Table of Contents

Preface ii

Introduction 1

Tutorial 1 – Scilab Environment 3

Exercise 1 – Scilab Environment 5

Tutorial 2 – The Workspace and Working Directory 6

Exercise 2 – The Workspace and Working Directory 8

Tutorial 3 – Creating Matrices and Some Simple Matrix Operations 9

Exercise 3 – Creating Matrices and Some Simple Matrix Operations 12

Tutorial 4 – Sub-matrices 13

Exercise 4 – Sub-matrices 15

Tutorial 5 – Statistics 16

Tutorial 6 – Working with Polynomials 17

Exercise 6 – Working with Polynomials 20

Tutorial 7 – Plotting Graphs 21

Tutorial 8 – Plotting 3D Graphs 24

Tutorial 9 – Scilab Programming Language 26

Tutorial 10 – Script Files and Function Files 30

Tutorial 11 – Functions in Scilab 32

Tutorial 12 – File Operations 35

Tutorial 13 – Polynomial Curve Fitting 37

Tutorial 14 – Reading Microsoft Excel Files 40

Tutorial 15 – Some Miscellaneous Commands 42

Tutorial 16 – Data Structures 43

Tutorial 17 – Directed Graphs 47

Tutorial 18 – Principal Stresses 51

Scilab is a scientific software package for numerical computations providing a powerful open computing environment for engineering and scientific applications. Developed since 1990 by researchers from INRIA (French National Institute for Research in Computer Science and Control) and ENPC (National School of Bridges and Roads), it is now maintained and developed by Scilab Consortium since its creation in May 2003 and integrated into Digiteo Foundation in July 2008. The current version is 5.2.1 (February 2010).

Since 1994 it is distributed freely along with source code through the Internet and is currently being used in educational and industrial environments around the world. From version 5 it is released under the GPL compatible CeCILL license.

Scilab includes hundreds of mathematical functions with the possibility to add interactively functions from various languages (C, Fortran...). It has sophisticated data structures (including lists, polynomials, rational functions, linear systems...), an interpreter and a high level programming language.

Scilab has been designed to be an open system where the user can define new data types and operations on these data types by using overloading.

A number of toolboxes are available with the system:

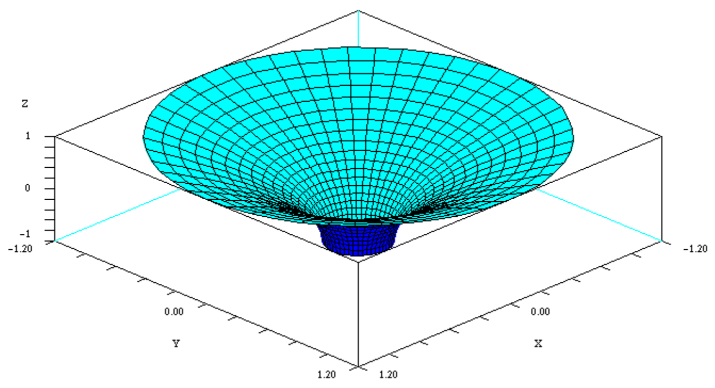
- 2-D and 3-D graphics, animation
- Linear algebra, sparse matrices
- Polynomials and rational functions
- Simulation: ODE solver and DAE solver
- Scicos: a hybrid dynamic systems modeler and simulator
- Classic and robust control, LMI optimization
- Differentiable and non-differentiable optimization
- Signal processing
- Metanet: graphs and networks
- Parallel Scilab using PVM
- Statistics
- Interface with Computer Algebra (Maple, MuPAD)
- Interface with Tcl/Tk
- And a large number of contributions for various domains.

Scilab works on most Unix systems including GNU/Linux and on Windows 9X/NT/2000/XP/Vista/7. It comes with source code, on-line help and English user manuals. Binary versions are available.

Some of its features are listed below:

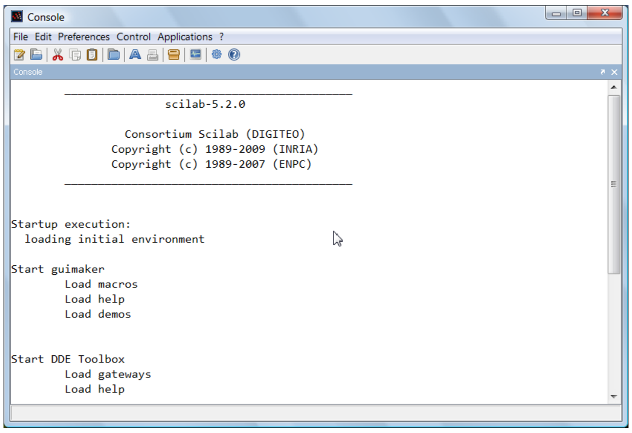
- Basic data type is a matrix, and all matrix operations are available as built-in operations.
- Has a built-in interpreted high-level programming language.
- Graphics such as 2D and 3D graphs can be generated and exported to

various formats so that they can be included into documents.



To the left is a 3D graph generated in Scilab and exported to GIF format and included in the document for presentation. Scilab can export to Postscript and GIF formats as well as to Xfig (popular free software for drawing figures) and LaTeX (free scientific document preparation system) file formats.

When you start up Scilab, you see a window like the one shown in Fig. 1 below. The user enters Scilab commands at the prompt (**-->**). But many of the commands are also available through the menu at the top. The most important menu for a beginner is the "Help" menu. Clicking on the "Help" menu opens up the **Help Browser**, showing a list of topics on which help is available. Clicking on the relevant topic takes you to hyperlinked documents similar to web pages. The Help Browser has two tabs – **Table of Contents** and **Search**.



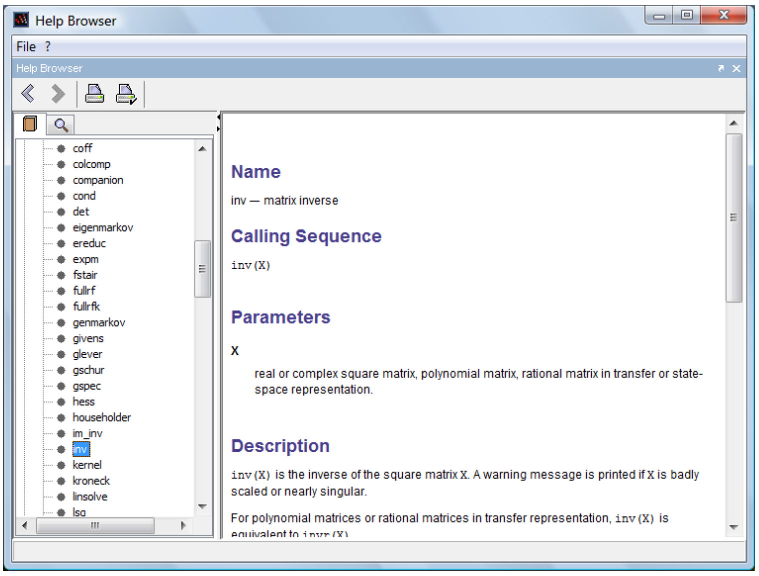
```
                    scilab-5.2.0

               Consortium Scilab (DIGITEO)
             Copyright (c) 1989-2009 (INRIA)
             Copyright (c) 1989-2007 (ENPC)

Startup execution:
  loading initial environment

Start guimaker
        Load macros
        Load help
        Load demos


Start DDE Toolbox
        Load gateways
        Load help
```

Fig. 1.1 Scilab environment

Table of Contents contains an alphabetically arranged list of topics. To see the list of functions available in Scilab, search for **Elementary Functions** in the contents page of Help Browser. Other useful functions may be available under different headings, such as, **Linear Algebra**, **Signal Processing**, **Genetic Algorithms**, **Interpolation**, **Metanet**, **Optimization and Simulation**, **Statistics**, **Strings**, **Time and Date** etc. Use the Search tab to search the help for string patterns.

Help on specific commands can also be accessed directly from the command line instead of having to navigate through a series of links. Thus, to get help on the Scilab command "**inv**", simply type the following command at the prompt. This is useful when you already know the name of the function and want to know its input and output arguments and learn how to use it.

```
-->help inv
```



Name

inv — matrix inverse

Calling Sequence

`inv(X)`

Parameters

X

real or complex square matrix, polynomial matrix, rational matrix in transfer or state-space representation.

Description

`inv(X)` is the inverse of the square matrix X. A warning message is printed if X is badly scaled or nearly singular.

For polynomial matrices or rational matrices in transfer representation, `inv(X)` is equivalent to `invr(X)`

Fig. 1.2 Scilab help browser

Scilab help browser offers two views of the help content. In the contents view, topics are listed in a tree structure which can be expanded or collapsed to browse through the help content. Alternately, the search view allows the user to type in search patterns and topics matching the pattern are listed. The content view gives a good overview while the search view take the user directly to the required topic.

Scilab can be used as a simple calculator to perform numerical calculations. It also has the ability to define variables and store values in them so that they can be used later. This is demonstrated in the following examples:

```
-->2+3
ans =
      5.
-->2/3
ans =
      .6666667
-->2^3
ans =
      8.
```

```
-->a = 2
a =
    2.
-->b = 3
b =
      3.
-->c = a + b
c =
      5.
```

```
-->pi = atan(1.0)*4
pi =
      3.1415927
-->sin(pi/4)
ans =
      0.7071068
-->exp(0.1)
ans =
      1.1051709
```

Usually the answer of a calculation is stored in a variable so that it could be used in the immediately subsequent expressions. If the user does not explicitly supply the name of the variable to store the answer returned by an expression, Scilab uses a variable named **ans** to store such results. If the user specifies a variable to store the results of an expression, results are stored in the user specified variable.

Ending an expression with a semicolon (;) suppresses the echo of the output of an expression. The expression is evaluated and results assigned, but the results are not echoed. Try the following command:

```
-->a = 5;
-->_
```

and you will notice that the prompt reappears immediately, without echoing **a=5**.

You could enter more than one command on the same line by separating the commands by semicolons (;) or a comma (,). The semicolon suppresses echoing of intermediate results but comma does not.

```
-->a=5; b=10; c=15;
-->_
-->disp([a b c])
    5.    10.    15.
-->a=10,b=20,c=30
 a  =
   10.
 b  =
   20.
 c  =
   30.
```

In fact, you could have a mix of semicolons and commas. Expressions followed by semicolons will not be echoed while expressions followed by commas will be echoed.

```
-->a=100; b=200, c=300;
 b  =
    200.
```

1. What are the other ways of obtaining online help in Scilab, other than typing “help” at the command prompt? (**Ans:** From the menu ? ⇨ Scilab Help or press `F1` key)
2. Where are the demo programs that demonstrate Scilab's capabilities? (**Ans:** Go to the menu ? ⇨ Scilab Demos)
3. Where can you find user contributed documents on using and applying Scilab? What are the categories for these documents? Which categories interest you? (**Ans:** Go to the menu ? ⇨ Weblinks ⇨ Contributions)
4. How can you search for a word or pattern in the Scilab online help browser?
5. Can you use the built-in variable “`ans`” in your calculations? If so, is it a good idea to do so or is it better to use your own named variables?
6. When do you think it is useful to use the semicolon (`;`) to suppress the output of a Scilab statement?
7. What are the rules for choosing names for variables in Scilab? Can you use a numeric character as the first character? Can you use underscore (_) as the first character? Can you use special characters, such as -, +, /, ? in a variable name?
8. Can you change the font used by Scilab? (**Ans:** Go to the menu Preferences ⇨ Choose Font)
9. What is the command to clear the screen? (**Ans:** `clc`)
10. What is the short cut key to clear the screen? (**Ans:** `F2` key)
11. What is command history? What are the shortcut keys to use the command history? (**Ans:** Go to the menu Edit ⇨ History. You can also use the arrow keys)
12. List areas within your proposed branch of specialization where Scilab can be a useful tool.
13. Is there a command to record all commands that you type and save them to a file so that you can see them later?
(**Ans:** Type `help diary`)
14. Can you describe some useful application of the diary command?
15. Can you customize Scilab startup to suit your specific needs?
(**Ans:** To customize Scilab at start up, create a file “C:\Documents and Settings\<User>\Scilab\scilab-<version>\ scilab.ini” and put any valid Scilab commands in it that you wish Scilab to execute each time it starts up. <User> must be replaced with your login name and Scilab <version> currently is 5.2.0 so that the folder name is scilab-5.2.0. The above location is for Microsoft Windows XP. For Windows Vista, use C:\Users\<User>\AppData\Roaming\Scilab\<scilab-<version>\scilab.ini).

While the Scilab environment is the visible face of Scilab, there is another that is not visible. It is the memory space where all variables and functions are stored, and is called the **Workspace**. Quite often it is necessary to inspect the workspace to check whether or not a variable or a function has been defined. The following commands help the user in inspecting the memory space: **`who`**, **`whos`** and **`who_user()`**. Use the online help to learn more about these commands.

The **`who`** command lists the names of variables in the Scilab workspace. Note the variable names preceded by the "**%**" symbol. These are special variables that are used often and therefore predefined by Scilab. It includes **`%pi`** (π), **`%e`** (e), **`%i`** ($\sqrt{-1}$), **`%inf`** (∞), **`%nan`** (NaN) and others.

The **`whos`** command lists the variables along with the amount of memory they take up in the workspace. The variables to be listed can be selected based on either their type or name. Some examples are:

`-->whos()`	Lists entire contents of the workspace, including functions, libraries, constants
`-->whos -type constants`	Lists only variables that can store real or complex constants. Other types are boolean, string, function, library, polynomial etc. For a complete list use the command **`-->help typeof`**.
`-->whos -name nam`	Lists all variables whose name begins with the letters **`nam`**

To understand how Scilab deals with numbers, try out the following commands and use the **`whos`** command as follows:

`-->a1=5;`	Defines a real number variable with name '**`a1`**'
`-->a2=sqrt(-4);`	Defines a complex number variable with name '**`a2`**'
`-->a3=[1, 2; 3, 4];`	Defines a 2x2 matrix with name '**`a3`**'
`-->whos -name a`	Lists all variables with name starting with the letter '**`a`**'

```
Name      Type              Size          Bytes

a3        constant          2 by 2        48
a2        constant          1 by 1        32
a1        constant          1 by 1        24
```

Now try the following commands:

`-->a1=sqrt(-9)`	Converts '**`a1`**' to a complex number
`-->whos -name a`	Note that '**`a`**' is now a complex number
`-->a1=a3`	Converts '**`a1`**' to a matrix

`-->a1=sqrt(-9)`	Converts '**`a1`**' to a complex number
`-->whos -name a`	Note that '**`a1`**' is now a matrix
`-->save('ex01.dat')`	Saves all variables in the workspace to a disk file **`ex01.dat`**
`-->load('ex01.dat')`	Loads all variables from a disk file **`ex01.dat`** to workspace

Note the following points:

- Scilab treats a scalar number as a matrix of size 1x1 (and not as a simple number) because the basic data type in Scilab is a matrix.
- Scilab automatically converts the type of the variable as the situation demands. There is no need to specifically define the type for the variable.

1. What are the available data types in Scilab?
(**Ans:** Type the command `help type`).
2. What is the command to list all variables in the Scilab workspace whose name begins with the letters “sa”?
(**Ans:** `whos -name sa`).
3. What is the command to list all variables in the Scilab workspace of type boolean? (**Ans:** `whos -type boolean`).
4. What is the Scilab constant for the value of π?
(**Ans:** `%pi`).
5. What is the command to find the Scilab current working directory? (**Ans:** `pwd` or `getcwd`).
6. Where can you find the Scilab current working directory?
(**Ans:** Fom the menu File ⇨ Get Current Directory).
7. How can you change the Scilab current working directory to a different location?
(**Ans:** `cd(“directory”)` or go to the menu File ⇨ Change Directory and choose the directory you want to go to, in the dialog box).
8. Why is it important to know the Scilab current working directory? (**Ans:** Because it is the default location where Scilab saves all files, unless you explicitly specify the full path).
9. Is Scilab a strongly typed language?
(**Ans:** Strongly typed languages are those in which each variable and its type must first be defined before it can be used. Further, once defined, its type usually cannot be changed. See Wikipedia for a definition of “strongly typed”. Is it the same as what I have written here? What is static typing and type safety?).

Tutorial 3 – Creating Matrices and Some Simple Matrix Operations

Matrix operations built-in into Scilab include addition, subtraction, multiplication, transpose, inversion, determinant, trigonometric, logarithmic, exponential functions and many others. Let us first learn creating matrices:

`-->a=[1 2 3]`	Create a row vector
`-->a=[1; 2; 3]`	Create a column vector
`-->a=[1 2 3]'`	Create a column vector, same as above

`-->x=[1 2 3]; y=[4 5 6 7];`
`-->a=[x y]`
	Create a matrix **a** which is a concatenation of **x** and **y**, provided they are compatible (same number of rows)

`-->a=[1 2 3; 4 5 6; 7 8 9];`	Define a 3x3 matrix. Semicolons indicate end of a row

We can now try out a few simple matrix operations:

`-->b = a';`	Transpose **a** and store it in **b**. Apostrophe (**'**) is the transpose operator.

`-->c = a + b`
`ans  =`
`   2.   6.   10.`
`   6.   10.  14.`
`   10.  14.  18.`
	Add **a** to **b** and store the result in **c**. **a** and **b** must be of the same size. Otherwise, Scilab will report an error.

`-->d = a – b;`	Subtract **b** from **a** and store the result in **d**.

`-->e=a*b`
`ans  =`
`   14.   32.   50.`
`   32.   77.   122.`
`   50.   122.  194.`
	Multiply **a** with **b** and store the result in **e**. **a** and **b** must be compatible for matrix multiplication.

`-->f=[3 1 2; 1 5 3; 2 3 6];`	Define a 3x3 matrix with name **f**.

`-->g = inv(f)`
`g  =`
` 0.4285714  0.        -0.1428571`
` 0.         0.2857143 -0.1428571`
`-0.1428571 -0.1428571  0.2857143`
	Invert matrix **f** and store the result in **g**. **f** must be square and positive definite. Scilab will display a warning if it is ill conditioned.

`-->f*g`	The answer must be an identity matrix

`-->det(f)`
`ans =`
`  49.`
	Determinant of **f**.

`-->log(a);`	Matrix of log of each element of **a**.

Some matrix operations such as multiplication, exponentiation can be performed on an element-wise basis.

`-->a .* b`
`ans =`
`  1.   8.   21.`
`  8.   25.  48.`
`  21.  48.  81.`
	Element by element multiplication.

`-->a^2`
`ans =`
`  30.  36.  42.`
`  66.  81.  96.`
`  102. 126. 150.`
	Same as **`a * a`**

`-->a .^2`
`ans  =`
`   1.   4.   9.`
`   16.  25.  36.`
`   49.  64.  81.`
	Element by element square.

Matrix addition and subtraction already being element-wise operations, element-wise addition and subtraction result in an error:

`-->a .+ b`
`-->a .- b`
	Error

There are some handy utility functions to generate commonly used matrices, such as zero matrices, identity matrices, diagonal matrices, matrix containing randomly generated numbers etc.

`-->a=zeros(5,8)`	Creates a 5x8 matrix with all elements zero.
`-->b=ones(4,6)`	Creates a 4x6 matrix with all elements 1
`-->c=eye(3,3)`	Creates a 3x3 identity matrix
`-->d=eye(3,3)*10`	Creates a 3x3 diagonal matrix, with diagonal elements equal to 10.
`-->x=rand(3, 5)`	Create a matrix with 3 rows and 5 columns, the elements being random numbers between 0 and 1.
`-->y=rand(3, 5) * 10`	Create a matrix with 3 rows and 5 columns, the elements being random real numbers between 0 and 10.
`-->y=int(rand(3, 5)*100);`	Create a matrix with 3 rows and 5 columns, the elements being random integer numbers between 0 and 100.

It is possible to determine the size, length and type of Scilab variables.

`-->y=[1:5; 6:10; 11:15];`	Size returns the number of rows and columns

`-->size(y)`
`ans =`
`    3.    5.`
	in **`y`**

`-->length(y)`
`ans =`
`    15.`
	Number of elements in **`y`**.

`-->y(1)`
`ans =`
`     1.`
	Element **`y(1)`** is the same as **`y(1, 1)`**

`-->y(2)`
`ans =`
`     6.`
	Element **`y(2)`** is the same as **`y(2, 1)`**.
	Elements are counted column-wise

`-->y(6)`
`ans =`
`    12.`
	Element **`y(6)`** is the same as **`y(3, 2)`**.
	Elements are counted column-wise

It is possible to generate a **range** of numbers to form a vector. Study the following commands:

`-->1:5`	Generates a row of numbers from 1 to 5 at an increment of 1
`-->(1:5)'`	Generates a column of numbers from 1 to 5 at an increment of 1
`-->a=[1:5]`	Creates a row vector with 5 elements. Same as **`[1, 2, 3, 4, 5]`**
`-->b=[1:5]'`	Creates a column vector with 5 elements. Same as **`[1; 2; 3; 4; 5]`**
`-->c=[0:0.5:5]`	Creates a vector with 11 elements as follows **`[0, 0.5, 1.0, 1.5, ... 4.5, 5.0]`**

A range requires a start value, an increment and an end value, separated by colons (**`:`**). If only two values are given (separated by only one colon), they are taken to be the start and end values and the increment is assumed to be 1 (**`a:b`** is a short form for **`a:1:b`**, where **`a`** and **`b`** are scalars). The increment must be negative when the start value is greater than the end value.

Ranges play an important role in sub-matrix operations, which we will see in the next tutorial.

You can also create an empty matrix with the command:

```
-->a=[]
-->size(a)
ans =
   0.      0.
-->whos -name a
Name                 Type       Size     Bytes
a                    constant   0 by 0   16
```

1. Can you use the `.+` operator like you can use `.*`? (**Ans:** No. In fact, there is no need to).
2. What is the size of an empty matrix `a = []`?
   (**Ans:** Size 0 x 0)
3. While generating a range, can you specify a negative increment?
   (**Ans:** Yes, if the start vale is greater than the end value)
4. What is the command to generate values from 0 to 2π at an increment of $\pi/16$? (**Ans:** `0:%pi/16:2*%pi`).
5. What is the command to extract the diagonal elements of a square matrix into a vector? (**Ans:** `a = diag(x)` creates a vector `a` containing the diagonal elements of matrix `x`).
6. Given a square matrix `a`, how can you create a matrix `b` whose diagonal elements are the same as those of `a` but the other elements are all zero?

   (**Ans:** `b = eye(a) .* a`).
7. Extract the off-diagonal terms (at an offset of 1) of a square matrix into a vector.
   (**Ans:** `b = diag(x, 1)` to extract the terms above the diagonal and `b = diag(x, -1)` to extract the terms below the diagonal. `b = diag(x)` is a shortcut for `b = diag(x, 0)`. The offset defaults to zero).
8. Create a matrix of size 5x5 having the required elements on the diagonal, above the diagonal and below the diagonal.
   (**Ans:** `b = diag([1:5])` creates a 5x5 matrix whose diagonal elements are the elements of the vector `[1 2 3 4 5]`).
9. Create a matrix of size 5x5 having the required elements on the diagonal above the main diagonal.
   (**Ans:** `b = diag([1:4], 1)` creates a 5x5 matrix of zeros and puts the elements of the vector `[1 2 3 4]` on the diagonal above the main diagonal. To place the vector on the diagonal below the main diagonal use `b = diag([1:4], -1)`).
10. Create a tri-diagonal matrix of size 5x5 with the specified elements on the main diagonal, above and below the main diagonal.
    (**Ans:** `b = diag([1:5]) + diag([6:9], 1) + diag([10:13], -1)` will put the vector `[1 2 3 4 5]` on the main diagonal, `[6 7 8 9]` on the diagonal above the main diagonal and `[10 11 12 13]` on the diagonal below the main diagonal).

It is possible to extract or replace an element of a matrix by referring to its row and column indices.

`-->a = [1 2 3; 4 5 6];`	Create a matrix with 2 rows and 3 columns.

```
-->b = a(2,2)
b =
    5.
```
	Extract the element of **a** at row 2 and column 2 and assign it to **b**

```
-->a(2, 2) = 100
a =
    1.    2.    3.
    4.  100.    6.
```
	Replace element of **a** at row 2 and column 2 with the value 100

Similar operations can be performed on a sub-matrix of an existing matrix. A sub-matrix can be identified by the row and column numbers at which it starts and ends. Let us first create a matrix of size 5x8.

`-->a = int(rand(5,8)*100)`	Generates a 5x8 matrix whose elements are random integer numbers between 0 and 100.

Since the elements are random numbers, each person will get a different matrix. Let us assume we wish to identify a 2x4 sub-matrix of **a** demarcated by rows 3 to 4 and columns 2 to 5. This is done with **a(3:4, 2:5)**. The range of rows and columns is represented by the range commands **3:4** and **2:5** respectively. Thus **3:4** defines the range 3, 4 while **2:5** defines the range 2, 3, 4, 5. However, matrix '**a**' remains unaffected.

`-->b=a(3:4, 2:5)`	This command copies the contiguous sub-matrix of size 2x4 starting from element at (3, 2) up to element (4, 5) of **a** into **b**.

A sub-matrix can be overwritten just as easily as it can be copied. To make all elements of the sub-matrix between the above range equal to zero, use the following command:

`-->a(3:4, 2:5) = zeros(2,4)`	This command creates a 2x4 matrix of zeros and puts it into the sub-matrix of **a** between rows 3:4 and columns 2:5.

Note that the sub-matrix on the left hand side and the matrix on the right side (a zero matrix in the above example) must be of the same size. However, it is possible to leave out the number of rows and columns in the **zeros()** command and Scilab will calculate the correct number of rows and columns required. Thus, the following command is the same as the one above:

```
-->a(3:4, 2:5) = zeros()
```

While using range to demarcate rows and/or columns, it is permitted to leave out both the start and end values in the range, in which case they are assumed to be 1 and the number of the last row (or column), respectively. To indicate all rows (or columns) it is enough to use only the colon (**:**). Thus, the sub-matrix consisting of all the rows and columns 2 and 3 of **a** is **a(:, 2:3)**. Naturally **a(:, :)** represents the whole matrix, which of course could be represented simply as **a**.

Scilab uses a special symbol to refer to the number of the last row (or column) by the symbol **$**. This can be used within range specifications to represent the number of the last row (or column).

It is not possible to specify only the start value of the range and leave out the end value or vice versa. Either both must be specified or both must be left out. The sub-matrix consisting of all rows of a and columns 3 to 8 is represented as:

```
-->a(:, 3:$)
```

In the below, we are extracting all rows and columns from column 3 to last but one, that is 7. Note that **$** is the last column and **$-1** is the last but one column.

```
-->a(:, 3:$-1)
```

Note that the sub-matrix need not necessarily consist of contiguous rows and/or columns. For example, to extract the odd rows and column from matrix **a**, you could use the following command:

`-->c = a(1:2:$, 1:2:$)`	This command copies the sub-matrix of **a** with rows 1, 3, 5 and columns 1, 3, 5, 7 into **b**. The rows are represented by the range **1:2:$** which implies start from 1, increment by 2 each time and up to **$**, which in this case is 5.
`-->c = a(:, $:-1:1)`	You can even reverse the order of columns (or rows or both). The range **$:-1:1** reverses the order of the columns (rows are unaltered), and the modified matrix are stored in **c**.

Exercise 4 – Sub-matrices

1. Extract the last column of a matrix **`a`** and store it in matrix **`b`**.
   (**Ans:** **`b = a(:, $)`**).
2. Extract the last but one column of a matrix **`a`** and store it in matrix **`b`**.
   (**Ans:** **`b = a(:, $-1)`**).
3. Replace the sub-matrix between rows 3 to 5 and columns 2 to the last but one column, in matrix **`a`** of size 5x5 with zeros.
   (**Ans:** **`a(3:5, 2:$-1) = zeros(3, 4)`**. Instead of **`zeros(3,4)`** you can use **`zeros(:,:)`** and Scilab will calculate the required number of rows and columns itself).
4. Replace the even numbered columns of matrix **`a`** having size 3x5 with ones. (**Ans:** **`a(:,2:2:$)=ones(:, :)`**).
5. What is the sub-matrix of **`a`** extracted by the following command **`a(1:3,$-2:$)`**?
   (**Ans:** It extracts the first 3 rows and last 3 columns of matrix **`a`**).
6. Assuming **`a`** to be a 5x8 matrix, are the following valid commands? If so, what do they do? If not, what is the correct command?

1.	**`A(1:, 5)`**	Incorrect. Should indicate end row number.
2.	**`A(:, 5)`**	Correct
3.	**`a(1:3, $-1:$)`**	Correct
4.	**`a(:$, 3:6)`**	Incorrect. Should indicate start row number.

Scilab can perform all basic statistical calculations. The data is assumed to be contained in a matrix and calculations can be performed treating rows (or columns) as the observations and the columns (or rows) as the parameters. To choose rows as the observations, the indicator is **`r`** or **`1`**. To choose columns as the observations, the indicator is '**`c`**' or **`2`**. If no indicator is furnished, the operation is applied to the entire matrix element by element. The available statistical functions are **`sum()`**, **`mean()`**, **`stdev()`**, **`st_deviation()`**, **`median()`**.

Let us first generate a matrix of 5 observations on 3 parameters. For the purpose of this demonstration, let the elements be random numbers. This is done using the following command:

`-->a=rand(5,3)`	Creates a 5x3 matrix of random numbers .

Assuming rows to be observations and columns to be parameters, the sum, mean and standard deviation are calculated as follows:

`-->s=sum(a, 'r')`	Sum of columns of **`a`**.
`-->m=mean(a, 1)`	Mean value of each column of **`a`**.
`-->sd=stdev(a, 1)`	Standard deviation of **`a`**. Population size standard deviation.
`-->sd2=st_deviation(a, 'r')`	Standard deviation of **`a`**. Sample size standard deviation.
`-->mdn=median(a,'r')`	Median of columns of **`a`**.

The same operations can be performed treating columns as observations by replacing the **`r`** or **`1`** with **`c`** or **`2`**.

When neither **`r`** (or **`1`**) nor **`c`** (or **`2`**) is supplied, the operations are carried out treating the entire matrix as a set of observations on a single parameter.

The maximum and minimum values in a column, row or matrix can be obtained with the **`max()`** and **`min()`** functions respectively in the same way as the statistical functions above, except that you must use **`r`** or **`c`** but not **`1`** or **`2`**.

Scilab has support for operations on polynomials. You can create polynomials, find their roots and perform operations on them such as addition, subtraction, multiplication, division, simplification etc.

A polynomial can be created in two ways. One way is to define the polynomial in terms of its roots and the other way is to define it in terms of its coefficients. While creating a polynomial, we must choose a name for the symbolic variable and indicate whether the polynomial is to be created in terms of roots or in terms of coefficients. The symbolic variable name can be of any length, but only the first four characters are significant, the other characters are ignored.

`-->p1 = poly([3 2], 'x')`
or
`-->p1 = poly([3 2], 'x', 'r')`
	Create a polynomial **p1** having roots 3 and 2, taking the name of the symbolic variable as **x**. The polynomial is **p1 =** $6-5x+x^2$.

`-->p2 = poly([6 -5 1], 'x', 'c')`	Create a polynomial **p2** having coefficients $6-5x+x^2$, assuming the name of the symbolic variable as **x**. The polynomial is **p2 =** $6-5x+x^2$.

The third parameter, when supplied, may be **'r'** or **'roots'** in which case the first parameter is a vector containing the roots of the polynomial or it may be **'c'** or **'coeff'** in which case the first parameter is a vector containing the coefficients of the polynomial, starting from the constant as the first element and power of the symbolic variable increasing by one for each element in the vector.

Thus, the polynomial with two roots is a polynomial of order two. Similarly, a polynomial with three coefficients is also a polynomial of order two. When the third parameter is not supplied, it defaults to **'r'** or **'roots'**.

It is possible to perform a number of operations on polynomials, such as, find its roots, add, subtract, multiply, divide and simplify.

`-->p1 = poly([6 -5 1], 'x')`
`p1 =`
`                2`
`      6 - 5x + x`
`-->roots(p1)`
`ans =`
`    2.`
`    3.`
	Create a polynomial **p1** having coefficients 6, -5 and 1, taking the name of the symbolic variable as **x**. The polynomial is **p1 =** $6-5x+x^2$. Find the roots using the function **roots(p1)**.

`-->p3 = p1 + p2`
`p3 =`
`                 2`
`    12 - 10x + 2x`
	Add the two polynomials **p1** and **p2** and store the result in the polynomial **p3**. Subtraction can be performed in a similar way.

```
-->p4 = p1 * p2
p4 =
                2      3    4
    36 - 60x 37x  - 10x  + x
```

Product of two polynomials is also a polynomial, and is calculated using the multiplication operator (*****).

```
-->p5 = p1 / p2
p5 =
     1
     -
     1
-->typeof(p5)
ans =
       rational
```

Dividing a polynomial with another results not in a polynomial, but a rational.

```
-->p1 == p2
ans =
      T
```

We can perform boolean operations on polynomials.

```
-->coeff(p1)
ans =
     6.   -5.   -1.
```

Find the coefficients of the polynomial **p1**.

```
-->derivat(p1)
ans =
      -5 + 2x
```

Find the derivative of the polynomial **p1**.

```
-->c = companion(p1)
c =
       5   -6
       1    0
```

Companion matrix of the polynomial is the matrix whose characteristic equation is the given polynomial.

```
-->roots(p1)'
ans =
       2.   3.
-->spec(c)
ans =
      3.
      2.
```

The roots of the characteristic equation are the eigenvalues of the companion matrix.

There are other ways of defining polynomials.

```
-->x = poly(0, 'x')
x =
      x
```

Create a variable named as **x**, which is a polynomial of degree zero. The variable **x** can be used as a **seed** to create other polynomials.

```
-->p6 = 6 - 5 * x + x^2
-->roots(p6)
ans =
      2.
      3.
```

Create the polynomial **p1** the same way you would write it on paper. This is the same as the previous polynomials **p1** and **p2** that we have created.

```
-->c=[5  -6; 1  0];
-->p7 = poly(c, 'x')
```

Create a polynomial from its companion matrix.

```
p7 =
     6 – 5x + x^2
```

Let us now take a look at rational polynomials and explore the functions related to them.

```
-->x = poly(0, 'x')
-->p=(1+2*x+3*x^2)/(4+5*x+6*x^2)
p =
                2
     1 + 2x + 3x
     -----------
                2
     4 + 5x + 6x
```

Create the variable **x** as a polynomial of degree 0, thus creating a **seed** for a polynomial in **x**.

Create the polynomial **p** with numerator as $1+2x+3x^2$ and denominator $4+5x+6x^2$.

```
-->numer(p)
ans =
                2
     1 + 2x + 3x
-->denom(p)
ans=
                2
     4 + 5x + 6x
```

Extract the numerator and denominator of the rational polynomial.

```
-->derivat(p)
ans =
                         2
              3 + 12x + x
      -------------------------
                    2     3     4
     16 + 40x + 73x  +60x  + 36
```

Find the derivative of the rational polynomial.

```
-->[n, d]=simp((x+1)*(x+2),
(x+1)*(x-2))
d =
    -2 + x
n =
    2 + x
```

Simplify a rational polynomial given the numerator and denominator of the rational polynomial. Returns the numerator and denominator after simplification.

Exercise 6 – Working with Polynomials

1. Create a polynomial with **`x`** as the symbolic variable such that its roots are 2 and 3.
   (**Ans:** `p = poly([2 3], 'x')`).
2. Create a polynomial to represent 6 – 5x + x^2?
   (**Ans:** `p = poly([6 -5 1], 'x', 'coeff')`)
3. Find the roots of this polynomial. (**Ans:** 2 and 3).
4. What operations can you perform on polynomials?
   (**Ans:** You can perform addition, subtraction, multiplication and division operations on polynomials. These operations are permitted provided the polynomials have the same symbolic variable. But operations such as trigonometric, logarithmic are not permitted).

Let us learn to plot simple graphs. We will have to generate the data to be used for the graph. Let us assume we want to draw the graph of $\cos(x)$ and $\sin(x)$ for one full cycle (2π radians). Let us generate the values for the x-axis, dividing the cycle into 16 equal intervals, with the following command:

```
-->x=[0:%pi/16:2*%pi]';
```

In the above command, note that **`%pi`** is a predefined constant representing the value of π . The command to create a range of values, **`0:%pi/16:2*%pi`**, requires a start value, an increment and an end value. In the above example, they are 0, $\pi/16$ and 2π respectively. Thus, **x** is a vector containing 33 elements.

Next, let us create the values for the y-axis, first column representing cosine and the second sine. They are created by the following commands:

```
-->y=[cos(x) sin(x)]
```

Note that **`cos(x)`** and **`sin(x)`** are the two columns of a new matrix which is first created and then stored in **y**. We can now plot the graph with the command:

```
-->plot2d(x,y)
```

The graph generated by this command is shown below. The graph can be enhanced and annotated. You can add grid lines, labels for x- and y-axes, legend for the different lines etc. You can learn more about the **`plot2d`** and other related functions from the online help.

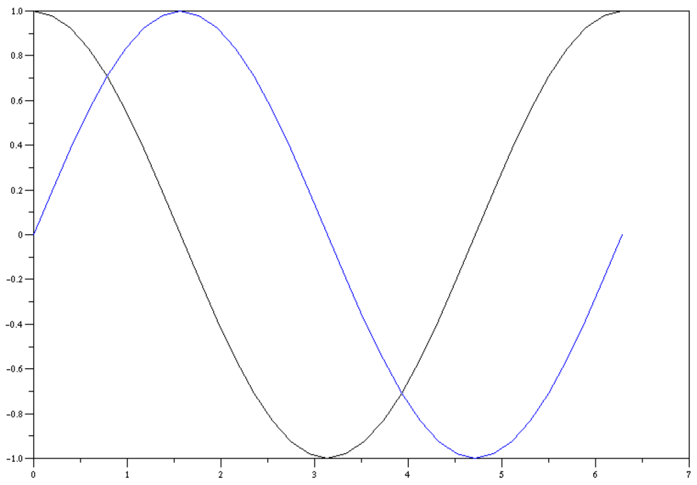


*Fig. 6.1 Graph of sin(x) and cos(x) using function **`plot2d()`***

In the above command, the number of rows in **x** and **y** must be the same and columns of **y** are plotted versus **x**. Since there is only one column in **x** and there are 2 columns in **y**, the two lines are plotted on the graph, taking **x** to be common for both lines.

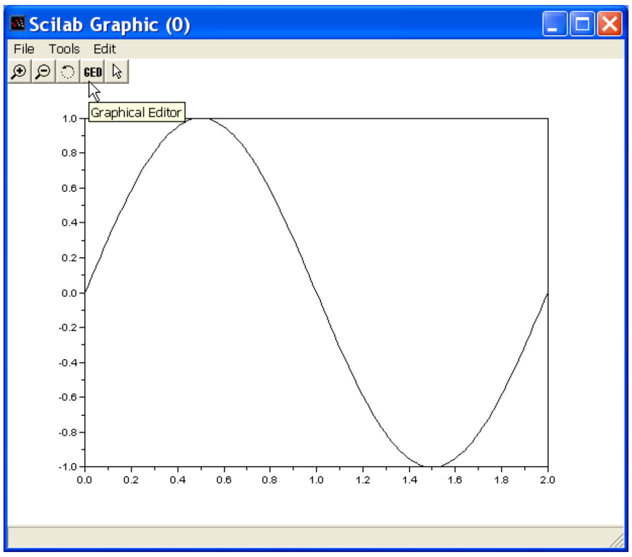
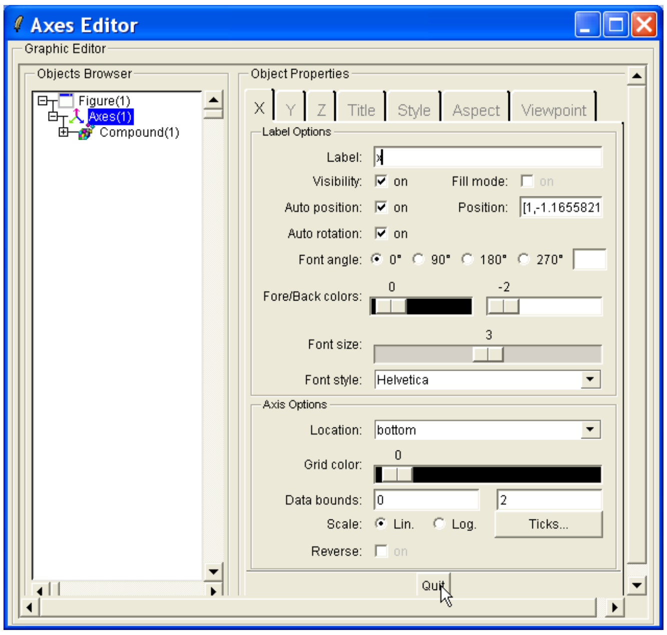


Fig. 6.2 Customization of graphs interactively

You can customize the graph interactively through a of dialog box. You can set axis labels (label, font, font size, colour etc.), grid (line type, colour etc.), legend for the graphs. To do so select Figure Properties from the Edit menu in the graphic window (Fig. 6.2).

To draw the grid lines parallel to the x and/or y axis, choose the colour for the grid as 0 or more (by default grid colour is set to -1, in which case grid lines are not drawn). You can type in a label for the axes and set the font, font size and colour. You could also customize the location of the axes (top, middle or bottom for x-axis and left, middle or right for y-axis). You can also specify the axis scaling to be either linear (the default) or logarithmic. You can redefine the minimum and maximum values for the axes and reverse the direction of the axis (right to left for x-axis for x-axis instead of left to right or top to bottom for y-axis instead of from bottom to top).



Label Options

Label: x

Visibility: [✓ on]   Fill mode: on

Auto position: [✓ on]   Position: [1,-1.1655821

Auto rotation: [✓ on]

Font angle: [✓ 0°] 90° 180° 270°

Fore/Back colors: 0 -2

Font size: 3

Font style: Helvetica

Axis Options

Location: bottom

Grid color: 0

Data bounds: 0 2

Scale: [✓ Lin.] Log.   Ticks...

Reverse: on

Fig. 6.3 Customization of axes interactively

You can also customize every other component of the graph, by clicking on the appropriate element in the object browser on the left.

You can export the graph to one of the supported formats (GIF, Postscript, BMP, EMF) by clicking on File ⇨ Export in the main menu of the graphic window.

Graphs can be annotated through command lines also. Grids, titles, axis labels, legends etc. can be added with the following commands:

```
-->x=[0:%pi/32;2*%pi]'; y(:,1)=cos(x); y(:,2)=sin(x); y(:3)=cos(x)
+sin(x);
-->plot(x, y); xgrid(1);
-->xtitle('TRIGINOMETRIC FUNCTIONS', 'x', 'f(x)');
-->legend('cos(x)', 'sin(x), 'cos(x) + sin(x)', 1, %F);
```

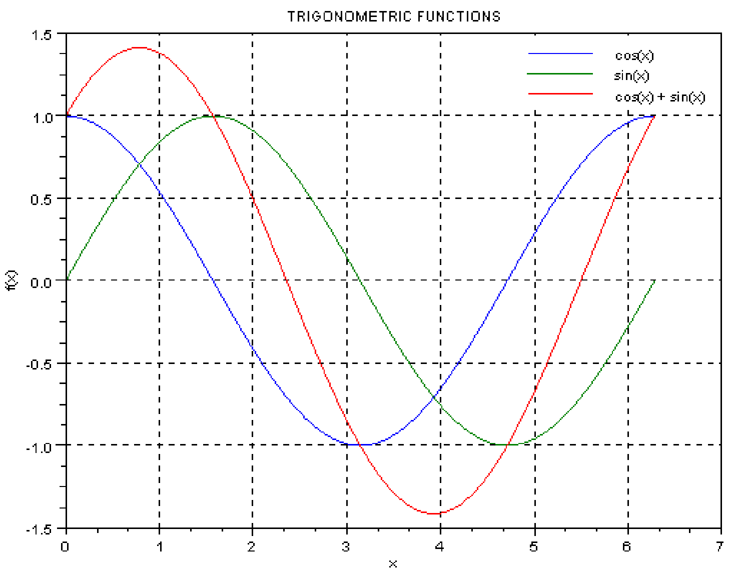


Fig. 6.4 Annotation of plot through commands

Use the online help system to learn more details about the commands **`xgrid`**, **`xtitle`**, **`legend`**. For example, the legend command above places the legend in the upper right (**`1`**) corner and does not draw a box around the legend (**`%F`**). The alternatives for position of legend are **`2`** for top left, **`3`** for bottom left and **`4`** for bottom right. To draw a box around the legend, either use **`%T`** or do not give any value and it defaults to **`%T`**. There are other commands, such as **`xlabel`** and **`ylabel`**, that you may find useful.

Three dimensional plots of surfaces can be plotted with the **plot3d()** family of functions. The plot3d() function in its simplest form requires a grid of (x, y) values and corresponding values of **z**. The grid of (x, y) values is specifies with the help of two vectors, one for x-axis and the other for y-axis.

```
-->x=[0:%pi/16:2*%pi]'; size(x)
ans =
    33.    1.
-->z=sin(x) * ones(x)';
-->plot3d(x, x, z);
```

The first command on line 1 generates a column vector for one full cycle (0 to 2π) at an interval of π/16 and has size 33x1. Let us assume the grid spacing to be the same along x and y directions. Therefore another vector to represent the spacing of points along y direction is not required. The second command generates the z values for each point on the grid as the sine of the x value. This generates a 3 dimensional sine wave surface.

It is possible to have different spacing of points along and y directions.

```
-->x=[0:%pi/16:2*%pi]'; disp(size(x))
   33.    1.
-->y=[0:0.5:5]; disp(size(y))
    1.   10.
-->z=sin(x)*ones(y); disp(size(z))
   33.   10.
-->plot3d(x, y, z)
```

This also generates a sine wave surface but the spacing of the grid along the y direction is different as compared to the spacing of the grid along x direction.

The function **plot3d1()** is similar to **plot3d()** except that it produces a colour level plot of a surface using same colours for points with equal height.

```
-->plot3d1(x, y, z)
```

The function **plot3d2(x, y, z)** generates a surface using facets rather than grids. Here x, y and z are each two dimensional matrices which describe a surface. The surface is composed of four sided polygons. x(i, j), x(i+1, j), x(i, j+1) and x(i+1, j+1) represent the x coordinates of one facet. Y and z represent the y and z coordinates in a similar way.

```
-->u=linspace(-%pi/2, %pi/2, 40);
-->v=linspace(0, 2*%pi, 20);
-->x=cos(u)' * cos(v);
-->y=cos(u)' * sin(v);
-->z=sin(u)' * ones(v);
-->plot3d2(x, y, z)
```

The function **plot3d3()** is similar to **plot3d2()** except that it generates a mesh of the surface instead of a shaded surface with hidden lines removed.

It is possible to plot multiple graphs on one page with the help of the **`subplot()`** function. It can subdivide the page into a rectangular array of rows and columns and position a graph in a chosen cell. For example s**`ubplot(235)`** divides the page into 2 rows and 3 columns resulting in 6 cells. Cells are counted in sequence starting from left top. The above command specifies that the output of the next plot command must be put in cell number 5, that is on row 2 and column 2. Try out the following:

```
-->clf();subplot(121);plot3d3(x,y,z);subplot(122);plot3d2(x,y,z)
```

This shows the same surface first as a mesh and the second as a surface with hidden lines removed, with the graphs placed side by side. The function **`clf()`** clears the current graphic figure.

Scilab has a built-in interpreted programming language so that a series of commands can be automated. The programming language offers many features of a high level language, such as looping (**for**, **while**), conditional execution (**if-then-else**, **select**) and functions. The greatest advantage is that the statements can be any valid Scilab commands.

To loop over an index variable **i** from 1 to 10 and display its value each time, you can try the following commands at the prompt:

```
-->for i=1:10
-->disp(i)
-->end
```

The **for** loop is closed by the corresponding **end** statement. Once the loop is closed, the block of statements enclosed within the loop will be executed. In the above example, the **disp(i)** command displays the value of **i**.

Conditional execution is performed using the **if-then-elseif-else** construct. Try the following statements on the command line:

```
-->x=10;
-->if x<0 then disp('Negative')
-->elseif x==0 then disp('Zero')
-->else disp('Positive')
    Positive
-->end
```

This will display the word **Positive**. You may also notice that the statement **disp('Positive')** is executed even before the keyword **end** is typed.

A list of all the Scilab programming language primitives and commands can be displayed by the command **what()**. The command produces the following list:

if	**else**	**for**	**while**	**end**	**select**
case	**quit**	**exit**	**return**	**help**	**what**
who	**pause**	**clear**	**resume**	**then**	**do**
apropos	**abort**	**break**	**elseif**		

You can learn about each command using the help command. Thus **help while** will give complete information about **while** along with examples and a list of other commands that are related to it. The greatest advantage is that you can test out every language feature interactively within the Scilab environment before putting them into a separate file as a function.

Following operators are available in Scilab:

Symbol	Operation	Symbol	Operation
[ ]	Matrix definition	**;**	Statement separator
( )	Extraction eg. m = a(k)	**( )**	Insertion eg. a(k) = m
'	Transpose	**+**	Addition
-	Subtraction	*****	Multiplication

`\`	Left division	`/`	Right division
`^`	Exponent	`.*`	Element wise multiplication
`.\`	Element wise left division	`./`	Element wise right division
`.^`	Element wise exponent		

Following are the boolean operators available in Scilab:

Symbol	Operation	Symbol	Operation
`==`	Equal to	`~=`	Not equal to (also `<>`)
`<`	Less than	`<=`	Less than or equal to
`>`	Greater than	`>=`	Greater than or equal to
`~`	Negation		
`&`	Element wise AND	`|`	Element wise OR

The boolean constants **`%t`** and **`%T`** represent **TRUE** and **`%f`** and **`%F`** represent **FALSE**. The following are examples for typical boolean operations:

`-->a = int(rand(3,4)*100)`	Generate matrix **a** with 3 rows and 4 columns and containing integer random numbers between 0 and 100.
`-->a == 0`	Returns a matrix **a** with same size as matrix **a**, with elements either T (if the element is equal to zero) or F (if the element is not zero).
`-->a < 20`	Creates matrix **a** with same size as matrix **a**, with elements either **T** (if the element is less than 20) or **F** (if the element is greater than or equal to 20).
`-->bool2s(a < 20)`	First creates a matrix of boolean values **T** or **F** depending on whether the element in matrix **a** is less than 20 or greater than or equal to 20. Then creates a matrix with numerical vales 1 or 0 corresponding to **T** or **F**, respectively
`-->find(a < 20)`	Creates a vector containing the indices of elements of matrix **a** which are less than 20

Let us generate a matrix **a** of size 3x4 containing random integer values between 0 and 100 with the command **`a = int(rand(3,4)*100)`**. If matrix **a** contains the elements as shown below:

$$[a]=\begin{bmatrix} 21 & 33 & 84 & 6 \\ 75 & 66 & 68 & 56 \\ 0 & 62 & 87 & 66 \end{bmatrix},$$ then the results of the various boolean operations are given below:

To test if elements of **a** are zero:

```
-->a == 0
 ans =
  F  F  F  F
  F  F  F  F
  T   F   F   F
```

To test if value of elements of **a** are less than 25:

```
-->a < 25
 ans =
  T   F   F   T
  F   F   F   F
  T   F   F   F
```

To test if elements of **a** are greater than or equal to 62:

```
-->a >= 62
 ans =
  F   F   T   F
  T   T   T   F
  F   T   T   T
```

To test if elements of **a** are not equal to 66:

```
a ~= 66
 ans =
  T   T   T   T
  T   F   T   T
  T   T   T   F
```

To test if elements of **a** are not equal to 66, and output 1 instead of T and 0 instead of F:

```
-->bool2s(a ~= 66)
 ans =
  1.   1.   1.   1.
  1.   0.   1.   1.
  1.   1.   1.   0.
```

To find indices of elements of **a** which are greater than 70:

```
-->b=find(a > 70)
 b =
  2.   7.   9.
```

The indices are counted as if **a** were a one dimensional vector and indices counted column wise starting from the top of the left most column. To print the values of **a** corresponding to these indices use the following command:

```
-->a(b)
 ans =
  75.
  84.
  87.
```

Compound boolean operations can be performed using the AND and OR operators. To test if elements of **a** are greater than 20 and less than 70:

```
-->(a > 20) & (a < 70)
 ans =
  T   T   F   F
  F   T   T   T
  F   T   F   T
```

To test if elements of **a** are either less than 25 or greater than 75:

```
-->(a < 25) | a > 75)
  ans =
  T  F  T  T
  F  F  F  F
  T  F  T  F
```

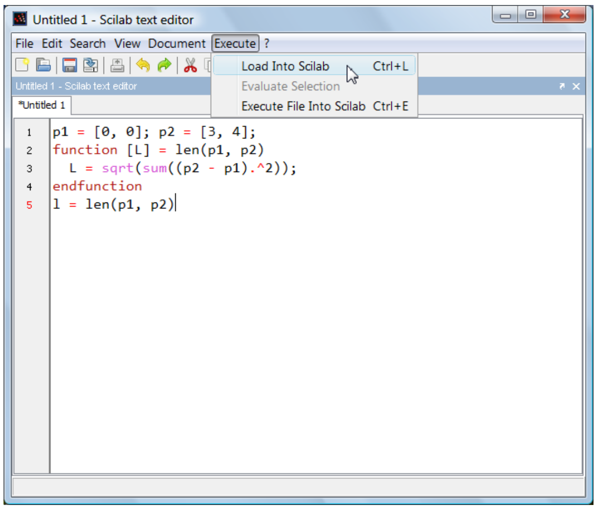
It is possible to test if all elements of matrix **a** meet a logical test using the **and()** function:

```
-->and(a > 20)
 ans =
  F
-->and(a < 100)
 ans =
  T
```

It is possible to test if at least one element of matrix **a** meets a logical test using the **or()** function:

```
-->or(a > 20)
 ans =
  T
-->or(a < 200)
 ans =
  T
```

Script files contain any valid Scilab statements and functions. Script files can be written in the built-in Scilab code editor called **SciPad**. Scipad is invoked from the Scilab main menu through **Application** ⇨ **Editor**. When a script file can be loaded into Scilab workspace from SciPad main menu **Execute** ⇨ **Load into Scilab** (keyboard short cut **Ctrl + L**). Code from **SciPad** can be loaded into Scilab workspace with the command **`exec('filename.sci')`**, provided the file **`filename.sci`** is located in the current working directory. To load function files in a directory other than the current working directory, specify the full path to the file.


Untitled 1 - Scilab text editor
File Edit Search View Document Execute ?
Load Into Scilab Ctrl+L
Evaluate Selection
Execute File Into Scilab Ctrl+E
Untitled 1 - Scilab text editor
*Untitled 1

```
p1 = [0, 0]; p2 = [3, 4];
function [L] = len(p1, p2)
  L = sqrt(sum((p2 - p1).^2));
endfunction
l = len(p1, p2)
```

Fig. 10.1 Script file in SciPad being loaded into Scilab workspace

Before being loaded into Scilab workspace, the file is checked for syntax. If code contains syntax errors, it is not loaded into Scilab workspace, instead, error messages are displayed in the Scilab window. If there are no errors, the code is compiled and loaded into Scilab workspace, and becomes available for use. If changes are made to the code in SciPad, the process of **Execute** ⇨ **Load into Scilab** must be repeated. When modified code replaces previously compiled code, Scilab issues a warning indicating that a function has been modified.

The difference between script files and function files is that script files do not contain any function definitions and contain only Scilab statements mainly for data input, calculations and output. The results of execution of a script file are loaded into Scilab workspace and hence have global scope. That is, all variables loaded from a script file are global variables and are accessible from the Scilab workspace.

On the other hand, a function file usually contains only function definitions and no directly executable Scilab statements. When loaded into Scilab workspace, only the function code is loaded. Functions operate in a modular fashion in that they exchange variable with Scilab workspace or other functions only through their input and output arguments and do not access the Scilab workspace directly. Variables within the functions are local to the function unless it is an input or output variable.

Unlike Matlab or Octave, one function file may contain more than one function definition and there is no stipulated rule for naming files except that they must have the extension **.sci** or **.sce**. Function files can be written using any text editor such as Notepad, and not necessarily only with **SciPad**. But unlike **SciPad**, other editors cannot load the code into Scilab workspace and will have to be loaded with the **`exec()`** command.

Matlab and Octave stipulatie that the name of the file must be the same as the name of the function it contains, with a **`.m`** extension. Consequently, each function file can contain only one function, but functions can be automatically loaded into the workspace as and when required instead of requiring that they be explicitly loaded into the workspace by the user. This also enables them to identify if the source code has been modified and recompile and load the function into the workspace on the fly when needed. They organize function files in folders and can be told in which folders to look when searching for a required function file.

Script files are an easy means of loading data into Scilab workspace and thus avoid file read operations. This is because script files have direct access to Scilab workspace and can read data from and write data to Scilab workspace. While this is easy to understand and use, it can lead to errors in the case of large and complex set of data and code.

Function files do not have direct access to Scilab workspace and therefore exchange data through the input and output parameters. This approach is more complex but modular. Essentially, we are giving up simplicity in exchange for modularity. Function files have the advantage of separating the data in the global workspace from the data used inside the body of a function. While this helps avoid errors, it adds a layer of complexity for the programmer. Due to their restricted and well defined means of data exchange, functions are modular and abstract a complex set of operations into a single function call. Functions are necessary when you wish to build large and complex programs.

Functions serve the same purpose in Scilab as in other programming languages. They are independent blocks of code, with their own input and output parameters which can be associated with variables at the time of calling the function. They modularize program development and encapsulate a series of statements and associate them with the name of the function.

Scilab provides a built in editor within Scilab, called **SciPad**, wherein the user can type the code for functions and compile and load them into the workspace. SciPad can be invoked by clicking on **Applications** ⇨ **Editor** on the main menu at the top of the Scilab work environment.

Let us write a simple function to calculate the length of a line in the x-y plane, given the coordinates of its two ends (x1, y1) and (x2, y2). The length of such a line is given by the expression $l=\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$. Let us first try out the calculation in Scilab for the points (1,4) and (10,6).

```
-->x1=1; y1=4; x2=10; y2=6;
-->dx=x2-x1; dy=y2-y1;
-->l=sqrt(dx^2 + dy^2)
l =
       9.2195445
```

Note that **x1**, **y1**, **x2** and **y2** are input values and the length '**L**' is the output value. To write the function, proceed as described below:

1. Open SciPad by clicking on **Application** ⇨ **Editor** on the main menu.
2. Type the following lines of code into SciPad and define the function **len()**:

   ```
   function [L] = len(x1, y1, x2, y2)
     dx = x2-x1; dy=y2-y1;
     L = sqrt(dx^2 + dy^2);
   endfunction
   ```

   In the code above, **function** and **endfunction** are Scilab keywords and must be typed exactly as shown. They signify the start and end of a function definition.

   The variable enclosed between square brackets is an output parameter, and will be returned to Scilab workspace (or to the parent function from where it is invoked). The name of the function has been chosen as **len** (short for length, as the name length is already used by Scilab for another purpose). The input parameters **x1**, **y1**, **x2** and **y2** are the variables bringing in input from the Scilab workspace (or the parent function from where it is called).
3. Save the contents of the file to a disk file by clicking File ⇨ Save in SciPad and choose a folder and name for the file. Note both the name and folder for later use.
4. Load the function into Scilab by clicking on **Execute** ⇨ **Load into Scilab** on the SciPad main menu.

This function can be invoked from Scilab prompt as follows:

```
-->ll = len(xx1, yy1, xx2, yy2)
```

where **ll** is the variable to store the output of the function and **xx1**, **yy1**, **xx2** and **yy2** are the input variables. Note that the names of the input and output variables need not match the corresponding names in the function definition.

With version 5 of Scilab, output from statements in a function are never echoed, irrespective of whether the statements end with the semicolon or not. Use the function **disp()** to print out intermediate results in order to debug a function.

To use the function during subsequent sessions, load it into Scilab by clicking on **Execute ⇨ Load into Scilab** in the main menu. If there are no syntax errors in your function definition, the function will be loaded into Scilab workspace, otherwise the line number containing the syntax error is displayed. If it is loaded successfully into the Scilab workspace, you can verify it with the command **whos -type function** and searching for the name of the function, namely, **len**.

We can improve the above function by representing a point as an array of size 1x2, where the elements represent the x and y coordinates of the point. This way, the function can be rewritten as follows:

```
function [L] = len(p1, p2)
  L = sqrt(sum((p2 – p1).^2))
endfunction
```

The modified function can be used as follows:

```
-->p1 = [0, 0]; p2 = [4, 3];
-->ll = len(p1, p2)
ll =
     5.
```

Scilab allows the creation of in-line functions and are especially useful when the body of the function is short. This can be done with the help of the function **deff()**. It takes two string parameters. The first string defines the interface to the function and the second string that defines the statements of the function. Several statements can be separated by semi colons and written as a single string.

The above modified function to calculate the length could be defined in-line as follows:

```
-->deff('[L]=len(p1, p2)', 'L=sqrt(sum((p2-p1).^2))');
```

Scilab allows the user to call a function with fewer input parameters than the number of input arguments in the function definition. It is an error to provide more input parameters than the number of input arguments. The same is the case with output parameters and output arguments.

It is possible to inquire and find the number of input and output arguments from code within the function. The function **argn()** returns the number of input and output arguments, which in turn can be used inside the function body to take decisions such as assigning default values in case the input argument has not been supplied. Using variables that have not been supplied as arguments results in an error and the function being aborted.

```
function [a, b, c] = testarg(x, y, z)
  [out, inp] = argn(0);
  a = 0; b = 0; c = 0;
  mprintf("Output: %d\n", out);
  mprintf(" Input: %d\n", inp);
endfunction
```

We can now load the above function into Scilab workspace and call it with different number of input and output parameters and see how Scilab responds:

```
-->[a, b, c] = testarg(1, 2, 3);
Output: 3
 Input: 3
-->[a, b, c] = testarg(1, 2);
Output: 3
 Input: 2
-->[a, b] = testarg(1);
Output: 2
 Input: 1
-->testarg(1);
Output: 1
 Input: 1
-->testarg(1, 2, 3, 4);
                    !--error 58
Wrong number of input arguments
Arguments are :
x    y    z
-->[a,b,c,d]=testarg(1,2)
                       !--error 59
Wrong number of output arguments
Arguments are :
a    b    c
```

Supplying fewer arguments than required is acceptable as long as that variable is not being used within the function, but then specifying an input variable and not using it would be pointless. But, checking the number of input arguments and assigning default values to those not supplied is a good use of the **`argn()`** function.

Supplying more arguments than required is an error and function execution is aborted.

Supplying fewer output arguments than required is acceptable, but the values of variables that are not caught are lost for subsequent use.

Supplying more output arguments than required is an error and function execution is aborted.

Tutorial 12 – File Operations

Scilab can operate on files on disk and functions are available for opening, closing reading and writing disk files. The following lines of code illustrate how this can be accomplished:

```
-->n=10; x=25.5; xy=[100 75;0 75; 200, 0];
-->fd=mopen("ex01.dat", "w"); // Opens a file ex01.dat for writing
-->mfprintf(fd, "n=%d, x=%f\n", n, x);
-->mfprintf(fd, "%12.4f\t%12.4f\n", xy);
-->mclose(fd);
```

You can now open the file **ex01.dat** in a text editor, such as notepad and see its contents. You will notice that the commands are similar to the corresponding commands in C, namely, **fopen()**, **fprintf()** and **fclose()**. Note that in the second command, the format string must be sufficient to print one full row of the matrix **xy**. The sequence of operations must always be, open the file and obtain a file descriptor, write to the file using the file descriptor and close the file using the file descriptor.

The different modes in which a file can be opened are

r	For reading from the file. The file must exist. If the file does not exist mopen return error code -2.
w	For writing to the file. If the file exists, its contents will be erased and writing will begin from the beginning of the file. If the file does not exist, a new file will be created.
a	For appending to the file. If the file exists, it will be opened and writing will start from the end of the file. If the file does not exist, a new one will be created.

The functions **mprintf()** and **msprintf()** are similar to **mfprintf()**, except that they send the output to the standard output stream and a designated string, respectively, instead of to a file. The statements to print the above data to the standard output (Scilab text window) are as follows:

```
-->mprintf("n=%d, x=%f\n", n, x);
-->mprintf("%12.4f\t%12.4f\n", xy);
```

Being a standard file, it is not necessary to explicitly open and close the standard output file. To output the same data to the strings **s1** and **s2**, use the following commands:

```
-->s1 = msprintf("n=%d, x=%f\n", n, x);
-->s2 = msprintf("%12.4f\t%12.4f\n", xy);
```

Reading data from files into variables in the Scilab workspace is equally easy. The file will have to be opened in read mode and after values have been read from files, the file will have to be closed. The function to read data is **mfscanf()**. The following code fragment reads an integer and a floating point values from the first line in the data file and store them in the variables **n** and **x**, respectively. The second **mfscanf()** statement executed within the loop, reads the values in the subsequent lines into the variable **a**. Following is the code:

```
[fd, err] = mopen('test.dat', 'r');
[n, x] = mfscanf(fd, “%f”);
for i = 1:4
  a(i,:) = mfscanf(fd, “%f %f %f\n”);
end
```

The function **`mfscanf()`** returns **`n`**, the number of values read and the value **`x`** read. In this example **`n`** must be 1. If this value is not 1, which may happen if the file does not contain enough data, it indicates an error. Test the above code with the following data read from the file:

```
-10.0
  10.   2.    5.
   1.   2.    3.
   4.   5.    6.
   7.   8.    9.
  10.  11.   12.
```

After executing the above statements, verify whether the data has been read correctly into the variables **`n`**, **`x`** and **`a`** by displaying their values.

A typical experiment collects data related to one parameter (say **x**) and the corresponding value of a dependent variable (say **y**). These observations can be stored in two vectors, namely, **x** and **y**. Using least square regression, it is possible to compute the coefficients of a polynomial function, of some selected degree, for **y** in terms of **x**. The equation for a polynomial of degree n can be expressed as:

$$y(x)=a_1+a_2x+a_3x^2+\cdots+a_{n+1}x^n=\sum_{i=1}^{n+1}a_ix^{i-1}$$

where $a_i, i=1\,to\,n+1$ are unknown coefficients which can be computed by solving the following set of linear simultaneous equations:

$$\begin{bmatrix} \sum x^0 & \sum x & \sum x^2 & \cdots & \sum x^n \\ \sum x & \sum x^2 & \sum x^3 & \cdots & \sum x^{n+1} \\ \sum x^2 & \sum x^3 & \sum x^4 & \cdots & \sum x^{n+2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum x^n & \sum x^{n+1} & \sum x^{n+2} & \cdots & \sum x^{2n} \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n+1} \end{Bmatrix} = \begin{Bmatrix} \sum y \\ \sum xy \\ \sum x^2y \\ \vdots \\ \sum x^ny \end{Bmatrix}$$

or

$$[X]\{a\}=\{b\}$$

We can solve for the unknown coefficients as follows:

$$a=[X]^{-1}b$$

Once the coefficients are determined, the predicted values of **y** for the observed values of **x** as follows:

$$y(x)=\sum_{i=1}^{n+1}a_ix^{i-1}$$

Let us consider the sample data given below:

x	-1	0	1	2	3	5	7	9
y	-1	3	2.5	5	4	2	5	4

Let us fit a fourth order polynomial equation to the above data, of the form

$$y=a_1+a_2x+a_3x^2+a_4x^3+a_5x^4$$

The required simultaneous equations that must be solved are as follows:

$$\begin{bmatrix} 8 & \sum x & \sum x^2 & \sum x^3 & \sum x^4 \\ \sum x & \sum x^2 & \sum x^3 & \sum x^4 & \sum x^5 \\ \sum x^2 & \sum x^3 & \sum x^4 & \sum x^5 & \sum x^6 \\ \sum x^3 & \sum x^4 & \sum x^5 & \sum x^6 & \sum x^7 \\ \sum x^4 & \sum x^5 & \sum x^6 & \sum x^7 & \sum x^8 \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{Bmatrix} = \begin{Bmatrix} \sum y \\ \sum xy \\ \sum x^2y \\ \sum x^3y \\ \sum x^4y \end{Bmatrix}$$

The elements of the above matrix equation are listed in the table on the next page.

$\sum x^0$	$\sum x^1$	$\sum x^2$	$\sum x^3$	$\sum x^4$	$\sum x^5$	$\sum x^6$	$\sum x^7$	$\sum x^8$
8	26	170	1232	9686	79256	665510	5686952	49208966

Similarly, the right hand vector is

$\sum y$	$\sum xy$	$\sum x^2 y$	$\sum x^3 y$	$\sum x^4 y$
24.5	106.5	676.5	5032.5	39904.5

The simultaneous equations are therefore as follows:

$$\begin{bmatrix} 8 & 26 & 170 & 1232 & 9686 \\ 26 & 170 & 1232 & 9686 & 79256 \\ 170 & 1232 & 9686 & 79256 & 665510 \\ 1232 & 9686 & 79256 & 665510 & 5686952 \\ 9686 & 79256 & 665510 & 5686952 & 49208966 \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{Bmatrix} = \begin{Bmatrix} 24.5 \\ 106.5 \\ 676.5 \\ 5032.5 \\ 39902.5 \end{Bmatrix}$$

Solving these equations gives

$a_1=2.6855$, $a_2=2.3015$, $a_3=-1.2326$, $a_4=0.2169$ and $a_5=-0.0118$

The fourth order polynomial equation is thus

$y=2.6855+2.3015x-1.2326x^2+0.2169x^3-0.0118x^4$

In Scilab, it is not necessary to use loops to compute the elements of the coefficient matrix and right hand vector. Instead, they can be computed through the following matrix multiplication:

```
-->xx = [x.^0 x x.^2 x.^3 x.^4];
-->X = xx' * xx;
-->b = xx' * y;
-->a = inv(X) * b
a =
 2.6855895
 2.3014597
-1.2326305
 0.2168915
-0.0118197
```

The predicted values of y based on this polynomial equation can be computed as follows:

```
-->y(:,2) = xx * a;
```

You can plot the graph of x against observed and predicted values with the following:

```
-->plot(x, y)
```

Now that we know how this calculation is to be done, let us write a function to automate this. Let the function interface be as follows:

Interface: `[a, yf] = polyfit(x, y, n)`
Input Parameters:
`x` = column vector of observed values of the independent variable,
`y` = column vector of observed values of dependent variable,

n = degree of the polynomial fit
Output parameters:
a = column vector of coefficients of the polynomial of order n that minimizes the least squares error of the fit, $a_i, i=1,2,\ldots,n+1$ where a_i is the coefficient of the term x^{i-1} . **a** is a column vector.
yf = a column vector of calculated values of **y** for a polynomial of order **n**.

The function **polyfit()** can be written in SciPad and then loaded into Scilab workspace:

```
function [a, yf] = polyfit(x, y, n)
  xx = zeros(length(x), n+1);
  for i = 1:n+1
    xx(:,i) = x.^(i-1);
  end
  X = xx' * xx;
  b = xx' * y;
  a = inv(X) * b;
  yf = xx * a;
endfunction
```

Once the function is successfully loaded into the Scilab workspace, to fit a fourth order polynomial, you can call it as follows:

```
-->[a4, y(:,2)] = polyfit(x, y, 4);
-->[a5, y(:,3)] = polyfit(x, y, 5);
-->plot(x, y, x, yf); xtitle('POLYNOMIAL CURVE FITTING', 'x', 'y');
```

The first line calculates the coefficients **a** of the polynomial fit and the calculated values **yf** for a 4th order polynomial fit and the second line does the same for a 5th order polynomial fit. The calculated values of **y** for each are stored in the 2nd and 3rd columns of **y**. The third line plots the graph of raw data and fitted values and adds a title for the graph and labels for the x and y axes.

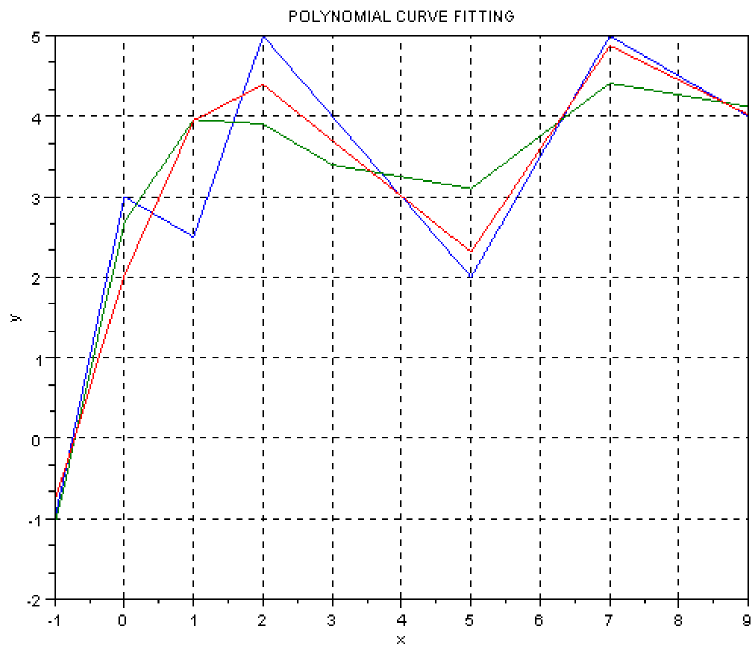
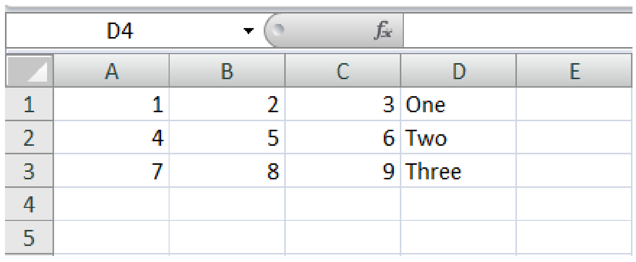


Fig. 13.1 Polynomial curve fitting by least square method

Scilab provides a set of functions to read Microsoft Excel files. Scilab version 5.1.1 can read Excel files up to version 2003 (.xls), but not Microsoft Excel 2007 (.xlsx) files. The steps to read an Excel file are: (i) open the file with **xls_open()** function, (ii) read the file contents with the **xls_read()** function and (iii) close the file with the **mclose()** function. Before opening the file, check the current working directory and change it if necessary. Alternatively, note the complete path to the file and use it to define the full path to the file location.



D4

	A	B	C	D	E
1	1	2	3	One	
2	4	5	6	Two	
3	7	8	9	Three	
4					
5					

Assuming the cells **A1:C3** in **Sheet1** contain data as shown in the adjoining figure, **Value** is a 3x4 matrix containing, **TextInd** is a 3x4 matrix and **SST** is a 1x3 matrix as shown in the output above.

Following is a simple example of how you can open and read an Excel file, assuming that the file is named **test.xls** and is located in the current working directory:

```
-->[fd, sst, sheetnames, sheetpos] = xls_open('test.xls');
-->[value, text] = xls_read(fd, sheetpos(1));
-->mclose(fd);
-->disp(value)
   1.    2.    3.    Nan
   4.    5.    6.    Nan
   7.    8.    9.    Nan
-->disp(text)
   0.    0.    0.    1.
   0.    0.    0.    2.
   0.    0.    0.    3.
-->disp(sst)
!   One    Two    Three  !
```

The function **xls_open()** returns four values, **fd** is the file descriptor that is required during subsequent operations on the file, **SST** is the vector of all strings in the file, **Sheetnames** is the vector of names of all sheets in the file and **Sheetpos** is a vector of numbers indicating the beginning of different sheets within the file. The function **xls_read()** requires the file descriptor **fd** and the **sheetpos** of the sheet to be read, previously obtained from **xls_open()** function call. To read the contents of the first sheet, use **sheetpos(1)**. This function returns two values, **Value** is a matrix of containing all numerical values in the sheet with **Nan** (Not a number) in the cell positions containing non-numeric data and **text** is a matrix containing zeros corresponding to cells containing numeric data and non-zero values corresponding to string data. The non-zero integers in **text** point to the corresponding string value stored in the variable **sst** returned by the **xls_open()** function call.

There is another function **`readxls()`** that can read Excel files. It is simpler to use compared to the **`xls_open()`**, **`xls_read()`** and **`mclose()`** combination.

```
 -->sheets = readxls('test.xls');
-->s1 = sheets(1);
-->a = s1.value
a =
!  1.   2.   3.   One   !
!  4.   5.   6.   Two   !
!  7.   8.   9.   Three !
-->s1.text
-->ans =
!     One   !
!     Two   !
!     Three !
```

The Scilab variable editor **`editvar`** can show the values read by **`readxls()`** function and also allow editing and updating of values to Scilab workspace.

```
-->editvar s1
```

Some commands related to operations on disks are important and they are listed below. They are required when you want to change the directory in which your function files are stored or from where they are to be read.

`pwd`	Prints the name of the current working directory
`getcwd()`	Same as **`pwd`**.
`chdir('dir')`	Changes the working directory to a different disk location named **`dir`**.

In version 5.1.1, changing directory has become easy and visual. In the main menu, go to **File ⇨ Change current directory...** and visually browse to the folder which you want to make the current directory and click **OK**. The error prone task of typing lengthy folder paths is no longer necessary.

It is possible to save all the variables in the Scilab Workspace to a disk file so that you can quickly reload all the variables and functions from a previous session and continue from where you left off. The commands used for this purpose are:

`save('pf.bin')`	Saves entire contents of the Scilab workspace (variables and functions) in the file **`pf.bin`** in the current working directory.
`load('pf.bin')`	Restores contents of the Scilab workspace from the file **`pf.bin`** in the current working directory.
`save('pf.bin', xy)`	Saves only the variable **`xy`** of the Scilab workspace in the file **`pf.bin`** in the current working directory.

It is possible to determine the size of variables in the Scilab workspace with the following command:

`size(x)`	Returns a 1x2 matrix containing the number of rows and columns in the matrix **`x`**.
`length(x)`	Returns the number of elements in matrix **`x`** (number of rows multiplied by number of columns).
`clc`	Clear the screen. Alternately, you can press the **`F2`** function key

Scilab maintains a history of the commands typed at the command prompt. You can scroll up and down the command history with the up arrow (↑) and down arrow (↓) keys on the keyboard. You can move to the left and right along one of the previous commands from the command history with the left arrow (←), right arrow (→), Home and End keys on the keyboard. You can delete any of the characters with the Delete or Backspace keys. Insert new characters. You can press the Enter key to complete the command.

Scilab has support for data structures, such as **`cell`**, **`hypermatrices`**, **`list`**, **`mlist`**, **`rlist`**, **`tlist`** and **`struct`**. This tutorial will touch upon some of them, namely, **`list`**, **`cell`** and **`struct`**.

List

The list data structure allows operations such as insertion, deletion, concatenation.

```
-->lst=list(1, [1, 2; 3, 4], [“a”, “b”, “c”]); // create a list
-->lst(1) // extract first element of lst
ans =
   1.
-->lst(2)
ans =
  1.   2.
  2.   4.
-->lst(3)
ans =
!a  b  c!
-->size(lst)
ans =
   3.
-->lst($) // extract last element of lst
ans =
!a  b  c!
-->lst($+1) = “Append to list”; // append to lst
-->lst($)
ans =
Append to list
-->lst(0) = “Insert before list”;
list(1)
ans =
Insert before list
```

Cell

The cell data structure allows the creation of matrices in which each element can be of different type. Normally, all elements of a matrix must be of the same data type. Thus a matrix cannot have one element as a number while another element is a string. The cell data structure does not have this limitation. In fact, each element in turn can be another data structure, thereby allowing construction of complex data structures.

```
-->a = cell(3) // create a cell with 3 rows and 3 columns
a =
!{}  {}  {} !
!           !
```

```
!{}  {}  {} !
!           !
!{}  {}  {} !
-->b = cell(3,1); // create a cell with 3 rows and 1 column
-->c = cell(2, 3, 4); // create a cell with 2 rows, 3 columns and 4
cards
-->b(1).entries = 1:3;
-->b(2).entries = “Scilab”;
-->b(3).entries = [1 2; 3 4];
-->b
b =
![1 2 3]        !
!               !
!”Scilab”       !
!               !
!{2x2 constant} !
-->b(1)
ans =
[1, 2, 3]
-->b(2)
ans =
“Scilab”
-->b(3)
ans =
ans =
{2x2 constant)
-->b(3).entries
ans =
  1.  2.
  3.  4.
-->b.dims
ans =
  3  1
-->size(b)
ans =
   3.   1.
```

Struct

The **`struct`** data structure allows the creation of user defined data types with user defined field names. The **`struct`** is analogous to the **`struct`** in C programming language, but is more flexible in that the new fields can be created even after the **`struct`** has been created. Each field can store any type of Scilab data, including matrices, cells, lists or even other **`struct`**s.

```
-->dt = struct('date', 15, 'month', 'Aug', 'year', 1947)
dt =
day: 15
month: “Aug”
year: 1947
```

The above data structure contains three fields, namely, day, month and year. While day and year store numeric data, month stores string data. We can access the individual fields with the . (dot) operator.

```
-->dt.day
ans =
  15.
-->dt.month
ans =
Aug
-->dt.year
ans =
  1947
```

We can create new fields by simply assigning data to a new filed name.

```
-->dt.desc = “Independence day”
dt =
day: 15
month: “Aug”
year: 1947
desc: “Independence day”
```

A struct need not be explicitly created using the **`struct()`** function and a field can also contain matrices.

```
-->s.type = “Space”;
-->s.unit.length = “m”;
-->s.unit.force = “kN”;
-->s.nodes = [1 0 0 0; 2 0 5 0; 3 8 5 0; 4 8 0 0];
-->s
s =
type: “Space
unit: [1x1 struct]
nodes: [4x4 constant]
-->s.unit
ans =
    length: “m”
    force: “kN”
-->s.nodes
ans =
  1.  0.  0.  0.
  2.  0.  5.  0.
  3.  8.  5.  0.
  4.  8.  0.  0.
```

Data structures help organize data in a hierarchical manner and simplify passing of data from one function to another. It now becomes possible to pack data into a single object and unpack the object to extract individual fields. Thus passing a single object as the argument to a function gives access to all the individual fields within the object inside it. Previously, we would have to pass each field of the data structures as an individual argument.

As an illustration, let the different attributes representing a structure be (i) type of structure (possible values are space, plane, truss, grid etc.), (ii) node coordinates (a matrix with 3 columns), (iii) member connectivity (matrix with 3 columns) and (iv) material (a matrix with 3 columns). To pass the information about a structure to a function, we will have to use four arguments. Instead, if we define a data structure with the four attributes as its fields, we will need to pass only one argument to the function, and within the function, we can access the individual fields.

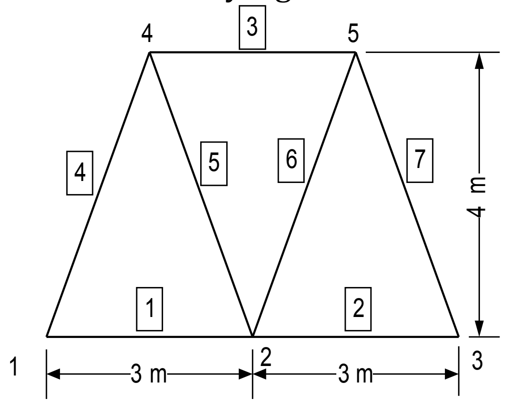
Problem Definition

It is intended to represent a two dimensional truss in matrix notation and develop functions to calculate the following:

1. function to calculate the length of a member given its Id
2. function to calculate the projections along x- and y-axes and angle made by a member with the x-axis
3. function to list the start and end nodes of a specified member
4. function to calculate the total length of all members
5. function to list the members meeting at a node
6. function to calculate the number of members meeting at each joint

Data Definition

Nodes of a truss have x and y coordinates. Interconnectivity of members is expressed in terms of a connectivity matrix. The connectivity matrix represents the numbers of the nodes connected by a given member.



Node Coordinates

Node	x	y
1	0.0	0.0
2	3.0	0.0
3	6.0	0.0
4	1.5	4.0
5	4.5	4.0

Member Connectivity

Member	Node i	Node j
1	1	2
2	2	3
3	4	5
4	1	4
5	2	4
6	2	5

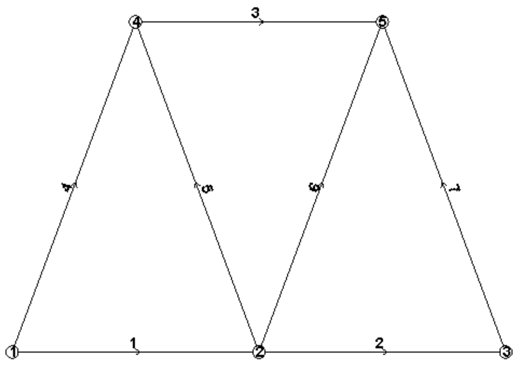
7 3 5

This data can be input into two variables, **xy** and **conn**, as follows:

```
-->xy = [0 0; 3 0; 6 0; 1.5 4; 4.5 4]
-->conn = [1 2; 2 3; 4 5; 1 4; 2 4; 2 5; 3 5]
```

Before attempting to write the functions, let us try out the commands interactively and verify our understanding of the steps. To display the graph of the truss, we must first define the graph data structure with the help of the **make_graph()** function and then use **show_graph()** function.

```
-->tail = conn(:,1)'; head = conn(:,2)';
-->[nodes, dummy] = size(xy);
-->g = make_graph('Truss', 1, nodes, tail, head);
-->g.nodes.graphics.x = xy(:,1)'*100;
-->g.nodes.graphics.y = xy(:,2)'*100;
-->show_graph(g);
```



The above commands produce the figure shown above. Let us now write the functions to complete the various assigned tasks.

Function **rows(x)** returns the number of rows in a matrix **x**.

```
function [r] = rows(x)
  [r,dummy] = size(x);
endfunction
```

Function **get_nodes(id, conn)** returns the numbers of the nodes **n1** and **n2** at the ends of member with number **id**.

```
function [n1, n2] = get_nodes(id, conn)
  n1 = conn(id, 1); // Column 1 of connectivity matrix stores start
node
  n2 = conn(id, 2); // Column 2 of connectivity matrix stores end
node
endfunction
```

Function **len(id, xy, conn)** returns the length **L** of a member.

```
function [L] = len(id, xy, conn)
  [n1, n2] = get_nodes(id, conn);
  p1 = xy(n1,:); p2 = xy(n2,:); // Coordinates of nodes n1 and n2
  dx = p2 - p1; // Difference of coordinates of n1 and n2
  L =sqrt(sum(dx .^2)); // Formula for length from analytical
geometry
endfunction
```

Function **`projections(id, xy, conn)`** returns the x and y projections **`dx`** and **`dy`** of a member as well as the angle **`theta`** made by the member with the x-axis.

```
function [dx, dy, theta] = projections(id, xy, conn)
  [n1, n2]=get_nodes(id, conn);
  p1 = xy(n1,:); p2 = xy(n2,:);
  difference = p2 - p1;
  dx = difference(1); // x-projection is in column 1 of diff
  dy = difference(2); // y-projection is in column 2 of diff
  if dx == 0 then
    theta = %pi/2;
  else
    theta = atan(dy/dx);
  end
endfunction
```

Function **`total_length(xy, conn)`** returns the total length of all members in a truss.

```
function [tot_L]=total_length(xy, conn)
  members = rows(conn);
  for id=1:members
    L(id)=len(id,xy,conn);
  end
  // disp(L);
  tot_L = sum(L);
endfunction
```

Function **`members_at_node(nodeid, conn)`** returns the numbers of members meeting at a node.

```
function [memlst] = members_at_node(nodeid, conn)
  memlst = find( (conn(:,1)==nodeid) | (conn(:,2)==nodeid) );
endfunction
```

Function **`num_members_at_node(nodeid, conn)`** returns the number of members meeting at node.

```
function [n] = num_members_at_node(nodeid, conn)
  memlst = members_at_node(nodeid, conn);
  n = length(memlst);
endfunction
```

Function **`make_truss(name, xy, conn)`** prepares the data structure to represent a truss.

```
function [g] = make_truss(name, xy, conn, scale)
  t = conn(:,1)'; // tail node numbers
  h = conn(:,2)'; // head node numbers
  g = make_graph(name, 1, rows(xy), t, h);
  g.nodes.graphics.x = xy(:,1)' * scale;
  g.nodes.graphics.y = xy(:,2)' * scale;
endfunction
```

Using the data previously entered, we can test the above functions as shown below:

```
-->g = make_truss('Truss', xy, conn, 100);
-->show_graph(g)
```

The other functions can also be tested as follows:

```
-->tot_len = total_length(xy, conn); disp(tot_len)
26.088007
-->for i = 1:rows(xy)
-->  n = num_members_at_node(i, conn);
-->  mem_lst = members_at_node(i, conn);
-->  disp([n mem_list])
-->end
   2.   1.   4.
   4.   1.   2.   5.   6.
   2.   2    7.
   3.   3    4.   5.
   3.   6.   7.
```

We can also test the length and projection functions in a single **`for`** loop:

```
-->for i = 1:rows(conn)
-->  L = len(i, xy, conn);
-->  [dx, dy, theta] = projections(i, xy, conn);
-->  disp([dx dy theta L])
-->end
  3.   0.  0.         3.
  3.   0.  0.         3.
  3.   0.  0.         3.
  1.5  4.  1.2120257  4.2720019
 -1.5  4. -1.2120257  4.2720019
  1.5  4.  1.2120257  4.2720019
 -1.5  4. -1.2120257  4.2720019
```

The node numbers and member numbers can be displayed on the graph using the View ⇨ Options from the main menu of the graphics window. Graph settings can be modified from the Graph ⇨ Settings from the main menu. Graph settings that can be changed are node diameter, edge width, border node width, edge colour index etc.

In the theory of elasticity, the stress components at a point are represented as a matrix consisting of 3 normal stresses on the diagonal and three shear stresses off the diagonal. The condition of equilibrium requires that the matrix be symmetric. A typical stress matrix is written as follows:

$$[\sigma]=\begin{bmatrix}\sigma_x & \tau_{xy} & \tau_{xz}\\ \tau_{xy} & \sigma_y & \tau_{yz}\\ \tau_{xz} & \tau_{yz} & \sigma_z\end{bmatrix}$$

The principal planes are defined as those planes on which only normal stresses act and are free of shear stresses. They also represent the maximum values of the normal stress at that point. The condition for a plane to be a principal plane and the stresses on it to be the principal stress is expressed in the form of a homogeneous equation:

$$\begin{bmatrix}(\sigma_x-\sigma) & \tau_{xy} & \tau_{xz}\\ \tau_{xy} & (\sigma_y-\sigma) & \tau_{yz}\\ \tau_{xz} & \tau_{yz} & (\sigma_z-\sigma)\end{bmatrix}\begin{Bmatrix}l_x\\ l_y\\ l_z\end{Bmatrix}=\begin{Bmatrix}0\\0\\0\end{Bmatrix}$$

where σ is the principal stress and l_x, l_y and l_z are the direction cosines of the outward normal to the principal plane. It is required to determine the principal stresses and the direction cosines of the principal planes. The characteristic equation of the above set of homogeneous equations is $\sigma^3-I_1\sigma^2+I_2\sigma-I_3=0$. Being a cubic equation, it has three roots and thus there are three values of principal stresses. Corresponding to each principal stress is one principal plane represented by the direction cosines of its outward normal. In mathematics, this is an eigenvalue problem, the eigenvalues are the principal stresses and eigenvectors are the direction cosines of the outward normal to the principal planes.

The principal stresses and principal planes can thus be obtained with the **spec** function in Scilab. As an example, let the stress matrix be as follows:

$$[\sigma]=\begin{bmatrix}3 & 2.5 & 4\\ 2.5 & 1 & 0\\ 4 & 0 & 2\end{bmatrix}$$

Then the solution can be obtained as follows:

```
-->s = [3, 2.5, 4; 2.5, 1, 0; 4, 0, 2];
-->[x, p] = spec(s)
p =
  -2.4316605   0.          0.
   0.          1.2968769   0.
   0.          0.          7.1347836
x =
   0.6529665  -0.0979278   0.7510293
  -0.4756928  -0.8246499   0.3060537
  -0.5893651   0.5571020   0.5850523
```

The eigenvalues are on the diagonal of the matrix **p**. The columns of eigenvector **x** correspond to the eigenvalue on the diagonal. Thus the first column of **x** corresponds to the principal stress value **-2.4316605**. The eigenvalues are arranged in increasing order whereas the principal stresses are, by convention, arranged in decreasing order. Thus the first principal stress is $\sigma_1=7.1347836$, second principal stress is $\sigma_2=1.2968769$ and the third principal stress is $\sigma_1=-2.4316605$. The first principal plane is defined by the direction cosines $l_x=0.7510293$, $l_y=0.3060537$ and $l_z=5850523$.

The solution can also be obtained by an alternative way, using polynomials. In this approach the eigenvalues can be be obtained easily but obtaining the eigenvectors may involve a few additional manual calculations. But it has the advantage that we can obtain the characteristic equation, which is not available in the first solution. Here, let us first define a seed for a polynomial as **p = poly(0, 'p')** and use it to represent the homogeneous equations as follows:

```
-->p = poly(0, 'p');
-->ss = s - eye(s) * p;
-->ch = det(ss)
ch =
                      2    3
    -22.5 + 11.25p + p  - p
-->roots(ch)
ans =
   1.2968769
  -2.4316605
   7.1347836
```

To obtain the eigenvector corresponding to a chosen eigenvalue, we must substitute the value of the eigenvalue into the homogeneous equations, and choose any two of the three equations and solve them for two of the direction cosines in terms of the third.

$$\begin{bmatrix}(\sigma_y-\sigma_i) & \tau_{yz}\\ \tau_{yz} & (\sigma_z-\sigma_i)\end{bmatrix}\begin{Bmatrix}l_y\\ l_z\end{Bmatrix}=l_x\begin{Bmatrix}-\tau_{xy}\\ -\tau_{xz}\end{Bmatrix}$$

The above equations can be solved to obtain l_y and l_z in terms of l_x. Let $l_y=k_y l_x$ and $l_z=k_z l_x$. Then using the fact that the sum of the square of the direction cosines is unity, all the three direction cosines can be obtained.

$$l_x^2+l_y^2+l_z^2=1 \text{ or } (1+k_y^2+k_z^2)l_z^2=1$$

Rearranging and solving we get:

$$l_x=\frac{1}{\sqrt{1+k_y^2+k_z^2}},\quad l_y=k_y\frac{1}{\sqrt{1+k_y^2+k_z^2}} \text{ and } l_z=k_z\frac{1}{\sqrt{1+k_y^2+k_z^2}}$$

We could write our own functions to do these calculations, taking care to sort the eigenvalues before proceeding with the eigenvector calculations.