An Introduction to Splus

Phil Spector
Statistical Computing Facility
University of California, Berkeley

November 10, 1999

1 Background

The S programming language was developed at AT&T Bell Labs, for internal use by a group of statisticians who wanted an interactive, graphics environment that encouraged development of new statististical techniques. As the program spread through universities, many people appreciated the same qualitites as the AT&T researchers, and the language became quite popular among academic statisticians. The program received a major rewrite in the late 1980s, eliminating an awkward macro language, and unifying some of the basic concepts. Finally, in the early 1990s, Statistical Sciences in Seattle Washington began distributing a commercial version of the S language known as Splus; that company was bought out by Mathsoft who now distributes Splus.

2 Strengths and Weaknesses

2.1 Strengths

- highly extensible and flexible
- implementation of modern statistical methods
- strong user community
- moderately flexible graphics with intelligent defaults

2.2 Weaknesses

- slow or impossible with large data sets
- non-standard programming paradigms
- runs on limited platforms

3 Basics

Splus is a highly functional language; virtually everything in Splus is done through functions. Arguments to functions can be named; these names should correspond to the names given in the help file or the function's definition. You can abbreviate the names of arguments if there are no other named arguments to the function which begin with the abbreviation you've used. If you don't provide a name for the arguments to functions, Splus will assume a one-to-one correspondence between the arguments in the function's definition and the arguments which you passed to the function.

To store the output of a function into an object, use the special assignment operator `<-`, not an equal sign. For example, to save the value of the mean of a vector `x` in a scalar called `mx`, use

```
mx <- mean(x)
```

This statement is read as "mx gets mean of x".

If you forget to save the output of a function this way, the answer will be temporarily available in an object with the name `.Last.value`.

Typing the name of any Splus object, including a function, will display a representation of that object. You can explicitly display an object with the functions `print` or `cat`. Splus provides online help through the `help` function, or by simply preceding a function's name with a question mark (`?`). If you are running Splus in an Xwindows environment, the command

```
help.start()
```

will provide a graphical interface to the help system. You should quickly get in the habit of consulting the help files while you use Splus, because many of the functions have a number of useful optional arguments which might not be apparent at first glance.

Splus supports a number of different data structures, accomodating virtually any type of data. At the simplest level, Splus supports vectors and matrices. These, however are just one- and two-dimensional examples of the more general concept of an array; Splus supports arrays of virtually unlimited dimensions. In addition, Splus implements the more traditional "observations and variables" format of rectangular data sets through objects known as data frames, where character and numeric variables can be freely mixed. Finally, Splus has a very general data structure known as a list, which can hold virtually any structure of data imaginable.

Besides numbers and character strings, the symbols `T` and `F` (or `TRUE` and `FALSE`) are reserved in Splus to represent the logical values true and false, respectively.

The objects you create during an Splus session are stored in a directory known as the working directory. By default the working directory is set to the directory `.Data` in your home directory. Note that, since this directory's name begins with a period, it will not be displayed by the UNIX `ls` command unless the `-a` option is given. You can create separate collections of objects for different projects by having `.Data` subdirectories in different directories; if Splus finds a directory called `.Data` in the current directory, it will override its usual default of using the `.Data` directory in your home directory.

In addition to your working directory, Splus searches for data and functions in a number of system directories. You can see the names of these directories by using the `search` function. Note that just typing the name `search` will display a print representation of the `search` function; to invoke the function with no arguments you must type

```
search()
```

You can see the contents of any directories in your search path through the `objects` function. With no arguments, it will list all the objects in your working directory. The optional argument `where` allows you to specify other directories in your search path by providing the index of the directory as provided by the `search` command. The optional argument `pattern` will restrict the listing of objects to those whose names contain the string or regular expression specified by the `pattern` argument. Quoted strings in Splus can be surrounded by either single (`'`) or double (`"`) quotes. Thus, the command

```
objects(p="dat")
```

will list those objects in your working directory whose names contain the string `dat`.

You can enter UNIX commands from an interactive session by preceding them with an explanation point (`!`). From inside source files or functions, you can pass a UNIX command to the function `.System` for execution.

You can interactively debug Splus functions by passing the function call of interest to the `inspect` function; once inside the debugger, type `help` for more information.

Optionally installed functions are organized into sections known as libraries. To see which libraries are available on a given installation of Splus, type the command `library()`; once you've found a library of interest, you can make the functions in the library available with a command of the form `library(libraryname)`. The command `library(help=libraryname)` will list the functions which are available, and the `help` command can then be used in the usual way to get more information about individual functions in the library.

4 Specific Tasks

4.1 Entering and Exiting the Program

To start an interactive session with Splus, type

```
Splus -e
```

at the UNIX prompt. The `-e` flag enables command line reediting; you can redisplay and edit previous commands using either `vi` or `emacs` keystrokes. Inside of Splus, the interactive prompt is a single greater-than sign (>).

Splus obeys standard UNIX redirection, so you can execute source files in the usual way, with a command like this one at the UNIX prompt:

```
Splus < infile >& outfile
```

The Splus commands to be executed would be in the file `infile`, and output would be sent to `outfile`. The `BATCH` command of Splus packages this capability in a slightly different fashion. Typing

```
Splus BATCH infile outfile
```

at the UNIX prompt would have a similar effect as the previous command.

To execute Splus statements from a file from within an interactive session, you can use the `source` command. Type

```
source("infile")
```

at the Splus prompt to execute the commands in the file `infile`.

To exit an interactive Splus session, type `q()`. Note that just typing the `q` without the parentheses will simply display a text representation of the `q` function.

4.2 Reading Data

The `c` function (mnemonic for combine) can be used to read small amounts of data directly into an Splus object. For example, you could create a vector called `x` containing five numbers by using a statement like the following

```
x <- c(12, 19, 22, 15, 12)
```

To read white-space-separated data into a vector, use the function `scan`. The `sep` argument can be used for separators other than the default of white space. For example, to read the data in the file `datafile` into a vector called x, use

```
x <- scan("datafile")
```

With no filename argument, `scan` reads your input from standard input; terminate the data entry with a blank line. To read character data, use the `what` argument as follows:

```
chardata <- scan("charfile", what="")
```

Often the data you are reading from a file or entering at the keyboard is a matrix. The function `matrix` can be used to reshape the elements of a vector into a matrix. Since the output of one Splus function is suitable as input to another Splus function, the calls to `scan` and `matrix` can be combined. Matrices in Splus are internally stored by columns, so if your data is arranged by rows (as is usually the case), you must set the `byrows` argument to the matrix function to `T`. Suppose that the file `matfile` contained a 10×5 matrix, stored by rows. The following statement would read the matrix into an Splus object called `mat`

```
mat <- matrix(scan("matfile"),ncol=5,byrow=T)
```

In addition to the ncol argument, there is also an nrow argument which could have be used (with a value of 5 in the previous example). As shown in the example, if one or the other of these two arguments is missing, Splus will figure out the other based on the number of input items it encounters. You can also provide descriptive labels for rows and columns using the dimnames function.

If your data has a mix of numeric and character variables, you will probably want to store it in a data frame. To read data from a file directly into a data frame, use the function read.table. To use read.table, all the variables for a given observation must be on the same line in the file to be read. If the optional argument headers=T is given, then the first line of the file is interpreted as a set of names to be used for the variables in the file, otherwise default names of V1, V2,, etc. will be used. The function data.frame can also be used to create data frames directly from other Splus objects.

It should be mentioned that Splus does not contain a wide range of functions to handle input data. If your input data is not suitable for scan or read.table, you may need to consider preprocessing the data with a program like perl or sas before reading it into Splus.

4.3 Storing Data Sets

All of the objects which you create during your Splus session are automatically stored in your current directory. You can remove them by using the rm() function from inside of Splus.

A running session of Splus is always aware of new objects created in the .Data directory during the current session. But if two different sessions of Splus are sharing the same .Data directory, they will not be aware of objects created by the other Splus session. While the easiest way to avoid problems caused by this fact is to not run two sessions sharing a common .Data directory, the synchronize function can be used to make Splus aware of objects placed in the .Data directory by other Splus sessions. This function needs a number representing the database which needs to be synchronized; since it will be the default .Data directory, the appropriate statement is

```
synchronize(1)
```

4.4 Accessing and Creating Variables

The basic tools for accessing the elements of vectors, matrices and data frames are subscripts; in S, subscripts are specified using square brackets ([ ]); parentheses are reserved for function calls. You can refer to an entire row or column of a matrix by omitting the subscript for the other dimension. For example, to access the third column of the matrix x, you could use the expression x[,3]. Keep in mind that if you use a single subscript, Splus will interpret it as the index into a vector created by stacking all the columns of the matrix together.

If you've assigned row or column names to a matrix with the dimnames function, you can also use character strings to access parts of a matrix. (Data frames automatically have row and column names assigned when they are created.) If you used statements like the following to create a matrix:

```
mat <- matrix(c(5,4,2,3,7,8,9,1,6),nrow=3,byrow=T)
dimnames(mat) <- list(NULL,c("X","Y","Z"))
```

then you could refer to the second column of the matrix as either mat[,2] or mat[,"Y"].

While the above techniques will work for data frames as well as matrices, there is a simpler way to refer to variables by name in a data frame, namely separating the data frame's name from the name of the variable with a dollar sign ($). For example, if a data frame called soil contained variables called Ca, K and pH, you could access the variable pH as soil$pH. Note that, like other identifiers in Splus, variable names in a data frame are case sensitive. Alternatively, you can use the attach command to make a data frame part of your search path, and refer to variable names directly. The dollar sign notation can also be used to extract named elements out of an Splus list.

You can create new objects, which will be stored in the current .Data directory with the assignment operator (<-) mentioned in Section 3. For example, to create a variable z which would be the ratio of two variables x and y, you could use the statement

```
z <- x / y
```

Virtually all operators and functions in Splus will operate on entire vectors and matrices in a single call. In the above example, if x and y were each vectors of length n, then z would also be a vector of length n.

4.5 Subsetting Data

A variety of subscripting expressions can be used to extract parts of Splus matrices and data frames. As mentioned previously, you can specify a subscript for either rows or columns to extract entire rows and columns of a matrix. You can also provide a vector of row or column numbers (or names, if dimnames were assigned to the matrix), to extract subsets of a matrix or data frame. You can also provide a vector of row or column numbers to extract subsets of a matrix or data frame. The colon operator (:), which generates sequences, is often useful in this regard; it generates a vector of integers, separated by 1, from its first argument to its second argument. For example, to extract the first, third and fourth variables for the first 10 observations of a data frame or matrix called data, you could use the following expression

```
data[1:10,c(1,3,4)]
```

The seq function provides additional capabilities for generating sequences. A further useful feature of Splus numeric subscripts is that negative subscripts represent all values except those specified in the subscripts. So in the previous example, if we wanted all the columns of data, except for the first, third and fourth, we could use the expression

```
data[,-c(1,3,4)]
```

Subscripts with a value of 0 are simply ignored.

If dimnames were assigned to the matrix, a vector of names can be substituted for the vector of numbers in the example just given. A vector of names can be composed using the c function by surrounding the names with double or single quotes.

Logical subscripts provide a powerful tool for subsetting data in Splus. When using logical subscripts, they must be the same length as the object which is being subscripted; those elements in the subscripted object corresponding to values of TRUE will be extracted. Thus, to select all the rows of the matrix data for which the third column is less than 10, the following expression could be used

```
data[data[,3] < 10,]
```

5 Missing Values

The value NA is used to represent missing values for input in Splus. An NA can be assigned to a variable directly to create a missing value, but to test for missing values, the function is.na must be used.

Splus propogates missing values throughout computations, so often computations performed on data containing missing values will result in more missing values. Some of the basic statistical functions (like mean, min, max, etc.) have an argument called na.rm, which, if set to TRUE, will remove the NAs from your data before calculations are performed. In addition, the statistical modeling functions (like aov, glm, loess, etc.) provide an argument called na.action. This argument can be set to a function which will be called to operate on the data before it is processed. One very useful choice for the na.action argument is na.omit. This function (which can be called idependently of the statistical modelling functions) will remove all the rows of a data frame or matrix which contain any missing values.

6 Graphics

There are basically two systems for producing graphs in Splus. The first, sometimes refered to as the core graphics, consists of functions similar to those found in most graphical software - vectors or matrices are

passed to these functions, along with options and other information, and a graph is either created or additions are made to an already created graph. The second system, know as Trellis graphics, uses the statistical modelling language as introduced in Section 8.2 to inform Splus about the graph you wish to produce. In addtion, Trellis graphics provides a simple mechanism for producing several plots on a page to facilitate side-by-side comparisons. This section will discuss the use of the core graphics system; more information on Trellis graphics are presented in Section 9

The first step in using the core graphics in Splus is to choose an appropriate device driver. When viewing graphics on an Xwindow terminal, the `motif()` function should be called. This will open a window on the screen, which, along with the display area, provides a button which will send your graph to the printer. Other devices include `tek4014`, for Tektronix emulation and `printer` for ASCII graphics. Use the command `help(Devices)` to get a complete list of supported devices. To save PostScript output in a file, use the `postscript()` device driver.

There are two types of graphics routines within Splus' core graphics, high level and low level. The high level functions produce a complete plot from your data, drawing appropriately scaled axes and including labels and titles which you provide to the function. Examples of high level functions include `barplot`, `boxplot`, `contour`, `coplot` (conditioning plots), `hist`, `pairs` (scatterplot matrix), `persp` (3-dimensional perspective), and `plot`. Low level routines, which provide finer control over the details of existing plots include `abline` (drawing regression lines), `axes` (for titles and labels), `axis` (for custom axes), `legend`, `lines`, `points`, `polygon`, `symbols`, and `text`. In addition to the graphics functions, a set of graphics parameters further controls the appearance of your graph. The graphical parameters can be set globally using the `par` function, or they are accepted as arguments to many of the other plotting functions to cause a temporary change. One graphics parameter which is often useful is `mfrow`, which determines the arrangement of multiple figures on a page. For example, the Splus statement

```
par(mfrow=c(3,2))
```

would result in six plots being placed on the page, with three rows of two columns each. The plots would be placed by rows; a similar function, `mfcol` defines the arrangement, put plots the multiple figures by columns. Keep in mind that you can not set parameters or call any graphics functions until a device driver has been loaded as described at the beginning of this section.

7 Programming

Most functions and operators in S will operate on entire vectors, the most efficient programming techniques in S are ones which utilize this approach. S does provide loops for more traditional programming, but they tend to be inefficient, especially for large problems. The `for` loop is a basic tool which can be used for repetitive processing. It takes the form

`for(`*`name`* `in` *`values`*`)` *`expression`*

where *`name`* is a variable which will be set equal to each element of *`values`* inside of *`expression`*. If *`expression`* contains more than one statement, the statements must be surrounded by curly braces (`{ }`). The `for` loop works for lists as well as vectors.

Logical subscripts, introduced in Section 4.5, can be used on the left hand side of an assignment statement to avoid the use of loops in many cases. A simple, but useful, example is replacing occurences of a particular value with missing values (`NA`). For example, a double loop could be used to replace all occurences of 9 in a matrix with `NA`:

```
for(i in 1:nrow(x))for(j in 1:ncol(x))if(x[i,j] == 9)x[i,j] <- NA
```

However, using logical subscripts, the following single statement is much simpler and far more efficient:

```
x[x == 9] <- NA
```

Many functions accept vectors as arguments, so loops can often be avoided by using a vector argument. For example, suppose we wish to form a vector with each of the numbers from 1 to 5 repeated 6 times. One approach would be to use a for loop:

```
result <- NULL
for(i in 1:5)result <- c(result,rep(i,6))
```

However, the same result can be acheived by using vector arguments:

```
result <- rep(1:5,rep(6,5))
```

It is worth noting that this gives a very different value from `rep(1:5,6)`:

```
> rep(1:5,6)
 [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

When processing each row or column of a matrix or data frame, the function `apply` can be used to eliminate the need for loops. For example, to calculate the sum of each row of a matrix called `mat`, the following call to `apply` could be used

```
rowsums <- apply(mat,1,sum)
```

The `1` in the apply call refers to processing by rows; a `2` results in processing by columns. The third argument to `apply`, in this case, `sum`, is a function which will be applied to each row or column of the matrix in turn. A similar function, `tapply`, will apply a function to different subsets of a vector based on the value of a second vector. For example, suppose the vector `score` represents tests scores in an experiment, and the vector `group` indicates which group each of the observations came from. To calculate the means for each group, `tapply` could be called as follows:

```
group.means <- tapply(score,group,mean)
```

If the function being passed to `apply` or `tapply` requires more than one argument, the additional arguments can be included in the argument list after the name of the function being passed.

For functions like `apply` and `tapply`, it is often useful to construct a function to be applied to each row or column of a matrix, if the function you need is not otherwise available. Often a simple function is all that is needed, and you can pass the function definition to `apply` or `tapply`. For example, suppose we have a 2 column matrix called `meanvar` whose rows contain a mean and corresponding variance, and we wish to generate a vector of random numbers from normal distributions with these means and variances. We could use `apply` by writing a function which calls the S function `rnorm` with appropriate arguments as follows:

```
rvars <- apply(meanvar,1,function(x)rnorm(1,x[1],x[2]))
```

The variable `x` in the function definition is a dummy variable which will take as its value each row of the matrix in turn. The returned value, `rvars`, will be a vector with as many elements as `meanvar`, and the `i`*th* element of `rvars` will be a random number from a normal distribution with mean equal to `meanvars[i,1]` and variance equal to `meanvars[i,2]`.

8 Statistical Functions

8.1 Descriptive Statistics

For descriptive statistics, Splus has individual functions for most common statistics. Interestingly, there is no built-in standard deviation function; you need to use the square root of the variance. The `summary` function, when used on a numeric vector, will provide the minimum, the maximum, and the first, second and third quartiles. The function `stem` provides a stemleaf diagram, along with a few descriptive statistics. For categorical variables, the `table` function can provide both one-way and multi-way contingency tables. The `crosstabs` function presents similar information in a possibly more familiar form.

8.2 Statisical Modeling

Splus provides a number of functions for statistical modeling, including a few for "modern" techniques which may not be available elsewhere. These functions include `lm` (linear models), `aov` (analysis of variance), `glm` (generalized linear models), `gam` (generalized additive models), `coxph` (Cox proportional hazard models), `loess` (local regression smoothing), `tree` (Recursive Tree models for classification or regression) and `nls` (Non-linear regression.) For a more complete list, choose "Statistical Models" from the right-hand menu displayed by the `help.start()` menu.

For most statistical modeling applications, there is a clear distinction between variables which enter the model as *factors* (discrete categorical variables), and *regressors* (continuous numeric variables). For example, in an analysis of variance, regressor variables are entered directly into the design matrix, while factor variables are entered as one or more columns of dummy variables.

For variables you have identified as factors, Splus will automatically generate appropriate dummy variables, and most of the functions which display the results of the analysis will treat these groups of dummy variables as a single effect. To let Splus know a variable is a factor, use the function `factor`. For example, to change a variable called `group` to a factor, use

```
group <- factor(group)
```

If the variable group were stored in a data frame called `mydata`, a similar call would be used:

```
mydata$group <- factor(mydata$group)
```

Using a data frame for statistical modeling is especially easy, because all the modeling functions accept an argument called `data`; when a data frame is given as a `data` argument, Splus will resolve variable names in that data frame, making your formulas more readable, and eliminating the need to repeat the data frame name over and over.

To construct a formula in Splus, you use the tilda (~), with your dependent variable on the left-hand side and your independent variables on the right-hand side. You can also construct other terms for the right hand side using the symbols in Table 1 below. Note that if you want any of the model operators in the table to behave in their usual fashion inside a formula, the term including the operator should be passed to the `I()` function.

Table 1: Operators Used in Model Formulas

Operator	Usual Meaning	Meaning in Formula
+	addition	Add term
-	subtraction	Remove or exclude term
*	multiplication	Main effect and interactions
/	division	Main effect and nesting
:	sequence	Interaction
^	exponentiation	Limit depth of interactions
%in%	none	Nesting

Splus uses an object-oriented approach to statistical modeling. That means that each of the modeling procedures produces an object which contains an attribute known as the *class* of the object, and that certain functions will do the "right" thing when the are called with such an object as their argument.

For example, suppose we have a data frame called `corn`, containing variables `Yield`, `Block`, and `Variety`. Since Splus will automatically treat character variables as factors, it is only necessary to identify those numeric variables which you would like to be treated as factors; suppose `Block` is one such variable, taking on the values 1, 2, 3 or 4. The statement

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

will identify the variable `Block` as a factor when it is used in a statistical model. Thus, an anaylsis of variance performed with the following statement:

```
corn.aov <- aov(Yield ~ Block*Variety, data=corn)
```

would correctly assign 3 degrees of freedom to the main effect for `Block`, instead of the single degree of freedom which would result if the variable was treated as a regressor. Functions like `print`, `summary`, `anova`, and `plot` would then all provide meaningful output when passed the `corn.aov` object.

8.3 Multivariate Analysis

Splus provides many functions for multivariate analysis - unfortunately, not all of them use the object oriented model of the statistical modeling functions. Some of these functions include `cancor` (canonical correlation), `discr` (discrimination analysis), `manova` (multivariate ANOVA), `prcomp` (principal components analysis), and `factanal` (factor analysis). A number of auxiliary functions are also provided for factor analysis, to perform rotations and extract loadings).

Among clustering techniques are `hclust` (hierarchial clustering) and `kmeans` (K-means clustering).

9 Trellis Graphics

Trellis graphics provide a graphical extension to the ideas behind the statistical modelling implemented in Splus. Instead of passing matrices and vectors to a plotting function, with Trellis, you pass a formula (and optionally a data frame in which to evaluate the formula), so that you can concentrate on studying the relationship of interest, without being concerned about the syntax of a particular function. In addition, you can provide a conditioning variable as part of a formula passed to any of the Trellis functions, which results in separate plots for each value or specified ranges of values of the conditioning variable. Unlike the core graphics system (Section 6), which by default rescales each plot in a multiple plots/page display, Trellis graphics insure that the scale of each plot is identical. In addition, Trellis graphics do not leave any white space between individual plots, further facilitating comparisons between the plots in a Trellis display.

To get an idea of what Trellis graphics can do, consider the data set `fuel.frame`, which is included in the datasets of library of Splus. Suppose we wish to model the variable `Mileage` as a function of the variable `Weight`. As explained in Section 8.2, the `lm()` (linear model) function could be called as:

```
lm(Mileage ~ Weight,data=fuel.frame)
```

The formula passed to `lm` concisely describes the relation we are studying. Now suppose that we wish to see if this relation changes as we consider different `Type`s (Small, Medium, Large ...) of cars. One way to study this relation is to plot the values of `Mileage` and `Weight`. The function `xyplot()` is the Trellis function appropriate for creating a scatterplot, so one could create the desired plots with the following statement:

```
xyplot(Mileage ~ Weight|Type,data=fuel.frame)
```

This plot is displayed in Figure 1.

Note that the scales are identical on each of the plots, simplifying comparisons between plots, as well as providing useful information about the range of variables at each level of the conditioning variable. Since the Trellis graphics are built on top of the core graphics, it is theoretically possible to get the same result using the core graphics, but there are two notable differences. First, to get separate plots for each group, and to force each plot to use the same scale would require additional programming without Trellis. Secondly, the core graphics functions were not specifically designed for the types of comparisons which Trellis is suitable for, so multiple plots are placed on the page with separate axes and labels for each plot, as well as copious white space between each plot. For many of the Trellis functions, like `xyplot()`, it's intuitive as to how to specify variables of interest in a formula; for some others (like `histogram`), it may not be so obvious. The basic rule is that the variable represented by the vertical ($y-$)axis should appear on the left side of the tilda (~), while the variable on the horizontal ($x-$)axis should appear on the right side. Thus the formula `~Income|Education` would produce separate histograms for the variable `Income` for each level of `Education`. Conditioning variables always are preceded by a bar (|), and immediately follow the formula. If you use more than one conditioning variable, they should be joined by asterisks (*).

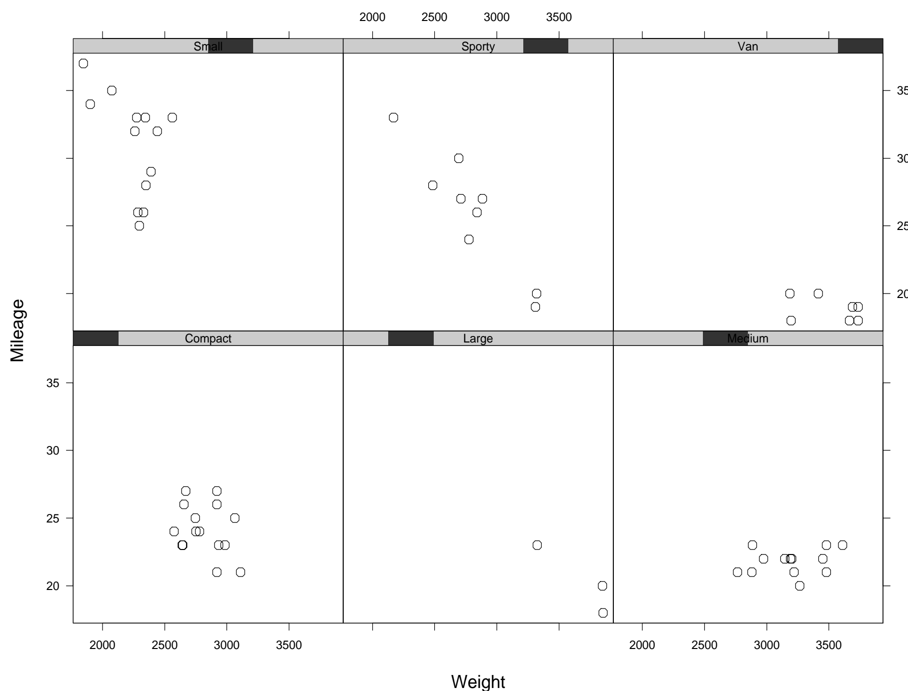

Figure 1: Scatterplot of `Mileage` vs. `Weight` conditioned on `Type`

Trellis graphics are designed to be highly customizable. The help file `trellis.args` is a good starting point to learn more. Two very useful arguments accepted by all Trellis functions are `panel`, which allows you to specify the function called to produce the individual plots, and `strip`, which controls the appearence of the labels in the strip on top of each of the plots. To learn more about Trellis, a good place to start is the Splus help file `trellis.examples`. For even more information on Trellis graphics, visit the web site

`http://netlib.bell-labs.com/cm/ms/departments/sia/project/trellis/`

10 Resources

10.0.1 Distributors Web Page:

`http://www.statsci.com`

10.0.2 Newsgroup Archive:

`http://lib.stat.cmu.edu/s-news`

10.0.3 Function repository:

`http://lib.stat.cmu.edu/s`

10.0.4 Books:

1. *The New S Language: A Programming Environment for Data Analysis and Graphics*, by R.A. Becker, J.M. Chambers and A.R. Wilks, Wadsworth & Brooks/Cole, Pacific Grove, 1988.

2. *Statistical Models in S*, by J.M. Chambers and T.J. Hastie (eds.), Wadsworth & Brooks/Cole, Pacific Grove, 1991.

3. *An Introduction to S and S-Plus*, by P. Spector, Duxbury Press, 1994.

4. *Modern Applied Statistics with S-Plus*, by W.N. Venables and B.D. Ripley, Springer-Verlag, 1994.

5. Various manuals available from Statistical Sciences, Inc.