Appendix G

Programming in R

G.1 Variables in R

A *variable* in **R** can store an atomic vector, group of atomic vectors or a combination of many **R** objects. A valid variable name consists of letters, numbers and the dot or underline characters. The variable name starts with a letter or the dot.

The variables can be assigned values using leftward, rightward and equal to operator. The values of the variables can be printed using `print()` function.

```
# Assignment using equal operator:
v1 = c(0,1,2,3)

# Assignment using leftward operator.
v2 <- c("learn","R")

# Assignment using rightward operator.
c(TRUE,1) -> v3
```

The vector `c(TRUE,1)` has a mix of logical and numeric class. So logical class is coerced to numeric class making `TRUE` as 1.

R is a dynamically typed language. This means that a variable itself is not declared of any data type, rather it gets the data type of the object assigned to it. This also means that we can change a the data type of a variable again and again when using it in a program.

The function `cat` produces a stream of characters:

```
> var_x <- "Hello"
> cat("The class of var_x is ",class(var_x),"\n")
```

```
> var_x <- 34.5
> cat("  Now the class of var_x is ",class(var_x),"\n")

> var_x <- 27L
> cat("   Next the class of var_x becomes ",class(var_x),"\n")
```

The above code produces the following result:

```
The class of var_x is  character
Now the class of var_x is  numeric
Next the class of var_x becomes  integer
```

The function `ls()` can be used to determine all variables currently available. Also the `ls()` function can use patterns to match the variable names as in `print(ls())`.

The `ls()` function can use patterns to match the variable names.

Example G.123. To list the variables starting with the pattern "x1" one could use:

```
> print(ls(pattern = "x1"))
```

Variables can be deleted by using the `rm()` function. Below we delete the variable `v`.

G.2 R packages

R *packages* are collections of functions and data sets developed by the **R** community. The official repository (CRAN) reached 10,000 packages published, and many more are publicly available through the internet.

A package is a suitable way to organize your own work and, if you want to, share it with others. Typically, a package will include

- code
- documentation for the package and its functions,
- and data sets.

The basic information about a package is provided in the description file, where you can find out what the package does, who the author is, what version the documentation belongs to, the date, the type of license its use, and the package dependencies.

The description file can be accessed with the command `packageDescription("package")`, via the documentation of the package `help(package = "package")`, or online in the repository of the package.

For example, for the `stats` package, these ways will be:

- packageDescription("stats")
- help(package = "stats")
- ??stats

A *repository* is a place where packages are located so you can install them from it. Typically, repositories are on-line and accessible to everyone. The most popular repository for R packages is `CRAN` that is a network of `ftp` and web servers maintained by the **R** community around the world. it.

To install a package from `CRAN`, say, `ksvm` you need to use:

```
install.packages("ksvm")
```

After running this, you will receive some messages on the screen.

`CRAN` is a network of servers (each of them called a *mirror*). The user can choose which one to use. If **R** is used through the RGui interface, then the mirror can be selected from a list. The list of available mirrors can be inspected using `getCRANmirrors()`.

To check what packages are installed you can use:

```
installed.packages()
```

Uninstalling a package is straightforward with the function `remove.packages`. For example, to remove the package `ksvm` one could write

```
remove.packages("ksvm")
```

G.3 Logical Operators

Logical operators in **R** serve to construct logical expression and include the following:

&	and	<	less than
—	or	<=	less or equal
==	equal to	>	greater than
!=	not equal	>=	greater or equal
!	negation	`xor`	exlusive or

Example G.124. Consider the following **R** fragment:

```
> x <-sqrt(2)
> x
[1] 1.414214
```

Appendix G

Programming in R

G.1 Variables in R

A *variable* in **R** can store an atomic vector, group of atomic vectors or a combination of many **R** objects. A valid variable name consists of letters, numbers and the dot or underline characters. The variable name starts with a letter or the dot.

The variables can be assigned values using leftward, rightward and equal to operator. The values of the variables can be printed using `print()` function.

```
# Assignment using equal operator:
v1 = c(0,1,2,3)

# Assignment using leftward operator.
v2 <- c("learn","R")

# Assignment using rightward operator.
c(TRUE,1) -> v3
```

The vector `c(TRUE,1)` has a mix of logical and numeric class. So logical class is coerced to numeric class making `TRUE` as 1.

R is a dynamically typed language. This means that a variable itself is not declared of any data type, rather it gets the data type of the object assigned to it. This also means that we can change a the data type of a variable again and again when using it in a program.

The function `cat` produces a stream of characters:

```
> var_x <- "Hello"
> cat("The class of var_x is ",class(var_x),"\n")
```

```
> var_x <- 34.5
> cat("  Now the class of var_x is ",class(var_x),"\n")

> var_x <- 27L
> cat("   Next the class of var_x becomes ",class(var_x),"\n")
```

The above code produces the following result:

```
The class of var_x is  character
Now the class of var_x is  numeric
Next the class of var_x becomes  integer
```

The function `ls()` can be used to determine all variables currently available. Also the `ls()` function can use patterns to match the variable names as in `print(ls())`.

The `ls()` function can use patterns to match the variable names.

Example G.123. To list the variables starting with the pattern "x1" one could use:

```
> print(ls(pattern = "x1"))
```

Variables can be deleted by using the `rm()` function. Below we delete the variable `v`.

G.2 R packages

R *packages* are collections of functions and data sets developed by the **R** community. The official repository (CRAN) reached 10,000 packages published, and many more are publicly available through the internet.

A package is a suitable way to organize your own work and, if you want to, share it with others. Typically, a package will include

- code
- documentation for the package and its functions,
- and data sets.

The basic information about a package is provided in the description file, where you can find out what the package does, who the author is, what version the documentation belongs to, the date, the type of license its use, and the package dependencies.

The description file can be accessed with the command `packageDescription("package")`, via the documentation of the package `help(package = "package")`, or online in the repository of the package.

For example, for the `stats` package, these ways will be:

- packageDescription("stats")
- help(package = "stats")
- ??stats

A *repository* is a place where packages are located so you can install them from it. Typically, repositories are on-line and accessible to everyone. The most popular repository for R packages is `CRAN` that is a network of `ftp` and web servers maintained by the **R** community around the world. it.

To install a package from `CRAN`, say, `ksvm` you need to use:

```
install.packages("ksvm")
```

After running this, you will receive some messages on the screen.

`CRAN` is a network of servers (each of them called a *mirror*). The user can choose which one to use. If **R** is used through the RGui interface, then the mirror can be selected from a list. The list of available mirrors can be inspected using `getCRANmirrors()`.

To check what packages are installed you can use:

```
installed.packages()
```

Uninstalling a package is straightforward with the function `remove.packages`. For example, to remove the package `ksvm` one could write

```
remove.packages("ksvm")
```

G.3 Logical Operators

Logical operators in **R** serve to construct logical expression and include the following:

&	and	<	less than
—	or	<=	less or equal
==	equal to	>	greater than
!=	not equal	>=	greater or equal
!	negation	`xor`	exlusive or

Example G.124. Consider the following **R** fragment:

```
> x <-sqrt(2)
> x
[1] 1.414214
```

```
> x * x
[1] 2
> x*x - 2
[1] 4.440892e-16
> x*x == 2
[1] FALSE
> all.equal(x*x,2)
[1] TRUE
```

Note that the first comparison `x*x == 2` returns `FALSE` because the error introduced in the computation of `> x <-sqrt(2)`. However, if we test equality within the tolerance limit of **R** as we do with `all.equal(x*x,2)`, the result is `TRUE`. By default, this tollerance is $2.220446e-16$.

The logical operators & and | can be applied to vectors of equal length in a componentwise manner. On other hand, the operators && and || evaluate left to right examining only the first element of each vector. Evaluation proceeds only until the result is determined.

The function `any` returns `TRUE` if there exists at least a component that is `TRUE`; the function `all` returns `TRUE` if all components of the vector are `TRUE`.

Example G.125. The next **R** fragment exemplifies the functions previously mentioned.

```
> x <- c(TRUE,FALSE,TRUE)
> y <- c(FALSE,TRUE,TRUE)
> x&y
[1] FALSE FALSE  TRUE
> x&&y
[1] FALSE
> x|y
[1] TRUE TRUE TRUE
> x||y
[1] TRUE
> any(x)
[1] TRUE
> all(x)
[1] FALSE
```

R contains two extended quantifiers, `any` and `all`. Both are applicable to vectors of logical values. The quantifier `any` returns `TRUE` if at least one component of its argument is `TRUE` and `FALSE` otherwise; `all` returns `TRUE` if all components of the vector argument are `TRUE` and `FALSE` otherwise.

This is illustrated next.

```
> u <- c(TRUE,TRUE,FALSE)
> v <- c(TRUE,FALSE,FALSE)
> any(u)
[1] TRUE
> any(v)
[1] TRUE
> all(u)
[1] FALSE
> all(v)
[1] FALSE
```

An optional argument of `any` and `all`, namely `na.rm` allows the removal of all components of the vector argument that equal `NA` when set to `TRUE`.

G.4 Control Structures

We discuss the main control structures of **R** : the conditional instruction `if`, the looping structures `for`, `while`, and `repeat`, and the selector `switch`.

The `if` statement has the form

```
if (Boolean expression) {
   statement
}
```

and results in the execution of the statement if the expression is `TRUE`.

Example G.126. The following **R** fragment contains a single-branch `if`:

```
> x <- 5;
> if (x > 4) {
+ print("x is greater than 4")
+ }
```

Several statements can be grouped together to form a *block* using braces. Since **R** reacts as an interpreter, so blocks are evaluated when a new line is entered after the closing brace.

Another variant of this structure is `if-else`:

```
if (expression)
   statement1
else
   statement 2
```

If the value `v` returned by the expression is a logical vector with the first element `TRUE`, `statement1` is evaluated; otherwise, `statement2` is evaluated. If `v` is a numerical vector, `statement1` is evaluated when the first component is non-zero; otherwise, `statement2` is evaluated.

Example G.127. The following code fragment is self-exaplanatory:

```
x <- 5;
> if (x >= 6) {
+ print("x is at least equal to 6")
+ } else {
+ print("x is less than 6")
+ }
[1] "x is less than 6"
```

The `for` loop has the syntax

```
for (var in vector)
    statement
```

For each element in `vector` (or list) the value of `variable` is set to that element and the `statement` is evaluated. The variable name still exists after the loop has concluded and its value is equal to the last component of the vector.

Example G.128. The following simple loop prints the first five perfect squares:

```
lim <- 5
> for(i in 1:lim)
+    print(i^2)
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

The `while` has the syntax

```
while (condition)
      statement list
```

If the value of `condition` is `TRUE`, `statement list` is evaluated and this process is repeated until the value of `condition` is `FALSE`. In general, the variables that occur in the condition are modified in the statement list; otherwise, statement list is not executed or is repeated indefinitely.

Example G.129. The following fragment generates six terms of the sequence (x_n) defined by the recurrence $x_{n+2} = 2x_{n+1} - x_n$, where $x_0 = 3$ and $x_1 = 8$:

```
> u <- 3
> v <- 8
> n <- 1
> while(n <= 6){
+ z <- 2*v - u
+ u <- v
+ v <- z
+ print(z)
+ n <- n+1
+ }
[1] 13
[1] 18
[1] 23
[1] 28
[1] 33
[1] 38
```

The `repeat` statement has the syntax

```
repeat
   statement list
```

The effect is a repeated evaluation of the `statement list`. This list must contain some computation and a test that determines whether to break out of the loop. This is usually achieved using the statement `break` that triggers the exit from the loop.

Example G.130. The same computation as in Example **??** can be achieved using the following code fragment:

```
 u <- 3
> v <- 8
> n <- 1
> repeat {
+   if(n > 6) break
+   z <- 2*v - u
+   u <- v
+   v <- z
+   print(z)
+   n <- n + 1
+ }
```

It is possible to skip an execution of the body of loop using the statement `next`.

Example G.131. The next code fragment prints the squares of all multiples of 5 less than 20:

```
> for(i in 1:20) {
+   if(i%%5 != 0) next()
+   print(i^2)
+ }
[1] 25
[1] 100
[1] 225
[1] 400
```

The `switch` statement has the syntax

```
switch (statement,list)
```

Its execution begins with the evaluation of the statement. If the resulting values is a number between 1 and the length of the list, the corresponding element of the list is evaluated and the result returned; if this is not the case, a `NULL` value is returned.

Example G.132. In the next fragment, the third element of the list is returned.

```
x <- 3
> switch(x,9,mean(1:10),median(1:99))
[1] 50
```

If the value returned by the statement is a string of characters, the element of the list having a name which matches this string is returned; if there is no match a single unnamed argument is returned as default. If no default is specified, `NULL` is returned.

Example G.133. In the next fragment `x` matches the label of the second element of the list.

```
> x <- "mammal"
> switch(x,reptile="snake",mammal = "horse", bird = "eagle", "neither")
[1] "horse"
```

G.5 Functions in R

A simple function that computes the difference between the largest and the smallest components of a vector can be created by an user using the following **R** code:

```
span <- function(u){
     if(is.numeric(u)){
     return(max(u) - min(u))
     }
     else
      print("Error!")
}
```

A call of span would produce the desired result

```
+ }
> v <- c(6,7,1,9,2)
> span(v)
[1] 8
```

Note that the result of the application of the function is transmitted using the function `return`.

Example G.134. The function `gcd` computes the greatest common divisor of integers `p` and `q` using Euclid's algorithm:

```
gcd <- function(p,q){
       if(p==round(p) && q==round(q)){
         if(q==0) {
           return(p)
         }
         else {
         return(gcd(q,p\%\%q))
         }
       }
       else print("Error: arguments are not integers")
}
```

Note that `gcd` is a recursive function. A call to `gcd` will yield:

```
> gcd(16,36)
[1] 4
```

Also, note that to test whether arguments are integers we used the test `p==round(p)`.

It is possible to define a default value for an argument of a function as in

```
gcd <- function(p,q=10){
       if(p==round(p) && q==round(q)){
         if(q==0) {
           return(p)
         }
         else {
         return(gcd(q,p\%\%q))
         }
       }
       else print("Error: arguments are not integers")
}
```

A call like

```
gcd(25)
```

will return the value `5`; the call is equivalent to `gcd(25,10)`.

A function in **R** is an object of the class `function`; its class can be tested with the `is.function`.

A function `f` can be applied to each component of a list `x` of arguments using the construct `lapply(x,f)`. A list of the same length as x is returned. The function `f` must be able to accept as input any of the components of `x`.

Example G.135. The function `mean` is applied to each component of the list `x` defined in Example E.120 using `lapply(x,mean)` and yields:hr

```
> lapply(x,mean)
$a
[1] 3

$b
[1] 4.535125

$c
[1] 0.25
```

The construct `sapply` is a variant of `lapply` that returns a vector or a matrix as shown next:

```
> sapply(x,mean)
       a        b        c
3.000000 4.535125 0.250000
```

The construct `mapply(f,args)` applies the function `f` to the arguments `args` that follow f.

Example G.136. Let `A` and `B` be two matrices, where `A` $\in \mathbb{R}m \times p$ and `B` $\in \mathbb{R}^{p\times n}$. To compute the special matrix product defined in Section 9.9 we could use the function `multiMatrix` defined below.

```
multiMatrix <- function(A, B)
{
  if(dim(A)[2] == dim(B)[1]){
m = dim(A)[1]
n = dim(B)[2]
C = matrix(0L, m, n)
for (i in 1:m)
for (j in 1:n)
C[i,j] = min(mapply(max, A[i,], B[,j]))
return (C)
  }
  else print("Error: dimension of matrices do not match")
}
```

R has several families of built-in functions, which we review next.

The first family of built-in functions serve for ordering, inverting, or permuting a vector:

`order`
`rev`
`rank`
`sort`

Below we present the simplest variants of using these funtions.

The function `order` returns a permutation of the indices of a vector which rearranges its components in ascending order.

Example G.137. By writing

```
> v <- c(10,50,20,40,90,65)
> order(v)
```

we obtain the permutation of the indices of `v`

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

that corresponds to the increasing order of the components of `v`. Therefore,

```
> v[order(v)]
```

returns the vector sorted in ascending order:

```
[1] 10 20 40 50 65 90
```

The function `rev` returns the components of `v` in reverse order as in

```
> rev(v)
[1] 65 90 40 20 50 10
```

The function `rank` gives the order of the components of `v`:

```
> rank(v)
[1] 1 4 2 3 6 5
```

Finally, the function `sort` rearranges the components of `v` in ascending order:

```
sort(v)
[1] 10 20 40 50 65 90
```

Example G.138.

```
x <- c(50,20,30,60,40,90,70,20,50,50)
o <- order(x)
plot(x[o],rank(x[o])/length(x),type="S")
```

will give a graph of the cummulative distribution of `x`.

Another group of functions is dedicated to computing various sums or products of arrays:

cummax	prod
cummin	sum
cumprod	range
cumsum	which.max
max	which.min
min	pmax
pmin	

The cummulative functions `cummax`, `cummin`, `cumprod`, and `cumsum` return a vector whose elements are the cumulative maxima, minima, products, or sums of the elements of the argument.

Example G.139. The next fragment illustrates an application of the cumulative functions:

```
> v <- c(1,5,2,4,9,6)
> cummax(v)
[1] 1 5 5 5 9 9
> cummin(v)
[1] 1 1 1 1 1 1
> cumprod(v)
[1]    1    5   10   40  360 2160
> cumsum(v)
[1]  1  6  8 12 21 27
```

The use of the functions `prod` and `sum` is clear. The function `range` produces the range of a vector as in

```
> range(v)
[1] 1 9
```

for the vector `v` defined above.

The functions `pmax` and `pmin` can be applied to one or more vectors as arguments; if the length of the vectors do not match, the components of the shorter vector are recycle to render all arguments to the same length. The functions return a single vector giving the 'parallel' maxima (or minima) of the arguments.

Example G.140. For the vectors `v` and`w` defined by

```
> v <- c(1,5,2,4,9,6)
> w <- c(9,7,1,2)
```

we obtain

```
> pmax(v,w)
[1] 9 7 2 4 9 7
```

The functions `which.max` and `which.min` return the indices of the first maximum (minimum) component of a vector, respectively.

Yet another group of function is used to round and truncate numeric arguments:

`ceiling`
`floor`
`round`
`trunc`

The function call `floor(x)` returns the largest integer value which is not greater than `x` while `trunc(x)` returns the integer formed by truncating the value of `x` toward 0. Similarly, `ceiling(x)` returns the smallest integer value which is not smaller than `x`, while `round(x)` the closest integer to x.

R allows trigonometric computations using the functions `sin`, `cos`, `tan`. Natural logarithms can be computed using the `log` function, and decimal logarithms can be obtained using the function `log10`. Absolute values can be computed with the `abs` function which can be applied to real or complex numbers as shown next:

```
> x = 3+4i
> abs(x)
[1] 5
> z = -8
```

```
> abs(z)
[1] 8
```

Further computations with complex numbers can be achieved using the functions `Re`, `Im`, `Arg` (which returns the argument of a complex number in radians), and `Conj`, with obvious effects.

The function `choose(n,k)` returns the binomial coefficient $\binom{n}{k}$.

Example G.141. To print a list of binomial coefficients $\binom{(}{1}0, k)$ for $0 \leqslant k \leqslant 10$ we can write

```
> for(i in 0:10)
+ print(choose(10,i))
[1] 1
[1] 10
[1] 45
[1] 120
[1] 210
[1] 252
[1] 210
[1] 120
[1] 45
[1] 10
[1] 1
```

G.6 Matrix Computations

Built-in **R** functions return, in general an object, whose components can be accessed using the "$" notation. For example, the function `eigen` computes the eigenvalues and the eigenvectors of a matrix m, as shown next.

```
m <- matrix(c(1,2,5,0,2,1,3,1,4),ncol=3)
> m
     [,1] [,2] [,3]
[1,]    1    0    3
[2,]    2    2    1
[3,]    5    1    4

> eigen(m)
$values
[1]  6.934914  1.598564 -1.533479

$vectors
          [,1]        [,2]       [,3]
```

```
[1,] 0.4239582  0.17942954  0.7412273
[2,] 0.3417759 -0.98311922 -0.2423938
[3,] 0.8387185  0.03580004 -0.6259611
```

The components of the list are:

- `values`, a vector containing the eigenvalues of m, sorted in decreasing order, according to their absolute values in the asymmetric case when they might be complex (even for real matrices);
- `vectors`, which is either a square matrix whose columns contain the eigenvectors of m; the vectors are normalized to unit length.

To verify the previous computation we write

```
> r <- eigen(m)
> V <- r$vectors
> lam <- r$values
> D = diag(lam)
> V %*% D %*% ginv(V)
```

which results in:

```
     [,1]          [,2] [,3]
[1,]    1 -2.775558e-16    3
[2,]    2  2.000000e+00    1
[3,]    5  1.000000e+00    4
```

The functions `head(l)` and `tail(l)` return, respectively, an initial and a final part of a list `l`.

Using the `sample` function we can construct the function `rpm` that computes a random permutation matrix as

```
rpm <- function(n) {
require(matlab)
A <- mat.or.vec(n,n)
x <- sample(1:n)
for(i in 1:n) A[i,x[i]] <- 1
return(A)
}
```

The determinant of a matrix can be computed using the function `det`:

```
m <- matrix(c(1,2,5,0,2,1,3,1,4),ncol=3)
d <- det(m)
> m
     [,1] [,2] [,3]
[1,]    1    0    3
[2,]    2    2    1
[3,]    5    1    4
> d
[1] -17
```

Singular value decomposition of a rectangular matrix can be computed using the `svd` function of the package `matlib`. To apply this function to a real or complex matrix `A` we write

```
s <- svd(A)
```

The returned object `s` is a list with the following components:

- `d`: a vector containing the singular values of `M` sorted decreasingly;
- `u`: a matrix whose columns contain the left singular vectors;
- `v`: a matrix whose columns contain the right singular vectors.

Example G.142. For the matrix `A` in $\mathbb{R}^{3\times 4}$ defined as

```
> A <- matrix(c(1:12),3,4)
> A
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

the application of the `svd` function

```
> s <- svd(A)
```

returns the result:

```
\$d
[1] 2.546241e+01 1.290662e+00 8.106158e-18

\$u
           [,1]        [,2]       [,3]
[1,] -0.5045331 -0.76077568  0.4082483
[2,] -0.5745157 -0.05714052 -0.8164966
[3,] -0.6444983  0.64649464  0.4082483

\$v
           [,1]        [,2]       [,3]
[1,] -0.1408767  0.82471435 -0.4777689
[2,] -0.3439463  0.42626394  0.4373910
[3,] -0.5470159  0.02781353  0.5585247
[4,] -0.7500855 -0.37063688 -0.5181468
```

By the definition of singular value decomposition we will have $A = U\mathsf{diag}(d)V'$, where U and V belong to $\mathbb{R}^{3\times 3}$ and $\mathbb{R}^{4\times 3}$. Then we should have $AV = U\mathsf{diag}(D)$. The left member of this equality is

```
> A%*% s$v
          [,1]        [,2]          [,3]
```

```
[1,] -12.84663 -0.98190402  2.220446e-16
[2,] -14.62855 -0.07374908 -1.110223e-16
[3,] -16.41048  0.83440586  4.440892e-16.
```

The second member is

```
> s$u %*% diag(s$d)
          [,1]        [,2]          [,3]
[1,] -12.84663 -0.98190402  3.309325e-18
[2,] -14.62855 -0.07374908 -6.618650e-18
[3,] -16.41048  0.83440586  3.309325e-18
```

and the equality $AV = U\mathsf{diag}(d)$ is approximatively satisfied.

Cholesky factorization can be achieved using the function `chol`.

Example G.143. To factor the positive definite matrix $A \in \mathbb{R}^{3\times 3}$

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

we write:

```
> R <- chol(A)
> R
         [,1]       [,2]       [,3]
[1,] 1.414214 -0.7071068  0.0000000
[2,] 0.000000  1.2247449 -0.8164966
[3,] 0.000000  0.0000000  1.1547005
```

A QR decomposition of a matrix A can be performed using the function `qr`:

```
> d <- qr(A)
```

Example G.144. The Q and R factors can be determined as

```
> Q <- qr.Q(d)
> R <- qr.R(d)
> Q
           [,1]       [,2]      [,3]
[1,] -0.8944272 -0.3585686 0.2672612
[2,]  0.4472136 -0.7171372 0.5345225
[3,]  0.0000000  0.5976143 0.8017837
> R
          [,1]      [,2]       [,3]
```

```
[1,] -2.236068  1.788854 -0.4472136
[2,]  0.000000 -1.673320  1.9123658
[3,]  0.000000  0.000000  1.0690450
```

The result can be verified by writing

```
> Q %*% R
     [,1] [,2]          [,3]
[1,]    2   -1 -4.440892e-16
[2,]   -1    2 -1.000000e+00
[3,]    0   -1  2.000000e+00
```

G.7 Graphics in R

The `plot` function is a basic constituent of the graphics facilities of **R** .

To produce the graph of the `sin` function we could write

```
> x <- seq(from=-pi,by=0.05,to=pi)
> y <- sin(x)
> plot(x,y)
```

This resulting graph can be printed or saved as an extended postscript file, a pdf file (as we see next), etc.

To produce a `pdf` file containing a set of points placed on the parabola $y = x^2$ begin by specifying the directory where the pdf file should be saved. Then, use the following code fragment:

```
> pdf("sincurve.pdf")
> x <- seq(from=-pi,by=0.05,to=pi)
> y <- sin(x)
> plot(x,y)
> dev.off()
```

This would result into a pdf file named `sincurve.pdf` shown in Figure G.1. The final function call `dev.off()` closes the current device; in our case this is the standard output device. Note that unless `dev.off()` the pdf file cannot be used by outside allpications.

Graphics parameters in **R** can be determined using the function `par`. This function may have a large number of parameters and allows the user to set or query current values of graphics parameters.

Example G.145. To change the symbol used to represent points in the previous graph we can write

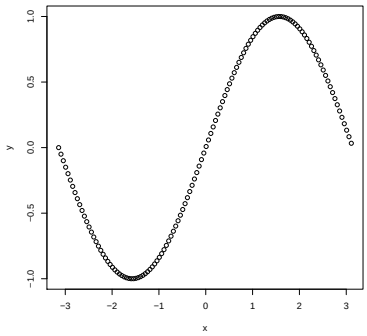

Fig. G.1 Set of points located on the curve $y = sin(x)$.

```
> pdf("sincurve17.pdf")
> x <- seq(from=-pi,by=0.05,to=pi)
> y <- sin(x)
> par(pch=17)
> plot(x,y)
> dev.off()
```

since 17 designates a solid triangle. This would result into the picture shown in Figure G.2

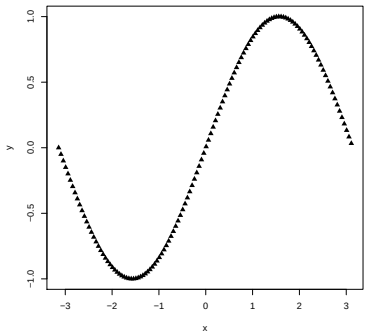

Fig. G.2 Points located on a sinus curve as solid triangles.

There is a vast collection of graphics parameters and the set of values that many parameters range are quite large; these sets are described in the

excellent documentation that accopmpanies each **R** instalation.

To produce several graphical windows as parts of a shared layout we could write, for example

```
> mylayout <- matrix(c(1:4),2)
> layout(mylayout)
> layout.show(4)
```

This will result in a four windows as shown in Figure G.3, that reflect the entries of the matrix `mylayout`.

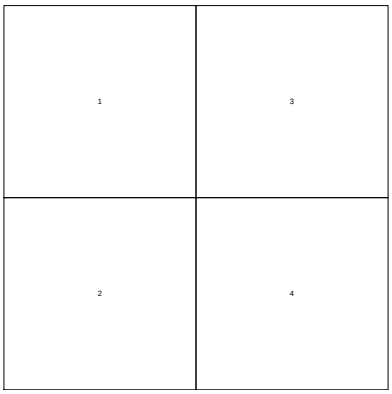

Fig. G.3 Four windows reflecting the matrix `mylayout`.

Example G.146. Next, we give a piece of **R** code that displays four graphs containing sinus curves drawn with a variety of point shapes.

```
> x <- seq(from=-pi,by=0.05,to=pi)
> y <- sin(x)
> mylayout <- matrix(c(1:4),2)
> layout(mylayout)
> plot(x,y,pch=17,sub="plot using pch = 17")
> plot(x,y,pch=18,sub="plot using pch = 18")
> plot(x,y,pch=19,sub="plot using pch = 19")
> plot(x,y,pch=20,sub="plot using pch = 20")
```

Figure G.4 contains the graphs mentioned above.

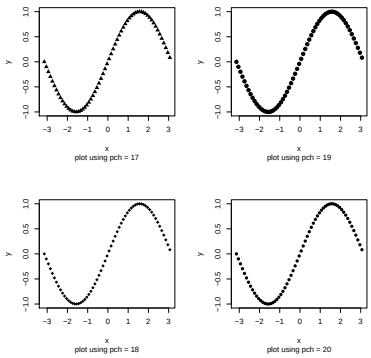

Fig. G.4 Four graphs drawn with various point types.