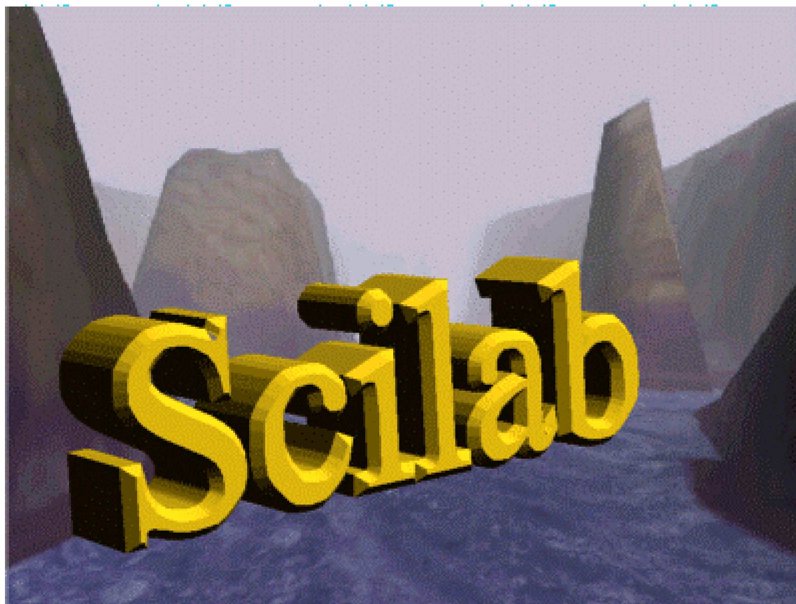
SCILAB FOR DUMMIES

Version 2.6

Written by: K. S. Surendran



Scientific software package for numerical computations in a user-friendly environment

Scilab homepage

For more details please send e-mail to surendran_ks@mailcity.com

To go tutorials, turn page

Contents

Introduction
- What is SCILAB?
- What are the difference between SCILAB and MATLAB?

Getting Started
- How to handle Matrices?
- Entering Matrices
- sum, transpose and diag
- Subscripts
- The Colon Operator

Expressions
- Variables
- Numbers
- Operators
- Expressions

Working with Matrices
- Generating Matrices
- Load
- SCI-Files
- Concatenation
- Deleting Rows and columns

The Command window
- The Format command
- Suppressing Output
- Long Command lines
- Command line editing

Graphics
- Creating a Plot
- Subplots
- Controlling axes
- Axes Labels and Titles
- Printing Graphics

Data Types
- Special Constants
- Matrices of Character Strings
- Polynomials and Polynomial Matrices
- Boolean Matrices
- Lists
- Functions
- Libraries
- Objects

More About Matrices and Arrays
- Linear Algebra
- Arrays
- Multivariate Data
- Scalar Expansion
- Matrix Operations
- The find Function

Flow Control
- if
- select
- for
- while
- break

Introduction

What is SCILAB?

Developed at INRIA, SCILAB has been developed for system control and signal processing applications. It is freely distributed in source code format. The following points will explain the features of SCILAB. It is a similar to MATLAB, which is not freely distributed. It has many features similar to MATLAB.

- Scilab is made of three distinct parts:
  - An interpreter
  - Libraries of functions (Scilab procedures)
  - Libraries of Fortran and C routines.

- SCILAB has an inherent ability to handle matrices (basic matrix manipulation, concatenation, transpose, inverse etc.,)

- Scilab has an open programming environment where the creation of functions and libraries of functions is completely in the hands of the user.
- Scilab has an open programming environment where the creation of functions and libraries of functions is completely in the hands of the user

What are the main differences between Scilab and MATLAB?

Functions

Functions in Scilab are NOT Matlab m-files but variables. One or several functions can be defined in a single file (say myfile.sci). The name of of the file is not necessarily related to the the name of the functions. The name of the function(s) is given by

```
function [y]=fct1(x)
...
function [y]=fct2(x)
```

...

The function(s) are not automatically loaded into Scilab. Usually you have to execute the command getf("myfile.sci") before using it.

Functions can also be defined on-line (or inside functions) by the command deff.

To execute a script file you must use exec("filename") in Scilab and in Matlab you just need to type the name of the file.

Comment lines

Scilab comments begins with: `//`

Matlab comments begins with: `%`

Variables

Predefined variables usually have the `%` prefix in Scilab (`%i, %inf,` ...). They are write protected.

Strings

Strings are considered as 1 by 1 matrices of strings in Scilab. Each entry of a string matrix has its own length.

Boolean variables

Boolean variables are `%T, %F` in Scilab and `0, 1` in Matlab. Indexing with boolean variables may not produce same result. Example `x=[1,2];x([1,1]`) [which is NOT `x([%T,%T]`)] returns [1,1] in Scilab and [1,2] in Matlab. Also if x is a matrix x(1:n,1)=[] or x(:)=[] is not valid in Matlab.

Polynomials

Polynomials and polynomial matrices are defined by the function poly in Scilab (built-in variables). They are considered as vectors of coefficients in Matlab.

Empty matrices

[ ]+1 returns 1 in Scilab and [ ] in Matlab.

Plotting

Except for the simple plot and mesh (plot3d) functions, Scilab and Matlab are not compatible.

Scicos

Scicos (Scilab) and Simulink (Matlab) are not compatible.

A dictionary

Most built in functions are identical in Matlab and Scilab. Some of them have a slightly different syntax. Here is a brief, partial list of commands with significant different syntax.

Matlab Scilab "equivalent"

all and
any or
balance balanc
clock unix('date')
computer unix_g('machine')
cputime timer
delete unix('rm file')
dir unix_g('ls')
echo mode
eig spec or bdiag
eval evstr
exist exists + type
fclose file('close')
feof
ferror
feval evstr and strcat
filter rtitr
finite (x < %inf)
fopen file('open')
fread read
fseek file
ftell
fwrite writeb
global
home
isglobal
isinf(a) a == %inf
isnan(a) a ~= a
isstr(a) type(a) == 10
keyboard pause + resume
lasterr
lookfor apropos
more lines
pack stacksize
pause halt
qz gspec+gschur
randn rand
rem modulo
setstr code2str
strcmp(a,b) a == b
uicontrol
uimenu getvalue
unix unix_g
version
which whereis
nargin [nargout,nargin]=argn(0)
nargout

Getting Started

Starting SCILAB

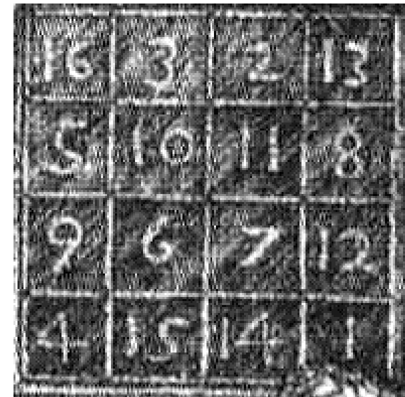
This tutorial is intended to help you start learning SCILAB. It contains a number of examples, so you should run SCILAB and follow along.

To run SCILAB on a PC, double-click on the runscilab icon. To run SCILAB on a UNIX system, type run-Scilab at the operating system prompt. To quit SCILAB at any time, type quit at the SCILAB prompt.

If you feel you need more assistance, type help at the SCILAB prompt, or pull down on the Help menu on a PC.

How to handle Matrices?

The best way for you to get started with SCILAB is to learn how to handle matrices. This section shows you how to do that. In SCILAB, a matrix is a rectangular array of numbers. in this section some simple commands. At the carriage return all the commands typed since the last prompt are interpreted.



A good example matrix, used throughout this tutorial, appears in the Renaissance engraving Melancholia I by the German artist and amateur mathematician Albrecht Dürer. The window in the house has a special importance. The right figure has the zoom of the window. You can see the numbers in the window which form a

peculiar pattern known as "magic square". The sum of any row or column will yield the same result. we will do lot of matrix operations using this magic square matrix.

Entering Matrices

You can enter matrices into MATLAB in several different ways.

- Start by entering Dürer's matrix as a list of its elements. You have only to follow a few basic conventions: Separate the elements of a row with blanks or commas.
- Use a semicolon, ; , to indicate the end of each row.
- Surround the entire list of elements with square brackets, [ ].

To enter Dürer's matrix , simply type,

```
S = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]
```

SCILAB displays the matrix you just entered,

```
S =

! 16. 3. 2. 13. !
! 5. 10. 11. 8. !
! 9. 6. 7. 12.  !
! 4. 15. 14. 1. !
```

This exactly matches the numbers in the engraving. Once you have entered the matrix, it is automatically remembered in the SCILAB workspace. You can refer to it simply as S.

Fine now let us see why the matrix looks so interesting.
Why is it magic power in it?

sum, transpose and diag

You're probably already aware that the special properties of a magic square have to do with the various ways of summing its elements. If you take the sum along any row or column, or along either of the two main diagonals, you will always get the same number. Let's verify that using SCILAB.

The first statement to try is,

```
sum(S,'c')
```

SCILAB replies with,

```
ans =

! 34. !
```

```
! 34. !
! 34. !
! 34. !
```

When you don't specify an output variable, SCILAB uses the variable ans, short for answer, to store the results of a calculation. You have computed a row vector containing the sums of the columns of S. Sure enough, each of the columns has the same sum, the magic sum, 34.

The next statement is also similar to the previous one.

```
sum(S,'r')
```

SCILAB displays

```
ans =

! 34. 34. 34. 34. !
```

The sum of the elements on the main diagonal is easily obtained with the help of the diag function, which picks off that diagonal.

```
diag(S)
```

produces

```
ans =

! 16. !
! 10. !
! 7.  !
! 1.  !
```

You have verified that the matrix in Dürer's engraving is indeed a magic square and, in the process, have sampled a few SCILAB matrix operations. The following sections continue to use this matrix to illustrate additional SCILAB capabilities.

Subscripts

The element in row i and column j of A is denoted by `S(i,j)`. For example, `A(4,2)` is the number in the fourth row and second column. For our magic square, `S(4,2)` is 15. So it is possible to compute the sum of the elements in the fourth column of S by typing

```
S(1,4)  + S(2,4) + S(3,4) + S(4,4)
```

This produces

```
ans =

34.
```

but is not the most elegant way of summing a single column.

It is also possible to refer to the elements of a matrix with a single subscript, S(k). This is the usual way of referencing row and column vectors. But it can also apply to a fully two-dimensional matrix, in which case the array is regarded as one long column vector formed from the columns of the original matrix. So, for our magic square, S(8) is another way of referring to the value 15 stored in S(4,2).

NOTE: If you try to use the value of an element outside the matrix , it is an error

```
temp = S(5,5)
        !--error 21
invalid index
```

On the other hand, if you store a value in an element outside of the matrix, the size increases to accommodate the newcomer:

```
temp =  S;

temp(4,5) =  71
```

This produces

```
temp =

! 16. 3. 2. 13. 0.    !
! 5. 10. 11. 8. 0.    !
! 9. 6. 7. 12. 0.      !
! 4. 15. 14. 1. 71. !
```

The Colon Operator

The colon, `:`, is one of SCILAB's most important operators. It occurs in several different forms. The expression,

```
1:10
```

is the row vector containing the integers from 1 to 10

```
ans =

! 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. !
```

To obtain nonunit spacing, specify an increment. For example

```
10:-2:0
```

is

```
ans  =

!   10.    8.    6.    4.    2.    0. !
```

Subscript expressions involving colons refer to portions of a matrix.

```
S(1:k,j)
```

is the first k elements of the jth column of S.

```
S(1,1:4)
```

produces

```
ans  =

!   16.    3.    2.    13. !
```

Okay, let's come to the magic square

Why is the magic sum for a 4-by-4 square equal to 34? If the integers from 1 to 16 are sorted into four groups with equal sums, that sum must be

```
sum(1:16)/4
```

which, of course, is

```
ans  =

34
```

Expressions

Like most other programming languages, SCILAB provides mathematical expressions, but unlike most programming languages, these expressions involve entire matrices. The building blocks of expressions are

Variables
Numbers
Operators
Functions

Variables

SCILAB does not require any type declarations or dimension statements. When SCILAB encounters a new variable name, it automatically creates the variable and allocates the appropriate amount of storage. If the variable already exists, SCILAB changes its contents and, if necessary, allocates new storage. For example

```
num_integer = 100
```

creates a 1-by-1 matrix named num_integer and stores the value 100 in its single element.

Variable names consist of a letter, followed by any number of letters, digits, or underscores. SCILAB is case sensitive; it distinguishes between uppercase and lowercase letters. S and s are not the same variable. To view the matrix assigned to any variable, simply enter the variable name.

Numbers

SCILAB uses conventional decimal notation, with an optional decimal point and leading plus or minus sign, for numbers. Scientific notation uses the letter e to specify a power-of-ten scale factor. Imaginary numbers use either i or j as a suffix.

Some examples of legal numbers are,

```
3                -199            0.00002
```

```
1.2345667          1.31e-31          1.563e24
0+1*%i               3-2*%i          1.25e5*%i
```

Operators

Expressions use familiar arithmetic operators and precedence rules.

+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Power
'	Complex conjugate transpose
( )	Specify evaluation order

Functions

SCILAB provides a large number of standard elementary mathematical functions, including `abs, sqrt, exp`, and `sin`. Taking the square root or logarithm of a negative number is not an error; the appropriate complex result is produced automatically.

Some of the functions are built-in and are very efficient.

Expressions

You have already seen several examples of SCILAB expressions. Here are a few more examples, and the resulting values.

```
rho = (1+sqrt(5))/2
rho =
    1.6180
a = abs(3+4*%i)
a =
    5
```

Working with Matrices

This section introduces you to other ways of creating matrices. The SCILAB is most powerful while handling matrices it allows you to manipulate the matrix as a whole.

Generating Matrices

SCILAB provides three functions that generate basic matrices:

`zeros`	All zeros
`ones`	All ones
`rand`	Random elements (either normal or uniform)

Some examples

```
zeros(3,3)

ans =

! 0. 0. 0. !
! 0. 0. 0. !
! 0. 0. 0. !

4*ones(3,3)

ans =

! 4. 4. 4. !
! 4. 4. 4. !
! 4. 4. 4. !

rand(4,4,'normal')

ans =

! 1.4739763   .2546697  -1.0327357   .0698768   !
!  .8529775  -.6834217   -.9239258  - 1.3772844 !
!  .7223316   .8145126   2.7266682  -.1728369   !
!  .6380837  -.1884803  -1.7086773  -.6019869   !
```

Load

The load command reads binary files containing matrices generated by earlier SCILAB sessions, or reads text files containing numeric data. The text file should be organized as a rectangular table of numbers, separated by blanks, with one row per line, and an equal number of elements in each row. For example, outside of SCILAB, create a text file containing these four lines:

```
magic =

  ! 16.0     3.0     2.0     13.0 !
  !  5.0    10.0    11.0      8.0  !
  !  9.0     6.0     7.0     12.0 !
  !  4.0    15.0    14.0      1.0  !
```

Store the file under the name `magic_square.dat`.

```
save magic_square
```

Then the command

```
load magic_square.dat
```

Reads the file and creates a variable, magic, containing our example matrix.

SCI-Files

You can create your own matrices using sci-files, which are text files containing SCILAB code. Just create a file containing the same statements you would type at the LAB command line. Save the file under a name that ends in .sci.

Note: To write a sci-file open a textpad or notepad and write the code in the text file. Then save the file with the extension <filename.sci>. On the command window go the file control button and click the exec option and choose the file you want to execute.

For example,

A sci-file which will plot a sine wave, (use notepad to write the code)

```
// this is comment line
// sci-file to plot sine wave
time = 0:.01:20;
plot(sin(time));
```

save this file as `My_prog.sci` and run the program as mentioned above. You can also run the program in command window by typing,

```
exec('Pathname')
```

i.e.,

```
exec('E:\Scilab-2.6\work\My_prog.sci')
```

On executing the program you will see the Sine wave in the figure window as shown on the next page.

Concatenation

Concatenation is the process of joining small matrices to make bigger ones. In fact, you made your first matrix by concatenating its individual elements. The pair of square brackets, [], is the concatenation operator.

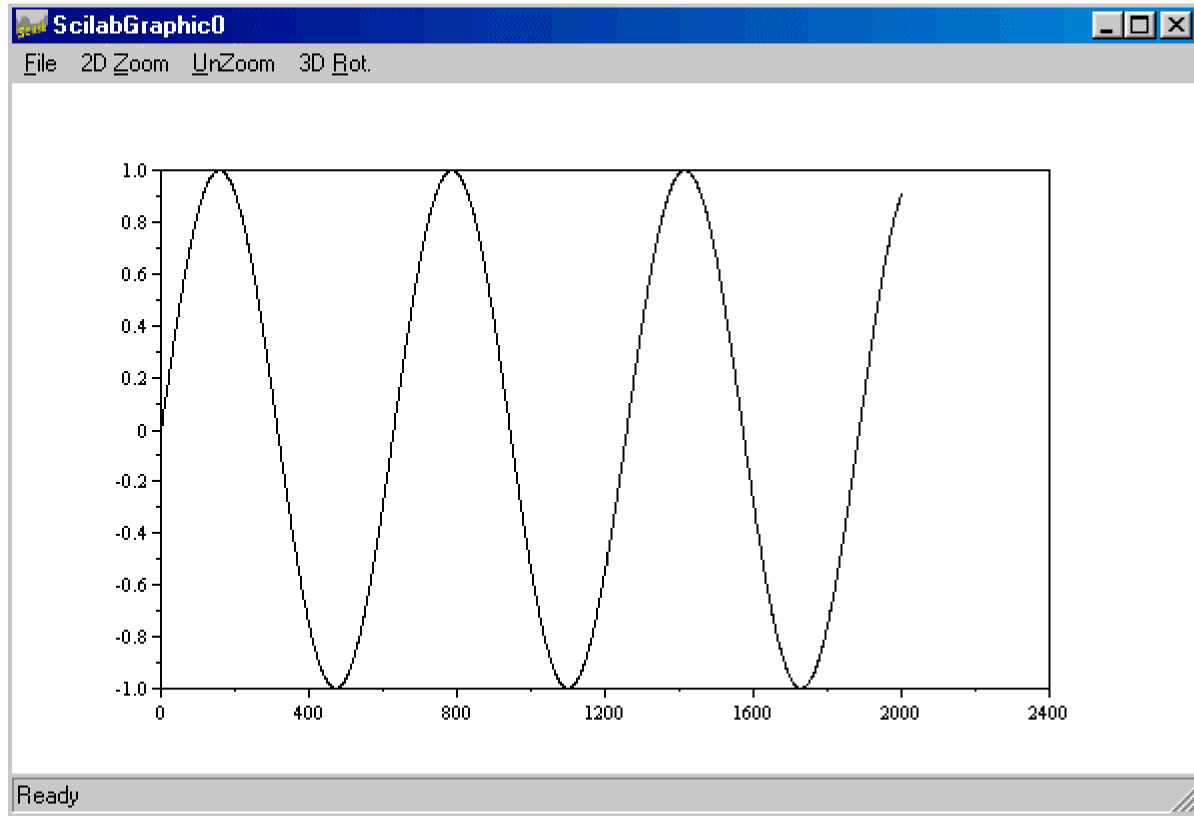
For example,

```
a = [ 1 2 3 ]; b= [ 4 5 6]; c = [ 7 8 9];
d= [ a b c]
    d =
    ! 1. 2. 3. 4. 5. 6. 7. 8. 9. !
```

Deleting Rows and Columns

You can delete rows or columns from a matrix by using just a pair of square brackets.

For example,




```
s = [ 1 2 3 4; 5 6 7 8; 9 10 11 12 ]
s =

! 1.   2.   3.   4. !
! 5.   6.   7.   8. !
! 9.  10.  11.  12. !

s(:,2) =[]
```

produces

```
s =

! 1.   3.   4. !
! 5.   7.   8. !
! 9.  11.  12. !
```

If you delete a single element from a matrix, the result isn't a matrix anymore.

So if you type expressions like ,

```
s(1,3) =[]

 !--error 15
submatrix incorrectly defined
```

will result in error.

However, using a single subscript deletes a single element, or sequence of elements, and reshapes the remaining elements into a column vector. So,

```
s(2:2:9) = [ ]
```

results in

```
s  =

!   1.  !
!   9.  !
!   6.  !
!   3.  !
!   11. !
!   4.  !
!   8.  !
!   12. !
```

The Command window

So far, you have been using the SCILAB command line, typing commands and expressions, and seeing the results printed in the command window. This section describes a few ways of altering the appearance of the command window. It is better to use fixed width font such as fixedays or courier to provide proper spacing.

The format Command

The format command controls the numeric format of the values displayed by SCILAB. The command affects only how numbers are displayed, not how SCILAB computes or saves them. Here are the different formats, together with the resulting output produced from a vector s with components of different magnitudes.

```
s = [ 10/3  1.234567e-6]

format('v',10);s

s =

! 3.3333333 .0000012 !

format(20);s

s =

! 3.3333333333333335 .0000012345670000 !

format('e',10)s

s =

! 3.333E+00 1.235E-06 !
```

Suppressing Output

If you simply type a statement and press Return or Enter, SCILAB automatically displays the results on screen. However, if you end the line with a semicolon, SCILAB performs the computation but does not display any output. This is particularly useful when you generate large matrices.

For example,

```
s = rand(1,100);
```

Long Command Lines

If a statement does not fit on one line, use three periods, `...`, followed by Return or Enter to indicate that the statement continues on the next line. For example

```
s = 1 -1/2 + 1/3 -1/4 + 1/5 - 1/6 + 1/7 ...
- 1/8 + 1/9 - 1/10 + 1/11 - 1/12;
```

Blank spaces around the =, +, and - signs are optional, but they improve readability

Command Line Editing

Various arrow and control keys on your keyboard allow you to recall, edit, and reuse commands you have typed earlier. For example, suppose you mistakenly enter

```
rho = (1 + sqt(5))/2
```

You have misspelled sqrt. SCILAB responds with

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

Instead of retyping the entire line, simply press the key. The misspelled command is redisplayed. Use the key to move the cursor over and insert the missing r. Repeated use of the key recalls earlier lines. Typing a few characters and then the key finds a previous line that begins with those characters.

The list of available command line editing keys is different on different computers.

```
ctrl-p
Recall previous line

ctrl-n
Recall next line

ctrl-b
Move back one character

ctrl-f
Move forward one character

home   ctrl-a
Move to beginning of line

end    ctrl-e
Move to end of line

esc    ctrl-u
```

```
Clear line

del    ctrl-d
Delete character at cursor

Backspace        ctrl-h
Delete character before cursor

ctrl-k
Delete to end of line
```

Graphics

SCILAB has extensive facilities for displaying vectors and matrices as graphs, as well as annotating and printing these graphs. This section describes a few of the most important graphics functions and provides examples of some typical applications.

It is possible to use several graphics windows ScilabGraphicx x being the number used for the management of the windows, but at any time only one window is active. On the main SCILAB window the button Graphic Window x is used to manage the windows: x denotes the number of the active window, and we can set (create), raise or delete the window numbered x: in particular we can directly create the graphics window numbered 10. The execution of a plotting command automatically creates a window if necessary.

There are 4 buttons on the graphics window:

- `3D Rot.`: for applying a rotation with the mouse to a 3D plot. This button is inhibited for a 2D plot. For the help of manipulations (rotation with specific angles ...) the rotation angles are given at the top of the window.
- `2D Zoom`: zooming on a 2D plot. This command can be recursively invoked. For a 3D plot this button is not inhibited but it has no effect.
- `UnZoom`: return to the initial plot (not to the plot corresponding to the previous zoom in case of multiple zooms).

These 3 buttons affecting the plot in the window are not always in use; we will see later that there are different choices for the underlying device and zoom and rotation need the record of the plotting commands which is one of the possible choices (this is the default).

- `File:` this button opens different commands and menus.

  The first one is simple : `Clear` simply rubs out the window (without affecting the graphics context of the window).

  The command `Print`... opens a selection panel for printing. Under Unix, the printers are defined in the main scilab script `SCIDIR/bin/scilab` (obtained by ``make all'' from the origin file `SCIDIR/bin/scilab.g`).

  The `Export` command opens a panel selection for getting a copy of the plot on a file with a specified format (Postscript, Postscript-Latex, Xfig).

  The `save` command directly saves the plot on a file with a specified name. This file can be loaded later in Scilab for replotting.

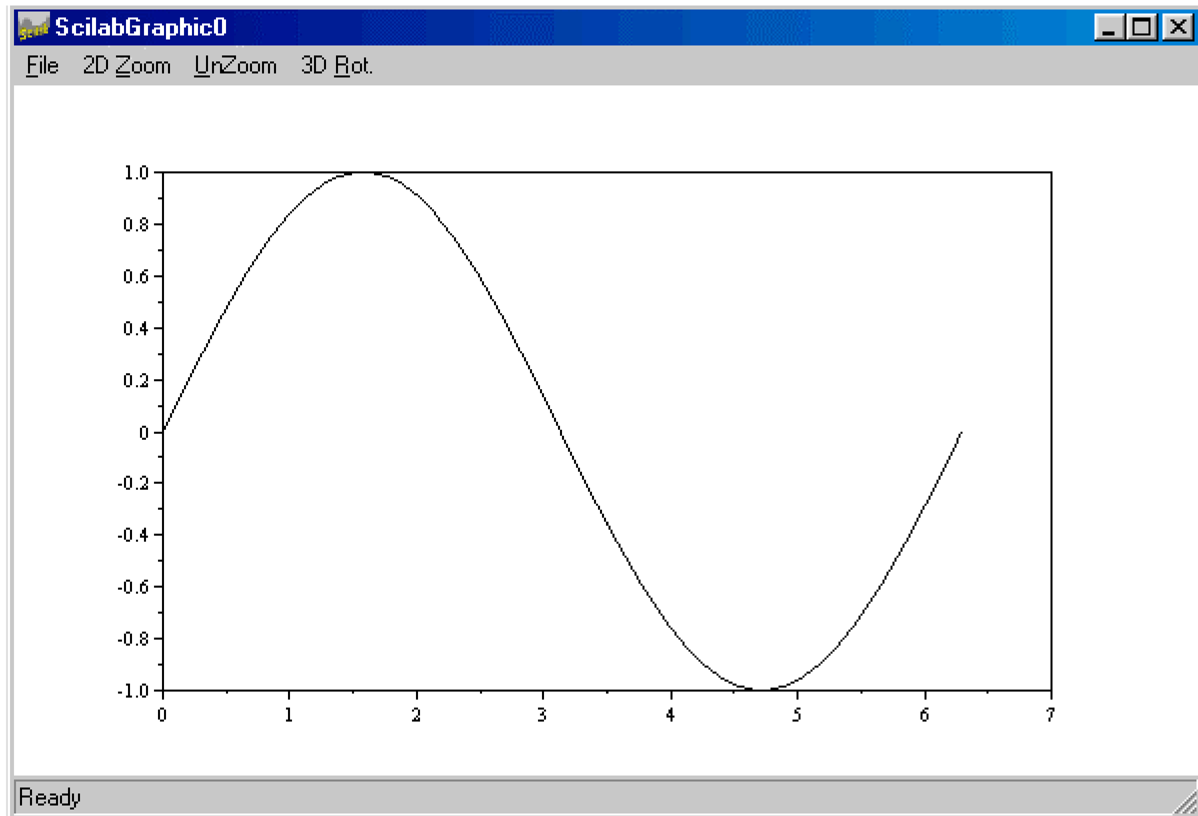
  The `Close` is the same command than the previous `Delete Graphic Window` of the menu of the main window, but simply applied to its window (the graphic context is, of course deleted).

Creating a Plot

The `plot` function has different forms, depending on the input arguments. If `y` is a vector, `plot(y)` produces a piecewise linear graph of the elements of `y` versus the index of the elements of `y`. If you specify two vectors as arguments, `plot(x,y)` produces a graph of `y` versus `x`.

For example, to plot the value of the sine function from zero to 2, use

```
t  =  0:%pi/100:2*%pi;
y = sin(t);
plot(t,y);
```



Multiple `x-y` pairs create multiple graphs with a single call to `plot`. SCILAB automatically cycles through a predefined (but user settable) list of colors to allow discrimination between each set of data. For example, these statements plot two related functions of `t`, each curve in a separate distinguishing color:

For example,

```
plot([sin(t);cos(t)]);
```

produces the plot shown on the next page,

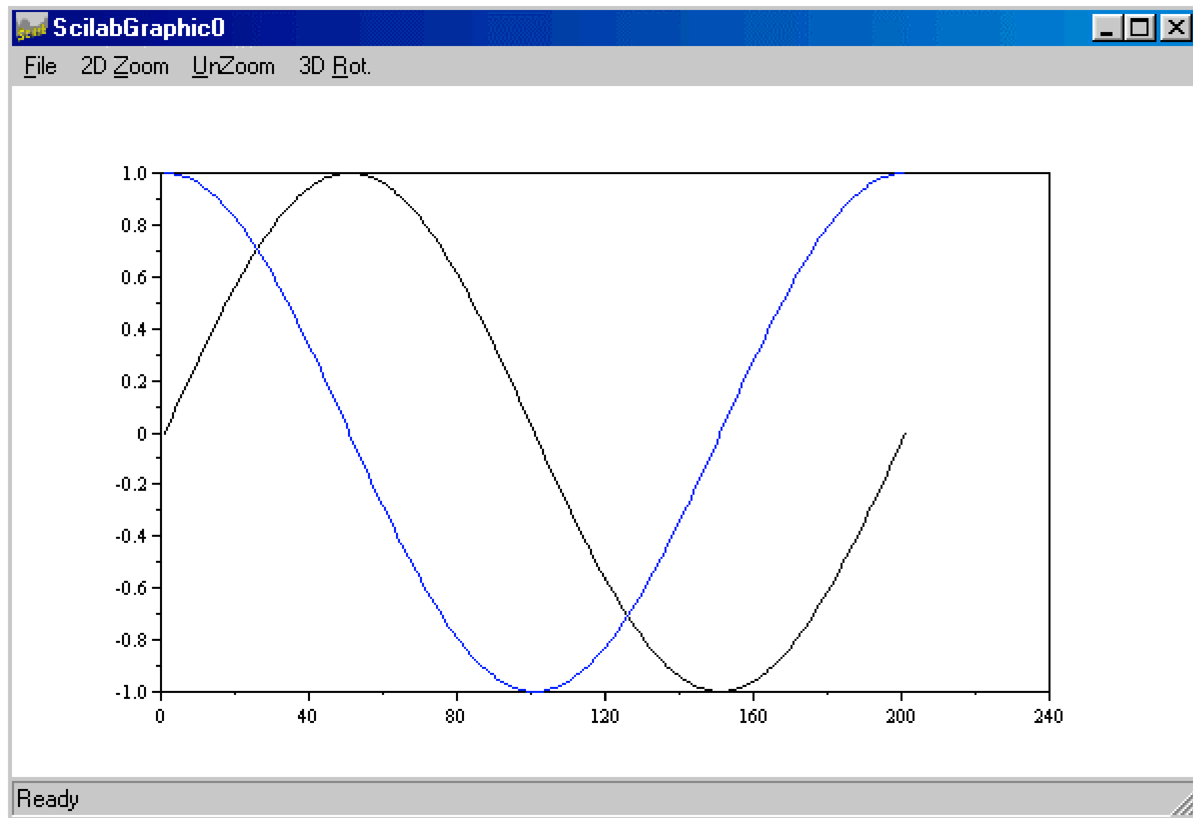
Subplots

The subplot function allows you to display multiple plots in the same window or print them on the same piece of paper.

Typing,

```
subplot(m,n,p)
subplot(mnp)
```

breaks the figure window into an m-by-n matrix of small subplots and selects the pth subplot for the current plot. The plots are numbered along first the top row of the figure window, then the second row, and so on. For example, to plot data in four different subregions of the figure window,




```
subplot(221)
plot2d()
subplot(222)
plot3d()
subplot(2,2,3)
param3d()
subplot(2,2,4)
 hist3d()
```

produces the plot shown on the next page,

Controlling Axes

Ordinarily, SCILAB finds the maxima and minima of the data and chooses an appropriate plot box and axes labeling. The axis function overrides the default by setting custom axis limits,

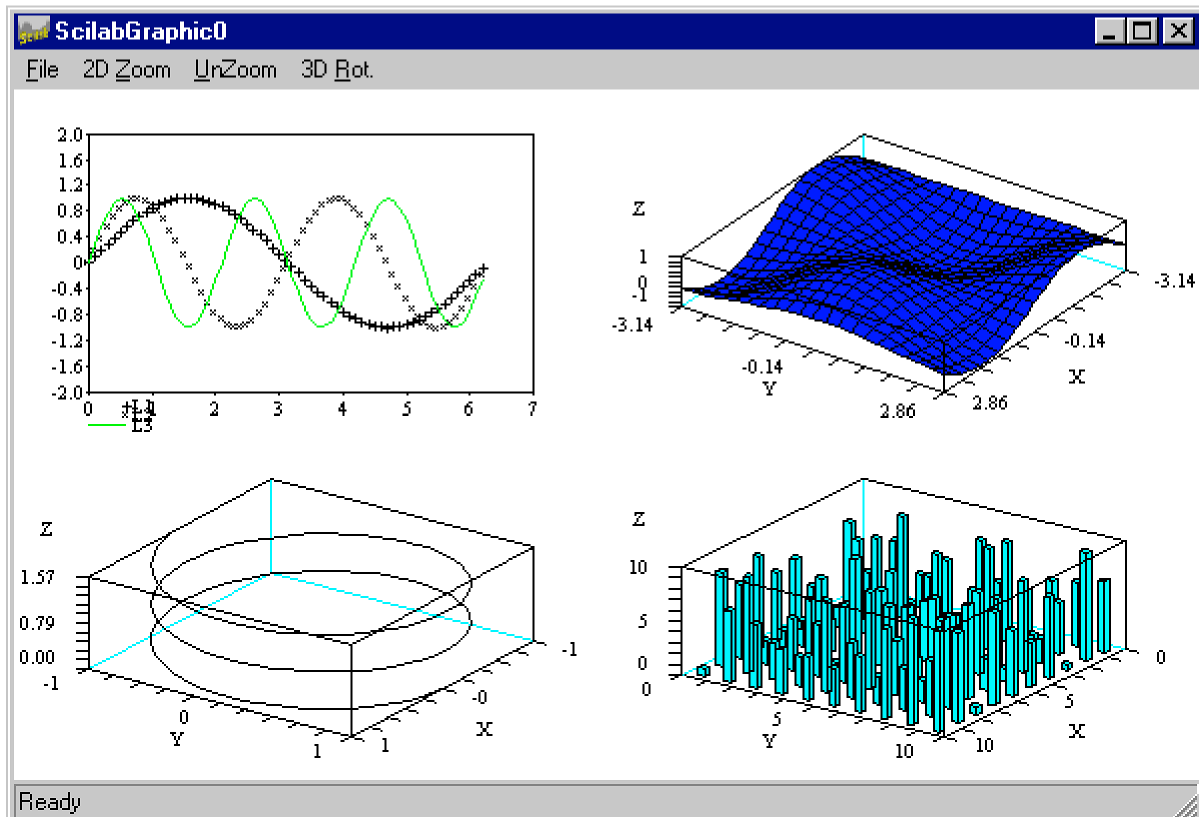
```
square(xmin xmax ymin ymax)
```

The requested values xmin, xmax, ymin, ymax are the boundaries of the graphics frame and square changes the graphics window dimensions in order to have an isometric plot.

By typing,

```
xset("default")
```

The original default size will be used for the graphic windows.



Axes Labels and Titles

The x-axis and y-axis labels and caption (Title) of the plot can be given in the plot function itself. typing

```
plot(x,y,[xcap,ycap,caption])
```

For example,

```
x=0:0.1:2*%pi;
// simple plot
plot(sin(x))
// using captions
xbasc()
plot(x,sin(x),"sin","time","plot of sinus")
```

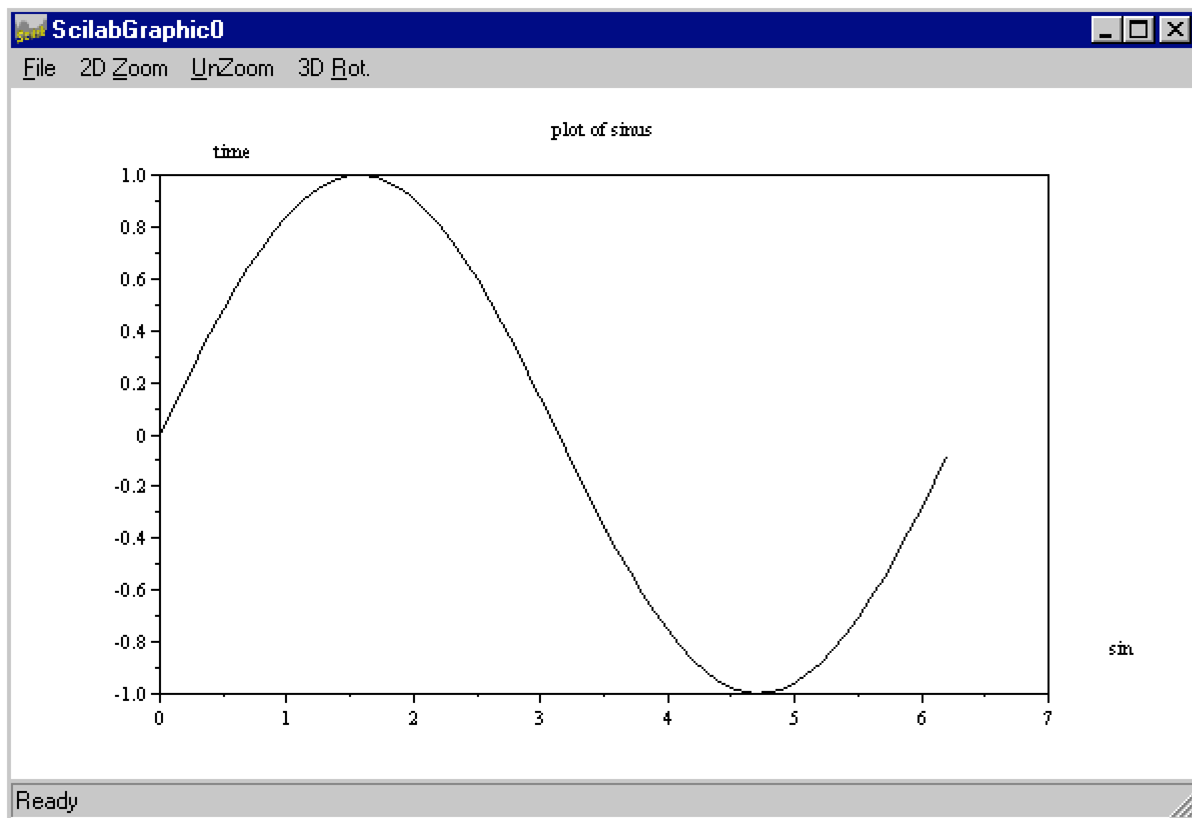
This produces the graphic shown on the next page,

Printing Graphics

The Print option on the File menu and the print command both print MATLAB figures. The Print menu brings up a dialog box that lets you to print the figure.

- Window to Paper

  The simplest command to get a paper copy of a plot is to click on the **print** button of the SCILAB graphic window.



- Creating a Postscript File

  The simplest way to get a Postscript file containing SCILAB plot is :

```
driver('Pos')          // selects a graphics driver
xinit('foo.ps')        // initialization of a graphics driver
plot3d1();             // demo of plot3d1
xend()                 // closes  graphics session
driver('Rec')
plot3d1()
```

```
xbasimp(0,'foo1.ps')    //send graphics to a Postscript printer or
                          in a file
```

The Postscript files (foo.ps or foo1.ps) generated by SCILAB cannot be directly sent to a Postscript printer, they need a preamble. Therefore, printing is done through the use of Unix scripts or programs which are provided with SCILAB. The program Blpr is used to print a set of SCILAB

Graphics on a single sheet of paper and is used as follows :

```
Blpr string-title file1.ps file2.ps > result
```

You can then print the file result with the classical Unix command :

```
lpr -Pprinter-name result
```

or use the **ghostview** Postscript interpreter on your Unix workstation to see the result.

Data Types

SCILAB recognizes several data types. Scalar objects are constants, Booleans, polynomials, strings and rationals (quotients of polynomials). These objects in turn allow to define matrices which admit these scalars as entries. Other basic objects are lists, typed-lists and functions. Only constant and Boolean sparse matrices are defined. The objective of this chapter is to describe the use of each of these data types.

Special Constants

SCILAB provides special constants similar to that of MATLAB. In general, these constants have % before them. These variables are considered as "predefined". They are protected, cannot be deleted and are not saved by the save command. It is possible for a user to have his own "predefined" variables by using the `predef` command.

The table lists the special constants and their functions,

%i	represents sqrt(-1)
%e	trigonometric constant e = 2.7182818
%nan	not a number
%s	is the polynomial s=poly(0,'s') with symbol s.
%f	Boolean constant which stand for false and %f is the same as ~%t.

%pi	P = 3.1415927 .....
%eps	constant representing the precision of the machine
%inf	infinity
%t	Boolean constant which stand for true and %t is the same as 1==1

Matrices of Character Strings

Character strings can be created by using single or double quotes. Concatenation of strings is performed by the + operation. Matrices of character strings are constructed as ordinary matrices, e.g. using brackets. A very important feature of matrices of character strings is the capacity to manipulate and create functions.

Furthermore, symbolic manipulation of mathematical objects can be implemented using matrices of character strings. The following illustrates some of these features.

```
s =['x' 'y';'z' 'w+v']
```

produces

```
s =

!   x    y   !
!            !
!  z    w+v  !
```

and

```
ss =trianfml(s)
```

produces

```
! z   w+v           !
!                   !
! 0   z*y-x*(w+v)   !
```

Substituting the value for x, y, z, and w

```
x=1;y=2;z=3;w=4;v=5;
```

and

```
evstr(ss)
```

This produces

```
ans =

! 3.    9.  !
! 0.  - 3.  !
```

Polynomials and Polynomial Matrices

Polynomials are easily created and manipulated in SCILAB. Manipulation of polynomial matrices is essentially identical to that of constant matrices. The poly primitive in SCILAB can be used to specify the coefficients of a polynomial or the roots of a polynomial.

```
p=poly([1 2],'s')      //polynomial defined by its roots
```

produces,

```
p =

            2
2 - 3s + s
```

and,

```
q=poly([1 2],'s','c')  //polynomial defined by its coefficients
```

This produces,

```
q =
```

```
1 + 2s
```

For example,

```
q/p
```

produces,

```
ans =
  1 + 2s
----------
           2
  2 - 3s + s
```

Boolean Matrices

Boolean constants are %t and %f. They can be used in Boolean matrices. The syntax is the same as for ordinary matrices i.e. they can be concatenated, transposed, etc... Operations symbols used with Boolean matrices or used to create Boolean matrices are == and ~.

If B is a matrix of Booleans or(B) and and(B) perform the logical or and and.

For example, typing

```
%t
```

produces,

```
%t =
T
```

Similarly,

```
[ 3,3] == [3,4]
```

This produces

```
ans =
! T F !
```

and

```
s = 1:6 ; s(s>3)
```

will display,

```
ans =
! 4. 5. 6. !
```

Similarly,

```
A = [%t  %f  %t  %f], B = [%f  %t  %f  %t]
```

produces,

```
A =
! T F T F !
B =
! F T F T !
```

and

```
A|B          // logical OR
ans =
! T T T T !
A&B          // logical AND
ans =
! F F F F !
```

Lists

SCILAB has a list data type. The list is a collection of data objects not necessarily of the same type. A list can contain any of the already discussed data types (including functions) as well as other lists. Lists are useful for defining structured data objects. There are two kinds of lists, ordinary lists and typed-lists. A list is defined by the list function.

Here is a simple example:

```
ls  = list(2,%i,'f',ones(3,3))    // a list made of four entires
```

This produces,

```
ls =
ls(1)
     2.
ls(2)
     i
ls(3)
     f
ls(4)
     ! 1. 1. 1. !
     ! 1. 1. 1. !
     ! 1. 1. 1. !
```

To extract the a entry from a list you have to use listname(listindex), for example,

als(4)

```
ans =

      ! 1. 1. 1. !
      ! 1. 1. 1. !
      ! 1. 1. 1. !
```

You can also create a nested list.

```
ls(2) =  list( %t, rand(2,2,'normal'))    // ls(2) is now a list
ls(2)(1)
      T
ls(2)(2)
      ! .6380837 - .6834217 !
      ! .2546697 .8145126   !
```

Typed lists have a specific first entry. This first entry must be a character string (the type) or a vector of character string (the first component is then the type, and the following elements the names given to the entries of the list). The general format is,

```
tlist(typ,a1,....an )
```

where `typ` argument specifies the list type. and `a1...an` is the the object.

Typed lists entries can be manipulated by using character strings (the names) as shown below.

```
lst  =  tlist(['random numbers';'Name';'Example'], ' Uni-
form',rand(3,3,'uniform'))
```

This produces,

```
lst(1)
      !random numbers !
      !                        !
      !Name                 !
      !                        !
      !Example             !
lst(2)
Uniform
lst(3)
      ! .2113249 .3303271 .8497452 !
      ! .7560439 .6653811 .6857310 !
      ! .0002211 .6283918 .8782165 !
```

And,

```
lst('Name')          // same as lst(2)
ans =
```

```
Uniform
```

Functions

Functions are collections of commands which are executed in a new environment thus isolating function variables from the original environments variables. Functions can be created and executed in a number of different ways. Furthermore, functions can pass arguments, have programming features such as conditionals and loops, and can be recursively called. Functions can be arguments to other functions and can be elements in lists. The most useful way of creating functions is by using a text editor, however, functions can be created directly in the SCILAB environment using the deff primitive.

Let us workout a simple function in the command window. the function will convert the input into dB.

```
deff('[out] = dB(inp)',' out = 10*log10(inp)')
```

Let us try with some value,

```
db(10)
```

produces,

```
ans =

    10.
```

Usually functions are defined in a file using an editor and loaded into SCILAB with `getf('filename')`. This can be done also by clicking in the File operation button. This latter syntax loads the `function(s)` in filename and compiles them.

The first line of filename must be as follows:

```
function [y1,...,yn]=macname(x1,...,xk)
```

where the yi's are output variables and the xi's the input variables.

Libraries

Libraries are collections of functions which can be either automatically loaded into the SCILAB environment when SCILAB is called, or loaded when desired by the user. Libraries are created by the lib command. Examples of libraries are given in the **SCIDIR/macros** directory.

Note that in these directory there is an ASCII file "**names**" which contains the names of each function of the library, a set of .sci files which contains the source code of the functions and a set of .bin files which contains the compiled code of the functions. The **Makefile** invokes SCILAB for compiling the functions and generating the .bin files. The compiled functions of a library are automatically loaded into SCILAB at their first call.

Objects

SCILAB objects can be viewed by using the function typeof. The general format is,

```
typeof(object)
```

For example,

```
d = 'suren';
typeof(d)
ans =
       string
```

The following table contains the list of SCILAB objects,

Name	Description
usual	for matrices with real or complex entries.
boolean	for boolean matrices.
function	for functions.
state-space	for linear systems in state-space form (syslin lists).
boolean sparse	for sparse boolean matrices.
state-space	(or rational) for syslin lists.

Name	Description
polynomial	for polynomial matrices: coefficients can be real or complex.
character	for matrices of character strings.
rational	for rational matrices (syslin lists)
sparse	for sparse constant matrices (real or complex)
list	for ordinary lists.
library	for library definition.

More About Matrices and Arrays

This sections shows you more about working with matrices and arrays, focusing on

Linear Algebra
Arrays
Multivariate Data
Indexing in Matrices and Lists

Linear Algebra

Informally, the terms ***matrix*** and ***array*** are often used interchangeably. More precisely, a ***matrix*** is a two-dimensional numeric array that represents ***a linear transformation***. The mathematical operations defined on matrices are the subject of ***linear algebra***.

We have discussed about the basics of the matrices earlier itself. we will take the same example for this section also. Let us use the magic square matrix,\

```
S = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]
```

Adding the transpose to a matrix results in symmetric matrix,

```
S+S'
```

produces,

```
ans =
! 32. 8. 11. 17.   !
```

```
! 8. 20. 17. 23.   !
! 11. 17. 14. 26. !
! 17. 23. 26. 2.   !
```

The multiplication symbol, *, denotes the ***matrix*** multiplication involving inner products between rows and columns. Multiplying a matrix by its transpose also produces a symmetric matrix.

```
S'*S
```

This produces,

```
ans =

! 378. 212. 206. 360. !
! 212. 370. 368. 206. !
! 206. 368. 370. 212. !
! 360. 206. 212. 378. !
```

The determinant of this particular matrix happens to be zero, indicating that the matrix is singular.

```
det(S)

ans =

     0.
```

Since the matrix is singular, it does not have an inverse. If you try to compute the inverse with,

```
inv(S)
```

This will produce a warning message,

```
warning
matrix is close to singular or badly scaled.
results may be inaccurate. rcond = 1.1755E-17

ans =

1.0E+14 *

! 1.2509999 3.7529997 - 3.7529997 - 1.2509999 !
! - 3.7529997 - 11.258999 11.258999 3.7529997 !
! 3.7529997 11.258999 - 11.258999 - 3.7529997 !
! - 1.2509999 - 3.7529997 3.7529997 1.2509999 !
```

Arrays

When they are taken away from the world of linear algebra, matrices become two dimensional numeric arrays. Arithmetic operations on arrays are done element-by-element. This means that addition and subtraction are the same for arrays and matrices, but that multiplicative operations are different. SCILAB uses a dot, or decimal point, as part of the notation for multiplicative array operations.

Array operations are useful for building tables. Suppose n is the column vector,

```
s =  [1:6]';
```

Using this column vector we can generate a table of algorithms,

```
[s ; log10(s)]
```

This produces,

```
ans =

! 1.          0.        !
! 2.          .30103    !
! 3.          .4771213  !
! 4.          .6020600  !
! 5.          .69897    !
! 6.          .7781513  !
```

Multivariate Data

SCILAB uses column-oriented analysis for multivariate statistical data. Each column in a data set represents a variable and each row an observation. The `(i,j)th` element is the `ith` observation of the `jth` variable.

For example, consider an data set with two variables,

```
m_val =

! 100.     82. !
! 120.     88. !
! 124.     92. !
! 97.      76. !
! 110.     80. !
```

The data contains the blood pressure of patient at various instants of time. Using SCILAB we can do various data analysis for this data set.

For example, if we need the average and deviation

```
avg = mean(m_val),dev = st_deviation(m_val)
```

This produces,

```
avg =

! 110.2 !
! 83.6  !
dev =

! 11.882761 !
! 6.3874878 !
```

Scalar Expansion

Matrices and scalars can be combined in several different ways. For example, a scalar is subtracted from a matrix by subtracting it from each element.

For example,

```
s = ones(4,4); s-1
```

This produces,

```
ans =

! 0. 0. 0. 0. !
! 0. 0. 0. 0. !
! 0. 0. 0. 0. !
! 0. 0. 0. 0. !
```

With scalar expansion, SCILAB assigns a specified scalar to all indices in a range.

For example,

```
s(1:2,2:3)=0
s =

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

Matrix Operation

The following Table gives the syntax of the basic matrix operations available in SCILAB

Name	Description
[]	matrix definition, concatenation
()	extraction m=a(k)
'	transpose
-	subtraction
\	left division
^	exponent
.\	elementwise left division
.^	elementwise exponent
./.	kronecker right division

Name	Description
;	row separator
()	insertion: a(k)=m
+	addition
*	multiplication
/	right division
.*	elementwise multiplication
./	elementwise right division
.*.	kronecker product
.\.	kronecker left division

The find Function

The find function determines the indices of array elements that meet a given logical condition. In its simplest form, **find** returns a column vector of indices. Transpose that vector to obtain a row vector of indices.

For example,

Let us use the find function to generate a random sequence whose elements 1 or -1.

```
r_seq  = rand(1,5,'normal');

r_seq(find(r_seq>=0)) =1;

r_seq(find(r_seq<0)) =-1

r_seq =

! - 1. - 1. - 1. 1. - 1. !
```

Flow Control

SCILAB has the following flow control constructs,

- **if Statements**
- **select statements**
- **for loops**
- **while loops**
- **break statements**

if

The `if` statement evaluates a logical expression and executes a group of statements when the expression is ***true***. The optional `elseif` and `else` keywords provide for the execution of alternate groups of statements.

An `end` keyword, which matches the `if`, terminates the last group of statements. The optional `elseif` and `else` provide for the execution of alternate groups of statements. The line structure given above is not significant, the only constraint is that each then keyword must be on the same line line as its corresponding `if` or `elseif` keyword.

The general expression is,

```
if  condition
// code
elseif  condition
// code
else
end
```

For example,

```
    if  modulo(num,2) == 0
    disp('The Number is Even');
    elseif modulo(num,2) ~=0
    disp('The Number is Odd');
    else
    disp('Invalid Number');
end
```

In this code there are some logical expressions like greater than, less than,etc..., these are used for the if conditions. The below is a table which gives a list of logical expressions

Name	Description
= =	Equal to
> =	Greater than or equal to
>	Greater than

Name	Description
~ =	Not equal to
< =	Less than or equal to
<	Less than

Select

The **select** statement executes groups of statements based on the value of a variable or expression. The key-words **case** delineate the groups. Only the first matching case is executed. There must always be an **end** to match the select.

The general expression is,

```
select  condition
    case 1
        // code
    case N
        // code
end
```

The above example can be written using select as follows,

```
select  modulo(num,2)
    case  0
        disp('The Number is Even');
    case  1
        disp('The Number is Odd');
    case 2
        disp('Inavlid Number');
end
```

Note: The select instruction can be used instead of multiple if statements. This has definite advantage over the multiple if statements.

for

The for loop repeats a group of statements a fixed, predetermined number of times. A matching end delineates the statements.

The general expression is,

```
for  variable = n1:step:n2
```

```
        // code ;
    end
```

The semicolon terminating the inner statement suppresses printing of the result.

For example,

```
//Program to generate  Bipolar signal  +1 / -1
mat = rand(1,10,'normal');
binary =zeros(size(mat));
for  count = 1:1:length(mat)
    if mat(count) >= 0
        binary(count) =1;
    else
       binary(count)  =-1;
    end
end
```

while

The while loop repeats a group of statements an indefinite number of times under control of a logical condition. A matching end delineates the statements.

The general expression is,

```
while condition
    // code
    // loop counter i.e., count =count +1;
end
```

The above can be written using a while loop as ,

```
mat = rand(1,10,'normal');
binary =zeros(size(mat));
count = 1;
   while( count <=  length(mat))
       if mat(count) >= 0
           binary(count) =1;
```

```
        else
            binary(count)  =-1;
        end
        count =count+1;
    end
```

break

The break statement lets you exit early from a for or while loop. In nested loops, break exits from the innermost loop only.

Let us take the previous example, if we want to exit the while loop when the value of count reaches 5. Using break statement we can achieve this.

```
    mat = rand(1,10,'normal');
    binary =zeros(size(mat));
    count = 1;
    while( count <=  length(mat))
        if mat(count) >= 0
            binary(count) =1;
        else
             binary(count)  =-1;
        end
        count =count+1;
// break condition
        if count == 5
            break;
        end
    end
```