2

Basics

R can be used at various levels. Of course, standard arithmetic is available, and hence it can be used as a (rather sophisticated) calculator. It is also provided with a graphical system that writes on a large variety of devices. Furthermore, R is a full-featured programming language that can be employed to tackle all typical tasks for which other programming languages are also used. It connects to other languages, programs, and data bases, and also to the operating system; users can control all these from within R.

In this chapter, we illustrate a few of the typical uses of R. Often solutions are not unique, but in the following we avoid sophisticated shortcuts. However, we encourage all readers to explore alternative solutions by reusing what they have learned in other contexts.

2.1 R as a Calculator

The standard arithmetic operators `+`, `-`, `*`, `/`, and `^` are available, where `x^y` yields x^y. Hence

```
R> 1 + 1

[1] 2

R> 2^3

[1] 8
```

In the output, `[1]` indicates the position of the first element of the vector returned by R. This is not surprising here, where all vectors are of length `1`, but it will be useful later.

The common mathematical functions, such as `log()`, `exp()`, `sin()`, `asin()`, `cos()`, `acos()`, `tan()`, `atan()`, `sign()`, `sqrt()`, `abs()`, `min()`, and `max()`, are also available. Specifically, `log(x, base = a)` returns the logarithm of `x` to base `a`, where `a` defaults to `exp(1)`. Thus

C. Kleiber, A. Zeileis, *Applied Econometrics with R*,
DOI: 10.1007/978-0-387-77318-6_2,

```
R> log(exp(sin(pi/4)^2) * exp(cos(pi/4)^2))

[1] 1
```

which also shows that `pi` is a built-in constant. There are further convenience functions, such as `log10()` and `log2()`, but here we shall mainly use `log()`. A full list of all options and related functions is available upon typing `?log`, `?sin`, etc. Additional functions useful in statistics and econometrics are `gamma()`, `beta()`, and their logarithms and derivatives. See `?gamma` for further information.

Vector arithmetic

In R, the basic unit is a vector, and hence all these functions operate directly on vectors. A vector is generated using the function `c()`, where `c` stands for "combine" or "concatenate". Thus

```
R> x <- c(1.8, 3.14, 4, 88.169, 13)
```

generates an object `x`, a vector, containing the entries `1.8`, `3.14`, `4`, `88.169`, `13`. The length of a vector is available using `length()`; thus

```
R> length(x)

[1] 5
```

Note that names are case-sensitive; hence `x` and `X` are distinct.

The preceding statement uses the assignment operator `<-`, which should be read as a single symbol (although it requires two keystrokes), an arrow pointing to the variable to which the value is assigned. Alternatively, `=` may be used at the user level, but since `<-` is preferred for programming, it is used throughout this book. There is no immediately visible result, but from now on `x` has as its value the vector defined above, and hence it can be used in subsequent computations:

```
R> 2 * x + 3

[1]   6.60   9.28  11.00 179.34  29.00

R> 5:1 * x + 1:5

[1]  10.00  14.56  15.00 180.34  18.00
```

This requires an explanation. In the first statement, the scalars (i.e., vectors of length 1) `2` and `3` are recycled to the length of `x` so that each element of `x` is multiplied by 2 before 3 is added. In the second statement, `x` is multiplied element-wise by the vector `1:5` (the sequence from 1 to 5; see below) and then the vector `5:1` is added element-wise.

Mathematical functions can be applied as well; thus

```
R> log(x)
```

```
[1] 0.5878 1.1442 1.3863 4.4793 2.5649
```

returns a vector containing the logarithms of the original entries of `x`.

Subsetting vectors

It is often necessary to access subsets of vectors. This requires the operator `[`, which can be used in several ways to extract elements of a vector. For example, one can either specify which elements to include or which elements to exclude: a vector of positive indices, such as

```
R> x[c(1, 4)]

[1]  1.80 88.17
```

specifies the elements to be extracted. Alternatively, a vector of negative indices, as in

```
R> x[-c(2, 3, 5)]

[1]  1.80 88.17
```

selects all elements but those indicated, yielding the same result. In fact, further methods are available for subsetting with `[`, which are explained below.

Patterned vectors

In statistics and econometrics, there are many instances where vectors with special patterns are needed. R provides a number of functions for creating such vectors, including

```
R> ones <- rep(1, 10)
R> even <- seq(from = 2, to = 20, by = 2)
R> trend <- 1981:2005
```

Here, `ones` is a vector of ones of length 10, `even` is a vector containing the even numbers from 2 to 20, and `trend` is a vector containing the integers from 1981 to 2005.

Since the basic element is a vector, it is also possible to concatenate vectors. Thus

```
R> c(ones, even)

 [1]  1  1  1  1  1  1  1  1  1  1  2  4  6  8 10 12 14 16 18 20
```

creates a vector of length 20 consisting of the previously defined vectors `ones` and `even` laid end to end.

2.2 Matrix Operations

A 2×3 matrix containing the elements `1:6`, by column, is generated via

```
R> A <- matrix(1:6, nrow = 2)
```

Alternatively, `ncol` could have been used, with `matrix(1:6, ncol = 3)` yielding the same result.

Basic matrix algebra

The transpose $A^\top$ of `A` is

```
R> t(A)

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

The dimensions of a matrix may be accessed using `dim()`, `nrow()`, and `ncol()`; hence

```
R> dim(A)

[1] 2 3

R> nrow(A)

[1] 2

R> ncol(A)

[1] 3
```

Single elements of a matrix, row or column vectors, or indeed entire submatrices may be extracted by specifying the rows and columns of the matrix from which they are selected. This uses a simple extension of the rules for subsetting vectors. (In fact, internally, matrices are vectors with an additional dimension attribute enabling row/column-type indexing.) Element a_{ij} of a matrix A is extracted using `A[i,j]`. Entire rows or columns can be extracted via `A[i,]` and `A[,j]`, respectively, which return the corresponding row or column *vectors*. This means that the dimension attribute is dropped (by default); hence subsetting will return a vector instead of a matrix if the resulting matrix has only one column or row. Occasionally, it is necessary to extract rows, columns, or even single elements of a matrix as a matrix. Dropping of the dimension attribute can be switched off using `A[i, j, drop = FALSE]`. As an example,

```
R> A1 <- A[1:2, c(1, 3)]
```

selects a square matrix containing the first and third elements from each row (note that A has only two rows in our example). Alternatively, and more compactly, `A1` could have been generated using `A[, -2]`. If no row number is specified, all rows will be taken; the `-2` specifies that all columns but the second are required.

`A1` is a square matrix, and if it is nonsingular it has an inverse. One way to check for singularity is to compute the determinant using the R function `det()`. Here, `det(A1)` equals -4; hence `A1` is nonsingular. Alternatively, its eigenvalues (and eigenvectors) are available using `eigen()`. Here, `eigen(A1)` yields the eigenvalues 7.531 and -0.531, again showing that `A1` is nonsingular.

The inverse of a matrix, if it cannot be avoided, is computed using `solve()`:

```
R> solve(A1)

     [,1]  [,2]
[1,] -1.5  1.25
[2,]  0.5 -0.25
```

We can check that this is indeed the inverse of `A1` by multiplying `A1` with its inverse. This requires the operator for matrix multiplication, `%*%`:

```
R> A1 %*% solve(A1)

     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Similarly, conformable matrices are added and subtracted using the arithmetical operators `+` and `-`. It is worth noting that for non-conformable matrices recycling proceeds along columns. Incidentally, the operator `*` also works for matrices; it returns the element-wise or Hadamard product of conformable matrices. Further types of matrix products that are often required in econometrics are the Kronecker product, available via `kronecker()`, and the cross product $A^\top B$ of two matrices, for which a computationally efficient algorithm is implemented in `crossprod()`.

In addition to the spectral decomposition computed by `eigen()` as mentioned above, R provides other frequently used matrix decompositions, including the singular-value decomposition `svd()`, the QR decomposition `qr()`, and the Cholesky decomposition `chol()`.

Patterned matrices

In econometrics, there are many instances where matrices with special patterns occur. R provides functions for generating matrices with such patterns. For example, a diagonal matrix with ones on the diagonal may be created using

```
R> diag(4)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
```

which yields the 4×4 identity matrix. Equivalently, it can be obtained by `diag(1, 4, 4)`, where the `1` is recycled to the required length 4. Of course, more general diagonal matrices are also easily obtained: `diag(rep(c(1,2), c(10, 10)))` yields a diagonal matrix of size 20×20 whose first 10 diagonal elements are equal to `1`, while the remaining ones are equal to `2`. (Readers with a basic knowledge of linear regression will note that an application could be a pattern of heteroskedasticity.)

Apart from setting up diagonal matrices, the function `diag()` can also be used for extracting the diagonal from an existing matrix; e.g., `diag(A1)`. Additionally, `upper.tri()` and `lower.tri()` can be used to query the positions of upper or lower triangular elements of a matrix, respectively.

Combining matrices

It is also possible to form new matrices from existing ones. This uses the functions `rbind()` and `cbind()`, which are similar to the function `c()` for concatenating vectors; as their names suggest, they combine matrices by rows or columns. For example, to add a column of ones to our matrix `A1`,

```
R> cbind(1, A1)
```

```
     [,1] [,2] [,3]
[1,]    1    1    5
[2,]    1    2    6
```

can be employed, while

```
R> rbind(A1, diag(4, 2))
```

```
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    4    0
[4,]    0    4
```

combines `A1` and `diag(4, 2)` by rows.

2.3 R as a Programming Language

R is a full-featured, interpreted, object-oriented programming language. Hence, it can be used for all the tasks other programming languages are also used

for, not only for data analysis. What makes it particularly useful for statistics and econometrics is that it was designed for "programming with data" (Chambers 1998). This has several implications for the data types employed and the object-orientation paradigm used (see Section 2.6 for more on object orientation).

An in-depth treatment of programming in S/R is given in Venables and Ripley (2000). If you read German, Ligges (2007) is an excellent introduction to programming with R. On a more advanced level, R Development Core Team (2008f,g) provides guidance about the language definition and how extensions to the R system can be written. The latter documents can be downloaded from CRAN and also ship with every distribution of R.

The mode of a vector

Probably the simplest data structure in R is a vector. All elements of a vector must be of the same type; technically, they must be of the same "mode". The mode of a vector `x` can be queried using `mode(x)`. Here, we need vectors of modes "`numeric`", "`logical`", and "`character`" (but there are others).

We have already seen that

```
R> x <- c(1.8, 3.14, 4, 88.169, 13)
```

creates a numerical vector, and one typical application of vectors is to store the values of some numerical variable in a data set.

Logical and character vectors

Logical vectors may contain the logical constants `TRUE` and `FALSE`. In a fresh session, the aliases `T` and `F` are also available for compatibility with S (which uses these as the logical constants). However, unlike `TRUE` and `FALSE`, the values of `T` and `F` can be changed (e.g., by using the former for signifying the sample size in a time series context or using the latter as the variable for an F statistic), and hence it is recommended not to rely on them but to always use `TRUE` and `FALSE`. Like numerical vectors, logical vectors can be created from scratch. They may also arise as the result of a logical comparison:

```
R> x > 3.5

[1] FALSE FALSE  TRUE  TRUE  TRUE
```

Further logical operations are explained below.

Character vectors can be employed to store strings. Especially in the early chapters of this book, we will mainly use them to assign labels or names to certain objects such as vectors and matrices. For example, we can assign names to the elements of `x` via

```
R> names(x) <- c("a", "b", "c", "d", "e")
R> x
```

```
    a     b     c     d     e
 1.80  3.14  4.00 88.17 13.00
```

Alternatively, we could have used `names(x) <- letters[1:5]` since `letters` and `LETTERS` are built-in vectors containing the 26 lower- and uppercase letters of the Latin alphabet, respectively. Although we do not make much use of them in this book, we note here that the character-manipulation facilities of R go far beyond these simple examples, allowing, among other things, computations on text documents or command strings.

More on subsetting

Having introduced vectors of modes numeric, character, and logical, it is useful to revisit subsetting of vectors. By now, we have seen how to extract parts of a vector using numerical indices, but in fact this is also possible using characters (if there is a `names` attribute) or logicals (in which case the elements corresponding to `TRUE` are selected). Hence, the following commands yield the same result:

```
R> x[3:5]

    c     d     e
 4.00 88.17 13.00

R> x[c("c", "d", "e")]

    c     d     e
 4.00 88.17 13.00

R> x[x > 3.5]

    c     d     e
 4.00 88.17 13.00
```

Subsetting of matrices (and also of data frames or multidimensional arrays) works similarly.

Lists

So far, we have only used plain vectors. We now proceed to introduce some related data structures that are similar but contain more information.

In R, lists are *generic vectors* where each element can be virtually any type of object; e.g., a vector (of arbitrary mode), a matrix, a full data frame, a function, or again a list. Note that the latter also allows us to create recursive data structures. Due to this flexibility, lists are the basis for most complex objects in R; e.g., for data frames or fitted regression models (to name two examples that will be described later).

As a simple example, we create, using the function `list()`, a list containing a sample from a standard normal distribution (generated with `rnorm()`; see

below) plus some markup in the form of a character string and a list containing the population parameters.

```
R> mylist <- list(sample = rnorm(5),
+    family = "normal distribution",
+    parameters = list(mean = 0, sd = 1))
R> mylist

$sample
[1]  0.3771 -0.9346  2.4302  1.3195  0.4503

$family
[1] "normal distribution"

$parameters
$parameters$mean
[1] 0

$parameters$sd
[1] 1
```

To select certain elements from a list, the extraction operators `$` or `[[` can be used. The latter is similar to `[`, the main difference being that it can only select a single element. Hence, the following statements are equivalent:

```
R> mylist[[1]]

[1]  0.3771 -0.9346  2.4302  1.3195  0.4503

R> mylist[["sample"]]

[1]  0.3771 -0.9346  2.4302  1.3195  0.4503

R> mylist$sample

[1]  0.3771 -0.9346  2.4302  1.3195  0.4503
```

The third element of `mylist` again being a list, the extractor functions can also be combined as in

```
R> mylist[[3]]$sd

[1] 1
```

Logical comparisons

R has a set of functions implementing standard logical comparisons as well as a few further functions that are convenient when working with logical values. The logical operators are `<`, `<=`, `>`, `>=`, `==` (for exact equality) and `!=` (for inequality). Also, if `expr1` and `expr2` are logical expressions, then `expr1 & expr2` is their intersection (logical "and"), `expr1 | expr2` is their union (logical "or"), and `!expr1` is the negation of `expr1`. Thus

```
R> x <- c(1.8, 3.14, 4, 88.169, 13)
R> x > 3 & x <= 4

[1] FALSE  TRUE  TRUE FALSE FALSE
```

To assess for which elements of a vector a certain expression is `TRUE`, the function `which()` can be used:

```
R> which(x > 3 & x <= 4)

[1] 2 3
```

The specialized functions `which.min()` and `which.max()` are available for computing the position of the minimum and the maximum. In addition to `&` and `|`, the functions `all()` and `any()` check whether all or at least some entries of a vector are `TRUE`:

```
R> all(x > 3)

[1] FALSE

R> any(x > 3)

[1] TRUE
```

Some caution is needed when assessing exact equality. When applied to numerical input, `==` does not allow for finite representation of fractions or for rounding error; hence situations like

```
R> (1.5 - 0.5) == 1

[1] TRUE

R> (1.9 - 0.9) == 1

[1] FALSE
```

can occur due to floating-point arithmetic (Goldberg 1991). For these purposes, `all.equal()` is preferred:

```
R> all.equal(1.9 - 0.9, 1)

[1] TRUE
```

Furthermore, the function `identical()` checks whether two R objects are exactly identical.

Due to coercion, it is also possible to compute directly on logical vectors using ordinary arithmetic. When coerced to numeric, `FALSE` becomes `0` and `TRUE` becomes `1`, as in

```
R> 7 + TRUE

[1] 8
```

Coercion

To convert an object from one type or class to a different one, R provides a number of coercion functions, conventionally named `as.`*foo*`()`, where *foo* is the desired type or class; e.g., `numeric`, `character`, `matrix`, and `data.frame` (a concept introduced in Section 2.5), among many others. They are typically accompanied by an `is.`*foo*`()` function that checks whether an object is of type or class *foo*. Thus

```
R> is.numeric(x)

[1] TRUE

R> is.character(x)

[1] FALSE

R> as.character(x)

[1] "1.8"    "3.14"   "4"      "88.169" "13"
```

In certain situations, coercion is also forced automatically by R; e.g., when the user tries to put elements of different modes into a single vector (which can only contain elements of the same mode). Thus

```
R> c(1, "a")

[1] "1" "a"
```

Random number generation

For programming environments in statistics and econometrics, it is vital to have good random number generators (RNGs) available, in particular to allow the users to carry out Monte Carlo studies. The R RNG supports several algorithms; see `?RNG` for further details. Here, we outline a few important commands.

The RNG relies on a "random seed", which is the basis for the generation of pseudo-random numbers. By setting the seed to a specific value using the function `set.seed()`, simulations can be made exactly reproducible. For example, using the function `rnorm()` for generating normal random numbers,

```
R> set.seed(123)
R> rnorm(2)

[1] -0.5605 -0.2302

R> rnorm(2)

[1] 1.55871 0.07051

R> set.seed(123)
R> rnorm(2)
```

```
[1] -0.5605 -0.2302
```

Another basic function for drawing random samples, with or without replacement from a finite set of values, is `sample()`. The default is to draw, without replacement, a vector of the same size as its input argument; i.e., to compute a permutation of the input as in

```
R> sample(1:5)

[1] 5 1 2 3 4

R> sample(c("male", "female"), size = 5, replace = TRUE,
+    prob = c(0.2, 0.8))

[1] "female" "male"   "female" "female" "female"
```

The second command draws a sample of size 5, with replacement, from the values `"male"` and `"female"`, which are drawn with probabilities 0.2 and 0.8, respectively.

Above, we have already used the function `rnorm()` for drawing from a normal distribution. It belongs to a broader family of functions that are all of the form `r`*dist*`()`, where *dist* can be, for example, `norm`, `unif`, `binom`, `pois`, `t`, `f`, `chisq`, corresponding to the obvious families of distributions. All of these functions take the sample size `n` as their first argument along with further arguments controlling parameters of the respective distribution. For example, `rnorm()` takes `mean` and `sd` as further arguments, with `0` and `1` being the corresponding defaults. However, these are not the only functions available for statistical distributions. Typically there also exist `d`*dist*`()`, `p`*dist*`()`, and `q`*dist*`()`, which implement the density, cumulative probability distribution function, and quantile function (inverse distribution function), respectively.

Flow control

Like most programming languages, R provides standard control structures such as `if`/`else` statements, `for` loops, and `while` loops. All of these have in common that an expression `expr` is evaluated, either conditional upon a certain condition `cond` (for `if` and `while`) or for a sequence of values (for `for`). The expression `expr` itself can be either a simple expression or a compound expression; i.e., typically a set of simple expressions enclosed in braces `{ ... }`. Below we present a few brief examples illustrating its use; see `?Control` for further information.

An `if`/`else` statement is of the form

```
if(cond) {
  expr1
} else {
  expr2
}
```

where `expr1` is evaluated if `cond` is `TRUE` and `expr2` otherwise. The `else` branch may be omitted if empty. A simple (if not very meaningful) example is

```
R> x <- c(1.8, 3.14, 4, 88.169, 13)
R> if(rnorm(1) > 0) sum(x) else mean(x)

[1] 22.02
```

where conditional on the value of a standard normal random number either the sum or the mean of the vector `x` is computed. Note that the condition `cond` can only be of length 1. However, there is also a function `ifelse()` offering a vectorized version; e.g.,

```
R> ifelse(x > 4, sqrt(x), x^2)

[1]  3.240  9.860 16.000  9.390  3.606
```

This computes the square root for those values in `x` that are greater than 4 and the square for the remaining ones.

A `for` loop looks similar, but the main argument to `for()` is of type `variable in sequence`. To illustrate its use, we recursively compute first differences in the vector `x`.

```
R> for(i in 2:5) {
+    x[i] <- x[i] - x[i-1]
+  }
R> x[-1]

[1]   1.34   2.66  85.51 -72.51
```

Finally, a `while` loop looks quite similar. The argument to `while()` is a condition that may change in every run of the loop so that it finally can become `FALSE`, as in

```
R> while(sum(x) < 100) {
+    x <- 2 * x
+  }
R> x

[1]   14.40   10.72   21.28  684.07 -580.07
```

Writing functions

One of the features of S and R is that users naturally become developers. Creating variables or objects and applying functions to them interactively (either to modify them or to create other objects of interest) is part of every R session. In doing so, typical sequences of commands often emerge that are carried out for different sets of input values. Instead of repeating the same steps "by hand", they can also be easily wrapped into a function. A simple example is

```
R> cmeans <- function(X) {
+    rval <- rep(0, ncol(X))
+    for(j in 1:ncol(X)) {
+      mysum <- 0
+      for(i in 1:nrow(X)) mysum <- mysum + X[i,j]
+      rval[j] <- mysum/nrow(X)
+    }
+    return(rval)
+  }
```

This creates a (deliberately awkward!) function `cmeans()`, which takes a matrix argument `X` and uses a double `for` loop to compute first the sum and then the mean in each column. The result is stored in a vector `rval` (our return value), which is returned after both loops are completed. This function can then be easily applied to new data, as in

```
R> X <- matrix(1:20, ncol = 2)
R> cmeans(X)

[1]  5.5 15.5
```

and (not surprisingly) yields the same result as the built-in function `colMeans()`:

```
R> colMeans(X)

[1]  5.5 15.5
```

The function `cmeans()` takes only a single argument `X` that has no default value. If the author of a function wants to set a default, this can be easily achieved by defining a `function` with a list of `name = expr` pairs, where `name` is the argument of the variable and `expr` is an expression with the default value. If the latter is omitted, no default value is set.

In interpreted matrix-based languages such as R, loops are typically less efficient than the corresponding vectorized computations offered by the system. Therefore, avoiding loops by replacing them with vectorized operations can save computation time, especially when the number of iterations in the loop can become large. To illustrate, let us generate $2 \cdot 10^6$ random numbers from the standard normal distribution and compare the built-in function `colMeans()` with our awkward function `cmeans()`. We employ the function `system.time()`, which is useful for profiling code:

```
R> X <- matrix(rnorm(2*10^6), ncol = 2)
R> system.time(colMeans(X))

   user  system elapsed
  0.004   0.000   0.005

R> system.time(cmeans(X))
```

```
   user  system elapsed
  5.572   0.004   5.617
```

Clearly, the performance of `cmeans()` is embarrassing, and using `colMeans()` is preferred.

Vectorized calculations

As noted above, loops can be avoided using vectorized arithmetic. In the case of `cmeans()`, our function calculating column-wise means of a matrix, it would be helpful to directly compute means column by column using the built-in function `mean()`. This is indeed the preferred solution. Using the tools available to us thus far, we could proceed as follows:

```
R> cmeans2 <- function(X) {
+    rval <- rep(0, ncol(X))
+    for(j in 1:ncol(X)) rval[j] <- mean(X[,j])
+    return(rval)
+  }
```

This eliminates one of the `for` loops and only cycles over the columns. The result is identical to the previous solutions, but the performance is clearly better than that of `cmeans()`:

```
R> system.time(cmeans2(X))

   user  system elapsed
  0.072   0.008   0.080
```

However, the code of `cmeans2()` still looks a bit cumbersome with the remaining `for` loop—it can be written much more compactly using the function `apply()`. This applies functions over the margins of an array and takes three arguments: the array, the index of the margin, and the function to be evaluated. In our case, the function call is

```
R> apply(X, 2, mean)
```

because we require means (using `mean()`) over the columns (i.e., the second dimension) of `X`. The performance of `apply()` can sometimes be better than a `for` loop; however, in many cases both approaches perform rather similarly:

```
R> system.time(apply(X, 2, mean))

   user  system elapsed
  0.084   0.028   0.114
```

To summarize, this means that (1) element-wise computations should be avoided if vectorized computations are available, (2) optimized solutions (if available) typically perform better than the generic `for` or `apply()` solution, and (3) loops can be written more compactly using the `apply()` function. In

fact, this is so common in R that several variants of `apply()` are available, namely `lapply()`, `tapply()`, and `sapply()`. The first returns a list, the second a table, and the third tries to simplify the result to a vector or matrix where possible. See the corresponding manual pages for more detailed information and examples.

Reserved words

Like most programming languages, R has a number of reserved words that provide the basic grammatical constructs of the language. Some of these have already been introduced above, and some more follow below. An almost complete list of reserved words in R is: `if`, `else`, `for`, `in`, `while`, `repeat`, `break`, `next`, `function`, `TRUE`, `FALSE`, `NA`, `NULL`, `Inf`, `NaN`, `...`). See `?Reserved` for a complete list. If it is attempted to use any of these as names, this results in an error.

2.4 Formulas

Formulas are constructs used in various statistical programs for specifying models. In R, formula objects can be used for storing symbolic descriptions of relationships among variables, such as the `~` operator in the formation of a formula:

```
R> f <- y ~ x
R> class(f)

[1] "formula"
```

So far, this is only a description without any concrete meaning. The result entirely depends on the function evaluating this formula. In R, the expression above commonly means "`y` is explained by `x`". Such formula interfaces are convenient for specifying, among other things, plots or regression relationships. For example, with

```
R> x <- seq(from = 0, to = 10, by = 0.5)
R> y <- 2 + 3 * x + rnorm(21)
```

the code

```
R> plot(y ~ x)
R> lm(y ~ x)

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
       2.00         3.01
```

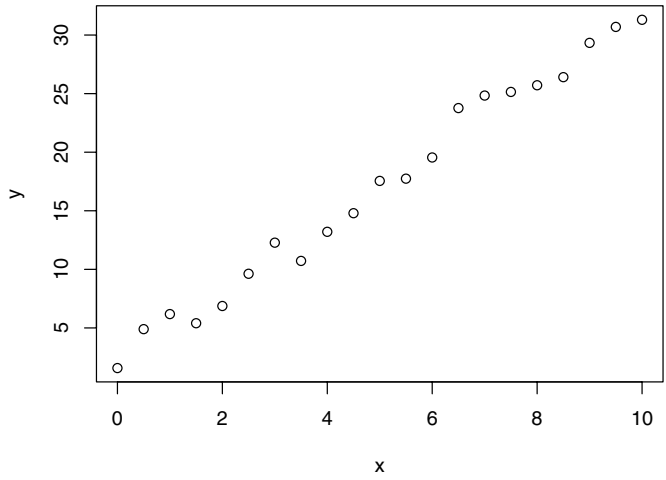

Fig. 2.1. Simple scatterplot of `y` vs. `x`.

first generates a scatterplot of `y` against `x` (see Figure 2.1) and then fits the corresponding simple linear regression model with slope 3.01 and intercept 2.00.

For specifying regression models, the formula language is much richer than outlined above and is based on a symbolic notation suggested by Wilkinson and Rogers (1973) in the statistical literature. For example, when using `lm()`, `log(y) ~ x1 + x2` specifies a linear regression of `log(y)` on two regressors `x1` and `x2` and an implicitly defined constant. More details on the formula specifications of linear regression models will be given in Chapter 3.

2.5 Data Management in R

In R, a data frame corresponds to what other statistical packages call a data matrix or a data set. Typically, it is an array consisting of a list of vectors and/or factors of identical length, thus yielding a rectangular format where columns correspond to variables and rows to observations.

Creation from scratch

Let us generate a simple artificial data set, with three variables named `"one"`, `"two"`, `"three"`, by using

```
R> mydata <- data.frame(one = 1:10, two = 11:20, three = 21:30)
```

Alternatively, `mydata` can be created using

```
R> mydata <- as.data.frame(matrix(1:30, ncol = 3))
R> names(mydata) <- c("one", "two", "three")
```

which first creates a matrix of size 10×3 that is subsequently coerced to a data frame and whose variable names are finally changed to `"one"`, `"two"`, `"three"`. Note that the same syntax can be used both for querying and modifying the names in a data frame. Furthermore, it is worth reiterating that although a data frame can be coerced from a matrix as above, it is internally represented as a list.

Subset selection

It is possible to access a subset of variables (i.e., columns) via `[` or `$`, where the latter can only extract a single variable. Hence, the second variable `two` can be selected via

```
R> mydata$two

 [1] 11 12 13 14 15 16 17 18 19 20

R> mydata[, "two"]

 [1] 11 12 13 14 15 16 17 18 19 20

R> mydata[, 2]

 [1] 11 12 13 14 15 16 17 18 19 20
```

In all cases, the object returned is a simple vector; i.e., the data frame attributes are dropped (by default).

To simplify access to variables in a certain data set, it can be `attach()`ed. Technically, this means that the attached data set is added to the `search()` path and thus variables contained in this data set can be found when their name is used in a command. Compare the following:

```
R> mean(two)

Error in mean(two) : Object "two" not found

R> attach(mydata)
R> mean(two)

[1] 15.5

R> detach(mydata)
```

Data frames should be attached with care; in particular, one should pay attention not to attach several data frames with the same column names or to have a variable with identical name in the global environment, as this is likely to generate confusion. To avoid attaching and detaching a data set for a single command only, the function `with()` can be handy, as in

```
R> with(mydata, mean(two))

[1] 15.5
```

It is often necessary to work with subsets of a data frame; i.e., to use only selected observations (= rows) and/or variables (= columns). This can again be done via `[` or, more conveniently, using the `subset()` command, whose main arguments are a data frame from which the subset is to be taken and a logical statement defining the elements to be selected. For example,

```
R> mydata.sub <- subset(mydata, two <= 16, select = -two)
```

takes all observations whose value of the second variable `two` does not exceed 16 (we know there are six observations with this property) and, in addition, all variables apart from `two` are selected.

Import and export

To export data frames in plain text format, the function `write.table()` can be employed:

```
R> write.table(mydata, file = "mydata.txt", col.names = TRUE)
```

It creates a text file `mydata.txt` in the current working directory. If this data set is to be used again, in another session, it may be imported using

```
R> newdata <- read.table("mydata.txt", header = TRUE)
```

The function `read.table()` returns a "`data.frame`" object, which is then assigned to a new object `newdata`. By setting `col.names = TRUE`, the column names are written in the first line of `mydata.txt` and hence we set `header = TRUE` when reading the file again. The function `write.table()` is quite flexible and allows specification of the separation symbol and the decimal separator, among other properties of the file to be written, so that various text-based formats, including tab- or comma-separated values, can be produced. Since the latter is a popular format for exchanging data (as it can be read and written by many spreadsheet programs, including Microsoft Excel), the convenience interfaces `read.csv()` and `write.csv()` are available. Similarly, `read.csv2()` and `write.csv2()` provide export and import of semicolon-separated values, a format that is typically used on systems employing the comma (and not the period) as the decimal separator. In addition, there exists a more elementary function, named `scan()`, for data not conforming to the matrix-like layout required by `read.table()`. We refer to the respective manual pages and the "R

Data Import/Export" manual (R Development Core Team 2008c) for further details.

It is also possible to save the data in the R internal binary format, by convention with extension `.RData` or `.rda`. The command

```
R> save(mydata, file = "mydata.rda")
```

saves the data in R binary format. Binary files may be loaded using

```
R> load("mydata.rda")
```

In contrast to `read.table()`, this does not return a single object; instead it makes all objects stored in `mydata.rda` directly available within the current environment. The advantage of using `.rda` files is that several R objects, in fact several arbitrary R objects, can be stored, including functions or fitted models, without loss of information.

All of the data sets in the package **AER** are supplied in this binary format (go to the folder `~/AER/data` in your R library to check). Since they are part of a package, they are made accessible more easily using `data()` (which in this case sets up the appropriate call for `load()`). Thus

```
R> data("Journals", package = "AER")
```

loads the `Journals` data frame from the **AER** package (stored in the file `~/AER/data/Journals.rda`), the data set used in Example 1 of our introductory R session. If the `package` argument is omitted, all packages currently in the search path are checked whether they provide a data set `Journals`.

Reading and writing foreign binary formats

R can also read and write a number of proprietary binary formats, notably S-PLUS, SPSS, SAS, Stata, Minitab, Systat, and dBase files, using the functions provided in the package **foreign** (part of a standard R installation). Most of the commands are designed to be similar to `read.table()` and `write.table()`. For example, for Stata files, both `read.dta()` and `write.dta()` are available and can be used to create a Stata file containing `mydata`

```
R> library("foreign")
R> write.dta(mydata, file = "mydata.dta")
```

and read it into R again via

```
R> mydata <- read.dta("mydata.dta")
```

See the documentation for the package **foreign** for further information.

Interaction with the file system and string manipulations

In the preceding paragraphs, some interaction with the file system was necessary to read and write data files. R possesses a rich functionality for interacting with external files and communicating with the operating system. This is far

beyond the scope of this book, but we would like to provide the interested reader with a few pointers that may serve as a basis for further reading.

Files available in a directory or folder can be queried via `dir()` and also copied (using `file.copy()`) or deleted (using `file.remove()`) independent of the operating system. For example, the Stata file created above can be deleted again from within R via

```
R> file.remove("mydata.dta")
```

Other (potentially system-dependent) commands can be sent as strings to the operating system using `system()`. See the respective manual pages for more information and worked-out examples.

Above, we discussed how data objects (especially data frames) can be written to files in various formats. Beyond that, one often wants to save commands or their output to text files. One possibility to achieve this is to use `sink()`, which can direct output to a `file()` connection to which the strings could be written with `cat()`. In some situations, `writeLines()` is more convenient for this. Furthermore, `dump()` can create text representations of R objects and write them to a `file()` connection.

Sometimes, one needs to manipulate the strings before creating output. R also provides rich and flexible functionality for this. Typical tasks include splitting strings (`strsplit()`) and/or pasting them together (`paste()`). For pattern matching and replacing, `grep()` and `gsub()` are available, which also support regular expressions. For combining text and variable values, `sprintf()` is helpful.

Factors

Factors are an extension of vectors designed for storing categorical information. Typical econometric examples of categorical variables include gender, union membership, or ethnicity. In many software packages, these are created using a numerical encoding (e.g., `0` for males and `1` for females); sometimes, especially in regression settings, a single categorical variable is stored in several such dummy variables if there are more than two categories.

In R, categorical variables should be specified as factors. As an example, we first create a dummy-coded vector with a certain pattern and subsequently transform it into a factor using `factor()`:

```
R> g <- rep(0:1, c(2, 4))
R> g <- factor(g, levels = 0:1, labels = c("male", "female"))
R> g

[1] male   male   female female female female
Levels: male female
```

The terminology is that a factor has a set of levels, say k levels. Internally, a k-level factor consists of two items: a vector of integers between `1` and `k` and a character vector, of length k, containing strings with the corresponding labels.

Above, we created the factor from an integer vector; alternatively, it could have been constructed from other numerical, character, or logical vectors. Ordinal information may also be stored in a factor by setting the argument `ordered = TRUE` when calling `factor()`.

The advantage of this approach is that R knows when a certain variable is categorical and can choose appropriate methods automatically. For example, the labels can be used in printed output, different summary and plotting methods can be chosen, and contrast codings (e.g., dummy variables) can be computed in linear regressions. Note that for these actions the ordering of the levels can be important.

Missing values

Many data sets contain observations for which certain variables are unavailable. Econometric software needs ways to deal with this. In R, such missing values are coded as `NA` (for "not available"). All standard computations on `NA` become `NA`.

Special care is needed when reading data that use a different encoding. For example, when preparing the package **AER**, we encountered several data sets that employed `-999` for missing values. If a file `mydata.txt` contains missing values coded in this way, they may be converted to `NA` using the argument `na.strings` when reading the file:

```
R> newdata <- read.table("mydata.txt", na.strings = "-999")
```

To query whether certain observations are `NA` or not, the function `is.na()` is provided.

2.6 Object Orientation

Somewhat vaguely, object-oriented programming (OOP) refers to a paradigm of programming where users/developers can create objects of a certain "class" (that are required to have a certain structure) and then apply "methods" for certain "generic functions" to these objects. A simple example in R is the function `summary()`, which is a generic function choosing, depending on the class of its argument, the summary method defined for this class. For example, for the numerical vector `x` and the factor `g` used above,

```
R> x <- c(1.8, 3.14, 4, 88.169, 13)
R> g <- factor(rep(c(0, 1), c(2, 4)), levels = c(0, 1),
+    labels = c("male", "female"))
```

the `summary()` call yields different types of results:

```
R> summary(x)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.80    3.14    4.00   22.00   13.00   88.20
```

```
R> summary(g)

  male female
     2      4
```

For the numerical vector `x`, a five-number summary (i.e., the minimum and maximum, the median, and the first and third quartiles) along with the mean are reported, and for the factor `g` a simple frequency table is returned. This shows that R has different `summary()` methods available for different types of classes (in particular, it knows that a five-number summary is not sensible for categorical variables). In R, every object has a class that can be queried using the function `class()`

```
R> class(x)

[1] "numeric"

R> class(g)

[1] "factor"
```

which is used internally for calling the appropriate method for a generic function.

In fact, R offers several paradigms of object orientation. The base installation comes with two different OOP systems, usually called S3 (Chambers and Hastie 1992) and S4 (Chambers 1998). The S3 system is much simpler, using a dispatch mechanism based on a naming convention for methods. The S4 system is more sophisticated and closer to other OOP concepts used in computer science, but it also requires more discipline and experience. For most tasks, S3 is sufficient, and therefore it is the only OOP system (briefly) discussed here.

In S3, a generic function is defined as a function with a certain list of arguments and then a `UseMethod()` call with the name of the generic function. For example, printing the function `summary()` reveals its definition:

```
R> summary

function (object, ...)
UseMethod("summary")
<environment: namespace:base>
```

It takes a first required argument `object` plus an arbitrary number of further arguments passed through `...` to its methods. What happens if this function is applied to an object, say of class "`foo`", is that R tries to apply the function `summary.foo()` if it exists. If not, it will call `summary.default()` if such a default method exists (which is the case for `summary()`). Furthermore, R objects can have a vector of classes (e.g., `c("foo", "bar")`, which means that such objects are of class "`foo`" inheriting from "`bar`"). In this case, R first tries to apply `summary.foo()`, then (if this does not exist) `summary.bar()`, and then (if both do not exist) `summary.default()`. All methods that are

currently defined for a generic function can be queried using `methods()`; e.g., `methods(summary)` will return a (long) list of methods for all sorts of different classes. Among them is a method `summary.factor()`, which is used when `summary(g)` is called. However, there is no `summary.numeric()`, and hence `summary(x)` is handled by `summary.default()`. As it is not recommended to call methods directly, some methods are marked as being non-visible to the user, and these cannot (easily) be called directly. However, even for visible methods, we stress that in most situations it is clearly preferred to use, for example, `summary(g)` instead of `summary.factor(g)`.

To illustrate how easy it is to define a class and some methods for it, let us consider a simple example. We create an object of class "`normsample`" that contains a sample from a normal distribution and then define a `summary()` method that reports the empirical mean and standard deviation for this sample. First, we write a simple class creator. In principle, it could have any name, but it is often called like the class itself:

```
R> normsample <- function(n, ...) {
+    rval <- rnorm(n, ...)
+    class(rval) <- "normsample"
+    return(rval)
+  }
```

This function takes a required argument `n` (the sample size) and further arguments `...`, which are passed on to `rnorm()`, the function for generating normal random numbers. In addition to the sample size, it takes further arguments—the mean and the standard deviation; see `?rnorm`. After generation of the vector of normal random numbers, it is assigned the class "`normsample`" and then returned.

```
R> set.seed(123)
R> x <- normsample(10, mean = 5)
R> class(x)

[1] "normsample"
```

To define a `summary()` method, we create a function `summary.normsample()` that conforms with the argument list of the generic (although `...` is not used here) and computes the sample size, the empirical mean, and the standard deviation.

```
R> summary.normsample <- function(object, ...) {
+    rval <- c(length(object), mean(object), sd(object))
+    names(rval) <- c("sample size","mean","standard deviation")
+    return(rval)
+  }
```

Hence, calling

```
R> summary(x)
```

```
sample size                mean standard deviation
    10.0000              5.0746              0.9538
```

automatically finds our new `summary()` method and yields the desired output.

Other generic functions with methods for most standard R classes are `print()`, `plot()`, and `str()`, which print, plot, and summarize the structure of an object, respectively.

2.7 R Graphics

It is no coincidence that early publications on S and R, such as Becker and Chambers (1984) and Ihaka and Gentleman (1996), are entitled "S: An Interactive Environment for Data Analysis and Graphics" and "R: A Language for Data Analysis and Graphics", respectively. R indeed has powerful graphics.

Here, we briefly introduce the "conventional" graphics as implemented in base R. R also comes with a new and even more flexible graphics engine, called **grid** (see Murrell 2005), that provides the basis for an R implementation of "trellis"-type graphics (Cleveland 1993) in the package **lattice** (Sarkar 2002), but these are beyond the scope of this book. An excellent overview of R graphics is given in Murrell (2005).

The function `plot()`

The basic function is the default `plot()` method. It is a generic function and has methods for many objects, including data frames, time series, and fitted linear models. Below, we describe the default `plot()` method, which can create various types of scatterplots, but many of the explanations extend to other methods as well as to other high-level plotting functions.

The scatterplot is probably the most common graphical display in statistics. A scatterplot of `y` vs. `x` is available using `plot(x, y)`. For illustration, we again use the `Journals` data from our package **AER**, taken from Stock and Watson (2007). As noted in Section 1.1, the data provide some information on subscriptions to economics journals at US libraries for the year 2000. The file contains 180 observations (the journals) on 10 variables, among them the number of library subscriptions (`subs`), the library subscription price (`price`), and the total number of citations for the journal (`citations`). These data will reappear in Chapter 3.

Here, we are interested in the relationship between the number of subscriptions and the price per citation. The following code chunk derives the required variable `citeprice` and plots the number of library subscriptions against it in logarithms:

```
R> data("Journals")
R> Journals$citeprice <- Journals$price/Journals$citations
R> attach(Journals)
```

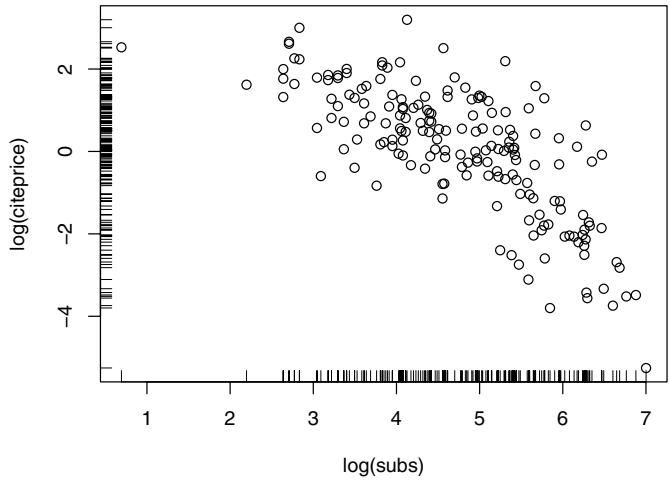

Fig. 2.2. Scatterplot of the journals data with ticks added.

```
R> plot(log(subs), log(citeprice))
R> rug(log(subs))
R> rug(log(citeprice), side = 2)
R> detach(Journals)
```

The function `rug()` adds ticks, thus visualizing the marginal distributions of the variables, along one or both axes of an existing plot. Figure 2.2 has ticks on both of the horizontal and vertical axes. An alternative way of specifying `plot(x, y)` is to use the formula method of `plot()`; i.e., `plot(y ~ x)`. This leads to the same scatterplot but has the advantage that a `data` argument can be specified. Hence we can avoid attaching and detaching the data frame:

```
R> plot(log(subs) ~ log(citeprice), data = Journals)
```

Graphical parameters

All this looks deceptively simple, but the result can be modified in numerous ways. For example, `plot()` takes a `type` argument that controls whether points (`type = "p"`, the default), lines (`type = "l"`), both (`type = "b"`), stair steps (`type = "s"`), or further types of plots are generated. The annotation can be modified by changing the `main` title or the `xlab` and `ylab` axis labels. See `?plot` for more details.

Additionally, there are several dozen graphical parameters (see `?par` for the full list) that can be modified either by setting them with `par()` or by

Table 2.1. A selective list of arguments to `par()`.

Argument	Description
`axes`	should axes be drawn?
`bg`	background color
`cex`	size of a point or symbol
`col`	color
`las`	orientation of axis labels
`lty, lwd`	line type and line width
`main, sub`	title and subtitle
`mar`	size of margins
`mfcol, mfrow`	array defining layout for several graphs on a plot
`pch`	plotting symbol
`type`	types (see text)
`xlab, ylab`	axis labels
`xlim, ylim`	axis ranges
`xlog, ylog, log`	logarithmic scales

supplying them to the `plot()` function. We cannot explain all of these here, but we will highlight a few important parameters: `col` sets the color(s) and `xlim` and `ylim` adjust the plotting ranges. If points are plotted, `pch` can modify the plotting character and `cex` its character extension. If lines are plotted, `lty` and `lwd` specify the line type and width, respectively. The size of labels, axis ticks, etc., can be changed by further `cex`-type arguments such as `cex.lab` and `cex.axis`. A brief list of arguments to `par()` is provided in Table 2.1. This is just the tip of the iceberg, and further graphical parameters will be introduced as we proceed.

As a simple example, readers may want to try

```
R> plot(log(subs) ~ log(citeprice), data = Journals, pch = 20,
+    col = "blue", ylim = c(0, 8), xlim = c(-7, 4),
+    main = "Library subscriptions")
```

This yields solid circles (`pch = 20`) instead of the default open ones, drawn in blue, and there are wider ranges in the `x` and `y` directions; there is also a main title.

It is also possible to add further layers to a plot. Thus, `lines()`, `points()`, `text()`, and `legend()` add what their names suggest to an existing plot. For example, `text(-3.798, 5.846, "Econometrica", pos = 2)` puts a character string at the indicated location (i.e., to the left of the point). In regression analyses, one often wants to add a regression line to a scatterplot. As seen in Chapter 1, this is achieved using `abline(a, b)`, where `a` is the intercept and `b` is the slope.

At this point, there does not seem to be a great need for all this; however, most users require fine control of visual displays at some point, especially when publication-quality plots are needed. We refrain from presenting artificial

examples toying with graphics options; instead we shall introduce variations of the standard displays as we proceed.

Of course, there are many further plotting functions besides the default `plot()` method. For example, standard statistical displays such as barplots, pie charts, boxplots, QQ plots, or histograms are available in the functions `barplot()`, `pie()`, `boxplot()`, `qqplot()`, and `hist()`. It is instructive to run `demo("graphics")` to obtain an overview of R's impressive graphics facilities.

Exporting graphics

In interactive use, graphics are typically written to a graphics window so that they can be inspected directly. However, after completing an analysis, we typically want to save the resulting graphics (e.g., for publication in a report, journal article, or thesis). For users of Microsoft Windows and Microsoft Word, a simple option is to "copy and paste" them into the Microsoft Word document. For other programs, such as LaTeX, it is preferable to export the graphic into an external file. For this, there exist various graphics devices to which plots can be written. Devices that are available on all platforms include the vector formats PostScript and PDF; other devices, such as the bitmap formats PNG and JPEG and the vector format WMF, are only available if supported by the system (see `?Devices` for further details). They all work in the same way: first the device is opened—e.g., the PDF device is opened by the function `pdf()`—then the commands creating the plot are executed, and finally the device is closed by `dev.off()`. A simple example creating a plot on a PDF device is:

```
R> pdf("myfile.pdf", height = 5, width = 6)
R> plot(1:20, pch = 1:20, col = 1:20, cex = 2)
R> dev.off()
```

This creates the PDF file `myfile.pdf` in the current working directory, which contains the graphic generated by the `plot()` call (see Figure 2.3). Incidentally, the plot illustrates a few of the parameters discussed above: it shows the first 20 plotting symbols (all shown in double size) and that in R a set of colors is also numbered. The first eight colors are black, red, green, blue, turquoise, violet, yellow, and gray. From color nine on, this vector is simply recycled.

Alternatively to opening, printing and closing a device, it is also possible to print an existing plot in the graphics window to a device using `dev.copy()` and `dev.print()`; see the corresponding manual page for more information.

Mathematical annotation of plots

A feature that particularly adds to R's strengths when it comes to publication-quality graphics is its ability to add mathematical annotation to plots (Murrell and Ihaka 2000). An S expression containing a mathematical expression can

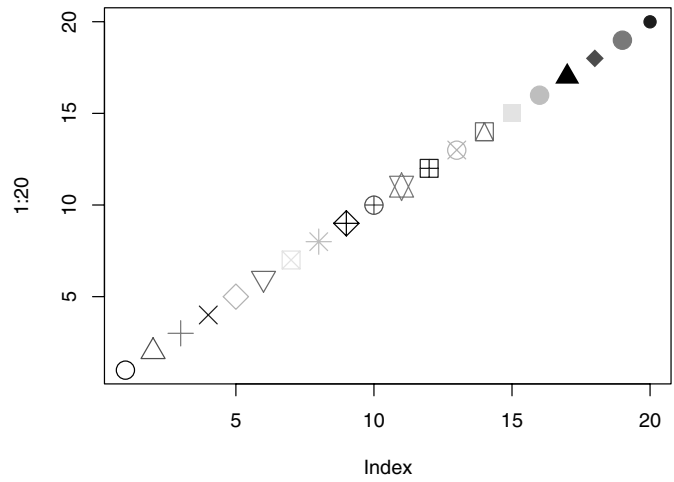

Fig. 2.3. Scatterplot written on a PDF device.

be passed to plotting functions without being evaluated; instead it is processed for annotation of the graph created. Readers familiar with LaTeX will have no difficulties in adapting to the syntax; for details, see `?plotmath` and `demo("plotmath")`. As an example, Figure 2.4 provides a plot of the density of the standard normal distribution (provided by `dnorm()` in R), including its mathematical definition

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

It is obtained via

```
R> curve(dnorm, from = -5, to = 5, col = "slategray", lwd = 3,
+    main = "Density of the standard normal distribution")
R> text(-5, 0.3, expression(f(x) == frac(1, sigma ~~
+    sqrt(2*pi)) ~~ e^{-frac((x - mu)^2, 2*sigma^2)}), adj = 0)
```

The function `curve()` plots the density function `dnorm()`, and then `text()` is used to add the `expression()` containing the formula to the plot. This example concludes our brief introduction to R graphics.

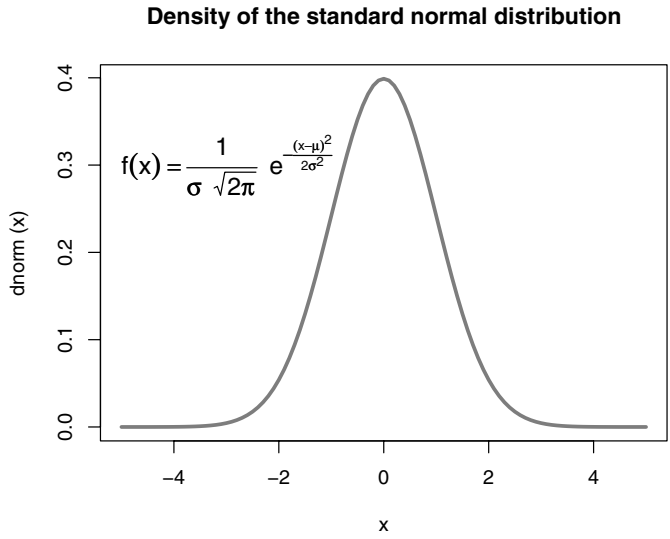

Fig. 2.4. Plot of the density of the standard normal distribution, including its mathematical expression.

2.8 Exploratory Data Analysis with R

In this section, we shall briefly illustrate some standard exploratory data analysis techniques. Readers seeking a more detailed introduction to basic statistics using R are referred to Dalgaard (2002).

We reconsider the `CPS1985` data taken from Berndt (1991), which were encountered in our introductory R session when illustrating several regression methods. After making the data available via `data()`, some basic information can be queried by `str()`:

```
R> data("CPS1985")
R> str(CPS1985)

'data.frame':        533 obs. of  11 variables:
 $ wage      : num   4.95  6.67  4.00  7.50 13.07 ...
 $ education : int  9 12 12 12 13 10 12 16 12 12 ...
 $ experience: int  42 1 4 17 9 27 9 11 9 17 ...
 $ age       : int  57 19 22 35 28 43 27 33 27 35 ...
 $ ethnicity : Factor w/ 3 levels "cauc","hispanic",..: 1 1 1..
 $ region    : Factor w/ 2 levels "south","other": 2 2 2 2 2 ..
 $ gender    : Factor w/ 2 levels "male","female": 2 1 1 1 1 ..
 $ occupation: Factor w/ 6 levels "worker","technical",..: 1 ..
 $ sector    : Factor w/ 3 levels "manufacturing",..: 1 1 3 3..
```

```
 $ union     : Factor w/ 2 levels "no","yes": 1 1 1 1 2 1 1 1..
 $ married   : Factor w/ 2 levels "no","yes": 2 1 1 2 1 1 1 2..
```

This reveals that this "`data.frame`" object comprises 533 observations on 11 variables, including the numerical variable `wage`, the integer variables `education`, `experience`, and `age`, and seven factors, each comprising two to six levels.

Instead of using the list-type view that `str()` provides, it is often useful to inspect the top (or the bottom) of a data frame in its rectangular representation. For this purpose, there exist the convenience functions `head()` and `tail()`, returning (by default) the first and last six rows, respectively. Thus

```
R> head(CPS1985)
```

```
   wage education experience age ethnicity region gender
1  4.95         9         42  57      cauc  other female
2  6.67        12          1  19      cauc  other   male
3  4.00        12          4  22      cauc  other   male
4  7.50        12         17  35      cauc  other   male
5 13.07        13          9  28      cauc  other   male
6  4.45        10         27  43      cauc  south   male
  occupation        sector union married
1     worker manufacturing    no     yes
2     worker manufacturing    no      no
3     worker         other    no      no
4     worker         other    no     yes
5     worker         other   yes      no
6     worker         other    no      no
```

Another useful way of gaining a quick overview of a data set is to use the `summary()` method for data frames, which provides a summary for each of the variables. As the type of the summary depends on the class of the respective variable, we inspect the `summary()` methods separately for various variables from `CPS1985` below. Hence, the output of `summary(CPS1985)` is omitted here.

As the `CPS1985` data are employed repeatedly in the following, we avoid lengthy commands such as `CPS1985$education` by attaching the data set. Also, to compactify subsequent output, we abbreviate two levels of occupation from "`technical`" to "`techn`" and from "`management`" to "`mgmt`".

```
R> levels(CPS1985$occupation)[c(2, 6)] <- c("techn", "mgmt")
R> attach(CPS1985)
```

Now variables are accessible by their names.

We proceed by illustrating exploratory analysis of single as well as pairs of variables, distinguishing among numerical variables, factors, and combinations thereof. We begin with the simplest kind, a single numerical variable.

One numerical variable

We will first look at the distribution of wages in the sample:

```
R> summary(wage)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    5.25    7.78    9.03   11.20   44.50
```

This provides Tukey's five-number summary plus the mean wage. The mean and median could also have been obtained using

```
R> mean(wage)

[1] 9.031

R> median(wage)

[1] 7.78
```

and `fivenum()` computes the five-number summary. Similarly, `min()` and `max()` would have yielded the minimum and the maximum. Arbitrary quantiles can be computed by `quantile()`.

For measures of spread, there exist the functions

```
R> var(wage)

[1] 26.43

R> sd(wage)

[1] 5.141
```

returning the variance and the standard deviation, respectively.

Graphical summaries are also helpful. For numerical variables such as `wage`, density visualizations (via histograms or kernel smoothing) and boxplots are suitable. Boxplots will be considered below in connection with two-variable displays. Figure 2.5, obtained via

```
R> hist(wage, freq = FALSE)
R> hist(log(wage), freq = FALSE)
R> lines(density(log(wage)), col = 4)
```

shows the densities of `wage` and its logarithm (that is, areas under curves equal 1, resulting from `freq = FALSE`; otherwise absolute frequencies would have been depicted). Further arguments allow for fine tuning of the selection of the breaks in the histogram. Added to the histogram in the right panel is a kernel density estimate obtained using `density()`. Clearly, the distribution of the logarithms is less skewed than that of the raw data. Note that `density()` only computes the density coordinates and does not provide a plot; hence the estimate is added via `lines()`.

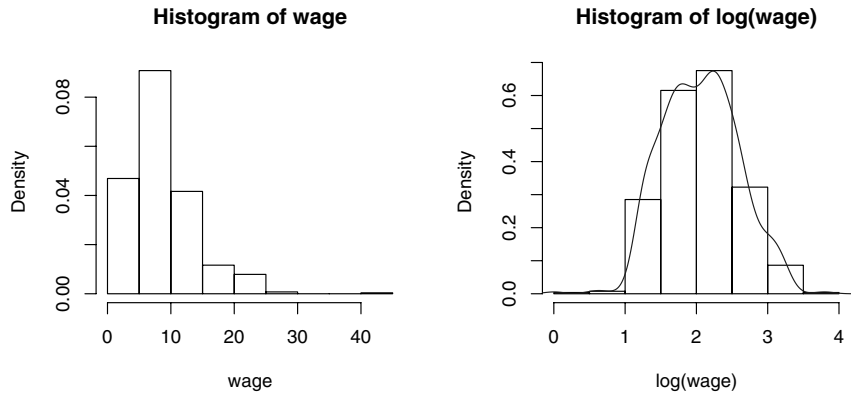

Fig. 2.5. Histograms of wages (left panel) and their logarithms with superimposed density (right panel).

One categorical variable

For categorical data, it makes no sense to compute means and variances; instead one needs a table indicating the frequencies with which the categories occur. If R is told that a certain variable is categorical (by making it a "`factor`"), it automatically chooses an appropriate summary:

```
R> summary(occupation)
```

```
  worker    techn services   office    sales     mgmt
     155      105       83       97       38       55
```

This could also have been computed by `table(occupation)`. If relative instead of absolute frequencies are desired, there exists the function `prop.table()`:

```
R> tab <- table(occupation)
R> prop.table(tab)
```

```
occupation
  worker    techn services   office    sales     mgmt
  0.2908   0.1970   0.1557   0.1820   0.0713   0.1032
```

Categorical variables are typically best visualized by barplots. If majorities are to be brought out, pie charts might also be useful. Thus

```
R> barplot(tab)
R> pie(tab)
```

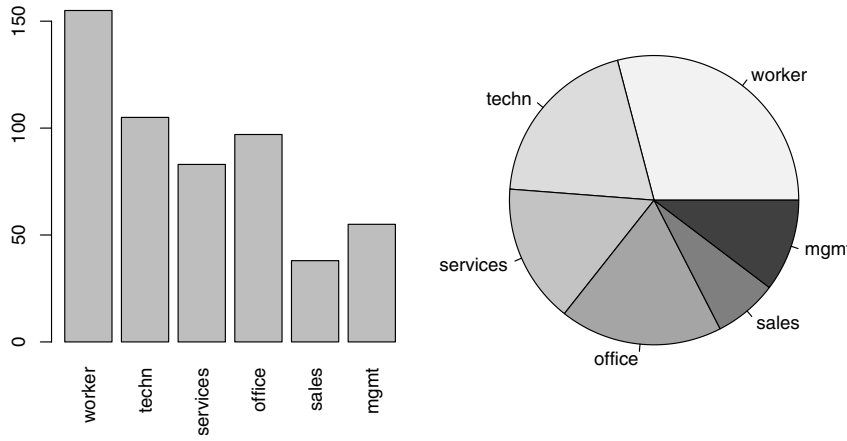

Fig. 2.6. Bar plot and pie chart of `occupation`.

provides Figure 2.6. Note that both functions expect the tabulated frequencies as input. In addition, calling `plot(occupation)` is equivalent to `barplot(table(occupation))`.

Two categorical variables

The relationship between two categorical variables is typically summarized by a contingency table. This can be created either by `xtabs()`, a function with a formula interface, or by `table()`, a function taking an arbitrary number of variables for cross-tabulation (and not only a single one as shown above).

We consider the factors `occupation` and `gender` for illustration:

```
R> xtabs(~ gender + occupation, data = CPS1985)

        occupation
gender   worker techn services office sales mgmt
  male      126    53       34     21    21   34
  female     29    52       49     76    17   21
```

which can equivalently be created by `table(gender, occupation)`. A simple visualization is a mosaic plot (Hartigan and Kleiner 1981; Friendly 1994), which can be seen as a generalization of stacked barplots. The plot given in Figure 2.7 (also known as a "spine plot", a certain variant of the standard mosaic display), obtained via

```
R> plot(gender ~ occupation, data = CPS1985)
```

shows that the proportion of males and females changes considerably over the levels of `occupation`. In addition to the shading that brings out the

Fig. 2.7. Mosaic plot (spine plot) of gender versus occupation.

conditional distribution of `gender` given `occupation`, the widths of the bars visualize the marginal distribution of `occupation`, indicating that there are comparatively many workers and few salespersons.

Two numerical variables

We exemplify the exploratory analysis of the relationship between two numerical variables by using `wage` and `education`.

A summary measure for two numerical variables is the correlation coefficient, implemented in the function `cor()`. However, the standard (Pearson) correlation coefficient is not necessarily meaningful for positive and heavily skewed variables such as `wage`. We therefore also compute a nonparametric variant, Spearman's ϱ, which is available in `cor()` as an option.

```
R> cor(log(wage), education)

[1] 0.379

R> cor(log(wage), education, method = "spearman")

[1] 0.3798
```

Both measures are virtually identical and indicate only a modest amount of correlation here, see also the corresponding scatterplot in Figure 2.8:

```
R> plot(log(wage) ~ education)
```

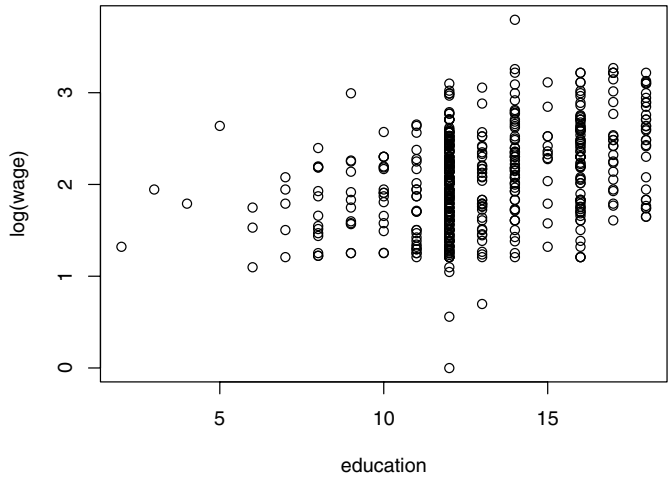

Fig. 2.8. Scatterplot of wages (in logs) versus education.

One numerical and one categorical variable

It is common to have both numerical and categorical variables in a data frame. For example, here we have `wage` and `gender`, and there might be some interest in the distribution of `wage` by `gender`. A suitable function for numerical summaries is `tapply()`. It applies, for a numerical variable as its first argument and a (list of) categorical variable(s) as its second argument, the function specified as the third argument. Hence, mean wages conditional on gender are available using

```
R> tapply(log(wage), gender, mean)
```

```
  male female
 2.165  1.935
```

Using similar commands, further descriptive measures or even entire summaries (just replace `mean` by `summary`) may be computed.

Suitable graphical displays are parallel boxplots and quantile-quantile (QQ) plots, as depicted in Figure 2.9. Recall that a boxplot (or "box-and-whiskers plot") is a coarse graphical summary of an empirical distribution. The box indicates "hinges" (approximately the lower and upper quartiles) and the median. The "whiskers" (lines) indicate the largest and smallest observations falling within a distance of 1.5 times the box size from the nearest hinge. Any observations falling outside this range are shown separately and would

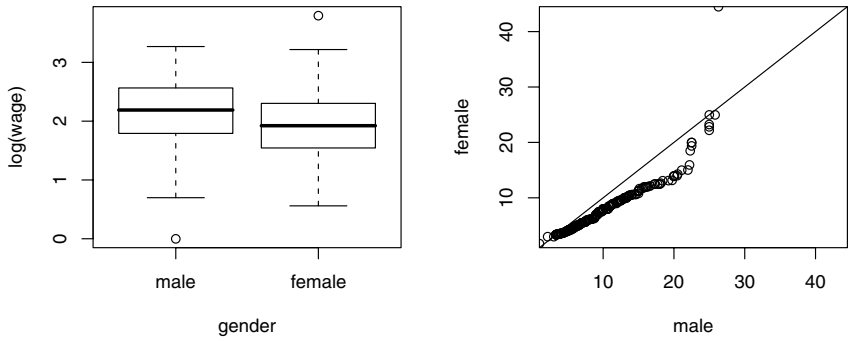

Fig. 2.9. Boxplot and QQ plot of wages stratified by gender.

be considered extreme or outlying (in an approximately normal sample). Note that there are several variants of boxplots in the literature.

The commands `plot(y ~ x)` and `boxplot(y ~ x)` both yield the same parallel boxplot if `x` is a "`factor`"; thus

```
R> plot(log(wage) ~ gender)
```

gives the left panel of Figure 2.9. It shows that the overall shapes of both distributions are quite similar and that males enjoy a substantial advantage, especially in the medium range. The latter feature is also brought out by the QQ plot (right panel) resulting from

```
R> mwage <- subset(CPS1985, gender == "male")$wage
R> fwage <- subset(CPS1985, gender == "female")$wage
R> qqplot(mwage, fwage, xlim = range(wage), ylim = range(wage),
+    xaxs = "i", yaxs = "i", xlab = "male", ylab = "female")
R> abline(0, 1)
```

where almost all points are below the diagonal (corresponding to identical distributions in both samples). This indicates that, for most quantiles, male wages are typically higher than female wages.

We end this section by detaching the data:

```
R> detach(CPS1985)
```

2.9 Exercises

1. Create a square matrix, say `A`, with entries $a_{ii} = 2$, $i = 2, \ldots, n-1$, $a_{11} = a_{nn} = 1$, $a_{i,i+1} = a_{i,i-1} = -1$, and $a_{ij} = 0$ elsewhere. (Where does this matrix occur in econometrics?)
2. "PARADE" is the Sunday newspaper magazine supplementing the Sunday or weekend edition of some 500 daily newspapers in the United States of America. An important yearly feature is an article providing information on some 120–150 "randomly" selected US citizens, indicating their profession, hometown and state, and their yearly earnings. The `Parade2005` data contain the 2005 version, amended by a variable indicating celebrity status (motivated by substantial oversampling of celebrities in these data). For the `Parade2005` data:
   (a) Determine the mean earnings in California. Explain the result.
   (b) Determine the number of individuals residing in Idaho. (What does this say about the data set?)
   (c) Determine the mean and the median earnings of celebrities. Comment.
   (d) Obtain boxplots of `log(earnings)` stratified by `celebrity`. Comment.
3. For the `Parade2005` data of the preceding exercise, obtain a kernel density estimate of the earnings for the full data set. It will be necessary to transform the data to logarithms (why?). Comment on the result. Be sure to try out some arguments to `density()`, in particular the plug-in bandwidth `bw`.
4. Consider the `CPS1988` data, taken from Bierens and Ginther (2001). (These data will be used for estimating an earnings equation in the next chapter.)
   (a) Obtain scatterplots of the logarithm of the real wage (`wage`) versus experience and versus education.
   (b) In fact, `education` corresponds to years of schooling and therefore takes on only a limited number of values. Transform `education` into a factor and obtain parallel boxplots of `wage` stratified by the levels of `education`. Repeat for `experience`.
   (c) The data set contains four additional factors, `ethnicity`, `smsa`, `region`, and `parttime`. Obtain suitable graphical displays of log(`wage`) versus each of these factors.