Statistical Models and Graphics in Splus

Phil Spector (`spector@stat.berkeley.edu`)
Statistical Computing Facility
Department of Statistics
University of California, Berkeley

1 Overview

One of the major additions to Splus in recent years has been a unified framework for working with statistical models. The key to this unification is the use of the data frame, which is an Splus object which can contain both numeric and character variables, and some special operators which are used to describe statistical models. More recently, this framework has been extended to the arena of statistical graphics, where the same model operators are used to describe a wide variety of graphical displays of data. The purpose of this paper is to introduce the concept of data frames and the statistical modeling operators and to show how these tools can be used both for working with statistical models and producing useful statistical graphics.

2 Data for Statistical Analysis

Most model building techniques make a distinction between variables which are factors (*i.e.* categorical variables), representing discrete levels of a quantity, and regressors, which are a numeric quantity expressed as a continuous range of values. For example, an analysis of variance model would estimate three parameters for a factor which takes on four distinct values, while it would estimate only one parameter for a regressor, regardless of the number of different values which it may have. It is very common for the different levels of a factor to be indexed by a character string, and, under old versions of Splus, this created some problems in storing factors and regressors in the same data object. To solve this problem, a new type of object called a data frame was created.

A data frame is a cross between a matrix and a list. Thus, it can have a mixture of character variables and numeric variables, but specific columns of the data frame, representing specific variables, can be accessed through either list techniques (using the `$` or `[[` operators), or through matrix subscripting techniques (using `[`). More importantly, since each variable is a separate object in the data frame's internal representation, a variety of information about each variable can be stored within the data frame. Data frames can be created in a variety of ways. If your data is either all character or all numeric, you can convert a data matrix into a data frame with the `as.data.frame` function; if you wish to combine a number of vectors of equal length, each representing a different variable into a data frame, you can use the `data.frame` function, and if you need to read raw data from a file directly into a data frame, you can use the `read.table` function. Each variable in the data frame must have a unique (within the data frame) name; these can be accessed or set with the `names` function (like a list) or by using the second component of the list returned by the `dimnames` function (like a matrix). It is often helpful to use descriptive names for the variables in a data frame, but be aware of the fact that if you include blanks in a variable name, it may be difficult to refer to the variables in some settings. Regardless of the names of the variables (columns) of a data frame, you can also access elements in a data frame in exactly the same way as is done with a matrix, using either numeric, character or logical subscripts.

3 An Example

Consider a data set containing information about growing conditions of apples, their mineral content, mean weight and incidence of bitter pits. (This data set is a modification of one from the book "Data" by Andrews and Herzberg.) The variables in the data set are treatment, block, total nitrogen, active nitrogen, phosphorus, potassium, calcium, magnesium, fruit weight and percent incidence of bitter pit. A sample of lines from the data set look like this:

```
Control  1 2880 1670 0836  9840 142 367 113.8  3.2
Urea     4 4340 1990 0916 10440 180 428  99.9 20.4
Urea     4 4130 1870 0710  9040 199 363  84.6  0.0
Nitrate  4 4370 2080 0874 12560 183 404 110.8 10.0
```

Note that, since one of the variables is a character variable, you can not create a matrix out of this data using the `scan` function of Splus. The simplest way to read this data set is using the `read.table` function, which automatically determines which variables are character and which are numeric, and whose output is a data frame:

```
    apple <- read.table("apple.data")
```

assuming the data was stored in a file called `apple.data`. If the first line of the file contains variable names, they are read automatically using the `header=T` argument to `read.table`. Otherwise the names can be assigned using the `names()<-` function:

```
 names(apple) <- c("Treatment","Block","TN","AN","P","K","Ca","Mg",
                "Weight","Bitter")
```

If desired, a character vector can be stored as a label for each row of the matrix. The function `row.names` works in an analogous manner to the function `names` in the example above. By default, the row names of a data frame are a character representation of the integers from 1 to the number of rows in the data frame.

Once the data frame has been created, variables can be identified as factors by using the `factor` function. Since Treatment is a character variable, there is no need to identify it as a factor, since Splus automatically changes character variables in a data frame into factors. Block, on the other hand, should not be used as a numeric variable, but as a factor. This information can be stored in the data frame as follows:

```
      apple$Block <- factor(apple$Block)
```

Whenever the variable Block in the apple data frame is referred to in a statistical formula, it will be treated as a categorical variable. If you want a label for the levels of a factor which is not just a character representation of the numeric value, you can use the `levels=` and `labels=` arguments to the `factor` function. For example, to use roman numerals to identify the levels of Block, you could use the following:

```
      apple$Block <- factor(apple$Block,levels=c(1,2,3,4),
                                        labels=c("I","II","III","IV"))
```

Note that once you have converted a variable to a factor, it's labelled values will appear whenever the variable is printed. To convert the labeled levels of a factor to a set of consecutive integers, you can use the function `code`.

Model	Function	Model	Function
Linear Models	`lm`	Local Regression	`loess`
Analysis of Variance	`aov`	Tree-based Models	`tree`
Generalized Linear Models	`glm`	Non-linear Models	`nls`
Generalized Additive Models	`gam`	Optimization	`ms`
Proportional Hazards	`coxph`	Variance Components	`varcomp`

Table 1: Statistical Modeling Functions

4 Functions for Statistical Modeling

Table 1 shows some of the statistical modeling functions provided as part of the Splus language; in addition, many user-written functions also support the modeling language, so this list should not be thought of as all-inclusive.

Each of the functions shares a common syntax and creates as output an object which can be displayed and manipulated by a set of common routines. Thus, once one analysis is carried out on a data set, it is very easy to modify either the method or the specifics of the analysis. In fact, the major breakthrough with regard to statistical models in Splus is more in the tools and methods surrounding the analyses than in the algorithms themselves. Using an object oriented framework, each of the statistical modeling functions produces an object which has a class attribute equal to the name of the analysis itself. So, for example, the `glm` function produces an object which has a class of "`glm`"; this fact is often abbreviated by saying that the `glm` function produces a glm object. A wide variety of functions, known as generic functions, are provided to operate on these different objects. You can recognize a generic function because the function body consists of nothing more than a call to the Splus function `UseMethod`. Table 2 shows some of these functions. For each generic function, there will exist a variety of related functions with names of the form *`function.object`*, representing the generic *`function`* for object type *`object`*. For example, the actual print function which would print a glm object is called `print.glm`; the function which would plot a tree object is called `plot.tree`. The utility of this system is that you need not concern yourself with these details; when you pass a tree object to the plot routine, it knows to use the `plot.tree` function. In addition, to extend the system to produce and operate on some new class of analysis object, you simply need to write a function to produce the object, and then write a collection of appropriately named functions (often called methods) to operate on the object. If a specific method for a class does not exist, then the system uses the function of the form *`functionname`*`.default`.

5 Expressing a Statistical Model in Splus

All of the modeling functions mentioned in Table 1 (except for nonlinear modeling and optimization) accept formulas as described in this section. In addition, the graphics functions described in Section 12 also use formulas to express the relations between the variables to be plotted. Statistical models in Splus are facilitated through a new operator, the tilde (˜), which can be loosely translated as "is modeled by". Thus, we can create a formula which models the weights of the apples as a function of the total nitrogen levels with the formula

```
Weight ~ TN
```

The exact nature of the function which is being used to model the response (in this example the variable Weight) varies from method to method. For example, the `lm` function creates a linear additive model with normally distributed errors, while the `glm` function models the response as a specified function (known as the link function) of a linear additive model, with an error term arising from one of a variety of distributions.

General Purpose Functions	
`print`	Displays the contents of an object
`plot`	Produces a graphical display on a graphics device
`text`	Annotates a graphical display with text
`all.equal`	Checks for equality of two objects
`summary`	Displays Summary information about an object
Statistical Modeling Functions	
`predict`	Calculates Predicted Values from a model
`fitted`	Calculates Fitted Values from a model
`coef`	Extracts Coefficients from a model
`deviance`	Reports the deviance for a model
`update`	Modifies an existing model
`resid`	Calculates residuals from a model
`anova`	Displays an ANOVA table for a model
`step`	Builds a model in a stepwise fashion
`add1`	Adds a term to a model
`drop1`	Drops a term from a model

Table 2: Generic Methods

(One special note about models using the binomial error distribution: in these cases, you should express your dependent variable as a two column matrix, with the first column representing the number of "successes" and the second column representing the number of "failures", unless your dependent variable is a vector of zeroes and ones representing individual observations.)

Additional terms can be added to the model using the "`+`" operator. To add Treatment and Block to the analysis, for example, you could use the following formula:

```
Weight ~ Treatment + Block + TN
```

By default, an intercept is fitted to all models. To fit a model without an intercept, include the term `-1` in the formula; to fit a model which contains only an intercept, explicitly include the term `1` as the only term on the right hand side of the formula.

More complicated terms can be created using the "`:`" (interaction), "`*`" (crossing) and "`/`"" (nesting) operators. To fit an interaction between Block and Treatment, we could add (using the "`+`" operator) the term `Block:Treatment`. Alternatively, we could fit both the Block and Treatment main effects along with the interaction with the term `Block*Treatment`. If the levels of Block had different meanings within each of the levels of treatment, then no main effect for Block alone should be fit; only a main effect for Treatment and the interaction between Block and Treatment. Such an effect can be added to the model using the term `Treatment/Block`, representing Block nested within Treatment. Finally, if there are more than two terms in a crossed effect, but higher order interactions are not desired in the model, you can specify the maximum order of interactions to include using the "`^`" operator. Thus, the term `(A*B*C)^2` would include only interactions up to order 2, i.e. `A`, `B`, `C`, `A:B`, `A:C` and `B:C`; the three way interaction `A:B:C` would not be included.

Terms in a model are not limited to just variables; any valid Splus expression is acceptable. For example, if we wished to fit a model where the variable TN is divided into three ranges with values 1, 2 and 3, we could fit the model:

```
Weight ~ cut(TN,3)
```

(Splus knows that the result of the cut command should be treated as a factor; if this is not the case, you can use the function `codes` as described above.)

Because some of the modeling operators have other meanings within Splus, care must be taken when using expressions within formulas. For example, suppose we wish to fit a term representing the square of calcium (Ca) concentration in the apple example. If we simply include the term `Ca*Ca` in the formula, nothing will be added to the model because, within the context of a formula, the "`*`" represents the interaction operator, and interactions which do not involve at least one factor are not defined. In cases like these, the function `I` can be used to "protect" the expression from expansion. Thus, to fit a third order polynomial model relating Weight to Ca, we could use the formula:

```
Weight ~ Ca + I(Ca*Ca) + I(Ca*Ca*Ca)
```

Of course problems such as these can usually be avoided by creating new variables.

6 Data Frames and Statistical Models

Since a data frame is a list inside of Splus, one way of specifying the variables within a model is by using their "complete" names. For example, to model the 0-1 variable indicating whether more than 10 per cent of apple seeds were bitter, we could express the model described above as

```
(apple$Bitter >= 10) ~ apple$TN + apple$Ca
```

Of course, having to type in the word "apple" each time becomes tiring. Since the idea of accessing a data frame as the source of all the variables in an analysis is a basic operation in data analysis, there are two methods provided in Splus to allow you to refer to the variable names directly, without specifying the name of the data frame each time.

First, each of the statistical modeling functions accepts an argument called `data` which allows you to specify the name of the data frame which should be used to resolve variable references within a formula.[1] For example, to fit the previously described model in a logistic regression, the following two statements are equivalent:

```
z <- glm((apple$Bitter >= 10) ~ apple$TN + apple$Ca,family = binomial)
z <- glm(data=apple,(Bitter >= 10) ~ TN + Ca,family = binomial)
```

One additional feature which is available when specifying the data frame through the `data=` argument is the ability to use the symbol "`.`" to represent the additive effect of all the other variables in a data frame. In the apple example, if we used the statement

```
z <- glm(data=apple,(Bitter >= 10) ~ .,family = binomial)
```

we would fit a logistic regression to the `Bitter >= 10` variable using all the other variables in the `apple` data frame. Since the `data=` argument will accept any part of a data frame, this can be a useful way to express a fairly complex model. Suppose we wished to fit a dependent variable `X` to five independent variables out of a set of 20 stored in a data frame called `mydata`. Assume that `X` is the first variable in the data frame, followed by the five variables of interest. Since data frames can be accessed using matrix operations, we could write

[1]A similar mechanism is available for evaluating arbitrary expressions in a given data frame through the function `eval()`. However, since the Splus parser would try to evaluate an unprotected expression, it must be passed to eval as a call to the function `expression()`. For example, to calculate the mean of the variable Bitter in the apple data frame, you could use `eval(expression(mean(Bitter)),apple)`.

```
        z <- glm(data=mydata[,1:6],X ~ .,family = binomial)
```

and the `.` in the model formula will represent only the selected variables. Alternatively, you can explicitly exclude certain variables from the model using a minus sign ("`-`"). This is especially useful when you are using a dot to represent all the other variables in the data frame, and you are also specifying a `weights=` variable, because in such a case you generally do not want to fit the `weights=` variable to the data. In the above example, if the sixth column of `mydata` was called `V6`, we could use it as a weighting variable as follows:

```
    z <- glm(data=mydata[,1:6],X ~ . - V6,family = binomial,weights=V6)
```

When you will be using a particular data frame more extensively, it may be easier to attach it to the current environment in much the same way that a directory is added to the search list of Splus functions. Following the example above, we could make the apple data frame the first area searched by Splus with the command

```
            attach(apple,1)
```

Now, whenever you refer to, say, `Bitter`, Splus will know that you are referring to the variable Bitter in the apple data frame. (If a variable does not exist in the first attached data frame or directory, Splus will continue searching through other attached data frames and/or directories to find it. To see what is attached, use the Splus function `search()`.) Two caveats apply to the technique of attaching a data frame. First, the `ls()` function does not operate on data frames. Instead, you must use the new function `objects()` which works correctly on both data frames and directories. Second, when a data frame is attached, only a copy of the data frame is actually being used by Splus. Thus, if you wish to save the changes made to an attached data base before you terminate your Splus session, you should detach the data frame using the `save=` argument of the `detach()` command.

7 Missing Values

Along with a unified approach to statistical modeling, the new modeling functions also provide new and unified facilities for handling missing values. (Missing values are represented in Splus by the symbol `NA`.) Each of the modeling procedures accepts an argument called `na.action=`, which is the name of an Splus function which will check the data set in question for missing values, and perform an appropriate action. The two most common functions supplied with Splus which are suitable for the `na.action=` argument are `na.fail`, the default, which simply causes the function to quit when any missing values are found in the data, and `na.omit`, which omits any case which is found to have missing values. Additionally, the function `na.include` will create an additional level (representing missing values) for any factor in your data which has missing values, since by default, `NA` is not a valid value for any variable. While these functions will serve most needs, you can customize missing value handling to suit your particular needs by writing a function which operates on the evaluation frame of the data in question, and passing it's name as the `na.action=` argument to any of the modeling functions. You can examine the code for the function `na.omit` by typing the function name if you wish to see how such functions are constructed.

Functions such as `na.omit` are also useful in their own right. If a modeling function which specifies `na.omit` as its `na.action` removes observations due to the presence of missing values, it may be necessary to recreate the data frame with those observations removed. Since the `na.omit` function returns a data frame with those observations removed, it can be called directly to create a data frame corresponding to the one used in the analysis.

8 Classes of Objects

Central to the way in which statistical models are implemented inside of Splus is the concept of a class of objects. A class of objects is simply a specification of an object which conforms to certain rules regarding the information which it contains. More importantly, methods can be developed which are capable of dealing with a wide class of objects. Producing objects and developing methods capable of dealing with them is the core of a technique known as object oriented programming. In Splus, methods are implemented for a specific type of object by appending a period and the name of the object class to a method's name. When confronted with a request for a method, the Splus parser will first check to see if the object being passed to the method is of a particular class; if so, it will automatically search for the appropriate function. For example, the `glm()` function returns a glm object; when you type `summary(z)`, where z is a glm object, the `summary()` function automatically searches for a function called `summary.glm()`. (If it fails to find the appropriate function for the glm object, it uses the function `summary.default()`.) In this way, useful summaries of different types of objects can be produced through a single user interface, namely the function `summary()`. Note that when you specify the method of choice, you do not need to specify or even be aware of the class of the object you are passing to the method; the resolution is done automatically. A number of method oriented functions are implemented within Splus to deal with statistical modeling. For example, a family of functions (based on the `print()` function), exists to allow printing of the various model objects (including the default action when an object's name is printed.) If you were to create a new type of object, say an "xyz" object, then the `print()` function would automatically try to access a function called `print.xyz()` (if it existed) whenever the `print()` function was called with an xyz object as it's argument. Some other functions which use this mechanism are shown in Table 2.

To provide ready access to information about the various classes, the help facility of Splushas been extended to provide information about objects and classes. Typing a question mark ("`?`") before the name of any Splus object will access information about either the object itself (like the `help()` function for functions and data sets) or about the nature of the class or classes to which the object belongs (for data frames, model objects and other entities which belong to a class.) Typing "`?methods(`*`action`*`)`" will display a menu providing access to the help files which implement *`action`* for various classes of objects.

9 Non-linear models

For most of the model fitting procedures in Splus, there is no need to explicitly state the parameters which are to be estimated, because models are expressed as a linear combination of variables or expressions separated by plus signs (`+`), to indicate the linearity. The procedure can then estimate one or more parameters for each term in the model, and assign a name to the parameter based on the variable which was fit. In the case of non-linear models, or for optimization of general functions, obviously this mechanism will no longer work, because of the wide range of functional forms which need to be accommodated. For example, suppose we wish to fit an exponential model to a data set consisting of two variables, x and y; in other words we are assuming that the expected value of y can be expressed as the following function of x:

$$\mathrm{E}(y) = \alpha + \beta e^{\gamma x}$$

Obviously, we must provide the non-linear modeling functions with more information than just the names of the variables to fit; we must explicitly express the model and make the routine aware of the parameters which need to be estimated. To do so, formulas for nonlinear models should explicitly include the parameters being estimated. In addition, starting values for the estimation procedure must be passed to the nonlinear modeling function through an argument named `start=`, which is a list with one named element for each of the parameters to be estimated, with the value of each element representing the starting value for the model fitting process.

The function which performs nonlinear modeling is called `nls`. For example, the model described above could be fit using the following Splus statements:

```
  z <- nls(y~alpha+beta*exp(gamma*x),start=list(alpha=a0,beta=b0,gamma=g0))
```

(It is assumed that the variables `a0`, `b0` and `g0` contain suitable starting values for `alpha`, `beta` and `gamma`, respectively.) As with the other modeling procedures, if the data were stored in a data frame, it's name could be specified using the `data=` argument. It is also possible to store starting values with the data frame, eliminating the need for the `start=` argument to the `nls()` function. The function `parameters()` stores a named list of parameters along with the data frame, creating an object called a parameterized data frame, or `pframe`. You can see the parameters stored in a pframe by calling the function `parameters()` with the pframe as an argument. Assuming that a data frame called `xy` existed, with variables `x` and `y`, the previous statement could be replaced by the following:

```
   parameters(xy) <- list(alpha=580,beta=-180,gamma=-0.16)
   z <- nls(data=xy,y~alpha+beta*exp(gamma*x))
```

The values stored as parameters in the pframe can be used anywhere in the model formula, and will be permanently stored along with the data in the pframe. The print method for pframes will display the values of both the parameters and the variables.

10 Derivatives with Nonlinear Models

In the previous example, no derivatives were passed to the `nls()` function. In such a case, the routine computes numerical derivatives, which may be more costly (in terms of additional function evaluations) than if the analytical derivatives were supplied. In addition, analytical derivatives often help `nls` produce more accurate parameter estimates, especially for more complex functions than the example given. Derivatives of the nonlinear function with respect to each of the parameters being estimated can be passed to `nls()` by using the "gradient" attribute of the formula being fitted. The gradient attribute should be assigned a matrix with as many rows as there are observations in the data set, and each column representing the derivative of the function being fit with respect to one of the parameters. In the exponential example above, we have:

$$y = \alpha + \beta e^{\gamma x} + \text{error}$$

$$\begin{aligned}\frac{\partial y}{\partial \alpha} &= 1\\ \frac{\partial y}{\partial \beta} &= e^{\gamma x}\\ \frac{\partial y}{\partial \gamma} &= \beta x e^{\gamma x}\end{aligned}$$

While we could express the gradient as a single expression, it is more efficient to use a function to calculate the gradient, since there is a common expression ($e^{\gamma x}$) in the derivative with respect to both β and γ. The following function will calculate both the function and the gradient for this example:

```
expfun <- function(x,alpha,beta,gamma)
{
  egx <- exp(gamma * x)
  value <- alpha + beta * egx
  attr(value,"gradient") <- cbind(rep(1,length(x)),egx,beta*x*egx)
  value
}
```

We could then use `nls()` to perform the model fitting, with the call to `expfun()` as the right hand side of the model. Note that the data (in this case, `x`) should be an explicit argument to the function, since evaluations in the function are performed in a separate frame, even if `nls()` is called with a `data=` argument. The Splus statement to perform the fit would now be:

```
      z <- nls(data=xy,y ~ expfun(x,alpha,beta,gamma))
```

The symbol `x` is correctly resolved in this case as being a variable in the `xy` data frame because arguments in the `nls()` call are evaluated with respect to that frame due to the `data=xy` argument to `nls()`. Note again that starting values need not be explicitly stated, because they have been stored in the pframe using the `parameters()` function. You can also use the `deriv` function in Splus to generate a function definition like the one above, with the associated gradient attribute. For simple models like the one described here, `deriv()` will create a function which can be used as is; for more complex models, it may complain that it doesn't know how to calculate a derivative, or it may create a function which needs to be edited before it will work properly. To create the `expfun()` function described above using `deriv()`, we could use the following statements:

```
expfun <- deriv(~alpha+beta*exp(gamma*x),c("alpha","beta","gamma"),
           function(x,alpha,beta,gamma)NULL)}
```

The first argument to `deriv()` is the right hand side of the formula to be fit, preceded by a tilde (~) to let Splus know that it is a formula, and not an expression to be evaluated. The second argument is a character vector containing the names of the parameters which are to be estimated. `deriv()` will create a gradient attribute with one column for each of the parameters named in the vector. Finally, the third argument is a dummy function with a body consisting of `NULL`, showing the argument list which should be used in the function which `deriv()` will create. Once it is established that `deriv()` has successfully created a function, it can be used in exactly the same way as one which was written by hand.

11 General Optimization

Optimization of general functions is performed using the `ms` function. Unlike the other modeling functions discussed here, formulas passed to `ms` will have no argument to the left of the tilde ( ), that is, they have no dependent variable. Starting values for optimization are passed to `ms` in exactly the same way as was done for `nls()`; either through the `start=` argument to `ms`, or by storing starting values in a pframe. It should be noted that `ms` performs minimization, so that if maximization of a function is desired, an appropriate transformation should be used. Usually it is sufficient to multiply the function in question by -1.

As an example of a maximization problem, consider maximum likelihood estimation for a beta distribution with two parameters, a and b. Problems such as these are usually solved by minimizing the negative of the sum of the log likelihood, that is we wish to minimize the quantity F, where

$$F = -\sum_{i=1}^{n} \log(f(a, b|x_i))$$

where n is the number of observations, x_i is the data for the ith observation, and $f(a, b|x)$ is the probability density function for the beta distribution, viewed as a function of the parameters to be estimated, a and b. The probability density function for the beta distribution is available in Splus as the function `dbeta`. Maximum likelihood estimates for this distribution, for a data vector `x` could be obtained through the following function call:

```
z <- ms(~-log(dbeta(x,a,b)),start=list(a=starta,b=startb))
```

Note that the sum does not have to be explicitly entered in the formula passed to `ms`. You must include a tilde (~) before the formula so that Splus interprets the model as a formula instead of evaluating it before passing it on to `ms()`. Thus, the tilde is required, even though there is no dependent variable. Information about derivatives is provided to `ms()` in the same way as `nls()`; if you are calculating analytical derivatives, you should set the gradient attribute of the returned value equal to a matrix with as many rows as there are observations in the data, and one column for each parameter being estimated, representing the partial derivative of the function being minimized with respect to that parameter. The `deriv()` function can also be used to help make this task easier.

12 Trellis Graphics

One of the most appealing features of Splus has always been the ability to produce high quality graphics, with complete control over most aspects of the graph. This has been achieved by using combining three separate elements: a set of graphical parameters (set with the `par` function) to control the overall appearance of the graph, a collection of high-level graphing functions (like `plot` or `hist`) which produce a complete graph with a single function call, and auxiliary routines, known as low-level graphing functions, which can be used to augment an existing display. Taken together, these elements are known as the core graphics of Splus.

But the core graphics have their limitations. The ability of different display devices makes it difficult to consistently produce graphics that look the same on different devices, like the screen and a printer. Displaying several graphs on the same page often requires a fair amount of programming, and getting control of some fine points of the appearance of graphics generally required a level of understanding of the core graphics which most users simply don't have. Another problem is that the core graphics routines have always been different from other Splus functions in that they did not correctly return an object representing the graphics which they produced. The Trellis graphics system solves these problems, and at the same time uses the formula notation described in Section 5 to choose which variables will be used to construct a graph, thereby focusing on the relationships which are being studied, instead of an arbitrary assignment to parameters. It should be mentioned, however, that all the Trellis functions do their work using the core graphics functions, so, at the lowest level, the user still has total control of the way the graphics will look.

Central to the overall design of the Trellis functions is the concept of a conditioning plot. A conditioning plot is actually a collection of plots, each with common axes, which represent the relationships among variables for several subgroups of the data. For example, to study the effects of a treatment on survival rates using several different treatments, we might produce a plot of time versus survival, conditioned on treatment. This means that, for each treatment, a separate graph of time versus survival will be produced. The Trellis functions will lay out the various plots in an ordered grid (hence the name trellis), appropriately labeling them and constructing their axes identically to promote easy and valid comparisons, using a set of parameters which was specifically chosen for the output device in use. Such a graph might be produced by the function `xyplot`, given a formula

```
Survival ~ Time | Treatment
```

The vertical bar (`|`) is used to delineate the relationship being studied from the so-called given variables, which are used to specify the subsets of the data which will be displayed. Two given variables can also be specified. In this case, the y-dimension of the grid will represent the varying levels of the first given variable, and the x-dimension will represent the levels of the second given variable. It should also be pointed out that, when no given variables are used, the Trellis functions will produce single panel displays which are often more attractive than those produced by the corresponding high-level core graphics routines. A list of some of the Trellis functions is provided in Table 12. One good way to learn about these functions is to view their help pages from within Splus. In addition, two other help files which you may find useful are `trellis.examples` which lists the names of a variety of functions which will produce examples of the various displays, and `trellis.args` which describes the arguments which are common to the Trellis functions.

Univariate Data	
`barchart`	Bar plot
`bwplot`	Box-and-whisker plot
`densityplot`	Kernel Density plot
`dotplot`	Dot plot
`histogram`	Histogram
`piechart`	Pie chart
`qqmath`	Quantile plot against specified distribution
`stripplot`	One-dimensional scatter plots
Bivariate Data	
`qq`	Quantile-Quantile plot for comparing 2 distributions
`timeplot`	Time Series Plot
`xyplot`	Scatter Plot
Trivariate Data	
`contourplot`	Contour Plot
`levelplot`	Level Plot
Multivariate Data	
`splom`	Scatterplot Matrix
`parallel`	Parallel Coordinate Plot
3-D Displays	
`wireframe`	Wireframe display (regular grid points)
`cloud`	3-D Point Cloud (irregularly spaced data)

Table 3: Trellis Display Functions

13 Using Trellis

One convenient feature of the Trellis system is that it will automatically open an appropriate display device when you first call any of the Trellis display functions. (If you want to override the default choice, or specify special parameters to the display device, use the function `trellis.device`.) If you are used to the core graphics of Splus, it may take a while to get used to the fact that the usual arguments to the `par` function will probably not do what you expect. The Trellis graphics are designed to make intelligent choices about most of the usual graphics parameters (margins, axes, scales, etc.), so you shouldn't need to change most of the values from their defaults. If you do need to make changes, they can be done through the function trellis.par.set. To see a list of the parameters you can control, examine either trellis.settings or its help file; to examine the value of a specific parameter use the function trellis.par.get. Also keep in mind that, if the default graphics device changes, many of the Trellis parameters will also change, since the display functions have been customized for different display types.

14 Trellis Objects

While the main goal of most Trellis commands is to produce a plot, all of the functions in Table 12 return an object of class trellis. When these functions are called without assigning their output to an object, they follow the usual default of being "printed"; in the case of graphical functions, that means that they are displayed on the current output device. So the Trellis functions can be assigned to objects and displayed at a later time, perhaps using a different output device. This is especially useful because the print.trellis function provides a number of options for combining several Trellis objects on one display, similar to the mfrow or mfcol graphical parameters of the core graphics system. In addition, the update function allows you to change your plot by modifying any of the arguments which were used when the plot was first created.

15 Formulas for Graphics

All of the Trellis display functions expect their first argument to be a formula representing the relationship to be graphically displayed. Since the Trellis functions all accept a data= argument, if you are working with a data frame, you can simply use the variable names, and specify the data frame just once with the data= argument. Perhaps the simplest case is plotxy which plots a bivariate scattergram; the function call

```
xyplot(y~x)
```

produces a plot with y on the vertical axis and x on the horizontal axis. Most other Trellis functions follow this general rule. For example, to produce a histogram of a numeric variable, you would use a call like:

```
histogram(~variable)
```

Since no variable is represented on the vertical axis of a histogram, the formula has no left hand side. Similarly, a box-and-whiskers plot for a variable, using a different vertical plot for each of several groups would be invoked through a call like

```
bwplot(group~variable)
```

because the variable's values will be plotted along the horizontal axis, and the different levels of group will be plotted along the vertical axis.

Simple formulas like those above will produce a single plot on the page, similar to plots produced by the older plot, hist, and boxplot functions. The true power of the Trellis graphics becomes clear when you use the vertical bar (|) to specify one or more given variables. The given variables are used to define subsets of the data which are graphed separately, in interlocking plots, all of which have identical scales.

Returning to the apple data set, suppose we wish to investigate the relationship between Ca the calcium concentration in the soil, and Bitter, a measure of the apple's bitterness. A logical tool would be a scatterplot of Bitter versus Ca. The following command produces the plot displayed in Figure 1.

```
xyplot(Bitter~Ca,data=apple)
```

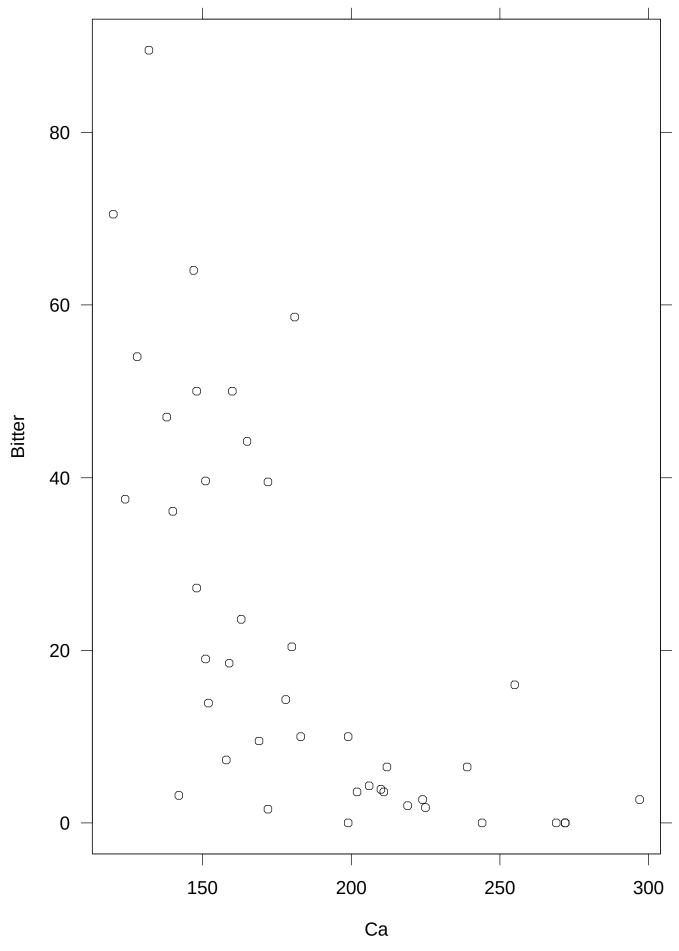


Figure 1: Plot of Bitter vs Ca for entire apple data set

To investigate whether this relationship changes with different `Treatment`s, we could introduce `Treatment` as a conditioning variable by including it in the formula after a bar (`|`):

```
xyplot(Bitter~Ca|Treatment,data=apple)
```

The resulting plot, shown in Figure 2, clearly shows that the relationship between `Bitter` and `Calcium` is only present in the test groups; no such relationship exists within the control group. The shaded areas on the labels of the individual plots in Figure 2 are intended as a visual cue to show the relative values of the conditioning variable. For a classification variable such as `Treatment` in this example, the shading is probably of little value. In Section 19, methods of modifying these labels will be discussed.

16 The `panel=` function

In Section 15, scatterplots for the relation of `Ca` and `Bitter` were compared for four different values of `Treatment`. To facilitate the comparison of these plots, it would be helpful to place a line representing the

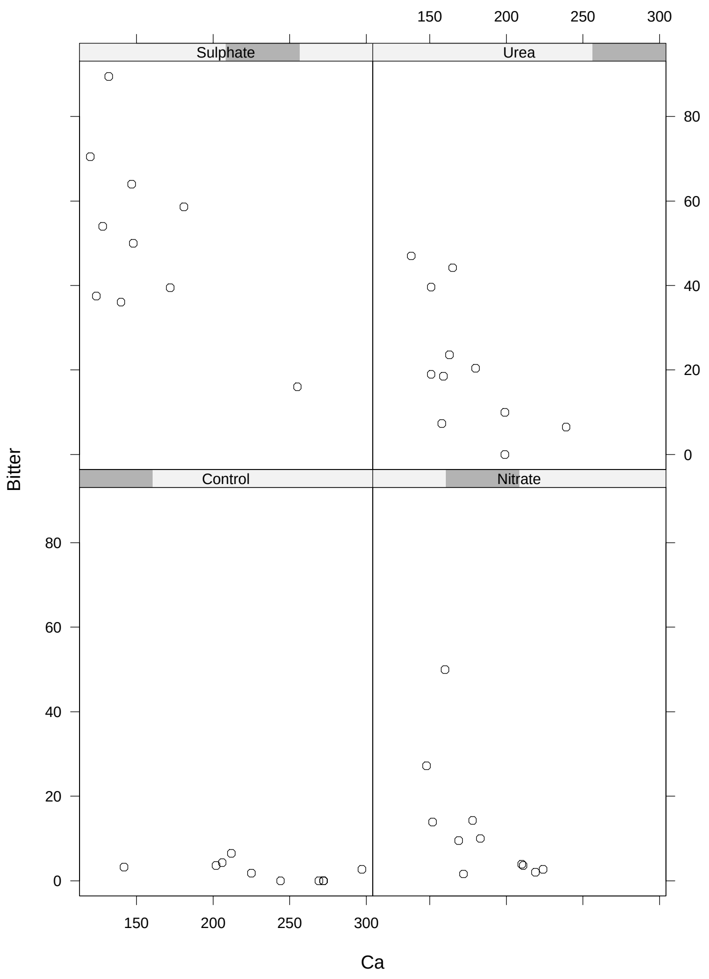


Figure 2: Plot of Bitter vs Ca conditioned by Treatment

least squares (or some other) fit for these points. In the Trellis system, you can specify exactly what gets plotted for each subset of data in the `panel=` argument. Since each of the Trellis functions has a corresponding default panel function named `panel.`*`functionname`*, panel functions often consist of a call to the default function, followed by some additional graphics commands. If this model does not suffice, then the default panel function can be used as a model for a customized panel function. In the present example, we need to add a regression line to the plots, which can be easily done with a call to the function `abline`, or the corresponding trellis function `panel.lmline`. Examination of the default panel function `panel.xyplot`, obtained in Splus by simply typing the function name, indicates that, among other arguments, it accepts two arguments `x` and `y`, representing the data to be plotted for each subset. So to add a regression line to each of the plots in Figure 2, the following command could be used:

```
xyplot(Bitter~Ca|Treatment, panel=function(x,y){
                                  panel.xyplot(x,y)
                                  panel.lmline(x,y)}, data=apple)
```

The resulting plot is shown in Figure 3.

Two other useful functions for customizing panel functions are `panel.grid` and `panel.fill`.

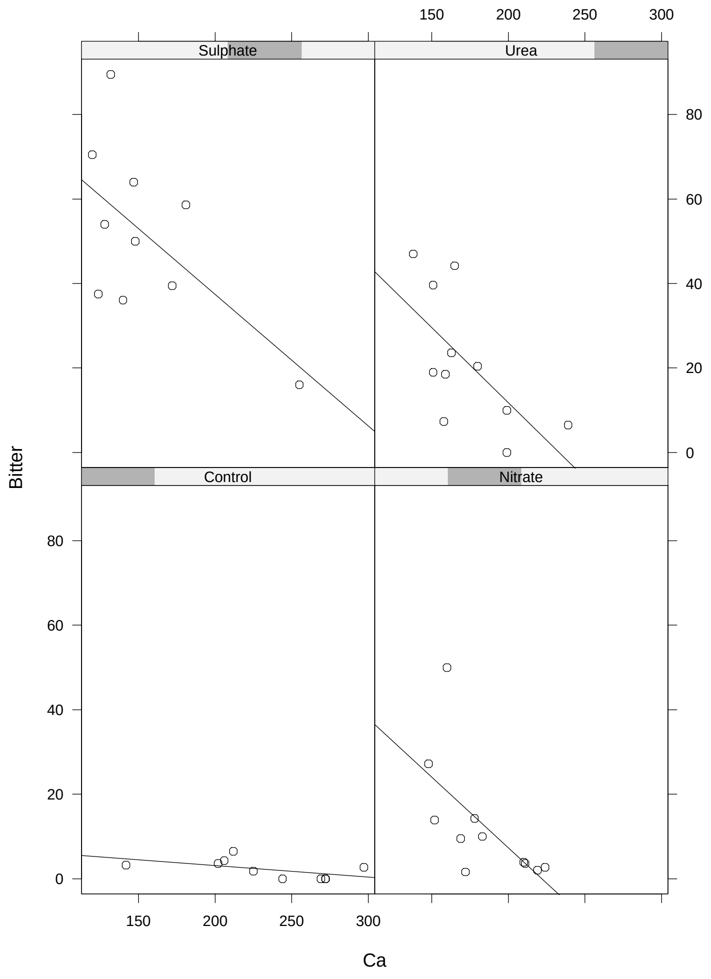


Figure 3: Plot of Bitter vs Ca condition by Treatment with regression line

17 Conditioning on Numeric Variables

In the previous example, where a plot was conditioned on a categorical variable, it was easy to decide how to divide the data to produce the various panels in the plot; simply create a separate panel for each level of Treatment. But for numeric variables, this choice is not so clear cut. To make it easier to generate groupings for numeric variables, two functions are provided: `equal.count` and `shingle`. Both functions return an object of class `shingle`, which is simply a collection of data with a set of intervals which maps the (continuous) variable into a small number of groups. The name shingle derives from the fact that the default behavior of both functions is to create overlapping groups, so that data points may fall into more than one grouping. (Contrast this with the `factor`s introduced in Section 3, where there is a one-to-one mapping of data to a factor value.) This overlap turns out to be very useful when examining conditioning plots. The function `equal.count` divides your data into groups with approximately equal numbers in each group, and the fraction of overlap specified by the `overlap=` argument, which defaults to 0.5. This means that when using the default value half of the data will be shared with adjacent intervals. As an alternative, you can specify a number greater than one, which will be interpreted as the number of points shared with adjacent intervals. Using the apple data as an example, suppose we wish to compare the distribution of the variable `Mg` (magnesium) for different values of `Ca`. Since `Ca` is a continuous numeric variable, it needs to be converted to a shingle before it can be used as a conditioning variable. Suppose we want to produce six histograms, one for each of six equal-count overlapped groups based on the value of `Mg`. The following commands produce the plot shown in Figure 4:

```
histogram(~Mg|equal.count(Ca),data=apple)
```

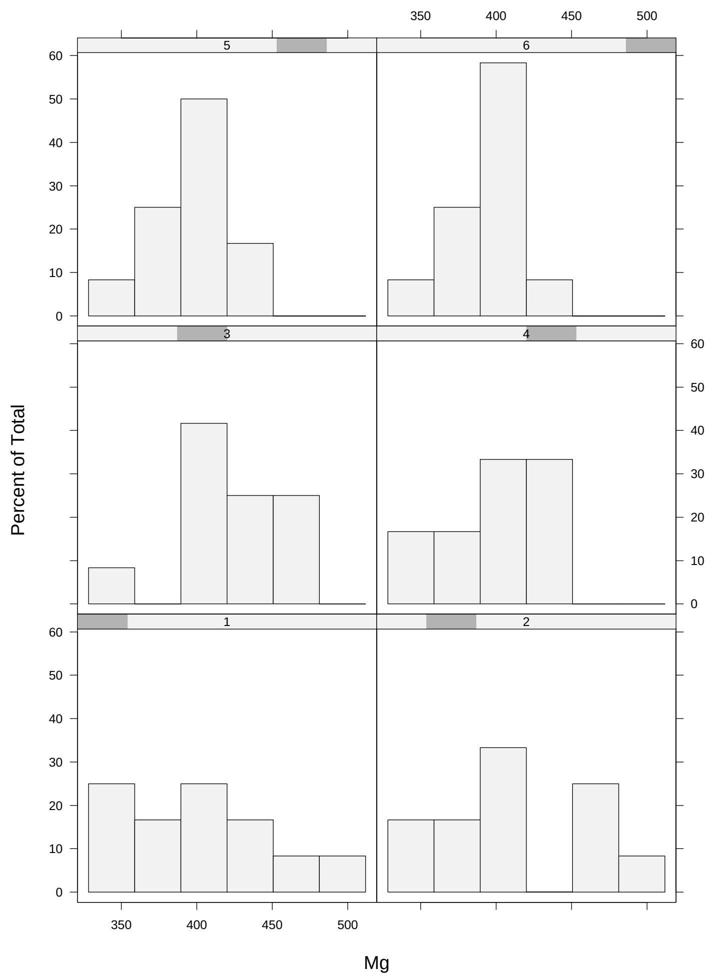


Figure 4: Histogram of `Mg`, conditioned by `Ca`

The shaded areas on at the top of each panel provide a visual cue as to which shingle of the data is being displayed; the plot based on data corresponding to the smallest values of the conditioning variable are represented in the panel with the leftmost shaded area within the strip, and the plot based on the data representing the largest value of the conditioning variable are represented in the plot with the shaded area on the right.

18 Multiple Conditioning Variables

You are not limited to a single conditioning variable when creating trellis plots; you can condition on two variables by joining their names after the bar of the formula with an asterisk (`*`). Two strips will then appear over each panel of the display, representing the levels of the two conditioning variables. As an example, Figure 5 is a quantile-quantile plot for the variable `Bitter` from the `apple` data set, conditioned by shingles formed from the variables `Ca` and `Mg`. This plot was produced with the statements

```
MgConc <- equal.count(apple$Mg,n=3)
CaConc <- equal.count(apple$Ca,n=3)
qqmath(~Bitter|CaConc*MgConc,dist=qnorm,data=apple,
       panel=function(x,y){
                  panel.qqmath(x,y)
                  panel.qqmathline(y,dist=qnorm)
                  })
```

To provide sufficient points for each panel, the number of intervals was limited to three for each shingle. The shingles were stored in named objects so that the strips above the plots would be appropriately labeled. As in the previous example, a reference line was added to the plot using the `panel=` argument to `qqmath`, this time invoking the function `panel.qqmathline` in addition to `panel.qqmath`. The shaded areas in the strips provide a relative notion of the range of the conditioning variables required to produce shingles containing (approximately) equal numbers.

19 Customizing Trellis Plots

In addition to the `panel=` argument to the trellis plot commands there are a number of other ways that you can customize trellis plots. The arguments `xlab=` and `ylab=` can provide labels for the x- and y-axes in the usual way, or they can be used to pass a list containing named graphics parameters (like `col`, `cex`, etc.) to control the way the labels are printed. The title at the top of each page of output can be controlled in a similar way through the argument `main=`.

The style of the strip on top of each panel can be controlled through the `strip=` argument. This argument should be a function which passes its arguments to the supplied function `strip.default`, with the necessary parameters changed to meet your needs. Perhaps the most common use of this parameter is to change the overall style of the strip. Figure 6 illustrates the five available types of panels, using the levels of `apple$Treatment` as an example. When the conditioning variable is a shingle, the interval number appears in place of the factor level; the name of the shingle can be included in the strip by setting the `strip.names=` argument of `strip.default` to `c(T,T)`.

Thus, to change from the default (`style=1`), to `style=5`, an argument of the form

```
strip = function(...)strip.default(...,style=5)
```

should be passed to the appropriate trellis function. Other arguments to `strip.default` can be modified in a similar way. Further modifications to the way labels print on strips can be achieved through the `par.strip.text=` argument of the trellis functions.

20 Advanced Topics

Many more functions are arguments are available within the Trellis system to provide more control over the appearance of your plots. The `prepanel=` argument allows you to specify a function which is called

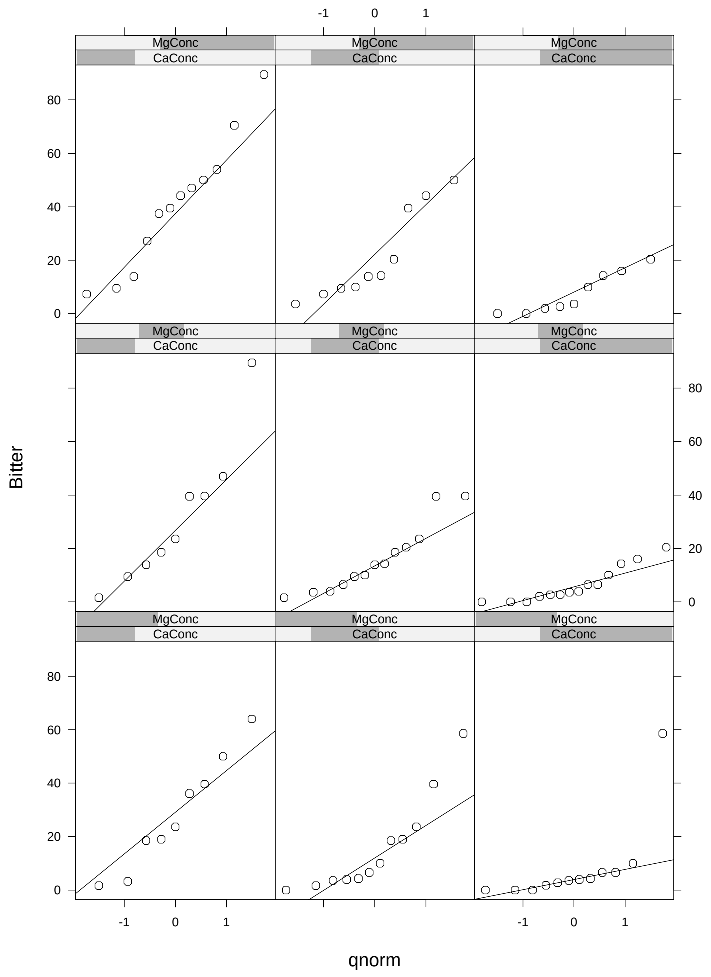
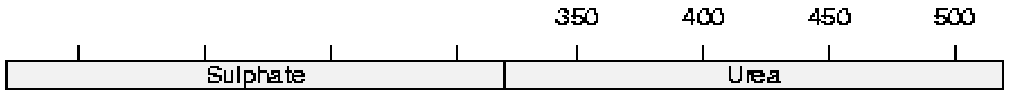
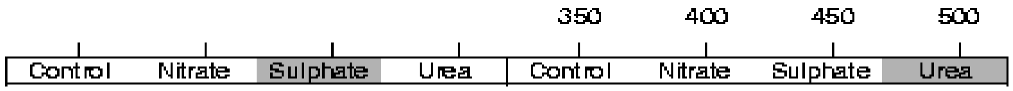


Figure 5: Q-Q plot of `Bitter`, conditioned by `Ca` and `Mg`

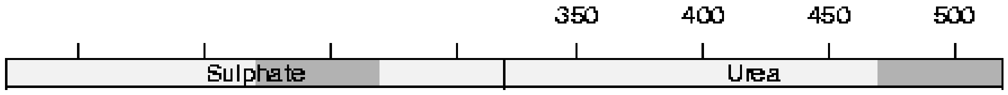
prior to plotting, to allow you set up axes and specify aspect ratios of the plots which will be produced. The `subscripts=` argument, when set to `T`, adds a third argument to the panel function, representing the subscripts of the selected observations for the current panel. This can be useful, for example, if plotting symbols based on some other variable in a data frame are to be used. A more complete description of the Trellis graphics, including an extended User's Guide, is available at the Bell Labs web site:
`http://netlib.bell-labs.com/cm/ms/departments/sia/project/trellis/`.



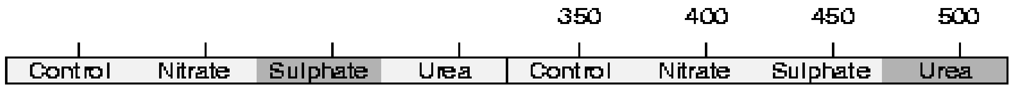
`style=1` - The active level name appears on a background colored strip with no shading



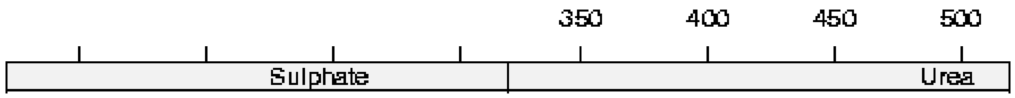
`style=2` - All level names appear in the strip - the active one is shaded



`style=3` - Level names appear in the strip, along with shading to serve as a visual cue which level is represented



`style=4` - Like `style=2`, but the strip is colored using the background color



`style=5` - Like `style=1`, but the placement of the label serves as a visual cue of what level is represented

Figure 6: Different strip styles available under Trellis