`A`	the whole matrix
`A(:,:)`	the whole matrix
`A(i:j,k)`	the elements at rows from i to j, at column k
`A(i,j:k)`	the elements at row i, at columns from j to k
`A(i,:)`	the row i
`A(:,j)`	the column j

Figure 24: Access to a matrix with the colon ":" operator.

```
    630.    - 12600.     56700.    - 88200.      44100.
-->A(1:2,3:4)
 ans  =
    1050.   - 1400.
  - 18900.    26880.
```

In some circumstances, it may happen that the index values are the result of a computation. For example, the algorithm may be based on a loop where the index values are updated regularly. In these cases, the syntax

```
A(vi,vj),
```

where `vi,vj` are vectors of index values, can be used to designate the elements of `A` whose subscripts are the elements of `vi` and `vj`. That syntax is illustrated in the following example.

```
-->A = testmatrix("hilb",5)
 A  =

    25.     - 300.      1050.     - 1400.      630.
  - 300.      4800.   - 18900.     26880.   - 12600.
    1050.   - 18900.    79380.    - 117600.    56700.
  - 1400.     26880.  - 117600.    179200.  - 88200.
    630.    - 12600.    56700.    - 88200.     44100.
-->vi=1:2
 vi  =
    1.    2.
-->vj=3:4
 vj  =
    3.    4.
-->A(vi,vj)
 ans  =
    1050.   - 1400.
  - 18900.    26880.
-->vi=vi+1
 vi  =
    2.    3.
-->vj=vj+1
 vj  =
    4.    5.
-->A(vi,vj)
 ans  =
    26880.    - 12600.
  - 117600.     56700.
```

There are many variations on this syntax, and figure 24 presents some of the possible combinations.

For example, in the following session, we use the colon operator in order to interchange two rows of the matrix A.

```
-->A = testmatrix("hilb",3)
 A  =
    9.    - 36.      30.
  - 36.     192.   - 180.
    30.   - 180.     180.
-->A([1 2],:) = A([2 1],:)
 A  =
  - 36.     192.   - 180.
    9.    - 36.      30.
    30.   - 180.     180.
```

We could also interchange the columns of the matrix A with the statement `A(:,[3 1 2])`.

In this section we have analyzed several practical use of the colon operator. Indeed, this operator is used in many scripts where performance matters, since it allows to access many elements of a matrix in just one statement. This is associated with the *vectorization* of scripts, a subject which is central to the Scilab language and is reviewed throughout this document.

4.7 The eye matrix

The `eye` function allows to create the identity matrix with the size which depends on the context. Its name has been chosen in place of `I` in order to avoid the confusion with an index or with the imaginary number.

In the following session, we add 3 to the diagonal elements of the matrix A.

```
-->A = ones(3,3)
 A  =
    1.    1.    1.
    1.    1.    1.
    1.    1.    1.
-->B = A + 3*eye()
 B  =
    4.    1.    1.
    1.    4.    1.
    1.    1.    4.
```

In the following session, we define an identity matrix B with the `eye` function depending on the size of a given matrix A.

```
-->A = ones(2,2)
 A  =
    1.    1.
    1.    1.
-->B = eye(A)
 B  =
    1.    0.
    0.    1.
```

Finally, we can use the `eye(m,n)` syntax in order to create an identity matrix with `m` rows and `n` columns.

4.8 Matrices are dynamic

The size of a matrix can grow or reduce dynamically. This allows to adapt the size of the matrix to the data it contains.

Consider the following session where we define a 2×3 matrix.

```
-->A = [1 2 3; 4 5 6]
 A  =
    1.    2.    3.
    4.    5.    6.
```

In the following session, we insert the value 7 at the indices $(3,1)$. This creates the third row in the matrix, sets the $A(3,1)$ entry to 7 and fills the other values of the newly created row with zeros.

```
-->A(3,1) = 7
 A  =
    1.    2.    3.
    4.    5.    6.
    7.    0.    0.
```

The previous example showed that matrices can grow. In the following session, we see that we can also reduce the size of a matrix. This is done by using the empty matrix ”`[]`” operator in order to delete the third column.

```
-->A(:,3) = []
 A  =
    1.    2.
    4.    5.
    7.    0.
```

We can also change the shape of the matrix with the `matrix` function. The `matrix` function allows to reshape a source matrix into a target matrix with a different size. The transformation is performed column by column, by stacking the elements of the source matrix. In the following session, we reshape the matrix `A`, which has $3 \times 2 = 6$ elements into a row vector with 6 columns.

```
-->B = matrix(A,1,6)
 B  =
    1.    4.    7.    2.    5.    0.
```

4.9 The dollar ”$” operator

Usually, we make use of indices to make reference *from the start* of a matrix. By opposition, the dollar ”$” operator allows to reference elements *from the end* of the matrix. The ”$” operator signifies ”the index corresponding to the last” row or column, depending on the context. This syntax is associated with an algebra, so that the index `$-i` corresponds to the index $\ell - i$, where ℓ is the number of corresponding rows or columns. Various uses of the dollar operator are presented in figure 25.

In the following example, we consider a 3×3 matrix and we access the element `A(2,1) = A(nr-1,nc-2) = A($-1,$-2)` because $nr = 3$ and $nc = 3$.

`A(i,$)`	the element at row i, at column nc
`A($,j)`	the element at row nr, at column j
`A($-i,$-j)`	the element at row $nr - i$, at column $nc - j$

Figure 25: Access to a matrix with the dollar "$" operator. The "$" operator signifies "the last index".

```
-->A=testmatrix("hilb",3)
 A  =
    9.   - 36.     30.
  - 36.    192.  - 180.
    30.  - 180.    180.
-->A($-1,$-2)
 ans  =
  - 36.
```

The dollar "$" operator allows to add elements dynamically at the end of matrices. In the following session, we add a row at the end of the Hilbert matrix.

```
-->A($+1,:) = [1 2 3]
 A  =
    9.   - 36.     30.
  - 36.    192.  - 180.
    30.  - 180.    180.
    1.     2.      3.
```

The "$" operator is used most of the time in the context of the "$+1" statement, which allows to *add at the end* of a matrix. This can be convenient, since it avoids the need of updating the number of rows or columns continuously although it should be used with care, only in the situations where the number of rows or columns cannot be known in advance. The reason is that the interpreter has to internally re-allocate memory for the entire matrix and to copy the old values to the new destination. This can lead to performance penalties and this is why we should be warned against bad uses of this operator. All in all, the only good use of the "$+1" statement is when we do not know in advance the final number of rows or columns.

4.10 Low-level operations

All common algebra operators, such as "+", "-", "*" and "/", are available with real matrices. In the next sections, we focus on the exact signification of these operators, so that many sources of confusion are avoided.

The rules for the "+" and "-" operators are directly applied from the usual algebra. In the following session, we add two 2×2 matrices.

```
-->A = [1 2
-->3 4]
 A  =
    1.    2.
    3.    4.
-->B=[5 6
-->7 8]
 B  =
       5.     6.
       7.     8.
-->A+B
 ans   =
       6.      8.
       10.     12.
```

When we perform an addition of two matrices, if one operand is a 1×1 matrix (i.e., a scalar), the value of this scalar is added to each element of the second matrix. This feature is shown in the following session.

```
-->A = [1 2
-->3 4]
 A   =
       1.     2.
       3.     4.
-->A + 1
 ans   =
       2.     3.
       4.     5.
```

The addition is possible only if the two matrices are conformable to addition. In the following session, we try to add a 2×3 matrix with a 2×2 matrix and check that this is not possible.

```
-->A = [1 2
-->3 4]
 A   =
       1.     2.
       3.     4.
-->B = [1 2 3
-->4 5 6]
 B   =
       1.     2.     3.
       4.     5.     6.
-->A+B
       !--error 8
Inconsistent addition.
```

Elementary operators which are available for matrices are presented in figure 26. The Scilab language provides two division operators, that is,the right division "/" and the left division "\". The right division is so that $X = A/B = AB^{-1}$ is the solution of $XB = A$. The left division is so that $X = A\backslash B = A^{-1}B$ is the solution of $AX = B$. The left division $A\backslash B$ computes the solution of the associated least square problem if A is not a square matrix.

Figure 26 separates the operators which treat the matrices as a whole and the elementwise operators, which are presented in the next section.

4.11 Elementwise operations

If a dot "." is written before an operator, it is associated with an elementwise operator, i.e. the operation is performed element-by-element. For example, with the usual multiplication operator "*", the content of the matrix C=A*B is $c_{ij} =$

`+`	addition	`.+`	elementwise addition
`-`	subtraction	`.-`	elementwise subtraction
`*`	multiplication	`.*`	elementwise multiplication
`/`	right division	`./`	elementwise right division
`\`	left division	`.\`	elementwise left division
`^` or `**`	power, i.e. x^y	`.^`	elementwise power
`'`	transpose and conjugate	`.'`	transpose (but not conjugate)

Figure 26: Matrix operators and elementwise operators.

$\sum_{k=1,n} a_{ik}b_{kj}$. With the elementwise multiplication ".*" operator, the content of the matrix C=A.*B is $c_{ij} = a_{ij}b_{ij}$.

In the following session, two matrices are multiplied with the "*" operator and then with the elementwise ".*" operator, so that we can check that the results are different.

```
-->A = ones(2,2)
 A  =
    1.    1.
    1.    1.
-->B = 2 * ones(2,2)
 B  =
    2.    2.
    2.    2.
-->A*B
 ans  =
    4.    4.
    4.    4.
-->A.*B
 ans  =
    2.    2.
    2.    2.
```

4.12 Conjugate transpose and non-conjugate transpose

There might be some confusion when the elementwise single quote " .' " and the regular single quote " ' " operators are used without a careful knowledge of their exact definitions. With a matrix of doubles containing real values, the single quote " ' " operator only transposes the matrix. Instead, when a matrix of doubles containing complex values is used, the single quote " ' " operator transposes *and conjugates* the matrix. Hence, the operation A=Z' produces a matrix with entries $A_{jk} = X_{kj} - iY_{kj}$, where i is the imaginary number such that $i^2 = -1$ and X and Y are the real and imaginary parts of the matrix Z. The elementwise single quote " .' " always transposes without conjugating the matrix, be it real or complex. Hence, the operation A=Z.' produces a matrix with entries $A_{jk} = X_{kj} + iY_{kj}$.

In the following session, an unsymetric matrix of doubles containing complex values is used, so that the difference between the two operators is obvious.

```
-->A = [1 2;3 4] + %i * [5 6;7 8]
 A  =
    1. + 5.i     2. + 6.i
    3. + 7.i     4. + 8.i
-->A'
 ans  =
    1. - 5.i     3. - 7.i
    2. - 6.i     4. - 8.i
-->A.'
 ans  =
    1. + 5.i     3. + 7.i
    2. + 6.i     4. + 8.i
```

In the following session, we define an unsymetric matrix of doubles containing real values and see that the results of the " ' " and " .' " are the same in this particular case.

```
-->B = [1 2;3 4]
 B  =
    1.    2.
    3.    4.
-->B'
 ans  =
    1.    3.
    2.    4.
-->B.'
 ans  =
    1.    3.
    2.    4.
```

Many bugs are created due to this confusion, so that it is mandatory to ask yourself the following question: what happens if my matrix is complex? If the answer is "I want to transpose only", then the elementwise quote " .' " operator is to be used.

4.13 Multiplication of two vectors

Let $\mathbf{u} \in \mathbb{R}^n$ be a column vector and $\mathbf{v}^T \in \mathbb{R}^n$ be a column vector. The matrix $A = \mathbf{u}\mathbf{v}^T$ has entries $A_{ij} = u_i v_j$. In the following Scilab session, we multiply the column vector `u` by the row vector `v` and store the result in the variable `A`.

```
-->u = [1
-->2
-->3]
 u  =
    1.
    2.
    3.
-->v = [4 5 6]
 v  =
    4.    5.    6.
-->u*v
 ans  =
    4.     5.     6.
    8.     10.    12.
    12.    15.    18.
```

`and(A,"r")`	rowwise "and"
`and(A,"c")`	columnwise "and"
`or(A,"r")`	rowwise "or"
`or(A,"c")`	columnwise "or"

Figure 27: Special comparison operators for matrices. The usual operators "<", "&", "|" are also available for matrices, but the `and` and `or` allow to perform rowwise and columnwise operations.

This might lead to some confusion because linear algebra textbooks consider column vectors only. We usually denote by $\mathbf{u} \in \mathbb{R}^n$ a column vector, so that the corresponding row vector is denoted by $\mathbf{u}^T$. In the associated Scilab implementation, a row vector can be directly stored in the variable `u`. It might also be a source of bugs, if the expected vector is expected to be a row vector and is, in fact, a column vector. This is why any algorithm which works only on a particular type of matrix (row vector or column vector) should check that the input vector has indeed the corresponding shape and generate an error if not.

4.14 Comparing two real matrices

Comparison of two matrices is only possible when the matrices have the same shape. The comparison operators presented in figure 16 are indeed performed when the input arguments `A` and `B` are matrices. When two matrices are compared, the result is a matrix of booleans. This matrix can then be combined with operators such as `and`, `or`, which are presented in figure 27. The usual operators "&", "|" are also available for matrices, but the `and` and `or` allow to perform rowwise and columnwise operations.

In the following Scilab session, we create a matrix `A` and compare it against the number `3`. Notice that this comparison is valid because the number `3` is compared element by element against `A`. We then create a matrix `B` and compare the two matrices `A` and `B`. Finally, the `or` function is used to perform a rowwise comparison so that we get the columns where one value in the column of the matrix `A` is greater than one value in the column of the matrix `B`.

```
-->A = [1 2 7
-->6 9 8]
 A  =
    1.    2.    7.
    6.    9.    8.
-->A>3
 ans  =
  F F T
  T T T
-->B=[4 5 6
-->7 8 9]
 B  =
    4.    5.    6.
    7.    8.    9.
-->A>B
```

```
 ans  =
  F F T
  F T F
-->or(A>B,"r")
 ans  =
  F T T
```

4.15 Issues with floating point integers

In this section, we analyze the problems which might arise when we use integers which are stored as floating point numbers. If used without caution, these numbers can lead to disastrous results, as we are going to see.

Assume that the matrix `A` is a square 2×2 matrix. In order to access the element `(2,1)` of this matrix, we can use a constant index, such as `A(2,1)`, which is safe. Moreover, we can access the element `(2,1)` by using floating point values, as in the following session.

```
-->A = testmatrix("hilb",2)
 A  =
    4.  - 6.
  - 6.    12.
```

Now, in order to access the element of the matrix, we can use variables `i` and `j` and use the statement `A(i,j)`, as in the following session.

```
-->i = 2
 i  =
    2.
-->j = 1
 j  =
    1.
-->A(i,j)
 ans  =
  - 6.
```

In the previous session, we emphasize that the variables `i` and `j` are doubles. This is why the following statement is valid.

```
-->A( 2 , [1.0 1.1 1.5 1.9] )
 ans  =
  - 6.  - 6.  - 6.  - 6.
```

The previous session shows that the floating point values 1.0, 1.1, 1.5 and 1.9 are all converted to the integer 1, as if the `int` function had been used to convert the floating point number into an integer. Indeed, the `int` function returns the floating point number storing the integer part of the given floating point number: in some sense, it rounds towards zero. For example, `int(1.0)`, `int(1.1)`, `int(1.5)` and `int(1.9)` all returns 1 while `int(-1.0)`, `int(-1.1)`, `int(-1.5)` and `int(-1.9)` all returns -1.

Notice that the rounding behavior can be explained by the `int` function, and not by the `floor` function. This can be seen when we consider negative integer values, where the returned values of the two functions are not the same. Indeed, assume that the matrix `A` is a 4×4 matrix, as created by `A = testmatrix("hilb",4)` for example. The triu function returns the upper triangle part of the input matrix. The statements `triu(A,-1)`, `triu(A,int(-1.5))` and `triu(A,-1.5)` produce the same result. Instead, the statement `triu(A,floor(-1.5))` produces the same result as `triu(A,-2)`.

This system allows to have a language which is both simple and efficient. But it may also have unfortunate consequences, sometimes leading to unexpected results. For example, consider the following session.

```
-->ones(1,1)
 ans  =
    1.
-->ones(1,(1-0.9)*10)
 ans  =
     []
```

If the computations were performed in exact arithmetic, the result of $(1 - 0.9) * 10$ should be equal to 1, leading to a 1×1 matrix. Instead, the statement `ones(1,(1-0.9)*10)` creates an empty matrix, because the value returned by the `int` function is equal to zero, as presented in the following session.

```
-->int((1-0.9)*10)
 ans  =
    0.
```

Indeed, the decimal number 0.9 cannot be exactly represented as a double precision floating point number. This leads to a rounding, so that the floating point representation of 1-0.9 is slightly smaller than 0.1. When the multiplication (1-0.9)*10 is performed, the floating point result is therefore slightly smaller than 1, as presented in the following session.

```
-->format(25)
-->1-0.9
 ans  =
    0.0999999999999999777955
-->(1-0.9)*10
 ans  =
    0.9999999999999997779554
```

Then the floating point number 0.999999999999999 is considered as the integer zero, which makes the `ones` function return an empty matrix. The origin of this issue is therefore the use of the floating point number 0.1, which should not have been used without caution to perform integer arithmetic with floating point numbers.

4.16 More on elementary functions

In this section, we analyse several elementary functions, especially degree-based trigonometry functions, logarithm functions and matrix-based elementary functions.

Trigonometry functions such as `sin` and `cos` are provided with the classical input argument in radian. But some other trigonometry functions, such as the `cosd` function for example, are taking an input argument in degree. This means that, in the mathematical sense $\text{tand}(x) = \tan(x\pi/180)$. These functions can be easily identified because their name ends with the letter "`d`", e.g. `cosd`, `sind` among others. The key advantage for the degree-based elementary functions is that they provide exact results when their argument has special mathematical values, such as multiples of 90°. Indeed, the implementation of the degree-based functions is based on an argument reduction which is exact for integer values. This allows to get exact floating point results for particular cases.

In the following session, we compute $\sin(\pi)$ and $\mathrm{sind}(180)$, which are mathematically equal, but are associated with different floating point results.

```
-->sin(%pi)
 ans  =
    1.225D-16
-->sind(180)
 ans  =
    0.
```

The fact that $\sin(\pi)$ is not exactly zero is associated with the limited precision of floating point numbers. Indeed, the argument π is stored in memory with a limited number of significant digits, which leads to rounding. Instead, the argument 180 is represented exactly as a floating point number, because it is a small integer. Hence, the value of `sind(180)` is computed by the `sind` function as `sin(0)`. Once again, the number zero is exactly represented by a floating point number. Moreover, the sin function is represented in the $[-\pi/2, \pi/2]$ interval by a polynomial of the form $p(x) = x + x^3 q(x^2)$ where q is a low degree polynomial. Hence, we get `sind(180)=sin(0)=0`, which is the exact result.

The `log` function computes the natural logarithm of the input argument, that is, the inverse of the function `exp`$= e^x$, where e is Euler's constant. In order to compute the logarithm function for other bases, we can use the functions `log10` and `log2`, associated with bases 10 and 2 respectively. In the following session, we compute the values of the `log`, `log10` and `log2` functions for some specific values of x.

```
-->x = [exp(1) exp(2) 1 10 2^1 2^10]
 x  =
    2.7182818    7.3890561    1.    10.    2.    1024.
-->[x' log(x') log10(x') log2(x')]
 ans  =
    2.7182818    1.           0.4342945    1.442695
    7.3890561    2.           0.8685890    2.8853901
    1.           0.           0.           0.
    10.          2.3025851    1.           3.3219281
    2.           0.6931472    0.30103      1.
    1024.        6.9314718    3.0103       10.
```

The first column in the previous table contains various values of x. The column number 2 contains various values of `log(x)`, while the columns 3 and 4 contains various values of `log10(x)` and `log2(x)`.

Most functions are elementwise, that is, given an input matrix, apply the same function for each entry of the matrix. Still, some functions have a special meaning with respect to linear algebra. For example, the matrix exponential of a function is defined by $e^X = \sum_{k=0,\infty} \frac{1}{k!} X^k$, where X is a square $n \times n$ matrix. In order to compute the exponential of a matrix, we can use the `expm` function. Obviously, the elementwise exponential function `exp` does not returns the same result. More generally, the functions which have a special meaning with respect to matrices have a name which ends with the letter "`m`", e.g. `expm`, `sinm`, among others. In the

`chol`	Cholesky factorization
`companion`	companion matrix
`cond`	condition number
`det`	determinant
`inv`	matrix inverse
`linsolve`	linear equation solver
`lsq`	linear least square problems
`lu`	LU factors of Gaussian elimination
`qr`	QR decomposition
`rcond`	inverse condition number
`spec`	eigenvalues
`svd`	singular value decomposition
`testmatrix`	a collection of test matrices
`trace`	trace

Figure 28: Some common functions for linear algebra.

following session, we define a 2×2 matrix containing specific multples of $\pi/2$ and use the `sin` and `sinm` functions.

```
-->A = [%pi/2 %pi; 2*%pi 3*%pi/2]
 A  =
    1.5707963    3.1415927
    6.2831853    4.712389
-->sin(A)
 ans  =
    1.           1.225D-16
  - 2.449D-16  - 1.
-->sinm(A)
 ans  =
  - 0.3333333    0.6666667
    1.3333333    0.3333333
```

4.17 Higher-level linear algebra features

In this section, we briefly introduce higher-level linear algebra features of Scilab.

Scilab has a complete linear algebra library, which is able to manage both dense and sparse matrices. A complete book on linear algebra would be required to make a description of the algorithms provided by Scilab in this field, and this is obviously out of the scope of this document. Figure 28 presents a list of the most common linear algebra functions.

4.18 Exercises

Exercise 4.1 (*Plus one*) Create the vector $(x_1 + 1, x_2 + 1, x_3 + 1, x_4 + 1)$ with the following x.

```
x = 1:4;
```

Exercise 4.2 (*Vectorized multiplication*) Create the vector $(x_1y_1, x_2y_2, x_3y_3, x_4y_4)$ with the following x and y.

```
x = 1:4;
y = 5:8;
```

Exercise 4.3 (***Vectorized invert***) Create the vector $\left(\frac{1}{x_1}, \frac{1}{x_2}, \frac{1}{x_3}, \frac{1}{x_4}\right)$ with the following x.

```
x = 1:4;
```

Exercise 4.4 (***Vectorized division***) Create the vector $\left(\frac{x_1}{y_1}, \frac{x_2}{y_2}, \frac{x_3}{y_3}, \frac{x_4}{y_4}\right)$ with the following x and y.

```
x = 12*(6:9);
y = 1:4;
```

Exercise 4.5 (***Vectorized squaring***) Create the vector $(x_1^2, x_2^2, x_3^2, x_4^2)$ with $x = 1, 2, 3, 4$.

Exercise 4.6 (***Vectorized sinus***) Create the vector $(sin(x_1), sin(x_2), \ldots, sin(x_{10}))$ with x is a vector of 10 values linearly chosen in the interval $[0, \pi]$.

Exercise 4.7 (***Vectorized function***) Compute the $y = f(x)$ values of the function f defined by the equation

$$f(x) = \log_{10}\left(r/10^x + 10^x\right) \tag{1}$$

with $r = 2.220.10^{-16}$ and x a vector of 100 values linearly chosen in the interval $[-16, 0]$.

5 Looping and branching

In this section, we describe how to make conditional statements, that is, we present the `if` statement. We present the `select` statement, which allows to create more complex selections. We present Scilab loops, that is, we present the `for` and `while` statements. We finally present two main tools to manage loops, that is, the `break` and `continue` statements.

5.1 The `if` statement

The `if` statement allows to perform a statement if a condition is satisfied. The `if` uses a boolean variable to perform its choice: if the boolean is true, then the statement is executed. A condition is closed when the `end` keyword is met. In the following script, we display the string "Hello!" if the condition `%t`, which is always true, is satisfied.

```
if ( %t ) then
  disp("Hello !")
end
```

The previous script produces:

```
 Hello !
```

If the condition is not satisfied, the `else` statement allows to perform an alternative statement, as in the following script.

```
if ( %f ) then
  disp("Hello !")
else
  disp("Goodbye !")
end
```

The previous script produces:

```
Goodbye !
```

In order to get a boolean, any comparison operator can be used, e.g. "==", ">", etc... or any function which returns a boolean. In the following session, we use the "==" operator to display the message "Hello !".

```
i = 2
if ( i == 2 ) then
  disp("Hello !")
else
  disp("Goodbye !")
end
```

It is important not to use the "=" operator in the condition, i.e. we must not use the statement `if ( i = 2 ) then`. It is an error, since the "=" operator allows to set a variable: it is different from the comparison operator "==". In case of an error, Scilab warns us that something wrong happened.

```
-->i = 2
 i  =
    2.
-->if ( i = 2 ) then
Warning: obsolete use of '=' instead of '=='.
         !
-->  disp("Hello !")

 Hello !
-->else
-->  disp("Goodbye !")
-->end
```

When we have to combine several conditions, the `elseif` statement is helpful. In the following script, we combine several `elseif` statements in order to manage various values of the integer `i`.

```
i = 2
if ( i == 1 ) then
  disp("Hello !")
elseif ( i == 2 ) then
  disp("Goodbye !")
elseif ( i == 3 ) then
  disp("Tchao !")
else
  disp("Au Revoir !")
end
```

We can use as many `elseif` statements that we need, and this allows to create as complicated branches as required. But if there are many `elseif` statements required, most of the time that implies that a `select` statement should be used instead.

5.2 The `select` statement

The `select` statement allows to combine several branches in a clear and simple way. Depending on the value of a variable, it allows to perform the statement corresponding to the `case` keyword. There can be as many branches as required.

In the following script, we want to display a string which corresponds to the given integer `i`.

```
i = 2
select i
case 1
  disp("One")
case 2
  disp("Two")
case 3
  disp("Three")
else
  disp("Other")
end
```

The previous script prints out ”Two”, as expected.

The `else` branch is used if all the previous `case` conditions are false.

The `else` statement is optional, but is considered a good programming practice. Indeed, even if the programmer thinks that the associated case cannot happen, there may still exist a bug in the logic, so that all the conditions are false while they should not. In this case, if the `else` statement does not interrupt the execution, the remaining statements in the script will be executed. This can lead to unexpected results. In the worst scenario, the script *still works* but with inconsistent results. Debugging such scripts is extremely difficult and may lead to a massive loss of time.

Therefore, the `else` statement should be included in most `select` sequences. In order to manage these unexpected events, we often combine a `select` statement with the `error` function.

The `error` function generates an error associated with the given message. When an error is generated, the execution is interrupted and the interpreter quits all the functions. The call stack is therefore cleared and the script stops.

In the following script, we display a message depending on the the value of the positive variable `i`. If that variable is negative, we generate an error.

```
i = -5;
select i
case 1
  disp("One")
case 2
  disp("Two")
case 3
  disp("Three")
else
  error ( "Unexpected value of the parameter i" )
end
```

The previous script produces the following output.

```
-->i = -5;
-->select i
-->case 1
-->  disp("One")
-->case 2
-->  disp("Two")
```

```
-->case 3
-->  disp("Three")
-->else
-->  error ( "Unexpected value of the parameter i" )
Unexpected value of the parameter i
```

In practice, when we see a `select` statement without the corresponding `else`, we may wonder if the developer wrote this on purpose or based on the assumption that *it will never happen*. Most of the time, this assumption can be discussed.

5.3 The `for` statement

The `for` statement allows to perform loops, i.e. allows to perform a given action several times. Most of the time, a loop is performed over integer values, which go from a starting to an ending index value. We will see, at the end of this section, that the `for` statement is in fact much more general, as it allows to loop through the values of a matrix.

In the following Scilab script, we display the value of `i`, from 1 to 5.

```
for i = 1 : 5
  disp(i)
end
```

The previous script produces the following output.

```
    1.
    2.
    3.
    4.
    5.
```

In the previous example, the loop is performed over a matrix of floating point numbers containing integer values. Indeed, we used the colon ":" operator in order to produce the vector of index values `[1 2 3 4 5]`. The following session shows that the statement `1:5` produces all the required integer values into a row vector.

```
-->i = 1:5
 i  =
    1.    2.    3.    4.    5.
```

We emphasize that, in the previous loop, the matrix `1:5` is a matrix of doubles. Therefore, the variable `i` is also a double. This point will be reviewed later in this section, when we will consider the general form of `for` loops.

We can use a more complete form of the colon operator in order to display the odd integers from 1 to 5. In order to do this, we set the step of the colon operator to 2. This is performed by the following Scilab script.

```
for i = 1 : 2 : 5
  disp(i)
end
```

The previous script produces the following output.

```
    1.
    3.
    5.
```

The colon operator can be used to perform *backward* loops. In the following script, we display the numbers from 5 to 1.

```
for i = 5 : - 1 : 1
  disp(i)
end
```

The previous script produces the following output.

```
    5.
    4.
    3.
    2.
    1.
```

Indeed, the statement `5:-1:1` produces all the required integers.

```
-->i = 5:-1:1
 i  =
    5.    4.    3.    2.    1.
```

The `for` statement is much more general than what we have previously used in this section. Indeed, it allows to browse through the values of many data types, including row matrices and lists. When we perform a `for` loop over the elements of a matrix, this matrix may be a matrix of doubles, strings, integers or polynomials.

In the following example, we perform a `for` loop over the double values of a row matrix containing $(1.5, e, \pi)$.

```
v = [1.5 exp(1) %pi];
for x = v
  disp(x)
end
```

The previous script produces the following output.

```
    1.5
    2.7182818
    3.1415927
```

We emphasize now an important point about the `for` statement. Anytime we use a `for` loop, we must ask ourselves if a vectorized statement could perform the same computation. There can be a 10 to 100 performance factor between vectorized statements and a `for` loop. Vectorization enables to perform fast computations, even in an interpreted environment like Scilab. This is why the `for` loop should be used only when there is no other way to perform the same computation with vectorized functions.

5.4 The `while` statement

The `while` statement allows to perform a loop while a boolean expression is true. At the beginning of the loop, if the expression is true, the statements in the body of the loop are executed. When the expression becomes false (an event which must occur at certain time), the loop is ended.

In the following script, we compute the sum of the numbers i from 1 to 10 with a `while` statement.

```
s = 0
i = 1
while ( i<= 10 )
  s = s + i
  i = i + 1
end
```

At the end of the algorithm, the values of the variables `i` and `s` are:

```
 s  =
    55.
 i  =
    11.
```

It should be clear that the previous example is just an example for the `while` statement. If we really wanted to compute the sum of the numbers from 1 to 10, we should rather use the `sum` function, as in the following session.

```
-->sum(1:10)
 ans  =
    55.
```

The `while` statement has the same performance issue as the `for` statement. This is why vectorized statements should be considered first, before attempting to design an algorithm based on a `while` loop.

5.5 The `break` and `continue` statements

The `break` statement allows to interrupt a loop. Usually, we use this statement in loops where, once some condition is satisfied, the loops should not be continued.

In the following example, we use the `break` statement in order to compute the sum of the integers from 1 to 10. When the variable `i` is greater than 10, the loop is interrupted.

```
s = 0
i = 1
while ( %t )
  if ( i > 10 ) then
    break
  end
  s = s + i
  i = i + 1
end
```

At the end of the algorithm, the values of the variables `i` and `s` are:

```
 s  =
    55.
 i  =
    11.
```

The `continue` statement allows to go on to the next loop, so that the statements in the body of the loop are not executed this time. When the `continue` statement is executed, Scilab skips the other statements and goes directly to the `while` or `for` statement and evaluates the next loop.