# PROGRAMMING USING SCILAB

## THEORY & PRACTICALS

For B.Sc. Course of Pondicherry University

**AKHILESH KUMAR**

# ACKNOWLWDGEMENTS

In the accomplishment of this book there is contribution of many people in terms of their support, encouragement, and assistance at multiple occasions and phases of writing of the book. I take this opportunity to acknowledge their important contributions with gratitude.

I wish to express my sincere gratitude towards the Dr. N. Viyasarayar, the Principal of Arignar Anna Government Arts & Science College, Karaikal, for his always welcoming behaviour and best wishes to make this book get compiled.

I extend my sincere thanks to Thiru. V. Karuppaiya Pillai, the Head, Department of Mathematics, Arignar Anna Government Arts & Science College, Karaikal for his everlasting support and encouragement for working towards the betterment of students in any possible mannar. I extend my gratitude to Dr. J. Prakash, Assistant Professor, Department of Mathematics, Avvaiyar Governmant College for Women, Karaikal for his inputs on Scilab package on differential equations. The best wishes of all faculty members of the Department of Mathematics and other Departments of Arignar Anna Government Arts & Science College, Karaikal are acknowledged.

A sincere thanks are due to Prof. Aparna Mehra, Professor, Department of of Mathematics, Indian Institute of Technology, Delhi for guiding me at every level of my career in academics and research and enabling me to compile my knowledge in the form of a book.

I wish to record my profound gratitude to my parents, relatives and friends for their all kinds of support and help for myself devoting additional time towards academic activities. Thank you all.

**(Dr. AKHILESH KUMAR)**

**Assistant Professor,**
**Department of Mathematics,**
**Arignar Anna Government Arts & Science College,**
**Karaikal**

*Dedicated to*

*my Parents*

*Shri. Omprakash Shastri and Smt. Promila Shastri*

# CONTENTS

# Preface

This book has come up mainly to help the students of fifth semester of B.Sc. Mathematics course of Pondicherry University for their theory and practical papers on Scilab. I have made my best efforts for writing this book which should enable the students to understand the concepts by self-study itself. For this purpose, there is a plethora of exemplary demonstrations through codes and figures.

The book comprises of seven chapters followed by the bibliography.

The Chapter 1 gives a basic introduction to Scilab giving basic idea on how to install Scilab, it components, and its use for basic mathematical calculations. The use of matrices and matrix operations in Scilab is introduced in Chapter 2. In Chapter 3, fundamentals of programming in Scilab as a programming language are introduced. Chapter 4 discusses on developing Scilab programs as functions for use of the outputs in further computations. Chapter 5 is about plotting of graphics. The Scilab package for numerically solving ordinary differential equations with initial conditions is discussed in Chapter 6. Dealing with polynomials using Scilab is detailed in Chapter 7.

It is suggested for readers of this book to learn each of introduced concepts practically through a side-by-side implementation and experimentation of provided examples and exercises in Scilab. I hope that readers will enjoy the reading and learning of practical concepts through the book.

**Karaikal, 07-01-2022**

# Chapter 1: An Introduction to Scilab

Scilab is open source software and can be downloaded for installation from the web page of its developer organization viz.: https://scilab.org. The developer organization is presently owned by ESI group. Scilab is supported on UNIX, Macintosh, and Windows environments.

Scilab is an interactive software system for developed for numerical computations and graphics. It is especially designed for matrix computations: solving systems of linear equations, performing matrix transformations, factoring matrices, and so forth. The developers of Scilab have created libraries of a large number of inbuilt mathematical functions. Over that, developers have supplemented Scilab with a wide range of packages of inbuilt programs, called *toolboxes*. These toolboxes, which are collections of inbuilt programs, have been developed for solving different problems of specific areas of practical applications by following specific methods or algorithms.

Further, Scilab is developed to work as a programming language also, in a sense that the users can code their programs also just like any other programming language like C, C++ etc. Also, the inbuilt functions can be used into the users' programs. In addition, it has a variety of graphical capabilities, and can be extended through programs written in its own programming language. This feature of Scilab makes it user friendly interactive software.

Just like its commercial counterpart MATLAB, in Scilab also all the types of variables namely, real, complex, Boolean, integer, string and polynomial variables, are considered as *matrices*. Another salient feature of Scilab is that it understands the difference between real numbers and purely real complex numbers.

**Scilab Advantages**

- It simplifies the analysis of mathematical models
- It frees you from coding in lower-level languages (saves a lot of time but with some computational speed penalties)
- Provides an extensible programming/visualization environment
- Provides professional looking graphs

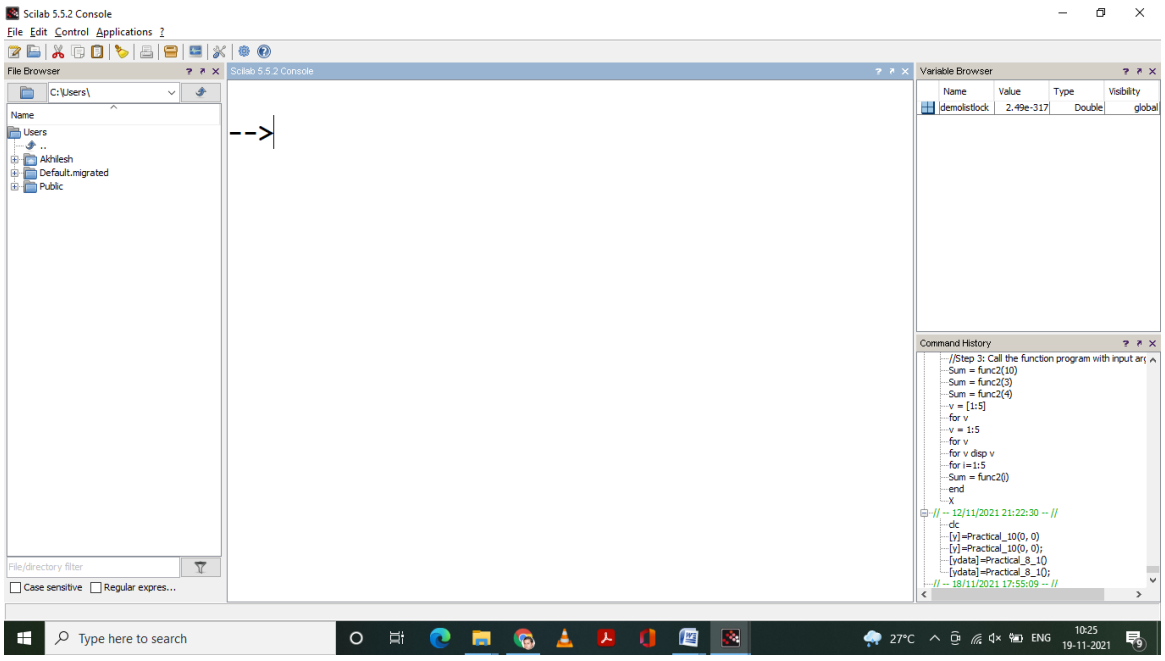
**Scilab Disadvantages**

The only disadvantage of Scilab over the lower level computational programming languages is that it being an interpreted (i.e., not a pre-compiled) language can turn out as slow during large scale computations.

**Remark:** The choice of preferring Scilab over lower level computational programming languages depends upon the requirement of its additional graphics features and availability of inbuilt programs over the scale of the data of the problem to be solved.

## 1.1 Getting Started

Here, we will learn about the installation and use of the software in Windows operating system. Its use is similar in other operating systems also. Installation of the software is easy like any other software. The software gets ready after its installation into a computer system. Just like any other software, Scilab can be opened by Double clicking on the Scilab icon in desktop or by clicking on the icon appearing after the entering the name Scilab in the Windows search bar. The Scilab window should come up on your screen. It looks like this:




This window is the default layout of the Scilab desktop. It is a set of tools for managing files, variables, and applications associated with Scilab.

1. The **Console** is a command window used for entering Scilab functions and other commands at the command line prompt appearing as -->
2. The **Command History Window** is used to view or execute previously run functions.
3. The **Current Directory/Workspace Window** lists the folders/files in the Current Directory (where you are working) or the values and attributes of the variables you have defined.

4. The **Current Directory** line at the top tells you where Scilab thinks your files are located. This should always point to the folder that you are working in so that your files are saved in your own directory. An example would be to enter the pathname

   ```
   C:\Users\Maths50\Scilab\yourname
   ```

   or use the ... button to browse for a folder.

   This should always be done at the start of a new session. When you open a Scilab document, it opens in the associated tool. If the tool is not already open, it opens when you open the document and appears in the position it occupied when last used. Figures open undocked, regardless of the last position occupied.

5. Scilab provides a **variable browser**, which displays the list of variables currently used in the environment.

6. The **Editor**, named as **SciNotes,** is used to access and edit Scilab program files. The Scilab program files are called *script files*. The editor can be accessed from the menu of the console, under the Applications > SciNotes menu, or from the console, as presented in the following session.

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

7. **Script Files**: Script files are normal ASCII (text) files that contain Scilab commands. There are two types of script files in Scilab, namely, programs and functions. It is essential to suffix an appropriate extension name after these files. Extension name for program files is ".sce" (e.g., scriptname.sce) and for functions is ".sci" (e.g., functionscript.sci).

8. **Executing a Script file**

   A script file can be executed using `exec` command, for example:

   ```
   -->exec('D:\Puducherry\Class_2021_10_12_for_Loop.sce', -1)
   ```

   This execution can alternatively be done by pressing the function key F5 while keeping the the script file in SciNotes as active Window.

9. Calling a function program

   A function program can be called by typing its "calling sequence" appropriately in the Scilab console while supplying appropriate values of input arguments.

   **Remark:** If a user programmed function is to be called for computation, then user must execute it before calling the same, in every Scilab session or in case the function program is edited by the user.

## 1.2 Using Scilab as a calculator

The basic arithmetic operators are + for addition, – for subtraction, * for multiplication, / for division, ^ for exponentiation, and these are used in conjunction with braces or commonly called round brackets ( ). The symbol ^ is used to get exponents (powers): 2^4 = 16. An alternative symbol used for the same purpose is **.

Example:

```
--> 2+3/4*5

ans =

    5.7500
```

Note that in this calculation the result was 2+(3/4)*5 and not 2+3/(4*5) because Scilab works according to the priorities of operations given in the following order.

1. quantities in brackets ( )
2. powers or exponent ^ or **
3. multiplication *, left division /, and right division \, working left to right
4. addition + and subtraction –, working left to right

## 1.3 Basic Elements of Scilab as a Programming Language

As Scilab is an interpreted language, therefore there is no need to declare the type of a variable before using it. Variables are created by Scilab at the moment when they are first set (*i.e.*, assigned a value). A value is assigned to a variable using the assignment operator, as detailed below.

### Assignment Operator

The assignment operator "=" is used for assigning a value (or a matrix) to variable. On the left hand side of "=" is placed a variable name to which the value on right hand side is to be assigned. For example, let us observe the following Scilab command demonstrating the use of *assignment operator*.

```
-->x=1

  x =

       1.
```

There are rules of defining variable names for their use in Scilab. These are given below.

## Variable Names

Variable names may be as long as the user wants, but only the first 24 characters are taken into account in Scilab. For consistency, we should consider only variable names which are not made of more than 24 characters. All ASCII letters from "a" to "z", from "A" to "Z" and from "0" to "9" are allowed, with the additional letters "%", "_", "#", "!", "$", "?".

**Caution:** Variable names for which the first letter is "%" have a special meaning in Scilab. These represent the mathematical pre-defined variables, which are discussed in a later section.

Variable names which are allowed and not allowed in Scilab are illustrated below.
Allowed: `NetCost, Left2Pay, x3, X3, z25c5`
Not allowed: `Net-Cost, 2pay, %x, @sign`

## Pre-defined mathematical variables

| Variable Name | Description |
|---|---|
| `%pi` | the mathematical constant $\pi$ |
| `%e` | Euler's constant $e$ |
| `%i` | the imaginary number $i$ |

## Input and output of Mathematical values in Scilab

Apart from the real numbers (expressed in the natural way), complex numbers and Booleans are provided as inputs to Scilab in the formats given below.

- **Complex numbers**

Example: The complex number $2 + 3i$ is input in Scilab as: `2 + 3 * %i` .

- **Booleans**

The truth value "True" is input in Scilab by using `%t` or `%T`, whereas the truth value "False" is input using `%f` or `%F`.

Both these are discussed in detail in later sections.

## Strings

String is a sequence of characters. All the characters including all letter in both cases (i.e., from a to z and from A to Z), digits from 0 to 9 can be used in a string. Strings can be defined and then stored to any variable names by delimiting them in double quotes " ". Let us learn how to define strings through examples given below.

```
-->A = "Scilab"

 A  =

Scilab

-->B = "Software"

 B  =

Software
```

**Remarks:**

- Strings have no direct mathematical use.
- They are used mainly for displaying a lingual message as a part of the output upon the execution of a program.

The concatenation (*i.e.*, join) of two strings can be done by using the concatenation operator (+). The use of concatenation operator is demonstrated below through a Scilab session.

```
-->"Scilab" + "Software"

 ans  =

ScilabSoftware

-->A+B

 ans  =

ScilabSoftware

-->A + " " + B

 ans  =

  Scilab Software
```

It is to be observed that only a string can be concatenated with a string. The concatenation of a string with a number is not possible. For this purpose, a function "`string`" is available in Scilab which converts a number to the string corresponding to

that number. This function can be used appropriately in a situation when there is a requirement of displaying an output message including the output value of some computation also. For example,

```
-->a = 1

 a  =

    1.

-->b = 2

 b  =

    2.

-->c = a + b

 c  =

    3.

-->"The sum of " + string(a) + " and " + string(b) + " is " +
string(c) + "."

 ans  =

  The sum of 1 and 2 is 3.
```

## Suppressing the display of output

The output of any command can be suppressed by ending the command that particular command with semicolon "`;`". For example, the Scilab command in console

```
-->x = 2
```

is returned with a display of the action performed as following

```
x  =

    2.
```

Whereas, the Scilab command

```
-->x = 2;
```

does not display the action performed but will keep output in the computer's temporary memory being used, in exactly same way as was done in the previous command. The value of this output for the variable `x` will remain there in the temporary memory (RAM) for further use, unless it changes by reassigning some other value to this variable

or the Scilab session is closed. Irrespective of the display of the output, the value assigned to the variable `x` can accessed any time from the variable browser in both the cases discussed above.

## Dynamic nature of variable in Scilab

As Scilab is an *interpreted language*, therefore it allows dealing with variables in a dynamic way. The dynamic nature being referred here is in the sense that a variable set to a value of one particular type can later be used to reassign a value of different type also. The same is illustrated below in a Scilab session in Console.

```
-->x = 2 + 3 * %i

 x  =

    2. + 3.i

-->y = 4*x

 y  =

     8. + 12.i

-->x = 5

 x  =

    5.

-->y = 2*x

 y  =

    10.
```

## Comments and continuation lines

Any text that follows // in a line is ignored by the compiler. The main purpose of this facility is to enable inserting comments in the script files. Inserting comments in a script file helps the programmer to read the code of a program with the help of summary messages about the commands. This provision of putting any line as a comment can be exploited even for editing or debugging script files also.

Commands which are too long to be typed in a single line can be continued in multiple subsequent lines by putting *two dots* at the end of each previous line. In Scilab,

any line which ends with two dots is considered to be the start of a new continuation line.

In the following Scilab session in Console, we give examples of Scilab comments and continuation lines.

```
-->// This is my comment .

-->x =1..

-->+2..

-->+3..

-->+4

 x  =

     10.
```

**Remark:** At this stage of introduction of Scilab, the use of comments and continuation lines is demonstrated here through Console, but they are more justifiably used in script file also.

## 1.4 Elementary mathematical functions

Scilab has in-built elementary mathematical functions for their direct use into computations. Most of these functions take one input argument and return one output argument. These functions are vectorized in the sense that their input and output arguments are matrices. This allows computing data with higher performance, without any loop. The list of elementary mathematical functions is present in following tables.[1]

| Function Name | Syntax | Type of variable as input argument | Description |
|---|---|---|---|
| `exp` | `exp(X)` | scalar, vector, or vector (real or complex entries) | exp(X) is the (element-wise) **exponential** of the entries of X. |
| `expm` | `expm(X)` | a square matrix with real or complex entries. | If X is a square matrix then expm(X) is the matrix $\text{expm}(X) = I + X + \frac{X^2}{2!} + \cdots$ |
| `log` | `log(x)` | scalar, vector or matrix | log(x) is the "element-wise" |

[1] A detailed description of these functions can be accessed from the help document of Scilab, which is available within the software and on the website of the Scilab developers as well.

| | | | |
|---|---|---|---|
| | | | logarithm $y(i,j) = log(x(i,j))$ |
| log10 | log10(x) | scalar, vector or matrix | base 10 logarithm |
| log1p | log1p(x) | scalar, vector or matrix | log1p(x) is the "element-wise" $\log(1+x)$ function. y(i,j)=log(1 + x(i,j)). This function, defined for x > -1, must be used if we want to compute log(1+x) with accuracy for \|x\| << 1. |
| log2 | log2(x) | scalar, vector or matrix | log2(x) is the "element-wise" base 2 logarithm y(i,j)=log2(x(i,j)). |
| logm | logm(x) | square matrix | logm(x) is the matrix logarithm of x. The result is complex if x is not positive or definite positive. If x is a symmetric matrix, then calculation is made by Schur form. Otherwise, x is assumed diagonalizable. One has expm(logm(x))=x. |
| max | max(A) | vector or matrix with real number values | maximum value of matrix A |
| min | min(A) | vector or matrix with real number values | minimum value of matrix A |
| modulo | modulo(n,m) | integers | remainder of n divided by m (n and m integers) |
| pmodulo | pmodulo(n,m) | integers | positive arithmetic remainder of n divided by m (n and m integers) |
| sign | sign(A) | real or complex matrix | returns the matrix made of the signs of A(i,j). For complex A, sign(A) = A./abs(A). |
| signm | signm(A) | real or complex matrix | for square and Hermitian matrices X=signm(A) is matrix signum function. |
| sqrt | sqrt(x) | real or complex scalar or vector | returns the vector of the square root of the $x$ elements. Result is complex if x is negative. |
| sqrtm | sqrtm(x) | real or complex square matrix | the matrix square root of the x x matrix (x=y^2) Result may not be accurate |

| | | | |
|---|---|---|---|
| | | | if x is not symmetric |

**Remark:** All the elementary mathematical functions listed above are discussed as real functions or real vector valued functions. Those mathematical functions which are extended from real to complex functions (for example, trigonometric, exponential and logarithmic functions) have same function names. This means that, if their input argument is a complex number, the same function returns the output as a complex number, behaving as a complex function. This feature of Scilab is phrased as "elementary functions in Scilab are overloaded for complex numbers". The following Scilab session illustrates this feature.

```
-->y = sin(%pi/2)

 y  =

     1.

 -->w = sin(2 + 3 * %i)

 w  =

     9.1544991 - 4.168907i
```

Some other functions provided in Scilab which help managing complex numbers are provided below.

## Functions to manage complex numbers

Another salient feature of Scilab is that it understands the difference between real numbers and purely real complex numbers.

| Function Name | Description |
|---|---|
| `real` | gives the real part of complex number |
| `imag` | gives the imaginary part of complex number |
| `imult` | performs multiplication of number in input with *i* |
| `isreal` | returns true if the variable has no complex entry |

Following Scilab session demonstrates the use of these functions.

```
--> z = 2 + 3 * %i
```

```
 z  =

     2. + 3.i

 -->x = real(z)

 x  =

     2.

 -->y = imag(z)

 y  =

     3.

-->z1 = imult(z)

 z1  =

  - 3. + 2.i

-->isreal(z)

 ans  =

  F

-->isreal(2)

 ans  =

  T
```

**Remark:** Scilab distinguishes between real and purely real complex numbers. This feature can be verified by appropriately using the `isreal` function as demonstrated below.

```
-->z2 = 2 + 0 * %i

 z2  =

     2.

-->isreal(z2)

 ans  =

   F
```

```
-->isreal(2)

 ans  =

   T
```

## Booleans

Some comparison operators and logical connective operators available in Scilab are described given below.

### Comparison operators

| Operator | Use | Description |
|---|---|---|
| == | a==b | returns truth value true if expression a is equal to b; otherwise false |
| ~= or <> | a~=b or a<>b | returns truth value true if expression a is not equal to b; otherwise false |
| < | a<b | returns truth value true if a real expression a is less than b; otherwise false |
| > | a>b | returns truth value true if a real expression a is greater than b; otherwise false |
| <= | a<=b | returns truth value true if a real expression a is less than or equal to b; otherwise false |
| >= | a>=b | returns truth value true if a expression a is greater than or equal to b; otherwise false |

### Logical connective operators

| Operator | Use | Description |
|---|---|---|
| & | A&B | logical **AND** operator: returns truth value true only for the case when both the logical expressions A and B have truth value true |
| \| | A\|B | logical **OR** operator: returns truth value true for when at least one of the logical expressions A and B have truth value true |
| ~ | ~A | logical **NOT** operator: returns truth value true if the logical expressions A has truth value false and vice-versa |

The use of all the operators is demonstrated through a Scilab session in Console, as given below.

```
-->x = 1;

-->y = 2;

-->z = x + y;

z  =

    3.

-->z==3

ans  =

   T

-->x<y

 ans  =

  T

-->x<=y

 ans  =

  T

-->x>z

 ans  =

  F

-->x>=1

 ans  =

  T

-->x~=1

 ans  =

  F

-->x<>1

 ans  =

  F

-->z = (x==1) & (y==2)
```

```
 z  =

   T

-->z1 = (x==1) | (y==3)

 z1  =

   T

-->z2 = ~z1

 z2  =

   F
```

## Strings

String is an array of characters. To represent a string in Scilab, a set of characters is enclosed within double quotes (for example, “A string in Scilab”).

- Strings can be stored in variables by using assignment operator, just as other values of real or complex numbers can be stored.
- Concatenation (i.e., joining) of strings is done by using the concatenation operator +. The following Scilab session demonstrates the concatenation of two strings.

```
-->x = "String "

 x  =

String

-->y = "Concatenation"

 y  =

Concatenation

-->z = x + y

 z  =

String Concatenation
```

- The Scilab in-built function `string` gives the output as a string of numeric characters corresponding to any number supplied as input. Its use is demonstrated through a Scilab session in console.

```
-->m = 2;

-->n = 4;

-->string(m) + " divides " + string(n)

ans  =

2 divides 4
```

## Exercise 1

In the console:

1. Use assignment operator for creating and setting (assigning) variables to float value, string, Boolean values,
2. Use of comment and continuation line,
3. Use of inbuilt Mathematical function and operators,
4. Use of pre-defined Mathematical variables,
5. Use of Booleans and comparison operators,
6. Use of complex numbers, and operations on them,
7. Use of strings and concatenation operator, and use of comparison operator '==' for comparison of two strings for checking their equality.
8. Swap the values assigned to two variables without using third variable.

# Chapter 2: Matrices

Scilab works with essentially only one kind of object a rectangular, numerical array of numbers, possibly complex, called a matrix. In some situations, $1 \times 1$ matrices are interpreted as scalars and matrices with only one row or one column are interpreted as vectors. Matrices can be introduced into Scilab in several different ways:

- Entered by an explicit list of elements.
- Generated by built-in statements and functions.
- Created in Script files.
- Loaded from external data files.

Scilab contains no size or type declarations for variables. Scilab allocates storage automatically, up to available memory.

## 2.1 Vectors

Vectors come in two formats - row vectors and column vectors. In either case they are lists of numbers separated by either commas or spaces.

- The number of entries is known as the "`length`" of the vector and the entries are called *elements* or *components* of the vector.
- The entries must be enclosed by square brackets "[" and "]". Entries of a row vector are separated by comma (`,`) or space. Whereas entries of a column vector are separated by semicolon (`;`).
- A row vector can be transposed to a column vector and vice-versa using the transpose operator (`.'`). Whereas, transpose conjugate operation be performed by using (`'`). Thus, both of these operators give the output for real vectors.
- Binary operations applicable on vectors are *addition* (+), *subtraction* (-), *scalar multiplication* (*), and dot product (.*).
- A row (column) vector can be joined with another row (column) vector. The same is demonstrated below.

Following Scilab session illustrates features of Scilab vectors discussed above.

```
--> v = [1, 3, sqrt(5)]
v =
    1.    3.    2.236068
--> length(v)
ans =
```

```
    3.
--> v2 = 3*v
v2 =

    3.    9.    6.7082039

-->v + v2

 ans  =

     4.    12.    8.9442719

-->v - v2

 ans  =

  - 2.  - 6.  - 4.472136

-->v3 = v.'

 v3  =

    1.

    3.

    2.236068

-->v4 = v2.'

 v4  =

    3.

    9.

    6.7082039

-->v5 = [v, v2]

 v5  =

     1.    3.    2.236068    3.    9.    6.7082039

 -->v6 = [v3; v4]

 v6  =

    1.

    3.

    2.236068
```

```
    3.
    9.
    6.7082039
 -->v6 = [v3, v4]
 v6  =
    1.          3.
    3.          9.
    2.236068    6.7082039
```

**Remark:** 1. It is to be noted from the above demonstration of Scilab session that the dimensions of vectors must agree for joining the vectors.

2. In all vector arithmetic with vectors of equal length, the operations are carried out element-wise.

Particular entries of a vector can be accessed and changed as demonstrated below.

```
-->v
 v  =
    1.    3.    2.236068
-->v(1)
 ans  =
    1.
-->v(2)
 ans  =
    3.
-->v(3)
 ans  =
    2.236068
-->v(2) = 5
 v  =
```

```
    1.    5.    2.236068
```

## 2.1.1 Operations on Vectors

In Scilab, operations are available for addition, subtraction, vector product, element-wise product, element-wise division, and element-wise power.

**Addition (+) and subtraction (-) of vectors**

The addition (+) and subtraction (-) operations in Scilab work element-wise by giving output as addition and subtraction of corresponding elements. For example:

```
-->a = [5 7 9; 1 -3 -7], b = [-1 2 5; 9 0 5]
 a  =
    5.    7.    9.
    1.  - 3.  - 7.
 b  =
   - 1.    2.    5.
    9.    0.    5.
-->a+b
 ans  =
    4.     9.    14.
    10.  - 3.  - 2.
-->a-b
 ans  =
     6.    5.    4.
   - 8.  - 3.  - 12.
```

**Vector product (*)**

We shall describe two ways in which a meaning may be attributed to the product of two vectors. In both cases, the vectors concerned must have the same length. The first product is the standard scalar product. Suppose that u and v are two vectors of length n, $u$ being a row vector and $v$ a column vector:

$$u = [u_1, u_2, \ldots\ldots, u_n], \qquad v = \begin{bmatrix} v_1 \\ v_2 \\ . \\ . \\ v_n \end{bmatrix}$$

The scalar product is defined by multiplying the corresponding elements together and adding the results to give a single number (scalar).

$$u * v = \sum_{i=1}^{n} u_i v_i$$

We can perform this product in Scilab by

```
--> u = [10, -11, 12], v = [20; -21; -22]

u =

    10.   -11.    12.

v =

    20.

   -21.

   -22.

--> prod=u*v        // row times column vector

prod =

    167
```

Suppose we also define a row vector w and a column vector z by

```
--> w=[2,1,3], z=[7;6;5]

w =
     2.     1.     3.
z =
     7.
     6.
     5.
```

and we wish to form the scalar products of `u` with `w` and `v` with `z`.

```
-->u*w
```

```
    !--error 10

Inconsistent multiplication.
```

An error results because w is not a column vector. Recall from earlier that transposing (with ') turns column vectors into row vectors and vice versa. So, to form the scalar product of two row vectors or two column vectors,

```
--> u*w.'       // u and w are row vectors
ans =
     45
--> u*u.'       // u is a row vector
ans =
     365
--> v.'*z       // v and z are column vectors
ans =

     -96
```

We shall refer to the Euclidean length of a vector as the norm of a vector; it is denoted by the symbol ‖. ‖ and defined by

$$\|u\| = \sqrt{\sum_{i=1}^{n} {u_i}^2}$$

where $n$ is its dimension. This can be computed in Scilab in one of two ways examplified below:

```
--> [sqrt(u*u.'),norm(u)]

ans =

         19.104973    19.104973
```

where `norm` is a built in Scilab function that accepts a vector $u$ as input and delivers a scalar $\|u\|$ as the output.

**Element-wise product or dot product (.*)**

The second way of forming the product of two vectors of the same length is known as the *Hadamard product*. It is not often used in Mathematics but is an invaluable Scilab feature. It involves vectors of the same type. If u and v are two vectors of the same type (both row vectors or both column vectors), the mathematical definition of this product, which we shall call the dot product, is the vector having the components

$$u.v = [u_1 v_1 , u_2 v_2 , \ldots\ldots , u_n v_1]$$

The result is a vector of the same length and type as u and v. Thus, we simply multiply the corresponding elements of two vectors. In Scilab, the product is computed with the operator .* and, using the vectors w, z defined earlier

```
--> w.*w
ans =
     4.      1.      9.

>> w.*z'
ans =

     14.     6.      15.
```

**Element-wise division of vectors: right-division (./) and left-division (.\)**

There is no mathematical definition for the division of one vector by another. However, in Scilab, operators "`./`" and "`.\`" is defined to give element-wise division. It is therefore only defined for vectors of the same size and type. The right-division opertor "`./`" divides elements of pre-factor with corresponding elements of post-factor. Whereas, left-division opertor "`.\`" divides elements of post-factor with corresponding elements of pre-factor. That is, for vectors `a` and `b` of same shape and length, `a./b` gives the vector with entries `a(i)/b(i)` for each index `i` and `a.\b` gives the vector with entries `b(i)/a(i)` for each index `i`.

```
--> a = 1:5, b=6:10, a./b, a.\b
a =
     1.      2.      3.      4.      5.
b =
     6.      7.      8.      9.     10.
ans =
    0.1666667    0.2857143    0.375    0.4444444    0.5

ans  =

    6.    3.5    2.6666667    2.25    2.

--> a./a
ans =
     1.      1.      1.      1.      1.

--> c = -2:2, a./b
c =
    -2.    -1.      0.      1.      2.
ans =
       0.1666667    0.2857143    0.375    0.4444444    0.5
```

**Remark:** The importance of these operations will be highlighted during their discussion on two dimensional matrices as same operations are applicable there also.

**Element-wise power of vectors (.^)**

To square each of the elements of a vector we could, for example, do `u.*u`. However, a neater way is to use the `.^` operator is demonstrated below.

```
--> u
u =
   10.    -11.    12.
--> u.^2
ans =
  100.    121.   144.
--> u.*u
ans =
  100.    121.   144.
--> u.*w.^(-2)
ans =

  2.5     -11.     1.3333333
```

Observe that powers (.^ in this case) are done first, before any other arithmetic operation.

## 2.1.2 The Colon Operator

Colon operator in Scilab enables to create a row vector having integers entries in an arithmetic progression.

1. The most **basic syntax** of the colon operator is:

   `v = i : j`

   where `i` is the starting index and `j` is the ending index, with $i \leq j$. This creates a row vector

   `v = (i, i+1, . . . , j).`

   The following Scilab session demonstrates the use of colon operator

   ```
   -->3:7
   ans  =
      3.    4.    5.    6.    7.
   ```

2. The **complete syntax** allows configuring the increment used when generating the index values, *i.e.*, the step. The complete syntax for the colon operator is

   `v = i : s : j`

where i is the starting index, j is the ending index and s is the step. This command creates the vector v = (i, i+s, i+2s, . . . , i+ns) where n is the greatest integer such that i + ns ≤ j, if s ≥ 0 and i ≤ j.

**Observation:**

(a) If s divides j – i, then the last index in the vector of index values is j, because in that case i + ns = j.

(b) In other cases, we have i + ns < j.

While, in most situations the step s is positive, it can be negative also. Following Scilab session demonstrates all the cases discussed here.

```
-->v = 4 : 2 : 10

 v  =

    4.    6.    8.    10.

 -->v = 3 : 2 : 10

 v  =

    3.    5.    7.    9.

 -->v = 10 : -2 : 4

 v  =

    10.    8.    6.    4.

 -->v = 10 : -2 : 3

 v  =

    10.    8.    6.    4.
```

## 2.1.3 Creating linearly spaced vector: linspace function

Scilab has an in-built function linspace for creating a linearly spaced vector with any specified values between two real or complex numbers. The syntax for calling this function is

```
[v] = linspace(x1, x2, n) .
```

Here, x1 and x2 are real or complex scalars or column vectors, and

n is a natural number whose value should be greater than or equal to 2.

For example:

```
-->eye(2, 2)

-->linspace(1,2,5)

 ans  =

     1.    1.25    1.5    1.75    2.

-->linspace(1+%i,2+2*%i,5)

 ans  =

     1. + i   1.25 + 1.25i  1.5 + 1.5i  1.75 + 1.75i  2. + 2.i

-->linspace([1:4]',[5:8]',5)

 ans  =

    1.    2.    3.    4.    5.

    2.    3.    4.    5.    6.

    3.    4.    5.    6.    7.

    4.    5.    6.    7.    8.
```

## 2.2 Two-dimensional matrices

### 2.2.1 Creating Matrices

There is a simple and efficient syntax to create a matrix with given values. The following is the list of symbols used to define a matrix:

- square brackets "[" and "]"mark the beginning and the end of the matrix,
- commas "," or spaces separate the values on different columns,
- semicolons ";" separate the values of different rows.

The following syntax can be used to define a matrix, where blank spaces are optional (but make the line easier to read) and "..." are designing intermediate values:

```
A = [a11, a12, ... , a1n; a21, a22, ... , a2n; ...; an1, an2,
... , ann]
```

For example, either of the statements

```
--> A = [1 2 3; 4 5 6; 7 8 9]
```

and

```
--> A = [1 2 3
--> 4 5 6
--> 7 8 9]
```

create the same matrix and assign it to the variable A. Scilab responds to this command by storing in the following matrix against the variable name A and displaying the same in Console.

```
A =
     1      2      3
     4      5      6
     7      8      9
```

Scilab always prints out the variable in the executed line (which can be very awkward and time consuming for large arrays) unless you end the line with a semicolan (;). It is good practice to use the semicolan at the end of the line. You can always ask Scilab to print the matrix later also by calling the variable name, as shown below.

```
--> A

A =
     1      2      3
     4      5      6
     7      8      9
```

## 2.2.2 Some Special Matrices through in-built functions

The built-in functions `rand`, `eye`, and `ones`, for example, provide an easy way to create matrices with which to experiment.

**Identity matrix**

The command `eye(n, n)` creates the identity matrix of order `n × n`. For example:

```
-->eye(2, 2)
 ans  =
    1.    0.
    0.    1.
```

**Remark:** For the case of $m \neq n$, the Scilab command `eye(m, n)` creates a rectangular matrix with `m` rows and `n` columns with the identity matrix of order $= \min\{m, n\}$ and additional rows or columns as zeros. For example:

```
-->eye(2, 3)
 ans  =
    1.    0.    0.
    0.    1.    0.
-->eye(3, 2)
 ans  =
    1.    0.
    0.    1.
    0.    0.
```

**Matrix with all entries one**

The command `ones(m, n)` creates the identity matrix of order `m × n`. For example:

```
-->ones(2, 3)
ans  =
    1.    1.    1.
    1.    1.    1.
```

**Matrix with all entries zero**

The command `zeros(m, n)` creates the identity matrix of order `m × n`. For example:

```
-->zeros(2, 3)
ans  =
    1.    1.    1.
    1.    1.    1.
```

**Random matrix**

The command `rand(m, n)` will create a matrix of order `m × n` with randomly generated entries distributed uniformly between 0 and 1

```
--> rand(5, 5)
ans =
```

```
    0.8147    0.0975    0.1576    0.1419    0.6557
    0.9058    0.2785    0.9706    0.4218    0.0357
    0.1270    0.5469    0.9572    0.9157    0.8491
    0.9134    0.9575    0.4854    0.7922    0.9340
    0.6324    0.9649    0.8003    0.9595    0.6787
```

while `rand(m,n)` will create an m × n array.

```
--> rand(3,4)

ans =
    0.7577    0.6555    0.0318    0.0971
    0.7431    0.1712    0.2769    0.8235
    0.3922    0.7060    0.0462    0.6948
```

## 2.2.3 Operations on matrices

Scilab provides operations on matrices of same size *viz*. addition (+), subtraction (-), element-wise multiplication or dot product (.*), element-wise right-division (./), and element-wise right-division (.\). Also the multiplication on any two matrices of comparable order is performed in Scilab using the multiplication operator (*). Herein, Scilab considers row and column vectors as row and column matrices. All these matrix operations are discussed below. Two more operators namely, right-division (/) and left-division (\) are also available in Scilab and are discussed later in this section.

**Addition (+) and subtraction (-) of matrices**

The addition and subtraction work element-wise, just as for vectors: corresponding elements are added together.

```
-->a = [5 7 9; 1 -3 -7], b=[-1 2 5; 9 0 5]

 a  =

    5.    7.    9.

    1.  - 3.  - 7.

 b  =

   - 1.    2.    5.

    9.    0.    5.

-->a+b

 ans  =
```

```
    4.     9.    14.

    10.  - 3.  - 2.

-->a-b

 ans  =

     6.    5.    4.

   - 8.  - 3.  - 12.
```

**Dot product of matrices (.*)**

The dot product works as for vectors: corresponding elements are multiplied together - so the matrices involved must have the same size.

```
--> a = [5 7 9; 1 -3 -7], b=[-1 2 5; 9 0 5]
a =
     5     7     9
     1    -3    -7

b =
    -1     2     5
     9     0     5

--> a.*b
ans =
    -5    14    45
     9     0   -35

--> c=[0 1;3 -2;4 2]
c =
     0     1
     3    -2
     4     2

-->a.*c
     !--error 9999
Inconsistent element-wise operation
-->a.*c'
ans =
     0    21    36
     1     6   -14
```

**Product of matrices**

To form the product of an $m \times n$ matrix $A$ and a $n \times p$ matrix $B$, written as $AB$, we visualize the first matrix ($A$) as being composed of $m$ row vectors of length $n$ stacked on

top of each other while the second (*B*) is visualized as being made up of *p* column vectors of length *n*. The entry in the *i*th row and *j*th column of the product is then the scalar-product of the *i*th row of *A* with the *j*th column of *B*. The product is an $m \times p$ matrix.

Check that you understand what is meant by this definition by taking through the following examples.

```
--> a=[5 7 9;1 -3 -7],b=[0 1;3 -2;4 2]

a =
     5      7      9
     1     -3     -7

b =
     0      1
     3     -2
     4      2

--> c=a*b
c =
    57      9
   -37     -7

--> d=b*a
d =
     1     -3     -7
    13     27     41
    22     22     22

--> e=b'*a'
e =
    57    -37
     9     -7
```

We see that *e* = *c*' suggesting that (*a* * *b*)' = *b*' * *a*' Why is *b* * *a* a 3 × 3 matrix while *a* * *b* is 2 × 2?

**Remark:** (Matrix-vector are vector-matrix product) As vectors are row or column matrices, therefore the product of a matrix with a vector with appropriate order is covered in this definition of the product operations of matrices in Scilab.

**Left-division (\) of matrices**

If A is a square matrix and A and B are matrices of comparable order for multiplication, then the left-division operator (\) works for `A\B` to give the result `inv(A)*B`. For example:

```
-->A = [1, 2; 3, 4]
 A  =
    1.    2.
    3.    4.
-->B = [2, 5, 4; 5, 6, 7]
 B  =
    2.    5.    4.
    5.    6.    7.
-->A\B
 ans  =
    1.   - 4.   - 1.
    0.5    4.5    2.5
 -->inv(A)*B
 ans  =
    1.   - 4.   - 1.
    0.5    4.5    2.5
```

This operator can be used to solve for $x$ the matrix equation[2]

$$Ax = b.$$

**For example:** For the system of equations

$$x_1 + 2x_2 = 3$$

$$3x_1 + 4x_2 = 7$$

the matrix form is $Ax = b$, where $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, and $b = \begin{bmatrix} 3 \\ 7 \end{bmatrix}$.

As determinant of $A$ is -2, which is non-zero, therefore the matrix $A$ is invertible. Hence, the solution of this system of equations can be obtained as $x = A^{-1}b$. This is can be solved through the following Scilab session.

[2] Solving the system of equations will be discussed again in detail in a later chapter and there this operator would be utilized as a part of the complete discussion.

```
-->A
 A  =
    1.    2.
    3.    4.
-->b = [3; 7]
 b  =
    3.
    7.
-->x = A\b
 x  =
    1.
    1.
```

Alternatively, the same can be achieved by using the in-built Scilab function `inv` for inverse of a matrix.

```
-->x = inv(A)*b
 x  =
    1.
    1.
```

**Right-division (/) of matrices**

If B is a square matrix and A and B are matrices of comparable order for multiplication, then the right-division operator (/) works for `A/B` to give the result `A*inv(B)`. For example:

```
-->C = [2, 3; 4, 5; 6, 7]
 C  =
    2.    3.
    4.    5.
    6.    7.
-->D = [1, 2; 3, 4]
```

```
 D  =

    1.    2.

    3.    4.

-->C/D

 ans  =

    0.5    0.5

  - 0.5    1.5

  - 1.5    2.5

-->C*inv(D)

 ans  =

    0.5    0.5

  - 0.5    1.5

  - 1.5    2.5
```

**Transpose of a matrix**

Just like on vectors, the transpose of a matrix is obtained by the command (.') and the transpose conjugate of a matrix is obtained by the command ('). For example:

```
-->A

 A  =

    1.    2.    3.

    4.    5.    6.

-->A'

 ans  =

    1.    4.

    2.    5.

    3.    6.

-->A.'

 ans  =

    1.    4.

    2.    5.
```

```
    3.    6.
-->B = [1 + 2 * %i, 2 + 3 * %i; %i, -%i]
 B  =
    1. + 2.i    2. + 3.i
    i         - i
-->B.'
 ans  =
    1. + 2.i    i
    2. + 3.i  - i
-->B'
 ans  =
    1. - 2.i  - i
    2. - 3.i    i
```

## Query Matrices

We can get the size (or order or dimensions) of a matrix with the command `size`. The size function returns the two output arguments `nr` and `nc`, which are the number of rows and the number of columns.

```
-->A = ones(2 ,3)
A =
    1.    1.    1.
    1.    1.    1.
-->[nr ,nc ]= size (A)
nc  =
    3.
 nr  =
    2.
```

The size function has also the following syntax

```
nr = size(A ,sel)
```

which allows to get only the number of rows or the number of columns and where sel can have the following values

- `sel=1` or `sel="r"`, returns the number of rows,
- `sel=2` or `sel="c"`, returns the number of columns.
- `sel="*"`, returns the total number of elements, that is, the number of columns times the number of rows.

In the following session, we use the size function in order to compute the total number of elements of a matrix.

```
-->A = ones(2 ,3)
 A  =
    1.    1.    1.
    1.    1.    1.
-->size (A,"*")
 ans  =
    6.
```

## Accessing the elements of a matrix

There are multiple ways of accessing matrices, completely or in part.

**Method 1: Calling the whole matrix**

For a matrix which is already defined and assigned to the variable `A`, the whole matrix can be assessed by calling the same by its variable name. For exampe,

```
-->A = ones (2 ,3);
-->A
 A  =
    1.    1.    1.
    1.    1.    1.
```

**Method 2: Calling an element of a matrix**

For a matrix which is already defined and assigned to the variable `A`, the element in row number `i` and column number `j` of the matrix `A` can be assessed by calling it through the syntax `A(i, j)`. For exampe,

```
-->B = [1, 2, 3; 4, 5, 6]
 B  =
    1.    2.    3.
    4.    5.    6.
-->B(2, 3)
```

```
 ans  =
    6.
-->B(1, 3)
 ans  =
    3.
-->B(1, 2)
 ans  =
    2.
```

**Method 3: Accessing a submatrix of a matrix**

For a matrix which is already defined and assigned to the variable `A`, its submatrix can be accessed by specifying the indices for row numbers and column numbers as row-vectors. For the vector of row indices is defined as `u` and vetcor for column indices is `v`, the submatrix of matrix `A` can be assessed by calling it through the syntax `A(u, v)`. For exampe, for a matrix `B` (defined below), the submatrix of with entries of 2nd and 4th row which lie in 1st and 4th coulmn can be accessed by the commands demonstrated in the following Scilab session.

```
-->B = [1, 2, 3, 4, 5; 6, 7, 8, 9, 10; 11, 12, 13, 14, 15; ..
16, 17, 18, 19, 20]
B  =
    1.     2.     3.     4.     5.
    6.     7.     8.     9.     10.
    11.    12.    13.    14.    15.
    16.    17.    18.    19.    20.
-->u = [2, 4], v = [1, 4]
 u  =
    2.    4.
 v  =
    1.    4.
-->B(u, v)
 ans  =
    6.     9.
    16.    19.
```

**Remarks:**

1. Colon operator can also be used for defining the above discussed vector of indices. For example, the submatrix of the above mentioned matrix `B` with entries from the first three rows which lie in 1st, 3rd and 5th column can be accessed by following the following Scilab command.

```
-->B(1:3, 1:2:5)
 ans  =
    1.     3.     5.
    6.     8.     10.
    11.    13.    15.
```

2. Complete rows or columns can be accessed by appropriately using the colon operator in place of index vector. Particularly,
   a. The complete *i*th row of a matrix `A` can be accessed by the command `A(i, :)`. For example, for the matrix `B` mentioned above, complete 3rd row can be accessed through the command

```
-->B(3, :)
 ans  =
     11.    12.    13.    14.    15.
```

   b. The complete *j*th column of matrix `A` can be accessed by the command `A(:, j)`. For example, for the matrix `B` mentioned above, complete 2nd column can be accessed through the command

```
-->B(:, 2)
 ans  =
    2.
    7.
    12.
    17.
```

3. The order in which indices are placed in the index vector `u` or `v` for accessing rows and columns, corresponding rows of the submatrix are arranged in the output obtained through the command `A(u, v)`. For example, for the matrix `B` mentioned above, observe the output obtained through the following command

```
-->u = [2, 1];
-->B(u, :)
 ans  =
    6.    7.    8.    9.    10.
    1.    2.    3.    4.    5.
```

   The command `B(u, :)` gives the complete 1st and 2nd rows but in reverse order, as the vector `u` depicts the row indices `2` first and `1` second.

**The dollor operator ($) for counting indices to assess matrices**

The dollor operator ($) enables to refer to the last row index or last column index of a matrix without manually finding those. For example, if `A` is a matrix of order $m \times n$, then for referring to the elements of matrix `A` while counting the indices in decending order starting from the last one, can be done through the following commands.

| Command | Description |
|---|---|
| `A(i, $)` | the element of matrix `A` in row `i` and the last column (*i.e.*, column `n`) |
| `A($, j)` | the element of matrix `A` in the last row (*i.e.*, row `m`) and column `i` |
| `A($-i, j)` | the element of matrix `A` in the row (`m - i`) and the column `j` |
| `A(i, $-j)` | the element of matrix `A` in the row `i` and the column (`n - j`) |
| `A($-i, $-j)` | the element of matrix `A` in the row (`m - i`) and the column (`n - j`) |

These commands are illustrated below through the following Scilab session.

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

## 2.2.4 Functions on matrices

There are some useful in-built functions available in Scilab which enable obtaining some important informations about matrices. Some of these are detailed in the table given below.[3]

| Function | Syntax | Description |
|---|---|---|
| det | det(A) | gives the determinant of a matrix A |
| inv | inv(A) | gives the inverse of a matrix A |
| linsolve | x=linsolve(A, b) | solves the system of linear equations Ax+b=0 |
| trace | a=trace(A) | gives the trace (sum of diagonal entries) of a matrix A |
| spec | [R,diagevals] =spec(B)<br>or<br>evals = spec(B) | R = square matrix of eigenvectors (specifically, the right-eigenvectors)<br>diagevals = gives a diagonal matrix with entries the eigenvalues of matrix A<br>evals = column vector comprising of eigenvalues of matrix A |

First four functions on matrices introduced in the table given above are self explanatory, whereas the spec function requires little elaboration through practical demonstrations through a Scilab session as given below. For this demonstration we have take simple most examples of matrices whose eigenvalues and eigenvectors can be identified at the first look due to known properties of linear algebra.

```
-->A = eye(2, 2)
 A  =
    1.    0.
    0.    1.
```

[3] Here only some of the in-built functions have been listed. Interested readers may refer to the help document of Scilab for exploring more in-built functions available in Scilab.

```
-->[R,diagevals] =spec(A)
 diagevals  =
    1.    0.
    0.    1.
 R  =
    1.    0.
    0.    1.

// If we call the spec function without specifying any output
// arguments or by specifying only one output argument, then
// only eigenvalues are returned by the call of function in
// the form of a column vector.
-->spec(A)
 ans  =
    1.
    1.
-->evals = spec(A)
 evals  =
    1.
    1.
-->B = [2, 0; 0, 3]
 B  =
    2.    0.
    0.    3.
-->spec(B)
 ans  =
    2.
    3.
```

```
-->[R,diagevals] =spec(B)
 diagevals  =
    2.    0.
    0.    3.
 R  =
    1.    0.
    0.    1.
-->B = [2, 0, 0; 0, 2, 0; 0, 0, 3]
 B  =
    2.    0.    0.
    0.    2.    0.
    0.    0.    3.
-->[R,diagevals] =spec(B)
 diagevals  =
    2.    0.    0.
    0.    2.    0.
    0.    0.    3.
 R  =
    1.    0.    0.
    0.    1.    0.
    0.    0.    1.
```

The discussion on matrices is vast and it is impossible to list its all features and ways to deal them in Scilab. The demonstrations made in this chapter are sufficient to explain the fundamentals for all the basic dealings of matrices. This makes it appropriate to conclude this chapter here to the scope of this book. For advanced topics, readers are suggested to refer to the help document of Scilab.

## Exercise 2

1. Perform following tasks:
   (a) Create an identity matrix of a pre-specified order.
   (b) Create a matrix of a pre-specified order with all entries ones.
   (c) Create a matrix of a pre-specified order with all entries zeros.
   (d) Use the function 'linspace' to create a vector with initial value 0, final value 10 and number of values in the vector as 5.
   (e) Use the function 'rand' and 'grand' for creating matrices.
   (f) Use the function 'testmatrix' for defining a matrix.
   (g) Create the following matrix in Scilab.

$$\begin{bmatrix} 12 & 14 & 11 & 2 \\ -2 & 3 & 16 & 19+2\mathrm{i} \\ 21 & 3 & 6 & 8 \end{bmatrix}$$

2. Use the function size with its various calling sequences, to obtain the shape of the matrix.
3. Write a code to (1) create a matrix with 3 rows and 4 columns and (2) add the entries of first column of the matrix.
4. Use colon operator to create
   (a) a vector of integers with starting value -5 and ending value 3.
   (b) a vector with all odd integers with starting value 5 and ending value -5.
5. Use the dollar operator to access the entry in 3rd row and 4th column of a matrix with order 4 x 5.
6. Use lower level operations for addition, subtraction, multiplication, right division, left division, power, and transpose-conjugate of matrices.
7. Use element-wise operations for element-wise addition, subtraction, multiplication, right division, left division, power of matrices. Obtain transpose (but not conjugate) of a matrix.

# Chapter 3: Scilab Programming

Scilab is a software which enables the users to work more freely for utilizing its available computing functionalities by developing their own programs for any method or algorithm. Scilab has its own programming syntax for various statements which are required for any programming language. Among foremost fundamental statements of such a programming language are branching statements and looping statements. In this chapter the syntax and use these statements are discussed in the context of Scilab as a programming language.

## 3.1 Branching statements

Branching statements are used to enable a program for handling the decision making situations while they are run with practical inputs. These statements are categorized depending upon the type of decisions to be taken. Such basic statements are described below one by one.

**The `if` statement**

This statement allows executing desirable sequence of actions depending on if a certain logical condition gets satisfied. The syntax for this statement is

```
if (logical test) then
     Scilab Command 1
     Scilab Command 2
     ...
end
```

The syntax of "`if` statement" in Scilab as given above, first evaluates the condition given through the expression "`logical test`". If the output of this evaluation comes out with Boolean value "`True`", then the Scilab commands written between the keywords `then` and `end` are executed sequentially. Whereas, for the case, when the output of this evaluation is comes out with Boolean value "`False`", then nothing is executed.

Let us understand this concept through a small program coded in Scilab editor "SciNotes".

**Example 3.1:** The following program having an "`if` statement" tests whether a given real number is equal to zero, and then displays an appropriate message only for the case when output of the logical test is "`True`" (*i.e.*, when the number is equal to zero).

```
clear
r = input("Enter a real number ")
if (r == 0) then
    disp("The given number is equal to zero.")
end
```

The output of the above program upon execution is demonstrated below for two different inputs.

Output:

```
-->exec('D:\SCILAB\Ch3_1_Ex1_if_Statement.sce', -1)
Enter a real number 0
  The given number is equal to zero.
-->exec('D:\SCILAB\Ch3_1_Ex1_if_Statement.sce', -1)
Enter a real number 2
-->
```

It is to be noted here that in case of first execution of the program, as the input is number zero, therefore a message is displayed due to the logical test `(r == 0)` having truth value "`True`". Whereas, when in second execution of the program a non-zero number is input, nothing appears as a message, because there is no command to be executed if the logical test gives truth value "`False`".

Let us consider a situation when it is required to execute certain sequence of commands for one outcome of the logical test whereas another sequence of commands to be executed for another outcome. Then it is appropriate to use another branching statement, as detailed below.

### The `if-else` statement

This statement allows executing **exactly one out of two** desirable sequences of actions depending on if a certain logical condition gets **satisfied** or **not**. The syntax for this statement is

```
if (logical test) then
     Scilab Command A1
     Scilab Command A2
     ...
else
```

```
        Scilab Command B1
        Scilab Command B2
        ...
    end
```

The syntax of "`if-else` statement" in Scilab as given above, first evaluates the condition given through the expression "`logical test`". If the output of this evaluation comes out with Boolean value "`True`", then the Scilab commands `Scilab Command A1...` written between the keywords `then` and `else` are executed sequentially. Whereas, for the case, when the output of this evaluation is comes out with Boolean value "`False`", then the Scilab commands `Scilab Command B1...` written between the keywords `else` and `end` are executed sequentially.

Let us understand this concept through another small program coded in Scilab editor "SciNotes".

**Example 3.2:** The following program having an "`if-else` statement" tests whether a given real number is equal to zero. It then displays a message for the case when output of the logical test is "`True`" (*i.e.*, when the number is equal to zero), whereas, in case of output of the logical test "`False`", displays another message.

```
clear
r = input("Enter a real number ")
if (r == 0) then
    disp("The given number is equal to zero.")
else
    disp("The given number is not equal to zero.")
end
```

The output of the above program upon execution is demonstrated below for two different inputs.

Output:

```
-->exec('D:\SCILAB\Ch3_1_Ex2_if_else_Statement.sce', -1)
Enter a real number 0.001
  The given number is not equal to zero.
-->exec('D:\SCILAB\Ch3_1_Ex2_if_else_Statement.sce', -1)
Enter a real number 0
  The given number is equal to zero.
-->
```

It is to be noted here that different messages are displayed in both the executions of the program, depending upon the truth value of the logical test `(r == 0)` as "`True`" or "`False`".

A branching statement with multiple logical tests can also be used sequentially for execution of different sequences of commands for different outcomes of logical tests. Such a statement is explained below.

**The `if-elseif-else` statement**

This statement allows multi-stage testing for execution of **exactly one** sequence of actions, among many, depending on which combination of logical conditions gets **satisfied** or **dissatisfied**. A basic syntax for this statement is

```
if (logical test 1) then
     Scilab Command A1
     Scilab Command A2
     ...
elseif (logical test 2) then
     Scilab Command B1
     Scilab Command B2
     ...
else
     Scilab Command C1
     Scilab Command C2
     ...
end
```

The syntax of "`if-elseif-else` statement" in Scilab as given above, first evaluates the condition given through the expression "`logical test 1`". If the output of this evaluation comes out with Boolean value "`True`", then the Scilab commands `Scilab Command A1...` written between the keywords `then` and `elseif` are executed sequentially. Whereas, for the case, when the output of this first evaluation is comes out with Boolean value "`False`", then further the expression "`logical test 2`" is evaluated. If the output of this second evaluation comes out with Boolean value "`True`", then the Scilab commands `Scilab Command B1...` written between the keywords `elseif` and `else` are executed sequentially. Whereas, for the case, when the output of this second evaluation is comes out with Boolean value "`False`", then the Scilab commands `Scilab Command C1...` written between the keywords `else` and `end` are executed sequentially.

Let us understand this concept through a small program coded in Scilab editor "SciNotes".

**Example 3.3:** The following program having an "`if-elseif-else` statement" tests whether a given real number is zero, positive, or negative.

```
clear
r = input("Enter a real number ")
if (r == 0) then
    disp("The given number is equal to zero.")
elseif (r > 0) then
    disp("The given number is positive.")
else
    disp("The given number is negative.")
end
```

The output of the above program upon execution is demonstrated below for three different inputs.

Output:

```
-->exec('D:\SCILAB\ Ch3_1_Ex3_if_elseif_else_Statement.sce', -
1)

Enter a real number 2

The given number is positive.

-->exec('D:\SCILAB\ Ch3_1_Ex3_if_elseif_else_Statement.sce', -
1)

Enter a real number -5

The given number is negative.

-->exec('D:\SCILAB\ Ch3_1_Ex3_if_elseif_else_Statement.sce', -
1)

Enter a real number 0

The given number is equal to zero.

-->
```

It is to be noted here that different messages are displayed in each of the executions of the program, depending upon the truth value of the logical test `(r == 0)` and further of the logical test `(r > 0)`.

A branching statement with multiple logical tests can also be used sequentially for execution of different sequences of commands for different outcomes of logical tests. Such a statement is explained below.

### The `select` statement

The select statement allows combining multi-stage testing several branches in a clear and simple way.

```
select n
case i1 then
     Scilab Command A1
     Scilab Command A2
     ...
case i2 then
     Scilab Command B1
     Scilab Command B2
     ...
     ...
else
     Scilab Command C1
     Scilab Command C2
     ...
end
```

The syntax of "`select` statement" in Scilab as given above, takes the value `n` and keeps matching with evaluations `i1` , `i2`, ... and implements the sequence of commands exactly for that case whose evaluation matches the value `n`. Otherwise, the Scilab commands written between the keywords `else` and `end` are executed sequentially.

Let us understand this concept through a small program coded in Scilab editor "SciNotes".

**Example 3.4:** The following program uses the "`select` statement" to check multiple test condition and executes the Scilab commands accordingly.

```
clear
r = input("Enter the Roll Number ")
select r
case 1 then
    disp("Your Roll Number is 1.")
case 2 then
    disp("Your Roll Number is 2.")
else
    disp("Entered Roll Number is not in the list.")
end
```

The output of the above program upon execution is demonstrated below with two different inputs.

Output:

```
-->exec('D:\SCILAB\Ch3_1_Ex4_select_Statement.sce', -1)
Enter the Roll Number 1
 Your Roll Number is 1.
```

```
-->exec('D:\SCILAB\Ch3_1_Ex3_ select_Statement.sce', -1)
Enter the Roll Number 5
 Entered Roll Number is not in the list.
-->
```

## Exercise 3.1

1. Write a Scilab program to test whether a given number divides the other given number.
2. Write a Scilab program to test whether a given number is even or odd.
3. Write a Scilab program to test whether a given number is purely real number or a complex number.
4. Write a Scilab program to test whether a given number is positive, negative, or zero.
5. Write a Scilab program to test whether a given number is positive, negative, or zero, using select statement.
6. Write a Scilab program to solve a Quadratic Equation $ax^2 + bx + c = 0$. The input to the function are the values "$a, b, c$" and the output of the function should be in the variable names "$p, q$" appropriately declared.

## 3.2 Looping statements

Looping statements are used to enable a program to repeat a sequence of commands for a finite number of times. There are two types of such statements available in Scilab. These are appropriately used depending on the situation that whether the termination of the loop is known exactly through the number of repetitions of commands or expressed in terms of some logical test condition.

### 3.2.1 The `for` statement

This statement allows executing desirable sequence of actions depending on if a certain logical condition gets satisfied. For any variable name `i` and a vector of values `v`, the syntax of the "`for` statement" is given below.

```
for i = v
     Scilab Command 1
     Scilab Command 2
     ...
end
```

Syntax of the "`for` statement" in Scilab as given above, indicates that the sequence of Scilab commands will be executed repeatedly sequentially for each value of the vector `v` assigned to the variable `i`. Let us learn this concept through some examples.

**Example 3.5:** The following program computes the sum of first 5 natural numbers.

```
Sum = 0
for i=1:5
    Sum = Sum + i
end
disp("Final Sum")
disp(Sum)
```

In the above program coded in SciNotes, the variable `i` takes repeatedly and sequentially values from the vector `1:5` and corresponding to each value taken by the variable `i`, it is added to the variable `Sum`. Each time in the loop, the value of `i` is added to the variable `Sum`, the variable `Sum` gets value of the partial sum to finally get the desired value at the termination of the loop after variable `i` completing the commands for value `i=5`. The output of this program is as following.

```
-->exec('D:\SCILAB\Ch3_2_Ex5_for_loop.sce', -1)

Final Sum

    15.
```

In the Example 3.4 discussed above, the basic colon operator is used. General colon operator with specific step length can also be used in "`for` loop", as demonstrated in the next example.

**Example 3.6:** The following program demonstrates the sum of odd natural numbers from 1 to 10.

```
Sum = 0
for i=1:2:10
    Sum = Sum + i
end
disp("Sum of odd natural numbers between 1 and 10")
disp(Sum)
```

In the above program coded in SciNotes, the variable `i` takes repeatedly and sequentially values from the vector `1:2:10` (*i.e.*, odd numbers between 1 and 10) and corresponding to each value taken by the variable `i`, it is added to the variable `Sum`. The output of this program in terms of the final value of variable `Sum` gives the desired results, which is depicted as following.

```
-->exec('D:\SCILAB\Ch3_2_Ex6_for_loop.sce', -1)

Sum of odd natural numbers between 1 and 10

    25.
```

In the Examples 3.4 and 3.5 discussed above, the colon operators are used appropriately to define vectors. Even, any vector of values can in general be used in "`for` loop", as demonstrated in the next example.

**Example 3.7:** The following program demonstrates the sum of values of a pre-defined vector.

```
Sum = 0
v = [1, 9, %e, 5]
for i=v
    Sum = Sum + i
end
disp("Sum of values in the vector v:")
disp(Sum)
```

In the above program coded in SciNotes, the variable `i` takes repeatedly and sequentially values from the vector `v` and corresponding to each value taken by the variable `i`, it is added to the variable `Sum`. The output of this program in terms of the final value of variable `Sum` gives the desired results, which is depicted as following.

```
-->exec('D:\SCILAB\Ch3_2_Ex7_for_loop.sce', -1)
```

```
Sum of values in the vector v:

     17.718282
```

**Exercise 3.2**

1. Write a Scilab program to compute sum of first 'n' natural numbers.
2. Write a Scilab program to compute factorial of a natural number 'n'.
3. Write a Scilab program to obtain the Fibonacci sequence with 'n' members, and Fibonacci series with 'n' terms.
4. Write a Scilab program to test whether a given number is prime number or not.
5. Write a Scilab program using `for` loop to compute the sum of two given matrices, if they are of comparable order.
6. Write a Scilab program using `for` loop to compute the matrix multiplication of two given matrices, if they are of comparable order. Verify the obtained matrix by using Scilab matrix multiplication operator '*'.
7. Write a Scilab program to sorting (arrange) a set of numbers in ascending and descending order.
8. Write a Scilab program to compute the number of permutations & number of combinations for given values of 'n' and 'r'.

### 3.2.2 The `while` statement

Some situations appropriate to looping statements are encountered some times during the programming where it the termination point for the loop cannot be identified in advance as an exact number, unlike the `for` loop. Rather, in such cases, the termination is identified through some criterion which is expressible as a logical test condition. Such situations are appropriate to be programmed as a "`while` statement".

The general format of `while` statement is

```
while (logical test)
     Scilab Command 1
     Scilab Command 2
     ...
end
```

Let us learn use of `while` statement through the following example.

**Example 3.8:** The following program computes the sum of digits of a natural number.

```
n=input("Enter a natural number: ")
t=n
Sum=0
while (t~=0)
    digit = pmodulo(t,10)
    Sum=Sum+digit
    t=int(t/10)
end
disp("The sum of digits of the number " + string(n) + " is "+
string(Sum)+".")
```

The output upon the execution of this program is as following.

```
-->exec('D:\SCILAB\Ch3_2_Ex8_while_loop.sce', -1)

Enter a natural number: 145

 The sum of digits of the number 145 is 10.
```

## Exercise 3.3

1. Write a Scilab program to find the number of digits of a natural number '*n*'.
2. Write a Scilab program to obtain a number with digits as the reverse of a given natural number '*n*'.
3. Write a Scilab program to test whether a given number is Palindrome.
4. Write a Scilab program to test whether a given number is Armstrong number.
5. Write a Scilab program to obtain the binary equivalent of a given decimal number.
6. Write a Scilab program to obtain the decimal equivalent of a given binary number.
7. Write a Scilab program to compute sum of first '*n*' prime numbers.

# Chapter 4: Functions

Functions in Scilab are programs developed to obtain mathematical outputs upon their execution. Due to their mathematical outputs, it becomes more appropriate to use function for further mathematical use computations. For example, if outputs of programs 4 and 5 of Exercise 3.3 are in the form of a string, then it is difficult to use these outputs for further processing of information drawn through their outputs. Rather, if the output of both the programs be obtained in terms of numbers or Boolean variables, then the further processing of outputs becomes convenient. To understand this, let us test whether a given number is a Palindrome and Armstrong too. If the output of both the programs are expressed in terms of numbers or Booleans instead of messages conveyed as strings, then their conclusions can further be treated through mathematical or logical operators. Knowing the usefulness of function programs, let us learn first about how to call a function and use its output(s).

## 4.1 Calling a function

There are three main components of call of a Scilab function, as listed below.

- name of the function
- input argument(s)
- output argument(s).

The calling sequence of any function in most general form is as following:

```
[y1, ... ,yn] = function_name(x1, ... , xn)
```

**Remarks:** In the calling sequence of a function,

1. the function name is `function_name`;
2. input arguments are `x1, ... , xn`;
3. output arguments are `y1, ... ,yn`;
4. if there are multiple input or output arguments, then they should to be separated by comma (,);
5. input argument(s) are to be enclosed in parenthesis (generally called round brackets), *i.e.*, "(" and ")";
6. input argument(s) are to be enclosed in square brackets (or simply called brackets), *i.e.*, "[" and "]";

**Caution:** Any space should not be given after the function name and after the left parenthesis.

**Built-in functions**

There are multiple built-in functions in Scilab. Some of them we have discussed in Section 1.4 of Chapter 1. Built-in functions are simply to be called for getting the values of their output arguments by simply entering their correct calling sequence.

## 4.2 Defining a function

Scilab has been developed so that it can be extended by the users. Users can develop their own programs as functions for better mathematical application. In this section, let us learn the procedure and syntax of coding a function program in Scilab. The coding of a function program is majorly like any general program except some typical differences, which are as following.

1. The code of any function program has to start with the keyword `function` and end with the keyword `endfunction`.
2. As function programs are intended to give mathematical outputs, so the desirable outputs are needed to be assigned to the variables written as output arguments.
3. As function programs, once developed, are used for multiple values of inputs arguments, therefore whichever variables are used as inputs they should be included as input arguments.

Syntax of a general function program code is as following:

```
function [y1, ... ,yn] = function_name(x1, ... , xn)
     Scilab Command 1
     Scilab Command 2
     ...
     y1 = assign value
     ...
     yn = assign value
endfunction
```

**Remarks:**

1. Space must be provided after output arguments and the assignment operator sign "=".
2. The name of the function must abide the rules of defining a valid variable name in Scilab.
3. User defined function names should be avoided to be same as built-in function names.

4. All the output variables must be assigned values in the code of the function program.
5. A used defined function program is required to be executed before being called for the first time. Execution is required again before calling the function program for every time when the code is edited.
6. General Scilab programs are saved in the permanent memory of the computer with the file extension ".sce". Whereas, the function programs in Scilab are saved with the file extension ".sci". Over that, the file name of the program file must be same as the function name.

Let us learn this through an example of a function program developed by modifying the program in Example 3.3 in Chapter 3.

```
function [y]=signum(x)
    if (x == 0) then
        y = 0;
    elseif (x > 0) then
        y = 1;
    else
        y = -1;
    end
endfunction
```

This function can be executed and then called for different inputs as demonstrated below.

```
-->exec('D:\SCILAB\signum.sci', -1)

-->[y] = signum(5)

 y  =

     1.

-->[y] = signum(-10)

 y  =

   - 1.

-->[y] = signum(0)

 y  =

     0.
```

**Remark:** Variable names used in input and output arguments are local to the function program. The interpretation of this concept demonstrated through the above-discussed function program is given below.[4]

1. Although, `x` is the input variable for the above function program but it can be called using any other variable name also. For example, for the above function program:

```
-->a = 5;

-->[y] = signum(a)

 y  =

     1.
```

   This feature is due to a mechanism of Scilab explained as following. During the call of function program `signum` with variable a used as input argument, Scilab assigns the value of variable `a` to the local variable `x` and then computes the value of variable `y` pertaining to the output argument.

2. Similarly, any variable name can be used as output variable during the call of a function program. For example, for the above function program:

```
-->[b] = signum(a)

 b  =

     1.
```

   This feature is due to another mechanism of Scilab explained as following. During the call of function program `signum` with input argument assigned value directly or through variable, Scilab computes the value of variable `y` pertaining to the output argument. It then assigns the computed value of variable `y` to the variable `b`, because the call of the function program uses output argument as `b`.

## Exercise 4

1. Write a Scilab program to solve a Quadratic Equation $ax^2 + bx + c = 0$. The input to the function are the values "$a, b, c$" and the output of the function should be in the variable names "$p, q$" appropriately declared.
2. Write a Scilab program to compute sum of first 'n' natural numbers.
3. Write a Scilab program to compute factorial of a natural number 'n'.

[4] Same applies to built-in functions of Scilab also.

4. Write a Scilab program using for loop to compute the sum of two given matrices, if they are of comparable order.
5. Write a Scilab program to compute the number of permutations & number of combinations for given values of ‘n’ and ‘r’.
6. Write a Scilab program to compute sum of digits of a natural number ‘n’.
7. Write a Scilab program to find the number of digits of a natural number ‘n’.
8. Write a Scilab program to obtain a number with digits as the reverse of a given natural number ‘n’.
9. Write a Scilab program to test whether a given number is Palindrome.
10. Write a Scilab program to test whether a given number is Armstrong number.
11. Write a Scilab program to obtain the binary equivalent of a given decimal number.
12. Write a Scilab program to obtain the decimal equivalent of a given binary number.

# Chapter 5: Plotting

One of most useful and rapid ways of analysing data is through plots and graphics. Supplementing reports on data analysis with plots and graphics is most prevalently used practice since long in industry, academia, and research.

Scilab provides multiple built-in features for creating and customizing various types of plots and charts. In this chapter, we will learn about creating 2D plots, contour plots and 3D plots. In the final section of this chapter, we will learn about graphics features available in Scilab for customizing the plots by labelling them with the title and and axes lables, and the legend of our graphics.

Scilab enables plotting of multiple charts like 2-dimensional plots, Contour plots, surface (3-dimensional) plots, histograms, bar charts, etc. The precise details of each of these can be accessed through the help document of Scilab. Details are presented here only for some of these basic plots while keeping into consideration the learning objectives of beginners.

## 5.1 The 2-dimensional plot

This plotting feature of Scilab enables to produce a plot of the curve of a real-valued function defined on a closed bounded interval. Of course, x-axis is used for depicting the independent variable and y-axis for the dependent variable.

The 2-dimentional plot for any function $f:[a,b]\to\mathbb{R}$, is obtained by joining multiple points on x-y plane which lie on the graph of the function (*i.e.*, $\{(x,f(x)):x\in[a,b]\}$). The steps for this are as following.

**Step 1.** First, the function is defined as a Scilab function. For example, for obtaining the 2D plot of the function $f:[-2,2]\to\mathbb{R}$ as $f(x)=x^2$. The function be defined as:

```
function f_x=f(x)
    f_x = x.^2
endfunction
```

(Here, the output variable of the Scilab function program is computed using the element-wise power. The reason for the same will get clear in Step 3.)

**Step 2.** For obtaining such multiple points of the graph of the function (say, $n$ in mumber), a vector of values in the domain $[a, b]$ are taken.

For example, for predefined values of $a$, $b$, and $n$, the vector may be obtained by the command:

```
xv = linspace(a, b, n)
```

**Step 3.** Then, the vector of corresponding image values can be obtained by calling the function defined in Step 1. For example,

```
yv = f(xv)
```

(Here it should be observed that the input to above function is a vector, therefore, the definition of function used appropriately the element-wise power to accommodate squaring of the values of each element of the input vector.)

**Step 4.** Finally, the plotting of the points on graph thus obtained is done by using the built-in function `plot` as demonstrated below.

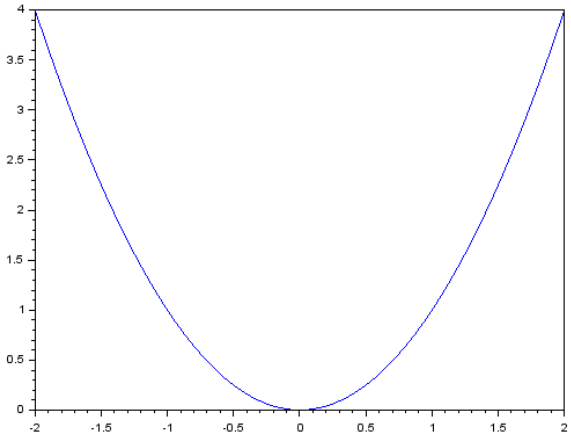
```
plot(xv, yv)
```

**Example 5.1:** Let us sequence these commands as a Scilab program to obtain the plot of the function $f(x) = x^2$, as defined in Step 1.

**Program:**

```
function f_x=f(x)
    f_x = x.^2
endfunction
a = -2; b = 2; n = 51
xv = linspace(a, b, n)
yv = f(xv)
plot(xv, yv)
```

Upon execution of this program, following figure appears as a 2D plot in the graphics window of Scilab.




## 5.2 Contour plot

Closed curves on the 2-dimensional plane are called contours. Equation of a contour is represented generally through implicit relation between variables $x$ and $y$ of the form:

$$f(x, y) = \text{constant} \tag{1}$$

In the equation given above, different curves are represented for different values of the constant. This indicates it as a family of contours across different values of constant. Contours can be related with surface plots (discussed in the next section) as level curves on a 3D surface. The reason behind this is that a 3D surface is represented through the equation

$$z = f(x, y). \tag{2}$$

Therefore, for each real number value of the constant in equation (1), contour represents closed curve(s) obtained by slicing the surface (2) parallel to the x-y plane at the level $z$ (*i.e.*, keeping the variable $z$ as constant).

The plotting of contours is thereby appropriately visualized on x-y plane. Also, in view of the above discussion, the computation for the purpose of plotting is done by taking both $x$ and $y$ as independent variables, then obtaining values for the variable $z$ using the equation (2), and then using an appropriate plotting function.

The steps for this are as following.

**Step 1.** First, the function $f(x, y)$ is defined as a Scilab function with two input arguments. For example, for obtaining contour plots for the function $f(x, y) = x^2 + y^2$ over the 2-dimensional region given by $[-2, 2] \times [-2, 2]$. The function be defined as:

```
function z=f2(x, y)
    z = x.^2 + y.^2
endfunction
```

(Here, the output variable of the Scilab function program is computed using the element-wise power due to the same reason as explained in the previous section.)

**Step 2.** Following the theory discussed above, vectors of values both $x$ and $y$ variables are obtained in their respective domains say, $[a, b] \times [c, d]$.
For example, for predefined values of $a$, $b$, $c$, $d$, and $n$, vectors may be obtained by the commands:

```
xv = linspace(-2, 2, 50)
yv = linspace(-2, 2, 50)
```

**Step 3.** Then, the vector of corresponding image values can be obtained by calling the function defined in Step 1. For example,

```
zv = f2(xv, yv);
```

**Step 4.** Finally, contours are plotted using the built-in function `contour` as demonstrated below.

```
contour(xv, yv, f2, n)
```

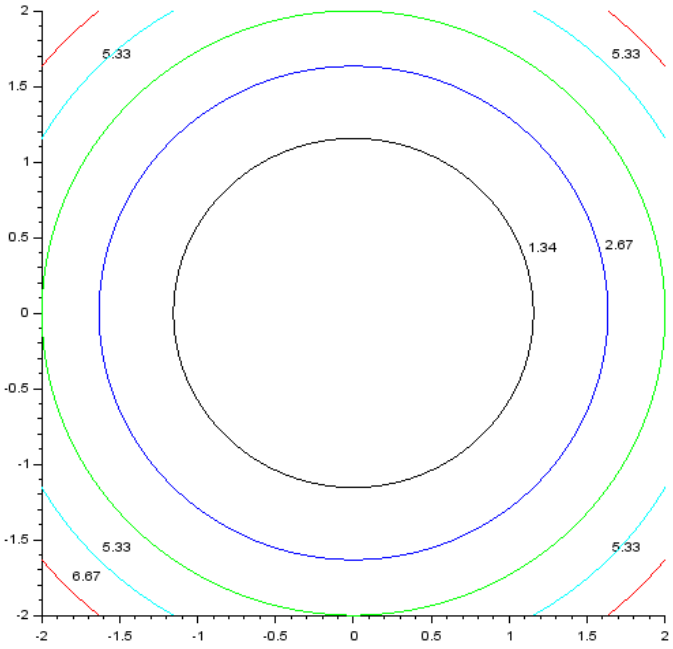
Here, `n` represents the number of contours to be plotted.

**Example 5.2:** Let us sequence these commands as a Scilab program to obtain the contour plot of the function $f(x) = x^2 + y^2$, as defined in Step 1.

**Program:**

```
function [z]= Contour_Plot()
    xdata = linspace(-2, 2, 50);
    ydata = linspace(-2, 2, 50);
    z = f2(xdata, ydata);
    xtitle("Contours for the function x^2 + y^2", "X-
axis", "Y-axis")
    contour(xdata, ydata, f2, 5)
endfunction
```

Upon execution of this program, following figure appears as contour plot in the graphics window of Scilab.




## 5.3 The 3-dimensional plot (Surface plot)

Suppose that a 3D surface be represented through the equation

$$z = f(x, y). \tag{2}$$

The plotting of 3D surface is done by performing the steps as given below.

**Step 1.** First, the function $f(x, y)$ is defined as a Scilab function with two input arguments. For example, for obtaining the 3-D plot for the surface $z = f(x, y) = xy(\sin x + 2\cos y)$ over the 2-dimensional region given by $[-2, 2] \times [-2, 2]$. The function be defined as:

```
function z=f2(x, y)
    X.*Y.*(sin(X) + 2*cos(Y));
endfunction
```

(Instead of defining the function separately, it may alternatively be used directly along the main program.)

**Step 2.** Following the same theory as discussed in the previous section, vectors of values both $x$ and $y$ variables are obtained in their respective domains say, $[a, b] \times [c, d]$.
For example, for predefined values of $a$, $b$, $c$, $d$, and $n$, vectors may be obtained by the commands:

```
xv = linspace(-2, 2, 50)
yv = linspace(-2, 2, 50)
```

**Step 3.** A meshgrid is to be created for obtaining points $(x, y)$ with coordinates given by components of vectors created in the previous step. The built-in function `meshgrid` is used for this purpose for creating the cross product of vectors `xv` and `yv`, as demonstrated below.

```
[X,Y] = meshgrid(xdata, ydata);
```

**Step 4.** Then, the vector of corresponding image values can be obtained by calling the function defined in Step 1. For example,

```
zv = f2(xv, yv);
```

**Step 5.** Finally, the 3-dimensional surface is plotted using the built-in function `surf` as demonstrated below.

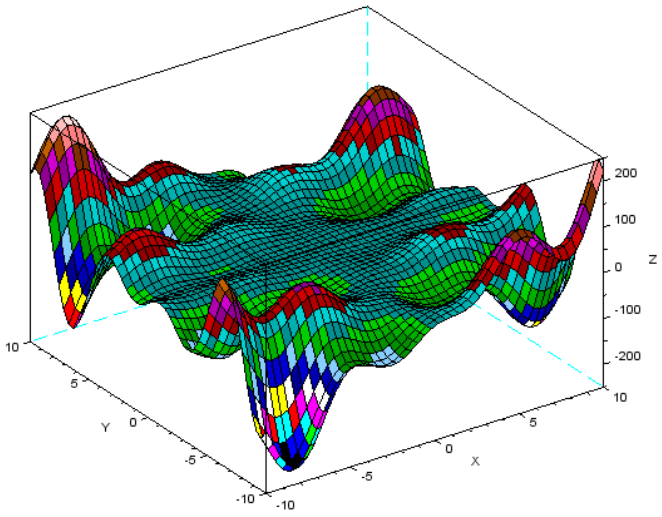
```
surf(xv, yv, zv)
```

**Example 5.3:** Let us sequence these commands as a Scilab program to obtain the 3-D plot of the surface given by $z = xy(\sin x + 2\cos y)$, as defined in Step 1.

**Program:**

```
function [X, Y, Z]=Surface_Plot()
    xdata = linspace(-10, 10, 50)
    ydata = linspace(-10, 10, 50)
    [X,Y] = meshgrid(xdata, ydata);
    Z = X.*Y.*(sin(X) + 2*cos(Y));
    surf(X,Y,Z)
endfunction
```

Upon execution of this program, following figure appears as a 3-D plot in the graphics window of Scilab.




## 5.4 Titles, axis, legends and Style options

Scilab is equipped with a wide variety of graphics features. In this section, we will learn about the most basic features, particularly, which enable the user to configure the title, axis and legends on x-y plot.

### Title

The title command enables to give the introductory detail of the figure in the form of a message coded as a string. The syntax of this command is

```
title("Title as a string")
```

### Axis

To configure the labelling of both the axis of a 2-D plotting, appropriate messages (coded as a strings) can be used through the command having a syntax as demonstrated below.

```
xtitle("Title as a string", "X-axis", "Y-axis")
```

### Legends

While plotting multiple curves in a single figure, labels can be given to each curve for its identification using the legends. The syntax of this command is

```
legend("String 1" , "String 2");
```

For using the `legend` command, the `plot` command is required to be used with its advanced feature by specifying it color or style option. Therefore, user needs to learn these options also for using the feature of configuring legend of a 2D-plot.

**Style options**

A more general syntax of `plot` command is as following.

```
plot(xv,yv, "Style option")
```

The style option in the plot command is a character string that consists of 1, 2 or 3 characters that specify the color and/or the line style. Different color, line-style and marker-style options are summarized in following Table.

| Color style-option | | Line style-option | | Marker style-option | |
|---|---|---|---|---|---|
| y | yellow | - | solid | + | plus sign |
| m | magenta | -- | dashed | o | circle |
| c | cyan | : | dotted | * | Asterisk |
| r | red | -. | dash-dot | x | x-mark |
| g | green | | | . | Point |
| b | blue | | | ^ | up triangle |
| w | white | | | s | square |
| k | black | | | d | diamond |

**Example 5.3:** All these graphics configurations are demonstrated through the following program for plotting of curves $y = x^2$ and $y = x^4$ in a 2D-plot.

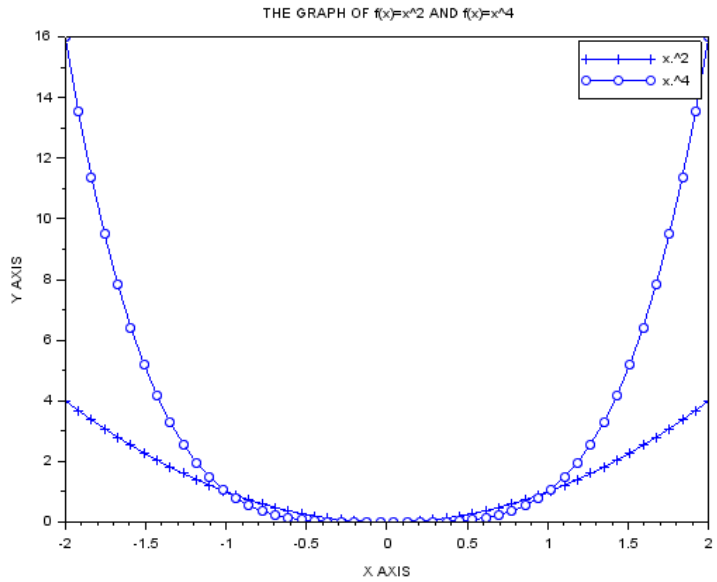
**Program:**

```
function f=myquadratic(x)
    f= x.^2
endfunction

function f=myquadratic2(x)
    f= x.^4
endfunction

xdata = linspace(-2,2,50);
ydata = myquadratic( xdata );
plot(xdata,ydata, "+-")
ydata2 = myquadratic2(xdata );
plot(xdata, ydata2 ,"o-")
xtitle("THE GRAPH OF f(x)=x^2 AND f(x)=x^4" , "X AXIS" ,
"Y AXIS" );
legend("x.^2" , "x.^4");
```

The output of this program comes out as a 2D-plot given below.




## Exercise 5

1. Write a Scilab program to obtain the plot of 2-dimensional graph of a given function with single input argument and single output argument. The given function is $f(x) = x^2$ in the domain $[-2, 2]$. Label the title of the figure and axes appropriately. Demonstrate the use of style options in the program.

2. Write a Scilab program to obtain the plot of 2-dimensional graph of two given function with single input argument and single output argument. The given function is $f(x) = x^2$ and $g(x) = x^3$ in the domain $[-2, 2]$. Label the title of the figure and axes appropriately. Demonstrate the use of style options and legends in the program to distinguish the two curves while plotting the figure.

3. Write a Scilab program to obtain plot a contour plot of a given function with two input arguments and single output argument. The given function is $f(x, y) = \frac{x^2}{4} + \frac{y^2}{9}$, where each of $x$ and $y$ to vary in the interval $[-10, 10]$. Label the title of the figure and axes appropriately.

4. Write a Scilab function program for 3-dimensional surface plot of a given function with two input arguments and single output arguments. The given function is $f(x, y) = x^2 + y^2$, where each of $x$ and $y$ to vary in the interval $[-50, 50]$. Label the title of the figure and axes appropriately.

# Chapter 6: Solving Ordinary Differential Equations

There are multiple numerical methods available to solve differential equations for approximate solutions. Programs can be coded for such methods by learning their procedural details. Also, a package of built-in functions for these methods is developed by creators of Scilab with the name `ode`. The package is detailed in the help document of Scilab. All those detailed are compiled here from the same source.

## 6.1 Solving first-order ordinary differential equations

The package `ode` which solves explicit ordinary different equations given by:

$$\frac{dy}{dt} = f(t, y)$$

$$y(t_0) = y_0.$$

**Calling sequence**

The simplest call of `ode` is:

```
y = ode(y0,t0,t,f)
```

In this calling sequence,

- `y0` is the vector of initial conditions, `t0` is the initial time,
- `t` is the vector of times at which the solution `y` is computed and `y` is matrix of solution vectors given by `y = [y(t(1)),y(t(2)),...]`.
- The input argument `f` defines the right hand side of the first order differential equation.
- This argument is a function with a specific header. If `f` is a Scilab function, its calling sequence must be

  ```
  ydot = f(t,y)
  ```

  where `t` is a real scalar (the time) and `y` is a real vector (the state) and `ydot` is a real vector (the first order derivative $\frac{dy}{dt}$).

**The Solver**

The type of problem solved and the method used depend on the value of the first optional argument `type` which can be one of the following strings:

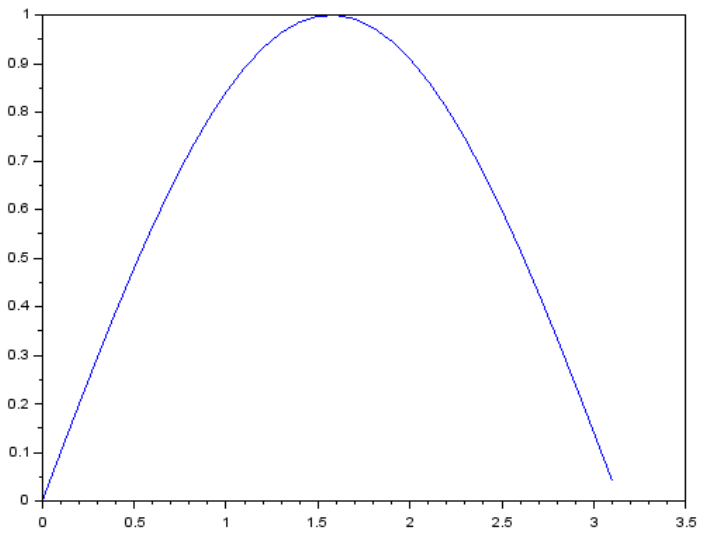
| | |
|---|---|
| `<not given>` | `lsoda` solver of package ODEPACK is called by default. It automatically selects between nonstiff predictor-corrector Adams method and stiff Backward Differentiation Formula (BDF) method. It uses nonstiff method initially and dynamically monitors data in order to decide which method to use. |
| `"adams"` | This is for nonstiff problems. `lsode` solver of package ODEPACK is called and it uses the Adams method. |
| `"stiff"` | This is for stiff problems. `lsode` solver of package ODEPACK is called and it uses the BDF method. |
| `"rk"` | Adaptive Runge-Kutta of order 4 (RK4) method. |
| `"rkf"` | The Shampine and Watts program based on Fehlberg's Runge-Kutta pair of order 4 and 5 (RKF45) method is used. This is for non-stiff and mildly stiff problems when derivative evaluations are inexpensive. This method should generally not be used when the user is demanding high accuracy. |
| `"fix"` | Same solver as `"rkf"`, but the user interface is very simple, *i.e.*, only `rtol` and `atol` parameters can be passed to the solver. * |
| `"root"` | ODE solver with rootfinding capabilities. The `lsodar` solver of package ODEPACK is used. It is a variant of the `lsoda` solver where it finds the roots of a given vector function. See help on ode_root for more details. |
| `"discrete"` | Discrete time simulation. See help on ode_discrete for more details. |

* The tolerances rtol and atol are thresholds for relative and absolute estimated errors. The estimated error on `y(i)` is: `rtol(i)*abs(y(i))+atol(i)` and integration is carried out as far as this error is small for all components of the state. If `rtol` and/or `atol` is a constant, `rtol(i)` and/or `atol(i)` are set to this constant value. Default values for `rtol` and `atol` are respectively `rtol=1.d-5` and `atol=1.d-7` for most solvers and `rtol=1.d-3` and `atol=1.d-4` for `"rfk"` and `"fix"`.

**Example 6.1:** In the following example, we solve the Ordinary Differential Equation $\frac{dy}{dt} = y^2 - y\sin t + \cos t$ with the initial condition $y(0) = 0$. We use the default solver.

```
function ydot=f(t, y)
    ydot=y^2-y*sin(t)+cos(t)
endfunction
y0=0;
t0=0;
t=0:0.1:%pi;
y = ode(y0,t0,t,f);
plot(t,y)
```

**Output:**




**Example 6.2:** In the following example, we solve the equation $\frac{dy}{dt} = Ay$. The exact solution is `y(t)=expm(A*t)*y(0)`, where `expm` is the matrix exponential. The unknown is the $2 \times 1$ matrix `y(t)`.

```
function ydot=f(t, y)
    ydot=A*y
endfunction
function J=Jacobian(t, y)
    J=A
endfunction
A=[10,0;0,-1];
y0=[0;1];
t0=0;
t=1;
```

```
y=ode("stiff",y0,t0,t,f,Jacobian)
disp("Solution given by the solver:")
disp("y = ")
disp(y)

// Compare with exact solution:
disp("Exact solution:")
disp("y = ")
disp(expm(A*t)*y0)
```

**Output:**

```
Solution given by the solver:

 y =
    0.
    0.3678794

 Exact solution:

 y =
    0.
    0.3678794
```

## 6.2 Solving second-order ordinary differential equations

As a package for solving first-order ordinary differential equations is available in Scilab, the same can be used to solve second-order ordinary differential equations also just by using a substitution procedure. In this section, we will learn the procedure for this purpose and then we can use the package learnt in previous. Let us consider a second-order ordinary differential equation with initial conditions as given below.

$$\frac{d^2y}{dt^2} + g(t)\frac{dy}{dt} = f(t,y)$$

$$y(t_0) = y_0, y'(t_0) = y_1.$$

Such a differential equation can be dealt with the substitution:

$$w = \frac{dy}{dt}.$$

This reduces the given second-order ordinary differential equation into a pair of first-order ordinary differential equations, as expressed below.

$$\frac{dy}{dt} = w$$

$$\frac{dw}{dt} + g(t)w = f(t,y)$$

This system of first-order ordinary differential equations can be arranged into a vector for as:

$$\frac{dx}{dt} = F(t,x),$$

where $x = \begin{bmatrix} y \\ w \end{bmatrix}$, $F(t,x) = \begin{bmatrix} w \\ f(t,y) - g(t)w \end{bmatrix} = [w;\ f(t,y) - g(t)w]$, and therefore

$$\frac{dx}{dt} = \begin{bmatrix} \frac{dy}{dt} \\ \frac{dw}{dt} \end{bmatrix} = \left[\frac{dy}{dt}\ ;\ \frac{dw}{dt}\right].$$

**Example 6.3:** Through this example, we solve demonstrate the procedure to simplify appropriately the second-order ordinary differential equation $\frac{d^2y}{dt^2} = \sin 2t$ with initial conditions $y(0) = 0$ and $y'(0) = -1/2$.

Procedure: Let us convert the given second-order ordinary differential equation into a first-order one, by an appropriate substitution

$$\frac{dy}{dt} = z.$$

Thereby, the given equation reduces to

$$\frac{dw}{dt} = \sin 2t.$$

This forms the vectorized first-order differential equation

$$\frac{dx}{dt} = f(t,x)$$

where $x = [y;\ \ w]$, $f(t,x) = [w;\ \sin 2t]$, and therefore

$$\frac{dx}{dt} = \left[\frac{dy}{dt}\ ;\ \frac{dw}{dt}\right].$$

The derivative function in Scilab can thus be defined as following.

```
function dx=f(t, x)
    dx(1)=x(2)+cos(t)
    dx(2)= sin(2*t)
endfunction
```

Some more examples are provided in the help document on solving ordinary differential equations by specifying other solvers as listed in the table given above. Details presented here are sufficient for learning the basics of `ode` package of Scilab and using the same for solving simpler ordinary differential equations with initial conditions. Readers interested in learning this package in a detail are suggested for referring to the `ode` package in the "Differential calculus, Integration" section of the help document.

## Exercise 6

1. Solve and plot the graph of the solution of following ordinary differential equation

$$\frac{dy}{dt} = y^2 - y\sin t + \cos t$$

with initial value $y(0) = 0$ (*i.e.*, $y_0 = 0$ at $t_0 = 0$).

2. Solve and plot the graph of the solution of following ordinary differential equation

$$\frac{dy}{dt} = 10y$$

with initial value $y(0) = 5$.

3. Solve and plot the graph of the solution of following ordinary differential equation

$$y'' = 2$$

with initial value $y(0) = 5$, $y'(0) = 6$.

# Chapter 7: Polynomials in Scilab

Polynomials play an important role in multiple areas of Mathematics. Especially, discussions of matrix theory and algebra mainly require dealing with polynomials. Scilab enables defining polynomials, their operations and handling of matrices of polynomials. Let us learn about defining polynomials in Scilab in the first section of this chapter. In the later sections, we will learn about different operations on polynomials.

## 7.1 Defining polynomials

Scilab-keyword `poly` is used for defining a polynomial with the calling sequence discussed below.

```
p = poly(a, vname, ["flag"])
```

where, the details of arguments of this calling sequence are as given below.

| | |
|---|---|
| `A` | a matrix or real number |
| `vname` | a string, for defining the symbolic variable name.<br>(The string must be maximum 4 characters.) |
| `"flag"` | string (`"roots"`, `"coeff"`), for specifying that values given in vector `a` are to be considered as roots or coefficients of the polynomial being defined.<br>default value is `"roots"`<br>Shortcuts can be also used: `"r"` for `"roots"` and `"c"` for `"coeff"`. |

Let us learn the concept of defining polynomials in some detail through the discussion given below. The discussion can be separated into two sections, depending on the type of first argument being used.

### 7.1.1 Case when the first argument `'a'` is a vector

For the case, when the first argument `'a'` is a vector, component values of the vector represent either the coefficients of the polynomial or the roots of the

polynomial, depending upon the specification through the third argument specified as "coeff" or "roots", respectively.

**Example 7.1:** For defining a polynomial $2 + 5x + 4x^3$ and $1 + 2y + 3y^2 + 4y^3$in Scilab the following commands are to be used.

```
p1 = poly([2, 5, 0, 4], "x", "coeff")

p2 = poly([1, 2, 3, 4], "y", "coeff")
```

**Output:** The output, when displayed, would appear in Scilab Console, as is demonstrated below.

```
p1  =
                3
    2 + 5x + 4x

p2  =
                2    3
    1 + 2y + 3y + 4y
```

**Example 7.2:** For defining a polynomial in Scilab, in variable $x$ which has roots four roots, namely, 2, 5, 0, and 4, the following command is to be used.

```
p3 = poly([2, 5, 0, 4], "x", "roots")
```

**Output:** The output, when displayed, would appear in Scilab Console, as is demonstrated below.

```
p3  =

              2     3    4
  - 40x + 38x - 11x + x
```

**Remark 1:** The same output can be achieved by a similar command in which the third input argument is not mentioned. This is demonstrated through the following command.

```
p3 = poly([2, 5, 0, 4], "x")
```

Another way of defining polynomials with specified coefficients and specified symbol variable is by first defining the seed for polynomial and then defining the polynomial as demonstrated below.

For defining a polynomial in a symbol variable 's', the seed for polynomial is defined using the following command.

```
s = poly(0,"s");
```

This command given above to define the seed for polynomial basically defines `s` as a polynomial in symbol variable `s`. A polynomial in the symbol variable 's' using this seed for polynomial can be defined using the following command.

```
p = 1+s+2*s^2;
```

This defines the polynomial $p = 1 + s + 2s^2$ in symbol variable "$s$".

**Remark 2:** Polynomials can be defined by either of the procedure as described above in the main discussion vis-à-vis through the procedure described in the remark 1. The same can be verified through a small exercise performed in the Console itself, as demonstrated below.

```
-->s=poly(0,"s")
 s  =
     s
 -->p=1+s+2*s^2
 p  =
                2
    1 + s + 2s
 -->p1 = poly([1, 1, 2], "s", "coeff")
 p1  =
                2
    1 + s + 2s
-->p==p1
 ans  =
  T
```

### 7.1.2 Case when the first argument 'a' is a 2-dimensional matrix

For the case, when the first argument `'a'` is a 2-dimensional matrix, the calling sequence

```
p = poly(a, "s")
```

gives the characteristic polynomial corresponding to the matrix stored in the variable name 'a'.

For example,

```
-->A=ones(2,2)

 A  =

    1.    1.

    1.    1.

-->poly(A,"x")

 ans  =

          2
  - 2x + x

-->B = [0, 1; 1, 0]

 B  =

    0.    1.

    1.    0.

-->poly(B,"x")

 ans  =

         2
  - 1 + x
```

## 7.2 Matrices of polynomials

The way matrices of real or complex numbers can be defined in Scilab, matrices of polynomials can also be defined in this software. For this purpose, first the seed for polynomials is needed to be defined, and then matrices of polynomials the defined symbol variable can be defined using the same rules as we define matrices of numbers. This procedure is demonstrated below through a Scilab session in Console.

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

 s  =
```

```
    s
-->M1 = [1+s, 1+s^2; s-s^3, 2+s]
 M1  =
                     2
    1 + s     1 + s
          3
    s - s     2 + s
```

## 7.3 Operations on polynomials or matrices of polynomials

### 7.3.1 Operations on polynomials

Algebraic operations on the polynomials can be performed with their natural symbols, as are used for numbers or matrices. The addition, subtraction, multiplications, division of polynomials is demonstrated through a Scilab session in Console.

```
-->p1 = poly([1, 2, 3, 4], 'x', 'coeff')
 p1  =
                2    3
    1 + 2x + 3x + 4x
-->p2 = poly([2, 4, 0, 9, 5], 'x', 'coeff')
 p2  =
                3    4
    2 + 4x + 9x + 5x
-->p3 = p1 + p2
 p3  =
                2     3    4
    3 + 6x + 3x + 13x + 5x
-->p4 = p1 - p2
```

```
 p4  =

                2    3    4

  - 1 - 2x + 3x - 5x - 5x

-->p5 = p1 * p2

 p5  =

                 2     3     4     5     6     7

    2 + 8x + 14x + 29x + 39x + 37x + 51x + 20x

-->p6 = p1/p2

 p6  =

                2    3

    1 + 2x + 3x + 4x

    ----------------

                3    4

    2 + 4x + 9x + 5x


-->p7 = 2*p1

 p7  =

                2    3

    2 + 4x + 6x + 8x
```

## 7.3.2 Operations on matrices of polynomials

Appropriate algebraic operations on the matrices of polynomials can be performed with their natural symbols, as are used for matrices of numbers. The addition, subtraction, multiplications, division of polynomials is demonstrated through a Scilab session in Console.

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

 s  =

    s
```

```
-->M1 = [1+s, 1+s^2; s-s^3, 2+s]

 M1  =

                   2
    1 + s     1 + s

          3
    s - s     2 + s

-->M2 = [1+s^2, -s; s^3, -2+s^2+s^3]

 M2  =

          2
    1 + s       - s

      3                2   3
    s          - 2 + s + s

 -->M1+M2

 ans  =

               2              2
    2 + s + s     1 - s + s

                         2   3

    s              s + s + s

-->M3 = 2*M1

 M3  =

                        2
    2 + 2s      2 + 2s

             3
    2s - 2s     4 + 2s

-->M4 = M1 - M2

 M4  =

          2               2
    s - s      1 + s + s

          3              2   3
    s - 2s     4 + s - s - s
```

```
-->M5 = M1*M2

 M5  =

              2    3   5                2   3   4   5
    1 + s + s + 2s + s   - 2 - s - 2s + s + s + s

          3   4   5                   2    3    4
    s + 2s + s - s       - 4 - 2s + s + 3s + 2s
```

## 7.4 Evaluation of polynomials

The evaluation of polynomials at some values of matrices is done in Scilab using the Scilab built-in function `horner`. The calling sequence of this function is as following.

```
horner(P,x)
```

In this calling sequence, the arguments have interpretation given by:

`P` polynomial or rational matrix

`x` array of numbers or polynomials or rationals

This function evaluates the polynomial or rational matrix `P = P(s)` when the variable `s` of the polynomial is replaced by `x`:

```
horner(P,x)=P(x)
```

Some examples from a Scilab session demonstrate the working of this function.

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

-->P=1 + 2*s + 3*s^2

P  =

                2

    1 + 2s + 3s

-->horner(P,1)

 ans  =

    6.

-->horner(P,0)
```

```
 ans  =

    1.
// Evaluation of polynomial at another polynomial
-->horner(P,s)
 ans  =

                2
    1 + 2s + 3s
-->horner(P,s^2)
 ans  =

           2    4
    1 + 2s + 3s
-->x=poly(0,'x')
 x  =

    x
-->horner(P,x)
 ans  =

                2
    1 + 2x + 3x
// Evaluation of polynomial at a vector
-->horner(P,[1 2 5])
 ans  =

    6.    17.    86.
// Evaluation of polynomial at a matrix
-->A=eye(2, 2)
 A  =
```

```
    1.    0.

    0.    1.

-->horner(P, A)

 ans  =

    6.    1.

    1.    6.

// Evaluation of polynomial at a complex number

-->horner(P,%i)

 ans  =

  - 2. + 2.i

// Evaluation of a matrix of polynomials or rationals

-->M = [s, 1/s]

-->horner(M, 1)

ans  =

    1.    1.

-->horner(M,1/s)

 ans  =

    1     s

    -     -

    s     1

// Evaluation of a polynomial for a matrix of numbers

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

 X  =

    1.    2.

    3.    4.
```

```
-->p

 p  =

               2
    1 + s + 2s

-->p=poly(1:3,'x','c')

 p  =

                2
    1 + 2x + 3x

-->m=horner(p, X)

 m  =

    6.     17.

    34.    57.

-->1*X.^0+2*X.^1+3*X.^2

 ans  =

    6.     17.

    34.    57.
```

**Observation:** The last command confirms that the `horner` function evaluates a polynomial on a matrix, in a pointwise manner. The same may be confirmed through the following commands in a continuation of previous ones.

```
-->Y= [1 2 3; 4 5 6]

 Y  =

    1.    2.    3.

    4.    5.    6.

-->m1=horner(p, Y)

 m1  =

    6.     17.    34.

    57.    86.    121.
```

```
-->1*Y.^0+2*Y.^1+3*Y.^2

 ans  =

    6.      17.     34.

    57.     86.     121.
```

There are many more built-in functions enabling mathematical treatment of polynomials. Listing and explaining each one of them with only add pages in the book. As details of each of such functions can be explored through the help document also of the software, therefore it would be sufficient to conclude the chapter here with ample demonstration of basics of the topic. Readers are suggested to explore the "Polynomials" section of the help document for acquiring further knowledge on additional functions.

## Exercise 7

1. Write a Scilab function program to perform the following:

   (a) Define and display two polynomials in $x$, as

   $$p_1(x) = 1 + 2x + 3x^2 + 4x^3,$$

   $$\text{and } p_2(x) = 2 + 4x + 9x^3 + 5x^4.$$

   (b) Obtain and display the sum, difference, product, and fraction of above two polynomials, as

   $$p_3(x) = p_1(x) + p_2(x),$$

   $$p_4(x) = p_1(x) - p_2(x),$$

   $$p_5(x) = p_1(x) * p_2(x),$$

   $$p_6(x) = \frac{p_1(x)}{p_2(x)}.$$

   (c) Evaluate each of above defined six polynomials at a given value or a matrix (pointwise, in case of matrices).

   Call this function for testing with input as (1) $x = 1$, (2) $x = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 1 \end{bmatrix}$.

2. Write a Scilab program to:

   (a) Obtain the characteristic polynomial as $p_A(x)$ for a given matrix $A$ as an input argument.

(b) Obtain the characteristic roots of the given matrix $A$ by solving the equation $p_A(x) = 0$ for its roots by calling the Scilab in-built function `roots`.

(c) Obtain characteristic roots of the given matrix $A$, the `spec` function on $A$.

(d) Test whether the characteristic roots obtained by method (b) and (c) are equal.

(e) Call this function for testing with input as each of the following three matrices

(1) $A = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$,

(2) $A = \begin{bmatrix} 1 & 2 \\ 0 & 2 \end{bmatrix}$,

(3) $A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 2 & 4 \\ 0 & 0 & 4 \end{bmatrix}$.