Scilab
A Hands on Introduction

by
Satish Annigeri Ph.D.
Professor of Civil Engineering
B.V. Bhoomaraddi College of Engineering & Technology, Hubli
satish@bvb.edu

July 2009

Table of Contents

Preface

Scilab is a software for numerical mathematics and scientific visualization. It is capable of interactive calculations as well as automation of computations through programming. It provides all basic operations on matrices through built-in functions so that the trouble of developing and testing code for basic operations are completely avoided. Its ability to plot 2D and 3D graphs helps in visualizing the data we work with. All these make Scilab an excellent tool for teaching, especially those subjects that involve matrix operations. Further, the numerous toolboxes that are available for various specialized applications make it an important tool for research. Being compatible with Matlab®, all available Matlab M-files can be directly used in Scilab. Scicos, a hybrid dynamic systems modeler and simulator for Scilab, simplifies simulations. The greatest features of Scilab are that it is multi-platform and is free. It is available for many operating systems including Windows, Linux and MacOS X. More information about the features of Scilab are given in the Introduction.

Scilab can help a student understand all intermediate steps in solving even complicated problems, as easily as using a calculator. In fact, it is a calculator that is capable of matrix algebra computations. Once the student is sure of having mastered the steps, they can be converted into functions and whole problems can be solved by simply calling a few functions. Scilab is an invaluable tool as solved problems need not be restricted to simple examples to suit hand calculations.

It is not the aim of this tutorial to be an exhaustive and in-depth look into Scilab. Instead, it attempts to get a novice started with the least fuss and is aimed at anyone who intends to start learning to use Scilab entirely on her own.

In this revision of the tutorial, exercises have been added to point the student to those aspects that were not explicitly covered by the tutorial and lead the student towards self-discovery and learning.

Acknowledgements

It goes without saying that my first indebtedness is to the developers of Scilab and the consortium that continues to develop it. I must also thank Dr. A.B. Raju, E&EE Department, BVBCET, Hubli who first introduced me to Scilab and forever freed me from using Matlab.

July 2009 *Satish Annigeri*

Introduction

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, `http://www.inria.fr/index.en.html`) and ENPC (National School of Bridges and Roads, `http://www.enpc.fr/english/int_index.htm`), it is now maintained and developed by Scilab Consortium (`http://scilabsoft.inria.fr/consortium/consortium.html`) since its creation in May 2003.

Distributed freely and open source through the Internet since 1994, Scilab is currently being used in educational and industrial environments around the world.

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.

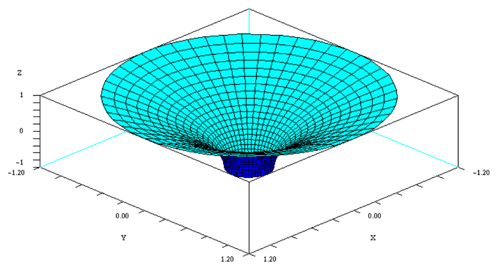
A number of toolboxes are available with the system:

- 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. It comes with source code, on-line help and English user manuals. Binary versions are available.

Some of its features are listed below:

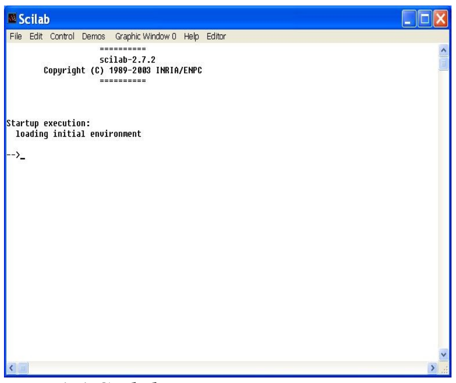
- Basic data type is a matrix, and all matrix operations are available as built-in operations.
- Has a built-in interpreted high-level programming language.
- Graphics such as 2D and 3D graphs can be generated and exported to various formats so that they can be included into documents.



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.

Tutorial 1 – Scilab Environment

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**. 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.



```
                   ==========
                   scilab-2.7.2
         Copyright (C) 1989-2003 INRIA/ENPC
                   ==========



Startup execution:
  loading initial environment

-->_
```

Fig. 1.1 Scilab environment

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
```

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.
```

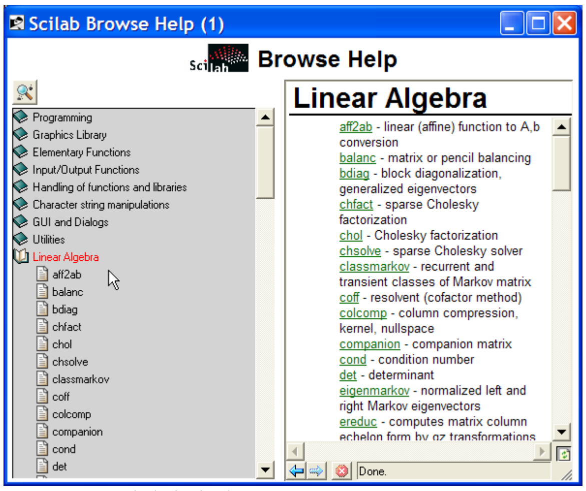
```
-->pi=atan(1.0)*4
pi =
      3.1415927
-->sin(pi/4)
ans =
      0.7071068
-->exp(0.1)
ans =
      1.1051709
```

Usually the answer of a calculation is stored in a variable so that it could be used later. If you do not explicitly supply the name of the variable to store the answer, Scilab uses a variable named "**ans**" to store such results.

You could enter more than one command on the same line by separating the commands by semicolons (**;**). The semicolon suppresses echoing of intermediate results. Try the command

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

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



Linear Algebra

aff2ab - linear (affine) function to A,b conversion
balanc - matrix or pencil balancing
bdiag - block diagonalization, generalized eigenvectors
chfact - sparse Cholesky factorization
chol - Cholesky factorization
chsolve - sparse Cholesky solver
classmarkov - recurrent and transient classes of Markov matrix
coff - resolvent (cofactor method)
colcomp - column compression, kernel, nullspace
companion - companion matrix
cond - condition number
det - determinant
eigenmarkov - normalized left and right Markov eigenvectors
ereduc - computes matrix column echelon form by qz transformations

Fig. 1.2 Scilab help browser

Exercise 1 – Scilab Environment

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 5.1.1 so that the folder name is scilab-5.1.1. The above location is for Microsoft Windows XP. For Windows Vista, use C:\Users\<User>\AppData\Roaming\Scilab\<scilab-<version>\scilab.ini).

Tutorial 2 – The Workspace and Working Directory

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**. Many a times 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
`-->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.

Exercise 2 – The Workspace and Working Directory

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?).
10. Create a polynomial with **`x`** as the symbolic variable such that its roots are 2 and 3.
    (**Ans:** **`p = poly([2 3], 'x')`**).
11. Create a polynomial to represent $6 - 5x + x^2$?
    (**Ans:** **`p = poly([6 -5 1], 'x', 'coeff')`**)
12. Find the roots of this polynomial. (**Ans:** 2 and 3).
13. 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).

Tutorial 3 – Matrix Operations

Matrix operations built-in into Scilab include addition, subtraction, multiplication, transpose, inversion, determinant, trigonometric, logarithmic, exponential functions and many others. Study the following examples:

`-->a=[1 2 3; 4 5 6; 7 8 9];`	Define a 3x3 matrix. Semicolons indicate end of a row
`-->b=a';`	Transpose `a` and store it in `b`. Apostrophe (`'`) is the transpose operator.
`-->c=a+b`	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`	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)`	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)`	Determinant of `f`.
`-->log(a)`	Matrix of log of each element of `a`.
`-->a .* b`	Element by element multiplication.
`-->a^2`	Same as `a*a`.
`-->a .^2`	Element by element square.

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.

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

`-->a=[1:5]`	Creates a vector with 5 elements as follows [1, 2, 3, 4, 5]
`-->b=[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 the short form for `a:1:b`, where `a` and `b` are numbers). The increment must be negative when the start value is greater than the end value.

You can create an empty matrix with the command:

```
-->a=[]
```

Exercise 3 – Matrix Operations

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 π/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 2 3 4 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 2 3 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 2 3 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 2 3 4 5]) + diag([6 7 8 9], 1) + diag([10 11 12 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).

Tutorial 4 – Sub-matrices

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=rand(5,8)*100`	Generates a 5x8 matrix whose elements are generated as random numbers.

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.

While using range to demarcate rows and/or columns, it is permitted to leave out both the start and end value 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`, the command is `a(:, 2:3)`. Naturally `a(:, :)` represents the whole matrix, which of course could be represented simply as `a`.

However, 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. 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 must also be noted 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.

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

Tutorial 5 – Statistics

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 above statistical functions, except that you must use **`r`** or **`c`** but not **`1`** or **`2`**.

Tutorial 6 – Plotting Graphs

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.

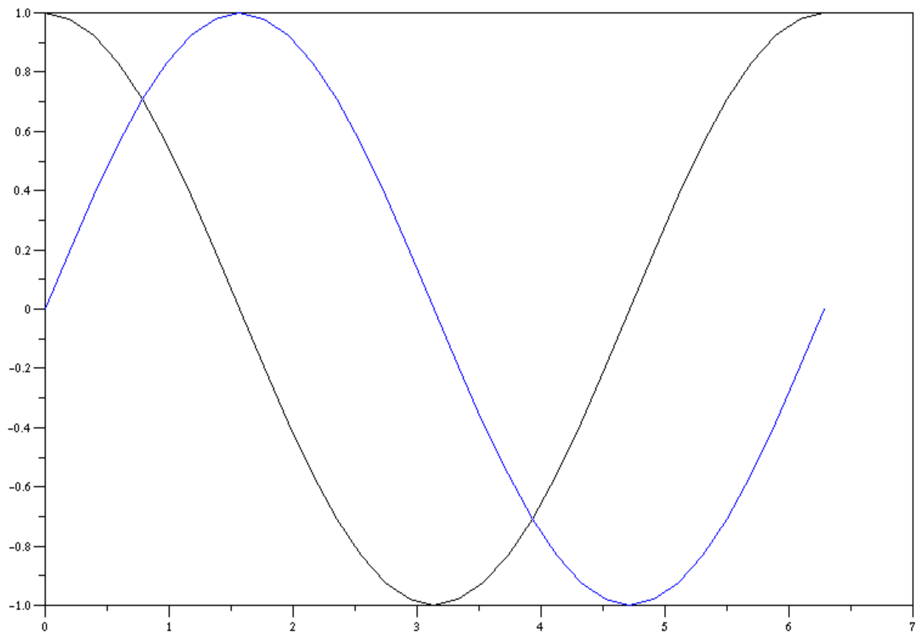


Fig. 6.1 Graph of sin(x) and cos(x) using function **`plot2d()`**

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.

You can also 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 press the GED button on the toolbar of the graph window (Fig. 3).

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).

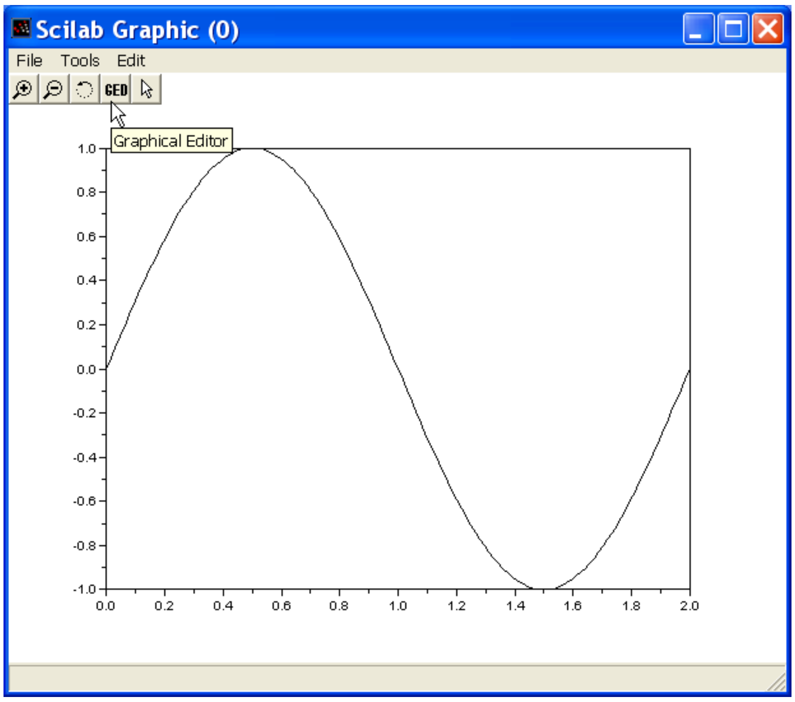
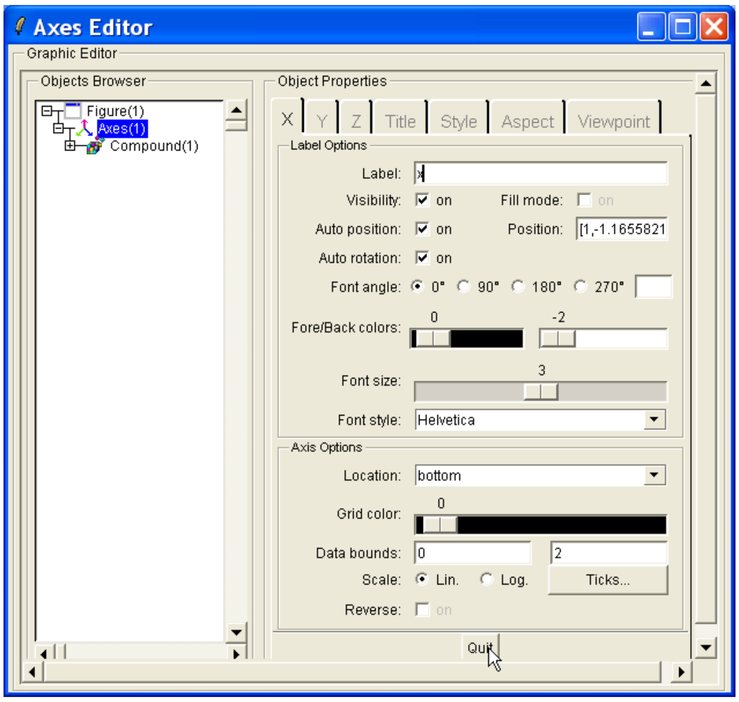


Fig. 6.2 Customization of graphs interactively

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



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 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.

Tutorial 7 – Plotting 3D Graphs

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)
-->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 $\pi/16$ 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 command on the second line 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 in 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 a 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 I spossible 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 subplot(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.

Tutorial 8 – Scilab Programming Language

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. 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
<	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=rand(3,4)*10`	Generate matrix `a` with 3 rows and 4 columns and containing random numbers between 0 and 10.
`-->a == 0`	Creates 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 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 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
```

Tutorial 9 – Script Files and Function Files

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 or keyboard short cut Ctrl + L. Before being loaded into Scilab workspace, the file is checked for syntax, and if here syntax error, code is not loaded into Scilab workspace, and error messages are displayed in the Scilab window. If there are no error, the code is compiled and loaded into Scilab workspace, and become available for use. If changes are made to the code in SciPad, the process of Execute ⇨ Load into Scilab must be repeated. When a 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, usually, do not contain any function definitions and contain only Scilab statements mainly for data input and calculations. 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.

Unlike Matlab or Octave, one function file may contain more than one function definitions 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, but they can be loaded into Scilab workspace only from SciPad. Matlab and Octave have the stipulation that the name of the file must be the same as the name of the function it contains with an extension .m and consequently are able to load functions into their workspace as and when required instead of explicitly being loaded into the workspace by the user. This also enables them to identify if the source code has been modified and recompile and load into the workspace on the fly. 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 the easiest means of loading data into Scilab workspace and file read operations can be completely avoided. Variables in Scilab workspace can be directly read from and written to from inside a script file. While this is easy to understand and use, it can lead to errors in the case of large and complex set of statements.

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.

Tutorial 10 – Functions in Scilab

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 **Application** ⇨ **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 **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 **Load into Scilab** on the SciPad main menu.

This function will be invoked 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 with the semicolon or otherwise. 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`**.

Tutorial 11 – 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 this standard output file. To output the same data to the strings **`s1`** and **`s2`**, commands are as follows:

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

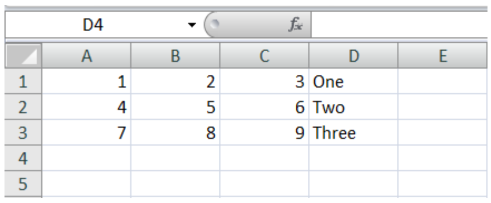
Tutorial 12 – Reading Microsoft Excel Files

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, read the file contents with the **`xls_read()`** function and 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 completely define the path to the file location.

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, TextInd] = xls_read(fd, Sheetpos(1));
-->mclose(fd);
-->disp(Value)
   1.   2.   3.   NaN
   4.   5.   6.   NaN
   7.   8.   9.   NaN
-->disp(TextInd)
   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 indicting 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 **`TextInd`** is a matrix containing zeros corresponding to cells containing numeric data and non-zero values corresponding to string data. The non-zero integers in **`TextInd`** point to the corresponding string value stored in the variable **`SST`** returned by the **`xls_open()`** function call.



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.

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

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

Tutorial 13 – Some Miscellaneous Commands

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`** (rows multiplied by columns).

It is possible to find the number of input and output arguments for a function call. The function **`argn()`** can return the input and output arguments which can be used inside the function body.

```
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

-->[a, b, c] = testarg(1, 2, 3);
Output: 3
 Input: 3
-->[a, b, c] = testarg(1, 2);
Output: 3
 Input: 2
-->[a, b, c] = testarg(1);
Output: 3
 Input: 1
```

The output will vary depending on how the function is invoked and not on how the function is defined. A function call can have fewer parameters than the number of input and output arguments in the function definition, but it is an error to use more parameters than the corresponding number of arguments in the function definition.

Tutorial 14 – Mini Project

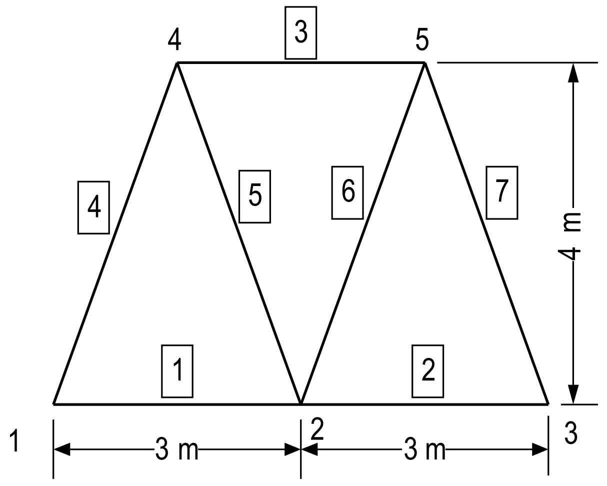
Problem Definition

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

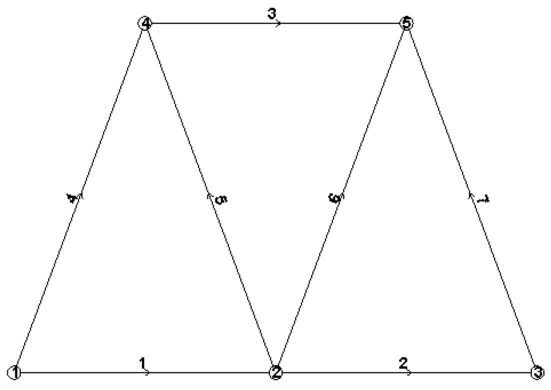
Member	Node i	Node j
1	1	2
2	2	3
3	4	5
4	1	4
5	2	4
6	2	5
7	3	5

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) plots the graph of the 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
```