SCILAB: A Short Introduction

MPHYCC-05 unit II (Sem.-II)

In this chapter, we learn about the some feature of SciLab. We present how to install the software and also learn to get help with provided in-line documentation. We learn the important feature of SciLab such as management of real matrices and linear algebra. We also learn how to define the function and management of input and output variables. Moreover, we also learn about SciLab's graphical features such as 2D plots many more.

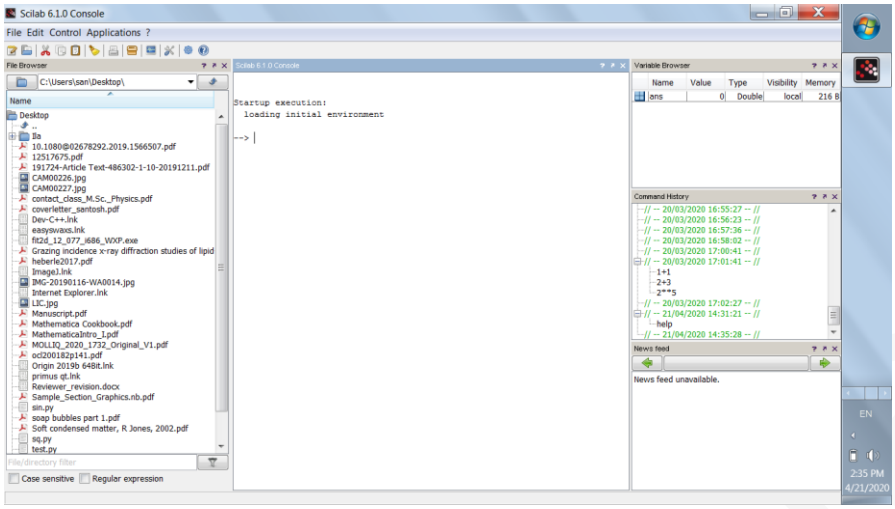
Document in a Glance:

Scilab is a programming language having a collection of numerical algorithms which can handle many aspects of scientific computing problems. Scilab is also an interpreted language which generally allows to-get faster development processes.

Scilab is free software and it is open source software, provided under the Cecill license. Most of the time, the user downloads and installs provides Windows, Linux and Mac OS executable versions. SciLab have the capabilities to solve the various problems belongs to the different flavour such as *linear algebra, Polynomials, Interpolation, Linear & non-linear, ordinary differential equations, signal processing, statistics, and many more.*

1.1: Install SciLab

Scilab binary source provided for 32 and 64 bits both platforms can be downloaded from **http://www.scilab.org/download** for Windows, Linux, or Mac and can be installed easily. Structure of the software will be as shown in the following figure.



1.2: Getting help in SciLab

Open the SciLab software and type *help* in the *console* the press <*Enter*>.

```
help
```

The command result into a console having document about various topic. One can try to browse the integrated help, find the optimization section and then click on the concern item to display its help. Another way to get help about a particular function is help followed by the name of the function which help is required. For example we can get help about plot by typing the following command in the console.

```
help plot
```

Scilab automatically opens the associated entry in the help. There are several ways of using Scilab here we discussed the following three methods:

- using the console in interactive mode
- using the exec function against a file
- using batch processing

1.2.1: Console mode

The first method is typing a command on console to use SciLab interactively. In the following example, the function *disp* is used in interactive mode to print out the string "Hello World! How are you?".

```
-->sp = "Hello World! How are you?"
```

```
sp =
"Hello World! How are you?"
-->disp (s)
sp = "Hello World! How are you?"
```

In the previous session, we did not type the characters "-->" which is the prompt, and which is managed by Scilab. We only type the statement sp = "Hello World! How are you?" with our keyboard and then hit the <Enter> key. Scilab answer is sp = "Hello World! How are you?" Then we type disp(s) and Scilab answer is "Hello World! How are you?"

If you type any command such as below

```
-->disp
```

Then we can type on the <Tab> key, makes a list-box to appear in the console, where items correspond to all functions which begin with the letters "disp". We can then use the up and down arrow keys to select the function we want.

1.2.2: The editor

This editor allows managing several files at the same time. The editor can be accessed from the menu of the console, under the Applications > Editor menu, or from the console, as presented in the following session.

```
--> editor ()
```

The editor provides a fast access to the inline help. Suppose that we have selected the disp statement, when we right-click in the editor, we get the context menu, where the Help about "disp" entry allows to open the help associated with the disp function.

1.2.3: Exec mode

When we need to execute the several commands, it may be more convenient to write these statements into a file with Scilab editor. The commands located in such a file can be executed by using the *exec* function can be used, followed by the name of the script. The script file generally has the extension .sce or .sci, depending on its content:

- The files having the .sci extension are containing Scilab functions and executing them loads the functions into Scilab environment but does not execute them,

- On the other hand the files having the .sce extension are containing both Scilab functions and executable statements. Executing a .sce file has generally an effect such as computing several variables and displaying the results in the console.

Assume that the content of the file test.sce is the following.

```
disp("Hello World! How are you?")
```

In the Scilab console, we can use the exec function to execute the content of this script.

```
-->exec (" test .sce")
-->disp Hello World! How are you?")
Hello World! How are you?
```

Scilab is an interpreted language that it allows to manipulate variables in a very dynamic way. In this section, we learn the basic features of the language; how to create a real variable, and what elementary mathematical functions can be applied to a real variable.

It is worth to state 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.

1.3: Variables

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

1.3.1: Creating real variables

We use "=" to set the variable on the left hand side to its value on the right hand side. The value of the variable is showed each time a statement is executed.

```
-->x=3
x =
    3
-->x = x * 2
x =
    6
```

The semicolon ";" character at the end of the line can suppressed it to display.

-->y=3;
-->y=y*3;
Following are the arithmetic operators and having the usual meaning.

+	Addition	
-	Subtraction	
*	Multiplication	
/	Right division	i.e. $x/y = x\,y^{-1}$
\	Left division	i.e. $x\backslash y = x^{-1}\,y$
^	Power	i.e. $x \hat{} y = x^{y}$
**	Power (same as ^)	
'	Transpose and conjugate	

1.3.2: Variable name

Variable names can be as long as the user wants but it should not me more than 24 characters. All ASCII letters from "a" to "z", from "A" to "Z" and from "0" to "9" are allowed, with the additional letters "%", " ", "#", "!", "$", "?". However, the names which first letter is "%" have a special meaning in Scilab, which presents the mathematical predefined variables. Scilab is case sensitive, thus upper and lower case letters are considered to be different by Scilab. For example:

-->B = 23
B =
23.
-->b = 2
b =
2.
-->B
B =
23.
-->b
b =
2.

1.3.3: Comments and continuation lines

In the SciLab, any line begins with two slashes "//" is considered as a comment and is ignored. However, comment out a block of lines is not possible in SciLab as it possible in C using "/* ……. */".

Further, if the executed statement is too long to express in a single line, then it breaks into many lines which ends with two dots. These dots are considered to be the start of a new continuation line. Followings are the example of comment and continuation lines.

```
-->// This is my comment in SciLab.
-->y =5..
- - >+6..
- - >+7..
```

1.3.4: Basic mathematical function

Some of the basic trigonometry and other mathematical functions are listed below. Most of these functions take one input argument and return one output argument:

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

Below is the example:

```
--> x=cos(5)
 x  =
```

```
  0.2836622
--> y=sin(5)
 y  =
  -0.9589243
--> x^2+y^2
 ans  =
  1.0000000
```

In Scilab, there are several mathematical variables which names are begins with a percent "%" character and are known as predefined variables. Some of them are summarized below.

%i	The imaginary number i
%e	Euler's constant e
%pi	The mathematical constant

13.5: Basic Boolean variable

Boolean are the logical variables and can store true or false values. In Scilab, true is denoted with "%t" or "%T" and false is written with "%f" or "%F". Followings are the list of several comparison operators which are available in Scilab. These operators return Boolean values and take as input arguments all basic data types such as real and complex numbers, integers and strings.

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

We can see the operation of Boolean operator in the following example:

```
--> a= (0==1)
 a  =
  F
```

1.3.6: Complex variable

Complex variables are stored in a pair of float in SciLab. The predefined variable "%i" represents the mathematical imaginary number i with $i^2 = -1$. Further, the character " ' " used to find the complex conjugate, and it will be clear from following example.

```
--> y=2+3*%i
 y  =
   2. + 3.i
--> x=y'
 x  =
   2. - 3.i
--> real(x)
 ans  =
   2.
--> imag(x)
 ans  =
  -3.
--> isreal(y)

ans  =

  F
```

1.3.7: String variable

Strings can be stored in variables such that they are delimited by double quotes "" ". The concatenation operation is available from the "+" operator.

```
--> x="coocoo"
 x  =
  "coocoo"
--> y="bar"
 y  =
  "bar"
--> x+y
 ans  =
  "coocoobar"
```

1.3.8: Dynamic variable

In the Scilab, we create and manage the variable, however, the Scilab allow changing the type of variable dynamically. Moreover, 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. This nature of the Scilab can be understood from the following example.

```
--> y=2
 y  =
   2.
--> y^4
 ans  =
   16.
--> y="Ram"
 y  =
  "Ram"
--> y+"Darvar"
 ans  =
  "RamDarvar"
```

1.4: Matrix

In this section, we learn how to create a various type of matrix, and access the elements of matrix. We also learn about different types of operator associated with matrix. *The components of matrix are; the number of rows, the number of columns and different types of data.*

The data type can be real, Boolean, string and polynomial. In matrix, simplest one is a vector that could be a row or a column matrix. Scalar variable is a matrix with 1 row and 1 column.

Most basic linear algebra operations, such as addition, subtraction, 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. The eigenvalue/eigenvector problems are also performed by optimized sourced code using slash "/"or backslash "\".

1.4.1: Create a matrix

Following syntax is the efficient way to create a matrix:

- Use square brackets [ ]
- Use commas "," to separate the values on different columns,
- Use semicolons ";" separate the values of different rows.

Below is the example to create nxm matrix.

A = [a_{11} , a_{12} , ... , a_{1m}; a_{21} , a_{22} , ... , a_{2m}; ...; a_{n1} , a_{n2} , ... , a_{nm}].

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

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

Matrix comes with different methods some of them are mentioned below.

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

```
--> // Creating empty matrix
--> B=[];
--> B
 B  =
    []
--> // Creating matrix of 1
--> A=ones(2,3)
 A  =
   1.   1.   1.
   1.   1.   1.
```

1.4.2: Query matrix

The following method allows changing and updating the matrix.

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

The size operator returns two out-puts; number of rows **nr** and number of columns **nc**. Further, 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 of matrix A. The syntax 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.*

```
--> A=[1,2,3,4;5,6,7,8;9,10,11,12];
--> [a,b]=size(A)
 a  =
   3.
 b  =
   4.
--> nr=size(A,"r")
 nr  =
   3.
--> nc=size(A,2)
 nc  =
   4.
--> size(A,"*")
 ans  =
   12.
```

1.4.3: Accessing the element of matrix

Element of the matrix can be easily access by using the following methods:

- Element by element with the syntax A (i, j)
- A range of index value with the help of colon (:) operator

```
--> A(1,1)
 ans  =
   1.
--> A(3,4)
 ans  =
   12.
```

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:j,k:l)	the elements at rows from i to j and at columns from k to l
A(i,:)	the row i
A(:,j)	the column j

The operation with ***colon (:)*** is very helpful to access the range of index of the matrix. There are different methods associated with the colon and these are listed in the above list.

```
--> A(1:2,3:4)
 ans  =
   3.   4.
   7.   8.
--> A
 A  =
   1.   2.   3.   4.
   5.   6.   7.   8.
   9.   10.  11.  12.
--> A(:,:)
 ans  =
   1.   2.   3.   4.
   5.   6.   7.   8.
   9.   10.  11.  12.
```

More about colon operator

```
--> //creating a vector
--> v=1:4
 v  =
```

```
   1.   2.   3.   4.
--> //creating a vector with step size 2
--> v1=1:2:10
 v1  =
   1.   3.   5.   7.   9.
--> //creating a vector with step size -3
--> v2=10:-3:1
 v2  =
   10.   7.   4.   1.
--> A=[1,2,3,4;5,6,7,8;9,10,11,12];
--> vr=1:3;
--> vc=2:4;
--> A(vr,vc)
 ans  =
   2.    3.    4.
   6.    7.    8.
   10.   11.   12.
```

The *dollar $* operator allows to get the elements from the end of the matrix. The special operator $ signifies "the index corresponding to the last" row or column, depending on the context. The index $-i corresponds to the index l-i, where l is the number of corresponding rows or columns. The example and the list given below can easily explain the use of the operator.

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

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

1.4.4: Low level operation with matrix

All common algebra operators, such as +, -, * and /, are available with real matrix provided that the matrix size should be compatible for the operation.

```
--> 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.
--> A-B
 ans  =
  -4.  -4.
  -4.  -4.
--> A*B
 ans  =
   19.   22.
   43.   50.
--> A/B
 ans  =
   3.  -2.
   2.  -1.
```

If we use dot (.) before these operation then it will lead to the element wise operation. Details are listed in the following list.

+	Addition	.+	Element wise Addition
-	Subtraction	.-	Element wise Subtraction
*	Multiplication	.*	Element wise Multiplication
/	Right division	./	Element wise Right division
\	Left division	.\	Element wise Left division
^ or **	Power	.^	Element wise Power
'	Transpose and conjugate	.'	Transpose but not conjugate

```
--> A=[1,2;3,4];
--> B=[5,6;7,8];
--> A*B
 ans  =
   19.   22.
   43.   50.
--> A.*B
 ans  =
```

5. 12.
21. 32.

1.4.5: High level operation with matrix

Scilab comes with the complete linear algebra library. However, the list given below can summarize the different method associated with the matrix.

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

1.5: Using looping and branching

In this section we learn about *if*, *for*, *while* statements. We also learn about break and continue statements.

1.5.1: The if statement

The **if** statement executed if the condition is satisfied. The if uses a Boolean variable to do its choice: if the boolean is true, then the statement is executed. A condition is closed with **end** keyword. The syntax starts with *if* followed by a condition and *then* afterwards having some action to do and closed with *end* keyword.

```
i = 2
if ( i == 2 ) then
disp (" Hello! How are you?")
else
disp (" Goodbye !")
end
```

The above script produces “Hi! How are you?”

When there are several combine conditions then *elseif* statement is useful as clear from the given example.

```
i = 2
if ( i == 1 ) then
disp (" Hello! How are you?")
elseif ( i == 2 ) then
disp (" Goodbye! I will see you again")
elseif ( i == 3 ) then
disp (" Nothing to see!")
else
disp ("Nothing to say !")
end
```

However, the *elseif* statement can combine as many conditions as we want but resulting into complicated branching. In this case *select* statement will be useful. It allows performing the statement corresponding to the case keyword. There can be as many branches as required. Example demonstrates the use of select statement.

```
i = 2*3
select i
case 1
disp ("One")
case 2
disp ("Two")
case 6
disp (" Three ")
else
disp (" Other ")
end
```

The above script produces “Three” as expected.

1.5.2: The for statement

The *for* statement allows to perform a given action several times that is looping.

Script

```
for i = 9 : - 2 : 1
 disp(i)
 end
```

output

```
   9.
   7.
   5.
   3.
   1.
```

1.5.3: The while statement

The while statement gets executed till when Boolean expression is true:

Script

```
p = 0
i = 1
while ( i <= 20 )
p = p + i
i = i + 1
end
```

output

```
--> p
 p  =
   210.
--> i
 i  =
   21.
```

1.5.4: The break and continue statements

The *break* statement allows interrupting a loop and used if some condition is satisfied, the loops should not be continued further. However, 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.

Script

```
p = 0
i = 1
while (%t)
   if(i>14) then
      break
      end
```

```
p = p + i
i = i + 1
end
disp(p)
disp(i)
```
output
```
  105.
  15.
```

In the following example, we compute the sum s = 1 + 2 + 4 + 5 + 7+8+10+11 = 48. The modulo (i,3) function returns 0 if the number i is multiple of 3. In this situation, the script increments the value of i and use the continue statement to go on to the next loop.

Script
```
p=0
i = 1
while ( i <= 12)
if ( modulo ( i , 3 ) == 0 ) then
   i=i+1
   continue
else
    p = p + i
    i = i + 1
end
end
```
output
```
--> p
 p  =
   48.
--> i
 i  =
   13.
```

1.6 Function

In this section we learn to create and load the function in Scilab. The sequence to create a function is as follows.

outputvariable = myfunction (inputvariable)

The name of the function is myfunction, inputvariable is the input arguments and outputvariable is the output arguments. y=cos(x) is an example of a function. However,

the number of input and output arguments can be more than one. Syntax for a function having fixed number of arguments is the following:

$$[o_1 , \dots , o_n] = \text{myfunction} (i_1 , \dots , i_n)$$

The input and output arguments are separated by commas ",". The input arguments are surrounded by opening and closing braces, while the output arguments are surrounded by opening and closing square braces.

Function is Scilab is defined with the help of two keywords; function and endfunction as shown in script as below.

```
function y = myfunction ( x )
y =  x**2a
endfunction
```

The possible way to define the function: First way is to write the script on the console in the interactive mode. Another way is to write the script in editor mode and then execute it. There is one more way that is writing the script and then save it to with an extension .sci. The .sci file can be used to load exec function which is clear from the following example.

```
--> exec ("C:\Users\san\Desktop\test1.sci")
--> function y = myfunction ( x )
--> y =  x**2
--> endfunction
--> myfunction(4)
 ans  =
   16.
```

Managing the output arguments: The function can be called in a various way and it is clear from the following script:

```
function [y1, y2]=sant(x1, x2)
y1 = x1*x2
y2 = 3 * (x2**x1)
endfunction

// way to managing the function
--> exec('C:\Users\san\Desktop\test.sce', -1)
--> sant(2,3)
 ans  =
   6.
```

```
--> [y1,y2]=sant(2,3)
 y1  =
   6.
 y2  =
   27.
```

1.7: Plotting

In this section, we learn how to create 2D plots and contour plots. Then we learn to design the title and the legend of our graphics. We use *plot* for 2D plot; *contour* for contour plot, *surf* for 3D plot, *histplot* for histrogram and *bar* for bar plot and may more. In-order to get the help about *plot*, we can write *plot ()* in the console as below.

```
--> plot()
```

Most commonly used plot functions are mentioned below.

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

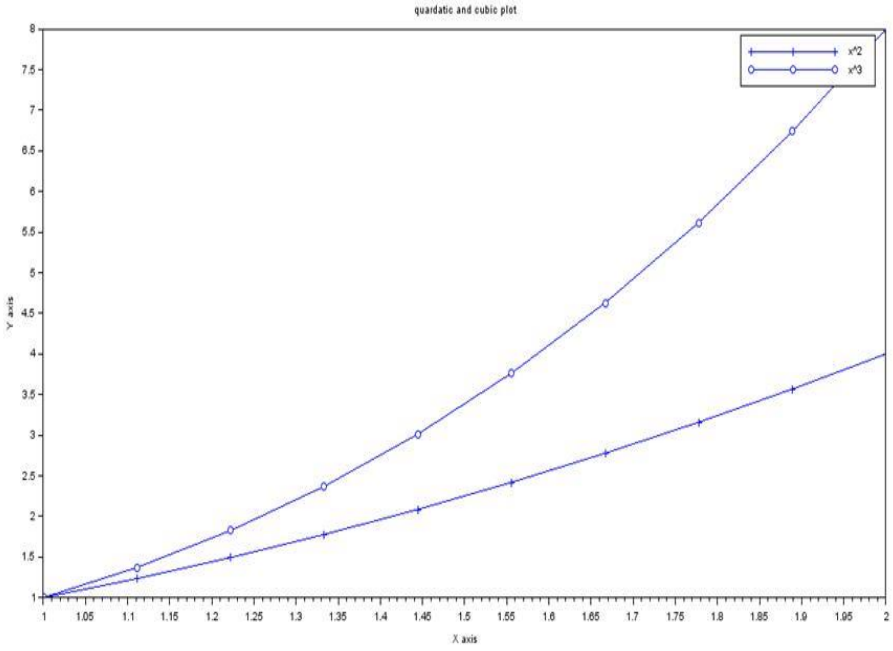
During the creation of a plot, we use several functions in order to create the data or to configure the plot; some of them are listed below.

linspace	linearly spaced vector
feval	evaluates a function on a grid
legend	configure the legend of the current plot
title	configure the title of the current plot
xtitle	configure the title and the legends of the current plot

1.7.1: 2D plotting

First way is to write the script on the console in the interactive mode. Another way is to write the script in editor mode and then execute it. The following script will generate a plot of quadratic and a cubic function.

```
function f1=function1(x)
f1 = x^2
endfunction
function f2=function2(x)
f2 = x^3
endfunction
xdata = linspace ( 1 , 2 , 10 );
ydata1 = function1 (xdata);
plot ( xdata , ydata1 , "+-" )
ydata2 = function2 (xdata);
plot ( xdata , ydata2 , "o-" )
xtitle ( "quardatic and cubic plot" , "X axis " , "Y axis " );
legend ( "x^2" , "x^3" );
```



We can export the file using interactive mode as File>Export to... menu of the graphics window. We can then set the name of the file and its type.