Table 5.9. Optional parameters for adding text

Parameter	Description
`text` and `mtext`: Text in a figure	
`adj=0.5`	Text adjustment. 0=left justified, 0.5=centered, 1=right justified
`cex=1`	Character expansion relative to standard size cex=2 draws characters twice as big
`col=1`	Color to plot in. 0 is the background color
`crt=0`	Character rotation in degrees, counterclockwise
`srt=0`	String rotation in degrees, counterclockwise
`font=1`	Font for plotting characters, device dependent
`mtext` only: Text in the margin of the figure	
`side=`n	Side to add text to. 1=bottom, 2=left, 3=top, 4=right
`outer=F`	Text not on the outer margin of the whole figure. If `outer=T`, text can be placed on top of a layout figure. For an example, see Figure 7.15 (page 213)
`line=`n	Places the text on line n toward the margin of the figure. Negative numbers lie inside the figure

Note: Text is typically added by using the functions `text` or `mtext`. Some devices like text terminals cannot provide all of the functionality, like rotating text.

A standard example for adding the text "extreme value" to a graph at the coordinates (1, 2), such that the text begins on the right side of the point, is

```
> text(1, 2, "extreme value", adj=0)
```

Sometimes, the text overlaps or touches the point on the graph just slightly. In such a case, add an extra space to the beginning of the text. The optional parameters of `text` and `mtext` offer more possibilities. They are listed in Table 5.9.

5.5 Setting Options

We have already seen that almost all graphics functions accept parameters for setting options like colors, character size, or string adjustment. There are many more parameters and almost all of the functions generating graphics output accept them as arguments. They can also be set generally, such that they become the system default for all succeeding graphics commmands. For doing this, we can use the `par` function.

Check the different options that can be set by `par`. Enter the command

```
> par()
```

to see all options and how they are currently set. The help pages provide a very detailed explanation.

We supply a brief overview of the most common parameters and their possibilities in Figure 5.4 and Table 5.10. If changed using the `par` function, these settings become global from then on. Nevertheless, if a new session is started they are set back to the original system defaults.

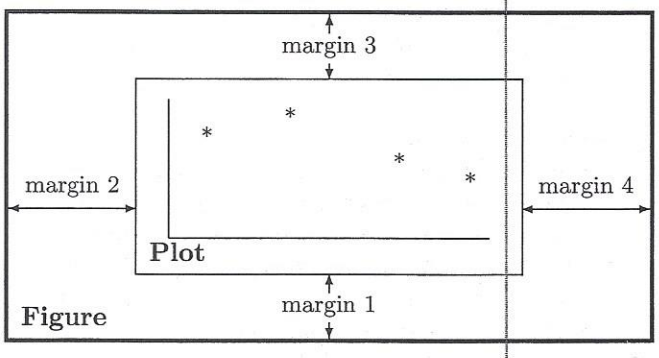

Figure 5.4. The S-PLUS definition of figure and plot regions.

Table 5.10. Layout parameters

Parameter	Description
`fin=c(`m`,`n`)`	Figure size in inches. m width, n height
`pin=c(`m`,`n`)`	Picture in inches, as in `fin`
`mar=c(5,4,4,2)+0.1`	All margins in lines
`mai=c(1.41,1.13,1.13,0.58)`	All margins in inches
`oma=c(0,0,0,0)`	Outer margin lines
`omi=c(0,0,0,0)`	Outer margin in inches
`plt=c(0.11,0.94,0.18,0.86)`	Plot region coordinates as fraction of figure region
`usr`	x-Axis and y-axis min. and max. Useful for querying a graph's boundaries
`mfrow=c(`m`,`n`)`	Multiple figure layout, rowwise plotting, to generate a (m, n) matrix of pictures
`mfcol=c(`m`,`n`)`	Multiple figures, columnwise plotting

Note: In the left column, the letters m and n are to be replaced by integer numbers. Settings given are the system default values. To query settings, use (for example) `par("usr")`.

The parameters referenced in Table 5.10 can be set by using the `par` function, as in the following example where the outer margin is set to be equal to 1 line on all sides, for all successive graphs.

```
> par(mar=c(1, 1, 1, 1))
```

A setting can be retrieved by entering

```
> par("mar")
```

or

```
> par()$mar
```

If you run S-PLUS under Windows, most of these settings can alternatively be changed by using the menu bars on top of the S-PLUS window.

Note You can add your own parameters to the system settings using an unused variable. The command

```
> options()
```

shows all settings not related to the graphics subsystem. A new variable `my.personal.setting` can be set to `TRUE` by entering

```
> options(my.personal.setting=T)
```

at the prompt. Your own functions can then use this parameter setting by asking for its setting as follows:

```
> options("my.personal.setting")
   T
```

◁

5.6 Figure Layouts

The graphical parameters we looked at in this chapter are all preset with meaningful values. Nevertheless, you might want to change the default values. This section deals with the most common cases to demonstrate how to achieve satisfying results quickly.

If you use a multiple-figure layout, the system may not always choose the layout as you prefer (the size of the characters, the white space between figures, etc.). We now show, for the standard layouts, how parameters are set to generate the graphics wanted.

Note Creating a multiple figure graph can sometimes be cumbersome. In some situations, you need to adjust the margins to be smaller or the font size to be larger.

A graph's exact layout is still to some extent device dependent, so if you plan on creating a PostScript graph, don't do the graph on the screen until the layout satisfies you. Adjust everything on the final output device.

If you don't use labels for the axes, you can shrink the blank space around the figure and thus obtain more space for the graph. For example, a good setting is often

```
> par(mar=c(2, 2, 1, 1))
```

such that there is a space of two lines on the lower and the left side of the graph, and a single line on top and on the right of the graph.

Note Once the graph page is filled with figures, a new page is opened (the screen is cleared) on creating another figure. One can jump to the next figure by using

```
> frame()
```

Entering `frame()` twice leaves a field blank. Restore the default settings by allowing for a single figure per page:

```
> par(mfrow=c(1,1))
```

◁

5.6.3 Multiple Screens Graphs

Another way of creating a set of figures in the same graph is to use the `split.screen` function. In its simplest use, it works like the `par(mfrow)` command. One would enter

```
> split.screen(c(2, 2))
```

to generate a 2×2 matrix of figures. In comparison to the `par(mfrow)` variant, splitting the screen provides the ability to hop around between areas. On the other hand, it does not automatically plot the next graph in the next plot area, but waits for explicit specification of another plot area.

```
> screen(2)
```

will activate the second screen in the display.

The matrix of screens can be filled either horizontally (which is the default) or vertically by setting `byrow=F` in the call to `split.screen`.

It is possible to have a matrix of differently sized graphs or even any layout. If a new graph is started, its initial coordinates range from 0 to 1, similar to 0 - 100%). Each screen in the graph is defined by its lower left and top right corner. The following command defines a layout of a 2×1 matrix, where the first row stretches from 0 to 1 on the x-axis and from 0 to 3/4 on the y-axis. The second row stretches from 0 to 1 on the x-axis and from 3/4 to 1 on the y-axis.

```
> split.screen(figs=matrix(c(0, 1, 0, 3/4, 0, 1, 3/4, 1),
+ 2, 4, byrow=T))
```

Note that the matrix argument to `figs` must have four columns. Each row defines the corners of one of the screens. The screen layout is closed by using `close.screen`.

```
> close.screen(all=T)
```

5.6.4 *Figures of Specified Size*

An alternative to what we have looked at so far is to fill an empty graph successively with figures and define the size of each figure as we go (i.e., right before the figure is created). We need to open a new graphics page, which is done by entering the command

```
> frame()                                  # open next graph
```

After having opened a new graphics page, the coordinates of the plot are set to (0,0) for the lower left corner and (1,1) for the top right corner. Let us now plot the Geyser data set that comes with S-PLUS.

We want to set up a figure such that a scatterplot of the data appears as the main plot with the addition of a histogram of the x data on top of the scatterplot and a boxplot of the y data to the right-hand side of it. The layout we have in mind looks like the one in Figure 5.5.

The plot itself should have the lower left corner coordinates (0, 0) and the top right corner coordinates (0.7, 0.7). Therefore, the figure on top of the data plot has the lower left corner (0.0, 0.7), and the upper right corner is (0.7, 1.0). For the vertical boxplot on the right side, we get the lower left corner (0.7,0) and the upper right corner (1.0, 0.7).

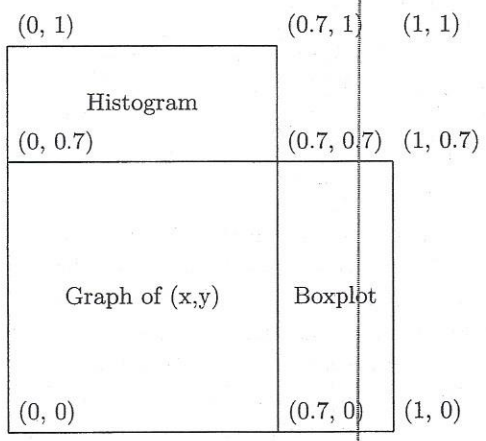

Figure 5.5. A customized graph layout.

By experimenting with the different sizes and the borders, we discover that it is better to have the figures overlapping a little bit.

```
> frame()
> par(fig=c(0, 0.7, 0, 0.7))
> plot(geyser$waiting, geyser$duration, pch="*",
+ xlab="Waiting Time", ylab="Duration of Eruption")
> title("\nOld Faithful Geyser Data Set")[1]
> par(fig=c(0, 0.7, 0.65, 1))
> hist(geyser$waiting)
> par(fig=c(0.65, 1, 0, 0.7))
> boxplot(geyser$duration)
```

Finally, we obtain Figure 5.6.

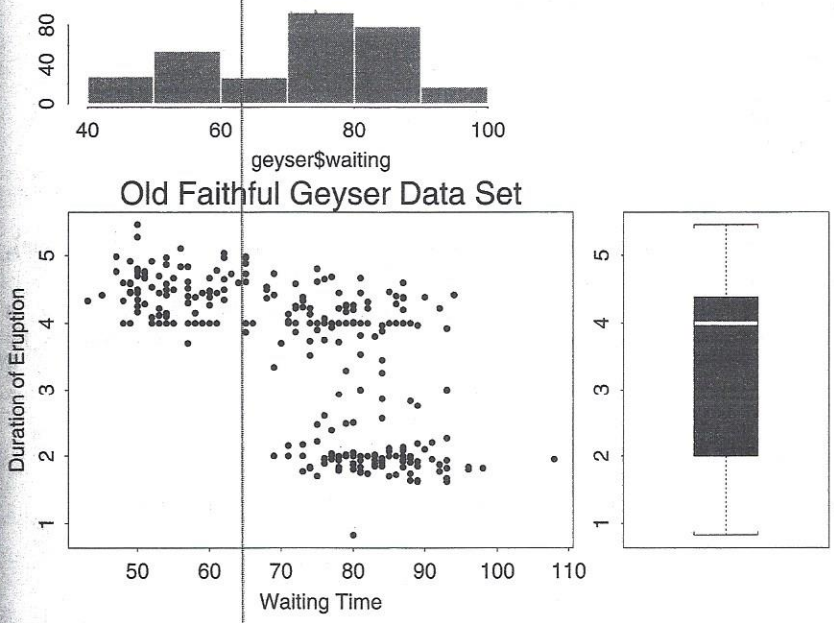

Figure 5.6. A customized display of the Geyser data set.

[1]The control character \n is a "linefeed" character, advancing to the next line. In this way, the title gets closer to the graph.

5.7 Exercises

Exercise 5.1

In a single figure, plot the functions sin, cos, and sin + cos with different colors and line styles. Use 1000 points in the interval $[-10, 10]$ and label the figure with a title, subtitle, and axis labels.

Exercise 5.2

Construct two vectors, x and y, such that the S-PLUS command

```
> plot(x, y, type="l")
```

creates the following figures:

a) a rectangle or square

b) a circle

c) a spiral

Hint: Do not think of y as being a function of x (in a mathematical sense), but think of how the trace of the figure can be created.

Hint for drawing circles: you might use the property that for any x, a point $(\sin(x), \cos(x))$ lies on the unit circle with origin $(0, 0)$ and radius 1.

Exercise 5.3

We consider the so-called Lissajous figures. They are defined as

$$z(x) = \begin{pmatrix} \sin(ax) \\ \sin(bx) \end{pmatrix} = \begin{pmatrix} z_1 \\ z_2 \end{pmatrix},$$

or, in different notation,

$$z_1(x) = \sin(ax)$$
$$z_2(x) = \sin(bx)$$

a, b positive integers, x between 0 and 2 π.

Plot z_1 against z_2, using lines to connect the points, and choose different pairs of values for a and b. Plot several figures on a single sheet.

On what do the forms of the curves depend? Compare, for example, the figures for $(a, b) = (3, 4)$, $(3, 6)$, and $(6, 8)$.

5.8 Solutions

Solution to Exercise 5.1

What we need to create this graph is a sequence of x values, for which we calculate the values $\sin(x)$ and $\cos(x)$. If several functions are to be plotted within one picture, the ordering of plotting the figures is important. The arguments to the `plot` function, usually the first curve, determine the boundaries of the figure. The following curves are then added to the existing picture within the existing limits. If you want to see more of the $\sin(x) + \cos(x)$ curve than that which fits within the boundaries of $\sin(x)$, extend the y limits by using the option `ylim` of the `plot` function or - and this is easier - plot the curve $\sin(x) + \cos(x)$ first.

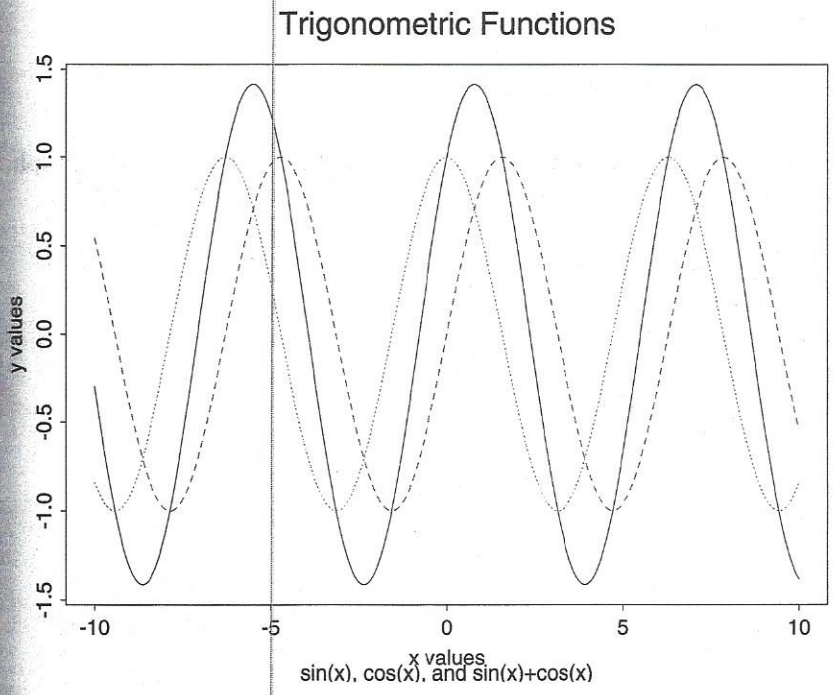

Figure 5.7. Trigonometric functions.

```
> x <- seq(-10, 10, length=1000)
> plot(x, sin(x)+cos(x), xlab="x values", ylab="y values")
> lines(x, cos(x), lty=2, col=2)
> lines(x, sin(x), lty=3, col=3)
> title("Trigonometric Functions",
+ "sin(x), cos(x), and sin(x)+cos(x)")
```

Solution to Exercise 5.2

The idea of this exercise is to learn how to think in terms of geometric figures and not in terms of mathematical functions, where y is a function of x.

(a) A rectangle is a simple figure consisting of four lines. In order to draw a rectangle, we need to create two vectors x and y containing the coordinates of the corners. These vectors must have *five* elements, as the line has to return to the first point to complete the figure.

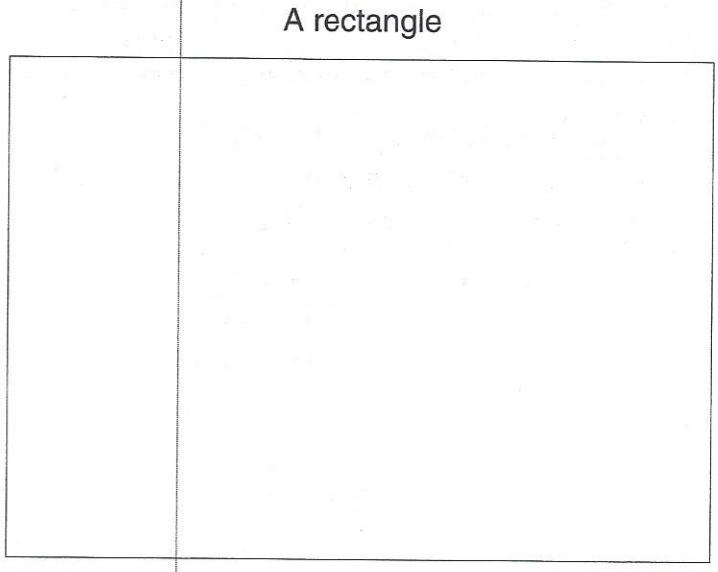

Figure 5.8. Geometric figures (a): A rectangle

```
> x <- c(1, -1, -1, 1, 1)
> y <- c(1, 1, -1, -1, 1)
> plot(x, y, type="l", axes=F, xlab="", ylab="")
> title("A rectangle")
```

Note that the first point defines the upper right corner of the rectangle, the second defines the upper left corner, and so on.

(b) It is possible to come up with several solutions for drawing a circle. We use the fact that a point with the coordinates $(\sin(x), \cos(x))$ lies (for all x) on the unit circle with radius 1 (because $\sin^2(x) + \cos^2(x)=1$). Then we simply create a sequence from 0 to 2π and plot sin(x) against cos(x). Another approach is to use $x^2 + y^2 = 1$ as a description for a circle. Then we have the solutions $y = \sqrt{1 - x^2}$ and $y = -\sqrt{1 - x^2}$. Try to use these formulas to draw the circle.

Finally, if you plot the vector x against y, you will discover that the resulting graph is not a circle but an ellipsoid. To obtain a circle, we need to tell S-PLUS that this plot's boundaries should be square instead of rectangular. For this purpose, we set the size of the plot, the parameter `pin` (picture in inches), in order to have a square picture shape. If we would have used this option in part (a), we would have obtained a square instead of a rectangle.

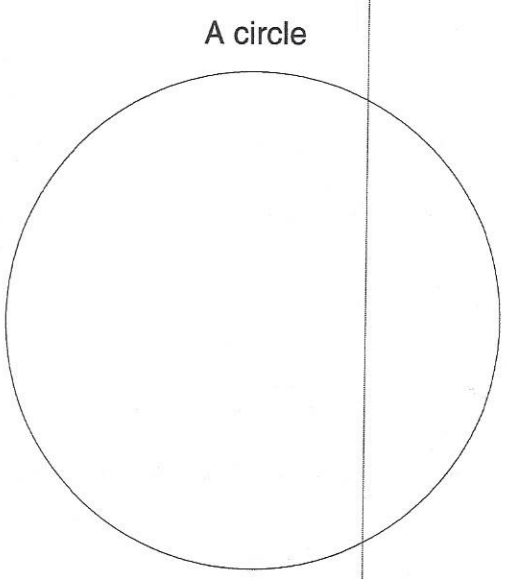

Figure 5.9. Geometric figures (b): a circle.

```
> z <- seq(0, 2*pi, length=1000)
> x <- sin(z)
> y <- cos(z)
> par(pin=c(5, 5))
> plot(x, y, type="l", axes=F, xlab="", ylab="")
> title("A circle")
```

(c) The spiral is simply made by revolving several times around the circle and changing the radius over time. The windings depend very much on the division. By changing the start, the end, or the division factor, you get very different pictures.

We select a sequence of values from 6π to 32π. With a sequence of length 2π, we obtain a circle by moving once around its area. Therefore, with a sequence length of 26π, the spiral must have 13 windings.

```
> z <- seq(6*pi, 32*pi, length=1000)
> x <- sin(z)/(0.1*z)
> y <- cos(z)/(0.1*z)
> plot(x, y, type="l", main="A spiral with 13 windings",
+ axes=F, xlab="", ylab="")
```

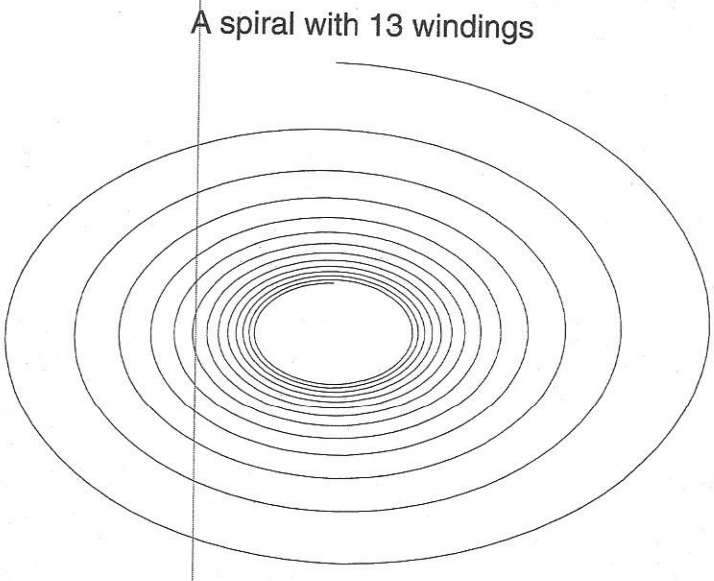

Figure 5.10. Geometric figures (c): a spiral.

Solution to Exercise 5.3

We first create the layout of the graphics window (open a window first if you haven't already). Then, we create a sequence from 0 to $2 * \pi$ and store this sequence in `x`.

It is not necessary to define values a and b and store $\sin(a*\texttt{x})$ and $\sin(b*\texttt{x})$ in variables before plotting them, because we do no longer need these data after plotting them. Instead, we supply these expressions directly to the `plot` function.

```
> par(mfrow=c(2, 2), mar=c(2,2,1,1))
> x <- seq(0, 2*pi, length=1000)
> plot(sin(3*x), sin(6*x), type="l")
> plot(sin(3*x), sin(8*x), type="l")
> plot(sin(3*x), sin(11*x), type="l")
> plot(sin(7*x), sin(8*x), type="l")
```

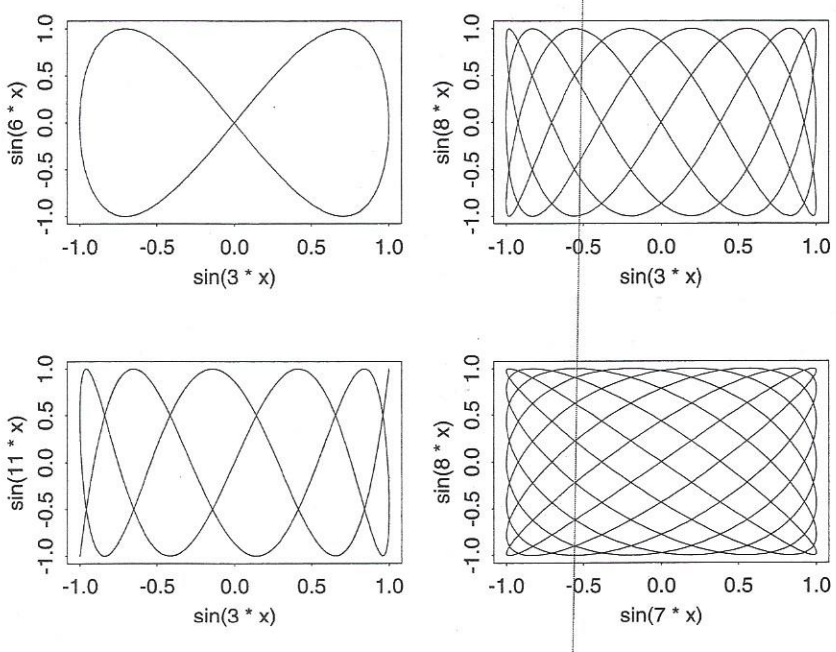

Figure 5.11. Lissajous figures for different values of a and b.

7
Exploring Data

In the preceding chapters, we have laid the foundation for understanding the concepts and ideas of the S-PLUS system. We explored basic ideas and how to use S-PLUS for performing calculations, and we have seen how data can be generated, stored, and accessed. Furthermore, we also looked at how data can be displayed graphically. All this will be useful as we explore real data sets in this chapter. We will explore data sets that come with S-PLUS, specifically the Barley and Geyser data sets.

Rather than presenting a list of available statistical functions, we will go through a typical data analysis as a way of introducing the more useful and common commands and the kind of output we'll encounter. We chose to use S-PLUS data sets so you can follow along with the analysis we present and complete the exercises at the end of this chapter. We divide the data analysis into two categories: "descriptive" and "graphical" exploration.

7.1 Descriptive Data Exploration

We will now explore the different variables contained in the Barley data set. We will first analyze the variables in one dimension, or, in other words, we will take a univariate approach. The analysis of the dependence between the variables and the exploration of higher dimensional structure follows later.

The Barley Data Set

The Barley data are measurements of yield in bushels per acre at different sites. The analysis comprises 6 sites planting 10 different varieties of barley in 2 successive years, 1931 and 1932. The data set therefore contains 120 measurements of barley yield. Our main goal will be to investigate differences in barley yields given by the different variable constellations, like the 1931 harvest of the fifth variety on site 4, and the 1932 harvest of the seventh variety at the same site.
Just enter

```
> barley
```

to see the data.

Exploratory data analysis (EDA) is an approach to investigating data that stresses the need to know more about the structure and information inherent in the data. The methods used with this approach are referred to as descriptive, as opposed to confirmatory. Descriptive simply means that simple summaries are used to describe the data: their shapes, sizes, relationships, and the like. Examples of descriptive statistics are means, medians, standard deviations, ranges, and so on.

Given the basic information about the Barley data, the following analysis is intended to gain more information and structural knowledge about the numbers we have.

A typical place to begin is, of course, looking at the data. If the data set is small, we can easily look at it by simply printing it out. We check the data size by entering

```
> dim(barley)
   120 4
```

The data set `barley` has 120 rows and 4 columns. We randomly pick some rows of the data matrix.

```
> barley[c(2, 17, 64, 70, 82, 98, 118), ]
           yield           variety  year          site
2   48.86667         Manchuria  1931        Waseca
17  29.66667          Svansota  1931  Grand Rapids
64  32.96667         Manchuria  1932     Crookston
70  26.16667           Glabron  1932     Crookston
82  32.06666            Velvet  1932     Crookston
98  44.70000           No. 462  1932        Waseca
118 35.90000  Wisconsin No. 38  1932     Crookston
```

This already shows us what the data look like. The `yield` is the number of bushels per acre, as we know from the data description (have another look at the manual or the online help pages to learn more about it). It is a decimal, and not always an integer. The number of bushels harvested was probably divided by the area of the corresponding plot of barley. The second

variable, `variety`, contains a string describing the name of the variety of barley. The `year` is either 1931 or 1932, denoting the year of planting and harvesting. Finally, the `site` variable contains the name of the site from which the data originate.

We can determine the structure in a more simplified format by using the `summary` function, which summarizes each variable on its own.

```
> summary(barley)
      yield          variety     year              site
 Min.:14.43   Svansota:12   1932:60   Grand Rapids:20
 1st Qu.:26.87   No. 462:12   1931:60   Duluth      :20
 Median:32.87   Manchuria:12             Univ Farm   :20
 Mean:34.42   No. 475:12                 Morris      :20
 3rd Qu.:41.40   Velvet:12               Crookston   :20
 Max.:65.77   Peatland:12                Waseca      :20
              Glabron:12
              No. 457:12
          Wisc No.38:12
              Trebi:12
```

In fact, the `summary` function shows per default up to seven different values for factor variables, such that entering the command as above gives a slightly compressed output. To get exactly what you see above, we used `summary(barley, maxsum=10)`. The argument `maxsum` is the parameter setting the maximum number of rows displayed in the `summary` function.

Now, we investigate the data further. To have easier access to the variables contained in `barley`, we attach the data set.

```
> attach(barley)
```

This gives us direct access to the variables `yield`, `variety`, `year`, and `site`.

How about having a look at our main variable `yield`? Let us determine what the distribution looks like by using the stem and leaf display.

```
> stem(yield)
  N = 120   Median = 32.86667
  Quartiles = 26.85, 41.46666

  Decimal point is 1 place to the right of the colon
    1 : 4
    1 : 579
    2 : 0011122223333
    2 : 555666666667777777889999999
    3 : 000000111222223333344444
    3 : 5555667777888899
    4 : 000112223334444
    4 : 567777779999
    5 : 00
```

```
  5 : 5889
  6 : 4
  6 : 6
```

The data are ordered and categorized by a base times a power of 10. The stem and leaf display tells us that the data are put into categories 10, 20, 30, .., 60, because "the decimal point is 1 place to the right of the colon." The left-hand side displays the category (or interval), and the right-hand side of the colon has one digit per observation, displayed by the first digit following the category digit. We can immediately see that the smallest values are 14, 15, 17, and 19 (rounded, of course), and by far the largest values are 64 and 66 bushels per acre. Most of the yields are in the high twenties and low thirties, and the distribution looks rather symmetric, having a longer tail toward the larger values (in other words, it is right-skewed).

The stem and leaf plot can be viewed as a sort of sideways histogram from which we can see that the data approximately exhibit the bell shape typical of a Normal or Gaussian distribution with no extreme values or outliers. As with the histogram, the display sometimes strongly depends on the categorization layout chosen. For glimpsing the distribution and detecting extreme values in the set, the above display is sufficient. However, if we want to determine whether the distribution is approximately bell shaped like a Normal distribution, we would need to display it with different categorizations or use more sophisticated techniques.

If we want to apply the `stem` display to the other variables of barley, we would get an error message.

```
> stem(variety)
    Problem in Summary.factor(x, na.rm = na.rm):
    A factor is not a numeric object
    Use traceback() to see the call stack
```

What we realize here is that S-PLUS behaves somewhat intelligently. It "knows" that `variety` is a factor variable with values like "Svansota" and "Manchuria" and that a stem and leaf display for such a variable does not make much sense. Therefore, S-PLUS simply refuses to do the desired display and tells us that our attempt is not meaningful for factors.

If we want to investigate the skewness of the `yield` variable further, we could use the `quantile` function and calculate the quantiles at 10, 20, 30, ..., 90%.

```
> quantile(yield, seq(0.1, 0.9, by=0.1))
      10%    20%    30%      40%      50%
 22.49667  26.08  28.09  29.94667  32.86667
      60%      70%    80%      90%
 35.13333  38.97333  43.32  47.45666
```

This shows us that the 10% quantile (22.5) is about 10 units away from the median (32.9), whereas the 90% quantile is 47.5, being about 15 bushels per

acre away from the median. For a symmetric distribution, the two quantiles would be about the same distance away from the median. A measure of the data spread is the interquartile distance, the difference between the 25% and the 75% quantile.

```
> quantile(yield, c(0.25, 0.75))

     25%   75%
  26.875  41.4
```

This tells us that approximately 50% of the data lie within the interval of 26.9 to 41.4 bushels per acre. For perfectly symmetric data, the median would be about 34, equally far away from both quartiles. The median 32.87 is a little off, such that judging about the symmetry of the `yield` variable might require further investigation.

Another measure of spread is the variance, or the standard deviation. S-PLUS has a built-in function for both, `var` for the variance and `stdev` for the standard deviation.[1]

```
> stdev(yield)
   10.33471
```

We can also use what we have learned about selecting subsets from data. We can calculate some interesting statistics by conditioning on certain values. Let us check if the barley harvest was very different in 1931 and 1932. Selecting only the values of `yield` that were obtained in 1931, we obtain

```
> summary(yield[year==1931])
    Min. 1st Qu. Median  Mean 3rd Qu.  Max.
   19.70 29.09   34.20  37.08 43.85   65.77
```

and for 1932, we obtain

```
> summary(yield[year==1932])
    Min. 1st Qu. Median  Mean 3rd Qu.  Max.
   14.43 25.48   30.98  31.76 37.80   58.17
```

As we can see, all the numbers are larger for 1931. The 1932 harvest seems to have been much worse than the 1931 yield, measured in bushels per acre.

We could determine the "most fruitful" sites for both years to see if they were the same in both. A good idea is to select the top 10%, the sites above the 90% quantile. We get the 90% quantile for both years as

```
> quantile(yield[year==1931], 0.9)
      90%
 49.90334
```

[1]The function `stdev` has been introduced with S-PLUS for Windows 2000 and S-PLUS for UNIX 5.

```
> quantile(yield[year==1932], 0.9)
    90%
  44.28
```

and retrieve all rows of the Barley data set where the yield in 1931 lies above the 90% quantile.

```
> barley[yield>49.90334 & year==1931, ]
         yield          variety year   site
 8  55.20000          Glabron 1931 Waseca
 20 50.23333           Velvet 1931 Waseca
 26 63.83330            Trebi 1931 Waseca
 32 58.10000          No. 457 1931 Waseca
 38 65.76670          No. 462 1931 Waseca
 56 58.80000 Wisconsin No. 38 1931 Waseca

> barley[yield>44.28 & year==1932, ]
         yield          variety year   site
 86  49.23330            Trebi 1932 Waseca
 87  46.63333            Trebi 1932 Morris
 98  44.70000          No. 462 1932 Waseca
 99  47.00000          No. 462 1932 Morris
 116 58.16667 Wisconsin No. 38 1932 Waseca
 117 47.16667 Wisconsin No. 38 1932 Morris
```

Remember what we have learned about selecting data from matrices? In the example above, we query if `yield` is larger than 44.28 and `year` is equal to 1932, giving us a vector of `TRUE/FALSE` elements. These are used as row indices for the matrix `barley`, and the column index is omitted to get all columns. All the rows of `barley` are returned where it is `TRUE` that `yield` is larger than 44.28 and `year` is equal to 1932.

The output above reveals that Waseca seemed to have had an incredible harvest in 1931 compared to the other sites. The year 1932 wasn't that bad for Waseca either, but Morris also had a good year. It turns out that both sites had their big yield in 1932 for the same varieties, Trebi, No. 462, and Wisconsin No. 38.

Another table confirms this finding. We generate a table of average yield per site and year.

```
> tapply(yield, list(site, year), mean)
                     1932     1931
Grand Rapids 20.81000 29.05334
      Duluth 25.70000 30.29333
University Farm 29.50667 35.82667
      Morris 41.51333 29.28667
   Crookston 31.18000 43.66000
      Waseca 41.87000 54.34667
```

The function `tapply` splits the data (here `yield`) according to the values of other variables (here `site` and `year`) and applies a specified function to the subsets. The result is returned in form of a table.

The above table confirms that the 1931 Waseca harvest of 54.4 bushels on average is by far the highest, and the Waseca 1932 harvest equals almost exactly the Morris harvest, although the Morris harvest in 1931 was pretty low on average.

We would already have some interesting questions to ask, if the person collecting the data was available. However, we will focus on the analysis tools S-PLUS offers and continue our investigation.

The `by` function is a useful tool for doing analyses by categorizing the data as we have just done. It applies functions to data by first splitting the data into subcategories. We can calculate the summaries for the Barley data by year, using the `by` function.

```
> by(barley, year, summary)
```

```
year:1932
      yield                  variety     year             site
   Min.:14.43  Svansota,No. 462:12  1932:60  Grand Rapids:10
 1st Qu.:25.48         Manchuria: 6  1931: 0        Duluth:10
  Median:30.98           No. 475: 6               Univ Farm:10
    Mean:31.76   Velvet,Peatland:12                  Morris:10
 3rd Qu.:37.80           Glabron: 6               Crookston:10
    Max.:58.17           No. 457: 6                  Waseca:10
              Wisc No. 38,Trebi:12
------------------------------------------------------------
year:1931
      yield                  variety     year             site
   Min.:19.70  Svansota,No. 462:12  1932: 0  Grand Rapids:10
 1st Qu.:29.09         Manchuria: 6  1931:60        Duluth:10
  Median:34.20           No. 475: 6               Univ Farm:10
    Mean:37.08   Velvet,Peatland:12                  Morris:10
 3rd Qu.:43.85           Glabron: 6               Crookston:10
    Max.:65.77           No. 457: 6                  Waseca:10
              Wisc No. 38,Trebi:12
```

This provides a summary of the whole data set, but in the form of two summary tables, one for 1931 and one for 1932. Note that S-PLUS displays only seven lines per default. Therefore, the 10 different varieties of barley show up in 4 categories of 1 variety and 3 categories of 2 varieties. In the table above, Svansota and No. 462 were aggregated into one category. If we want S-PLUS to display more than just 7 lines (because we have 10 varieties), we can add the `maxsum=10` setting as an additional parameter to the `by` function. Similar to the call to `summary` above, where we entered

```
> summary(barley, maxsum=10)
```

we can now enter

```
> by(barley, year, summary, maxsum=10)
```

to obtain the same formatting. In the same way, you could calculate the summary statistics or other figures for each site or each variety separately. There is no need for explicit subsetting and cycling through the sites or varieties.

You have seen that a combination of only a handful of functions offers many ways of looking at a data set. Table 7.1 summarizes the S-PLUS functions and some of their parameters we have just seen. In addition, it contains a few more functions that can be helpful in other applications.

Table 7.1. Descriptive statistics functions

S-PLUS Function	Description
`quantile`	Quantiles of data
`mean`	(Optionally trimmed) mean of data
`median`	Median of data
`stem`	Stem and leaf display
`var`	Variance of data or, if two variables are supplied, covariance matrix
`by`	Applies a function to data split by indices
`summary`	Summary statistics of an object
`apply, lapply, sapply, tapply`	Calculations on rows or columns of matrices and arrays (`apply`), on components of lists (`lapply`, `sapply`), and data subsets (`tapply`)
`aggregate`	Aggregate data by performing a function (like mean) on subsets

Note If you want to use the `by` function to calculate the mean value of the yield for each year, you would encounter a little difficulty.

```
> by(yield, year, mean)
    Problem in as.double: Cannot coerce mode list to
    double: list(value = c(26.9, 33.46667, 34.36666, ....
    Use traceback() to see the call stack
```

The problem arises because the `mean` function does not work on lists. In detail, `by` passes the `barley` elements - which are internally stored as lists - to `mean`, which, in turn, does not know what to do with them. Entering

```
> mean(list(123))
```

does not work either. A quick solution is to convert the list structure to a vector before calculating the mean, using `unlist`. We write a little function

```
> newmean <- function(x) { mean(unlist(x)) }
```

and use our function newmean instead.

```
> by(yield, year, newmean)
   INDICES:1932
   [1] 31.76333
   ------------------------------------------------
   INDICES:1931
   [1] 37.07778
```

◁

We continue to analyze the relationships between variables by examining them together or by conditioning one variable on the value of another. We saw how the barley yield differs when we condition on the year 1931 only, or on 1932. We will now explore the relationship in more detail.

Note that we focus on data exploration by using exploratory techniques. If you are interested in deterministic techniques and know how these methods work, it will not be difficult to find the appropriate S-PLUS functions in the manual and apply them to a specific data set.

Table 7.2 shows functions used in the following analyses.

Table 7.2. Tabulation and split functions

S-PLUS Function	Description
`hist2d`	Tabulation of two-dimensional continuous data by counting the number of occurrences in intervals
`table`	Tabulation of discrete data of any dimension. Counts the number of data equal to the cell values
`crosstabs`	Creation of a contingency table from factor data
`split`	Split up a data set and get a list with one component per group value

To see how a variable is distributed in two dimensions, S-PLUS offers the `table` and the `hist2d` function. We can determine how often different combinations of `year` and `site` occur by entering

```
> table(year, site)
           Grand          Univ
        Rapids  Duluth  Farm  Morris  Crookston  Waseca
  1932      10      10    10      10         10      10
  1931      10      10    10      10         10      10
```

We see that every site had 10 plantings in each year. If we look at the three-dimensional table of `year`, `site`, and `variety`, we can see that every combination of `year`, `site`, and `variety` occurs exactly once, as we have 10 different varieties. We shorten the output because of its excessive length.

```
> table(year, site, variety)
 ,,Svansota
      Grand           Univ
      Rapids  Duluth  Farm  Morris  Crookston  Waseca
1932      1       1     1       1          1       1
1931      1       1     1       1          1       1

,,No. 462
      Grand           Univ
      Rapids  Duluth  Farm  Morris  Crookston  Waseca
1932      1       1     1       1          1       1
1931      1       1     1       1          1       1
...
```

S-PLUS shows the third dimension by "walking through the dimensions." It keeps the first value of the third dimension's variable fixed (in this case, the Svansota variety) and shows a two-dimensional table for all data having Svansota as its variety. The next table is shown for the second value for `variety`, here No. 462, and so on, until all third-dimension values are taken.

To tabulate a variable like `yield`, which is a metric variable, we would not want to use `table`, because every observation would get a category of its own, as all the values are different. We could round the values and use `table`, or we can categorize them with the `hist2d` function. The `hist2d` function takes the data and categorizes them into intervals, just as the `hist` function does, but without graphical display. Because of the lack of two metric variables in the `barley` data, we tabulate `yield` against itself. The output consists of five elements; the midpoints of the x and y categories, the table containing the counts of the data, and two vectors giving the interval limits.

```
> hist2d(yield, yield)
 $x:
 [1] 15 25 35 45 55 65
 $y:
 [1] 15 25 35 45 55 65
 $z:
           10 to 20 to 30 to 40 to 50 to 60 to
              20    30    40    50    60    70
10 to 20       6     0     0     0     0     0
20 to 30       0    42     0     0     0     0
30 to 40       0     0    39     0     0     0
40 to 50       0     0     0    26     0     0
50 to 60       0     0     0     0     5     0
60 to 70       0     0     0     0     0     2
 $xbreaks:
 [1] 10 20 30 40 50 60 70
```

```
$ybreaks:
[1] 10 20 30 40 50 60 70
```

Because of the leading $ sign, we can see that the returned data is stored in a list.

If we have a mixture of continuous and categorical variables and want to tabulate them, we can build classes on our own by rounding the metric variable. Let us round the yield to the nearest factor of 10. The `round` function has, besides the data, a second argument `digits`, telling us the number of digits to the right of the decimal point. If `digits` is negative, it refers to the number of digits to round off *to the left* of the comma. Rounding to the nearest factor of ten means that we need to set `digits` to -1.

```
> yield.round <- round(yield, -1)
```

We can check the result by tabulating our new variable, `yield.round`, which contains only the values 10, 20, 30, 40, 50, 60, and 70.

```
> table(yield.round)
 10  20  30  40  50  60  70
  1  18  51  31  13   5   1
```

Now, we can tabulate `variety` against `yield.round`. You might want to see what happens if you tabulate `variety` against the original variable `yield` - without rounding the values - using the `table` function.

```
> table(variety, yield.round)
                    10 20 30 40 50 60 70
          Svansota   0  3  4  4  1  0  0
           No. 462   0  3  4  2  2  0  1
         Manchuria   0  2  8  1  1  0  0
           No. 475   0  4  4  3  1  0  0
            Velvet   0  2  5  4  1  0  0
          Peatland   0  0  8  3  1  0  0
           Glabron   1  0  5  5  0  1  0
           No. 457   0  2  5  3  1  1  0
Wisconsin No. 38     0  1  4  3  2  2  0
             Trebi   0  1  4  3  3  1  0
```

We can see that most of the yields are around 30 bushels per acre. There is only a single harvest in the category 70, stemming from the species No. 462, but the species Wisconsin No. 38 and Trebi have mostly high and only a few low yields.

Using the `by` function, we can investigate this a little further. Let us calculate a summary for each variety by using the function `summary`.

```
> by(yield, variety, summary)
 INDICES:Svansota
        x
 Min.   :16.63
 1st Qu.:24.83
 Median :28.55
 Mean   :30.38
 3rd Qu.:35.97
 Max.   :47.33
 ------------------------------
 INDICES:No. 462
        x
 Min.   :19.90
 1st Qu.:25.41
 Median :30.45
 Mean   :35.38
 3rd Qu.:45.28
 Max.   :65.77
 ------------------------------
 ...
 ------------------------------
 INDICES:Wisconsin No. 38
        x
 Min.   :20.67
 1st Qu.:31.07
 Median :36.95
 Mean   :39.39
 3rd Qu.:47.84
 Max.   :58.80
 ------------------------------
 INDICES:Trebi
        x
 Min.   :20.63
 1st Qu.:30.39
 Median :39.20
 Mean   :39.40
 3rd Qu.:46.71
 Max.   :63.83
```

It seems that Trebi and Wisconsin are the more fruitful varieties, as all figures are high in comparison to the other varieties.

If we wanted to split our data set into parts, for example to analyze each site on its own, we could select only a single variety by entering

```
> barley.Svansota <- barley[variety=="Svansota", ]
```

to select only the Svansota data, or we could split up the data by using the `split` function.

```
> barley.split.by.variety <- split(barley, variety)
```

The first component of the list `barley.split.by.variety` looks like this now:

```
> barley.split.by.variety$Svansota
          yield   variety  year  site
    13 35.13333 Svansota  1931  Univ Farm
    14 47.33333 Svansota  1931  Waseca
    15 25.76667 Svansota  1931  Morris
    16 40.46667 Svansota  1931  Crookston
    17 29.66667 Svansota  1931  Grand Rapids
    18 25.70000 Svansota  1931  Duluth
    73 27.43334 Svansota  1932  Univ Farm
    74 38.50000 Svansota  1932  Waseca
    75 35.03333 Svansota  1932  Morris
    76 20.63333 Svansota  1932  Crookston
    77 16.63333 Svansota  1932  Grand Rapids
    78 22.23333 Svansota  1932  Duluth
```

We have all the Barley data of the Svansota variety in a separate structure. The variable `barley.split.by.variety` contains a list in which the element names are the different varieties. If you enter

```
> names(barley.split.by.variety)
```

you will see the names of the list components. We see that for Svansota, for example, 1931 was a much better year than 1932, and that Waseca, Crookston, and University Farm had their biggest harvest in 1931, whereas the biggest harvests in 1932 were in Waseca, Morris, and University Farm.

7.2 Graphical Exploration

We have gained a basic impression and some insights into the Barley data set structure by studying some summary figures. The next step will lead us into graphical analyses. Table 7.3 lists many of the graphical functions available in S-PLUS. In addition, we will be using the Trellis functions as summarized in Table 6.1 (page 143).

Let us start by getting a graphical impression of the distribution of the barley yield. We could either use the standard plotting function `hist` or the Trellis function `histogram`.

We decide to use the `histogram` function.

```
> histogram(~yield, data=barley)
```

Table 7.3. Graphical data exploration functions

S-PLUS Function	Description
`boxplot`	Boxplot display
`density`	Density estimate for the distribution of the data. Use `plot(density(x))` to see it graphically
`dotchart`	Dot chart
`hist`	Histogram display
`identify`	Identification of the data point next to the point clicked on in the graphics window
`pie`	Pie chart
`plot`	Standard plot, depending on the data
`qqnorm`	Quantile–quantile plot for the Normal distribution
`qqplot`	Quantile–quantile plot to check how well two data sets overlay
`scatter.smooth`	Plot of the data plus a smoothed regression line

See also Table 6.1 (page 143) for Trellis graphics functions.
Note: Most of the graphics functions have a long list of optional parameters. It is, of course, possible to use a function only in its elementary form (like `pie(x)`), but if you want to add colors, modify the labels, or explode one of the pies in the pie chart, you need to know what you can do. The `boxplot` function, for example, currently has 27 optional parameters. Have a look at the documentation to learn more about all the possibilities offered.

Figure 7.1 shows that all yield data lie between 10 and 70 bushels per acre, and that there is more data in the lower than in the upper region.

Let us graphically explore how the yield depends on the site, the year of harvest, or the variety of barley planted. A Trellis displays of histograms where the barley yield is conditioned on the site is shown in Figure 7.2. It is created with the following command:

```
> histogram(~yield | site, data=barley)
```

Figure 7.2 shows that Waseca has obtained the highest yields, and Grand Rapids the lowest. In fact, all yields above 60 were obtained in Waseca, and all yields below 20 were obtained in Grand Rapids.

Let us note how we could produce the same display using the `hist` function that does not allow conditioning on variables. We would need to split the yield data and draw the histograms one by one.

```
> par(mfrow=c(2,3))
> attach(barley)
> hist(yield[site=="Morris"], main="Morris")
> hist(yield[site=="Crookston"], main="Crookston")
> ...
```

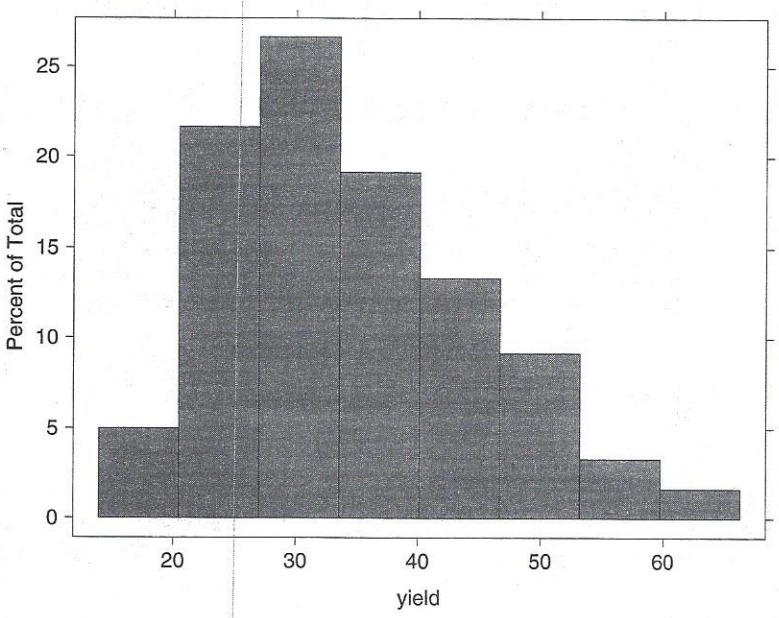

Figure 7.1. Histogram of the variable `yield` in the Barley data set.

Alternatively, we could use the `split` function that splits a variable according to values of another variable.

```
> yield.split <- split(barley$yield, barley$site)
> par(mfrow=c(2,3))
> lapply(yield.split, hist)
```

With both solutions, we did not make sure that we get the same histogram bins. The bins need to be calculated beforehand and supplied as a parameter to each call of the `hist` function. We better stop here and be happy that we can use the `histogram` function that comes as part of the Trellis library.

Continuing our investigation, we condition the yield on year and site. We create histograms for each combination of `site` and `year`. As there are 6 sites in the data sets, each having two yields for 1931 and 1932, we obtain $6 \times 2 = 12$ histograms. They show 10 values for the 10 different varieties planted.

We use the Trellis function `histogram` again to condition the yield data according to the values of `year` and `site`. Figure 7.3 shows the histogram display by `year` and `site`. The figure was obtained by entering

```
> histogram(~yield | year+site)
```

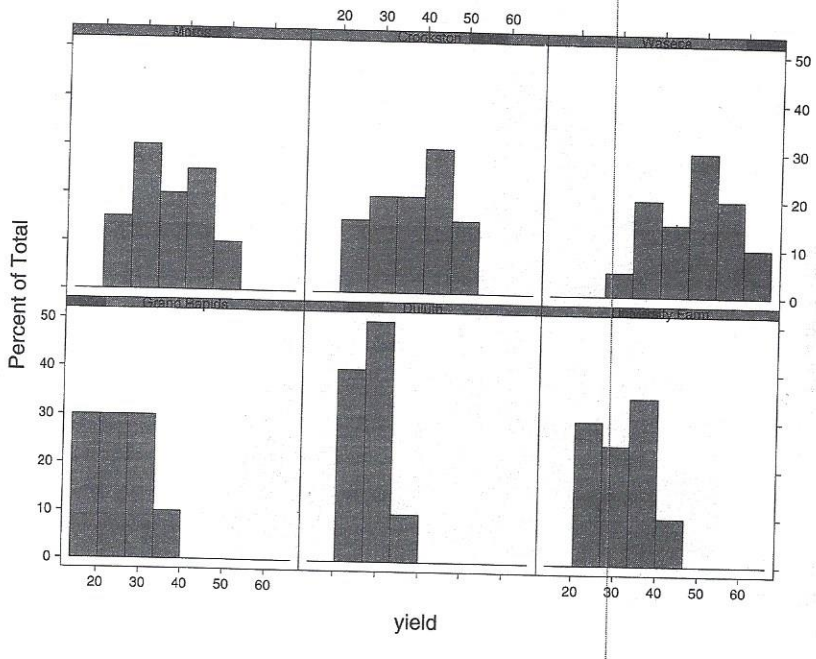

Figure 7.2. Histogram of the barley yield by site.

In this way, we are able to review our data set in a concise form. The histograms can be compared columnwise, observing the yields for the different sites in 1931 and 1932, respectively. If they are viewed rowwise, we can see the different yields for the 2 years for a given site.

Let us now graphically compare the sites' yields for the two harvests in 1931 and 1932. We choose the `boxplot` display to show all sites for a specific year in the same plot.

By reading the online documentation or the manuals, we find that if we want to display more than one data set in the same picture, the `boxplot` function expects a list of variables to plot. What we need to do is create a list containing one vector of yields for each of the sites. We have learned that the S-PLUS function `split` splits a vector according to another variable, which we are going to use now. We split the `yield` by `site`, as we have done before, and supply the outcome to the `boxplot` function. Since we want to produce two plots, one for 1931 and one for 1932, we extract the data for the year first, then split the extracted data.

```
> is.1931 <- year==1931              # True or False for 1931/1932
> boxplot(split(yield[ is.1931], site[ is.1931]),
+ main="Year 1931", ylim=range(yield))
> boxplot(split(yield[!is.1931], site[!is.1931]),
```

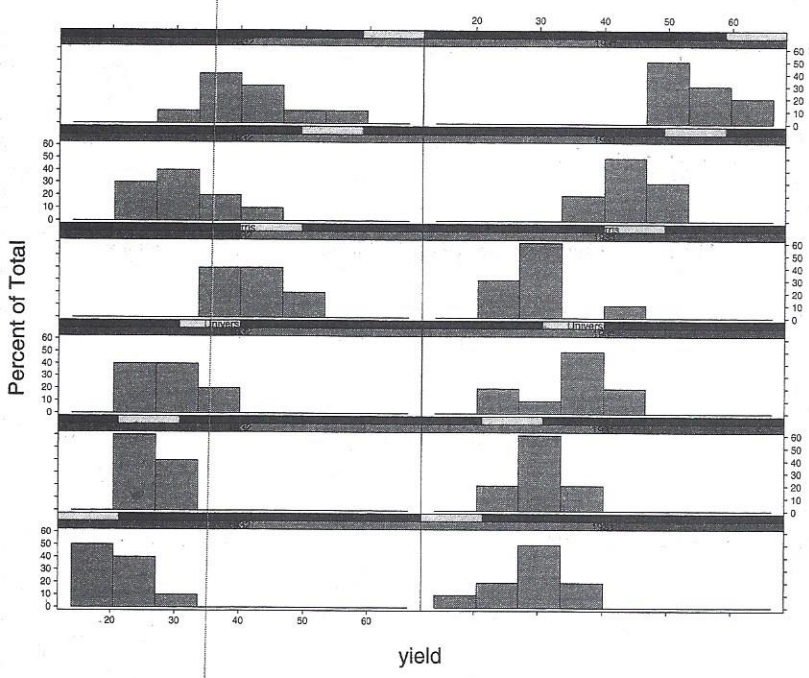

Figure 7.3. The barley yield by `site` and `year`.

```
+ main="Year 1932", ylim=range(yield))
```

The result obtained is shown in Figure 7.4. Note the extraction of data by using the variable `is.1931`. The result is true for all data for which `year` is equal to 1931, and false otherwise (for all 1932 observations). In the first `boxplot` call, we select the data for which `is.1931` is true; in the second `boxplot` call, we select the data for which it is not true.

We have made sure that both graphs are shown on the same scale by adding `ylim=range(yield)` to the call.

If you are not familiar with boxplot displays, a box consists of a few basic elements: the lower quartile, the upper quartile, and the median (the dash in the middle). Therefore, 50% of the data lie within the box limits. The box position tells us a lot about the data spread within a site and gives a comparison between the sites. The whiskers show the interval of values outside the box, and values far outside are represented by horizontal dashes. To learn more about this standard display tool, you might want to study Tukey (1977) or other books covering descriptive statistics or exploratory data analysis.

Figure 7.4. Boxplot display of the barley yield for all sites, split by year of harvest.

Note If you try out the boxplot example, you will realize that there is a slight disturbance with the boxplot labels. The site names are sometimes too long, like "Grand Rapids" or "University Farm," and they overlap. What we can do in this case is to split the label into two lines by taking the result of `split`, a list, storing it into a variable, and renaming some labels. For example, we can make "Grand Rapids" a two-liner by inserting the new line control character `\n` in the middle. The command would look like this.

```
> yield.split.1931 <- split(yield[is.1931], site[is.1931])
> names(yield.split.1931)          # to find "Grand Rapids"
> names(yield.split.1931)[1] <- "Grand\nRapids"
> boxplot(yield.split.1931)
```

The labels in Figure 7.4 were created in this fashion. We will learn more about control characters later. They are summarized in Table 9.2 on page 286. ◁

You might have noted already that we could have done the same graph using the Trellis function `bwplot`. The name "bwplot" is short for "box and whisker plot."

```
> bwplot(site~yield | year, data=barley)
```

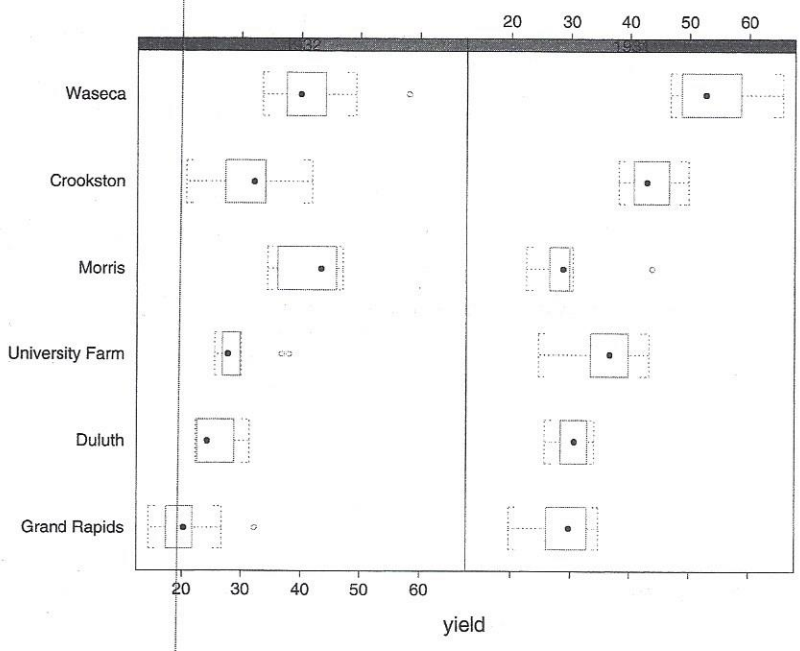

Figure 7.5. Boxplot display of the barley yield by site and year.

The graph generated is shown in Figure 7.5. Many of the details in the preceding displays can be found again in this display. We see immediately that the Waseca site has very high yields, especially in 1931, but the yields cover a much broader range than at Duluth, for example. Grand Rapids is always pretty low, and Morris is somewhat strange, as its yield in 1931 is very low, and in 1932, it is very high, in contrast to the other sites. For the other sites, the ordering is very stable for both years if ordered by yield size.

Displaying the boxplots in a different arrangement gives us some more information. Figure 7.6 clearly reveals that the Morris site is the only one that had a lower yield in 1931 than in 1932. For all other sites, the upper box is clearly more toward the right than the lower box, indicating a higher yield in 1931.

The graph was generated by swapping the two conditional variables, year and site, such that all sites in a specific year show up in a common graph (compare Figures 7.5 and 7.6 and the commands used to generate them).

```
> bwplot(year~yield | site, data=barley)
```

Using these Trellis displays, Cleveland (1993) and Becker, Cleveland, and Shyu (1996) conclude that the years 1931 and 1932 must have been swapped for Morris. Of course, this can no longer be proven, but there is some clear

Figure 7.6. The Barley data in boxplot displays, categorized by site.

evidence for it. Although this data set has undergone many analyses in the statistical literature, this obscureness had not been detected, which shows how graphical methods can help provide more insight into a data set.

To illustrate further techniques, we examine the Geyser data set in detail. This data set was previously introduced in Section 3.6.

The Geyser Data Set

The Geyser data consist of continuous measurements of the eruption length and the waiting time between two eruptions of the Old Faithful Geyser in Yellowstone National Park in 1985 in minutes. Some duration measurements, taken at night, were originally recorded as S (short), M (medium), and L (long). These values have been coded as 2, 3, and 4 minutes, respectively. The original publication by Azzalini and Bowman (1990) provides more details.

Having a look at the `geyser` data (have a look!) shows that they are stored in a list with two components: `waiting` and `duration`. Lists can be attached, such that we can use `waiting` instead of typing `geyser$waiting` all the time.

```
> attach(geyser)
```

We might want to begin by examining the basic data characteristics.

```
> summary(waiting)
      Min. 1st Qu. Median   Mean 3rd Qu.   Max.
     43.00   59.00  76.00  72.31  83.00  108.00
> summary(duration)
      Min. 1st Qu. Median   Mean 3rd Qu.   Max.
    0.8333 2.0000 4.0000 3.4610 4.3830 5.4500
```

This gives us a basic impression. The duration of an eruption lies between less than a minute and almost 6 minutes, and the waiting time for the next eruption lies between a little more than 40 minutes and more than 100 minutes. On average, if you just missed the last eruption, then you would have to wait for more than an hour to see an eruption of approximately 4 minutes.

Next we are interested in a graphical data display. We display the two variables in a simple scatterplot.

```
> plot(waiting, duration,
+ xlab="Waiting Time for the Eruption",
+ ylab="Eruption Length")
> title("Old Faithful Geyser Data\n
+ Waiting Time and Eruption Length")[2]
```

Figure 7.7 shows what the data look like. We can see some surprising details from the graph. We already know that parts of the data were set to 2, 3, and 4 minutes, such that quite a few points lie on these lines parallel to the x-axis. Second, it seems that the data also lie on lines parallel to the y-axis. Examining the data confirms that `waiting` time was measured in integer minutes.

Next, we see three clusters of point clouds. Interestingly enough, they are more or less clearly separable. We can use an interactive facility to determine the cluster's boundaries. Plot the data and use the mouse to click on some points in the graph. Calling the function `locator`, S-PLUS will return the coordinates of the points you click on with the left mouse button. Stop recording the clicks by clicking on the right mouse button.

```
> plot(waiting, duration)
> locator()
```

If we cut the x-axis at x=67 and the y-axis at y=3.1, the plot is divided into four areas, three of which contain a cluster of data points, and the fourth, the one in the lower left position, contains no data at all. This might require questioning the Park Ranger who collected the data.

To show it to the ranger, we plot the graph again and add the dividing lines to it. Furthermore, we choose a different plotting character for each

[2]The `title` command should be on a single line with no space around the `\n`, which is impossible here due to space constraints.

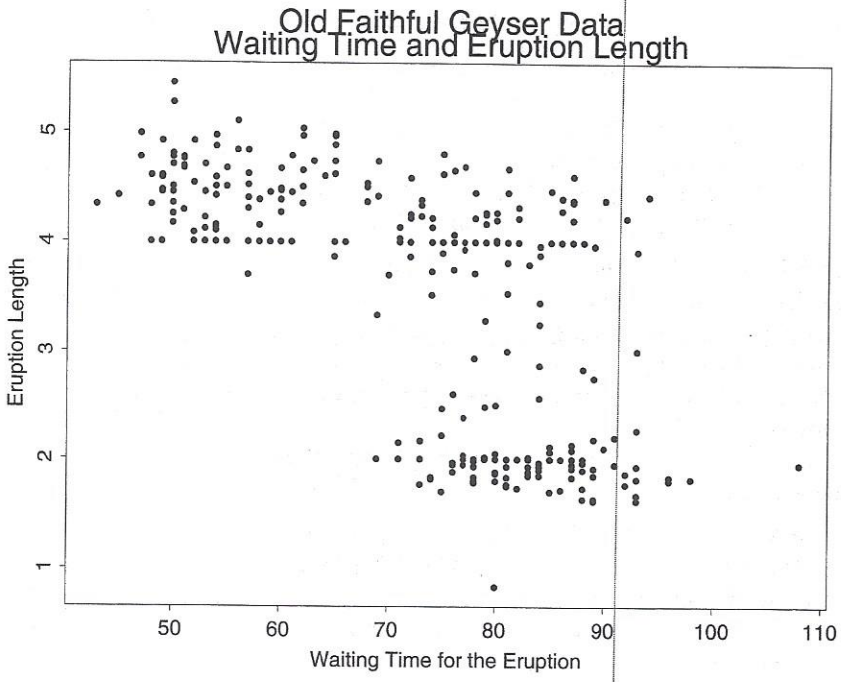

Figure 7.7. The Old Faithful Geyser data.

of the four subsets, and label the extreme value in the lower right corner (the only one with a waiting time of more than 100 minutes).

The following commands produced Figure 7.8. We first plot a graph *with no points*, which contains everything except for the points (the axes, labels, and other layout parameters). We then add the horizontal and vertical lines and, finally, the points.

```
> plot(x, y, type="n", ylab="Eruption Length",
+ xlab="Waiting Time for the Eruption")
> title("Old Faithful Geyser Data\nWaiting
+ Time and Eruption Length")[3]
> abline(h=3.1)
> abline(v=67)
```

The data are divided into three subcategories, which we determine by the logical expressions `subset1`, ..., `subset3`. These vectors of logical `TRUE` and `FALSE` elements determine whether or not a point belongs to the subset.

[3]The `title` command should be on a single line, with no space around the `\n`, which is impossible here due to space constraints.

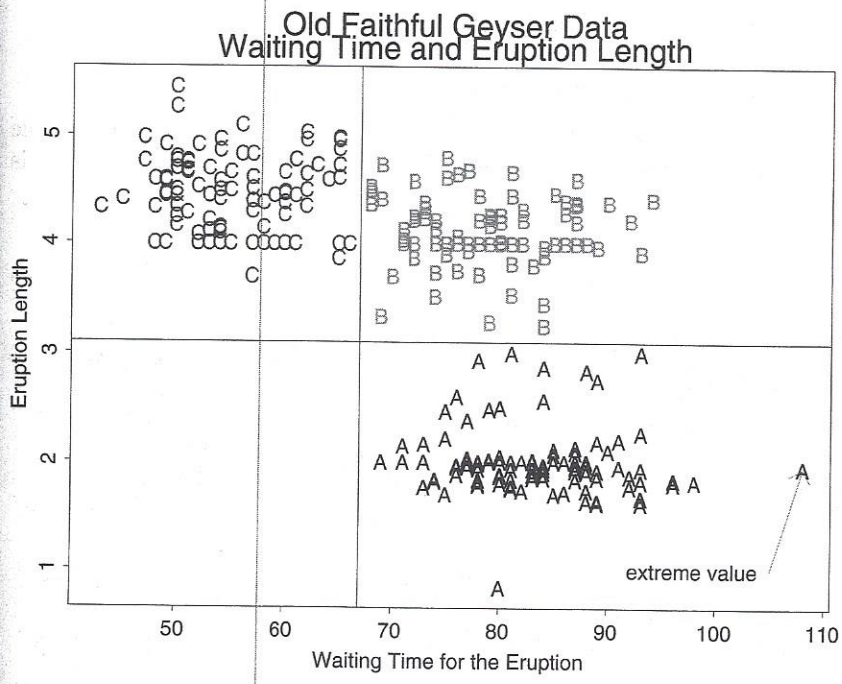

Figure 7.8. The Old Faithful Geyser data in subsets.

```
> subset1 <- (waiting >= 67) & (duration <  3.1)
> subset2 <- (waiting >= 67) & (duration >= 3.1)
> subset3 <- (waiting <  67) & (duration >= 3.1)
```

The next step is to plot the groups separately by using the `points` function, but displaying different characters (instead of points) and using different colors to demarcate the groups.

```
> points(waiting[subset1],duration[subset1],pch="A",col=1)
> points(waiting[subset2],duration[subset2],pch="B",col=2)
> points(waiting[subset3],duration[subset3],pch="C",col=3)
```

Finally, we add an arrow pointing to the extreme point with a waiting time of about 110 minutes. Note that we do not determine the end point of the arrow, but go to its waiting time (which we know is the maximum value) and the corresponding duration (by selecting the index of the maximum waiting time). We want a V-shaped arrow; therefore, `open=T` is set. Finally, we place the text "extreme value" on the bottom line of the plot, ending where the arrow starts (`adj=1` means the text is right-justified). The text we want to add has two extra spaces at the end, such that there is a little gap between the text and the arrow's end point.

```
> arrows(105, 1, max(waiting),
+ duration[waiting==max(waiting)], open=T)
> text(105, 1, "extreme value  ", adj=1)
```

We will now continue with graphical functions that can deal with multiple variables: multivariate graph types. We start with an overview of what is available in Table 7.4.

Table 7.4. Multivariate graphical data exploration functions

S-PLUS Function	Description
`barplot`	Barplot (bars with subbars stacked)
`biplot`	Plot principal components and factor analysis results, show data and variables in a two-dimensional coordinate system
`boxplot`	Boxplot(s) of one or more variables in a single graph
`contour`	Contour lines of a two-dimensional distribution to access the common distribution function
`coplot`	Plot matrix of two variables conditioned on the values of a third variable
`dotchart`	Dotchart (values on parallel lines stacked) for a vector with optional variable grouping
`faces`	Chernoff faces illustrating by means of face elements a high-dimensional data set (up to 15 variables)
`hexbin`	Hexagonal binning, a display technique for spatial data using hexagonally shaped bins
`image`	Image plots, color scales for (geographical) heights or density values on a two-dimensional grid
`matplot`	Plotting columns of matrices against other matrix columns by using different symbols for each combination
`pairs`	Two-dimensional x-y scatterplots in a matrix of scatterplots for each combination of variables
`persp`	Perspective three-dimensional surface plots on a grid for density estimates
`stars`	Star plots of multivariate data in which each observation is displayed as a star. Each point is further away from the middle the larger the value is

See also Table 6.1 (page 143) for Trellis graphics functions.

We have already seen that the function `hist2d` is useful for putting continuous data like the Geyser data into a table frame. If we do it with our Geyser data set, we get back a list with five components, telling us the mid-values of the chosen intervals and the end points, together with the tabulated data.

```
> hist2d(waiting, duration)
   $x:
   [1]  45  55  65  75  85  95 105

   $y:
   [1] 0.5 1.5 2.5 3.5 4.5 5.5

   $z:
             0 to 1 1 to 2 2 to 3 3 to 4 4 to 5 5 to 6
  40 to  50      0      0      0      0     16      0
  50 to  60      0      0      0      1     56      3
  60 to  70      0      0      1      2     28      1
  70 to  80      0     13     19      9     40      0
  80 to  90      1     31     25      9     24      0
  90 to 100      0     11      3      2      3      0
 100 to 110      0      1      0      0      0      0

   $xbreaks:
   [1]  40  50  60  70  80  90 100 110

   $ybreaks:
   [1] 0 1 2 3 4 5 6
```

The table is put into the component `z`, and we see that the visibility of effects depends heavily on the intervals chosen. The effect of the three point clouds is not that clearly visible, although you might guess at them by studying the values. Experimenting with different choices of interval bounds might give more insights.

You can use the `hist2d` function introduced before to set up data to be used as input to other functions. The `persp` function produces a 3D-type plot that accepts input from `hist2d`.

```
> persp(hist2d(waiting, duration))
```

Figure 7.9 shows a perspective plot of the surface of the empirical distribution. The x-axis and y-axis are set up by taking the variables `xbreaks` and `ybreaks` of the `hist2d` output, and the matrix of values in the `z` component determines surface height. Try to reveal the three peaks more clearly by choosing another set of intervals for `hist2d` to display the perspective surface plot.

You can use the same data from `hist2d` to get an image display of the surface. The image display and the corresponding S-PLUS function `image` are often used to display geographic data, using latitude and longitude as axes, and the geographical height of the area is color coded. This is the principle of every contour map. We use the color display to look at the tabulated Geyser data "from the top," getting the heights in different colors.

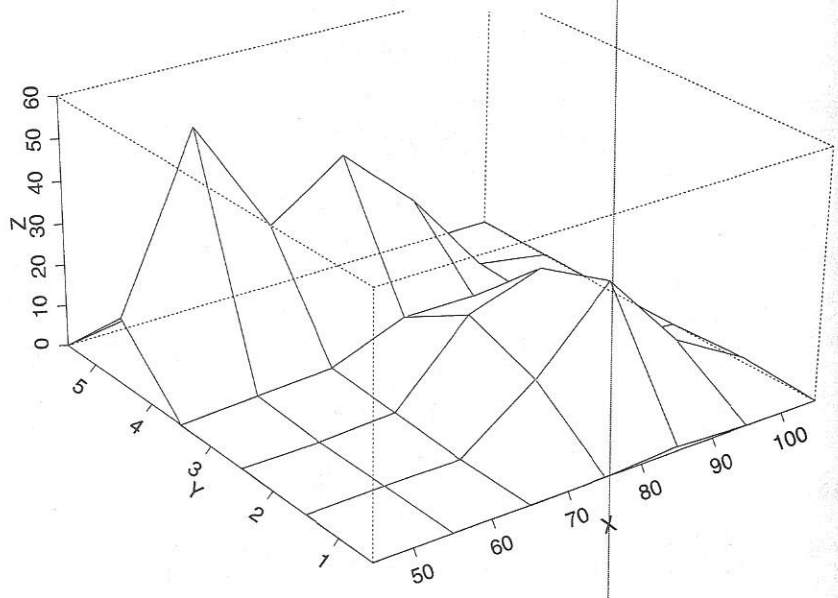

Figure 7.9. A perspective plot of the Old Faithful Geyser data.

```
> image(hist2d(waiting, duration,
+ xbreaks=seq(40, 110, by=5), ybreaks=seq(0, 6, by=0.5))
> title("An image plot of the geyser data")
```

For a nicer display, we chose different classes than the default settings used above. The result is shown in Figure 7.10, and although the loss of the coloring in this picture is a disadvantage, we can still see the three "mountains" as well as the "desert area" in the lower left corner. Other options available are different color schemes, like a "heat color scheme" ranging from cold (blue) to hot (red).

It becomes obvious that we need to experiment with parameters if we examine Figure 7.11. We show two pictures of the same data, but the underlying categorizations into tables are different. From the picture on the left-hand side we see the three groups in the data, whereas the picture on the right-hand side leads us to conclude that we have only two mountain tops.

This figure shows contour lines or, in other words, height lines of a data set. The `contour` function also accepts input from `hist2d`.

```
> par(mfrow=c(1,2))
> contour(hist2d(waiting, duration),
+ xlab="waiting time", ylab="eruption time")
> title("Contour plot (1)")
```

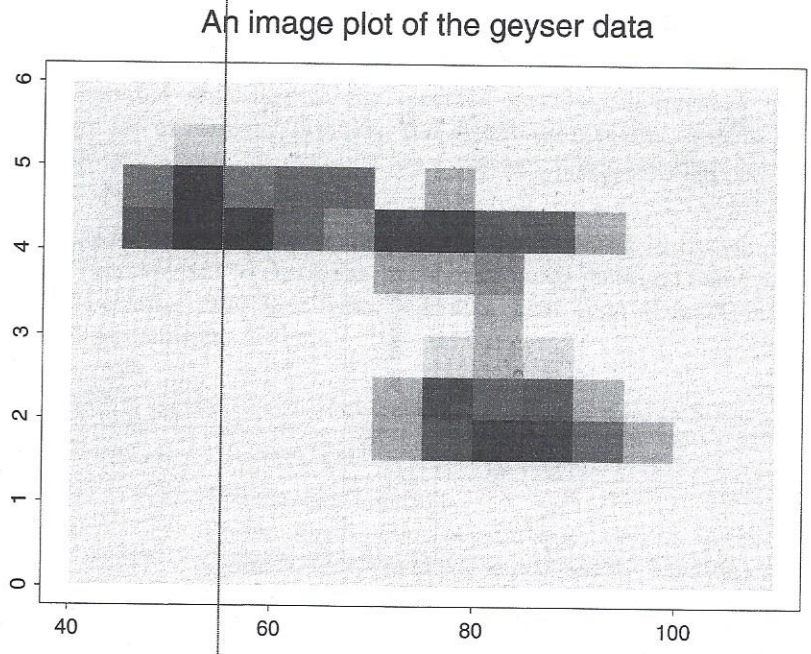

Figure 7.10. An image plot of the Old Faithful Geyser data.

```
> contour(hist2d(waiting, duration,
+ xbreaks=seq(40, 110, by=5), ybreaks=seq(0, 6, by=1.5)),
+ xlab="waiting time", ylab="eruption time")
> title("Contour plot (2)")
```

This excursion into data analysis using graphics for exploring structure reveals that the term "exploratory data analysis" describes a search for effects discovered by using many different displays and summaries of the same data set. A single number or display can never describe the whole complexity of a data set.

If you want to explore the Barley or Geyser data sets with interactive techniques and rotating point clouds, try out the functions listed in Table 7.5. Use the mouse to define a brush and move it over point clouds to mark ("highlight") them. If you get stuck, consult the manuals. Just do it.

Table 7.5. Multivariate dynamic graphics functions

S-PLUS Function	Description
`brush`	Interactive marking of subsets on 2D scatterplots
`spin`	Rotating data, marking and highlighting of subsets

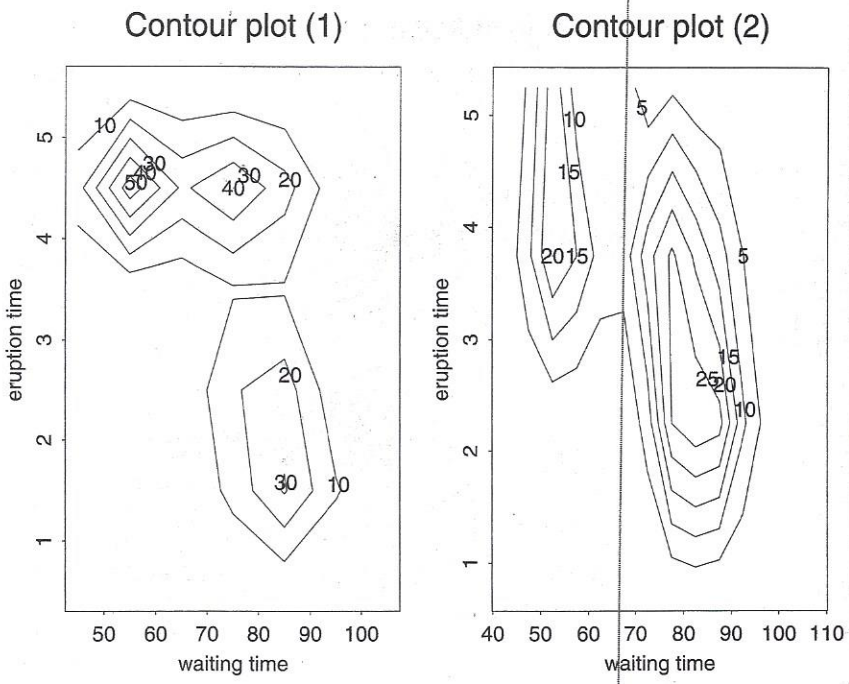

Figure 7.11. A contour plot of the Old Faithful Geyser data.

7.3 Distributions and Related Functions

This section examines a variety of distributions covered by built-in S-PLUS functions. As we will see later, adding further distributions to the existing set is also very straightforward. We divide the distributions themselves into continuous and discrete distributions.

Statistical distributions occur in many practical data analysis situations. Most models to describe data behavior are based on distributional assumptions and derive the estimates for unknown model parameters from the underlying (assumed) distributions.

The most important functions related to a distribution are as follows.

- The density function, which specifies a random variate distribution (with fixed parameters).
- The probability function, which is the integral over the distribution function or, in the discrete case, the sum up to the specified point. In other words, the probability function at x gives the probability of the random variable X being less than or equal to x.

- The quantile function, which is the inverse of the probability function.
- The random number generating function, which generates numbers distributed according to the specified distribution function. For a large set of random numbers, the distribution of the random numbers (visualized, for example, by a histogram), looks approximately like the density function.

S-PLUS has a systematic naming of functions related to distributions. The functions related to the same distribution have the same name except for the first letter. The first letter indicates what type of function it is. Table 7.6 explains the system.

Table 7.6. Categorization of distribution-related functions in S-PLUS

Type[1] Character	Function Type
d	Distribution function
p	Probability function or cumulated density function
q	Quantile function (inverse probability function)
r	Random number generation

[1] The significant letter for identifying the functionality of the distribution function is either d, p, q, or r. See also Table 7.7.

To identify a specific function, we need to know the abbreviations for the distributions available to us, which are listed in Table 7.7 and Table 7.8. To determine the function's name, combine the characteristic letter from Table 7.6 with the abbreviation in Table 7.7 or 7.8. For example, to generate random numbers from the Normal distribution, the function to use is `rnorm` ("r" + "norm").

Table 7.7. Distribution-related functions in S-PLUS(1): Discrete distributions

Distribution	S-PLUS Name[1]	Parameters[2]	
Binomial	`binom`	size	prob
Geometric	`geom`	prob	
Hypergeometric	`hyper`	m, n, k	
Neg. binomial	`nbinom`	size	prob
Poisson	`pois`	lambda	
Wilcoxon (rank sum)	`wilcox`	m	n

[1] The characteristic letter, one of d, p, q, r, plus the listed name, gives the name of the S-PLUS function.
Use the `sample` function to generate random numbers from a discrete distribution.

Table 7.8. Distribution-related functions in S-PLUS(2): Continuous distributions

Distribution	S-PLUS Name[1]	Parameters[2]	
Beta	`beta`	shape1	shape2
Cauchy	`cauchy`	location=0	shape=1
Chi-square	`chisq`	df	
Exponential	`exp`	rate=1	
F	`f`	df1	df2
Gamma	`gamma`	shape	
Logistic	`logis`	location=0	scale=1
Lognormal	`lnorm`	meanlog=0	sdlog=1
Normal	`norm`	mean=0	sd=1
Normal, multivariate[3]	`mvnorm`	mean=rep(0,d)	cov=diag(d)
Stable	`stab`	index	skewness=0
Student t	`t`	df	
Uniform	`unif`	min=0	max=1
Weibull	`weibull`	shape	

[1] The characteristic letter, one of d, p, q, r, plus the listed name, gives the name of the S-PLUS function.
[2] If parameters are preset with a default value, like in mean=0, they do not need to be specified unless another parameter value than the default is desired.
[3] There is no quantile function for the multivariate Normal distribution.

There are four functions related to the Normal distribution

`dnorm` for calculating the value of the distribution function

`pnorm` for calculating the value of the probability function

`qnorm` for calculating the inverse probability function

`rnorm` for generating random numbers from the Normal distribution

These four functions all have the same parameters, as they refer to the same distribution, namely `mean` and `sd`, the mean and the standard deviation, respectively. Both arguments are optional. They are preset to define the standard Normal distribution with mean 0 and standard deviation 1.

Note The standard deviation, not the variance, is the argument used for dispersion by the functions for the Normal distribution. The standard deviation is simply the square root of the variance. ◁

All these functions take numbers and vectors as input arguments. For a vector, the corresponding value for each element is computed. For example,

```
> dnorm(-3:3, 0, 1:7)
```

calculates `dnorm(-3, 0, 1)`, `dnorm(-2, 0, 2)`, ..., `dnorm(3, 0, 7)`.

You can check out to see the numerical accuracy of the distributional functions. Calculate the difference between some values x and p(q(x)) or q(p(x)), which should be 0 in theory (p stands for the probability and q for the quantile function of the distribution).

Note As you might know, the random numbers generated by a computer are never really random. They consist of a sequence of numbers, where each number depends somehow on the previous number generated. S-PLUS uses the variable `.Random.seed` to store the state of the random number generator. You can use `set.seed(`n`)`, where n is an integer number, to produce the same sequence of random numbers several times, maybe to test a self-written routine. ◁

Graphing Distributions

You will often want to see what a distribution looks like, which usually means looking at the density function. Here, we need to make a distinction between continuous and discrete distributions.

Recall that a continuous density has a continuous support; that is, all values of the density function in a given interval are greater than zero. This interval usually ranges from minus infinity to plus infinity. For this reason, we cannot graph the "whole" distribution from minus to plus infinity, but have to restrict ourselves to a part of it. A good idea is to plot, for example, 90% of the density support.

To calculate these 90% limits, we cut 5% off each side and get the boundaries by calculating the 5% and 95% quantiles. As an example, we plot the standard Normal(0, 1), t(5), and standard Cauchy distribution and calculate the boundaries as described.

```
> x <- c(0.05, 0.95)
> bounds.normal <- qnorm(x)
> bounds.t      <- qt(x, 5)
> bounds.cauchy <- qcauchy(x)
```

Next, we create a sequence of points at which we want to calculate the densities. Therefore, we calculate the common range of the boundaries of the three distributions.

```
> bounds <- range(bounds.normal, bounds.t, bounds.cauchy)
> points.x <- seq(bounds[1], bounds[2], length=1000)
```

Now the values of the density functions are easily obtained.

```
> points.normal <- dnorm(points.x)
> points.t      <- dt(points.x, 5)
> points.cauchy <- dcauchy(points.x)
```

Plotting the functions is straightforward now. Be sure to calculate the maximum value of the y-axis data first, in order to have all values inside the plotting range. A trick is to plot the "most outward" curve first, the curve having the largest and smallest values, so that the other ones will fit into the given range. We calculate the boundaries first, plot an empty frame, and then add the three lines.

```
> yrange <- range(points.normal, points.t, points.cauchy)
> plot(0, 0, type="n", xlim=bounds,
+ ylim=yrange, xlab="", ylab="Density Value")
> lines(points.x, points.normal, col=1, lty=1)
> lines(points.x, points.t,      col=2, lty=2)
> lines(points.x, points.cauchy, col=3, lty=3)
```

Finally, we add a legend using the function `key` and a title.

```
> key(min(bounds), max(yrange), corner=c(0,1),
+ lines=list(lty=1:3, col=1:3),
+ text=list(paste(c("Normal(0,1)", "t(5)", "Cauchy"),
+ "Distribution"), adj=0, cex=0.8))
> title("A graphical comparison of distributions")
```

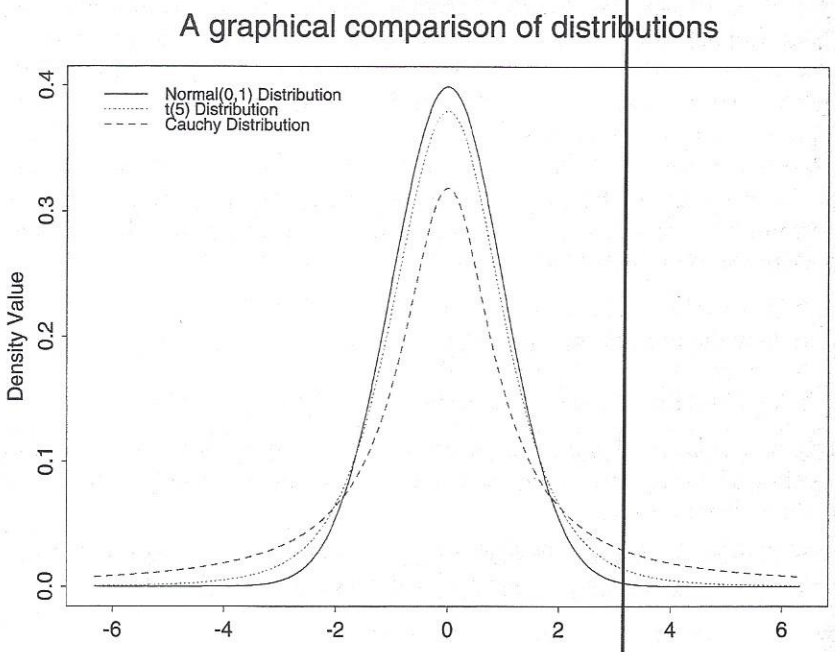

Figure 7.12. A graphical comparison of three distributions.