```
ls(SCI+"/modules/graphics/demos")
exec(SCI+"/modules/graphics/demos/2d_3d_plots/contourf.dem.sce")
exec(SCI+"/modules/graphics/demos/2d_3d_plots/contourf.dem.sce");
```

3 Basic elements of the language

Scilab is an interpreted language, which means that it allows to manipulate variables in a very dynamic way. In this section, we present the basic features of the language, that is, we show how to create a real variable, and what elementary mathematical functions can be applied to a real variable. If Scilab provided only these features, it would only be a super desktop calculator. Fortunately, it is a lot more and this is the subject of the remaining sections, where we will show how to manage other types of variables, that is booleans, complex numbers, integers and strings.

It seems strange at first, but it is worth to state it right from the start: *in Scilab, everything is a matrix*. To be more accurate, we should write: *all real, complex, boolean, integer, string and polynomial variables are matrices*. Lists and other complex data structures (such as tlists and mlists) are not matrices (but can contain matrices). These complex data structures will not be presented in this document.

This is why we could begin by presenting matrices. Still, we choose to present basic data types first, because Scilab matrices are in fact a special organization of these basic building blocks.

In Scilab, we can manage real and complex numbers. This always leads to some confusion if the context is not clear enough. In the following, when we write *real variable*, we will refer to a variable which content is not complex. Complex variables will be covered in section 3.7 as a special case of real variables. In most cases, real variables and complex variables behave in a very similar way, although some extra care must be taken when complex data is to be processed. Because it would make the presentation cumbersome, we simplify most of the discussions by considering only real variables, taking extra care with complex variables only when needed.

3.1 Creating real variables

In this section, we create real variables and perform simple operations with them.

Scilab is an interpreted language, which implies that there is no need to declare a variable before using it. Variables are created at the moment where they are first set.

In the following example, we create and set the real variable `x` to 1 and perform a multiplication on this variable. In Scilab, the ”=” operator means that we want to set the variable on the left hand side to the value associated with the right hand side (it is not the comparison operator, which syntax is associated with the ”==” operator).

```
-->x=1
 x  =
    1.
-->x = x * 2
```

+	addition
-	subtraction
$*$	multiplication
/	right division, i.e. $x/y = xy^{-1}$
\	left division, i.e. $x\backslash y = x^{-1}y$
^	power, i.e. x^y
$**$	power (same as ^)
'	transpose conjugate

Figure 12: Scilab elementary mathematical operators.

```
 x  =
    2.
```

The value of the variable is displayed each time a statement is executed. That behavior can be suppressed if the line ends with the semicolon ";" character, as in the following example.

```
-->y=1;
-->y=y*2;
```

All the common algebraic operators presented in figure 12 are available in Scilab. Notice that the power operator is represented by the hat "^" character so that computing x^2 in Scilab is performed by the "x^2" expression or equivalently by the "x**2" expression. The single quote " ' " operator will be presented in more depth in section 3.7, which presents complex numbers. It will be reviewed again in section 4.12, which deals with the conjugate transpose of a matrix.

3.2 Variable names

Variable names may be as long as the user wants, but only the first 24 characters are taken into account in Scilab. For consistency, we should consider only variable names which are not made of more than 24 characters. All ASCII letters from "a" to "z", from "A" to "Z" and digits from "0" to "9" are allowed, with the additional characters "%", "_", "#", "!", "$", "?". Notice though that variable names, whose first letter is "%", have a special meaning in Scilab, as we will see in section 3.5, which presents the pre-defined mathematical variables.

Scilab is case sensitive, which means that upper and lower case letters are considered to be different by Scilab. In the following script, we define the two variables A and a and check that these two variables are considered to be different by Scilab.

```
-->A = 2
 A  =
    2.
-->a = 1
 a  =
    1.
-->A
 A  =
    2.
```

```
-->a
 a  =
    1.
```

3.3 Comments and continuation lines

Any line which begins with two slashes "//" is considered by Scilab as a comment and is ignored. There is no possibility to comment out a block of lines, such as with the "/* ... */" comments in the C language.

When an executable statement is too long to be written on a single line, the second and subsequent lines are called continuation lines. In Scilab, any line which ends with two dots is considered to be the start of a new continuation line. In the following session, we give examples of Scilab comments and continuation lines.

```
-->// This is my comment.
-->x=1..
-->+2..
-->+3..
-->+4
 x  =
    10.
```

3.4 Elementary mathematical functions

Tables 13 and 14 present a list of elementary mathematical functions. Most of these functions take one input argument and return one output argument. These functions are vectorized in the sense that their input and output arguments are matrices. This allows to compute data with higher performance, without any loop.

In the following example, we use the `cos` and `sin` functions and check the equality $\cos(x)^2 + \sin(x)^2 = 1$.

```
-->x = cos(2)
 x  =
  - 0.4161468
-->y = sin(2)
 y  =
    0.9092974
-->x^2+y^2
 ans  =
    1.
```

3.5 Pre-defined mathematical variables

In Scilab, several mathematical variables are pre-defined variables, whose names begin with a percent "%" character. The variables which have a mathematical meaning are summarized in figure 15.

In the following example, we use the variable `%pi` to check the mathematical equality $\cos(x)^2 + \sin(x)^2 = 1$.

`acos`	`acosd`	`acosh`	`acoshm`	`acosm`	`acot`	`acotd`	`acoth`
`acsc`	`acscd`	`acsch`	`asec`	`asecd`	`asech`	`asin`	`asind`
`asinh`	`asinhm`	`asinm`	`atan`	`atand`	`atanh`	`atanhm`	`atanm`
`cos`	`cosd`	`cosh`	`coshm`	`cosm`	`cotd`	`cotg`	`coth`
`cothm`	`csc`	`cscd`	`csch`	`sec`	`secd`	`sech`	`sin`
`sinc`	`sind`	`sinh`	`sinhm`	`sinm`	`tan`	`tand`	`tanh`
`tanhm`	`tanm`						

Figure 13: Scilab elementary mathematical functions: trigonometry.

`exp`	`expm`	`log`	`log10`	`log1p`	`log2`	`logm`	`max`
`maxi`	`min`	`mini`	`modulo`	`pmodulo`	`sign`	`signm`	`sqrt`
`sqrtm`							

Figure 14: Scilab elementary mathematical functions: other functions.

```
-->c=cos(%pi)
 c  =
  - 1.
-->s=sin(%pi)
 s  =
    1.225D-16
-->c^2+s^2
 ans  =
    1.
```

The fact that the computed value of $\sin(\pi)$ is *not exactly* equal to 0 is a consequence of the fact that Scilab stores the real numbers with floating point numbers, that is, with limited precision.

3.6 Booleans

Boolean variables can store true or false values. In Scilab, true is written with `%t` or `%T` and false is written with `%f` or `%F`. Figure 16 presents the several comparison operators which are available in Scilab. These operators return boolean values and take as input arguments all basic data types (i.e. real and complex numbers, integers and strings). The comparison operators are reviewed in section 4.14, where the emphasis is made on comparison of matrices.

In the following example, we perform some algebraic computations with Scilab booleans.

`%i`	the imaginary number i
`%e`	Euler's constant e
`%pi`	the mathematical constant π

Figure 15: Pre-defined mathematical variables.

`a&b`	logical and
`a|b`	logical or
`~a`	logical not
`a==b`	true if the two expressions are equal
`a~=b` or `a<>b`	true if the two expressions are different
`a<b`	true if `a` is lower than `b`
`a>b`	true if `a` is greater than `b`
`a<=b`	true if `a` is lower or equal to `b`
`a>=b`	true if `a` is greater or equal to `b`

Figure 16: Comparison operators.

`real`	real part
`imag`	imaginary part
`imult`	multiplication by i, the imaginary unitary
`isreal`	returns true if the variable has no complex entry

Figure 17: Scilab complex numbers elementary functions.

```
-->a=%T
 a  =
  T
-->b = ( 0 == 1 )
 b  =
  F
-->a&b
 ans  =
  F
```

3.7 Complex numbers

Scilab provides complex numbers, which are stored as pairs of floating point numbers. The pre-defined variable `%i` represents the mathematical imaginary number i which satisfies $i^2 = -1$. All elementary functions previously presented before, such as `sin` for example, are overloaded for complex numbers. This means that, if their input argument is a complex number, the output is a complex number. Figure 17 presents functions which allow to manage complex numbers.

In the following example, we set the variable x to $1+i$, and perform several basic operations on it, such as retrieving its real and imaginary parts. Notice how the single quote operator, denoted by " ' ", is used to compute the conjugate of a complex number.

```
-->x= 1+%i
 x  =
    1. + i
-->isreal(x)
 ans  =
  F
```

int8	int16	int32
uint8	uint16	uint32

Figure 18: Scilab integer data types.

```
-->x'
 ans  =
    1. - i
-->y=1-%i
 y  =
    1. - i
-->real(y)
 ans  =
    1.
-->imag(y)
 ans  =
  - 1.
```

We finally check that the equality $(1+i)(1-i) = 1-i^2 = 2$ is verified by Scilab.

```
-->x*y
 ans  =
    2.
```

3.8 Integers

We can create various types of integer variables with Scilab. The functions which allow to create such integers are presented in figure 18.

In this section, we first review the basic features of integers, which are associated with a particular range of values. Then we analyze the conversion between integers. In the final section, we consider the behaviour of integers at the boundaries and focus on portability issues.

3.8.1 Overview of integers

There is a direct link between the number of bits used to store an integer and the range of values that the integer can manage. The range of an integer variable depends on the number of its bits.

- An n-bit signed integer takes its values from the range $[-2^{n-1}, 2^{n-1}-1]$.
- An n-bit unsigned integer takes its values from the range $[0, 2^n - 1]$.

For example, an 8-bit signed integer, as created by the `int8` function, can store values in the range $[-2^7, 2^7-1]$, which simplifies to $[-128, 127]$. The map from the type of integer to the corresponding range of values is presented in figure 19.

In the following session, we check that an unsigned 32-bit integer has values inside the range $[0, 2^{32}-1]$, which simplifies to $[0, 4294967295]$.

```
-->format(25)
-->n=32
```

`y=int8(x)`	a 8-bit signed integer in $[-2^7, 2^7-1] = [-128, 127]$
`y=uint8(x)`	a 8-bit unsigned integer in $[0, 2^8-1] = [0, 255]$
`y=int16(x)`	a 16-bit signed integer in $[-2^{15}, 2^{15}-1] = [-32768, 32767]$
`y=uint16(x)`	a 16-bit unsigned integer in $[0, 2^{16}-1] = [0, 65535]$
`y=int32(x)`	a 32-bit signed integer in $[-2^{31}, 2^{31}-1] = [-2147483648, 2147483647]$
`y=uint32(x)`	a 32-bit unsigned integer in $[0, 2^{32}-1] = [0, 4294967295]$

Figure 19: Scilab integer functions.

iconvert	conversion to integer representation
inttype	type of integers

Figure 20: Scilab integer conversion functions.

```
 n  =
    32.
-->2^n - 1
 ans  =
    4294967295.
-->i = uint32(0)
 i  =
  0
-->j=i-1
 j  =
  4294967295
-->k = j+1
 k  =
  0
```

3.8.2 Conversions between integers

There are functions which allow to convert to and from integer data types. These functions are presented in figure 20.

The `inttype` function allows to inquire about the type of an integer variable. Depending on the type, the function returns a corresponding value, as summarised in table 21.

`inttype(x)`	Type
1	8-bit signed integer
2	16-bit signed integer
4	32-bit signed integer
11	8-bit unsigned integer
12	16-bit unsigned integer
14	32-bit unsigned integer

Figure 21: Types of integers returned by the `inttype` function.

When two integers are added, the types of the operands are analyzed: the resulting integer type is the larger, so that the result can be stored. In the following script, we create an 8-bit integer `i` (which is associated with `inttype`=1) and a 16-bit integer `j` (which is associated with `inttype`=2). The result is stored in `k`, a 16-bit signed integer.

```
-->i=int8(1)
 i  =
  1
-->inttype(i)
 ans  =
    1.
-->j=int16(2)
 j  =
  2
-->inttype(j)
 ans  =
    2.
-->k=i+j
 k  =
  3
-->inttype(k)
 ans  =
    2.
```

3.8.3 Circular integers and portability issues

The behaviour of integers at the range boundaries deserves a particular analysis, since it is different from software to software. In Scilab, the behaviour is *circular*, that is, if an integer at the upper limit is incremented, the next value is at the lower limit. An example of circular behaviour is given in the following session, where

```
-->uint8(0+(-4:4))
 ans  =
  252  253  254  255  0  1  2  3  4
-->uint8(2^8+(-4:4))
 ans  =
  252  253  254  255  0  1  2  3  4
-->int8(2^7+(-4:4))
 ans  =
  124  125  126  127 -128 -127 -126 -125 -124
```

This is in contrast with other mathematical packages, such as Octave or Matlab. In these packages, if an integer is at the upper limit, the next integer stays at the upper limit. In the following Octave session, we execute the same computations as previously.

```
octave-3.2.4.exe:1> uint8(0+(-4:4))
ans =
  0  0  0  0  0  1  2  3  4
octave-3.2.4.exe:5> uint8(2^8+(-4:4))
ans =
  252  253  254  255  255  255  255  255  255
octave-3.2.4.exe:2> int8(2^7+(-4:4))
ans =
      124   125   126   127   127   127   127   127   127
```

The Scilab circular way allows for a greater flexibility in the processing of integers, since it allows to write algorithms with fewer `if` statements. But these algorithms must be checked, particularly if they involve the boundaries. Moreover, translating a script from another computation system into Scilab may lead to different results.

3.9 Floating point integers

In Scilab, the default numerical variable is the double, that is the 64-bit floating point number. This is true even if we write what is mathematically an integer. In [9], Cleve Moler call this number a "flint", a short for floating point integer. In practice, we can safely store integers in the interval $[-2^{52}, 2^{52}]$ into doubles. We emphasize that, provided that all input, intermediate and output integer values are strictly inside the $[-2^{52}, 2^{52}]$ interval, the integer computations are exact. For example, in the following example, we perform the exact addition of two large integers which remain in the "safe" interval.

```
-->format(25)
-->a=  2^40 - 12
 a  =
    1099511627764.
-->b=  2^45 + 3
 b  =
    35184372088835.
-->c = a + b
 c  =
    36283883716599.
```

Instead, when we perform computations outside this interval, we may have unexpected results. For example, in the following session, we see that additions involving terms slightly greater than 2^{53} produce only even values.

```
-->format(25)
-->(2^53 + (1:10))'
 ans  =
    9007199254740992.
    9007199254740994.
    9007199254740996.
    9007199254740996.
    9007199254740996.
    9007199254740998.
    9007199254741000.
    9007199254741000.
    9007199254741000.
    9007199254741002.
```

In the following session, we compute 2^{52} using the floating point integer 2 in the first case, and using the 16-bit integer 2 in the second case. In the first case, no overflow occurs, even if the number is at the limit of 64-bit floating point numbers. In the second case, the result is completely wrong, because the number 2^{52} cannot be represented as a 16-bit integer.

```
-->2^52
```

```
 ans  =
    4503599627370496.
-->uint16(2^52)
 ans  =
  0
```

In section 4.15, we analyze the issues which arise when indexes involved to access the elements of a matrix are doubles.

3.10 The `ans` variable

Whenever we make a computation and do not store the result into an output variable, the result is stored in the default `ans` variable. Once it is defined, we can use this variable as any other Scilab variable.

In the following session, we compute exp(3) so that the result is stored in the `ans` variable. Then we use its content as a regular variable.

```
-->exp(3)
 ans  =
    20.085536923187666792368
-->t = log(ans)
 t  =
    3.
```

In general, the `ans` variable should be used only in an interactive session, in order to progress in the computation without defining a new variable. For example, we may have forgotten to store the result of an interesting computation and do not want to recompute the result. This might be the case after a long sequence of trials and errors, where we experimented several ways to get the result without taking care of actually storing the result. In this interactive case, using `ans` may allow to save some human (or machine) time. Instead, if we are developping a script used in a non-interactive way, it is a bad practice to rely on the `ans` variable and we should store the results in regular variables.

3.11 Strings

Strings can be stored in variables, provided that they are delimited by double quotes ” `"` ”. The concatenation operation is available from the ”+” operator. In the following Scilab session, we define two strings and then concatenate them with the ”+” operator.

```
-->x = "foo"
 x  =
 foo
-->y="bar"
 y  =
 bar
-->x+y
 ans  =
 foobar
```

They are many functions which allow to process strings, including regular expressions. We will not give further details about this topic in this document.

3.12 Dynamic type of variables

When we create and manage variables, Scilab allows to change the type of a variable dynamically. This means that we can create a real value, and then put a string variable in it, as presented in the following session.

```
-->x=1
 x  =
    1.
-->x+1
 ans  =
    2.
-->x="foo"
 x  =
 foo
-->x+"bar"
 ans  =
 foobar
```

We emphasize here that Scilab is not a typed language, that is, we do not have to declare the type of a variable before setting its content. Moreover, the type of a variable can change during the life of the variable.

3.13 Exercises

Exercise 3.1 (*Precedence of operators*) What are the results of the following computations (think about it before trying in Scilab) ?

```
2 * 3 + 4
2 + 3 * 4
2 / 3 + 4
2 + 3 / 4
```

Exercise 3.2 (*Parentheses*) What are the results of the following computations (think about it before trying in Scilab) ?

```
2 * (3 + 4)
(2 + 3) * 4
(2 + 3) / 4
3 / (2 + 4)
```

Exercise 3.3 (*Exponents*) What are the results of the following computations (think about it before trying in Scilab) ?

```
1.23456789d10
1.23456789e10
1.23456789e-5
```

Exercise 3.4 (*Functions*) What are the results of the following computations (think about it before trying in Scilab) ?

```
sqrt(4)
sqrt(9)
sqrt(-1)
sqrt(-2)
exp(1)
log(exp(2))
exp(log(2))
```

```
10^2
log10(10^2)
10^log10(2)
sign(2)
sign(-2)
sign(0)
```

Exercise 3.5 (*Trigonometry*) What are the results of the following computations (think about it before trying in Scilab) ?

```
cos(0)
sin(0)
cos(%pi)
sin(%pi)
cos(%pi/4) - sin(%pi/4)
```

4 Matrices

In the Scilab language, matrices play a central role. In this section, we introduce Scilab matrices and present how to create and query matrices. We also analyze how to access the elements of a matrix, either element by element, or by higher-level operations.

4.1 Overview

In Scilab, the basic data type is the matrix, which is defined by:

- the number of rows,
- the number of columns,
- the type of data.

The data type can be real, integer, boolean, string and polynomial. When two matrices have the same number of rows and columns, we say that the two matrices have the same *shape*.

In Scilab, vectors are a particular case of matrices, where the number of rows (or the number of columns) is equal to 1. Simple scalar variables do not exist in Scilab: a scalar variable is a matrix with 1 row and 1 column. This is why in this chapter, when we analyze the behavior of Scilab matrices, there is the same behavior for row or column vectors (i.e. $n\times 1$ or $1\times n$ matrices) as well as scalars (i.e. 1×1 matrices).

It is fair to say that Scilab was designed mainly for matrices of *real* variables. This allows to perform linear algebra operations with a high-level language.

By design, Scilab was created to be able to perform matrix operations as fast as possible. The building block for this feature is that Scilab matrices are stored in an internal data structure which can be managed at the interpreter level. Most basic linear algebra operations, such as addition, substraction, transpose or dot product are performed by a compiled, optimized, source code. These operations are performed with the common operators "+", "-", "*" and the single quote " ' ", so that, at the Scilab level, the source code is both simple and fast.

With these high-level operators, most matrix algorithms do not require to use loops. In fact, a Scilab script which performs the same operations with loops is typically from 10 to 100 times slower. This feature of Scilab is known as the *vectorization*. In order to get a fast implementation of a given algorithm, the Scilab developer should always use high-level operations, so that each statement processes a matrix (or a vector) instead of a scalar.

More complex tasks of linear algebra, such as the resolution of systems of linear equations $Ax = b$, various decompositions (for example Gauss partial pivotal $PA = LU$), eigenvalue/eigenvector computations, are also performed by compiled and optimized source codes. These operations are performed by common operators like the slash "`/`" or backslash "`\`" or with functions like `spec`, which computes eigenvalues and eigenvectors.

4.2 Create a matrix of real values

There is a simple and efficient syntax to create a matrix with given values. The following is the list of symbols used to define a matrix:

- square brackets "`[`" and "`]`" mark the beginning and the end of the matrix,
- commas "`,`" separate the values in different columns,
- semicolons "`;`" separate the values of different rows.

The following syntax can be used to define a matrix, where blank spaces are optional (but make the line easier to read) and "`...`" denotes intermediate values:

```
A = [a11 , a12 , ..., a1n; a21 , a22 , ..., a2n; ...; an1 , an2 , ..., ann].
```

In the following example, we create a 2×3 matrix of real values.

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

A simpler syntax is available, which does not require to use the comma and semicolon characters. When creating a matrix, the blank space separates the columns while the new line separates the rows, as in the following syntax:

```
A = [a11 a12 ... a1n
a21 a22 ... a2n
...
an1 an2 ... ann]
```

This allows to lighten considerably the management of matrices, as in the following session.

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

`eye`	identity matrix
`linspace`	linearly spaced vector
`ones`	matrix made of ones
`zeros`	matrix made of zeros
`testmatrix`	generate some particular matrices
`grand`	random number generator
`rand`	random number generator

Figure 22: Functions which generate matrices.

The previous syntax for matrices is useful in the situations where matrices are to be written into data files, because it simplifies the human reading (and checking) of the values in the file, and simplifies the reading of the matrix in Scilab.

Several Scilab commands allow to create matrices from a given size, i.e. from a given number of rows and columns. These functions are presented in figure 22. The most commonly used are `eye`, `zeros` and `ones`. These commands take two input arguments, the number of rows and columns of the matrix to generate.

```
-->A = ones(2,3)
 A  =
    1.    1.    1.
    1.    1.    1.
```

4.3 The empty matrix `[]`

An empty matrix can be created by using empty square brackets, as in the following session, where we create a 0×0 matrix.

```
-->A=[]
 A  =
     []
```

This syntax allows to delete the content of a matrix, so that the associated memory is freed.

```
-->A = ones(100,100);
-->A = []
 A  =
     []
```

4.4 Query matrices

The functions in figure 23 allow to query or update a matrix.

The `size` function returns the two output arguments `nr` and `nc`, which are the number of rows and the number of columns.

```
-->A = ones(2,3)
 A  =
    1.    1.    1.
    1.    1.    1.
-->[nr,nc]=size(A)
```

`size`	size of objects
`matrix`	reshape a vector or a matrix to a different size matrix
`resize_matrix`	create a new matrix with a different size

Figure 23: Functions which query or modify matrices.

```
 nc  =
    3.
 nr  =
    2.
```

The `size` function is of important practical value when we design a function, since the processing that we must perform on a given matrix may depend on its shape. For example, to compute the norm of a given matrix, different algorithms may be used depending on if the matrix is a column vector with size $nr \times 1$ and $nr > 0$, a row vector with size $1 \times nc$ and $nc > 0$, or a general matrix with size $nr \times nc$ and $nr, nc > 1$.

The `size` function has also the following syntax

```
nr = size( A , sel )
```

which allows to get only the number of rows or the number of columns and where `sel` can have the following values

- `sel=1` or `sel="r"`, returns the number of rows,
- `sel=2` or `sel="c"`, returns the number of columns.
- `sel="*"`, returns the total number of elements, that is, the number of columns times the number of rows.

In the following session, we use the `size` function in order to compute the total number of elements of a matrix.

```
-->A = ones(2,3)
 A  =
    1.    1.    1.
    1.    1.    1.
-->size(A,"*")
 ans  =
    6.
```

4.5 Accessing the elements of a matrix

There are several methods to access the elements of a matrix `A`:

- the whole matrix, with the `A` syntax,
- element by element with the `A(i,j)` syntax,
- a range of index values with the colon ":" operator.

The colon operator will be reviewed in the next section.

To make a global access to all the elements of the matrix, the simple variable name, for example `A`, can be used. All elementary algebra operations are available for matrices, such as the addition with "+", subtraction with "-", provided that the two matrices have the same size. In the following script, we add all the elements of two matrices.

```
-->A = ones(2,3)
 A  =
    1.    1.    1.
    1.    1.    1.
-->B =  2 * ones(2,3)
 B  =
    2.    2.    2.
    2.    2.    2.
-->A+B
 ans  =
    3.    3.    3.
    3.    3.    3.
```

One element of a matrix can be accessed directly with the `A(i,j)` syntax, provided that i and j are valid index values.

We emphasize that, by default, the first index of a matrix is 1. This contrasts with other languages, such as the C language for instance, where the first index is 0. For example, assume that `A` is an $nr \times nc$ matrix, where nr is the number of rows and nc is the number of columns. Therefore, the value `A(i,j)` has a sense only if the index values i and j satisfy $1 \leq i \leq nr$ and $1 \leq j \leq nc$. If the index values are not valid, an error is generated, as in the following session.

```
-->A = ones(2,3)
 A  =
    1.    1.    1.
    1.    1.    1.
-->A(1,1)
 ans  =
    1.
-->A(12,1)
          !--error 21
Invalid index.
-->A(0,1)
         !--error 21
Invalid index.
```

Direct access to matrix elements with the `A(i,j)` syntax should be used only when no other higher-level Scilab commands can be used. Indeed, Scilab provides many features which allow to produce simpler and faster computations, based on vectorization. One of these features is the colon ":" operator, which is very important in practical situations.

4.6 The colon ":" operator

The simplest syntax of the colon operator is the following:

```
v = i:j
```

where `i` is the starting index and `j` is the ending index with $i \leq j$. This creates the vector $v = (i, i+1, \ldots, j)$. In the following session, we create a vector of index values from `2` to `4` in one statement.

```
-->v = 2:4
 v  =
    2.    3.    4.
```

The complete syntax allows to configure the increment used when generating the index values, i.e. the *step*. The complete syntax for the colon operator is

```
v = i:s:j
```

where `i` is the starting index, `j` is the ending index and `s` is the step. This command creates the vector $v = (i, i+s, i+2s, \ldots, i+ns)$ where n is the greatest integer such that $i + ns \leq j$. If s divides $j - i$, then the last index in the vector of index values is j. In other cases, we have $i + ns < j$. While in most situations, the step `s` is positive, it might also be negative.

In the following session, we create a vector of increasing index values from `3` to `10` with a step equal to `2`.

```
-->v = 3:2:10
 v  =
    3.    5.    7.    9.
```

Notice that the last value in the vector `v` is $i+ns = 9$, which is smaller than $j = 10$.

In the following session, we present two examples where the step is negative. In the first case, the colon operator generates decreasing index values from `10` to `4`. In the second example, the colon operator generates an empty matrix because there are no values lower than 3 and greater than 10 at the same time.

```
-->v = 10:-2:3
 v  =
    10.    8.    6.    4.
-->v = 3:-2:10
 v  =
      []
```

With a vector of index values, we can access the elements of a matrix in a given range, as with the following simplified syntax

```
A(i:j,k:l)
```

where `i,j,k,l` are starting and ending index values. The complete syntax is `A(i:s:j,k:t:l)`, where `s` and `t` are the steps.

For example, suppose that `A` is a 4×5 matrix, and that we want to access the elements $a_{i,j}$ for $i = 1, 2$ and $j = 3, 4$. With the Scilab language, this can be done in just one statement, by using the syntax `A(1:2,3:4)`, as showed in the following session.

```
-->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.
```