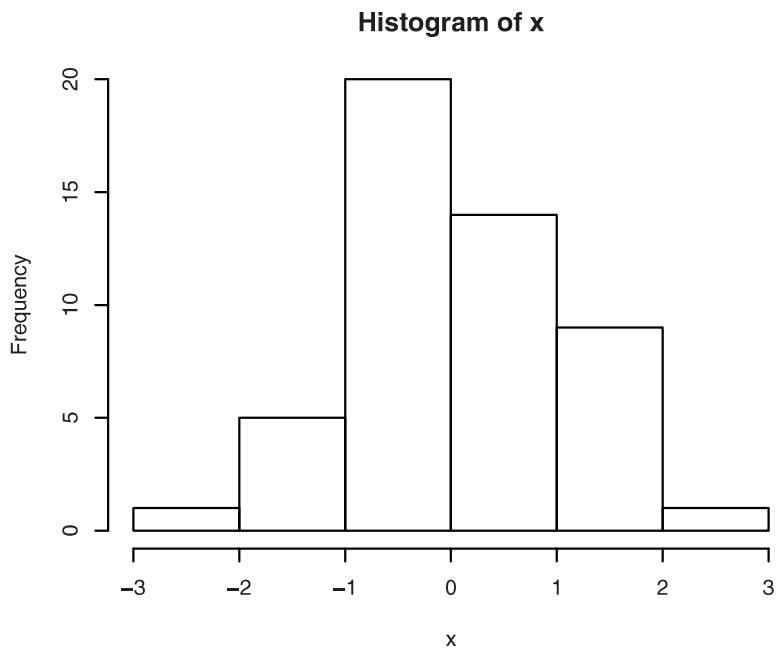

Figure 4.1. Histogram.

4.2 Graphical display of distributions

4.2.1 Histograms

You can get a reasonable impression of the shape of a distribution by drawing a histogram; that is, a count of how many observations fall within specified divisions ("bins") of the *x*-axis (Figure 4.1).

```
> hist(x)
```

By specifying `breaks=n` in the `hist` call, you get *approximately n* bars in the histogram since the algorithm tries to create "pretty" cutpoints. You can have full control over the interval divisions by specifying `breaks` as a vector rather than as a number. Altman (1991, pp. 25–26) contains an example of accident rates by age group. These are given as a count in age groups 0–4, 5–9, 10–15, 16, 17, 18–19, 20–24, 25–59, and 60–79 years of age. The data can be entered as follows:

```
> mid.age <- c(2.5,7.5,13,16.5,17.5,19,22.5,44.5,70.5)
> acc.count <- c(28,46,58,20,31,64,149,316,103)
> age.acc <- rep(mid.age,acc.count)
```

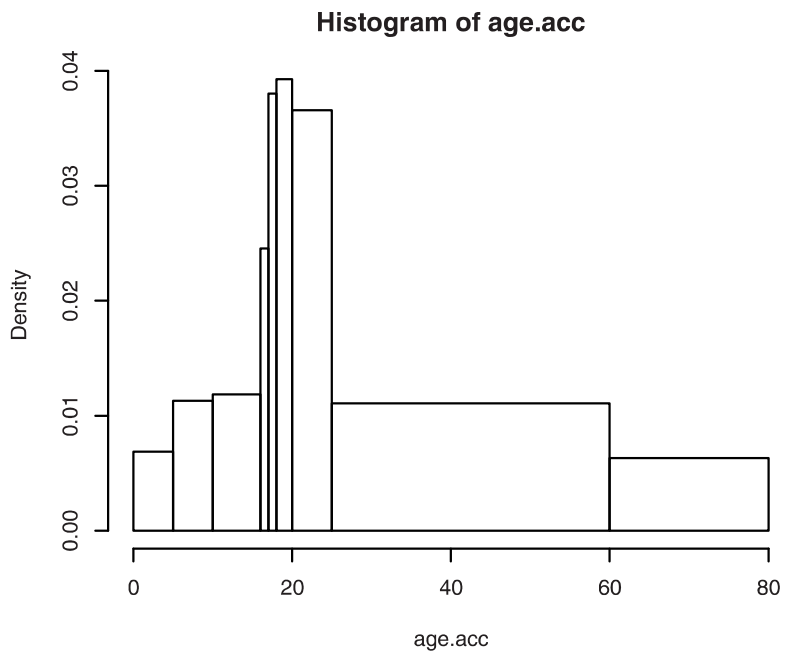

Figure 4.2. Histogram with unequal divisions.

```
> brk <- c(0,5,10,16,17,18,20,25,60,80)
> hist(age.acc,breaks=brk)
```

Here the first three lines generate pseudo-data from the table in the book. For each interval, the relevant number of "observations" is generated with an age set to the midpoint of the interval; that is, 28 2.5-year-olds, 46 7.5-year-olds, etc. Then a vector `brk` of cutpoints is defined (note that the extremes need to be included) and used as the `breaks` argument to `hist`, yielding Figure 4.2.

Notice that you automatically got the "correct" histogram where the *area* of a column is proportional to the number. The y-axis is in density units (that is, proportion of data per x unit), so that the total area of the histogram will be 1. If, for some reason, you want the (misleading) histogram where the column height is the raw number in each interval, then it can be specified using `freq=T`. For equidistant breakpoints, that is the default (because then you can see how many observations have gone into each column), but you can set `freq=F` to get densities displayed. This is really just a change of scale on the y-axis, but it has the advantage that it becomes possible to overlay the histogram with a corresponding theoretical density function.

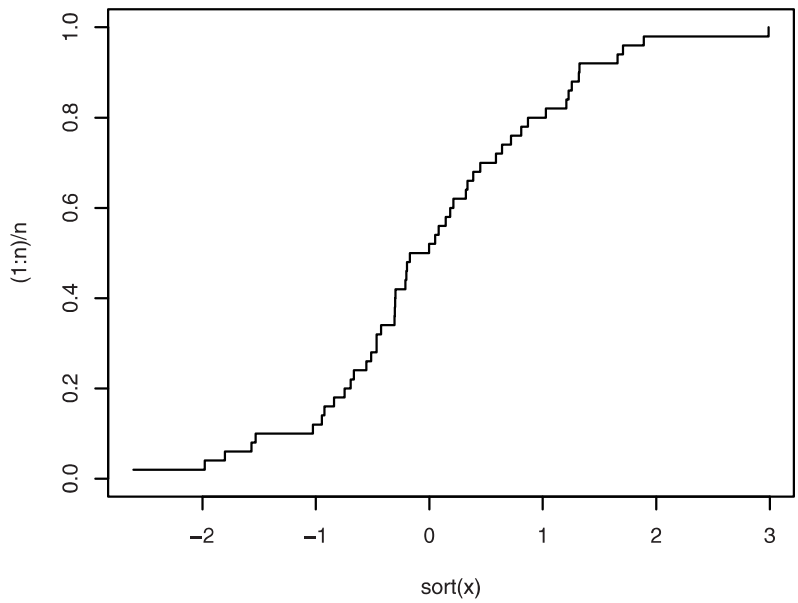

Figure 4.3. Empirical cumulative distribution function.

4.2.2 *Empirical cumulative distribution*

The empirical cumulative distribution function is defined as the fraction of data smaller than or equal to x. That is, if x is the kth smallest observation, then the proportion k/n of the data is smaller than or equal to x (7/10 if x is no. 7 of 10). The empirical cumulative distribution function can be plotted as follows (see Figure 4.3) where `x` is the simulated data vector from Section 4.1:

```
> n <- length(x)
> plot(sort(x),(1:n)/n,type="s",ylim=c(0,1))
```

The plotting parameter `type="s"` gives a *s*tep function where (x, y) is the left end of the steps and `ylim` is a vector of two elements specifying the extremes of the y-coordinates on the plot. Recall that `c(...)` is used to create vectors.

Some more elaborate displays of empirical cumulative distribution functions are available via the `ecdf` function. This is also more precise regarding the mathematical definition of the step function.

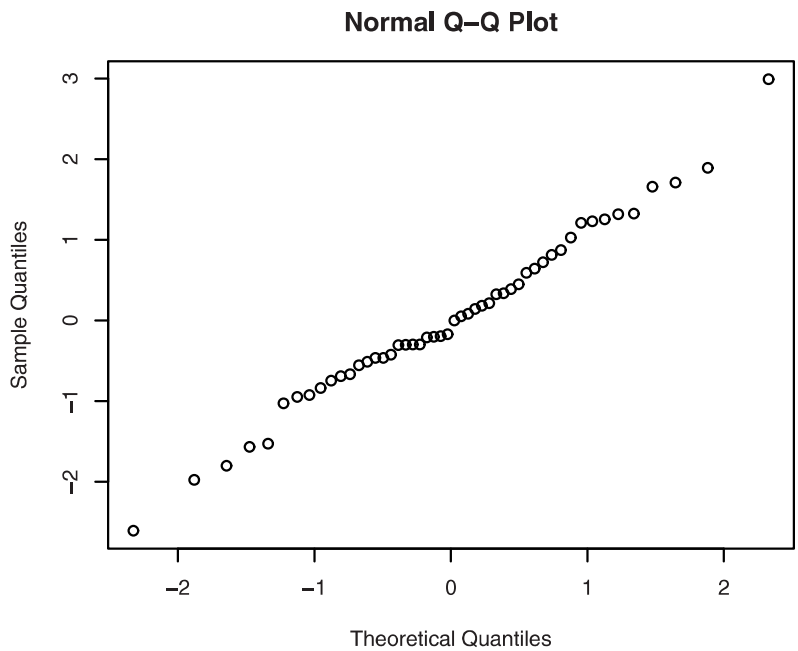

Figure 4.4. Q–Q plot using `qqnorm(x)`.

4.2.3 *Q–Q plots*

One purpose of calculating the empirical cumulative distribution function (c.d.f.) is to see whether data can be assumed normally distributed. For a better assessment, you might plot the kth smallest observation against the expected value of the kth smallest observation out of n in a standard normal distribution. The point is that in this way you would expect to obtain a straight line if data come from a normal distribution with *any* mean and standard deviation.

Creating such a plot is slightly complicated. Fortunately, there is a built-in function for doing it, `qqnorm`. The result of using it can be seen in Figure 4.4. You only have to write

```
> qqnorm(x)
```

As the title of the plot indicates, plots of this kind are also called "Q–Q plots" (quantile versus quantile). Notice that x and y are interchanged relative to the empirical c.d.f. — the observed values are now drawn along the y-axis. You should notice that with this convention the distribution has heavy tails if the outer parts of the curve are steeper than the middle part.

Some readers will have been taught "probability plots", which are similar but have the axes interchanged. It can be argued that the way R draws the plot is the better one since the theoretical quantiles are known in advance, while the empirical quantiles depend on data. You would normally choose to draw fixed values horizontally and variable values vertically.

4.2.4 Boxplots

A "boxplot", or more descriptively a "box-and-whiskers plot", is a graphical summary of a distribution. Figure 4.5 shows boxplots for IgM and its logarithm; see the example on page 23 in Altman (1991).

Here is how a boxplot is drawn in R. The box in the middle indicates "hinges" (nearly quartiles; see the help page for `boxplot.stats`) and median. The lines ("whiskers") show the largest or smallest observation that falls within a distance of 1.5 times the box size from the nearest hinge. If any observations fall farther away, the additional points are considered "extreme" values and are shown separately.

The practicalities are these:

```
> par(mfrow=c(1,2))
> boxplot(IgM)
> boxplot(log(IgM))
> par(mfrow=c(1,1))
```

A layout with two plots side by side is specified using the `mfrow` graphical parameter. It should be read as "*m*ulti*f*rame, *row*wise, 1×2 layout". Individual plots are organized in one row and two columns. As you might guess, there is also an `mfcol` parameter to plot columnwise. In a 2×2 layout, the difference is whether plot no. 2 is drawn in the top right or bottom left corner.

Notice that it is necessary to reset the layout parameter to `c(1,1)` at the end unless you also want two plots side by side subsequently.

4.3 Summary statistics by groups

When dealing with grouped data, you will often want to have various summary statistics computed within groups; for example, a table of means and standard deviations. To this end, you can use `tapply` (see Section 1.2.15). Here is an example concerning the folate concentration in red blood cells according to three types of ventilation during anesthesia (Alt-

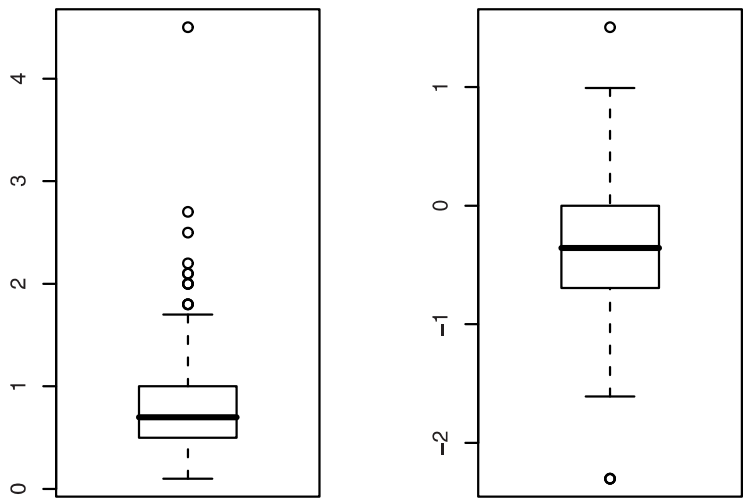

Figure 4.5. Boxplots for IgM and log IgM.

man, 1991, p. 208). We return to this example in Section 7.1, which also contains the explanation of the category names.

```
> attach(red.cell.folate)
> tapply(folate,ventilation,mean)
N2O+O2,24h  N2O+O2,op     O2,24h
  316.6250   256.4444   278.0000
```

The `tapply` call takes the `folate` variable, splits it according to `ventilation`, and computes the mean for each group. In the same way, standard deviations and the number of observations in the groups can be computed.

```
> tapply(folate,ventilation,sd)
N2O+O2,24h  N2O+O2,op     O2,24h
  58.71709   37.12180   33.75648
> tapply(folate,ventilation,length)
N2O+O2,24h  N2O+O2,op     O2,24h
         8          9          5
```

Try something like this for a nicer display:

```
> xbar <- tapply(folate, ventilation, mean)
> s <- tapply(folate, ventilation, sd)
> n <- tapply(folate, ventilation, length)
> cbind(mean=xbar, std.dev=s, n=n)
                mean  std.dev n
N2O+O2,24h 316.6250 58.71709 8
N2O+O2,op  256.4444 37.12180 9
O2,24h     278.0000 33.75648 5
```

For the `juul` data, we might want the mean `igf1` by `tanner` group, but of course we run into the problem of missing values again:

```
> tapply(igf1, tanner, mean)
  I  II III  IV   V
 NA  NA  NA  NA  NA
```

We need to get `tapply` to pass `na.rm=T` as a parameter to `mean` to make it exclude the missing values. This is achieved simply by passing it as an additional argument to `tapply`.

```
> tapply(igf1, tanner, mean, na.rm=T)
       I       II      III       IV        V
207.4727 352.6714 483.2222 513.0172 465.3344
```

The functions `aggregate` and `by` are variations on the same topic. The former is very much like `tapply`, except that it works on an entire data frame and presents its results as a data frame. This is useful for presenting many variables at once; e.g.,

```
> aggregate(juul[c("age","igf1")],
+           list(sex=juul$sex), mean, na.rm=T)
  sex      age     igf1
1   M 15.38436 310.8866
2   F 14.84363 368.1006
```

Notice that the grouping argument in this case must be a list, even when it is one-dimensional, and that the names of the list elements get used as column names in the output. Notice also that since the function is applied to all columns of the data frame, you may have to choose a subset of columns, in this case the numeric variables.

The indexing variable is not necessarily part of the data frame that is being aggregated, and there is no attempt at "smart evaluation" as there is in `subset`, so you have to spell out `juul$sex`. You can also use the fact that data frames are list-like and say

```
> aggregate(juul[c("age","igf1")], juul["sex"], mean, na.rm=T)
  sex      age     igf1
1   M 15.38436 310.8866
2   F 14.84363 368.1006
```

(the "trick" being that indexing a data frame with single brackets yields a data frame as the result).

The `by` function is again similar, but different. The difference is that the function now takes an entire (sub-) data frame as its argument, so that you can for instance summarize the Juul data by sex as follows:

```
> by(juul, juul["sex"], summary)
sex: M
      age        menarche   sex          igf1        tanner
 Min.   : 0.17   No  :  0   M:621   Min.   : 29.0   I   :291
 1st Qu.: 8.85   Yes :  0   F:  0   1st Qu.:176.0   II  : 55
 Median :12.38   NA's:621           Median :280.0   III : 34
 Mean   :15.38                      Mean   :310.9   IV  : 41
 3rd Qu.:16.77                      3rd Qu.:430.2   V   :124
 Max.   :83.00                      Max.   :915.0   NA's: 76
                                    NA's   :145.0
    testvol
 Min.   :  1.000
 1st Qu.:  1.000
 Median :  3.000
 Mean   :  7.896
 3rd Qu.: 15.000
 Max.   : 30.000
 NA's   :141.000
------------------------------------------------
sex: F
      age        menarche   sex          igf1        tanner
 Min.   : 0.25   No  :369   M:  0   Min.   : 25.0   I   :224
 1st Qu.: 9.30   Yes :335   F:713   1st Qu.:233.0   II  : 48
 Median :12.80   NA's:  9           Median :352.0   III : 38
 Mean   :14.84                      Mean   :368.1   IV  : 40
 3rd Qu.:16.93                      3rd Qu.:483.0   V   :204
 Max.   :75.12                      Max.   :914.0   NA's:159
                                    NA's   :176.0
    testvol
 Min.   : NA
 1st Qu.: NA
 Median : NA
 Mean   :NaN
 3rd Qu.: NA
 Max.   : NA
 NA's   :713
```

The result of the call to `by` is actually a list of objects that has has been wrapped as an object of class `"by"` and printed using a print method for that class. You can assign the result to a variable and access the result for each subgroup using standard list indexing.

The same technique can also be used to generate more elaborate statistical analyses for each group.

4.4 Graphics for grouped data

In dealing with grouped data, it is important to be able not only to create plots for each group but also to compare the plots between groups. In this section we review some general graphical techniques that allow us to display similar plots for several groups on the same page. Some functions have specific features for displaying data from more than one group.

4.4.1 Histograms

We have already seen in Section 4.2.1 how to obtain a histogram simply by typing `hist(x)`, where `x` is the variable containing the data. R will then choose a number of groups so that a reasonable number of data points fall in each bin while at the same time ensuring that the cutpoints are "pretty" numbers on the x-axis.

It is also mentioned there that an alternative number of intervals can be set via the argument `breaks`, although you do not always get exactly the number you asked for since R reserves the right to choose "pretty" column boundaries. For instance, multiples of 0.5 MJ are chosen in the following example using the `energy` data introduced in Section 1.2.14 on the 24-hour energy expenditure for two groups of women.

In this example, some further techniques of general use are illustrated. The end result is seen in Figure 4.6, but first we must fetch the data:

```
> attach(energy)
> expend.lean <- expend[stature=="lean"]
> expend.obese <- expend[stature=="obese"]
```

Notice how we separate the `expend` vector in the `energy` data frame into two vectors according to the value of the factor `stature`.

Now we do the actual plotting:

```
> par(mfrow=c(2,1))
> hist(expend.lean,breaks=10,xlim=c(5,13),ylim=c(0,4),col="white")
> hist(expend.obese,breaks=10,xlim=c(5,13),ylim=c(0,4),col="grey")
> par(mfrow=c(1,1))
```

We set `par(mfrow=c(2,1))` to get the two histograms in the same plot. In the `hist` commands themselves, we used the `breaks` argument as already mentioned and `col`, whose effect should be rather obvious. We also used `xlim` and `ylim` to get the same x and y axes in the two plots. However, it is a coincidence that the columns have the same width.

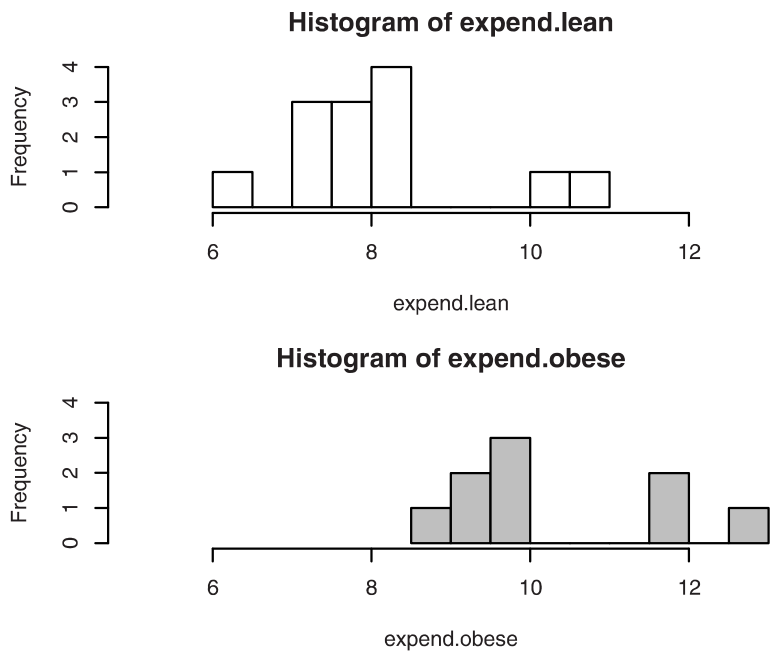

Figure 4.6. Histograms with refinements.

As a practical remark, when working with plots like the above, where more than a single line of code is required, it gets cumbersome to use command recall in the R console window every time something needs to be changed. A better idea may be to start up a script window or a plain-text editor and cut and paste entire blocks of code from there (see Section 2.1.3). You might also take it as an incentive to start writing simple functions.

4.4.2 *Parallel boxplots*

You might want a set of boxplots from several groups in the same frame. `boxplot` can handle this both when data are given in the form of separate vectors from each group and when data are in one long vector and a parallel vector or factor defines the grouping. To illustrate the latter, we use the `energy` data introduced in Section 1.2.14.

Figure 4.7 is created as follows:

```
> boxplot(expend ~ stature)
```

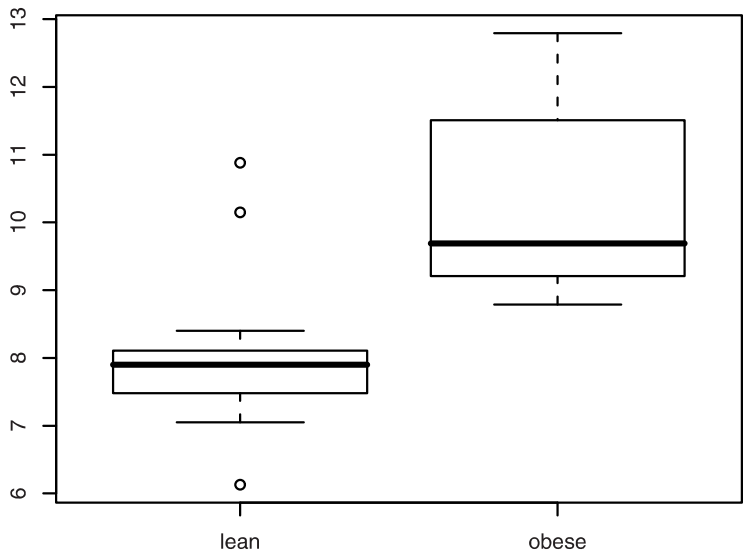

Figure 4.7. Parallel boxplot.

We could also have based the plot on the separate vectors `expend.lean` and `expend.obese`. In that case, a syntax is used that specifies the vectors as two separate arguments:

```
> boxplot(expend.lean,expend.obese)
```

The plot is not shown here, but the only difference lies in the labelling of the x-axis. There is also a third form, where data are given as a single argument that is a list of vectors.

The bottom plot has been made using the complete `expend` vector and the grouping variable `fstature`.

Notation of the type `y ~ x` should be read "`y` described using `x`". This is the first example we see of a *model formula*. We see many more examples of model formulas later on.

4.4.3 Stripcharts

The boxplots made in the preceding section show a "Laurel & Hardy" effect that is not really well founded in the data. The cause is that the in-

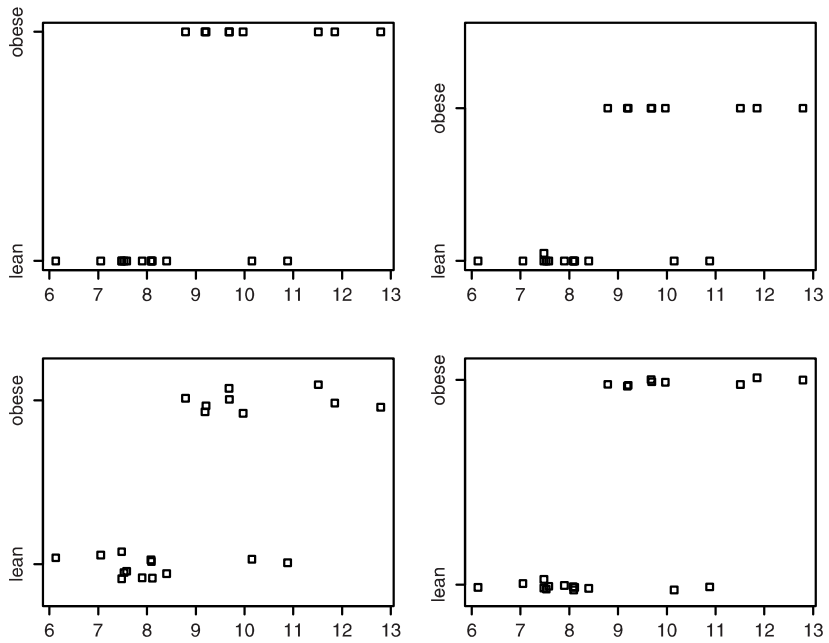

Figure 4.8. Stripcharts in four variations.

terquartile range is quite a bit larger in one group than in the other, making the boxplot appear "fatter". With groups as small as these, the quartiles will be quite inaccurately determined, and it may therefore be more desirable to plot the raw data. If you were to do this by hand, you might draw a dot diagram where every number is marked with a dot on a number line. R's automated variant of this is the function `stripchart`. Four variants of stripcharts are shown in Figure 4.8.

The four plots were created as follows:

```
> opar <- par(mfrow=c(2,2), mex=0.8, mar=c(3,3,2,1)+.1)
> stripchart(expend ~ stature)
> stripchart(expend ~ stature, method="stack")
> stripchart(expend ~ stature, method="jitter")
> stripchart(expend ~ stature, method="jitter", jitter=.03)
> par(opar)
```

Notice that a little `par` magic was used to reduce the spacing between the four plots. The `mex` setting reduces the interline distance, and `mar` reduces the number of lines that surround the plot region. This can be done for these plots since they have neither main title, subtitle, nor x and y labels.

All the original values of the changed settings can be stored in a variable (here `opar`) and reestablished with `par(opar)`.

The first plot is a standard stripchart, where the points are simply plotted on a line. The problem with this is that some points can become invisible because they are overplotted. This is why there is a `method` argument, which can be set to either `"stack"` or `"jitter"`.

The former method stacks points with identical values, but it only does so if data are *completely identical*, so in the upper right plot, it is only the two replicates of 7.48 that get stacked, whereas 8.08, 8.09, and 8.11 are still plotted in almost the same spot.

The "jitter" method offsets all points a random amount vertically. The standard jittering on plot no. 3 (bottom left) is a bit large; it may be preferable to make it clearer that data are placed along a horizontal line. For that purpose, you can set `jitter` lower than the default of 0.1, which is done in the fourth plot.

In this example we have not bothered to specify data in several forms as we did for `boxplot` but used `expend~stature` throughout. We could also have written

```
stripchart(list(lean=expend.lean, obese=expend.obese))
```

but `stripchart(expend.lean, expend.obese)` cannot be used.

4.5 Tables

Categorical data are usually described in the form of tables. This section outlines how you can create tables from your data and calculate relative frequencies.

4.5.1 Generating tables

We deal mainly with two-way tables. In the first example, we enter a table directly, as is required for tables taken from a book or a journal article.

A two-way table can be entered as a matrix object (Section 1.2.7). Altman (1991, p. 242) contains an example on caffeine consumption by marital status among women giving birth. That table may be input as follows:

```
> caff.marital <- matrix(c(652,1537,598,242,36,46,38,21,218
+ ,327,106,67),
+ nrow=3,byrow=T)
> caff.marital
     [,1] [,2] [,3] [,4]
[1,]  652 1537  598  242
[2,]   36   46   38   21
[3,]  218  327  106   67
```

The `matrix` function needs an argument containing the table values as a single vector and also the number of rows in the argument `nrow`. By default, the values are entered columnwise; if rowwise entry is desired, then you need to specify `byrow=T`.

You might also give the number of columns instead of rows using `ncol`. If exactly one of `ncol` and `nrow` is given, R will compute the other one so that it fits the number of values. If both `ncol` and `nrow` are given and it does not fit the number of values, the values will be "recycled", which in some (other!) circumstances can be useful.

To get readable printouts, you can add row and column names to the matrices.

```
> colnames(caff.marital) <- c("0","1-150","151-300",">300")
> rownames(caff.marital) <- c("Married","Prev.married","Single")
> caff.marital
               0 1-150 151-300 >300
Married      652  1537     598  242
Prev.married  36    46      38   21
Single       218   327     106   67
```

Furthermore, you can name the row and column names as follows. This is particularly useful if you are generating many tables with similar classification criteria.

```
> names(dimnames(caff.marital)) <- c("marital","consumption")
> caff.marital
              consumption
marital          0 1-150 151-300 >300
  Married      652  1537     598  242
  Prev.married  36    46      38   21
  Single       218   327     106   67
```

Actually, I glossed over something. Tables are not completely equivalent to matrices. There is a `"table"` class for which special methods exist, and you can convert to that class using `as.table(caff.marital)`. The `table` function below returns an object of class `"table"`.

For most elementary purposes, you can use matrices where two-dimensional tables are expected. One important case where you do need `as.table` is when converting a table to a data frame of counts:

```
> as.data.frame(as.table(caff.marital))
          marital consumption Freq
1         Married           0  652
2    Prev.married           0   36
3          Single           0  218
4         Married       1-150 1537
5    Prev.married       1-150   46
6          Single       1-150  327
7         Married     151-300  598
8    Prev.married     151-300   38
9          Single     151-300  106
10        Married        >300  242
11   Prev.married        >300   21
12         Single        >300   67
```

In practice, the more frequent case is that you have a data frame with variables for each person in a data set. In that case, you should do the tabulation with `table`, `xtabs`, or `ftable`. These functions will generally work for tabulating numeric vectors as well as factor variables, but the latter will have their levels used for row and column names automatically. Hence, it is recommended to convert numerically coded categorical data into factors. The `table` function is the oldest and most basic of the three. The two others offer formula-based interfaces and better printing of multiway tables.

The data set `juul` was introduced on p. 68. Here we look at some other variables in that data set, namely `sex` and `menarche`; the latter indicates whether or not a girl has had her first period. We can generate some simple tables as follows:

```
> table(sex)
sex
  M   F
621 713
> table(sex,menarche)
   menarche
sex  No Yes
  M   0   0
  F 369 335
> table(menarche,tanner)
        tanner
menarche   I II III IV   V
     No  221 43  32 14   2
     Yes   1  1   5 26 202
```

Of course, the table of menarche versus sex is just a check on internal consistency of the data. The table of menarche versus Tanner stage of puberty is more interesting.

There are also tables with more than two sides, but not many simple statistical functions use them. Briefly, to tabulate such data, just write, for example, `table(factor1,factor2,factor3)`. To input a table of cell counts, use the `array` function (an analogue of `matrix`).

The `xtabs` function is quite similar to `table` except that it uses a model formula interface. This most often uses a one-sided formula where you just list the classification variables separated by `+`.

```
> xtabs(~ tanner + sex, data=juul)
      sex
tanner   M   F
   I   291 224
   II   55  48
   III  34  38
   IV   41  40
   V   124 204
```

Notice how the interface allows you to refer to variables in a data frame without attaching it. The empty left-hand side can be replaced by a vector of counts in order to handle pretabulated data.

The formatting of multiway tables from `table` or `xtabs` is not really nice; e.g.,

```
> xtabs(~ dgn + diab + coma, data=stroke)
, , coma = No

     diab
dgn    No Yes
  ICH  53   6
  ID  143  21
  INF 411  64
  SAH  38   0

, , coma = Yes

     diab
dgn    No Yes
  ICH  19   1
  ID   23   3
  INF  23   2
  SAH   9   0
```

As you add dimensions, you get more of these two-sided subtables and it becomes rather easy to lose track. This is where `ftable` comes in. This function creates "flat" tables; e.g., like this:

```
> ftable(coma + diab ~ dgn, data=stroke)
    coma  No     Yes
    diab  No Yes  No Yes
dgn
ICH       53   6  19   1
ID       143  21  23   3
INF      411  64  23   2
SAH       38   0   9   0
```

That is, variables on the left-hand side tabulate across the page and those on the right tabulate downwards. `ftable` works on raw data as shown, but its `data` argument can also be a table as generated by one of the other functions.

Like any matrix, a table can be transposed with the `t` function:

```
> t(caff.marital)
           marital
consumption Married Prev.married Single
    0           652           36    218
    1-150      1537           46    327
    151-300     598           38    106
    >300        242           21     67
```

For multiway tables, exchanging indices (generalized transposition) is done by `aperm`.

4.5.2 *Marginal tables and relative frequency*

It is often desired to compute marginal tables; that is, the sums of the counts along one or the other dimension of a table. Due to missing values, this might not coincide with just tabulating a single factor. This is done fairly easily using the `apply` function (Section 1.2.15), but there is also a simplified version called `margin.table`, described below.

First, we need to generate the table itself:

```
> tanner.sex <- table(tanner,sex)
```

(`tanner.sex` is an arbitrarily chosen name for the crosstable.)

```
> tanner.sex
      sex
tanner   M   F
   I   291 224
   II   55  48
   III  34  38
   IV   41  40
   V   124 204
```

Then we compute the marginal tables:

```
> margin.table(tanner.sex,1)
tanner
  I  II III  IV   V
515 103  72  81 328
> margin.table(tanner.sex,2)
sex
  M   F
545 554
```

The second argument to `margin.table` is the number of the marginal index: 1 and 2 give row and column totals, respectively.

Relative frequencies in a table are generally expressed as proportions of the row or column totals. Tables of relative frequencies can be constructed using `prop.table` as follows:

```
> prop.table(tanner.sex,1)
      sex
tanner         M         F
   I   0.5650485 0.4349515
   II  0.5339806 0.4660194
   III 0.4722222 0.5277778
   IV  0.5061728 0.4938272
   V   0.3780488 0.6219512
```

Note that the *rows* (1st index) sum to 1. If a table of percentages is desired, just multiply the entire table by 100.

`prop.table` cannot be used to express the numbers relative to the grand total of the table, but you can of course always write

```
> tanner.sex/sum(tanner.sex)
      sex
tanner          M          F
   I   0.26478617 0.20382166
   II  0.05004550 0.04367607
   III 0.03093722 0.03457689
   IV  0.03730664 0.03639672
   V   0.11282985 0.18562329
```

The functions `margin.table` and `prop.table` also work on multiway tables — the `margin` argument can be a vector if the relevant margin has two or more dimensions.

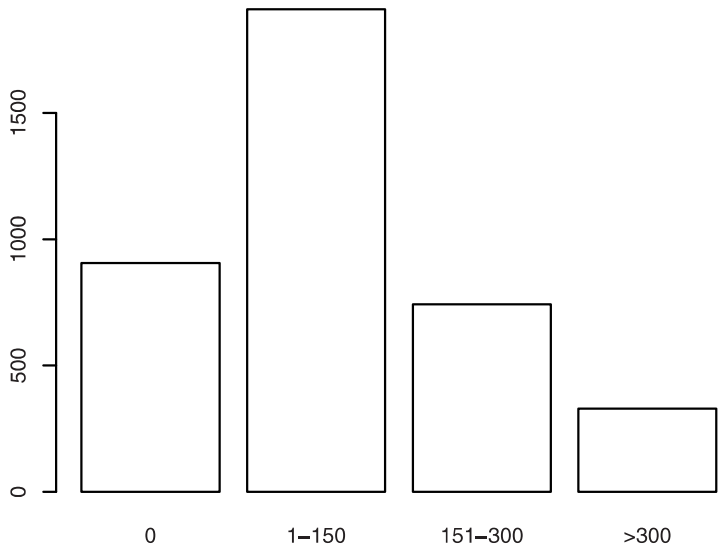

Figure 4.9. Simple `barplot` of total caffeine consumption.

4.6 Graphical display of tables

For presentation purposes, it may be desirable to display a graph rather than a table of counts or percentages. In this section, the main methods for doing this are described.

4.6.1 Barplots

Barplots are made using `barplot`. This function takes an argument, which can be a vector or a matrix. The simplest variant goes as follows (Figure 4.9):

```
> total.caff <- margin.table(caff.marital,2)
> total.caff
consumption
      0   1-150 151-300    >300
    906    1910     742     330
> barplot(total.caff, col="white")
```

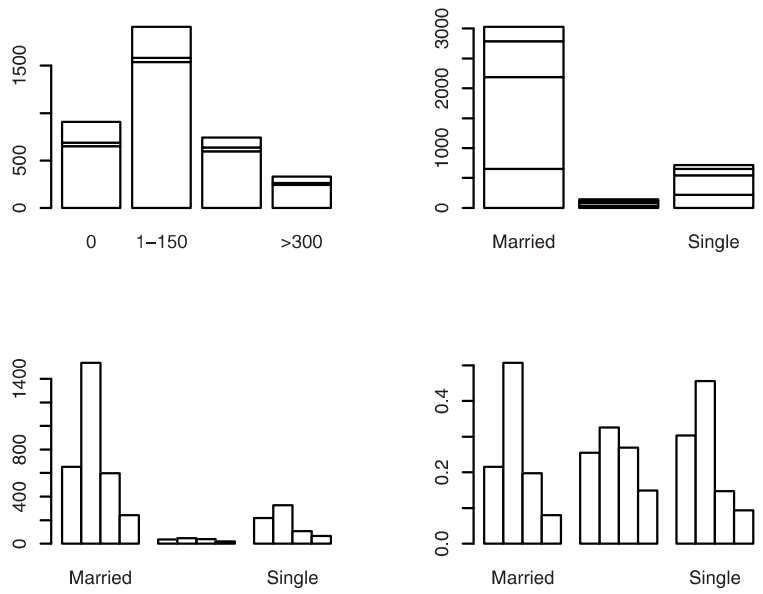

Figure 4.10. Four variants of `barplot` on a two-way table.

Without the `col="white"` argument, the plot comes out in colour, but this is not suitable for a black and white book illustration.

If the argument is a matrix, then `barplot` creates by default a "stacked barplot", where the columns are partitioned according to the contributions from different rows of the table. If you want to place the row contributions beside each other instead, you can use the argument `beside=T`. A series of variants is found in Figure 4.10, which is constructed as follows:

```
> par(mfrow=c(2,2))
> barplot(caff.marital, col="white")
> barplot(t(caff.marital), col="white")
> barplot(t(caff.marital), col="white", beside=T)
> barplot(prop.table(t(caff.marital),2), col="white", beside=T)
> par(mfrow=c(1,1))
```

In the last three plots, we switched rows and columns with the transposition function `t`. In the very last one, the columns are expressed as proportions of the total number in the group. Thus, information is lost on the relative sizes of the marital status groups, but the group of previously married women (recall that the data set deals with women giving birth)

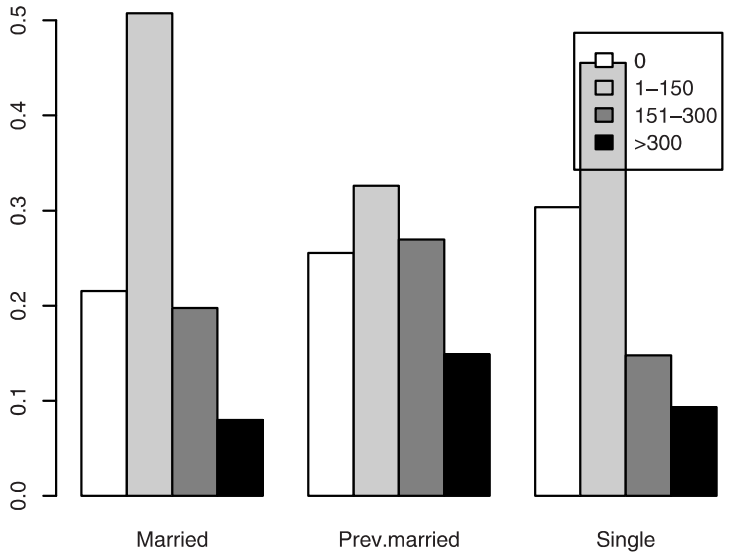

Figure 4.11. Bar plot with specified colours and legend.

is so small that it otherwise becomes almost impossible to compare their caffeine consumption profile with those of the other groups.

As usual, there are a multitude of ways to "prettify" the plots. Here is one possibility (Figure 4.11):

```
> barplot(prop.table(t(caff.marital),2),beside=T,
+ legend.text=colnames(caff.marital),
+ col=c("white","grey80","grey50","black"))
```

Notice that the legend overlaps the top of one of the columns. R is not designed to be able to find a clear area in which to place the legend. However, you can get full control of the legend's position if you insert it explicitly with the `legend` function. For that purpose, it will be helpful to use `locator()`, which allows you to click a mouse button over the plot and have the coordinates returned. See p. 209 for more about this.

4.6.2 *Dotcharts*

The Cleveland dotcharts, named after William S. Cleveland (1994), can be employed to study a table from both sides at the same time. They contain