Introduction to Scilab version 5

Stéphane Mottelet

Équipe de Mathématiques Appliquées
Département de Génie Informatique
Université de Technologie de Compiègne

Contents

Chapter 1

Starting with Scilab

This first chapter explains the bare necessity to be able to start using Scilab. It explains

- How to enter a matrix, the syntax of Scilab commands and the type of variables
- How to obtain information on the workspace, how to save variables or the whole workspace, how to quit.
- Numbers, mathematical expressions and usual functions
- The way Scilab displays results.

You can already lauch Scilab on your machine in order to be ready to type the examples that will be proposed. You should see something similar to this :

```
         -------------------------------------------
                       scilab-5.2.1

                  Consortium Scilab (DIGITEO)
                Copyright (c) 1989-2010 (INRIA)
                Copyright (c) 1989-2007 (ENPC)
         -------------------------------------------


Startup execution:
  loading initial environment

 -->
```

The sign `-->` means that Scilab is waiting for a command. In the following, each exemple is easily recognized by the font (`typewriter` characters), and each command you have to enter will be preceded by the same sign `-->`.

1.1 Enter a matrix

Scilab essentially works with one main type of object, rectangular matrices contraining real or complex numbers (eventually characters). In some situations, a particular signification is given to 1×1 matrices, which are scalars and to matrices with only on line or columns, which are vectors. The operators and command of Scilab mainly act on this type of object.

There are several ways to entre a matrix in the workspace:

- Enter after the prompt (the $-->$ characters) an explicit list of elements
- Generate this matrix by a sequence of commands
- Load this matrix from a data file

The Scilab language does not include a dimension or type declaration instruction, unlike some low-level languages. The necessary memory space for a matrix is automatically allocated at its creation time.

The simplest way is to type a list of elements, with the following conventions :

- Elements of a same row are separated by spaces of commas
- The list of elements has to be enclosed by brackets [ ]
- Each row (excepted the last one) is ended by a ; (semi-colon)

For example, the following command (you have to type it yourself)

```
--> A = [1  2  3;  4  5  6;  7  8  9]
```

produces the folllowing output

```
A  =

   1.   2.   3.
   4.   5.   6.
   7.   8.   9.
```

The matrix `A` is kept in memory for a future use.

A big matrix can be entered by separating each row by a carriage-return (by pressing the Return key) instead of a semi-colon. For example the above matrix can be entered like this:

```
--> A = [1  2  3
    4  5  6
    7  8  9]
```

1.2 The elements of a matrix

The element of a matrix can be any Scilab expression, e.g.

```
--> x = [-1.3  sqrt(3)  (1+2+3)*4/5]
```

produces the output

```
x  =

 - 1.3   1.7320508   4.8
```

The individual elements of a matrix can be referenced with their row and column indices, given within two parenthesis (). If we carry on our example

```
--> x(5) = abs(x(1))
```

gives the output

```
x  =

 - 1.3   1.7320508   4.8   0.   1.3
```

Note that the size of `x` has been automatically increased to add the new element, and that undefined elements have been initialized to zero.

1.3 Scilab commands and variables

Scilab has a simple and well-defined syntax. A command will always have the form

```
variable = expression
```

or sometimes, more simply

```
expression
```

An expression can be composed of matrices separated with operators and names of variables. The evaluation of an expression produces a matrix which is displayed and stored in `variable`. If the name of the variable and the equal sign are ommited, Scilab creates a variable named `ans` (for answer). For example, the following expression

```
--> 1900/81
```

gives the following output

```
ans =

   23.4568
```

A command is ended by a strike on the Return (or Enter) key. However, if the last character of a command is a semi-colon ; (semi-colon), the result is not displayed even if the command has been interpreted. This is useful for intermediary computations whose result does not need to be displayed. For example

```
--> y = sqrt(3)/2;
```

assigns to y the value of $\sqrt{3}/2$ but does not display the result.

The nam eof a variable must start with a letter, followed by an unspecified number of letters and numbers. Warning : Scilab does only take into account the 19 first characters [1]. Scilab makes also the difference between uppercase and lowercase letters : for example the following commands

```
--> z = log(2);
--> Z
```

give the following output

```
!--error    4
  undefined variable : Z
```

1.4 Obtaining information on the workspace

The preceding examples have created some variables in the Scilab's workspace. To obtain the list of these variables, you can enter the command

```
 --> who_user

Your variables are :

z         ans       y         x         A         home

Uses 38 element on 4965628
```

You find on the beginning of this list the 4 variables created in the preceding examples. To see the size of these variables, you can use the command

[1] this is enough in most cases

```
   --> whos
Name                    Type          Size          Bytes

help                    function                    5176
whos                    function                    9000
z                       constant      1 by 1        24
y                       constant      1 by 1        24
x                       constant      1 by 5        56
A                       constant      3 by 3        88
who_user                function                    6720
.
.
.
```

(we only give the beginning of the list of all variables which actually appear). Each element of a matrix represents 8 bytes in the memory, and for each matrix 16 complimentary bytes are used to store various informations (size, ...). It explains why the matrix `A`, which is of size 3×3, takes up $9 \times 8 + 16 = 88$ bytes.

1.5 Save the workspace and quit

To quit Scilab, just type `quit` or `exit` or close the main command window. One a Scilab session is terminated, all the workspace variables are lost. It is possible to save the workspace before quitting by using the `quit` command

```
--> save work.dat
```

This command saves all the variables in the current working directory in the file `work.dat`. It is possible to load the content of this file in a subsequent session by typing the command `load work.dat`.

You can use the commands `save` and `load` with other file names and you can also save only some variables. The command

```
--> save('work.dat',A,x)
```

saves in the same file only `A` and `x`. The commande `load work.dat` will load all the variables stored in this file (here `A` and `x`).

1.6 Numbers, arihtmetical expressions and usual functions

Scilab uses the classical decimal notation, preceded with the minus sign for negative numbers. It is also possible to add a power of ten factor as in the clasiscal scientific

notation. Some examples of legal Scilab numbers are

```
3          -99          0.0001
9.37821312    1.6023E-20    6.02252e23
```

The relative precision of numbers is given by the permanent variable %eps (permanent means that you cannot clear this variable)

```
--> %eps

eps =

  2.220e-16
```

This value means that the precision is of roughly 16 significative figures. The following examples illustrates this:

```
--> (1 + 1e-16) - 1

ans =

   0.
```

Hence, you mave to keep in mind that computer arithmetics is never exact. The interval of representable numbers goes from 10^{-308} to 10^{308}.

You can build expressions with usual operators, which respect the classical priority order:

- ^ power
- / right division
- \ left division
- * multiplication
- + addition
- - substraction

We will se in a forthcoming section how these operators can be used on matrices. Scilab has also all the classical usual functions:

`sqrt`	square root
`log`	neperian logartihm
`log10`	decimal logarithm
`sin`	sine
`cos`	cosine
`tan`	tangent
`atan`	reciprocial tangent
`exp`	exponential
`cosh`	hyperbolic cosine
`floor`	lower integer part
`round`	round to closest integer
`abs`	absolute value or modulus/magnitude of a complex number
`real`	real part of a complex number
`imag`	imaginary part of a complex number
`modulo`	rest of an Euclidian division

Of course these functions accept complex numbers as arguments. For example

```
--> sqrt(-1)

   ans =

      i

--> exp(%i*%pi/3)

ans  =

0.5 + 0.8660254i
```

Some functions simply return special values. For example, the permanent variable `%pi`

```
--> %pi

ans =

  3.1416
```

returns the value of π, precomputed as the value of `4*atan(1)`.

Chapter 2

Operations on matrices

2.1 Transposition

The special character ’ (prime or apostrophe) denotes the operation of transposition. The following commands

```
--> A = [1  2  3;  4  5  6;  7  8  0]
--> B = A’
```

give the results

```
A =

    1    2    3
    4    5    6
    7    8    0

B =

    1    4    7
    2    5    8
    3    6    0
```

abs the command

```
--> x = [-1  0  2]’

x =

   -1.
    0.
    2.
```

The operation of transposition denotes the Hermitian transposition. If `Z` is a complex matrix, then `Z’` désigne the conjugate transpose of `Z`.

2.2 Addition and substraction

The operators + and - operate on matrices. These operations are valid if the dimensions of matrices are thesame. For example with the matrices of preceding example the addition

```
--> A + x
      !--error    8
inconsistent addition
```

is not valid since A is 3×3 and x is 3×1. On the contrary the following operation is valid:

```
--> C = A + B

C =

    2.    6.   10.
    6.   10.   14.
   10.   14.    0.
```

Addition and substraction are also defined if one of the operands is a scalar, i.e. a 1×1 matrix. In this case the scalar is added or substracted from all the elements of the other matrix:

```
--> z = x - 1

z =

   -2.
   -1.
    1.
```

2.3 Multiplication

The symbol * denotes the operator of matrix multiplication. This opération is valid if the dimensions of opérandes are compatible, i.e. the number of columns of left operand must be equal to the number of rows of the right operand. For example the following operation is not valid

```
--> x*z
    !--error   10
inconsistent multiplication
```

but the following one

```
--> x' * z

ans =

    4.
```

gives the scalar product of x and z. Another valid command is the following:

```
--> b = A*x

b =

      5.
      8.
     -7.
```

The multiplication of a matrix and a scalar is always valid:

```
 --> A*2

ans =

      2.    4.    6.
      8.   10.   12.
     14.   16.    0.
```

2.4 Inverse of a matrix and left division

The inverse of a square inversible matrix can be obtained with the function inv :

```
--> B = inv(A)

 B  =

 - 1.7777778   0.8888889  - 0.1111111
   1.5555556  - 0.7777778   0.2222222
 - 0.1111111   0.2222222  - 0.1111111

--> C=B*A

 C  =
```

```
   1.   0.   0.
   0.   1.   0.
   0.   0.   1.

--> C(2,1)

 ans  =

- 1.110E-16
```

You will note that the off-diagonal terms are not exactly equal to zero but are of the same order as the machine precision.

The matrix division has a sense in Scilab and has the following signification : if B is invertible, the expression `A/B` gives the result of the operation AB^{-1}. Hence it is mathematically equivalent to the expression `A*inv(B)`. For the left division, the expression `A\B`gives the result of the operation $A^{-1}B$. The compatibility of dimensions of both matrices must be respected otherwise this operation does not make sense.

The left division is classically used to compute the solution of a system of linear equations. For example if you want to solve the linear system $Ay = x$, where A is invertible, just type

```
--> y = A\x

y  =

   1.5555556
 - 1.1111111
 - 0.1111111
```

When Scilab interprets this expression, the matrix `A` is not inverted and then right multiplied by `x`; it actually solves the system of equations. The solution can be checked by typing the following commands:

```
--> A*y - x

ans =

  1.0E-15 *

   0.2220446
   0.2775558
   0.
```

Once again, note the presence of errors of the order of machine precision.

2.5 Element-wise operations

Usual operations on matrices can be done element-wise; this amounts to considering matrices as and not mathematical objects representing linear applciations. For addition and substraction both point of views are the same since these two operations already work element-wise.

The operator `.*` denotes element-wise multiplication. If `A` and `B` have the same dimensions, then `A.*B` denotes the array whose elements are the product of individual elements of `A` and `B`. For example if `x` and `y` are defined by:

```
--> x = [1  2  3];  y = [4  5  6];
```

then the command

```
--> z = x .* y
```

gives the result

```
z =

    4.    10.    18.
```

The division works in the same manner:

```
--> z = x ./ y

z =

    0.25   0.4   0.5
```

The operator `.^` denotes element-wise power function:

```
--> x .^ y

ans =

   1.    32.    729.

--> x .^ 2

ans =
```

```
     1.    4.    9.

--> 2 .^ x

ans =

     2.    4.    8.
```

Note that for `.^` one of the two operands can be a scalar.

2.6 Relational operators

Six relational operators are available for compare two matrices of equal dimensions:

`<`	lower than
`<=`	lower than or equal
`>`	greater than
`>=`	greater tha or equal
`==`	equal
`<>`	different

Scilab compares corresponding elements; the result is a matrix of boolean constants , the `F` representing the value “false” and the `T` the value “true”. For example

```
--> 2 + 2 <> 4

ans =

    F
```

Relational operators allows to see, in a matrix, which elements verify a given conditions. For example let us take the matrix

```
--> A = [1  -1  2; -2  -4  1; 8  1  -1]

A =

     1.   -1.    2.
    -2.   -4.    1.
     8.    1.   -1.
```

The command

```
--> P = (A < 0)

 P  =

 F T F
 T T F
 F F T
```

returns a matrix P showing by a T negative elements of A.

2.7 Logical operators

The operators &, | and ~ denote respectively the operators "and", "or" and "not". They are used to build some logical expressions. For example if we take the matrix of preceding example, the command

```
--> P = (A < 0) & (modulo(A,2) == 0)

P =

 F F F
 T T F
 F F F
```

allows to find in A the negative and even elements.

2.8 Usual functions applied to a matrix

Usual functions acting on reals and complex numbers also apply element-wise on matrices. For example :

```
--> A = [0  1/4;  1/2  3/4]

A =

   0.    0.25
   0.5   0.75

--> cos(%pi*A)

ans =
```

```
1.    0.7071068
0.  - 0.7071068
```

Chapter 3

Manipulating matrices and vectors

3.1 How to generate matrices and vectors

One can easily generate a null matrix of given size with the command `zeros`:

```
--> A = zeros(3,2)

A =

   0.   0.
   0.   0.
   0.   0.
```

This command can be used to create and initialize a matrix.

One can generate the identity matrix with the command `eye` in the following way:

```
--> I = eye(3,3)

I =

  1.   0.   0.
  0.   1.   0.
  0.   0.   1.
```

Particular vectors can be generated with the operator : as it is shown in the following example:

```
--> x = 1:5
```

```
x =

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

This command has created the vector `x` contraining the integer numbers `1` to `5`. A particular increment, other than 1, can be specified in the following way:

```
--> y = 0:%pi/4:%pi

y =

     0.    0.7853982    1.5707963    2.3561945    3.1415927
```

Note that this command always produce a row vector. Of course a negative increment can be used:

```
--> y = 6:-1:1

y =

 6.   5.   4.   3.   2.   1.
```

One can easily create tables by using this command (we will also see in the following chapter that it also allows to prepare data for graphical representation). For example:

```
--> x = (0:0.2:3)';
--> y=exp(-x).*sin(x);
--> [x y]

ans =

   0.    0.
   0.2   0.1626567
   0.4   0.2610349
   0.6   0.3098824
   0.8   0.3223289
   1.    0.3095599
   1.2   0.2807248
   1.4   0.2430089
   1.6   0.2018104
   1.8   0.1609759
   2.    0.1230600
```

```
   2.2   0.0895840
   2.4   0.0612766
   2.6   0.0382881
   2.8   0.0203707
   3.    0.0070260
```

There is also a command allowing to specify only the minimum and maximum value and the number of desired values:

```
--> k = linspace(-%pi,%pi,5)

k =

 - 3.1415927  - 1.5707963   0.   1.5707963   3.1415927
```

3.2 How to manipulate the elements of a matrix

You can reference individual elements of a matrix by specifying the row and column numbers separated by a comma and betwen parenthesis following the name of the matrix. Take the matrix

```
--> A = [1  2  3;  4  5  6;  7  8  9]
```

The following command allows to replace a_{33} by $a_{13} + a_{31}$:

```
--> A(3,3) = A(1,3) + A(3,1)

A =

   1.   2.   3.
   4.   5.   6.
   7.   8.   10.
```

One can also easily extract a row or a column of a matrix; for example the command

```
--> v = A(:,1)

v =

     1.
     4.
     7.
```

assigns to vector `v` the first column of matrix `A`. In the same way, the command

```
--> v = A(1,:)
```

extracts the first row of A.

A row or column index can be a vector of indices. For example if we take the vector

```
--> x = 0:2:8
```

then the following command allows to apply a permutation permutation of elements of x:

```
--> v = [3 5 1 2 4];
--> x(v)

ans =

    4.    8.    0.    2.    6.
```

You can also constuct a matrix by assembling smaller matrices. For example, to add a row to matrix A :

```
--> l = [10  11  12];
--> A = [A;  l]
```

gives

```
   A =

  1.    2.    3.
  4.    5.    6.
  7.    8.    10.
 10.   11.   12.
```

You can also manipulate sub-matrices of a matrix, we already considered this above for the case of a single row or column:

```
--> A(1:2,1:2) = eye(2,2)

A =

   1.    0.    3.
   0.    1.    6.
   7.    8.    10.
   10.   11.   12.
```

Here we have replaced the principal matrix of order 2 by par the identity matrix. You can extract a sub-matrix of arbitrary size. For example

```
--> B = A( 1:2, : )
```

Selects the first two rows of `A` and assigns them to `B`.
To finish with matrices, you can obtain the size of a matrix (number of rows and columns) with the function `size`:

```
--> size(A)

ans =

      4.    3.

--> [n,m]=size(v)

m =

    5.

n =

    1.
```

The first of the above commands returns a vecteur with two components : the first one is the number of rows and the second one is the number of columns. The second command assigns these two values to differents variables, `n` for the rows and `m` for the columns.

Chapter 4

Scripts and functions

The way Scilab has been used in preceding chapters can give the feeling that Scilab is just a “big calculator” devoted to execute commands entered on the keyboard. In fact , Scilab is able to execute a sequence of commands stored in a file, and this way of using it will be interesting as soon as the number of commands will increase ? This also allows to change only a parameter and see the modified result, without having to retype everything.
Scripts and functions are usual textfiles created with Scilab text editor.

- A *script* allows to execute a long sequence of commands.
- A *function* allows to extend the standard library of Scilab function or simply to structurate a program where the same computation is done many times but for different data. The power of Scilab particularly relies on this last feature.

4.1 Scripts

When a script is executed (we will see below how to do it), Scilab interprets the commands as if they were type on the keyboard. It means that the variables created duting execution are variables of the workspace.

Here is an example : we are going to create a script to compute some terms of the well-known Fibonacci sequence whose definition is the following:

$$\left\{\begin{array}{rcl} u_0 & = & 1, \\ u_1 & = & 1, \\ u_{k+2} & = & u_{k+1}+u_k,\ k \geq 0 \end{array}\right.$$

In the first place, you have to create the file which will contain the commands. To this purpose, select `Editor` in the `Applications` menu of Scilab. The window of the editor should be similar to the one depicted in figure 4.1. The created file has the default name `Untitled.sce`. To change it, select the item ”Save as” in the ”File”

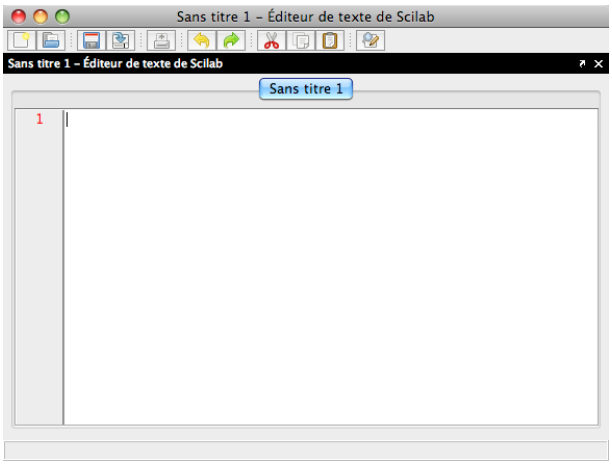

Figure 4.1: Window of the text editor

menu of the editor and choose the name, `fibo.sce`. Now, you can type the following lines in the window of the editor:

```
// A script to compute the n first terms of Fibonacci sequence
n = 10;
u = zeros(n,1);
u(1:2) = [1;1];

for i = 1:n-2
   u(i+2) = u(i+1) + u(i);
end

u
```

The characters `//` allows to precise that the following characters, up to the end of the line, should not be interpreted. This allows to insert comments in the script.

We will se how to use mode precisely the instruction `for` a bit later. Once you have typed the text in the window, save the file (use the "Save" item of the `File` menu of the editor).

To execute the script `fibo.sce`, just type at Scilab's promt:

```
--> exec fibo.sce

u =

    1
    1
    2
```

```
 3
 5
 8
13
21
34
55
```

You can also execute the script from the editor's window by typing `Ctrl-L`.

4.2 Functions

The best way to define a function is to do it within a script. The declaration of a function starts with the keyword `function` and ends with the keyword `endfunction`. A function differs from the *script* we have create just before in the sense that we will be able to give it some input parameter and will be able to return output parameters. The variable which will be defined in the function are *local* variables and won't exist after the function has been called. The only link with the workspace variables is done by passing input parameters to the function.

As an example, we will create a function which does not exist in Scilab, the factorial of an integer. In the editor, create a file named `fact.sci` and type the following text in the window.

```
function [f] = fact(n)

// This function computes the factorial of a integer number
// syntax : variable = fact(n)

if (n - floor(n) ~= 0) | n < 0
   error('error in fact : argument must be a positive integer');
end

if n == 0
   f = 1;
else
   f = prod(1:n);
end

endfunction
```

We give now some explanations:

- The line `if (n - floor(n) ~= 0...` allows to test if the number passed in argument is a positive integer. The construct `if ... end` is rather classical and works like with low level languages like *C*.
- The command `error` allows to stop the execution of the function while displaying an error message for the user.
- The remaining lines show an example of `if ... else ... end` construction.
- The function `prod` computes the product of the elements of the vector argument.

Once you have type the text in the window, save the file. In order that Scilab can be aware of this new function, you just have to execute the script by typing

```
--> exec fact.sci
```

or by typing `Ctrl-L`. An important remark : you can define several functions in the same file. In this case, l'instruction `exec` considers all the function definitions.
You can now use the new function:

```
--> fact(5)

ans =

   120.

--> p = fact(7);
```

You can also define a new function "inline", i.e. directly on the command line of Scilab. This is convenient when the function code is very short:

```
--> deff('c=plus(a,b)','c=a+b');

-->plus(1,2)

ans  =

   3.
```

4.3 Controling execution

The construct `if ... else ... end` is a classical structure to control the execution in a script or a function. We have already given an example just before.

The construct `for ... end` allows to make loops loops. Once again, its use is similar as its use in the `C` language. Its general syntax is the following:

```
for v = expression
   instructions
end
```

or in a more compact form:

```
for v = expression, instructions, end
```

In general `expression` will be something like `m:n` or `m:p:n`, as in the example of the script `fibo.sce` considered before, but if `expression` is a matrix, thenthe variable `v` will be successively assigned each column of this matrix; the following example illustrates this principle:

```
--> A = [1 2 3;4 5 6];
--> for v = A, x = v.^2, end

x =

    1
    16

x =

    4
    25

x =

    9
    36
```

Inlike `for ... end` the construct `while ... end` allows to make loops when the number of iterations is not known *a priori*. For example, here is a function making the euclidian division of two integers:

```
function [quotient,rest] = divEucl(p,q)

rest = p;
quotient = 0;
while rest >= q
   rest = rest - q;
   quotient = quotient + 1;
end

endfunction
```

Type this function within the editor, execute the script, and experiment it, e.g. with the following example :

```
-->[q,r] = divEucl(7,2)
 r  =

    1.
 q  =

    3.
```

Note that when you call a function, you can choose output parameters names as you want, i.e. they can be different from the names which you have used in the definition of the function.

4.4 Interaction with the user

It is possible, in a script, to ask the user to interactively enter some data. For example, in the script *fibo.sce* it could be interesting to ask to the user the value of `n`. This can be done with the function `input`. Modify your script `fibo.sce`: you just have to replace the line

```
n = 10;
```

by the line

```
n = input('What is the value of n ? ');
```

You can experiment your modified script by typing `Ctrl-L` or

```
--> exec fibo.sce
```

You can also generate a menu when the user has to make a choice between several options, with the command `x_choose`, which has to be used in the following way:

```
--> choice = x_choose(['French fries';...
'Spaghetti Bolognese';'kaoya'],'Todays menu')
```

Note that when you find that a line of the script is too long and need more readability of the code, you can end you line by three dots `...` and continue in the following line. The menu should have more or less the following aspect:

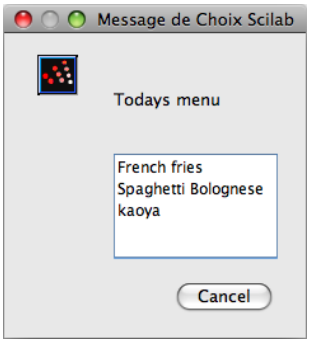

Figure 4.2: Menu with the `x_choose` command

The first argument of `x_choose` is a matrix of character strings containing the text for each option.The second argument is the title of the menu. When you click on one of the options with the mouse cursors, the function returns the number of the corresponding option, or 0 if the user has closed the window ot clicked on the `cancel` button.

Chapter 5

Graphics

This chapter gives a brief aspect of Scilab graphic. It is possible to generate graphics in two and three dimensions, to act on colors, linetypes, and so on.

5.1 Two dimensions graphics

5.1.1 The `plot` command

This command is used to generate graphs in two dimensions. Here is a first example:

```
--> t = 0:%pi/4:2*%pi;
--> plot(t,sin(t))
```

The syntax is rather simple: `plot(x,y)` allows to plot a curve joining the points whose coordinates are given in vectors `x` for the abcsissa and `y` for the ordinates. An important remark : the points are joined by segments, so the more intermediate points the more the graph will be faithfull to the mathematical object, as it is shown in the following example:

```
--> t = 0:%pi/16:2*%pi;
--> plot(t,sin(t))
```

Here we have drawn the graph of a function, but we can also draw a parametric curve of the type

$$\begin{aligned} x &= f(t), \\ y &= g(t), \end{aligned}$$

for $t \in [a,b]$.

For example, here is a circle:

```
--> t = 0:%pi/32:2*%pi;
--> plot(cos(t),sin(t))
--> set(gca(),"isoview","on")
```

Note the command `set(gca(),"isoview","on")` which gives the same scale on both axis.

It is possible to superimpose several curves on the same graph with one single call to `plot`.

```
--> plot(t,cos(t),t,sin(t))
```

You can change change the colors and the line type used: for example if you want a green cosine curve and a red sine curve, just type

```
--> plot(t,cos(t),'g',t,sin(t),'r')
```

You can also draw only disconnected points. In this case each point is represented by a marker (e.g. point, circle, star, cross). For example

```
--> plot(t,cos(t),'g^-',t,sin(t),'ro-')
```

You can modify the color and the type of drawing (line, maker or both) by adding after each `x,y` couple a string composed of at most three characters specifying color, marker and linetype. The table below gives the possible markers and colors.

Symbol	**Color**	**Marker**	**Linetype**
y	yellow	. dot	- plain line
m	magenta	o circle	– dashed line
c	cyan	x cross	-. dashdot line
r	red	+ plus	: dotted line
g	green	* star	
b	blue	d diamond	
w	white	^ triangle	
k	black	v triangle	

When you will have time after the lab, type

```
--> help plot
```

to see all the other possibilities of `plot`.

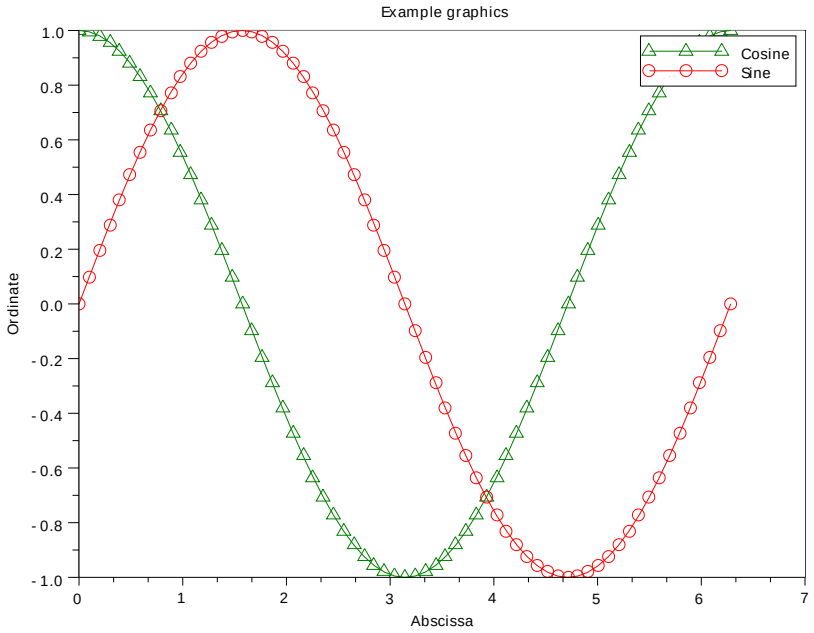

Figure 5.1: Graphics obtained with `plot`

5.1.2 Titles and legends

You can add axis labels and a title to a graph:

```
--> xlabel('Abscissa')
--> ylabel('Ordinate')
--> title('Example graphics)
```

You can add a legend allowing to give the signification of each curve:

```
--> legend('Cosine','Sine')
```

The two strings are given in the same order as the abcissa-ordinate couples in the `plot` which had been used.

5.2 Handling several graphics

The window which as been created when you have called `plot` is window number `0`. If you need a second window, e.g. if you want to compare two graphs, you jus have to type

```
--> figure(1)
```

This command creates a second window with number `1`. You can create as many windows as you need. When you have several windows, you have to activate the window where you want to draw something:

```
--> t = linspace(-%pi,%pi,32);
--> figure(0); plot(t,sin(t))
--> figure(1); plot(t,cos(t))
```

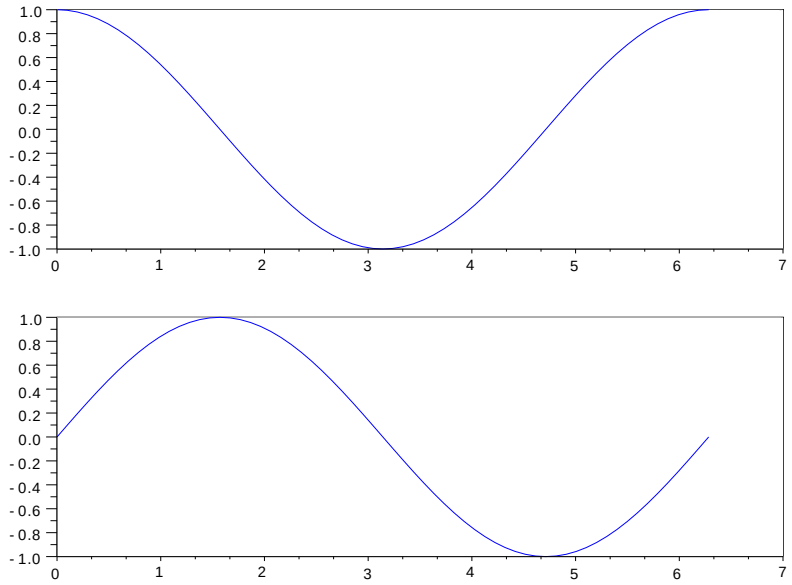

Figure 5.2: Graph using the `subplot` function

A single window can also include several non superimposed graphs by using the `subplot` command, which can be used in the following way:

```
--> subplot(2,1,1)
--> plot(t,cos(t))
--> subplot(2,1,2)
--> plot(t,sin(t))
```

The `subplot(n,m,k)` command allows to divide the window in `n*m` panels, with `n` rows and `m` columns. These panels are numbered form left to right and from top to bottom. The value of `k` allows to specify in which panel the plot is to be done.

5.3 Three dimensions graphics

5.3.1 Curves

You can draw parametric curves with the `plot3` command, for example an helix :

```
--> clf
--> t = 0:%pi/32:8*pi;
--> param3d(cos(t),sin(t),t)
```

The command `clf` allows to clear the current graphic window.

5.3.2 Surfaces

Scilab allows to represent graphs of functions of two variables

$$z = f(x,y)$$

as well as parametric surfaces

$$\begin{aligned} x &= f(t,s), \\ y &= g(t,s), \\ z &= h(t,s), \end{aligned}$$

for $(t,s) \in [a,b] \times [c,d]$.

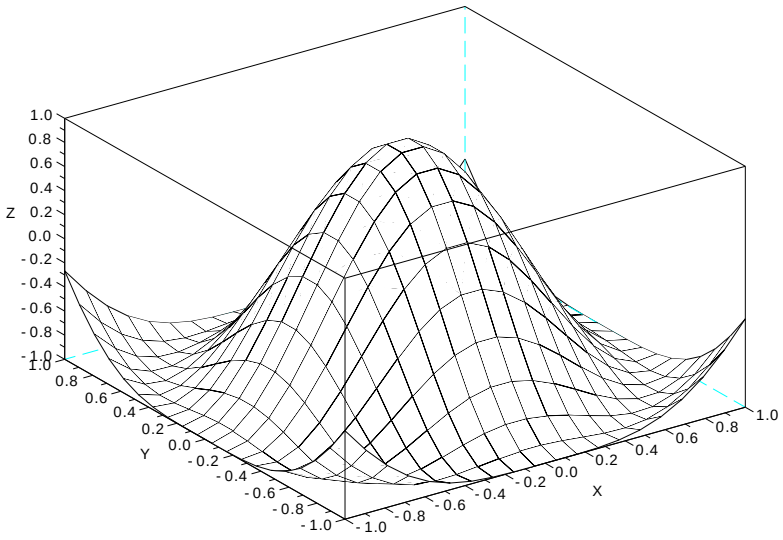

Figure 5.3: Result of the `mesh` command

Here is a first example where we daw the graph of function

$$f(x,y) = \cos\left(\pi\sqrt{x^2+y^2}\right),$$

on the domain $(x,y) \in [-1,1] \times [-2,2]$. As for the two dimensionnal plots we have to choose a sampling step, since the domaine $[-1,1] \times [-2,2]$ has two dimensions. Here we take 20 values in the x direction and 40 values in the y direction.

```
--> x = linspace(-1,1,20);
--> y = linspace(-2,2,40);
```

These two commands allowed us to obtain two vectors $(x_i)_{i=1..20}$ and $(y_j)_{j=1..40}$, but this is not enough because we need the points (x_i, y_j) for $(i, j) \in \{1..20\} \times \{1..40\}$. In other words we need a "matrix of couples" (x_i, y_j) of size 20×40. Scilab is not able to handle such an object but it can give us two matrices, one for the x value and another one for the y values. The command allowing to generate these two matrices is `meshgrid`. For our example, we will use it in the following way :

```
--> [X,Y] = meshgrid(x,y);
```

We can then compute in only one scilab expression all the values of f at points (x_i, y_j) and draw the surface with the command `mesh` :

```
--> Z = cos(%pi*sqrt( X.^2 + Y.^2 ));
--> mesh(X,Y,Z)
```

This command draws a kind of wireframe representation of the surface.

It is possible to have a nicer rendering with the following commands:

```
--> set(gcf(),'color_map',jetcolormap(128))
--> surf(X,Y,Z)
--> set(gce(),'color_flag',3)
```

The command `surf` draws the surface and gives to each point a value (by default equal the z coordinate of the point) which is proportional to the height. The command `set(gcf(),'color_map',jetcolormap(128))` allows to define the table of colors which allows to associate a color to each point of the surface. The surface is discretized in triangles. The command `set(gce(),'color_flag',3)` interpolates the color in each triangle. This allows to have a better "smooth" aspect without having to take many (x,y) points.

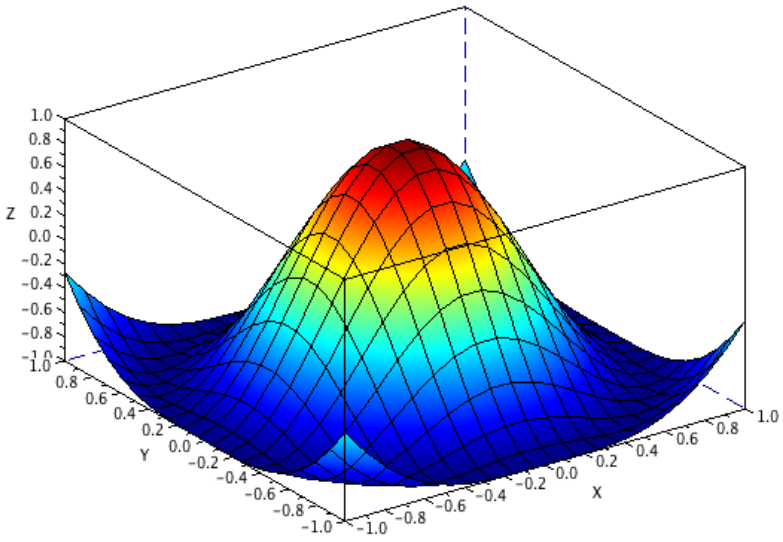

Figure 5.4: Result of the `surf` command

Index