspace{1cm} The \texttt{type=} argument controls the type of plot produced, as follows:

\texttt{type="p"}  \hspace{1cm} Plot individual points (the default)

\texttt{type="l"}  \hspace{1cm} Plot lines

\texttt{type="b"}  \hspace{1cm} Plot points connected by lines (“both”)

\texttt{type="o"}  \hspace{1cm} Plot points overlaid by lines

\texttt{type="h"}  \hspace{1cm} Plot vertical lines from points to the zero axis (“high-density”)

\texttt{type="s"}  
\texttt{type="S"}  \hspace{1cm} Step-function plots. In the first form, the top of the vertical defines the point; in the second, the bottom.

\texttt{type="n"}  \hspace{1cm} 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.

\texttt{xlab="string"}  \hspace{1cm} 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.

\texttt{ylab="string"}  \hspace{1cm} Figure title, placed at the top of the plot in a large font.

\texttt{sub="string"}  \hspace{1cm} 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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>points(x, y)</td>
<td>Adds points or connected lines to the current plot. <code>plot()</code>’s <code>type=</code> argument can also be passed to these functions (and defaults to &quot;p&quot; for points () and &quot;l&quot; for lines())</td>
</tr>
<tr>
<td>lines(x, y)</td>
<td>Add text to a plot at points given by x, y. Normally <code>labels</code> is an integer or character vector in which case <code>labels[i]</code> is plotted at point <code>(x[i], y[i])</code>. The default is <code>1:length(x)</code>. Note: This function is often used in the sequence &gt; <code>plot(x, y, type=&quot;n&quot;); text(x, y, names)</code> The graphics parameter <code>type=&quot;n&quot;</code> suppresses the points but sets up the axes, and the <code>text()</code> function supplies special characters, as specified by the character vector <code>names</code> for the points.</td>
</tr>
<tr>
<td>abline(a, b)</td>
<td>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 <code>lm.obj</code> may be list with a <code>$coefficients</code> component of length 2 (such as the result of model-fitting functions,) which are taken as an intercept and slope, in that order.</td>
</tr>
<tr>
<td>polygon(x, y, ...)</td>
<td>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.</td>
</tr>
<tr>
<td>legend(x,y, legend,...)</td>
<td>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 <code>legend</code>. At least one other argument <code>v</code> (a vector the same length as <code>legend</code>) with the corresponding values of the plotting unit must also be given, as follows:</td>
</tr>
<tr>
<td>legend( ,angle=v)</td>
<td>Shading angles</td>
</tr>
<tr>
<td>legend( ,density=v)</td>
<td>Shading densities</td>
</tr>
<tr>
<td>legend( ,fill=v)</td>
<td>Colours for filled boxes</td>
</tr>
<tr>
<td>legend( ,col=v)</td>
<td>Colours in which points or lines will be drawn</td>
</tr>
<tr>
<td>legend( ,lty=v)</td>
<td>Line styles</td>
</tr>
<tr>
<td>legend( ,pch=v)</td>
<td>Plotting characters (character vector)</td>
</tr>
<tr>
<td>legend( ,marks=v)</td>
<td>Plotting symbols, as obtained when using a numeric argument to <code>pch=</code> (numeric vector).</td>
</tr>
<tr>
<td>title(main,sub)</td>
<td>Adds a title <code>main</code> to the top of the current plot in a large font and (optionally) a sub-title <code>sub</code> at the bottom in a smaller font.</td>
</tr>
<tr>
<td>axis(side,...)</td>
<td>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 <code>plot()</code> with the <code>axes=FALSE</code> argument.</td>
</tr>
</tbody>
</table>

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 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:

\[\text{locator(n,type)}\]

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

\[> \text{text(locator(1), "Outlier", adj=0)}\]

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.

\[\text{identify(x,y,}\]
\[\text{labels)}\]

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:

\[> \text{plot(x,y)}\]
\[> \text{identify(x,y)}\]

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 and highlighted with its index number (i.e. 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 does not always produce exactly that which is required. You can, however, customise almost every aspect of the display using `graphics parameters`. S 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.
11.5 Graphics parameters list

The following sections detail many of the commonly-used graphical parameters. The S 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, i.e. 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 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 |

```r
> legend(locator(1), as.character(0:18), marks=0:18)
```
11.5 Graphics parameters list

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lty=2</td>
<td>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.</td>
</tr>
<tr>
<td>lwd=2</td>
<td>Line widths. Desired width of lines, in multiples of the ‘standard’ line width. Affects axis lines as well as lines drawn with lines(), etc.</td>
</tr>
<tr>
<td>col=2</td>
<td>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.</td>
</tr>
<tr>
<td>font=2</td>
<td>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.</td>
</tr>
<tr>
<td>adj=0.1</td>
<td>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 percentage 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.</td>
</tr>
<tr>
<td>cex=1.5</td>
<td>Character expansion. The value is the desired size of text characters (including plotting characters) relative to the default text size.</td>
</tr>
</tbody>
</table>

11.5.2 Axes and Tick marks

Many of S’s high-level plots have axes, and you can 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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lab=c(5,7,12)</td>
<td>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!</td>
</tr>
<tr>
<td>las=1</td>
<td>Orientation of axis labels. 0 means always parallel to axis, 1 means always horizontal, and 2 mean always perpendicular to the axis.</td>
</tr>
<tr>
<td>mgp=c(3,1,0)</td>
<td>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.</td>
</tr>
<tr>
<td>tck=0.01</td>
<td>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.</td>
</tr>
<tr>
<td>xaxs=&quot;s&quot; yaxs=&quot;d&quot;</td>
<td>Axis styles for the x and y axes, respectively. With styles &quot;s&quot; (standard) and &quot;e&quot; (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 &quot;i&quot; (internal) and &quot;r&quot; (the default) tick marks always fall within the range of the data, however style &quot;r&quot; leaves a small amount of space at the edges. Setting this parameter to &quot;d&quot; (direct axis) &quot;locks in&quot; 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.</td>
</tr>
</tbody>
</table>
11.5.3 Figure Margins

A single plot in S 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 ??.

Graphics parameters controlling figure layout include:

\[
\text{mar} = c(4, 2, 2, 1) \quad \text{Similar to \texttt{mai}, except the measurement unit is text lines.}
\]

\[
\text{mai} = c(1, 0.5, 0.5, 0) \quad \text{Widths of the bottom, left, top and right margins, respectively, measured in inches.}
\]

\texttt{mar} and \texttt{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 \texttt{postscript()} driver with the \texttt{height=4} argument will result in a plot which is about 50% margin unless \texttt{mar} or \texttt{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 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 ??.

The graphical parameters relating to multiple figures are as follows:
Figure 11: Page layout in multiple figure mode

\begin{itemize}
  \item \texttt{mfcol=c(3,2)}
  \item \texttt{mfrow=c(2,4)}
  \begin{itemize}
    \item 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 \texttt{mfcol} causes figures to be filled by column; \texttt{mfrow} fills by rows. The arrangement in Figure ?? would have been created by setting \texttt{mfrow=c(3,2)}; the figure shows the page after four plots have been drawn.
  \end{itemize}
  \item \texttt{mfg=c(2,2,3,2)}
  \begin{itemize}
    \item 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.
  \end{itemize}
  \item \texttt{fig=c(4,9,1,4)/10}
  \begin{itemize}
    \item 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.
  \end{itemize}
\end{itemize}
\texttt{oma=c(2,0,3,0)} \quad \text{Size of outer margins. Like \texttt{mar} and \texttt{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 \texttt{mtext()} function with argument \texttt{outer=T}. There are no outer margins by default, however, so you must create them explicitly using \texttt{oma} or \texttt{omi}.

### 11.6 Device drivers

\textit{S} can generate graphics (of varying levels of quality) on almost any type of display or printing device. Before this can begin, however, \textit{S} needs to be informed what type of device it is dealing with. This is done by starting a \textit{device driver}. The purpose of a device driver is to convert graphical instructions from \textit{S} (‘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 \texttt{help(Devices)} for a list of them all. For example, issuing the command

\begin{verbatim}
> postscript()
\end{verbatim}

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

\begin{itemize}
  \item \texttt{motif()} \quad For use with the X11 or Open Windows window systems.
  \item \texttt{openlook()} \quad \texttt{X11()} \quad For use with the SunView windowing system.
  \item \texttt{suntools()} \quad For printing on \texttt{PostScript} printers, or creating \texttt{PostScript} graphics files.
  \item \texttt{printer()} \quad For terminals with little or no graphics capabilities. ASCII-based graphics are generated.
  \item \texttt{crt()} \quad \texttt{postscript()} \quad \texttt{pdf()} \quad \texttt{ps()} \quad \texttt{png()} \quad \texttt{svg()} \quad \texttt{svg+pdf()} \quad \texttt{svg+png()}
\end{itemize}

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

\begin{verbatim}
> dev.off()
\end{verbatim}

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

\begin{verbatim}
> postscript("file.ps", height=4)
\end{verbatim}

will produce a file containing \texttt{PostScript} code for a figure 4 inches high, perhaps for inclusion in a document. 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 \texttt{S} session.

\textbf{Warning}: The \texttt{PostScript} code produced by the \texttt{postscript()} device driver is not Encapsulated \texttt{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 \texttt{fig()} driver (available from \texttt{statlib}) and use a conversion program (such as \texttt{fig2dev} to convert the resultant \texttt{fig} code to Encapsulated \texttt{PostScript}.  

---

\textsuperscript{8}Warning: The \texttt{PostScript} code produced by the \texttt{postscript()} device driver is not Encapsulated \texttt{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 \texttt{fig()} driver (available from \texttt{statlib}) and use a conversion program (such as \texttt{fig2dev} to convert the resultant \texttt{fig} code to Encapsulated \texttt{PostScript}.  

---
### 11.6.2 Multiple graphics devices

In advanced use of S it is often useful to have several graphics devices in use at the one 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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>motif()</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>postscript()</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>dev.list()</strong></td>
<td>returns the number and name of the graphics device next to, or previous to the current device, respectively.</td>
</tr>
<tr>
<td><strong>dev.next()</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>dev.prev()</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>dev.copy(device, ..., which=k)</strong></td>
<td>Make a copy of the device k. 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()).</td>
</tr>
<tr>
<td><strong>dev.print(device, ..., which=k)</strong></td>
<td>Terminate all graphics devices on the list, except the null device.</td>
</tr>
</tbody>
</table>

### A S-PLUS: An Introductory Session

The following session is intended to introduce you to some features of the S-PLUS environment by using them. Many features of the system will be unfamiliar and puzzling at first, but this will soon disappear.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>login:</strong></td>
<td>Login, start your windowing system (ask a demonstrator if you need help), and check that your working directory has a subdirectory .Data, which in turn contains the files .First, .Last and possibly .Audit.</td>
</tr>
<tr>
<td><strong>...</strong></td>
<td>You should also have the file mceley.data in your working directory. If not, see a demonstrator. If you have, proceed.</td>
</tr>
<tr>
<td><strong>&gt; ls -a</strong></td>
<td>Start Splus with the inbuilt command line editor enabled.</td>
</tr>
<tr>
<td><strong>&gt; ls -a .Data</strong></td>
<td>The S-PLUS program begins, with a banner. (Within S-PLUS the prompt on the left hand side will not be shown to avoid confusion.)</td>
</tr>
<tr>
<td><strong>help.findsim(&quot;.Data&quot;)</strong></td>
<td>Set things up for the help facility. (Need only be done once for this directory.)</td>
</tr>
<tr>
<td><strong>help.start()</strong></td>
<td>Start the help facility. You should briefly explore the features of this facility with the mouse. Standard X–window conventions apply.</td>
</tr>
<tr>
<td><strong>X11()</strong></td>
<td>Iconify the help window and move on to the next part.</td>
</tr>
<tr>
<td><strong>T</strong></td>
<td>Turn on the graphics window. You may need to re-position and re-size to make it convenient to work with both windows.</td>
</tr>
</tbody>
</table>
x <- rnorm(50)  
y <- rnorm(x)        
hull <- chol(x, y)  
plot(x, y)         
polygon(x[hull], y[hull], 
            dens=15)   
objects()          
rm(x, y, hull)     
x <- 1:20          
w <- 1 + sqrt(x)/2 
dummy <- data.frame(x=x, 
                     y=x + rnorm(x)*w)  
dummy     
fm <- lm(y~x, data=dummy)  
summary(fm)       
fm1 <- lm(y~x, data=dummy, 
          weight=1/w^2)   
summary(fm1)      
lrf <- loess(y~x, dummy)   
attach(dummy)     
plot(x, y)        
lines(x, fitted(lrf))  
abline(0, 1, lty=3) 
abline(coef(fm))   
abline(coef(fm1), lty=4) 
plot()            
plot(fitted(fm), 
     resid(fm), 
     xlab="Fitted values", 
     ylab="Residuals", main= 
     "Residuals vs Fitted")  
qqnorm(resid(fm), main= 
     "Residuals Rankit Plot") 
rms(fm, fm1, lrf, x, dummy) 

The next section will look at data from the classical experiment of Michaelson and Morley to measure the speed of light.

!more morley.data 

mm <- read.table("morley.data") 
mm attach(mm, 1) 
objects()  
Expt <- factor(Expt) 
Run <- factor(Run) 

Optional. Temporarily interrupt S-PLUS and look at the file. This is a standard way to escape to the operating system. 
Read in the MM data as a data frame, and look at it. There are five experiments (col. Expt) and each has 20 runs (col. Run) and a1 is the recorded speed of light, suitably coded. 
Place mm on the top of the search list, (position 1). 
Change Expt and Run into factors.
Save the changes and make the data frame visible at position 2 (the default).

Compare the five experiments with simple boxplots.

Analyse as a randomized block, with ‘runs’ and ‘experiments’ as factors.

Fit the sub-model omitting ‘runs’, and compare using a formal analysis of variance.

Cleanup before moving on.

We now look at some more graphical features: contour and 3-dimensional perspective plots.

$x$ is a vector of 50 equally spaced values in $-\pi \leq x \leq \pi$. $y$ is the same.

$f$ is a square matrix, with rows and columns indexed by $x$ and $y$ respectively, of values of the function $\cos(y)/(1+x^2)$.

Save the plotting parameters and set the plotting region to “square”.

Make a contour map of $f$; add in more lines for more detail.

fa is the “asymmetric part” of $f$. (t( ) is transpose).

Make a contour, and restore the old graphics parameters.

Make some pretty perspective and high density image plots, (of which you can get hardcopies if you wish)

and clean up before moving on.

S-PLUS can do complex arithmetic, also. 1i is used for the complex number $i$

Plotting complex arguments means plot imaginary versus real parts. This should be a circle.

Suppose we want to sample points within the unit circle. One method would be to take complex numbers with standard normal real and imaginary parts – and to map any outside the circle onto their reciprocal.

All points are inside the unit circle, but the distribution is not uniform.
w <- sqrt(runif(100))*
exp(2*pi*runif(100)*1i)
plot(w, xlim=c(-1,1),
ylim=c(-1,1), pch="*",
xlabel="x", ylab="y")
lines(z)
rm(th,w,z)  The second method uses the uniform distribution. The points should now look more
evenly spaced over the disc.

par(oldpar) Restore standard graphics parameters.
butterfly() An old favourite. Take a hardcopy if you wish.
rm(oldpar) Cleanup again.
q() Quit the S-PLUS program.
> and return to Unix.

B  The Inbuilt Command Line Editor in S-PLUS

B.1 Preliminaries

The August 1991 release of S-PLUS has inbuilt command line editor that allows recall, editing and re-submission of prior
commands.

To use it, start the S-PLUS programme with

$ Splus -e

Inside the editor either emacs or vi conventions are available, according to the shell environment variable S_CLEditor. To get
the emacs conventions use (in csh and variants)

$ setenv S_CLEditor emacs

and for the vi conventions to apply, put vi instead of emacs. This statement would normally be included in your .login file
(or equivalent) and would then be done automatically at login time. To avoid forgetting to include the -e a handy alias for
your .cshe file is, say

alias S+ 'Splus -e'

after which S+ is the command to start Splus with command line editor.

The usual typographical conventions apply: "^M means Hold the Control down while you press the m key, but Esc m means
First press the Esc key and then the m key. Note that after Esc case is significant.

B.2 Editing Actions

The S-PLUS programme keeps a history of the commands you type, including the error lines, and commands in your history
may be recalled, changed if necessary, and re-submitted as new commands. In emacs style command line editing any straight
typing you do while in this editing phase causes the characters to be inserted in the command you are editing, displacing
any characters to the right of the cursor. In vi mode character insertion mode is started by Esc i or Esc a, characters are
typed and insertion mode is finished by typing a further Esc .

Pressing the Return command at any time causes the command to be re-submitted.

Other editing actions are summarised in the following table.

Unfortunately it does not seem to be possible to bind the motion keys, for example, to the arrow keys, which is something
of a nuisance.
### 1. Command recall and vertical motion

<table>
<thead>
<tr>
<th>emacs style</th>
<th>vi style</th>
</tr>
</thead>
<tbody>
<tr>
<td>#P</td>
<td>Esc k</td>
</tr>
<tr>
<td>#N</td>
<td>Esc j</td>
</tr>
<tr>
<td>#R text</td>
<td>Esc ? text</td>
</tr>
</tbody>
</table>

### 2. Horizontal motion of the cursor

<table>
<thead>
<tr>
<th>emacs style</th>
<th>vi style</th>
</tr>
</thead>
<tbody>
<tr>
<td>#A</td>
<td>Esc ^</td>
</tr>
<tr>
<td>#E</td>
<td>Esc $</td>
</tr>
<tr>
<td>Esc b</td>
<td>Esc b</td>
</tr>
<tr>
<td>Esc f</td>
<td>Esc w</td>
</tr>
<tr>
<td>#B</td>
<td>Esc h</td>
</tr>
<tr>
<td>#F</td>
<td>Esc l</td>
</tr>
</tbody>
</table>

### 3. Editing and re-submission

<table>
<thead>
<tr>
<th>emacs style</th>
<th>vi style</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert text at the cursor</td>
<td>text Esc i text Esc</td>
</tr>
<tr>
<td>Append text after cursor</td>
<td>#Ftext Esc a text Esc</td>
</tr>
<tr>
<td>Delete the previous character (left of the cursor)</td>
<td>Delate Esc shift-x</td>
</tr>
<tr>
<td>Delete the character under the cursor</td>
<td>#D Esc x</td>
</tr>
<tr>
<td>Delete rest of the word under the cursor, and ‘save’ it</td>
<td>Esc d Esc dw</td>
</tr>
<tr>
<td>Delete from cursor to end of command, and ‘save’ it</td>
<td>#K Esc shift-d</td>
</tr>
<tr>
<td>Insert (yank) the last ‘saved’ text here</td>
<td>#Y Esc shift-y</td>
</tr>
<tr>
<td>Transpose the character under the cursor with the next</td>
<td>#T Esc xp</td>
</tr>
<tr>
<td>Change the rest of the word to lower case</td>
<td>Esc l</td>
</tr>
<tr>
<td>Change the rest of the word to capitals (upper case)</td>
<td>Esc c</td>
</tr>
<tr>
<td>Re-submit the command to S-PLUS</td>
<td>Return Return</td>
</tr>
</tbody>
</table>

**NOTE:** With vi style commands the Esc need only be issued before the first recall command, and to terminate insert and append commands, as is usual in vi.

The final Return terminates the command line editing sequence for commands of either style.