**Open Source Software**

**Introduction to scilab**

Scilab is a programming language associated with a rich collection of numerical algorithms covering many aspects of scientific computing problems. Scilab is an interpreted language. The Scilab language is meant to be extended so that user-defined data types can be defined with possibly overloaded operations. Scilab users can develop their own modules so that they can solve their particular problems. The Scilab language allows to dynamically compile and link other languages such as Fortran and C:

**Features of Scilab**:

Scilab comes with many features.

1. Linear algebra, sparse matrices,
2. Polynomials and rational functions,
3. Interpolation, approximation,
4. Linear, quadratic and non linear optimization,
5. Ordinary Di\_erential Equation solver and Di\_erential Algebraic Equations solver,
6. Classic and robust control, Linear Matrix Inequality optimization,
7. Differentiable and non-differentiable optimization,
8. Signal processing,
9. Statistics.

**Installation of Scilab Windows**

Scilab is distributed as a Windows binary and an installer is provided so that the installation is really easy. On Windows, if your machine is based on an Intel processor, the Intel Math Kernel Library (MKL) enables Scilab to perform faster numerical computations.

**Installation of Scilab Linux**

Under Linux, the binary versions are available from Scilab website as .tar.gz files. There is no need for an installation program with Scilab under Linux: simply unzip the file in one target directory. Once done, the binary \_le is located in <path>/scilab-5.x.x/bin/scilab. When this script is executed, the console immediately appears and looks exactly the same as on Windows.

Scilab is also distributed with the packaging system available with Linux distributions based on Debian (for example, Ubuntu).

**Basic syntax**

**Variable Name**:

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

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.

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.

**Comments and continuation lines**

Any line which begins with two slashes "//" is considered by Scilab as a comment and is ignored. 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.

**Mathematical Operators**

|  |  |
| --- | --- |
| Symbol | Operation |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Right Division (x / y) |
| \ | Left Division (x \ y) |
| ^ | Power (Xy) |
| \*\* | Power (Same as ^) |
| ‘ | Transpose conjugate |

**Data Type**:

In Scilab, everything is a matrix. 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).

**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.

a&b logical and

a|b logical or

~a logical not

a==b true if the two expressions are equal

as=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

**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.

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

exp expm log log10 log1p log2 logm max

maxi min mini modulo pmodulo sign signm sqrt sqrtm

**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 i2 = -1.

**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 [-2n-1; 2n-1 - 1]. An n-bit unsigned integer takes its values from the range [0; 2n-1]. For example, an 8-bit signed integer, as created by the int8 function, can store values in the range [-27; 27-1], which simplifies to [-128; 127].

Scilab integer functions.

y=int8(x) a 8-bit signed integer

y=uint8(x) a 8-bit unsigned integer

y=int16(x) a 16-bit signed integer

y=uint16(x) a 16-bit unsigned integer

y=int32(x) a 32-bit signed integer

y=uint32(x) a 32-bit unsigned integer

**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.

**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 general, the **ans** variable should be used only in an interactive session, in order to progress in the computation without defining a new variable.

-->exp (3)

ans =

20.08553692318766792368

**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

**Matrix**:

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

* the number of rows,
* the number of columns,
* 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.

**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.

**Predefined constants**: Constants are those whose values cannot be changed.

Scilab Description

%i Imaginary unit (*√−*1)

%e Euler’s constant (*e* = 2*.*7182818 *· · ·* )

%pi Ratio of circumference to diameter of a circle; (*π* = 3*.*14159 *· · ·* )

%eps Machine *ϵ* (*≈* 2*.*2 *·* 10*−*16); smallest number such that 1 + *ϵ >* 1

%inf infinity (*∞*)

%nan Not a number

%s Polynomial s=poly(0,'s')

%z Polynomial z=poly(0,'z')

%t Boolean variable: logical true

%f Boolean variable: logical false

%io Two-element vector with file identifiers for standard I/O

List of Scilab Constants

**Built in Arithmetic functions**

Scilab Description

abs(a) Absolute value of a, *|a|*

bool2s Replace %t (or non-zero entry) in matrix by 1 and %f by zero

ceil(a) Round the elements of a to the nearest integers *≥*a

clean “Clean” matrices; i.e. small entries are set to zero

fix(a) Rounds the elements of a to the nearest integers towards zero

floor(a) Rounds the elements of a to the nearest integers *≥* a

gsort(a) Sort elements/rows/columns of a

imag Imaginary part of a matrix

intersect(str1,str2) Returns elements common to two vectors str1 and str2

max Maximum of all elements of a vector or array

maxi Maximum of all elements of a vector or array

mean Mean of all elements of a vector or array

median Median of all elements of a vector or array

min Minimum of all elements of a vector or array

mini Minimum of all elements of a vector or array

prod Product of the elements of a matrix

real Real part of a matrix

round(a) Round the elements of a to the nearest integers

sign(a) Signum function, *a/|a|* for *a ̸*= 0

sqrt(a) *√a*

st deviation Standard deviation

sum Sum of all elements of a matrix

union(a,b) Extract the unique common elements of a and b

unique(a) Return the unique elements of a in ascending order

**Complex numbers**

A complex number is a number which contains a pair of real numbers and it is written in the following manner:

c=a+b⋅i

where:  
*c* –complex number  
*a* – real number  
*b* – real number  
*i* – imaginary unit

In Scilab we define the complex numbers by using the special constant **%i**, in the following manner:

-->c = 2 + 3\*%i

c =

2. + 3.i

-->

**Polynomials**

A polynomial is defined as a **mathematical expression** of determined length which contains operations between variables and constants (coefficients). The allowed operations within a polynomial are additions, subtractions, multiplications and positive integer exponents.

Here is an example of a polynomial:

P(x)=x3–4x2+5x1–2

where:  
*P* – polynomial  
*x* – variable (unknown or indeterminate)  
*4, 5, 2* – constants (coefficients)  
*3, 2, 1* – positive integer exponents

A variable form a mathematical polynomial is not the same thing as a Scilab variable. In fact a Scilab variable defined as a polynomial contains variables (unknowns or indeterminate) and constant numbers.

In Scilab, in order to define a polynomial, we have to use the **poly(arg1,arg2,arg3)** function. With this function we can define a polynomial in two ways:

1. By supplying the roots of the polynomial. In this case the function will be called with these arguments: **poly([roots], arg2, “roots”)**. Example with “x” being the unknown:

-->p=poly([1 2 1],"x","roots")

p =

2   3

- 2 + 5x - 4x + x

-->

2. By supplying the coefficients of the polynomial. In this case the function will be called with these arguments: **poly([coefficients], arg2, “coeffs”)**. Example with “x” being the unknown:

-->p=poly([-2 5 -4 1],"x","coeffs")

p =

2   3

- 2 + 5x - 4x + x

-->

**Vectors**

A **matrix** is an array of values arranged in **rows** and **columns**. If the matrix has only one column or one row it is called a **vector**. In fact Scilab treats also the vectors as matrices, it makes no difference from the manipulation point of view.

A vector is in fact a matrix but only with one row or column. We can define the vector either by inserting the elements of the row:

-->V1=[1 2 3 4]

V1 =

1. 2. 3. 4.

-->

or by inserting the elements of the column using the “;” terminator:

-->V2=[1;2;3;4]

V2 =

1.

2.

3.

4.

-->

Matrix operations (functions like inv(), spec(), zeros(), ones(), eye(), rand(), . Handling these data structures using built in functions.

Programming:

**Functions:**

**Functions** are segments of code that have well defined input and output as well as local variables. The simplest way to define a **function** is by using the command `deff'. **Scilab** allows the creation of in-line **functions** and are especially useful when the body of the **function** is short

**Description**

function <lhs\_arguments>=<function\_name><rhs\_arguments>

<statements>

endfunction

Where

<function\_name>

stands for the name of the function

<rhs\_arguments>

stands for the input argument list. It may be

* a comma separated sequence of variable names enclosed in parenthesis, like (x1,...,xm). Last variable name can be the key word varargin (see [varargin](https://help.scilab.org/doc/5.5.2/en_US/varargin.html))
* the sequence () or nothing, if the function has no input argument.

<lhs\_arguments>

stands for the output argument list. It may be

* a comma separated sequence of variable names enclosed in brackets, like [y1,...,yn]. Last variable name can be the key word varargout (see [varargout](https://help.scilab.org/doc/5.5.2/en_US/varargout.html))
* the sequence [], if the function has no output argument. In this case the syntax may also be: function <function\_name><rhs\_arguments>

<statements>

stands for a set of Scilab instructions (statements). This syntax may be used to define function (see [functions](https://help.scilab.org/doc/5.5.2/en_US/functions.html)) inline or in a script file (see [exec](https://help.scilab.org/doc/5.5.2/en_US/exec.html)). For compatibility with old Scilab versions, functions defined in a script file containing only function definitions can be "loaded" into Scilab using the exec function.

The function <lhs\_arguments>=<function\_name><rhs\_arguments> sequence cannot be split over several lines. This sequence can be followed by statements in the same line if a comma or a semicolon is added at its end.

|  |  |
| --- | --- |
|  | Note that <function\_name> must respect some syntax rules. |

Function definitions can be nested.

**Examples**

*//inline definition (see function)*

function [**x**, **y**]=**myfct**(**a**, **b**)

**x**=**a**+**b**

**y**=**a**-**b**

endfunction

[x,y]=**myfct**(3,2)

*//an one line function definition*

function **y**=**sq**(**x**),**y**=**x**^2,endfunction

**sq**(3)

*//nested functions definition*

function **y**=**foo**(**x**)

a=[sin](https://help.scilab.org/doc/5.5.2/en_US/sin.html)(**x**)

function **y**=**sq**(**x**), **y**=**x**^2,endfunction

y=**sq**(a)+1

endfunction

**foo**(%pi/3)

*// definition in a script file (see exec)*

[exec](https://help.scilab.org/doc/5.5.2/en_US/exec.html) SCI/modules/elementary\_functions/macros/asinh.sci;

Loops(for & while)

WHILE loop syntax:

[Scilab programming - WHILE loop synthax](http://x-engineer.org/wp-content/uploads/2016/08/Scilab-programming-WHILE-loop-synthax.jpg?c4395d&c4395d)

For Loop:

The general syntax of the **for** loop in Scilab is the following:

[Scilab programming - FOR loop syntax](https://x-engineer.org/wp-content/uploads/2016/08/Scilab-programming-FOR-loop-synthax.jpg?c4395d&c4395d)

The **expression** contains a set of Scilab operations which perform:

* variable initialisation
* variable comparison (condition)
* variable incrementation each time the instruction is executed

The **instruction** is the piece of code that is evaluated each time the control variable is incremented as long as the condition within the expression is satisfied.

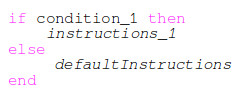
for x=1:1:5

f(x) = x^2 + sqrt(x);

end

Conditional statements ,

The general structure of an IF ELSE statement in Scilab is:

[](https://x-engineer.org/wp-content/uploads/2016/08/Scilab-programming-IF-ELSE-synthax.jpg?c4395d&c4395d)

**Handling .sci files**:

When several commands are to be executed, it may be more convenient to write these statements into a file with Scilab editor. These are called as SCRIPT files. To execute the commands written in such a script file, the exec function can be used, followed by the name of the script file. These file generally have the extension .sce or .sci, depending on its content. Files having the .sci extension contains Scilab functions and/or user defined functions and executing them loads the functions into Scilab environment.

Installation of additional packages e.g. „optimization‟

**Graphics handling: 2D, 3D**

There are numerous Scilab graphics commands. A plots can be graphs of functions, surfaces, curves, or data.

The plot2d command generates two-dimensional graphs. Given vectors

*x* = [*x*1*, . . . , xn*] and *y* = [*y*1*, . . . , yn*]*,*

plot2d(x,y) will plot the points *{*(*x*1*, y*1)*, . . . ,* (*xn, yn*)*}*. There are several options for connecting the points, which can be specified using optional calling arguments. The default in plot2d is to connect them with a straight line. Some of the other most popular alternatives are already coded as plot2d2 , which assumes that the graph is a piecewise constant function, so it looks like a bar graph; plot2d3, which plots a vertical line up to each point; and plot2d4, which connects the plotted points with arrows.

By default, plot2d creates a graphics window called Scilab Graphic (0). This window

can be cleared by the command clf

Scilab can produce many types of 2D and 3D plots. It can create x-y plots with the plot function, contour plots with the contour function, 3D plots with the surf function, histograms with the histplot function and many other types of plots.

In order to get an example of a 3D plot, we can simply type the statement surf() in the Scilab console.

-->surf ()

During the creation of a plot, we use several functions in order to create the data or to configure the plot.

**2D plot Function:**

plot 2D plot

surf 3D plot

contour contour plot

pie pie chart

histplot histogram

bar bar chart

barh horizontal bar chart

hist3d 3D histogram

polarplot plot polar coordinates

Matplot 2D plot of a matrix using colors

Sgrayplot smooth 2D plot of a surface using colors

grayplot 2D plot of a surface using colors

**Generating .jpg files**

# xs2jpg export graphics to JPG

### Calling Sequence

**xs2jpg** ( win\_num, file\_name [, compression\_quality] )

**xs2jpg** ( fig, file\_name [, compression\_quality] )

### Arguments

win\_num: an integer, ID of the figure to export.

Fig: handle of the figure to export.

file\_name a string, name of the exported file.

compression\_quality a real scalar, a number between 0 and 1 (best quality). The default compression quality is set to 0.95.

### Description

xs2jpg exports the display of a graphic window into a JPG file.

### Examples

[scf](https://help.scilab.org/docs/5.5.2/en_US/scf.html)(0);

[plot2d](https://help.scilab.org/docs/5.5.2/en_US/plot2d.html)();

*//JPG export*

xs2jpg(0,'foo.jpg');

xs2jpg([gcf](https://help.scilab.org/docs/5.5.2/en_US/gcf.html)(),'foo.jpg');

xs2jpg(0,'foo\_0.jpg', 0); *// poor quality*

xs2jpg([gcf](https://help.scilab.org/docs/5.5.2/en_US/gcf.html)(),'foo\_1.jpg', 1); *// best quality*

Function plotting,

Data plotting ,

Applications :

Numerical Linear Algebra (Solving linear equations, eigen values atc.) solving

Ordinary Differential Equations,

Numerical Analysis –

iterative methods ,

GUI in scillab,

Plotting 2D graphs,

Comparison with C / C++/ Matlab.