Appendix C

Basic Objects and Types in R

C.1 Objects in R

The basic constituent of the **R** language is an *object.* Each may have one or several *modes.* **R** defines objects that have a single mode as *atomic objects.* Objects are *simple* or *recursive.* Simple objects contain components having the same mode, while recursive objects contain components of several modes.

The main modes of an **R** object are:

Mode	Example
null	NULL
logical	TRUE or T, FALSE or F
numeric	1, 2.719, pi
character	'Hello', "Dolly"
complex	2 + 3i

The functions `mode` and `typeof` allow the display of the mode and the type of an object, while the function `attributes` returns the attributes of an object.

To test if an object belongs to a certain type one could use the following functions:

is.character
is.complex
is.integer
is.logical
is.numeric

Example C.15. After creating the complex number z, we test its type as follows:

```
> z <- 2+3i
> is.integer(z)
[1] FALSE
> is.complex(z)
[1] TRUE
```

The functions `mode` and `typeof` can be used in a similar manner:

```
mode(z)
[1] "complex"
> typeof(z)
[1] "complex"
```

C.2 Vectors and Factors in R

Vectors are atomic objects that accommodate sequences of elements that have the same mode. A *scalar* in **R** is a vector of length 1.

A scalar is created using an assignment as we show next.

```
x <- 6.9
```

Its length obtained by `length(x)` is 1.

A vector of length n can be defined using the concatenation function `c` (whose name is derived from the word "combine"). To create the vector $(10, 20, 25)$ we write

```
y <- c(10,20,25)
```

The length of `y` is 3.

Individual components of a vector `v` can be accessed using the notation $v[i]$. For example,

```
> y[2]
```

returns 20.

A vector of length 0 can be created by

```
> v <- vector("numeric")
> length(v)
[1] 0
```

Vector components can be named. Consider the vector `perfsq`. To assign names to its components we write:

```
perfsq <- c(1,4,9,16,25,36,49,64,81)
> names(perfsq) <- c("one","two","three","four","five","six","seven",
+ "eight","nine")
```

When the value of `perfsq` is inspected we get both the components of the vector and their names:

```
> perfsq
  one   two three  four  five   six seven eight  nine
    1     4     9    16    25    36    49    64    81
```

A portion of a vector defined by subsets of its index values can be extracted by

```
> perfsq[2:5]
```

returning

```
[1] 4 9 16 25
```

Random samples can be constructed using the function `sample`.

Example C.16. Let `x` be a vector of 10 integer created by

```
x <- 1:10
```

To sample five of its components we write

```
y <- sample(x,5)
```

which may return

```
> y
[1]  6  9 10  7  8
```

If the second argument is omitted, we obtain a random permutation of `x` as in

```
> z <- sample(x)
> z
 [1]  8  4 10  6  5  9  3  7  1  2
```

Samples with replacement can be obtained by using the parameter `replace = TRUE`. For example, we have

```
 w <- sample(x,replace=TRUE)
> w
 [1]  8 10 10 10  3  9  1  8  5  7
```

Finally, to produce a quasi-random sequence of 20 binary digits we write

```
> sample(c(0,1),20,replace=TRUE)
 [1] 0 1 0 1 0 1 0 0 0 1 0 1 1 0 0 0 1 1 1 0
```

Let us consider other useful examples of manipulating vectors.

The combination operation c is associative.

Example C.17. If we define the vector `ps` as

```
ps <- c(1,4,9,16,25,36,49,64,81)
```

and then write

```
ps <- c(ps,c(8,27,64,125))
```

the result is equivalent to writing

```
ps <- c(1,4,9,16,25,36,49,64,81,8,27,64,125)
```

A vector can be involved in a condition. This condition will be tested for each of its component and a vector of Boolean values will be generated, as it is shown below:

```
> v<- c(1,5,2,4,9,6)
> v <= 6
[1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
```

The condition can be used to extract the components of the vector that satisfy a condition as in

```
> v[v<6]
[1] 1 5 2 4
```

Example C.18. Let us index the components of ps with numbers between 1 and 13:

```
> names(ps) <- 1:13
> ps
  1   2   3   4   5   6   7   8   9  10  11  12  13
  1   4   9  16  25  36  49  64  81   8  27  64 125
```

To extract the components of ps corresponding to the fourth to the seventh component we can write

```
> idx <- 4:7
> ps[idx]
 4  5  6  7
16 25 36 49
```

To eliminate all components of ps between the 4th and the 7th we can use negative indexes:

```
> ps[-idx]
  1   2   3   8   9  10  11  12  13
  1   4   9  64  81   8  27  64 125
```

Similarly, to obtain components of ps outside the range from 2 to 7 of the index we can use:

```
> ps[-2:-7]
  1   8   9  10  11  12  13
  1  64  81   8  27  64 125
```

A vector can be sorted using the function `sort`. When applied to `ps` we obtain

```
> pt <- sort(ps)
> pt
  1   2  10   3   4   5  11   6   7   8  12   9  13
  1   4   8   9  16  25  27  36  49  64  64  81 125
```

This allows us to identify the indexes of `ps` that correspond to the components that are less than 40 by writing

```
pt[pt < 40]
```

which yields:

```
 pt[pt < 40]
 1  2 10  3  4  5 11  6
 1  4  8  9 16 25 27 36
```

Finally, to extract the indices of the original vector that correspond to components less than 40 we write:

```
ll <- as.integer(names(pt[pt<40]))
```

which gives the desired answer:

```
> ll
[1]  1  2 10  3  4  5 11  6
```

The components of a vector can be provided through the console using the function `scan()`.

Example C.19. To create the vector $(5, 6, 0, 2)$ we can write

```
> x <- scan()
1: 5
2: 6
3: 0
4: 2
5:
Read 4 items
> x
[1] 5 6 0 2
```

An alternative method is using the function `data.entry`. For example, if we write

```
 y <- vector("numeric",5)
> data.entry(y)
```

the vector `y` is created and, as an effect of the `data.entry` function, a window opens which allows us to enter the components of `y`.

The function `append` allows the insertion of a vector in another vector at a prescribed position.

Example C.20. After creating the vectors `x` and `y`, the vector `y` is inserted in `x` after the 4th component using the following commands:

```
> x <- c(1,2,3,4,5,6,7)
> y <- c(10,11)
> append(x,y,4)
[1]  1  2  3  4 10 11  5  6  7
```

To add components at the beginning or the end of a vector we can use the function `c`.

Example C.21. Let `x` be a vector having the components 9,4, and 20. To add 15 and 16 at the beginning of `x`, and 50, 60, 70 at the end we can write:

```
x <- c(9,4,10)
> print(x)
[1]  9  4 10
> x <- c(15,16,x)
> print(x)
[1] 15 16  9  4 10
> x <- c(x,50,60,70)
> print(x)
[1] 15 16  9  4 10 50 60 70
```

The function `str` allows showing attributes of existing objects.

Example C.22. To obtain the attributes of `x` we can write:

```
> str(x)
num [1:8] 15 16 9 4 10 50 60 70
```

To verify that an object is a vector we can use the function `is.vector()` as in

```
> is.vector(x)
(1) TRUE
```

It is possible to access only certain portions of a vector. The following list summarizes several access methods:

`v[k]`	the k^{th} component of `v`
`v[k:h]`	components of `v` between k^{th} and h^{th}
`v[c(2,4,6)]`	the second, fourth and sixth components of `v`
`v[-4]`	all components of `v` except the fourth
`v[v > 2]`	all components of `v` that are greater than 2

The function `seq` generates arithmetc progessions. Its most common usage is

```
seq(from = a, to = b, by = r),
```

yielding an arithmetic progression with initial term `a`, increment `r`, and having its last term not larger than *b*.

Example C.23. We generate two arithmetic progessions using the function `seq`:

```
> seq(from=5, by=2,to=12)
[1]  5  7  9 11
> seq(from=5, by=2,to=13)
[1]  5  7  9 11 13
```

Also, `seq` can be used to extract selected components of a vector as in

```
> x <- c(1,2,3,4,5,6,7,8,9,10)
> x[seq(from=1,by=2,to=8)]
[1] 1 3 5 7
```

Using the function `rep` it is possible to construct a vector containing several copies of another vector.

Example C.24. After constructing the vector `v`, the vector `w` is put together from three copies of `v`:

```
> v <- c(1,2,3)
> w <- rep(v,times=3)
> w
[1] 1 2 3 1 2 3 1 2 3
```

It is possible to eliminate duplicate components of a vector using the function `unique`; the results can be sorted using the function `sort`.

Example C.25. Starting from the vector `v` we eliminate duplicate components using the function `unique`; then, we sort the resulting vector `w`:

```
> v <- c(1,3,2,4,1,4,3,1)
> w <- unique(v)
> w
[1] 1 3 2 4
> z <- sort(w)
> z
[1] 1 2 3 4
```

Equivalently, the functions `unique` and `sort` can be cascades as in:

```
> t <- sort(unique(v))
> t
[1] 1 2 3 4
```

A number of set-theoretical operations can be performed on vectors. For instance, starting from the vectors `v` and `z` and using the `union` function one can construct a vector that contains one copy of every component of `v` and `z`.

Example C.26. If `v` is the vector introduced in Example C.25 then the"union" of `v` and `z`, defined here is:

```
> z <- c(1,4,6)
> union(v,z)
[1] 1 3 2 4 6
```

Similarly, the "intersection" is given by

```
> intersect(v,z)
[1] 1 4
```

and the "set difference" is

```
> setdiff(v,z)
[1] 3 2
> setdiff(z,v)
[1] 6
```

The equality of the set of components of two vectors can be tested using the function `setequal`:

```
> y <- c(6,4,4,1)
> y
[1] 6 4 4 1
> setequal(z,y)
[1] TRUE
```

Note that the union (or the intersection) of a vector with itself results into a vector that contains the distinct components of the vector, as shown below:

```
> union(v,v)
[1] 1 3 2 4
> intersect(v,v)
[1] 1 3 2 4
```

The inclusion between the sets of components of vectors can be texted using the function `is.element`.

Example C.27. After defining the vectors u, v, w as

```
> u <- c(1,2,2,3,4)
> v <- c(1,2,4)
> w <- c(1,2,5)
```

we can test inclusion between the sets of components of these vectors by writing

```
> is.element(u,v)
[1]  TRUE  TRUE  TRUE FALSE  TRUE
> is.element(v,u)
[1] TRUE TRUE TRUE
> is.element(w,u)
[1]  TRUE  TRUE FALSE
```

The function `match` can also be used to determine if the components of a vector `y` are contained in the set of components of a vector `x`. The result is a vector with the same length as `x`. Components of `y` that occur in `x` are printed in the same position as they occur in `x`; the remaining components of `x` are designed by the special value `NA`, which stands for "not available".

Example C.28. The usage of the function `match` is shown below:

```
> x <- c(1,2,3,2,1,5)
> y <- c(1,3)
> match(x,y)
[1]  1 NA  2 NA  1 NA
> match(y,x)
[1] 1 3
```

Complex vectors can also be used in **R** and several functions that apply to complex numbers are given below.

`Re`	the real part of a complex number
`Im`	the imaginary part of a complex number
`Conj`	the conjugate of a complex number
`Mod`	the module of a complex number
`Arg`	the module of a complex number

These functions can be applied not only to complex numbers but also to vectors with complex components.

Example C.29. Below we compute the conjugates of the components of the complex vector `v` as well as the modules of its components:

```
> v <- c(1+2i,1-3i,5+4i)
> Conj(v)
[1] 1-2i 1+3i 5-4i
> Mod(v)
[1] 2.236068 3.162278 6.403124
```

The `c` function allows the creation of a vector of strings of characters.

Example C.30. The vector `nes` that contains designation of New England states can be created as

```
> nes <- c("MA","ME","NH","VT","CT")
> nes
[1] "MA" "ME" "NH" "VT" "CT"
```

R contains many common operations that involve strings. For example, the functions `tolower` and `toupper` convert a string of characters to lower case, and to upper case, respectively. These functions are applicable to character strings, or to vectors of strings as shown next.

Example C.31. The components of the vector `nes` defined in Example C.30 are converted to lower case in the following **R** fragment:

```
> nes1 <- tolower(nes)
> nes1
[1] "ma" "me" "nh" "vt" "ct"
```

The function `nchar` counts the number of characters in a string and can be applied to a vector to produce the number of characters in each component.

Example C.32. Using the same vector `nes` introduced above we have:

```
> nchar(nes)
[1] 2 2 2 2 2
```

The function `grep` extracts the the indices of the components of a vector of strings of characters that contain a certain substring.

Example C.33. The function `grep` is seeks to determine the components of a vector that contain the pattern "Vic":

```
> w <- c("Victoria","Albert","Victor","Alfred")
> grep("Vic",w)
[1] 1 3
```

If we wish to print the actual components of `w` that contain the "Vic" substring we write

```
> p <- grep("Vic",w)
> print(w[p])
[1] "Victoria" "Victor"
```

To print those components of `w` that do not contain the substring "Vic" we write

```
> print(w[-p])
[1] "Albert" "Alfred"
```

The special vectors `letters` and `LETTERS` contain the small and capital letters of the Latin alphabet, as shown next.

```
> letters
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"
> LETTERS
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
[20] "T" "U" "V" "W" "X" "Y" "Z"
```

A *factor* is a vector used to specify a grouping of the components of other vectors of the same length.

Conceptually, factors are vectors whose components take on a limited number of different values; such variables are often refered to as *categorical variables*. One of the most important uses of factors is in statistical modeling; since categorical variables enter into statistical models differently than continuous variables, storing data as factors insures that the modeling functions will treat such data correctly.

The `factor` function is used to create a factor. The only required argument to factor is a vector of values which will be returned as a vector of factor values. Both numeric and character variables can be made into factors, but a factor's levels will always be character values.

The possible levels for a factor can be accessed through the `levels` command.

Factors represent an efficient way of storing character values. While the data itself is stored as a vector of integers and each character value is stored only *once*.

Example C.34. We construct the factor `fdata` starting from the vector `data`. The levels of `fdata` are displayed using the function `factor`

```
> data = c(1,2,3,4,4,3,2,1)
> fdata = factor(data)
> fdata
[1] 1 2 3 4 4 3 2 1
Levels: 1 2 3 4
```

Starting from the `data` vector another factor is created, where levels represent the roman numerical equivalents:

```
> rdata = factor(data,labels = c("I","II","III","IV"))
> rdata
[1] I   II  III IV  IV  III II  I
Levels: I II III IV
```

To impose a certain order on the levels of a factor we can write

```
> ordata = factor(data,labels = c("I","II","III","IV"),ordered=TRUE)
> ordata
[1] I   II  III IV  IV  III II  I
Levels: I < II < III < IV
```

C.3 Matrices and Frames

The function `matrix` serves for transforming a vector into a matrix. Its first argument is a vector, whose components are rearranged to form a matrix. The resulting matrix can be defined using the either the parameter `ncol` specifying the number of columns of the matrix or `nrow` specifying the number of rows of the matrix.

Example C.35. Starting from the vector v defined as

```
v <- c(1,2,3,4,5,6,7,8,9,10,11,12)
```

we can produce a matrix with 4 columns or with 4 rows, respectively:

```
> m1 <- matrix(v,ncol=4)
> m1
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> m2 <- matrix(v,nrow=4)
> m2
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
```

```
[3,]    3    7   11
[4,]    4    8   12
```

Note that for the matrices `m1` or `m2` the order of the elements obtained by reading the columns successively is the order of the elements in `v`. This corresponds to setting the parameter `byrow` to `FALSE`:

```
 m3 <- matrix(v,ncol=4,byrow=FALSE)
> m3
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

An alternative ordering can be obtaining by setting `byrow` to `TRUE`:

```
> m4 <- matrix(v,ncol=4,byrow=TRUE)
> m4
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

To enter a matrix from the keyboard one could use the function `scan`.

Example C.36. Below, we enter a matrix having two rows. Note that the matrix must contain an even number of entries placed on two rows; if this is not the case an error message will be produced.

```
> matrix(scan(),nrow=2,byrow=TRUE)
1: 5
2: 2
3: 4
4: 9
5: 6
6: 1
7:
Read 6 items
     [,1] [,2] [,3]
[1,]    5    2    4
[2,]    9    6    1
```

To generate a diagonal matrix we can use the function `diag`.

Example C.37. A diagonal matrix containing the sequence $(1, 2, 3, 4)$ on its diagonal can be created as follows:

```
> diag(1:4)
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    2    0    0
[3,]    0    0    3    0
[4,]    0    0    0    4
```

The next example illustrates various methods for accessing matrix components.

Example C.38. To extract the third column of the matrix `a` defined below we write `a[,3]`; the third row of `a` is obtained with `a[3,]`. To extract all rows except the first we write `a[-1,]`, as shown below.

```
> a
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12

> a[,3]
[1]  3  7 11

> a[3,]
[1]  9 10 11 12

> a[-1,]
     [,1] [,2] [,3] [,4]
[1,]    5    6    7    8
[2,]    9   10   11   12

> a[,-2]
     [,1] [,2] [,3]
[1,]    1    3    4
[2,]    5    7    8
[3,]    9   11   12
> a[2,3]
[1] 7
```

The functions `cbind` and `rbind` start with a number of vectors and produce a matrix. In the first case, vectors are treated as columns; in the second, they are treated as rows.

Example C.39. Starting from the vectors `x`, `y`, and `z` defined as

```
> x <- c(1,2,3,4)
> y <- c(5,6,7,8)
> z <- c(9,10,11,12)
```

we define a matrix having `x`, `y`, and `z` as columns using the function `cbind` and a similar matrix having these vectors as rows using the function `rbind`:

```
> cbind(x,y,z)
     x y  z
[1,] 1 5  9
[2,] 2 6 10
[3,] 3 7 11
[4,] 4 8 12

> rbind(x,y,z)
  [,1] [,2] [,3] [,4]
x    1    2    3    4
y    5    6    7    8
z    9   10   11   12
>
```

Array of higher dimensionalities can be built using the function `array`. The first argument of this function is a vector that defines the content of the array; the second argument is a list of dimensions.

Example C.40. To arrange numbers ranging from 1 to 24 in a 3-dimensional array of dimensions 2, 3, and 4 we can write:

```
> array(1:24,c(2,3,4))
, , 1

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

, , 2

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

, , 3

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

```
, , 4

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

To compute the determinant of a square matrix defined by

```
s <- matrix(c(1,0,-1,2,3,4,1,8,9),nrow=3)
```

we write

```
> det(s)
[1] -18
```

The distance between rows of a matrix can be computed by the function `dist` which produces a triangular object as in

```
> m <- matrix(c(1,2,3,4,2,3,4,5,3,4,5,6),nrow=3)
> m
     [,1] [,2] [,3] [,4]
[1,]    1    4    4    4
[2,]    2    2    5    5
[3,]    3    3    3    6
> dist(m)
         1        2
2 2.645751
3 3.162278 2.645751
```

The value returned by `dist` is useful for various functions in the package `class` that deals with classification. However, if a matrix form of these distances is needed we can use the function `as.matrix`:

```
> as.matrix(dist(m))
         1        2        3
1 0.000000 2.645751 3.162278
2 2.645751 0.000000 2.645751
3 3.162278 2.645751 0.000000
```

A *data frame* is a similar to a matrix, so it is a tabular object. However, unlike matrices, its columns have names and may vary in type.

The definition of a data frame begins with the definitions of the vectors that constitute its columns. Then, these columns are assembled into a data frame using the function `data.frame`.

Example C.41. A data frame `wh` can be created as follows:

```
> weight <- c(180,150,210,140,170)
> height <- c(1.78,1.64,1.90,1.50,1.89)
> wh <- data.frame(weight,height)
```

This results in the data frame `wh` shown below:

```
> wh
  weight height
1    180   1.78
2    150   1.64
3    210   1.90
4    140   1.50
5    170   1.89
```

An alternative technique for creating a data frame is reading its content from a csv file by using the function `read.csv`.

Example C.42. Suppose that we have a `csv` file named `consEU.csv`. By default, **R** assumes that the file contains a header and, in our case, the header consists of

```
code prot fat
```

Thus, upon writing

```
 fatprot <- read.csv("C:/Users/Dan/Desktop/cs724-SPRING2014/handouts/consEU.csv")
```

and

```
> fatprot
```

System **R** will return:

```
   code prot fat
1    AL   97  87
2    AT  107 155
3    BY   88  97
4    BE   97 164
5    BA   86  67
6    BG   79 101
7    HR   74  97
8    CY   99 133
9    CZ   95 121
10   DK  108 135
11   EE   88  96
12   FI  105 127
13   FR  117 164
14   GE   77  58
15   DE   99 142
16   GR  117 146
```

```
17   HU   90 145
18   IS  128 143
19   IE  115 135
20   IT  113 158
21   LV   87 116
22   LT  112 105
23   LU  124 164
24   MK   72 102
25   MT  116 110
26   MD   73  59
27   NL  103 135
28   NO  104 144
29   PL  100 113
30   PT  114 137
31   RO  110 107
32   RU   92  87
33   YU   75 116
34   SK   72 108
35   SI  102 131
36   ES  109 152
37   CH   91 152
```

`fatprot` is a new data frame.

The country codes can be obtained using the "$" operator:

```
fatprot$code
```

which returns

```
> fatprot$code
 [1] AL AT BY BE BA BG HR CY CZ DK EE FI FR GE DE GR HU IS IE IT LV LT LU MK MT
[26] MD NL NO PL PT RO RU YU SK SI ES CH
37 Levels: AL AT BA BE BG BY CH CY CZ DE DK EE ES FI FR GE GR HR HU IE ... YU
```

The categorical attribute `code` has 37 values in its domain; these values are the *levels* of the attribute `code`.

If the operator "$" is used in conjunction with a numerical attribute as in

```
fatprot$fat
```

R returns the list of values that occur in the data frame under `fat`:

```
> fatprot$fat
 [1]  87 155  97 164  67 101  97 133 121 135  96 127 164  58 142 146 145 143 135
[20] 158 116 105 164 102 110  59 135 144 113 137 107  87 116 108 131 152 152
```

Projections of a dataframe can be obtained by enclosing a list of attributes between square brackets. For example, we can write:

```
fatprot[c("fat","prot")]
```

This will return

```
   fat prot
1   87   97
2  155  107
3   97   88
4  164   97
5   67   86
6  101   79
7   97   74
8  133   99
9  121   95
10 135  108
11  96   88
12 127  105
13 164  117
14  58   77
15 142   99
16 146  117
17 145   90
18 143  128
19 135  115
20 158  113
21 116   87
22 105  112
23 164  124
24 102   72
25 110  116
26  59   73
27 135  103
28 144  104
29 113  100
30 137  114
31 107  110
32  87   92
33 116   75
34 108   72
35 131  102
36 152  109
37 152   91
```

Equivalently, we could enter

```
fatprot[2:3]
```

To extract the projection of a selected set of rows of the data frame we need to specify two arrays between the square brackets that designate the projection. To return the `code prot` of the first four rows we can write:

```
> fatprot[c(1,2,3,4),c(1,2)]
  code prot
1   AL   97
2   AT  107
3   BY   88
4   BE   97
```

To inspect the structure of a data frame **R** one could use the function `str`.

Example C.43. The function call `str(fatprot)` returns

```
> str(fatprot)
'data.frame':   37 obs. of  3 variables:
 $ code: Factor w/ 37 levels "AL","AT","BA",..: 1 2 6 4 3 5 18 8 9 11 ...
 $ prot: int  97 107 88 97 86 79 74 99 95 108 ...
 $ fat : int  87 155 97 164 67 101 97 133 121 135 ...
```

C.4 Linear Algebra in R

The sum of matrices having the same format is computed using the usual operator "+".

Example C.44. Let `a` and `b` be the 2×3-matrices constructed as

```
> a <- rbind(c(1,-1,0),c(2,3,1))
> a
     [,1] [,2] [,3]
[1,]    1   -1    0
[2,]    2    3    1
> b <- rbind(c(2,2,3),c(-1,2,1))
> b
     [,1] [,2] [,3]
[1,]    2    2    3
[2,]   -1    2    1
```

Their sum is obtained as

```
> a+b
     [,1] [,2] [,3]
[1,]    3    1    3
[2,]    1    5    2
```

Matrix multiplication is done using the operator `%*%`. Multiplication of a matrix `A` by a constant `c` is done by simply writing `c*A`.

Example C.45. Starting from the vectors `x`, `y`, `u`, `v` defined by

```
> x <- c(1,2,3,4)
> y <- c(4,5,6,7)
> u <- c(1,-1,0,1)
> v <- c(-1,2,1,-1)
```

form the 2×4-matrix `a` and the 4×2-matrix `b` as

```
> a <- rbind(x,y)
> b <-cbind(u,v)
```

Then the `c`, the product of `a` and `b` is obtained as

```
> c <- a %*% b
> c
  u v
x 3 2
y 6 5
```

The product of 4 and `a` is obtained as

```
> 4*a
  [,1] [,2] [,3] [,4]
x    4    8   12   16
y   16   20   24   28
```

The transpose of a metrix `a` is computed using the function `t`.

Example C.46. For the matrix `a` defined as

```
> a
     [,1] [,2] [,3]
[1,]    1   -1    0
[2,]    2    3    1
```

the transpose is computed as

```
> t(a)
     [,1] [,2]
[1,]    1    2
[2,]   -1    3
[3,]    0    1
```

The function `diag` displays a certain polymorphism. When called as `diag(k)`, where `k` is a positive integer, it returns a $k \times k$ unit matrix.

Example C.47. To obtain a 3×3 unit matrix we write:

```
> diag(3)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

On the other hand, when `a` is a $n \times n$-square matrix, `diag(a)` returs an n-dimensional vector that contains the diagonal elements of `a`.

Example C.48. The diagonal elements of a matrix `a` are obtained by applying the function `diag`:

```
> a <- matrix(1:9,c(3,3))
> a
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> diag(a)
[1] 1 5 9
```

The same function applied to an n-dimensional vector produces a $n \times n$-diagonal matrix having the components of `v` as its diagonal elements.

Example C.49. Let `v` be the vector defined as

```
v <- c(1,2,3)
```

Then, we have

```
> v <- c(1,2,3)
> diag(v)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
```

The function `solve` is also polymorphic. When called with a single parameter which is non-singular matrix, it returns its inverse if this inverse exists.

Example C.50. Let `a` be the matrix formed from three rows

```
> l1 <- c(1,-1,0)
> l2 <- c(3,1,2)
> l3 <- c(1,3,1)
> a <- rbind(l1,l2,l3)
```

Its inverse is

```
> solve(a)
        l1    l2   l3
[1,]  1.25 -0.25  0.5
[2,]  0.25 -0.25  0.5
[3,] -2.00  1.00 -1.0
```

When called with two arguments `a` and `b`, where `a` is a $n \times n$-matrix and `b` is an n-dimensional vector, the function return the solution `x` of the linear system `ax = b`.

Example C.51. Using the matrix `a` defined in Example C.50 the function `solve` returns the solution of the system mentioned above:

```
> b <- c(1,2,3)
> solve(a,b)
[1]  2.25  1.25 -3.00
```

This can be easily verified as

```
> a %*% c(2.25,1.25,-3)
   [,1]
l1    1
l2    2
l3    3
```

The reader should pay attention to the fact that in linear algebra vectors are usually denoted as columns, where in **R** they are denoted as rows. Thus, with the usual notations of relational algebra, the linear system discussed above is

$$a \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}.$$

The eigenvalues of a matrix a can be computed using the function `eigen`.

Example C.52. For the matrix `a` defined in Example C.50 we obtain the eigenvalue 2 with multiplicity 2 and the simple eigenvalue -1, as shown below

```
> eigen(a)
eigen() decomposition
\$values
[1]  2+0i  2-0i -1+0i

\$vectors
```

```
              [,1]          [,2]          [,3]
[1,]  0.4082483-0i  0.4082483+0i -0.2407717+0i
[2,] -0.4082483-0i -0.4082483+0i -0.4815434+0i
[3,] -0.8164966+0i -0.8164966+0i  0.8427010+0i
```

Note that the object returned by `eigen` has two components, `eigen(a)$values` and `eigen(a)$vectors`.

Singular vector decompositions of matrices can be computed using the function `svd`. Its standard usage for an $n \times p$-matrix `x` is

```
svd(x, nu, nv)
```

where `nu` is the number of left singular vectors to be computed (which must be between 0 and `n`) and `nv` is the number of right singular vectors to be computed (between 0 and p). The arguments `nu` and `nv` are optional and have the default values n and p, respectively.

```
> svd(x)
$d
[1] 9.5255181 0.5143006

$u
           [,1]       [,2]
[1,] -0.6196295 -0.7848945
[2,] -0.7848945  0.6196295

$v
           [,1]       [,2]
[1,] -0.2298477  0.8834610
[2,] -0.5247448  0.2407825
[3,] -0.8196419 -0.4018960
```

The functions `rowMeans`, `rowSums`, `colMeans`, `colSums` compute the means and sums for rows and columns of a matrix, respectively.

Example C.53. We illustrate the application of the previously listed functions for a 3×4-matrix `a`:

```
> a <- matrix(1:12,nrow=3)
> a
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> rowMeans(a)
```

```
[1] 5.5 6.5 7.5
> rowSums(a)
[1] 22 26 30
> colMeans(a)
[1]  2  5  8 11
> colSums(a)
[1]  6 15 24 33
```

The inner product of two vectors `x` and `y` in $\mathbb{R}^n$ can be computed as `sum(x*y)`:

Example C.54.

```
> x <- c(1,2,-1,0)
> y <- c(2,-1,2,2)
> sum(x*y)
[1] -2
```

If `x` and `y` are **R** vectors, representing the vectors $\mathbf{x}, \mathbf{yy} \in \mathbb{R}^{\mathbf{n}}$, respectively, the outer product $\mathbf{x}'\mathbf{y}$ can be computed as `outer(x.y)`.

Example C.55. Consider the vectors

$$\mathbf{x} = \begin{pmatrix} 1 \\ 2 \\ -1 \\ 0 \end{pmatrix} \in \mathbb{R}^4$$

and

$$\mathbf{y} = \begin{pmatrix} 3 \\ 4 \\ 5 \end{pmatrix} \in \mathbb{R}^3$$

Their counterparts in **R** are:

```
> x <- c(1,2,-1,0)
> x
[1]  1  2 -1  0
> y <- c(3,4,5)
> y
[1] 3 4 5
```

The outer products $\mathbf{x}\mathbf{y}'$ and $\mathbf{y}\mathbf{y}'$ are computed respectively as:

```
> outer(x,y)
     [,1] [,2] [,3]
[1,]    3    4    5
[2,]    6    8   10
[3,]   -3   -4   -5
[4,]    0    0    0
> outer(y,x)
     [,1] [,2] [,3] [,4]
[1,]    3    6   -3    0
[2,]    4    8   -4    0
[3,]    5   10   -5    0
```

If the package `matlab` is installed, new functions become available for matrices.

```
> eye(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
> one(4)
Error in one(4) : could not find function "one"
> ones(4)
     [,1] [,2] [,3] [,4]
[1,]    1    1    1    1
[2,]    1    1    1    1
[3,]    1    1    1    1
[4,]    1    1    1    1
```

When matlab is installed we can multiply matrices using the usual operator `*`.

C.5 Lists

Lists are structures similar to vectors. The main difference is that there is no homogeneity requirements for lists. In other words, list elements may be numbers, strings, or logical values.

Unlike vectors (which can be created using the `c()` functions, lists are created using the function `list`. Elements can be named, which allows a dual access for list components: either by name or by their numbered position. For instance, a list created as

```
> student <- list(name = "John Doe", grad = TRUE, gpa = 3.7)
```

allows us to access its first component either by

```
> student$name
[1] "John Doe"
```

or by

```
> student[1]
$name
[1] "John Doe"
```

C.6 Arrays

Arrays are structures that accomodate multidimensional collections of data. Array components are accessed using the square bracket notation.

To create an array that has three dimension containing $4 \times 2 \times 5$ elements (40 elements in total), we write

```
> a <- array(1:40,c(4,2,5))
```

The content of this array is shown below. Note that for each of the five values of the last index we have a 4×2 array, as shown below.

```
> a
, , 1

     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8

, , 2

     [,1] [,2]
[1,]    9   13
[2,]   10   14
[3,]   11   15
[4,]   12   16

, , 3

     [,1] [,2]
[1,]   17   21
[2,]   18   22
```

```
[3,]   19   23
[4,]   20   24

, , 4

     [,1] [,2]
[1,]   25   29
[2,]   26   30
[3,]   27   31
[4,]   28   32

, , 5

     [,1] [,2]
[1,]   33   37
[2,]   34   38
[3,]   35   39
[4,]   36   40
```

To extract the subarray that corresponds to the value 1 of the first index and the value 2 of the last we write

```
> a[1, ,2]
```

which results in

```
[1]  9 13
```

C.7 Numeric Computations in R

R provides the usual arithmetic operators, $+, -, *$, and ˆ, standing for addition, subtraction, multiplication and power. In addition $x\%\%y$ stands for "x modulo y".

Trigonometric computations can be performed using

```
sin cos tan asin acos atan
```

The hyperbolic functions

```
sinh cosh tanh asinh acosh atanh
```

are also present. Exponentials and logarithms are computed with

```
exp log log10 logb,
```

where `log` computes natural logarithms, log10 computes base 10 logarithms, and log2 computes base 2 logarithms. The general form `log(x,` base) computes logarithms with base base. In addition, log1p(x) computes log(1+x) accurately for $|x| << 1$.

The functions gamma and lgamma compute the Euler function $\Gamma(a) = \int_0^\infty t^{a-1}e^{-t}\,dt$ and $\ln|\Gamma(a)|$, respectively, when $a \notin \{n \in \mathbb{Z} \mid n \leqslant 0\}$.

The functions beta and lbeta compute the beta function $B(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$ and the natural logarithm of the beta function, respectively.

The number of combinations of k among of n objects is computed by choose(n,k) and its logarithm is calculated with lchoose(n,k). Finally, factorial(n) computes $n!$ and its logarithm is produced by lfactorial(x).

Example C.56. An approximation of $\sqrt{1+x}$ can be computed using the binomial series $\sum_{k=0}^{\infty}\binom{\frac{1}{2}}{k}x^k$. For $x = 0.25$ we obtain, using the first 6 terms of this series, the value

```
> k <- 0:5
> sum(choose(1/2,k)* 0.25^k)
[1] 1.118038
```