

# Chapter 1

## Introduction

Splus is a rich graphical data analysis system and object-oriented programming language. It is an integrated suite of software facilities for data manipulation, calculation and graphical display. Among other things it has

- an effective data handling and storage facility,
- a suite of operators for calculations on arrays, in particular matrices,
- a large, coherent, integrated collection of intermediate tools for data analysis,
- graphical facilities for data analysis and display either at a workstation or on hardcopy, and
- a well developed, simple and effective programming language which includes conditionals, loops, user defined recursive functions and input and output facilities.

S-PLUS is an extension of S. S language was developed at AT&T's Bell Labs. *The New S Language*, by Becker, is the original description of the S language. S-PLUS was distributed at StatSci Inc., Seattle, Washington. S-PLUS emphasizes presentation graphics, exploratory data-analytic methods, statistical methods, computational methods for developing new statistical tools, and extensibility.

The name S (or S-PLUS ), as with many names within the UNIX world, is not explained, but left as a cryptic puzzle, and probably a weak pun. However its authors insist it does *not* stand for "Statistics"!

### 1.1 Using S-PLUS Interactively

When you use the S-PLUS program it issues a prompt when it expects input commands. The default prompt is ">". When you first time use S-PLUS , the system will automatically create subdirectory .Data for use by S-PLUS . All S-PLUS data set and program which you

wrote will be saved in this subdirectory.

In using S-PLUS the suggested procedure is as follows:

1. Start the S-PLUS program with the command

```
/home/student/zhao : Splus
S-PLUS : Copyright (c) 1988, 1999 MathSoft, Inc.
S : Copyright Lucent Technologies, Inc.
Version 5.1 Release 1 for Sun SPARC, SunOS 5.5 : 1999
Working data will be in .Data
>
```

2. At this point S-PLUS commands may be issued (see later).
3. To quit the S-PLUS program the command is
 

```
> q()
```

Use the S-PLUS program, terminating with the q() command at the end of the session.

Note: S-PLUS is a *function language* with a very simple syntax. It is *case sensitive* as are most UNIX based packages, so A and a are different variables.

## 1.2 S-PLUS and UNIX

S-PLUS allows escape to the operating system at any time in the session. If a command, on a new line, begins with an exclamation mark then the rest of the line is interpreted as a UNIX command. So for example to look through a data file without leaving S-PLUS you could use

```
> !more grade.dat
```

When you finish paging the file the S-PLUS session is resumed.

## 1.3 Getting Help With Functions and Features

S-PLUS has an inbuilt help facility similar to the man facility of UNIX. To get more information on any specific named function, for example solve the command is

```
> help(solve)
```

An alternative is

```
> ?solve
```

A much more comprehensive help facility is available. The command

```
> help.start(gui="motif")
```

causes a “help window” to appear (with the “motif” graphical user interface). It is at this point possible to select items interactively from a series of menus, and the selection process again causes other windows to appear with the help information.

## 1.4 Executing Commands From, or Diverting Output to, a File

If commands are stored on an external file, say `program.s`, they may be executed at any time in an S-PLUS session with the command

```
> source("program.s")
```

Similarly

```
> sink("output.lis")
```

will divert all subsequent output from the terminal to an external file, `output.lis`. The command

```
> sink()
```

restores it to the terminal once again.



# Chapter 2

## Data Manipulations

When using S-PLUS , you should think of your data sets as *data objects* having certain *attributes*. The attributes vary depending on the data object *type*. The types of data objects in S-PLUS include vectors, matrices, lists, data frames, arrays, factors, time series, and functions, among others.

The most fundamental type of data object is a *one-way array* of ordered data called a *vector*. A vector object might consist of numbers: `-2.0, 1.8, 6.7, -6.2`. Or it might consist of logical values: `T T F F F T T F`, where T stands for TRUE and F stands for FALSE. Or it might consist of an ordered set of character strings: `"Warm", "Cold"`. Character strings are delimited by placing quotation marks(") or apostrophes(') around them. All vector have the attributes *length* and *mode*. The above vectors have length 4, 8, and 2 and modes *numeric*, *logical*, and *character*, respectively. All the elements of a vector must be of the same mode.

### 2.1 Assigning Data Objects

To name and store data in S-PLUS , use the *assignment* operator (`<-`). For example, to create a vector consisting of the numbers 10.4, 5.6, 3.1, 6.4 and 21.7 and store it with the name `x`, use the `c` function and type:

```
> x<-c(10.4, 5.6, 3.1, 6.4, 21.7)
```

You also can use the underscore character (`_`) for assignment. The following assignment expressions are identical to the one above:

```
> x_c(10.4, 5.6, 3.1, 6.4, 21.7)
```

For the Solaris platform one can use the standard character (`=`) for assignment. The following assignment expression is identical to the ones above:

```
> x=c(10.4, 5.6, 3.1, 6.4, 21.7)
```

If an expression is used as a complete command, the value is printed *and lost*. So now if we were to use the command

```
> 1/x
```

```
[1] 0.09615385 0.17857143 0.32258065 0.15625000 0.04608295
> x
[1] 10.4  5.6  3.1  6.4 21.7
```

the reciprocals of the five values would be printed at the terminal (and the value of `x`, of course, unchanged). The further assignment

```
> y <- c(x, 0, x)
> y
[1] 10.4  5.6  3.1  6.4 21.7  0.0 10.4  5.6  3.1  6.4 21.7
```

would create a vector `y` with 11 entries consisting of two copies of `x` with a zero in the middle place.

## 2.2 Expressions

Vectors can be used in arithmetic expressions, in which case the operations are performed element by element. Vectors occurring in the same expression need not all be of the same length. If they are not, the value of the expression is a vector with the same length as the longest vector which occurs in the expression. Shorter vectors in the expression are *recycled* as often as need be (perhaps fractionally) until they match the length of the longest vector. In particular a constant is simply repeated. So with the above assignments the command

```
v <- 2*x + y + 1
```

generates a new vector `v` of length 11 constructed by adding together, element by element, `2*x` repeated 2.2 times, `y` repeated just once, and `1` repeated 11 times.

The elementary arithmetic operators are the usual `+`, `-`, `*`, `/` and `^` for raising to a power. In addition all of the common arithmetic functions are available. `log`, `exp`, `sin`, `cos`, `tan`, `sqrt`, and so on, all have their usual meaning. `max` and `min` select the largest and smallest elements of an vector respectively. `range` is a function whose value is a vector of length two, namely `c(min(x), max(x))`. `length(x)` is the number of elements in `x`, `sum(x)` gives the total of the elements in `x` and `prod(x)` their product.

## 2.3 Generating Regular Sequences

`S-PLUS` has a number of facilities for generating commonly used sequences of numbers. For example `1:30` is the vector `c(1,2, ...,29,30)`. The colon operator has highest priority within an expression, so, for example `2*1:15` is the vector `c(2,4,6, ...,28,30)`. Put `n<-10` and compare the sequences `1:n-1` and `1:(n-1)`.

The function `seq()` is a more general facility for generating sequences. For example:

```
> s1 <- seq(-5, 5)
generates in s1 the vector c(-5, -4, ..., 4, 5).
```

```
> s2 <- seq(-5, 5, .2)
```

generates in `s2` the vector `c(-5.0, -4.8, -4.6, ..., 4.6, 4.8, 5.0)`.

A related function is `rep()` which can be used for replicating a structure in various complicated ways. The simplest form is

```
> s3 <- rep(x, times=5)
```

which will put five copies of `x` end-to-end in `s3`.

## 2.4 Arrays and Matrices

### 2.4.1 Arrays

An array can be considered as a multiply subscripted collection of data entries, for example numeric. A dimension vector is a vector of positive integers. If its length is `k` then the array is `k`-dimensional. The values in the dimension vector give the upper limits for each of the `k` subscripts. The lower limits are always 1.

A vector can be used by S-PLUS as an array only if it has a dimension vector as its *dim* attribute. Suppose, for example, `z` is a vector of 24 elements.

```
> z<-seq(1:24)
> z
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
```

The assignment

```
> dim(z)<-c(2,3,4)
```

gives it the *dim* attribute that allows it to be treated as a  $2 \times 3 \times 4$  array.

```
> z
, , 1
  [,1] [,2] [,3]
[1,]   1   3   5
[2,]   2   4   6

, , 2
  [,1] [,2] [,3]
[1,]   7   9  11
[2,]   8  10  12

, , 3
  [,1] [,2] [,3]
[1,]  13  15  17
[2,]  14  16  18
```

```

, , 4
      [,1] [,2] [,3]
[1,]   19   21   23
[2,]   20   22   24

```

The assignment

```
> dim(z)<-c(4,6)
```

gives it the *dim* attribute that allows it to be treated as a  $4 \times 6$  matrix

```

> z
      [,1] [,2] [,3] [,4] [,5] [,6]
[1,]     1     5     9    13    17    21
[2,]     2     6    10    14    18    22
[3,]     3     7    11    15    19    23
[4,]     4     8    12    16    20    24

```

You also can use function `matrix( )` to create matrix.

```

> z<-matrix(seq(1:24),4)
> z
      [,1] [,2] [,3] [,4] [,5] [,6]
[1,]     1     5     9    13    17    21
[2,]     2     6    10    14    18    22
[3,]     3     7    11    15    19    23
[4,]     4     8    12    16    20    24

```

### 2.4.2 Matrix Facilities. Multiplication, Inversion and Solving Linear Equations.

S-PLUS contains many operators and functions that are available only for matrices. For example `t(X)` is the matrix transpose function. The functions `nrow(A)` and `ncol(A)` give the number of rows and columns in the matrix `A` respectively.

The operator `%%` is used for matrix multiplication.

If, for example, `A` and `B` are square matrices of the same size, then

```
> A * B
```

is the square matrix of element by element products and

```
> A %% B
```

is the matrix product  $(AB)$ . If `x` is a vector, then

```
> x %% A %% x
```

is a quadratic form  $(x^T Ax)$ .

The function `crossprod()` forms “crossproducts”, meaning that

```
> crossprod(X, y) is the same as t(X) %*% y
```

but the operation is more efficient. If the second argument to `crossprod()` is omitted it is taken to be the same as the first.

Other important matrix functions include `solve(A, b)` for solving equations ( $A\mathbf{x} = \mathbf{b} \implies \mathbf{x} = A^{-1}\mathbf{b}$ ), `solve(A)` for the matrix inverse ( $A^{-1}$ ), `svd()` for the singular value decomposition, `qr()` for QR decomposition and `eigen()` for eigenvalues and eigenvectors of symmetric matrices.

The meaning of `diag()` depends on its argument. `diag(vector)` gives a diagonal matrix with elements of the vector as the diagonal entries. On the other hand `diag(matrix)` gives the vector of main diagonal entries of `matrix`. Also, somewhat confusingly, if `k` is a single numeric value then `diag(k)` is the  $k \times k$  identity matrix!

### 2.4.3 Forming Partitioned Matrices. `cbind()` and `rbind()`.

Matrices can be built up from given vectors and matrices by the functions `cbind()` and `rbind()`. Roughly `cbind()` forms matrices by binding together matrices horizontally, or column-wise, and `rbind()` vertically, or row-wise.

In the assignment

```
> X <- cbind(arg1, arg2, arg3, ...)
```

the arguments to `cbind()` must be either vectors of any length, or matrices with the same column size, that is the same number of rows. The result is a matrix with the concatenated arguments `arg1`, `arg2`, ... forming the columns.

If some of the arguments to `cbind()` are vectors they may be shorter than the column size of any matrices present, in which case they are cyclically extended to match the matrix column size (or the length of the longest vector if no matrices are given).

The function `rbind()` does the corresponding operation for rows. In this case any vector argument, possibly cyclically extended, are of course taken as row vectors.

Suppose `X1` and `X2` have the same number of rows. To combine these by columns into a matrix `X`, together with an initial column of 1s we can use

```
> X <- cbind(1, X1, X2)
```

The result of `rbind()` or `cbind()` always has matrix status. Hence `cbind(x)` and `rbind(x)` are possibly the simplest ways explicitly to allow the vector `x` to be treated as a column or row matrix respectively.

## 2.5 Reading Data From Files

Large data objects will usually be read as values from external files rather than entered during an S-PLUS session at the keyboard.

Suppose the data vector is stored in external file "data.dat", we can use `scan()` function to read into S-PLUS data set `x`.

```
> x<-scan("data.dat")
```

If the external data set contains more than 2 vectors, you want to read this data set as matrix, you can use `matrix(scan())` function.

For example, you have external data set "data.dat"

---

52.00	111.0	830	5	6.2
54.75	128.0	710	5	7.5
57.50	101.0	1000	5	4.2
57.50	131.0	690	6	8.8
59.75	93.0	900	5	1.9

---

The assignment

```
> m<-matrix(scan("data.dat"),ncol=5,byrow=T)
```

will create  $5 \times 5$  matrix `m`

```
> m
      [,1] [,2] [,3] [,4] [,5]
[1,] 52.00 111  830   5  6.2
[2,] 54.75 128  710   5  7.5
[3,] 57.50 101 1000   5  4.2
[4,] 57.50 131  690   6  8.8
[5,] 59.75  93  900   5  1.9
```

# Chapter 3

## Statistical Analysis

S-PLUS comes with a wide variety of classical and modern statistical functions. You will find almost all standard statistical capabilities that are available in well-known statistical packages, including parametric and non-parametric statistical inference functions for both continuous and discrete data, ANOVA functions, linear and nonlinear regression, survival analysis (including Cox regression), ARIMA models, autocorrelation functions, cluster analysis, quality control charts, factor analysis, principal components analysis, and more. S-PLUS takes data analysis one step further, however, by providing modern and robust methods that are not generally available, plus a powerful, unified modeling paradigm that allows you to specify many types of statistical models with one basic syntax.

### 3.1 Summary Statistical Analysis

Using S-PLUS, you can easily get some basic summary statistics, such as mean, variance, median, quantile, etc.

- > `sum(x)`            the sum of  $\mathbf{x}$ ,  $\sum_1^n x_i$
- > `max(x)`            the maximum value of  $\mathbf{x}$ ,
- > `min(x)`            the minimum value of  $\mathbf{x}$ ,
- > `mean(x)`           the mean of  $\mathbf{x}$ ,
- > `median(x)`        the median of  $\mathbf{x}$ ,
- > `var(x)`            the variance of  $\mathbf{x}$ ,  $\frac{1}{n-1} \sum_1^n (x_i - \bar{\mathbf{x}})^2$
- > `var(x,y)`        the covariance of  $\mathbf{x}$  and  $\mathbf{y}$
- > `cor(x,y)`        the correlation coefficient of  $\mathbf{x}$  and  $\mathbf{y}$
- > `summary(x)`      mean, median, minimum, maximum, 25% quantile, and 75% quantile.

## 3.2 Hypothesis Test

### 3.2.1 Student's $t$ -Tests

Performs a one-sample, two-sample, or paired  $t$ -test, or a Welch modified two-sample  $t$ -test.

USAGE:

```
t.test(x, y=NULL, alternative="two.sided", mu=0, paired=F,
       var.equal=T, conf.level=.95)
```

Example:

```
> t.test(x) - Two-sided one-sample  $t$ -test. For test:
 $H_0 : \mu_x = 0$  vs  $H_1 : \mu_x \neq 0$ 
> t.test(x, y, mu=2) - Two-sided standard two-sample  $t$ -test. For test:
 $H_0 : \mu_x - \mu_y = 2$  vs  $H_1 : \mu_x - \mu_y \neq 2$ 
> t.test(x, y, alternative="less", paired=T)
One-sided paired  $t$ -test.  $x$  and  $y$  has same length. For test:
 $H_0 : d = 0$  vs  $H_1 : d < 0$  where  $d = \mu_x - \mu_y$ 
> t.test(x, y, var.equal=F, conf.level=0.90)
Two-sided Welch modified two-sample  $t$ -test. For test:
 $H_0 : \mu_x - \mu_y = 0$  vs  $H_1 : \mu_x - \mu_y \neq 0$ 
90% confidence interval for  $\mu_x - \mu_y$  is given.
```

### 3.2.2 Exact Binomial Test

Test hypotheses about the parameter  $p$  in a Binomial( $n, p$ ) model given  $x$ , the number of successes out of  $n$  trials.

USAGE:

```
> binom.test(x, n, p=0.5, alternative="two.sided")
```

EXAMPLES:

```
> x <- rnorm(100)
> y <- sum(x>0)
> binom.test(y, 100)
Exact binomial test
```

```
data: y out of 100
number of successes = 57, n = 100, p-value = 0.1633
alternative hypothesis: true p is not equal to 0.5
```

### 3.2.3 Kruskal-Wallis Rank Sum Test

Performs a Kruskal-Wallis rank sum test on data following a one-way layout.

USAGE:

```
> kruskal.test(y, groups)
```

where `y` is numeric vector of observations. `groups` is factor or category object of the same length as `y`, giving the group (treatment) for each corresponding element of `y`.

EXAMPLES:

```
> y <- rnorm(20)
> groups <- c(rep(1,7),rep(2,7),rep(3,6))
> kruskal.test(y, groups)
```

```
      Kruskal-Wallis rank sum test
```

```
data:  y and groups
```

```
Kruskal-Wallis chi-square = 3.0497, df = 2, p-value = 0.2177
```

```
alternative hypothesis: two.sided
```

### 3.2.4 Wilcoxon Rank Sum and Signed Rank Sum Tests

Computes Wilcoxon rank sum test for two sample data (equivalent to the Mann-Whitney test) or the Wilcoxon signed rank test for paired or one sample data.

USAGE:

```
> wilcox.test(x, y, alternative="two.sided", mu=0, paired=F, exact=T,
              correct=T)
```

where

`x`: numeric vector;

`y`: numeric vector, If supplied a two sample test is performed. If `paired=TRUE` then `x` and `y` must have the same length.

`mu`: the location shift for the distribution of `x`.

`paired`: if `TRUE`, the Wilcoxon signed rank test is computed. The default is the Wilcoxon rank sum test.

`exact`: if `TRUE` the exact distribution for the test statistic is used to compute the p-value if possible.

EXAMPLES:

```
> wilcox.test(x, y) - A Wilcoxon rank sum test
> wilcox.test(x, y, paired=T) - A Wilcoxon signed rank sum test
```

### 3.2.5 Test Homogeneity of Variance

Performs an  $F$  test to compare variances of two samples from normal populations.

USAGE:

```
> var.test(x, y, alternative="two.sided", conf.level=.95)
```

EXAMPLES:

```
> var.test(x, y, conf.level=.9)
```

For test:  $H_0 : \sigma_x^2 = \sigma_y^2$  vs  $H_1 : \sigma_x^2 \neq \sigma_y^2$ . The confidence interval for the ratio of the population variances will have a confidence level of 90%.

```
> var.test(x, y, alternative="greater")
```

For test:  $H_0 : \sigma_x^2 = \sigma_y^2$  vs  $H_1 : \sigma_x^2 > \sigma_y^2$

# Chapter 4

## Regression Models

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

```
> fitted.model <- lm(formula)
```

For example

```
> fm1 <- lm(y ~ x1 + x2)
```

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

In the follow table, a few examples of *formula* will be given. Suppose  $y$ ,  $x$ ,  $x_0$ ,  $x_1$ ,  $x_2$ , ... are numeric variables,  $X$  is a matrix and  $A$ ,  $B$ ,  $C$ , ... are factors. The following formulæ on the left side below specify statistical models as described on the right.

---

$y \sim x$ $y \sim 1 + x$	Both imply the same simple linear regression model of $y$ on $x$ . The first has an implicit intercept term, and the second an explicit one.
$y \sim -1 + x$ $y \sim x - 1$	Simple linear regression of $y$ on $x$ through the origin, (that is, without an intercept term).
$\log(y) \sim x_1 + x_2$	Multiple regression of the transformed variable, $\log(y)$ , on $x_1$ and $x_2$ (with an implicit intercept term).
$y \sim \text{poly}(x,2)$ $y \sim 1 + x + I(x^2)$	Polynomial regression of $y$ on $x$ of degree 2. The first form uses orthogonal polynomials, and the second uses explicit powers, as basis.
$y \sim X + \text{poly}(x,2)$	Multiple regression $y$ with model matrix consisting of the matrix $X$ as well as polynomial terms in $x$ to degree 2.
$y \sim A$	Single classification analysis of variance model of $y$ , with classes determined by $A$ .
$y \sim A + x$	Single classification analysis of covariance model of $y$ , with classes determined by $A$ , and with covariate $x$ .

---

---

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

---

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

---

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

---

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

---

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

$$response \pm term_1 \pm term_2 \pm term_3 \pm \dots$$

where:

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

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

*term* is either

- a vector or matrix expression, or 1,
- a factor.

### 4.0.6 Updating Fitted Models

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

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

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

```
> fm1 <- lm(y ~ x1 + x2 + x3 + x4 + x5)
> fm2 <- update(fm1, . ~ . + x6)
> fm3 <- update(fm2, sqrt(.) ~ .)
```

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

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

```
> fmfull <- lm(y ~ .)
```

would fit a model with response y and all other variables as regressor variables.



# Chapter 5

## Graphics in S-PLUS

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. S-PLUS has a rich and flexible environment for composing graphics for both exploratory data analysis and for final publication. you have complete control over all aspects of the plot, including the following:

- axis placement, scale, and labels,
- tick mark placement and size,
- titles and text within the plot region,
- plotting characters for data points,
- plot style for lines and curves,
- color selection.

The system chooses “sensible” defaults for all unspecified parameters, so it is not necessary to specify each element unless a choice other than the default is desired. Defaults can be permanently changed if you want different settings. Multiple plots can be composed on page, and plots can be superimposed on one another. A plot can be composed an element at a time, so there is no need to plan the entire appearance of the plot before beginning.

### 5.1 Opening Graphics Windows

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

```
> motif()  
or
```

```
> x11()
```

or

```
> openlook()
```

(which creates a separate window in which high-quality graphical output will appear.)

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

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

## 5.2 Turning Graphics Windows Off

You can turn off any active window using `dev.off`. To turn off the current graphics window, use `dev.off` with no arguments:

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

The *next* graphics window becomes current. When you turn off the last active window, S-PLUS returns the following:

```
> dev.off()
null device
      1
```

To turn off a device other than the current device, use `dev.off` with the `which` argument:

```
> dev.off(which=2)
> dev.off(which=dev.next())
```

To turn off all active windows, use `graphics.off`:

```
> graphics.off()
```

## 5.3 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.

### 5.3.1 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.

---

<code>plot(x,y)</code>	If <code>x</code> and <code>y</code> are vectors, <code>plot(x,y)</code> produces a scatterplot of <code>x</code> against <code>y</code> .
<code>plot(x)</code>	Produces a time series plot if <code>x</code> is a numeric vector or time series object, or an Argand diagram if <code>x</code> is a complex vector.

---

### 5.3.2 Displaying Multivariate Data

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

```
> pairs(X)
```

produces a pairwise scatterplot matrix of the variables defined by the columns of `X`, 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`.

### 5.3.3 Display Graphics

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

---

<code>tsplot(x1,x2,...)</code>	Plots any number of time series on the same scale. This automatic simultaneous scaling feature is also useful when the <code>x<sub>i</sub></code> 's are ordinary numeric vectors, in which case they are plotted against the numbers 1, 2, 3, ...
--------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

---

<code>qqnorm(x)</code> <code>qqplot(x,y)</code>	Distribution-comparison plots. The first form is normal probability plot which plots the numeric vector <code>x</code> against the expected Normal order scores. The second form is q-q plot which plots the quantiles of <code>x</code> against those of <code>y</code> to compare their respective distributions.
<code>hist(x)</code>	Produces a histogram of the numeric vector <code>x</code> .
<code>dotchart(x)</code>	Construct a dotchart of the data in <code>x</code> .
<code>barplot(x)</code>	Creates a bar graph. Vertical or horizontal bars may be produced, shading patterns created, and other options chosen.
<code>pie(x)</code>	Creates a pie chart from a vector of data, including the possibility of some pieces displaced.

---

### 5.3.4 Arguments of Plotting Functions

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

---

<code>log="x"</code> <code>log="y"</code> <code>log="xy"</code>	Causes the $x$ , $y$ or both axes to be logarithmic. Only works for scatterplots (and variants).
<code>type=</code> <code>type="p"</code> <code>type="l"</code> <code>type="b"</code> <code>type="o"</code> <code>type="h"</code> <code>type="s"</code> <code>type="S"</code> <code>type="n"</code>	The <code>type=</code> argument controls the type of plot produced, as follows: Plot individual points (the default) Plot lines Plot points connected by lines ("both") Plot points overlaid by lines Plot vertical lines from points to the zero axis ("high-density") Step-function plots. In the first form, the top of the vertical defines the point; in the second, the bottom. 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.
<code>xlab="string"</code> <code>ylab="string"</code>	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.
<code>main="string"</code> <code>sub="string"</code>	Figure title, placed at the top of the plot in a large font. Sub-title, placed just below the $x$ -axis in a smaller font.

---

## 5.4 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:

---

<code>points(x,y)</code> <code>lines(x,y)</code>	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 "p" for <code>points()</code> and "l" for <code>lines()</code> .)
<code>text(x, y, labels)</code>	Add text to a plot at points given by <code>x</code> , <code>y</code> . Normally <code>labels</code> is an integer or character vector in which case <code>labels<sub>i</sub></code> is plotted at point $(x_i, y_i)$ . The default is <code>1:length(x)</code> .
<code>abline(a, b)</code> <code>abline(h=y)</code> <code>abline(v=x)</code> <code>abline(lm.obj)</code>	Adds a line of slope <code>b</code> and intercept <code>a</code> ( $y = a + bx$ ) to the current plot. <code>h=y</code> may be used to specify <code>y</code> -coordinates for the heights of horizontal lines to go across a plot, and <code>v=x</code> similarly for the <code>x</code> -coordinates for vertical lines. Also <code>lm.obj</code> will add regression line ( $y = \widehat{\beta}_0 + \widehat{\beta}_1 x$ ) to the current plot.
<code>legend(locator(1), legend=...)</code> <code>legend(,angle=v)</code> <code>legend(,fill=v)</code> <code>legend(,col=v)</code> <code>legend(,lty=v)</code> <code>legend(,pch=v)</code>	Adds a legend to the current plot at the any position. There are some option in legend command, such as: Shading angles Colours for filled boxes Colours in which points or lines will be drawn Line styles Plotting characters (character vector)
<code>title(main,sub)</code>	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.

---

## 5.5 Multiple Figure Environment

S-PLUS allows you to create an  $n \times m$  array of figures on a single page. Each figure has its own style.

The graphical parameters relating to multiple figures are as follows:

---

<code>mfc01=c(3,2)</code> <code>mfrow=c(2,4)</code>	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 <code>mfc01</code> causes figures to be filled by column; <code>mfrow</code> fills by rows.
<code>mfg=c(2,2,3,2)</code>	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.
<code>fig=c(4,9,1,4)/10</code>	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.
<code>oma=c(2,0,3,0)</code> <code>omi=c(0,0,0.8,0)</code>	Size of outer margins. The first measures in text lines and the second in inches, starting with the bottom margin and working clockwise.

---

# Chapter 6

## Output of Graphics

Whatever what device you use to display graphics, they can be sent directly to a printer.

### 6.1 Copying Graphics Between Windows

You can copy graphics from one window to another using `dev.copy`. By default, `dev.copy` copies the graphic in the current window to the *next* window.

You can also copy to the active window of your choice, using `dev.copy` with the `which` argument:

```
> dev.copy(which=2)
```

The window copied to becomes the current window.

### 6.2 Getting Hard Copy

Each graphics window also offers a simple, straightforward way to get a hard copy of the picture you have composed on the screen: the Print option on the Graph pull-down menu on the `motif` or `openlook` or `X11` devices. You can exercise even more control over your instant hard copy, such as specifying whether the copy is in landscape or portrait orientation, which printer the hard copy is sent to.

In `SCL`, the default printer is `SP` - SunPics. This printer is laser printer. If you want to get color graphic, you can send your hard copy to `NeWSprinterCL`.

### 6.3 Getting Output file

If you don't want instant hard copy of your graphy, you can use `postscript` to send graphics to a specified file:

```
> postscript("graphy.ps")
```

All future graphics output to be sent to the file `graphy.ps` in PostScript format.

You also can use `printgraph` to produce separate files for each graphic you produce, as soon as you have finished composing it on a windowing graphics device:

```
> printgraph("graphic1.ps")
```

# Chapter 7

## Writing Your Own Functions

S-PLUS has over 1000 built-in functions, such as `mean()`, `var()` and so on. You also can write your own functions. To define a new function, you type an expression of the following form:

```
newfunction <- function(arguments){  
  body of definition  
}
```

where *newfunction* is the name you have chosen for your new function, *arguments* are the names for the arguments, if any, used by the function, and the *body of definition* contains one or more valid S-PLUS expressions, separated by semicolons or newlines.

It should be emphasized that the name of function, *newfunction*, must differ from the S-PLUS built-in functions' name.

A call to the function then usually takes the form `newfunction(arguments)` and may occur anywhere a function call is legitimate.

For example, suppose you are analyzing a number of data sets to determine the shapes of their distributions. After starting a graphics device, you could type the following sequence of commands for each data set `x`:

```
> hist(x)  
> boxplot(x)  
> iqd <- summary(x)[5] - summary(x)[2]  
> plot(density(x, with = 2*iqd), xlab="x", ylab="", type="l")  
> qqnorm(x)  
> qqline(x)
```

After typing this for one data set, you might notice that setting `par(mfrow=c(2,2))` would let you see all the plots on a single screen. With command line editing, recalling these commands is not too burdensome, but incorporating all the commands into a single function call is more appealing still. You can use UNIX command `vi` to edit one file, called *eda.s*:

```

/home/student/zhao : cat eda.s
function(x){
par(mfrow=c(2,2))
hist(x)
boxplot(x)
iqd <- summary(x)[5] - summary(x)[2]
plot(density(x, with = 2*iqd), xlab="x", ylab="", type="l")
qqnorm(x)
qqline(x)
}

```

Then you start S-PLUS and use S-PLUS command `source` to create new function `eda.shape`:

```

> eda.shape <- source("eda.s")
> eda.shape
function(x)
{
    par(mfrow = c(2, 2))
    hist(x)
    boxplot(x)
    iqd <- summary(x)[5] - summary(x)[2]
    plot(density(x, with = 2 * iqd), xlab = "x", ylab = "", type = "l")
    qqnorm(x)
    qqline(x)
}

```

From one data set to the next, the only part of this function call that needs to change is the argument name `x`. If you have more than one or two data sets to analyze, it is faster and easier for you to create `eda.shape` than to try and simply reissue all the above commands for each data set. The advantage of typing simple

```

> eda.shape(my.data)

```

should be clear.