UNIT 8 : INSTALLATION OF SCILAB

UNIT STRUCTURE

8.1 Learning Objectives

8.2 Introduction

8.3 Scilab

8.3.1 Installation of Scilab in Windows

8.3.2 Installation of Scilab in Linux

8.4 Let Us Sum Up

8.5 Further Reading

8.6 Answers to Check Your Progress

8.7 Model Questions

8.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- install Scilab in Windows
- install Scilab in Linux

8.2 INTRODUCTION

In the previous units you have been introduced to different applications of Latex like writing resume, writing question paper, writing articles/research papers. Applications like presentation using beamer, inserting graphics, drawing graphics and putting equations also had been covered in detail in the previous units.

In this unit you will be introduced to installation of Scilab in both Windows and Linux environment. In the next unit we will discuss Scilab in detail and learn about basic syntax, mathematical operators, predefined constants, how to handle .sci files and install additional packages.

8.3 SCILAB

Scilab is an open source powerful computing environment for scientific and engineering applications. Scilab is developed by the Scilab

team within ESI Group. It is a major alternative for Matlab. Scilab is available under the GPL License. This means that the user has the freedom to:

- use the software for any purpose,
- change the software to suit different needs,
- share the software with friends and
- share the changes made by the user.

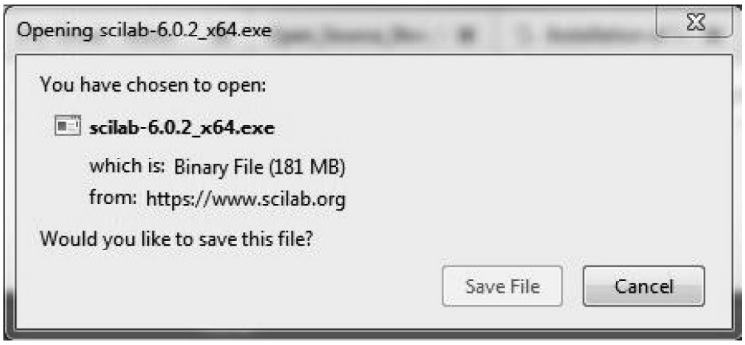
We will be now learn how to install Scilab in Windows and Linux environment.

8.3.1 Installation of Scilab in Windows

- Download Scilab from official Scilab website *https://www.scilab.org/*, under download section and download the latest installation package. Download the Windows 64 bit package if you have a 64 bits machine or download the Windows 32 bit package in case of a 32 bits machine.

  Save the *scilab-x.x.x.exe* file in a location in your system.

Figure 8.1: Opening Scilab



- Run the *scilab-x.x.x.exe* file as administrator.

Figure 8.2: Run Scilab



- Select the language for installation by using the dropdown list.

Figure 8.3: Select Language



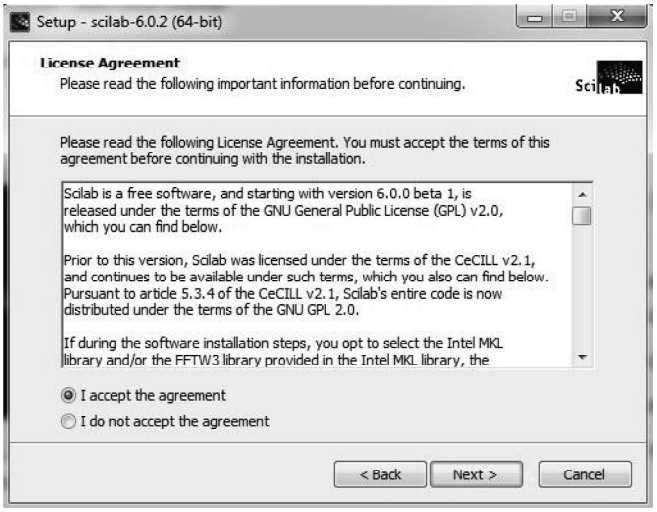
- Launch the Scilab setup wizard. Click “Next” button.

Figure 8.4: Scilab Setup



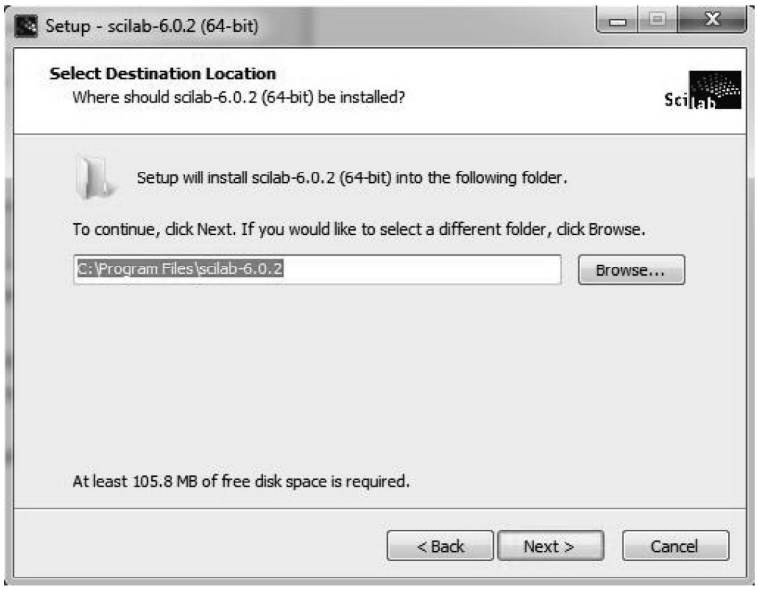
- Read the license agreement and select “I accept the agreement” then click on the “Next” button.

Figure 8.5: License Agreement



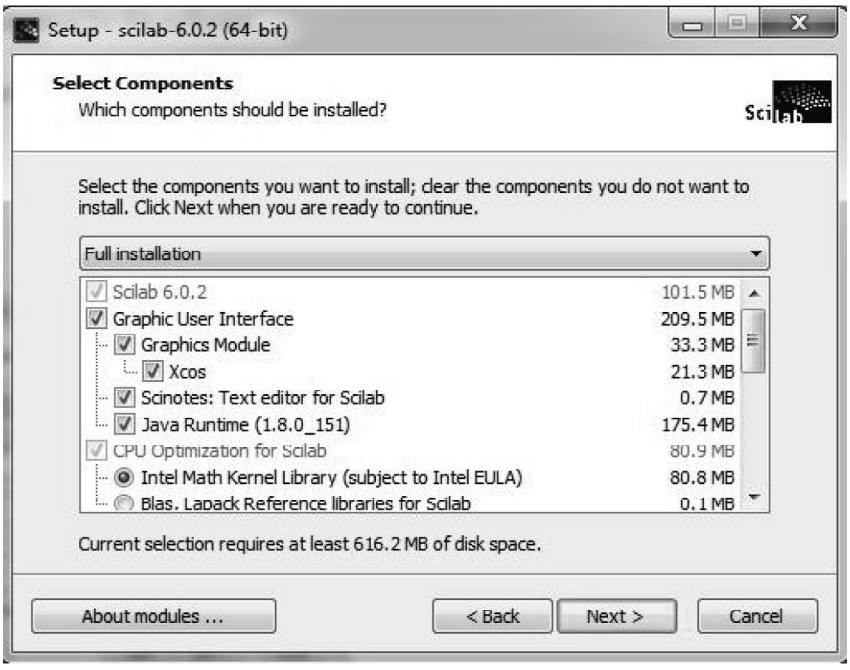
- Select a local hard drive location for the installation and make sure you have enough free disk space for Scilab installation. Click “Next” button.

Figure 8.6: Select Location



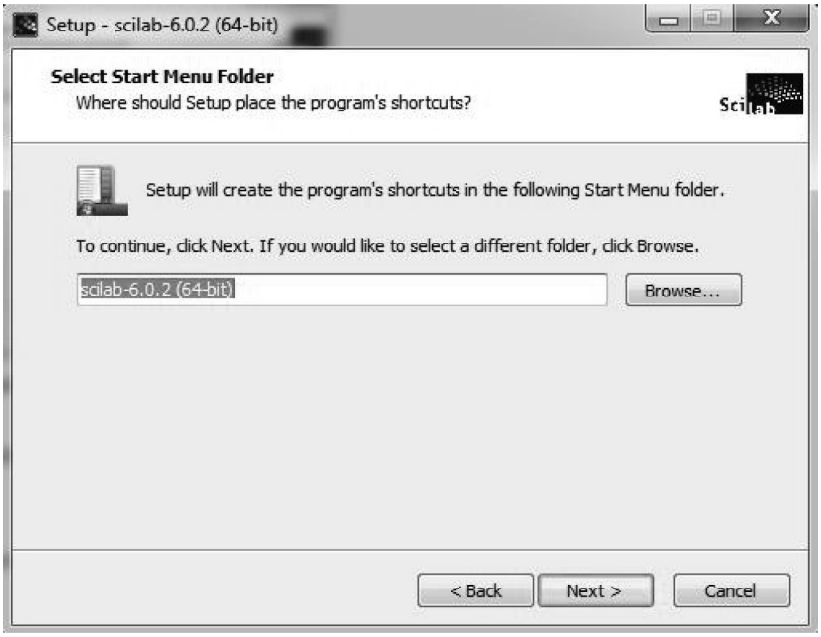
- Select the type of installation. “Full installation” will install the complete/full package of components/modules. Click “Next” button.

Figure 8.7: Type of Installation



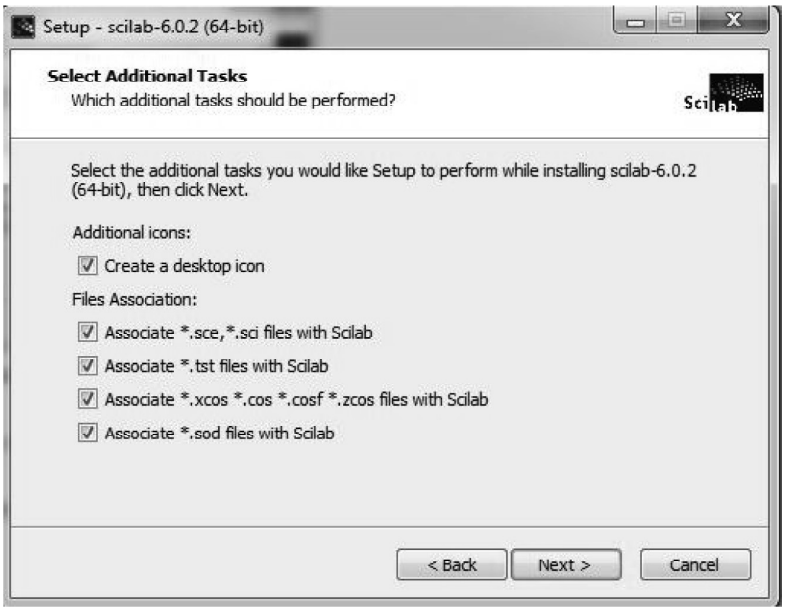
- Select start menu folder. If you want a different location than the default one, click the “Browse…” button. then click “Next” button.

Figure 8.8: Start Menu Folder



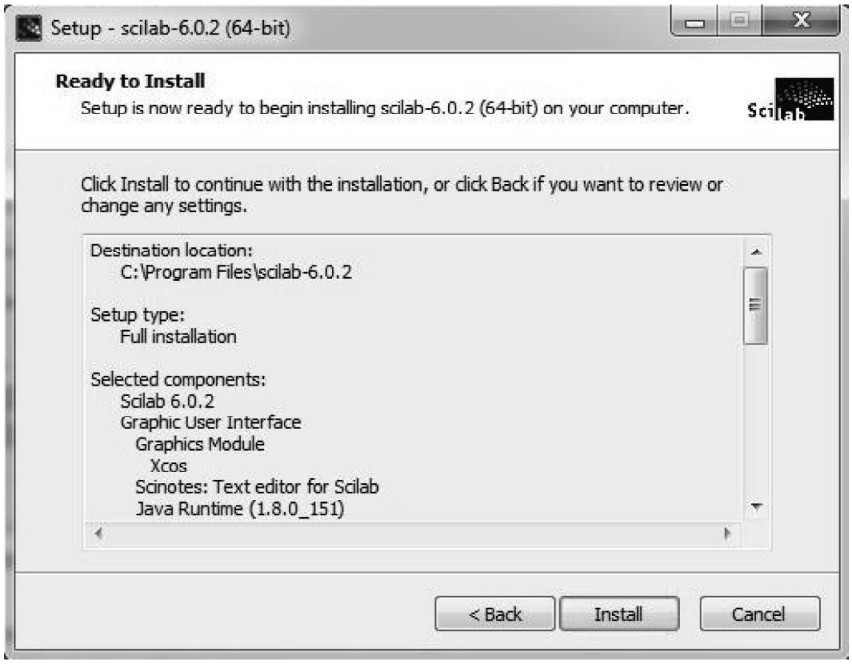
- Select additional tasks. If you want to create a Desktop icon make sure to check the box for “Create a desktop icon”. Check the three boxes for different file association. Then click “Next” button.

Figure 8.9: Additional Tasks



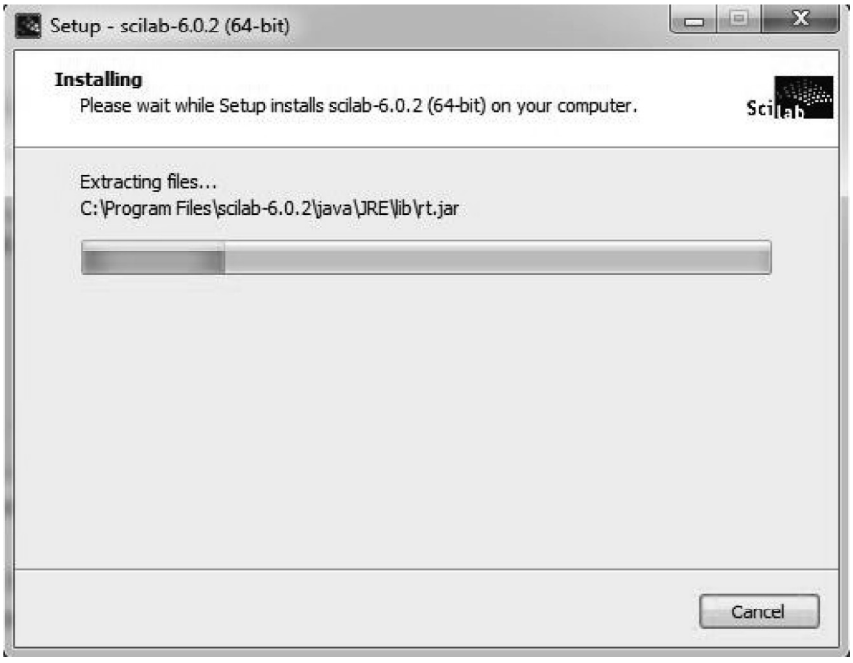
- Review the summary window with installation settings. If all the settings are as expected click the “Install” button.

Figure 8.10: Review summary



- Wait for the installation to complete until the progress bar fills up.

Figure 8.11: Installation progress bar



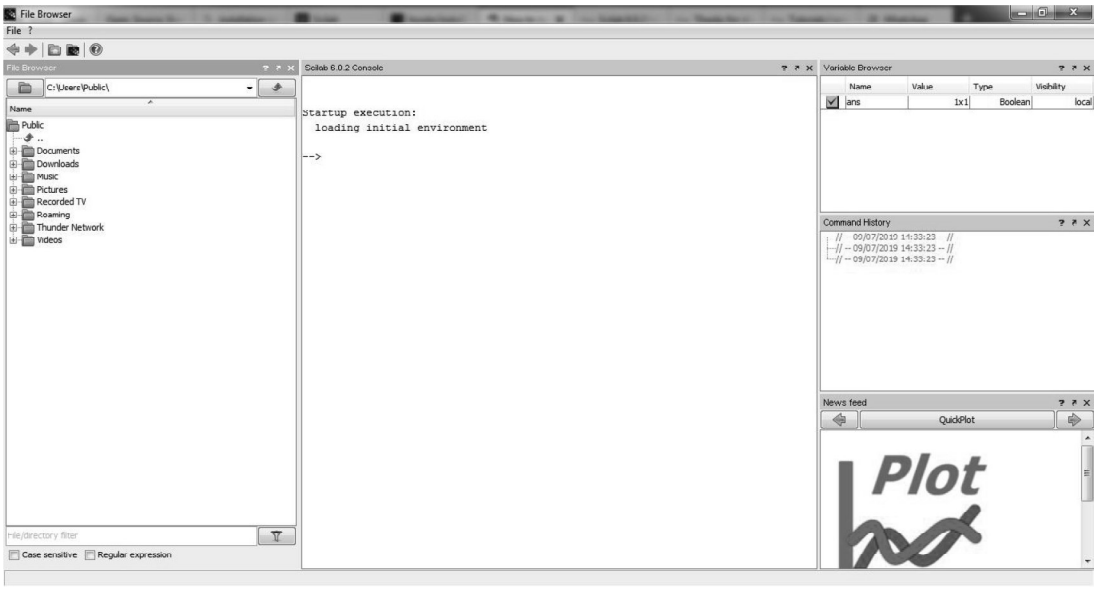
- Setup is finished. Then check the box “Launch Scilab” and click “Finish” button to launch Scilab.

Figure 8.12: Completion of Scilab Installation



- After Scilab is launched you get the start screen shown below.

Figure 8.13: Start Screen after Launch



8.3.2 Installation of Scilab in Linux

The steps for installation of Scilab in Linux are given below:

- Step 1: Get the binary file and put it in a local directory where you want to install Scilab (<scilab-path>).
- Step 2: Launch the following commands to get the binary file and install Scilab.
  - **For Linux 32 bits:**
    1) *[$SHELL] wget http://www.scilab.org/download/6.0.2/scilab-6.0.2.bin.linux-i686.tar.gz*
    2) *[$SHELL] cd <scilab-path>*
    3) *[$SHELL] tar xzvf scilab-6.0.2.bin.linux-i686.tar.gz*
  - **For Linux 64 bits:**
    1) *[$SHELL] wget http://www.scilab.org/download/6.0.2/scilab-6.0.2.bin.linux-x86_64.tar.gz*
    2) *[$SHELL] cd <scilab-path>*
    3) *[$SHELL] tar xzvf scilab-6.0.2.bin.linux-x86_64.tar.gz*
- Step 3: To launch Scilab.
  - *[$SHELL] cd <scilab-path>/scilab-5.5.2*
  - *[$SHELL] ./bin/scilab*



CHECK YOUR PROGRESS

Q 1: From where can we download Scilab Python for windows installation?

.....

.....

Q 2: What is the command to launch Scilab in Linux?

.....

.....



8.4 LET US SUM UP

- Scilab can be downloaded from the official Scilab website https://www.scilab.org/.
- For both Windows and Linux operating systems there are 64 and 32 bit packages for 64 and 32 bit machines.
- In Linux we can launch Scilab using the command "./bin/scilab".



8.5 FURTHER READING

1) Verma, A. K., (2018). SCILAB: A beginner's Approach.
2) Nagar, S., (2018). Introduction to Scilab: For Engineers and Scientists.



8.6 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No 1: https://www.scilab.org/

Ans to Q No 2: ./bin/scilab



8.7 MODEL QUESTIONS

Q 1: What is the official Scilab website?

Q 2: How can Scilab be installed in Linux?

Q 3: Give the steps to install Scilab in Windows.

*** ***** ***

UNIT 9 : INTRODUCTION TO SCILAB

UNIT STRUCTURE

9.1 Learning Objectives

9.2 Introduction

9.3 Scilab

9.4 Basic Syntax

9.5 Mathematical Operators

9.6 Predefined Constants

9.7 Handling .sci Files

9.8 Installation of Additional Packages

- 9.8.1 Accessing ATOMS GUI
- 9.8.2 Installing Packages
- 9.8.3 Removing Packages

9.9 Let Us Sum Up

9.10 Further Reading

9.11 Answers to Check Your Progress

9.12 Model Questions

9.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- define Scilab
- write basic syntax
- define mathematical operators
- define predefined constants
- handle .sci files
- install additional packages

9.2 INTRODUCTION

In the previous unit you had been introduced to installation of Scilab in both Windows and Linux environment. In this unit you will be introduced to basic syntax of Scilab. We will learn about the mathematical operators

and predefined constants in Scilab. We will also learn about handling .sci files and installing additional packages.

In the next unit you will be introduced to matrix operations and functions in Scilab. We will learn about built in functions, complex numbers, polynomials, vectors, matrix operations and handling these data structures using built in functions.

9.3 SCILAB

Scilab is a major alternative for Matlab. Similar to Matlab, Scilab uses matrix as the main data type. All the operations are performed with respect to the matrix. Scilab has many mathematical operators which we will be learning at the later part of the unit.

In the next section you will learn about the basic syntax and how to use the Scilab console.

9.4 BASIC SYNTAX

After we launch Scilab, Scilab environment consist of the following docked windows – File Browser, Console, Variable Browser, Command History and News feed.

In the console, after the prompt "—>", just typing a command and pressing the Enter key gives the corresponding result of the command. For example, if we type 2+3 we get the addition result in the default variable "**ans**" as 5.

```
—>  2+3
ans   =
      5.
```

Similarly, we can perform other mathematical operations too in the console.

```
—>  (2+10)*5
ans   =
      60.
```

Now, if we want to define a variable in Scilab, then we do the definition as given below.

```
-->  a=5
a     =
      5.
```

Here, in the above example, we have defined a variable "*a*" and assigned a value of 5 to it.



CHECK YOUR PROGRES

Q 1: What is Scilab an alternative to?

.....

.....

Q 2: What are the windows docked in a Scilab environment?

.....

.....

9.5 MATHEMATICAL OPERATORS

In Scilab, the mathematical operators available for mathematical operations are addition (+), subtraction (-), multiplication (*), division (/) and exponentiation (^).

For example:

addition (+)

```
-->     2+3
ans      =
         5.
```

subtraction (-)

```
-->     5-3
ans      =
         2.
```

multiplication (*)

```
-->     4*5
ans      =
        20.
```

division (/)

```
-->     6/4
ans      =
        1.5
```

exponentiation (^)

```
-->     2^3
ans      =
         8.
```

9.6 PREDEFINED CONSTANTS

In Scilab, there are some special variables whose values are predefined and they are known as predefined constants. These constants have values which cannot be changed. The character "%" is used to access these variables followed by the name of the variable.

Table 9.1: predefined Constants

Constant	Description/Properties	Value
%pi	Greek letter π, sin(%pi/2) = 1	3.1415927
%e	Euler's constant , log(%e) = 1	2.7182818
%eps	Greek letter ε	$2.2 \cdot 10^{-16}$
%i	Imaginary unit	square root of -1
%inf	Infinity	1/0
%nan	not a number	∞ − ∞
%t	True (boolean)	1
%f	False (boolean)	1
%s	Polynomial expression	s=poly(0,"s")
%z	Polynomial expression	z=poly(0,"z")



CHECK YOUR PROGRESS

Q 3: What are the available mathematical operators in Scilab?

.....

.....

Q 4: Define two predefined constants in Scilab.

.....

.....

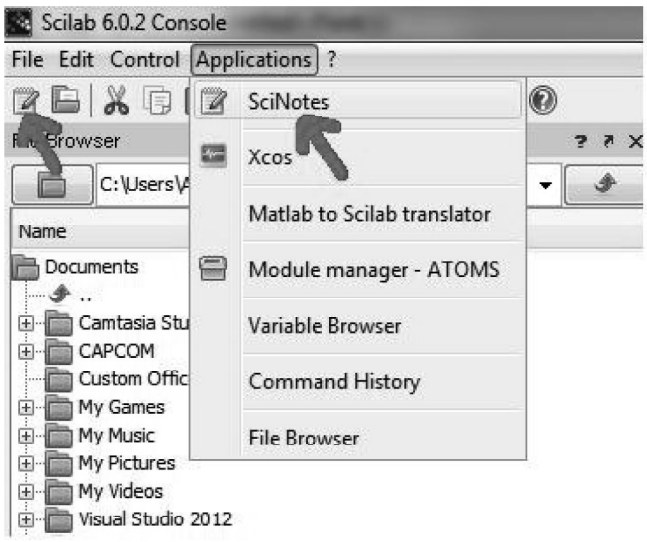
9.7 HANDLING .SCI FILES

While writing commands in the console it is not possible to save the commands and it is not easy to edit multiple lines of commands. A .sci file is a script used by Scilab, containing these commands or instructions. A .sci file basically is a function file with a function statement at the start. It can also have multiple function definitions containing multiple

Scilab commands performing operations on the function arguments.

To write multiple commands, the editor is the best tool available in Scilab. To open an editor in Scilab, just click the first icon in the toolbar. We can also open an editor from the Menubar, under “Applications” and select “SciNotes”.

Figure 9.1: Opening Scilab editor



The editor opens with a default file name “Untitled 1”. To save the file, we select from the Menubar the “File” option and then we select “Save as” option. We can now give a suitable name for the file and save the file with “.sci” extension.

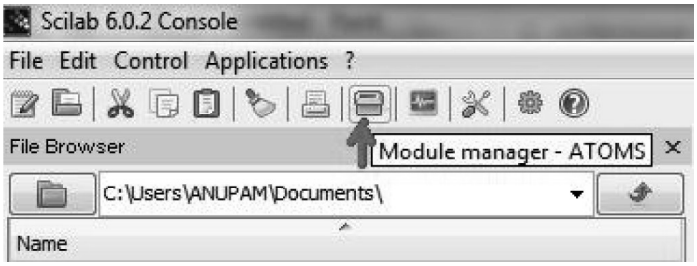
9.8 INSTALLATION OF ADDITIONAL PACKAGES

In Scilab we can add many additional packages or modules called as “Toolboxes” which have been created by the Scilab community. ATOMS (AuTomatic mOdules Management for Scilab) is the repository for these packaged extension modules. Scilab includes a GUI to facilitate the use of ATOMS.

9.8.1 Accessing ATOMS GUI

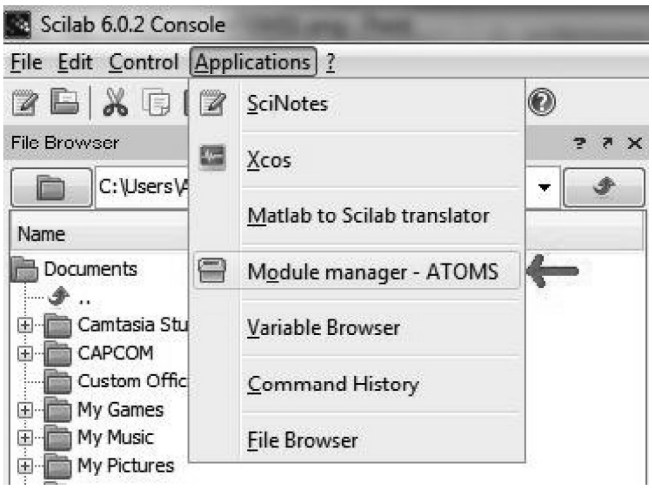
The ATOMS GUI is available through the Scilab using the icon shown below in figure 9.2.

Figure 9.2: ATOMS GUI



This GUI is also available through the Scilab menu "Applications".

Figure 9.3: Module manager ATOMS

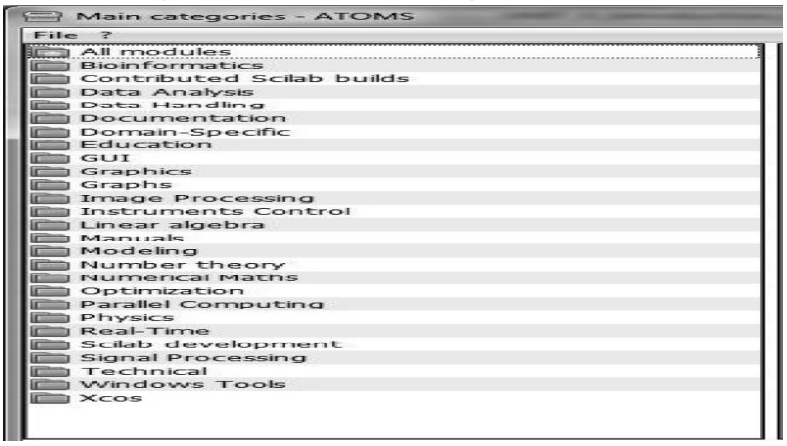


The GUI can also be accessed by entering the command "*atomsGui()*" in the console.

—> *atomsGui();*

The GUI gives the list of available packages for installation.

Figure 9.4: List of packages in ATOMS



The list of available packages can also be displayed in console by entering the command "*atomsList();*" in the console.

—> *atomsList();*

9.8.2 Installing Packages

The packages can be installed using the GUI as shown below in figure 9.5. In this figure, we are installing package "Diffcode" available under Optimization packages. This is a Scilab code for computing a variable y depending on a variable x and a direction dx that allows evaluation of y together with the directional derivative Grad(y)*dx. For installing packages, we need to select the package to be installed and click "Install".

Figure 9.5: Installing Packages "diffcode"



The same package can be installed by entering the command "*atomsInstall('<package_name>')*" in the console.

```
—> atomsInstall ('Diffcode')
```

9.8.3 Removing Packages

For removing a package in GUI we need to select the

package to be removed and just click “Remove”. A package can also be removed by entering the command “*atomsRemove <package_name>*” in the console.

```
—>atomsRemove  Diffcode
```



CHECK YOUR PROGRESS

Q 5: What is a .sci file?

.....

.....

Q 6: How to install a package in Scilab?

.....

.....



9.9 LET US SUM UP

- Scilab is similar to Matlab and uses matrix as the main data type.
- In the Scilab console, after the prompt “—>”, just typing a command and pressing the Enter key gives the corresponding result of the command.
- In Scilab, the mathematical operators available for mathematical operations are addition (+), subtraction (-), multiplication (*), division (/) and exponentiation (^).
- In Scilab, there are some special variables whose values are predefined and they are known as predefined constants.
- The character “%” is used to access these variables followed by the name of the variable.
- A .sci file is a script used by Scilab, containing multiple commands or instructions.
- To write multiple commands, the editor is the best tool available in Scilab.
- In Scilab we can add many additional packages or modules called as “Toolboxes” which have been created by the Scilab community.

- ATOMS (AuTomatic mOdules Management for Scilab) is the repository for these packaged extension modules.



9.10 FURTHER READING

1) Verma, A. K., (2018). SCILAB: A beginner's Approach.
2) Nagar, S., (2018). Introduction to Scilab: For Engineers and Scientists.



9.11 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No 1: Scilab is an alternative to Matlab.

Ans to Q No 2: File Browser, Console, Variable Browser, Command History and News feed.

Ans to Q No 3: addition (+), subtraction (-), multiplication (*), division (/) and exponentiation (^).

Ans to Q No 4: %pi (Greek letter π), %e (Euler's constant)

Ans to Q No 5: A .sci file is a script used by Scilab, containing mutiple lines of Scilab commands or instructions.

Ans to Q No 6: The packages can be installed using the ATOMS GUI or by entering the command "atomsInstall('<package_name>')" in the console.



9.12 MODEL QUESTIONS

Q 1: What is Scilab?

Q 2: What is the main data type in Scilab?

Q 3: How to use a Scilab console to execute instructions in Scilab?

Q 4: How to declare a variable in Scilab?

Q 5: What are the mathematical operators in Scilab?

Q 6: What are constants in Scilab?

Q 7: Give 5 examples of constants in Scilab.

Q 8: What is the extension of a Scilab file?

Q 9: How to create Scilab files?

Q 10: What are packages in Scilab?

Q 11: How to install packages in Scilab?

Q 12: How to remove packages in Scilab?

*** ***** ***

UNIT 10 : MATRIX AND BUILT-IN FUNCTIONS

UNIT STRUCTURE

10.1 Learning Objectives

10.2 Introduction

10.3 Matrix

10.4 Built-in Functions

- 10.4.1 Length()
- 10.4.2 Size()
- 10.4.3 Zeros()
- 10.4.4 Ones()
- 10.4.5 Eye()
- 10.4.6 Rand()
- 10.4.7 Inv()
- 10.4.8 Spec()

10.5 Complex Numbers

10.6 Polynomials

10.7 Let Us Sum Up

10.8 Further Reading

10.9 Answers to Check Your Progress

10.10 Model Questions

10.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- represent matrix and vectors
- use the different build in functions available in Scilab
- represent complex numbers in Scilab
- represent polynomials in Scilab

10.2 INTRODUCTION

In the previous unit you have been introduced to basic syntax of Scilab. We also know how to use the Scilab console. In this unit you will

learn about matrix operations and functions in Scilab. You have also learn in detail about different built in functions that can be used on matrixes. You will also learn how to represent complex numbers and polynomials in Scilab. In the next unit, you will learn about the different conditional statements and loops that can be used in Scilab. You will learn about conditional statements like if else, ifelseif, select and looping statements like for loop, while loop etc.

10.3 MATRIX

You are already familiar with the Scilab console and know how to perform basic mathematical operations. Scilab is very powerful at computations with variables such as vectors and matrices. It has a lot of built-in functions that allows the user to perform complex computations and manipulations on vectors (1-D matrices) and matrices.

Let us look at how the matrixes are defined in Scilab. In the example below, we have defined a matrix variable “p” which contains three values.

```
—> a = [1 2 3];
```

Matrixes can also be defined by using comma’s between the values. For example,

```
—> b = [4, 5, 6];
```

Two dimensional matrixes can also be defined in Scilab. For example,

```
—> c = [ 1 2 3; 4 5 6; 7 8 9];
```

If we now output the values of the variable “c” it will be represented in Scilab as the following.

```
—> c
 c  =
     1.    2.    3.
     4.    5.    6.
     7.    8.    9.
```

We can perform different mathematical functions on two or more matrixes. For example, we perform addition of two matrixes “c” and “d”

and put the sum in a third matrix “add” below.

```
-> c = [ 1 2 3; 4 5 6; 7 8 9];
-> d = [ 1 2 3; 1 2 3; 1 2 3];
-> add= c + d
 add   =
    2.    4.    6.
    5.    7.    9.
    8.    10.   12.
```

We can also find the transpose of any vector or matrix by using apostrophe. For example, the transpose of matrix “add” is stored in matrix “add_t” below.

```
-> add_t = add'
 add_t   =
    2.    5.    8.
    4.    7.    10.
    6.    9.    12.
```

10.4 BUILT-IN FUNCTIONS

Scilab has a large range of build-in functions that can be used on matrixes. Let us look at some of the most commonly used such build-in functions.

10.4.1 Length()

We can find the length of any vector in Scilab by using the built-in function **length**(). For example, the vector “a” defined below has three entries, and hence the length of the vector “a” is returned as 3.

```
-> a = [1 2 3];
-> length(a)
 ans   =
    3.
```

10.4.2 Size()

We can find the size of any matrix in Scilab by using the built-in function **size()**. For example, the matrix "x" defined below has two columns and three rows, and hence the size of the marix "x" is returned as (3,2). The first result of size() represents the number of rows and the second result represents the number of columns.

```
—> x = [ 1 2; 3 4; 5 6]
 x  =
   1.   2.
   3.   4.
   5.   6.
—> size(x)
 ans  =
   3.   2.
```

We can create some special matrixes in Scilab by using certain build-in functions. Let us look at some of these functions.

10.4.3 Zeros()

We can create a special matrix consisting of only zeros by using the function **zeros()**. For example, we can create a matrix of 3 rows and 4 columns consisting only of zero values by simply using the built in function zeros(3,4). For functions like these, the first entry always represents the number of rows and the second entry represents the number of columns.

```
—> zeros(3,4)
 ans   =
    0.    0.    0.    0.
    0.    0.    0.    0.
    0.    0.    0.    0.
```

10.4.4 Ones()

We can create another special matrix consisting of only

ones by using the function **ones()**. Similar to the zeros() function, if we want to create a matrix of 2 rows and 3 columns consisting of entries with values only “1” then we can do this by simply using the built in function ones(2,3). For this function also, the first entry always represents the number of rows and the second entry represents the number of columns.

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

10.4.5 Eye()

We can create identity matrixes by using the function **eye()**. Similar to the earlier two functions, if we want to create an identity matrix by simply specifying the number of rows and columns using the built in function **eye()**. For example, if we want to create an identity function for a 3*3 square matrix then we would simply type eye(3,3) in the Scilab console.

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

10.4.6 Rand()

We have now learned that we can create matrixes with values of only zeros or ones or simply identity matrixes. Apart from these, we can also create matrixes with random values. For creating random value matrixes we simply need to use the build-in function **rand()**. For example, if we want to create a random matrix of 2 rows n 5 column then we can simply write the function rand(2,5).

```
—> rand(2,5)
```

```
 ans    =
0.2113249  0.0002211  0.6653811 0.8497452 0.8782165
0.7560439  0.3303271  0.6283918 0.685731  0.068374
```

10.4.7 Inv()

The **inv()** function is used to find the inverse of a square matrix. A warning message is printed if the matrix is badly scaled or nearly singular. For example, we have found the inverse of a matrix below.

```
-> A=rand(3,3)
 A   =
    0.4818509     0.2806498     0.211903
    0.2639556     0.1280058     0.1121355
    0.4148104     0.7783129     0.6856896
—> inv(A)*A
 ans   =
       1.            0.          0.
       -1.305D-15     1.         0.
       0.            2.220D-16   1.
```

10.4.8 Spec()

The **spec()** function is used to find the Eigen values of a matrix. We can find the Eigen values of a matrix like A in the example below.

```
—> A=[2,1,1;1,3,2;3,1,4]
 A   =
    2.     1.     1.
    1.     3.     2.
    3.     1.     4.
—> spec(A)
 ans   =
    6.095824
    1.452088  +  0.4336988i
```



CHECK YOUR PROGRESS

Q 1: How are matrixes defined in Scilab?

.....

.....

Q 2: Give the purpose of using rand() function.

.....

.....

Q 3: How can we find the transpose of a matrix?

```
1.452088  -  0.4336988i
```

.....

.....

10.5 COMPLEX NUMBERS

Complex numbers are numbers which contain a pair of real numbers with a imaginary part. Complex numbers can be represented as follows:

c = a + b*i

where, c is the complex number

a is the real number

b is the real number

i is the imaginary part

If the real number (a) is zero, the complex number is purely imaginary. Also, if the real number (b) is zero, the complex number becomes a real number. In Scilab, we define the complex numbers by using the special constant **%i**. For example, a complex number c which has real part 4 and imaginary part 5i can be represented as follows.

```
-> c = 4 + 5*%i
c  =
   4. + 5.i
```

10.6 POLYNOMIALS

Polynomials can be defined as an mathematical expression of

determined length which contains operations between variables and constants (coefficients). Generally, polynomials can be expressed as follows:

$$\mathbf{p(x) = a_0x^0 + a_1x^1 + \ldots + a_nx^n}$$

where, a_n are real numbers ($a_n \in R$) that represent the coefficients of the polynomial

x is a symbolic variable of the polynomial

n is a natural number ($a_n \in N$) that represents the exponents of the polynomial.

In Scilab, polynomials can be defined using the built-in function **poly().** The syntax of the Scilab poly() function is:

```
p = poly(data, 'var', 'options')
```

where,

data is a vector or real number that represents the coefficients or the roots of the polynomial,

'var' is a string that represents the symbolic variable name of the polynomial,

'options' are string that represents the type of the polynomial definition; the possible values are predefined as: 'roots' (short 'r'), default value, for the definition of the polynomial based on its roots or 'coeff' (short 'c'), for the definition of the polynomial based on its coefficients

p is the variable defined as a polynomial.

Now, if we want to represent a polynomial which has roots: $x_1 = 2$ and $x_2 = -2$, then it can be represented by the following example.

```
-> p=poly([2, -2],'x','r')
 p   =
          2
   -4  +x
```

The result of the above polynomial definition is: $-4 + x^2$

Let us try to define the polynomial: $4 + 5x - 6x^2 + 7x^3$. Below, we have represented the polynomial using the coefficients:

$a_0 = 4, a_1 = 5, a_2 = -6$ and $a_3 = 7$.

```
-> p=poly([4  5  -6  7],'x','c')
```

```
p    =
                    2       3
    4  +5x  -6x    +7x
```

The roots of a polynomial can be found by using the function roots(). For example, let us try to find the roots of the polynomial p, where p is represented as $p = -4 + x^2$. For this, we can simply type the following in the Scilab console.

```
—> poly1 = poly([-4  0  1],'x','c')
 poly1   =
          2
   -4  +x
—> roots(poly1)
 ans   =
    2.
```



CHECK YOUR PROGRESS

Q 4: How are complex numbers represented in Scilab?

.....

.....

Q 5: Give the syntax of the poly() function.

.....

.....



10.7 LET US SUM UP

- Scilab is very powerful at computations with variables such as vectors and matrices.
- Matrixes can also be defined by using commas between the values.
- We can afind the transpose of any vector or matrix by using apostrophe.
- The length of any vector in Scilab is found by using the built-in

function length().

- We can create a special matrix consisting of only zeros by using the function zeros().
- Identity matrixes n be created by using the function eye().
- For creating random value matrixes we simply need to use the build-in function rand().
- The inv() function is used to find the inverse of a square matrix.
- The spec() function is used to find the Eign values of a matrix.
- In Scilab we define the complex numbers by using the special constant **%i**.
- Polynomials can be defined as an mathematical expression of determined length which contains operations between variables and constants (coefficients).



10.8 FURTHER READING

1) Verma, A. K., (2018). *SCILAB: A beginner's Approach*.

2) Nagar, S., (2018). *Introduction to Scilab: For Engineers and Scientists*.



10.9 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No 1: A matrix variable "p" which contains three values can be defined as follows.

—> a = [1 2 3];

Ans to Q No 2: The rand() function is used for creating random value matrixes.

Ans to Q No 3: We can also find the transpose of any vector or matrix by using apostrophe symbol after the matrix.

Ans to Q No 4: In Scilab, complex numbers are represented by using the special constant **%i**. For example, a complex number c which

has real part 3 and imaginary part 6i can be represented as c = 3 + 6*%i.

Ans to Q No 5: The syntax of the Scilab poly() function is: p = poly(data, 'var', 'options')

where,

data is a vector or real number that represents the coefficients or the roots of the polynomial,

'var' is a string that represents the symbolic variable name of the polynomial,

'options' are string that represents the type of the polynomial definition; the possible values are predefined as: 'roots' (short 'r'), default value, for the definition of the polynomial based on its roots or 'coeff' (short 'c'), for the definition of the polynomial based on its coefficients and

p is the variable defined as a polynomial.



10.10 MODEL QUESTIONS

Q 1: Describe how vectors and matrixes are represented in Scilab.

Q 2: Describe in brief any five built-in functions related to matrixes.

Q 3: Give an example using inv() function.

Q 4: Give an example using spec() function.

Q 5: How are rand(), ones() and zeros() functions useful in real life applications?

Q 6: How are complex numbers represented in Scilab?

Q 7: Give an example for finding roots of a polynomial.

Q 8: Show addition of any two polynomials in Scilab.

Q 9: How are polynomials represented in Scilab?

Q 10: Represent the polynomial, $p = 4x^4 - 3x^3 + 2x^2 - x + 15$ in Scilab.

*** ***** ***

UNIT 11 : CONDITIONAL STATEMENTS IN SCILAB

UNIT STRUCTURE

11.1 Learning Objectives

11.2 Introduction

11.3 Conditional Statements

- 11.3.1 if statement
- 11.3.2 if else statement
- 11.3.3 if elseif statement
- 11.3.4 select statement
- 11.3.5 break statement
- 11.3.6 continue statement

11.4 Loops

- 11.4.1 for loop
- 11.4.2 while loop

11.5 Let Us Sum Up

11.6 Further Reading

11.7 Answers to Check Your Progress

11.8 Model Questions

11.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- use if, if-else and if-elseif statements in scilab programs
- use break and continue in scilab programs
- use select statements in scilab programs
- use while loops in scilab programs
- use for loops in scilab programs

11.2 INTRODUCTION

In the previous units, you had been introduced to basic syntax of Scilab. You also learnt about matrix operations and functions in Scilab.

You had learned about built in functions, complex numbers, polynomials, vectors, matrix operations.

In this unit, you will learn about the different conditional statements and loops that can be used in Scilab. You will learn about conditional statements like if else, ifelseif, select etc. Apart from these, you will also learn in detail how to use loop statements like for loop and while loop in Scilab. In the next unit you will be introduced to graphics in Scilab, handling 2D and 3D graphics, generating .jpg files, plotting functions, data plotting, creating GUI in Scilab and plotting 2D graphs.

11.3 CONDITIONAL STATEMENTS

You are already familiar with the Scilab console and know how to perform basic mathematical operations and matrix operations as well. Generally programs and instructions in all languages like C, Java are executed sequentially. This is true for Scilab also. However, this happens when no conditions are put around the instructions or statements. Sometimes we may want to put certain conditions or restrictions on the flow of execution. This can be done with the help of decision making or conditional statements.

Let us look at the different conditional statements that can be used in Scilab. Some of the conditional statements are listed below.

- IF ELSE
- IF ELSEIF
- SELECT CASE
- break
- continue

We will look at each of these conditional statements in detail now.

11.3.1 IF conditional statement

The IF conditional statement is used in Scilab when we need to put certain conditions around instructions. It is one of the simplest conditional statements in Scilab. The general structure

for IF statements is given below.

General structure:

```
if condition_1 then
      instructions_1
end
```

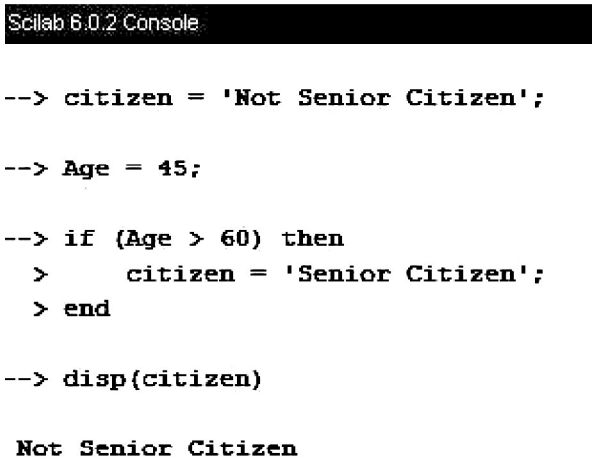
In the above general structure given for IF, the statement is started by using the keyword **if** followed by the condition. Once the condition is stated it is followed by the keyword **then.** The instructions which are to be followed when the condition is true are to be specified between the keyword **if** and keyword **end**. The **end** keyword specifies the ending of the conditional statements. An example of **if** statement bold shown in example 11.1.

Example 11.1: Example showing use of If statement

```
citizen = 'Not Senior Citizen';
Age = 45;
if (Age > 60) then
      citizen = 'Senior Citizen';
end
disp(citizen)
```

In the above example, we have first defined a variable "citizen" with value "`Not Senior Citizen`" and a variable "age" with value "45". In the following statements the variable age is compared with the value 60 and if the condition is true the next line will be executed where the value of variable citizen will be modified to "`Senior Citizen`". In this example, since the value in variable **"age"** is not greater than **"60"** so the **if** statement will not be executed and the value in the variable "citizen" will remain "`Not Senior Citizen`". The output is shown in the figure 11.1.

Figure 11.1: Output of example 11.1



11.3.2 IF ELSE conditional statements

IF ELSE is one of the most commonly used conditional statements in Scilab. The general structure for IF ELSE statements is given below.

General structure:

```
if condition_1 then
 instructions_1
else
 default instructions
end
```

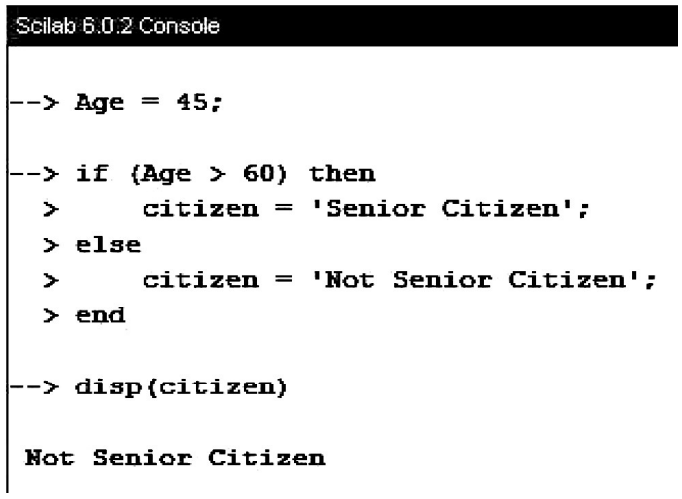
In the above general structure given for IF ELSE, the statement is started by using the keyword **if** command followed by the condition. Once the condition is stated it is followed by the keyword **then.** This is followed by the keyword **else**. The series of instructions that are to be followed when this condition is true are to be specified between the keyword **then** and keyword **else**. The instructions which are to be followed when the condition is not true are to be specified between the keyword **else** and keyword **end**. The **end** keyword specifies the ending of the conditional statements. An example for if else statement is given in example 11.1.

Example 11.2: Example showing use of If Else statement

```
Age = 45;
if (Age > 60) then
    citizen = 'Senior Citizen';
else
    citizen = 'Not Senior Citizen';
end
disp(citizen)
```

In the above example, we have first defined a variable "age" with value "45". In the following statements the variable age is compared with the value 60 and if the condition is true the next line will be executed. Otherwise, the statements after the else keyword will be executed. In this example the value that will be stored in the variable "citizen" will be "`Not Senior Citizen`". The output is shown in the figure 11.2.

Figure 11.2: Output of example 11.2



11.3.3 IF ELSEIF conditional statements

The general structure for IF ELSEIF statements is given below.

General structure:

```
if condition_1 then
     instructions_1
elseif condition_2 then
```

```
      instructions_2
……….
elseif  condition_N  then
      instructions_N
else
      default  instructions
end
```

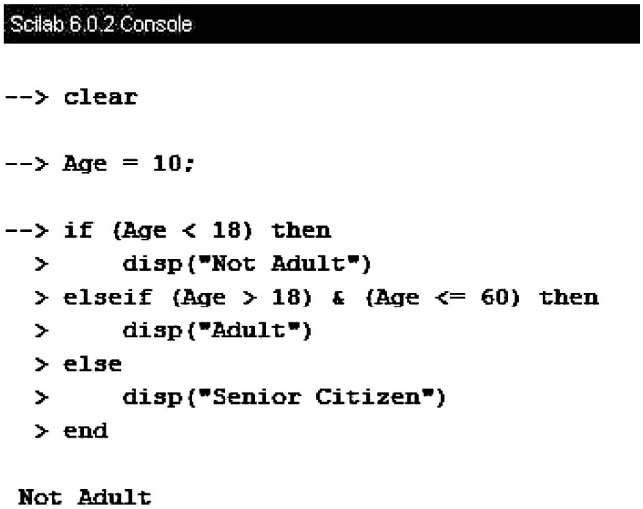
In the above general structure given for IF ELSEIF, the statement is started by using the keyword **if** command followed by the condition. Multiple conditions can be stated using **elseif** statements. The instructions which are to be executed when the particular condition is true are specified between the keyword **then** and the next **elseif** or **else** keyword. The **end** keyword specifies the ending of the elseif conditional statements. An example for elseif statement is given in example 11.3.

Example 11.3: Example showing use of If Elseif statement

```
Age  =  10;
if  (Age  <  18)  then
      disp(“Not  Adult”)
elseif  (Age  >  18)  &  (Age  <=  60)  then
      disp(“Adult”)
else
      disp(“Senior  Citizen”)
end
```

In the above example, we have first defined a variable “age” with value “10”. In the following statements the variable age is first compared with the value 18. Since this condition is true so the variable will not be compared anymore. The output is shown in the figure 11.3.

Figure 11.3: Output of example 11.3



11.3.4 SELECT CASE conditional statements

The general structure for SELECT CASE statements is given below.

General Structure

```
select controlVariable
case value_1 then instructions_1
case value_2 then instructions_2
…
case value_N then instruction_N
else defaultInstructions
end
```

The current value of the control variable (controlVariable) is evaluated and compared with each of the cases starting with the first one (value_1). When a match case is found, the corresponding instruction is executed.

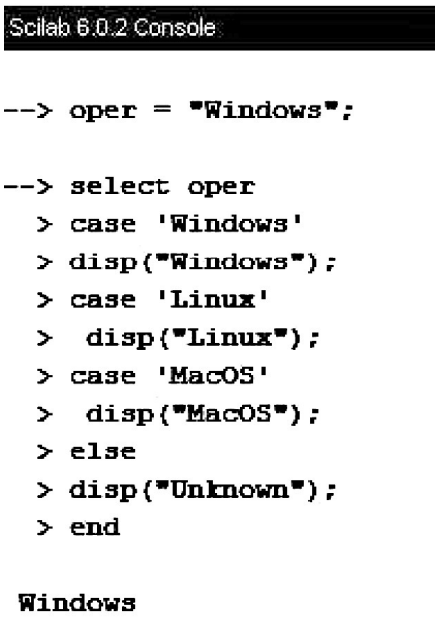
Example 11.4: Example showing use of Select case statement

```
oper = “Windows”;
select oper
case ‘Windows’
disp(“Windows”);
case ‘Linux’
```

```
  disp(“Linux”);
case  ‘MacOS’
disp(“MacOS”);
else
disp(“Unknown”);
end
```

In the above example, we have first defined a variable “oper” with value “Windows”. In the following statements the variable “oper” is compared with different values using different case statements. Since the value in the first case condition matches with the variable “oper”, so the set of instructions for that case are executed. The output is shown in the figure 11.4.

Figure 11.4: Output of select case example 11.4



11.3.5 Break Statement

The general structure for **break** is given below.

General Structure

break

Let us look at an example to understand the use of break statement. Inside a **for** or **while** loop, the command bold break forces the end of the loop. This is shown in example 11.5.

Example 11.5: Example showing use of break statement

```
k=0;
while 1==1,
k=k+1;
if k > 100 then
break
end;
end
```

In the above example, we have first defined a variable “k” with value “0”. In the following statements the variable “k” is incremented and then compared if it is greater than 100. The loop stops only when the value of k gets bigger than 100 and executes the break statement.

11.3.6 Continue Statement

The general structure for **continue** is given below.

General Structure

Let us look at an example to understand the use of continue statement. In example 11.6, inside a for or while loop, the command break forces the end of the loop. Inside a for or while loop, the command **continue** passes control to the next iteration of the loop in which it appears, skipping any remaining statements between this instruction and the loop’s end instruction. This is shown in example 11.6.

Example 11.6: Example showing use of continue statement

```
for k=1:10
if k>2 & k<=8 then
continue
disp('hello')
end
k
end
```

In the above example, when the values of the variable “k” has values greater than 2 and lesser than 8 then the if statement will be true and since there is a continue statement inside the if condition so, the control will then ignore the execution sequence and control will jump to the next iteration of the for loop. So, the values of k will be output for 1, 2, 9 and 10 only.

Figure 11.5: Output of example 11.6

```
Scilab 6.0.2 Console

-1-> for k=1:10
  > if k>2 & k<=8 then
  > continue
  > disp('hello')
  > end
  > k
  > end
 k  =

   1.

 k  =

   2.

 k  =

   9.

 k  =

   10.
```



CHECK YOUR PROGRESS

Q 1: What are conditional statements?

.....

.....

Q 2: Give the syntax of if else statements.

.....

.....

11.4 LOOP STRUCTURES

Conditional statements are used to execute specific instructions when certain conditions are true. Similarly, if we want to execute a particular instruction for a given number of times then we can use loops or looping statements. The looping statements evaluate an instruction repeatedly until a condition is satisfied. Conditional loop is to execute a set of instructions repeatedly until a condition is satisfied.

Let us look at the different loop statements that can be used in Scilab. Some of them are listed below.

- For Loop
- While Loop

11.4.1 For Loop

The general structure of **for loop** is given below.

General Structure

```
for  expression
     instructions
end
```

The expression contains a set of Scilab operations which perform:

- variable initialisation
- variable incrementation each time the instruction is executed and
- variable comparison (condition)

The instruction is the piece of code that is evaluated each time the control variable is incremented as long as the condition within the expression is satisfied.

Example 11.7: Example showing use of for loop structure

```
for  x=1:1:5
disp(x);
end
```

In the above example, in the first line, the variable x is first initialized to one and then the variable is to incremented by one till it reaches the value of five. Till the condition is true i.e. till the variable reaches the value of 5 the instructions within the loop are executed. So value of 1 2 3 4 5 will be shown as output. Once the value of the variable x is incremented to 6, the condition for the loop to execute is false and hence the iteration stops.

11.4.2 While Loop

` The general structure for **while loop** is given below.

General Structure

```
while  condition
      instructions
end
```

In while loop, the condition is usually a comparison of a variable to a constant. The instructions are a set of Scilab operations that are executed as long as the condition is true.

Example 11.8: Example showing use of while loop structure

```
x  =  1;
while  x<=5  do
disp(x);
x  =  x  +  1;
end
```

In the above example, in the first line, the variable x is first initialized to one. In the second line the condition is given to check whether the value of the variable x is less than or equal to 5. If the condition is true, the set of instructions that are given inside the while loop are executed sequentially. First the value of variable x is shown as output and then the value of the variable is incremented by one till the condition remains true. Once the value of the variable x is incremented to 6, the condition for the while loop is false and hence the iteration stops.



CHECK YOUR PROGRESS

Q 3: What are loop statements?

.....

.....

Q 4: Give the syntax of for loop.

.....

.....



11.5 LET US SUM UP

- Conditional statements are statements with the help of which we can put certain conditions or restrictions on the flow of execution.
- In select case, the current value of the control variable (controlVariable) is evaluated and compared with each of the cases starting with the first one (value_1). When a match case is found, the corresponding instruction is executed.
- The looping statements are statements that evaluate an instruction repeatedly until a condition is satisfied.
- The instruction is the piece of code that is evaluated each time the control variable is incremented as long as the condition within the expression is satisfied.
- In while loop, the condition is usually a comparison of a variable to a constant. The instructions are a set of Scilab operations that are executed as long as the condition is true.



11.6 FURTHER READING

1) Verma, A. K., (2018). *SCILAB: A beginner's Approach*.

2) Nagar, S., (2018). Introduction to Scilab: For Engineers and Scientists.



11.7 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No 1: Conditional statements are statements with the help of which we can put certain conditions or restrictions on the flow of execution.

Ans to Q No 2: The syntax for if else is given below:

```
if condition_1 then
        instructions_1
else
        default instructions
end
```

Ans to Q No 3: The looping statements are statements that evaluate an instruction repeatedly until a condition is satisfied.

Ans to Q No 4: The syntax of for loop is given below:

```
for expression
        instructions
end
```

The expression contains a set of Scilab operations which perform:

- variable initialisation
- variable incrementation each time the instruction is executed and
- variable comparison (condition)



11.8 MODEL QUESTIONS

Q 1: Describe any two conditional statements with syntax and example.

Q 2: Give an example using if-elseif statements.

Q 3: Give an example using if-else statements.

Q 4: Give an example using select statements.

Q 5: What is the difference between continue and break? Explain using example.

Q 6: Define loop statements. Why do we need loop statements?

Q 7: Give an example using while loop statements.

Q 8: Give an example using for loop statements.

Q 9: Give an example using continue statements.

Q 10: Explain the need for conditional statements.

*** ***** ***

UNIT 12 : GRAPHICS IN SCILAB

UNIT STRUCTURE

12.1 Learning Objectives

12.2 Introduction

12.3 Graphics handling

12.3.1 2D Graphics

12.3.2 3D Graphics

12.4 Generating .jpg files

12.5 Function Plotting

12.6 Data Plotting

12.7 GUI in Scillab

12.8 Plotting 2D Graphs

12.9 Let Us Sum Up

12.10 Further Reading

12.11 Answers to Check Your Progress

12.12 Model Questions

12.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- handle 2D and 3D graphics
- generate .jpg files
- plot functions
- plot data
- create GUI in Scilab
- plot 2D graphs

12.2 INTRODUCTION

In the previous unit you had been introduced to conditional statements and loops in Scilab. In this unit you will be introduced to Graphics in Scilab. We will learn about handling 2D and 3D graphics, generating .jpg files, plotting functions, data plotting, creating GUI in

Scilab and plotting 2D graphs.

In the next unit you will be introduced to applications of Scilab. You will learn about numerical linear algebra by solving linear equations, eigen values, etc. We will also look into solving ordinary differential equations, numerical analysis using iterative methods and comparison of Scilab with C/C++/Matlab.

12.3 GRAPHICS HANDLING

In Scilab, Graphics is used to handle and produce many types of graphs. Scilab native functions are used to plot these graphs. Scilab offers many ways to create and customize various types of 2D and 3D plots.

12.3.1 2D Graphics

2D Graphics are handled using 2D plot functions available in Scilab. Some of these functions are shown in table 12.1

Table 12.1: Some 2D Graphics Functions

Function	Description
LineSpec	to quickly customize the lines appearance in a plot
Matplot	2D plot of a matrix using colors
Matplot1	2D plot of a matrix using colors
Matplot properties	Description of the Matplot entities properties
Sfgrayplot	smooth 2D plot of a surface defined by a function using colors
Sgrayplot	smooth 2D plot of a surface using colors
champ	2D vector field plot
champ1	2D vector field plot with colored arrows
champ properties	description of the 2D vector field entity properties
comet	2D comet animated plot.
contour2d	level curves of a surface on a 2D plot

contour2di	compute level curves of a surface on a 2D plot
contour2dm	compute level curves of a surface defined with a mesh
contour	filled level curves of a surface on a 2D plot
errbar	add vertical error bars on a 2D plot
fchamp	direction field of a 2D first order ODE
fec	pseudo-color plot of a function defined on a mesh
fec properties	description of the fec entities properties
fgrayplot	2D plot of a surface defined by a function using colors
fplot2d	2D plot of a curve defined by a function
grayplot	2D plot of a surface using colors
grayplot properties	description of the grayplot entities properties
graypolarplot	Polar 2D plot of a surface using colors
histplot	plot a histogram
paramfplot2d	animated plot of a 2D parameterized curve.
plot	2D plot
plot2d	2D plot
plot2d2	2D plot (step function)
plot2d3	2D plot (vertical bars)
plot2d4	2D plot (arrows style)
polarplot	Plot polar coordinates
scatter	2D scatter plot

12.3.2 3D Graphics

3D Graphics are handled using 3D plot functions available in Scilab. Some of these functions are shown in Table 12.2.

Table 12.2: 3D Graphics Functions

Function	**Description**
comet3d	3D comet animated plot.
contour	level curves on a 3D surface
eval3d	values of a function on a grid => **this function is obsolete**

eval3dp fac3d	compute facets of a 3D parametric surface 3D plot of a surface. **This function is obsolete.**
fplot3d	3D plot of a surface defined by a function
fplot3d1	3D gray or color level plot of a surface defined by a function
genfac3d	Compute facets of a 3D surface
geom3d	projection from 3D on 2D after a 3D plot
hist3d	3D representation of a histogram
mesh	3D mesh plot
nf3d	Rectangular facets to plot3d parameters
param3d1	plots a set of curves in a 3D cartesian frame. 1D mesh plot
param3d properties	description of the 3D curves entities properties
plot3d	3D plot of a surface
plot3d1	3D gray or color level plot of a surface
plot3d2	plot surface defined by rectangular facets
plot3d3	mesh plot surface defined by rectangular facets
scatter3	3D scatter plot
secto3d	3D surfaces conversion
surf	3D surface plot
surface properties	description of the 3D entities properties

12.4 GENERATING .jpg FILES

In Scilab, we can generate .jpg files from the graphic window by using the native function *xs2jpg*. *xs2jpg* exports the display of a graphic window into a JPG file.

```
xs2jpg(win_num, file_name [, compression_quality])
xs2jpg(fig, file_name [, compression_quality])
```

where, win_num → an integer, ID of the figure to export

fig → handle of the figure to export

file_name → a string, name of the exported file

compression quality → a real scalar, a number between 0 and 1

The default compression quality is set to 0.95.

Example 12.1: Using XS2 jpg function

```
scf(0);    // set the current graphic figure (window)
plot2d();  // 2D plot
//JPG export
xs2jpg(0,'foo.jpg'); // export as filename foo.jpg
xs2jpg(gcf(),'foo.jpg');
// gcf() returns handle of current graphic window
xs2jpg(0,'foo_0.jpg', 0);
// poor quality since compression quality=0
xs2jpg(gcf(),'foo_1.jpg', 1);
// best quality since compression quality=1
```



CHECK YOUR PROGRESS

Q 1: Name two 2D plot and two 3D plot functions.

.....

.....

Q 2: What is the function used to generate .jpg files?

.....

.....

12.5 FUNCTION PLOTTING

In Scilab, function *fplot2d* can be used to plot a 2D plot of a curve defined by a function.

```
fplot2d (xr, f, [style, strf, leg, rect, nax])
fplot2d (xr, f, <opt_args>)
```

Example 12.2: Say for an instance to define a function

f(x) = x |x|/(1+x2)

in the console we would write

```
deff('[y] = f(x)',['y = x*abs(x)/(1+x^2)']);
```

This will define a Scilab function with the name f.

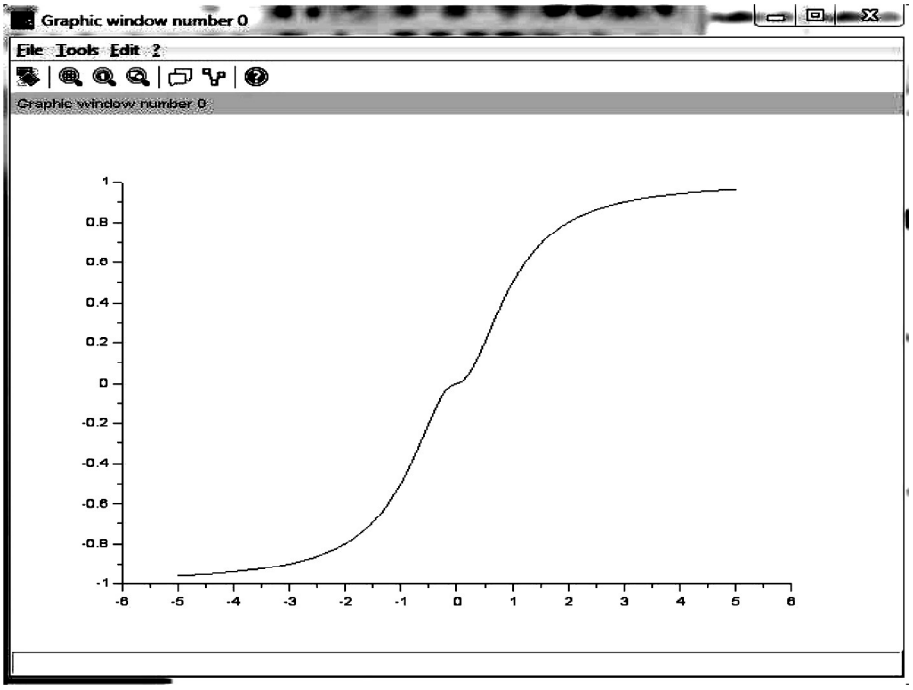
Now to plot the function f in the interval [0,1] we use

```
x = (-5:0.1:5)';
    fplot2d(x, f);
```

The first command produces a column vector of x from 0 to 1 in steps 0.1. The second produces the plot of the function f.

Output:

Figure 12.1: Plot of function f in example 12.2



Similarly, function *fplot3d* can be used to plot a 3D plot of a surface defined by a function.

```
fplot3d (xr, yr, f, [theta, alpha, leg, flag, ebox])
fplot3d (xr, yr, f, <opt_args>)
```

Example 12.3: Say for an instance to define a function

f(x,y) = x4 – y4

in the console we would write

```
    deff('z = f(x,y)','z = x^4 - y^4');
```

This will define a Scilab function with the name f.

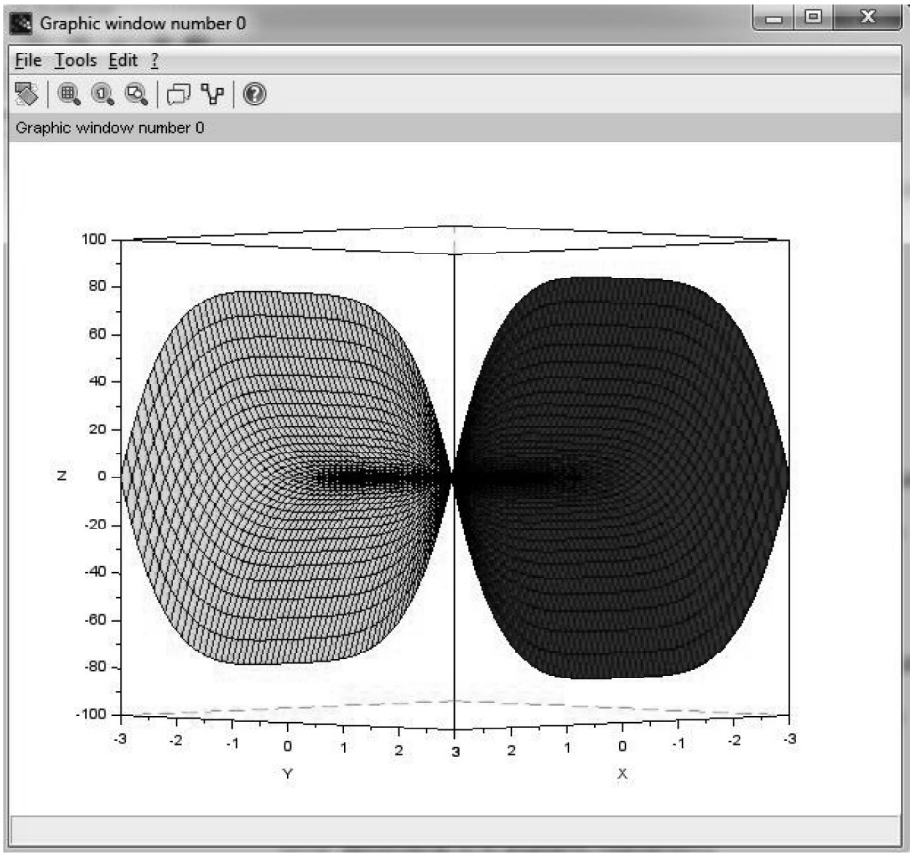
Now to plot the function f in the interval [-3,3] we use

```
x = (-3:0.1:3)';
y=x;
    fplot3d(x, y, f);
```

The first command produces a column vector of x from -3 to 3 in steps 0.1. The second command produces a column vector of y same as x i.e. -3 to 3 in steps 0.1. The third produces the plot of the function f.

Output:

Figure 12.2: Plot of function f in example 12.2



12.6 DATA PLOTTING

In Scilab, function *plot2d* can be used to plot a 2D plot of a given data.

```
plot2d ([logflag,][x, ], y[, style[, strf[, leg[, rect[, nax]]]]])
plot2d ([logflag,][x, ], y, <opt_args>)
```

Example 12.4: Say for an instance to plot the data given below,

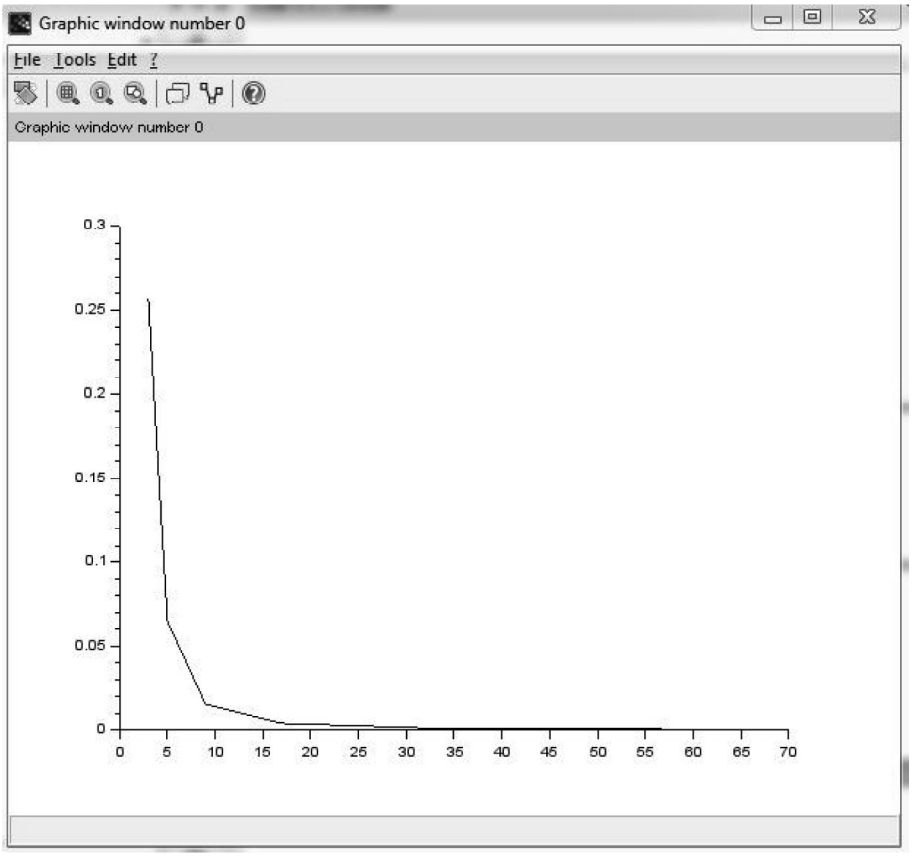
```
n = [ 3 5 9 17 33 65 ]';
s = [ 2.57e-1 6.46e-2 1.51e-2 3.96e-3 9.78e-4 2.45e-4]';
```

```
plot2d (n, s);
```

The command plot2d(n, s) produces the plot of the given data.

Output:

Figure 12.3: Plot of given data in example 12.4



Similarly, function *plot3d* can be used to plot a 3D plot of a given data.

```
plot3d (x, y, z, [theta, alpha, leg, flag, ebox])
plot3d (x, y, z, <opt_args>)
plot3d (xf, yf, zf, [theta, alpha, leg, flag, ebox])
plot3d (xf, yf, zf, <opt_args>)
plot3d (xf, yf, list(zf, colors),[theta, alpha, leg,
flag, ebox])
plot3d (xf, yf, list(zf, colors), <opt_args>)
plot3d (z)
```

Example 12.5: Say for an instance to plot the data given below:

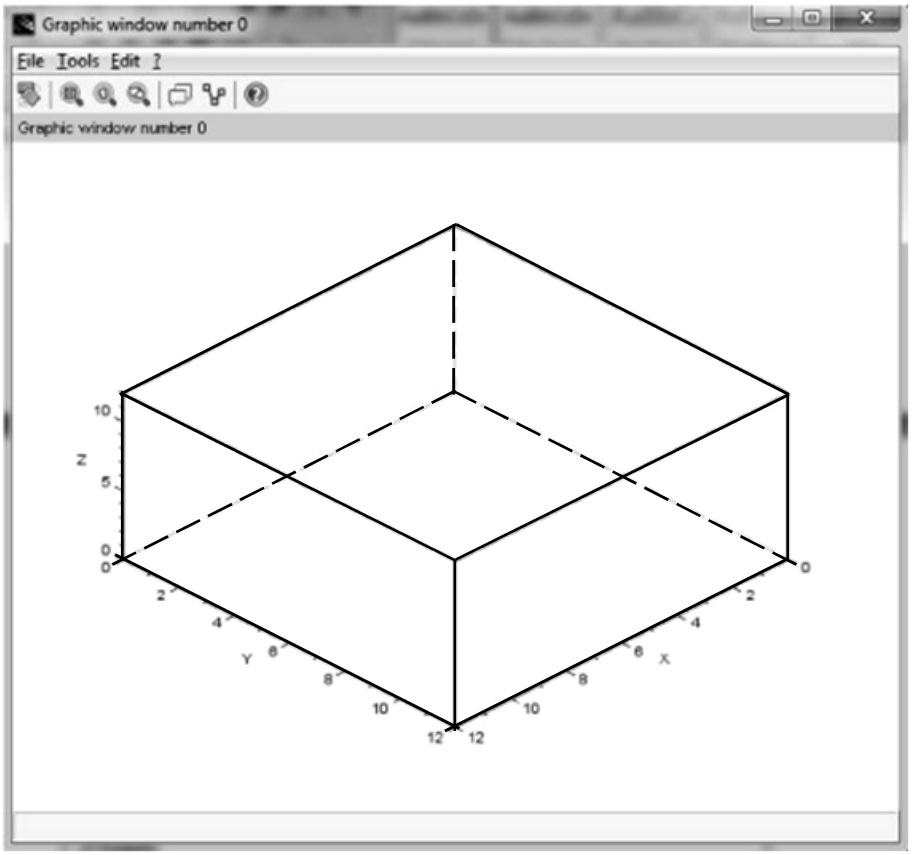
```
x=0:0.1:12;
```

```
y=0:0.1:12;
z=0:0.1:12;
plot3d (x, y, z);
```

The command plot3d(x, y, z) produces the plot of the given data.

Output:

Figure 12.4: Plot of function f in example 12.5





CHECK YOUR PROGRESS

Q 3: Define two functions used to plot a given function in Scilab.

.....

.....

Q 4: Define two functions used to plot given data in Scilab.

.....

.....

12.7 GUI IN SCILAB

A GUI is a human-computer interface that uses graphical objects like windows, menus and icons to interact with users through the use of mouse in keyboard.

The function that creates a GUI object in Scilab is "*uicontrol*".

```
h = uicontrol (PropertyName, PropertyValue,...)
h = uicontrol (parent, PropertyName, Property Value,...)
h = uicontrol (uich)
```

The function *h = uicontrol(PropertyName, PropertyValue,...)* creates an uicontrol and assigns the specified properties and values to it. It assigns the default values to any properties you do not specify. The default uicontrol style is a "Pushbutton". The default parent is the current figure.

The function *h = uicontrol(parent, PropertyName, PropertyValue,...)* creates a uicontrol in the object specified by the handle, parent. If you also specify a different value for the Parent property, the value of the Parent property takes precedence. parent is the handle of a figure.

The function *h = uicontrol(uich)* gives focus to the uicontrol specified by uich.

The style of an uicontrol must be set at creation using the "Style" property and cannot be changed once the uicontrol is created. The available styles are listed below.

- Checkbox: a button with two states
- Edit: an editable string zone.
- Frame: a container for other uicontrols or axes.
- Image: a component where a specified image is displayed.
- Layer: a container for frame style uicontrols.
- Listbox: a control representing a list of items that can be scrolled.
- Popupmenu: a button which makes a menu appear when clicked.
- Pushbutton: a rectangular button generally used to run a callback.
- Radiobutton: a button with two states
- Slider: a scale control, that is a slider used to set values between

in range with the mouse.

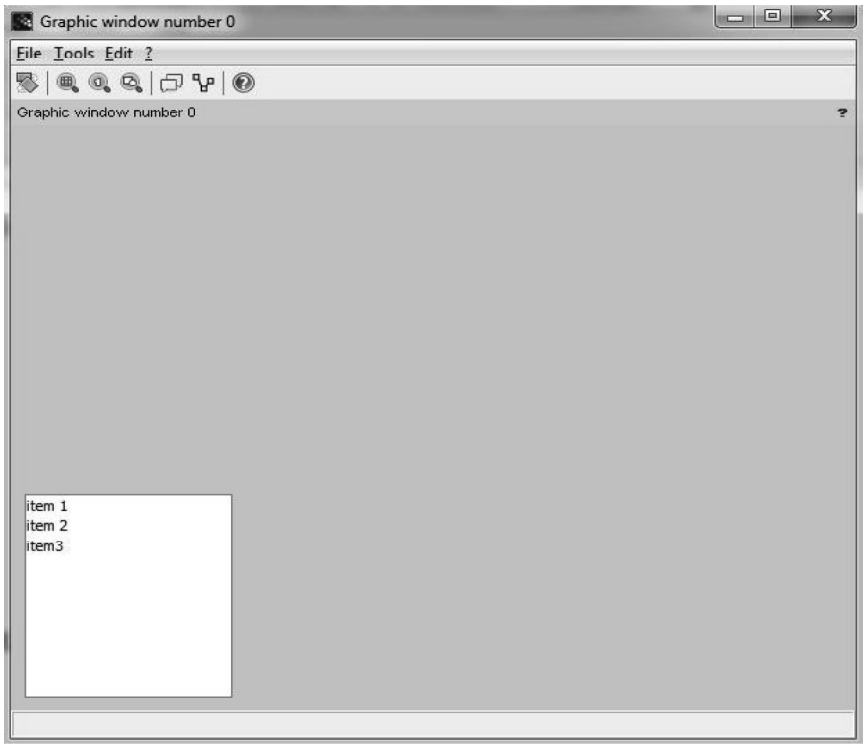
- Spinner: a component which enables the user to select/edit a value between bounds with a fixed step.
- Tab: a container for frame style uicontrols enabling to switch between them by clicking on a tab with a given title and/or icon.
- Table: an editable table.
- Text: a text control

Example 12.6: Using uicontrol() Function

```
f=figure();                    // create a figure
h=uicontrol(f,'style','listbox','position', [10 10 150
160]);           // create a listbox
set(h, 'string', "item 1|item 2|item3");    // fill
the list
set(h, 'value', [1 3]);     // select item 1 and 3 in
the list
close(f);                      // close the figure
```

Output:

Figure 12.5: Graphic window in example 12.6



12.8 PLOTTING 2D GRAPHS

In Scilab, we can create and customize various types of 2D plots. Scilab can create x-y plots, contour plots, pie chart, histograms, bar chart, horizontal bar chart and many other types of plots.

Table 12.3: Plotting 2D Graphs Functions

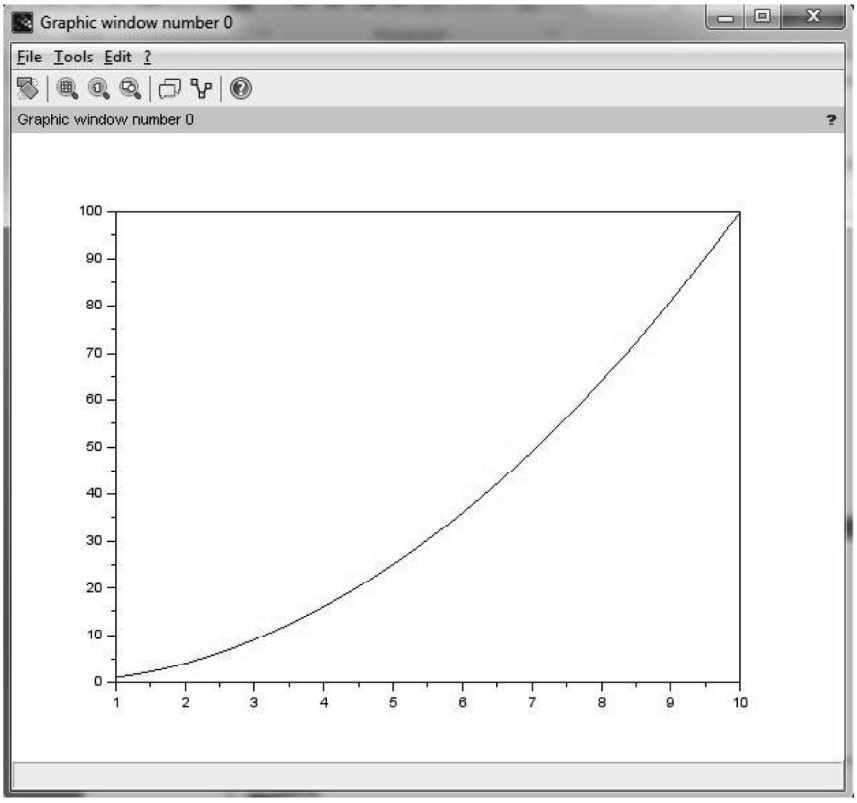
Functions	Descriptions
plot	2D plot
contour	contour plot
pie	pie chart
histplot	histogram
bar	bar chart
barh	horizontal bar chart
polarplot	plot polar coordinates
Matplot	2D plot of matrix using colors
Sgrayplot	smooth 2D plot of surface using colors
grayplot	2D plot of a surface using colors

Example 12.7: Example to create a simple x-y plot.

```
function f=myquadratic(x)
// function squaring the input argument x
f = x^2;
endfunction
xdata = linspace ( 1 , 10 , 50 );
// linespace function to produce 50 values
ydata = myquadratic ( xdata );
// pass myquadratic function and get the function
value
plot ( xdata , ydata );
```

Output:

Figure 12.6: Simple x-y plot from example 12.7



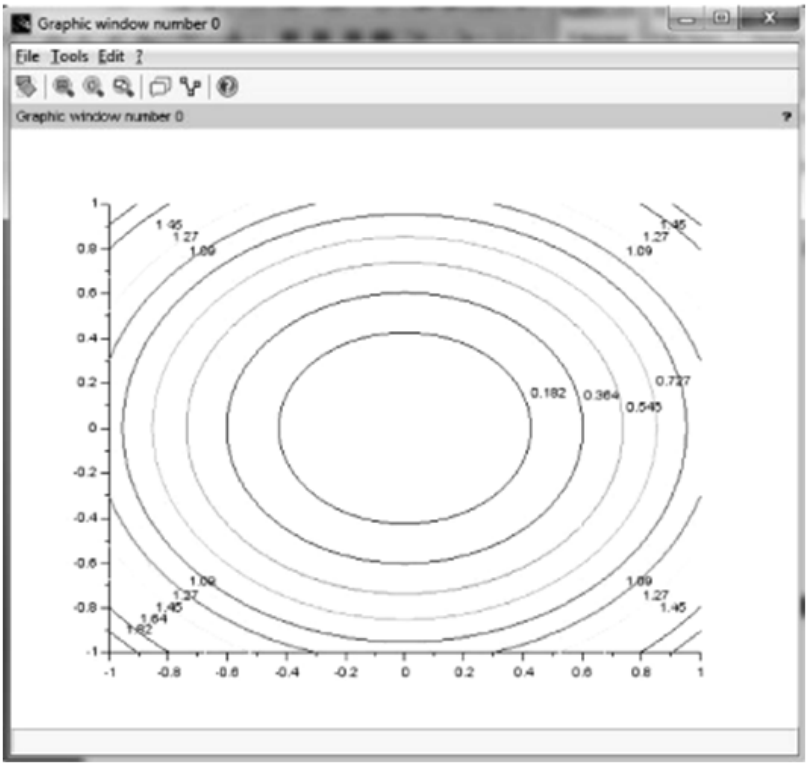
Example 12.8: Example for a contour plot.

```
function f=myquadratic2arg(x1, x2)
// function takes two input arguments x1 and x2
f = x1^2 + x2^2;     // returns f(x1, x2) = x12 +
x22
endfunction

xdata = linspace ( -1 , 1 , 100 );
// linespace function to produce 100 values
ydata = linspace ( -1 , 1 , 100 );
// linespace function to produce 50 values
contour ( xdata , ydata , myquadratic2arg , 10);
```

Output:

Figure 12.7: Contour x-y plot from example 12.8





CHECK YOUR PROGRESS

Q 5: Name two "Style" property of an uicontrol.

.....

.....

Q 6: Name two functions of 2D plotting.

.....

.....



12.9 LET US SUM UP

- Scilab native functions are used to plot graphs.
- Scilab offers many ways to create and customize various types of 2D and 3D plots.

- Function xs2jpg exports the display of a graphic window into a JPG file.
- Function fplot2d can be used to plot a 2D plot of a curve defined by a function.
- Function fplot3d can be used to plot a 3D plot of a surface defined by a function.
- Function plot2d can be used to plot a 2D plot of a given data.
- Function plot3d can be used to plot a 3D plot of a given data.
- Function that creates a GUI object in Scilab is “uicontrol”.
- Scilab can create x-y plots, contour plots, pie chart, histograms, bar chart, horizontal bar chart and many other types of plots.



12.10 FURTHER READING

1) Verma, A. K., (2018). *SCILAB: A beginner’s Approach*.

2) Nagar, S., (2018). *Introduction to Scilab: For Engineers and Scientists*.



12.11 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No 1: LineSpec, Matplot and comet3d, surf

Ans to Q No 2: xs2jpg

Ans to Q No 3: fplot2d, fplot3d

Ans to Q No 4: plot2d, plot3d

Ans to Q No 5: Checkbox, Frame

Ans to Q No 6: pie, histplot



12.12 MODEL QUESTIONS

Q 1: How to plot graphs in Scilab?

Q 2: How to create 2D and 3D plots in Scilab?

Q 3: Name five functions used to create 2D plots in Scilab.

Q 4: Name five functions used to create 3D plots in Scilab.

Q 5: How to create a .jpg file in Scilab?

Q 6: How to plot a 2D plot of a curve defined by a function in Scilab?

Q 7: How to plot a 3D plot of a surface defined by a function in Scilab?

Q 8: How to plot a 2D plot of a given data in Scilab?

Q 9: How to plot a 3D plot of a given data in Scilab?

Q 10: Which function is used to create GUI in Scilab.

Q 11: Name five “Style” property of an uicontrol.

Q 12: How to create 2D graphs in Scilab?

Q 13: Name five functions used to create 2D graphs in Scilab.

*** ***** ***

UNIT 13 : APPLICATIONS OF SCILAB

UNIT STRUCTURE

13.1 Learning Objectives

13.2 Introduction

13.3 Applications

13.3.1 Numerical Linear Algebra

13.3.2 Ordinary Differential Equations

13.4 Numerical Analysis

13.4.1 Iterative Methods

13.5 Comparison with C/C++/Matlab

13.6 Let Us Sum Up

13.7 Further Reading

13.8 Answers to Check Your Progress

13.9 Model Questions

13.1 LEARNING OBJECTIVES

After going through this unit, you will be able to–

- use Scilab applications
- solve linear equations
- solve eigen values
- solve differential equations
- perform numerical analysis using iterative methods
- differentiate between Scilab and C/C++/Matlab

13.2 INTRODUCTION

In the previous unit you had been introduced to Graphics in Scilab. We have learned about handling 2D and 3D graphics and generating .jpg files. We have also learned how to plot functions, data, 2D graphs and how to create GUI in Scilab.

In this unit you will be introduced to applications of Scilab. We will learn about numerical linear algebra by solving linear equations, eigen

values, etc., solving ordinary differential equations, numerical analysis using iterative methods and comparison of Scilab with C/C++/Matlab. In the next unit you will be introduced to Python. We will learn how to install Python and also learn how to open a Python console.

13.3 APPLICATIONS

Scilab being a computing environment for scientific and engineering applications provides various functions for computation of any type of scientific and engineering applications. We will be going through some of these applications and will also be using some of these functions to do some of these computations.

13.3.1 Numerical Linear Algebra

Scilab supports computation of numerical linear algebra. Numerical linear algebra consists of problems like Eigenvalue and Singular value, Factorization, Kernel, Linear Equations, Markov Matrices, Matrix Analysis, Matrix Pencil, State-Space Matrices, Subspaces and projection.

We will be looking at how to solve linear equations and eigen values.

Linear equations:

Say there are three equations with three unknowns:

$$2x_1 + 3x_2 - 5x_3 - 13 = 0$$
$$x_1 - 3x_2 + 8x_3 + 13 = 0$$
$$2x_1 - 2x_2 + 4x_3 + 6 = 0$$

This can be written as the matrix equation A.x + c = 0, if

$$A = \begin{bmatrix} 2 & 23 & -5 \\ 1 & -3 & 8 \\ 2 & -2 & 4 \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \text{and} \quad c = \begin{bmatrix} -13 \\ 13 \\ 6 \end{bmatrix}$$

If a unique solution exists it represents the point of intersection of the three planes in the coordinate system (x_1, x_2, x_3).

Now using Scilab function ***linsolve*** we try to find the solution.

```
—>A = [2, 3, -5; 1, -3, 8; 2, -2, 4]
—>C = [-13; 13; 6]
—>linsolve(A, C)
```

Output:

```
ans =
1.
2.
       -1.
```

The function ***linsolve*** gives the unique solution of the 3 equations.

Eigen value:

Say there is a matrix A.

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

Now using Scilab function ***spec*** we try to find the eigen values.

```
—>A = [2, 2, 0; 1, 2, 1; 0, 1, 2]
—>spec(A)
```

Output:

```
ans=
     0.5857864
     2.
     3.4142136
```

The function *spec* gives the eigen values of the matrix A.

13.3.2 Ordinary Differential Equations

Scilab uses a function ***ode*** to evaluate an ordinary differential equation or a set of coupled first order differential equations.

```
y=ode(y0,x0,x,f)
```

where,

y_0=initial value of y

x_0=initial value of x

x=value of x at which you want to calculate y.

Example 13.1: Solve the given differential equation.

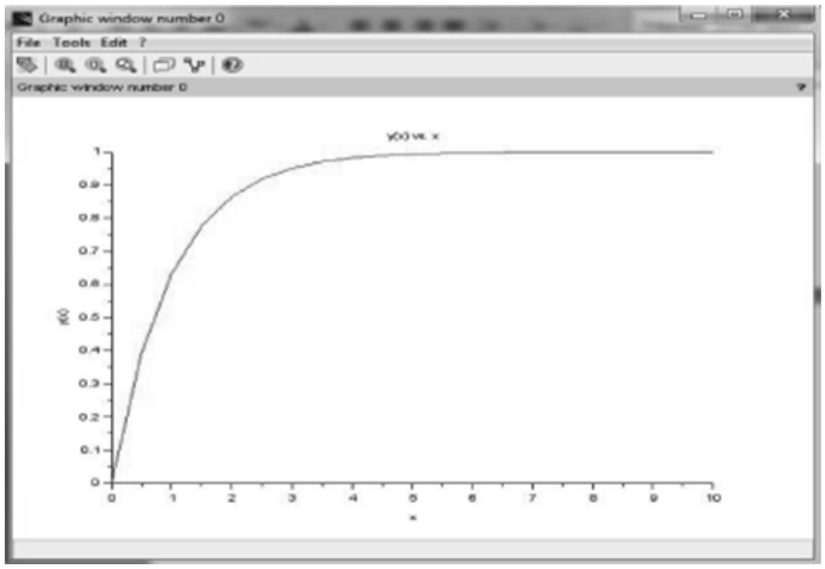
$dy/dx = e^{-x}$ with y=0 for x=0

Solution:

```
funcprot(0)
clf;
function dx=f(x, y)
dx=exp(-x);
end function
y0=0;
x0=0;
x=[0:0.5:10];
sol=ode(y0,x0,x,f);   // function ode
plot2d(x,sol,5)       // plot the solution
xlabel('x');
ylabel('y(x)');
xtitle('y(x) vs. x');
```

Output:

Figure 13.1: Plot for example 13.1



13.4 NUMERICAL ANALYSIS

Scilab supports numerical analysis. In numerical linear algebra, the Jacobi method and Gauss Seidel method are iterative methods used to solve a linear system of equations.

13.4.1 Iterative Methods

In computational mathematics, an iterative method is a procedure that starts from an initial guess to generate a sequence of solutions that converge to the desired solution.

Let us see the computation of solving a linear system of equations using Jacobi and Gauss Seidel method using Scilab.

- **Jacobi Method:**

```
A = [8 , -3, 2; 4, 11, -1; 6 , 3,12];   /   /
equation matrix
B = [20; 33; 36]        // solution matrix
for i =0:19
       X(i+1 ,1)=i;
end
for i =2:4
       X(1,i)=0;
end
for r =1:19
       for i =1:3
             k =0;
             for j =1:3
                   if(i~=j)
                         k=k-A(i,j)*X(r,j+1);
                   end
             end
             X(r+1,i +1) =(k+B(i ,1) )/A(i,i);
       end
end
```

```
printf ( ' r \t x ( r ) \t \t y ( r ) \t
z ( r ) ' );
printf ( ' \n ————————————————————' )
disp (X)
printf ( ' \n \n After 18 iterations exact
solution is : \n x=%i \t y=%i \t z=%i ' ,X(19
,2) ,X (19 ,3) ,X (19 ,4) )
```

Output:

Figure 13.2: Output of Jacobi method

Scilab 6.0.2 Console

```
--> exec('C:\Users\ANUPAM\jacobi.sci', -1)
r        x ( r )         y ( r )         z ( r )
 ---------------------------------------------------
   0.    0.          0.          0.
   1.    2.5         3.          3.
   2.    2.875       2.3636364   1.
   3.    3.1363636   2.0454545   0.9715909
   4.    3.0241477   1.9478306   0.9204545
   5.    3.0003228   1.9839876   1.0009685
   6.    2.9937532   1.9999707   1.0038417
   7.    2.9990286   2.0026208   1.0031307
   8.    3.0002001   2.0006379   0.9998305
   9.    3.0002816   1.9999118   0.9997405
   10.   3.0000318   1.999874    0.9998813
   11.   2.9999824   1.9999776   1.0000156
   12.   2.9999877   2.0000078   1.0000144
   13.   2.9999993   2.0000058   1.0000042
   14.   3.0000011   2.0000006   0.9999989
   15.   3.0000005   1.9999995   0.9999993
   16.   3.          1.9999997   0.9999999
   17.   2.9999999   2.          1.0000001
   18.   3.          2.          1.
   19.   3.          2.          1.


 After 18 iterations exact solution is :
 x=2     y=2     z=1
-->
```

- **Gauss Seidel Method**

```
A = [8, -3, 2; 4, 11, -1; 6, 3, 12];
//equation matrix
B = [20; 33; 36]     // solution matrix
```

```
for  i  =0:10
       X(i+1  ,1)=i;
end
for  i  =2:4
       X(1,i)=0;
end
for  r  =1:10
       for  i  =1:3
             k1  =0;
             for  j=1:i  -1
                   k1=k1 -A(i,j)*X(r+1,j +1) ;
             end
             k2  =0;
             for  j=i  +1:3
                   k2=k2  -A(i,j)*X(r,j  +1)  ;
             end
             X(r+1,i +1)  =( k1+k2+B(i  ,1)  )/
             A(i,i);
       end
end
printf ( ' r \t x ( r ) \t \t y ( r ) \t z (
r ) ' );
printf ( ' \n ————————————————————' )
disp (X)
printf ( ' \n \nAfter 9 iterations exact
solution is : \n x =%i \ty=%i \t z=%i '
,X(10 ,2) ,X (10 ,3) ,X (10 ,4) )
```

Output:

Figure 13.3: Output of Gauss Seidel method

```
Scilab 6.0.2 Console

--> exec('C:\Users\ANUPAM\gauss_seidel.sci', -1)
 r       x ( r )          y ( r )          z ( r )
 ------------------------------------------------
   0.    0.               0.               0.
   1.    2.5              2.0909091        1.2272727
   2.    2.9772727        2.0289256        1.0041322
   3.    3.009814         1.9968069        0.9958912
   4.    2.9998298        1.9996884        1.000163
   5.    2.9998424        2.0000721        1.0000608
   6.    3.0000119        2.0000012        0.9999938
   7.    3.000002         1.9999987        0.9999993
   8.    2.9999997        2.0000001        1.0000001
   9.    3.               2.               1.
   10.   3.               2.               1.
   11.   2.9999824        1.9999776        1.0000156
   12.   2.9999877        2.0000078        1.0000144
   13.   2.9999993        2.0000058        1.0000042
   14.   3.0000011        2.0000006        0.9999989
   15.   3.0000005        1.9999995        0.9999993
   16.   3.               1.9999997        0.9999999
   17.   2.9999999        2.               1.0000001
   18.   3.               2.               1.
   19.   3.               2.               1.


After 9 iterations exact solution is :
 x=2     y=2       z=1
-->
```



CHECK YOUR PROGRESS

Q 1: Name the functions used to solve linear equations and eigen values.

.....

.....

Q 2: Name the function used to evaluate an ordinary differential equation.

.....

.....

13.5 COMPARISON WITH C/C++/MATLAB

Let us now compare Scilab with C/C++ and Matlab and learn how the different programming languages differ from each other. The comparison between Scilab and C/C++ is given in the table 13.1 below.

Table 13.1: Comparison between Scilab and C/C++

Scilab	C/C++
One-based indexing delimited by parentheses and commas. For example, A(4,5)	Zero-based indexing delimited by square brackets. For example, A[3][4].
a++, a—, a +=b, a -=b, a *=b, a \=b are not supported	a++, a—, a +=b, a -=b, a *=b, a \=b are supported
if, *for* and *while* statements ends with an *end*	End of *if*, *for* and *while* statements are ended by using parenthesis
Complex number notation a + bi is supported	Complex number notation is not supported
String data not supported	String data supported

Now, let us look at some of the differences between Scilab and Matlab. The comparison between Scilab and Matlab is given in the table 13.2 below.

Table 13.2: Comparison between Scilab and Matlab

Scilab	Matlab
One or more functions can be defined in a single file.	Only one function can be defined in a single file.
Functions are not automatically loaded	Functions are automatically loaded
Functions can be defined on-line or inside functions	Functions cannot be defined on-line
exec(“filename”) is used to execute a script file	Just need to type the name of the file
Scilab comments begins with //	Matlab comments begins with %
Predefined variables are prefixed with % in Scilab	Predefined variables are not prefixed
Boolean variables are %T and %F	Boolean variables are 0 and 1



13.6 LET US SUM UP

- Scilab supports computation of numerical linear algebra.
- The function **linsolve** is used to solve linear equations.
- The function **spec** gives the eigen values of the matrix A.
- The function **ode** is used to evaluate an ordinary differential equation or a set of coupled first order differential equations.
- In computational mathematics, an iterative method is a procedure that starts from an initial guess solution to generate a sequence of solutions that converge to the desired solution.



13.7 FURTHER READING

1) Verma, A. K., (2018). *SCILAB: A beginner's Approach*.
2) Nagar, S., (2018). *Introduction to Scilab: For Engineers and Scientists*.



13.8 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No 1: *linsolve* and *spec*

Ans to Q No 2: *ode*



13.9 MODEL QUESTIONS

Q 1: What are some problems in numerical linear algebra?

Q 2: What function is used to solve linear equations in Scilab?

Q 3: What function is used to solve eigen values in Scilab?

Q 4: What function is used to evaluate ordinary differential equation in Scilab?

Q 5: What are iterative methods?

Q 6: Name two iterative methods.

Q 7: Mention five differences of Scilab and C/C++.

Q 8: Mention five differences of Scilab and Matlab.

*** ***** ***

UNIT 14 : INTRODUCTION TO PYTHON

UNIT STRUCTURE

14.1 Learning Objectives

14.2 Introduction

14.3 Python

14.4 Installation of Python

14.4.1 Python installation in Windows

14.4.2 Python installation in Linux

14.5 Opening Python Console

14.5.1 Windows

14.5.2 Linux

14.6 Let Us Sum Up

14.7 Further Reading

14.8 Answers to Check Your Progress

14.9 Model Questions

14.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- define Python
- describe the steps for installing Python
- perform installation of Python
- open Python console in both Windows and Linux

14.2 INTRODUCTION

In the previous unit you had been introduced to applications of Scilab like numerical linear algebra (solving linear equations, eigen values etc.), solving ordinary differential equations and numerical analysis using iterative methods.

In this unit you will be introduced to Python. We will learn how to install Python and also learn how to open a Python console. In the next unit you will be introduced to some basic Python commands. We will also

learn the basics of python programming in the next unit.

14.3 PYTHON

Python is a high level object oriented programming language. It is an interpreted programming language since we can execute instructions directly and freely without needing to previously compile a program before execution. It is also an interactive language since we can use a Python prompt and interact directly with the interpreter to write programs.

14.4 INSTALLATION OF PYTHON

The procedure to install python in Windows and Linux environment are described in detail in the subsections below.

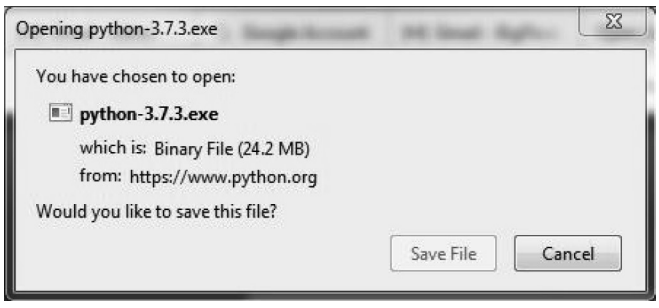
14.4.1 Python installation in Windows

For installation of Python in Windows, we need to follow the instructions given below:

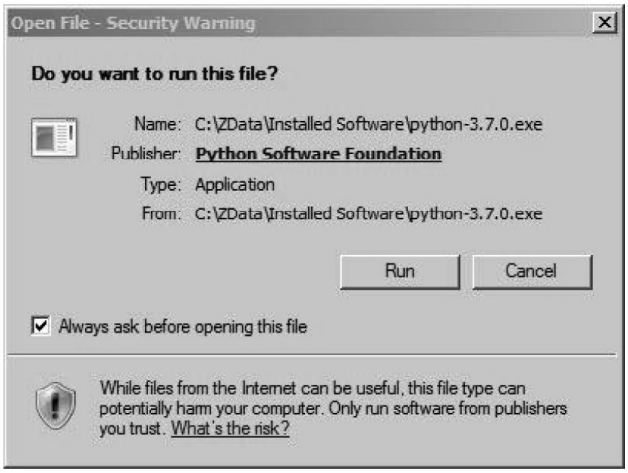
Step 1: We first need to download the latest Python version from the website *https://www.python.org/downloads/.*



Once we visit the website, we then save the downloaded file in a permanent location.



Step 2: We need to double-click the icon labeling the file python-x.x.x.exe. An **Open File - Security Warning** pop-up window will appear like shown below.

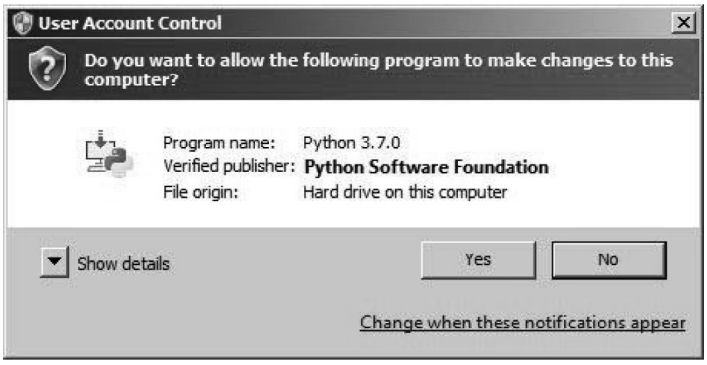


Step 3: Click Run. A Python 3.7.0 (32-bit) Setup pop-up window will appear.



Ensure that the "Install launcher for all users (recommended)" and the "Add Python 3.7 to PATH" checkboxes at the bottom are checked.

Step 4: Highlight the "Install Now" message, and then click it. A User Account Control pop-up window will appear, posing the question "Do you want to allow the following program to make changes to this computer?"



Step 5: We then need to click the **Yes** button. A new Python 3.7.0 (32-bit) Setup pop-up window will appear with a Setup Progress message and a progress bar. Soon, a new Python 3.7.0 (32-bit) Setup pop-up window will appear with a Setup was successfully message.



Step 6: Click the Close button.

Step 7: We first navigate to the directory where Python was installed. We then double click the icon/file python.exe. The following command prompt pop-up window will appear. We can now type our commands in this window.



14.4.2 Python installation in Linux

For installation of Python in Linux, we need to follow the instructions given below:

Step 1: First, we need to check if Python is already installed in the system. To see if Python is already installed, open a command prompt and run.

$ python —version or *$ python3 –version*

Step 2: If Python 2.7 or later is not installed, install Python with your distribution's package manager. The command and package name varies:

a. On Debian derivatives such as Ubuntu, use apt.

$ sudo apt-get install python3

b. On Red Hat and derivatives, use yum.

$ sudo yum install python3

c. On SUSE and derivatives, use zypper.

$ sudo zypper install python3

Step 3: Open a command prompt or shell and run the following command to verify that Python installed correctly.

$ python3 —version

14.5 OPENING PYTHON CONSOLE

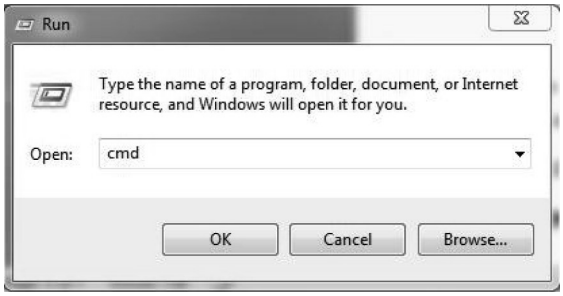
In this section, we will look at how to open the Python console in both Windows and Linux.

14.5.1 Windows

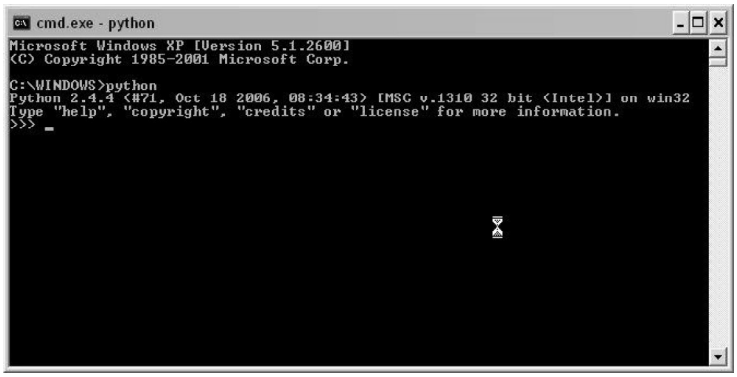
For opening the console in Windows, we need to follow the instructions given below:

Step 1: We first open the Command line. This is done by clicking the **Start Menu** and then typing **run** in the search bar. Typing run will open the Run window where we need to type "**cmd**".

Start menu -> Run and type *cmd*



Step 2: Once we get to the command prompt window, we type *python*.

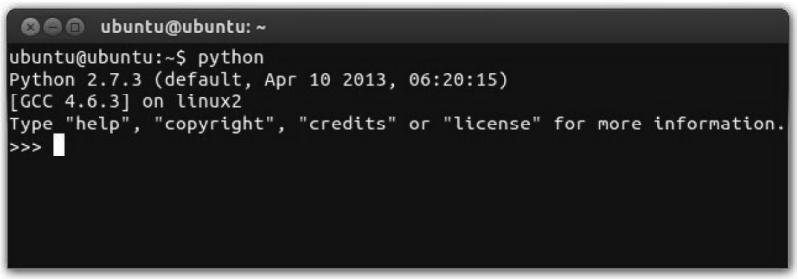


14.5.2 Linux

For opening the console in Linux, we need to follow the instructions given below:

Step 1: Open a command terminal.

Step 2: Type *python*.





CHECK YOUR PROGRESS

Q 1: State whether true or false.

a. Python is a low level language.

b. Python is an interpreted programming language.

c. Python is an interactive language.

Q 2: From where can we download Python for windows installation?

.....

.....

Q 3: What command is used in Linux to know if Python is installed or not?

.....

.....



14.6 LET US SUM UP

- Python is a high level language.
- Python is an object oriented programming language.
- Python is an interpreted programming language.
- Python is an interactive programming language.
- Latest Python version for Windows can be downloaded from *https://www.python.org/downloads/.*
- To see if Python is already installed in Linux environment, we open a command prompt and run *"python –version"* or *"python3 –version".*
- On Debian derivatives such as Ubuntu, we use apt and run command *"sudo apt-get install python3".*
- On Red Hat and derivatives, we use yum and run command "*sudo yum install python3".*
- On SUSE and derivatives, we use zipper and run command "*sudo zypper install python3".*
- For both Windows and Linux environment we use command "*python*" to start Python.



14.7 FURTHER READING

1) Barry, P. (2016). *Head First Python: A Brain-Friendly Guide. " O'Reilly Media*, Inc.".

2) Matthes, E. (2015). *Python crash course: a hands-on, project-based introduction to programming*. No Starch Press.

3) Zelle, J. M. (2004). *Python programming: an introduction to computer science*. Franklin, Beedle & Associates, Inc..



14.8 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No 1: a. False

b. True

c. True

Ans to Q No 2: *https://www.python.org/downloads/*

Ans to Q No 3: *$ python —version* or *$ python3 —version*



14.9 MODEL QUESTIONS

Q 1: What is Python?

Q 2: Why Python is an interpreted programming language?

Q 3: How is Python is an interactive programming language?

Q 4: From where can we download the latest Python version for Windows Environment?

Q 5: How can we install Python in Linux environment?

Q 6: How can we find if Python is installed in Linux environment?

Q 7: How do we find the Python version installed in Linux environment?

*** ***** ***

UNIT 15 : BASIC PYTHON COMMANDS

UNIT STRUCTURE

15.1 Learning Objectives

15.2 Introduction

15.3 Basic Python Commands

- 15.3.1 Python in Interpreter Mode
- 15.3.2 Loading Python Commands from the Library
- 15.3.3 Defining own Functions in Python

15.4 Python Programming

- 15.4.1 Identifiers
- 15.4.2 Keywords
- 15.4.3 Standard Data Types
- 15.4.4 Operators
- 15.4.5 Decision Making
- 15.4.6 Loops
- 15.4.7 Functions

15.5 Writing Python Programs

15.6 Let Us Sum Up

15.7 Further Reading

15.8 Answers to Check Your Progress

15.9 Model Questions

15.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- run Python commands in interpreter mode
- load Python Commands from the Library
- describe basic Python commands
- describe Python programming basics like data types, operators
- write simple Python programs
- write functions in Python

15.2 INTRODUCTION

In the previous unit you have been introduced to Python installation process in both Windows and Linux environment. We have learned about the different steps involved in installing Python. We have also learned about how to open a Python console in both the environments.

In this unit you will be introduced to basic Python commands and learn python programming. We will discuss all the programming basics of Python- data types, operators, decision making statements and loops in detail in this unit. We will also learn how to use functions using Python programming language. This is the last unit of this course.

15.3 BASIC PYTHON COMMANDS

There are various ways of executing Python commands. They are discussed in the sections below:

15.3.1 Python in Interpreter mode

The easiest way to run Python commands is as an interpreter. In the interpreter mode, we just enter a command and then we get an answer from the Python environment. We know that Python is running in interpreter mode if we get the following message in Windows environment.

```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019,
21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license"
for more information.
>>>
```

For Linux environment, we know that Python is running in interpreter mode if we get the following message

```
Python 3.7.3 (default, Mar 25 2019, 21:26:53)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license"
for more information.
>>>
```

Here, in the above messages the three symbols **>>>** indicates that the prompt is waiting for input.

The >>> prompt can be followed by any text. The Python interpreter can also be used as a calculator.

```
>>>2*1024
2048

>>>1+2+3+4
10

>>>2**10
1024
```

In the above code, multiplication is represented by *, addition by + and exponents by **.

Below is an example to print a table of perfect squares.

```
>>>for n in [1, 2, 3, 4, 5, 6]
…     print(n**2)
…
1
4
9
25
36
```

In the interpreter mode we can quit a Python session by typing “CTRL+D” (i.e. hold down the CTRL key while pressing the D key.) We can also quit by using the menu. If the interpreter gets stuck in an infinite loop, you can also quit the current execution by CTRL+C.

15.3.2 Loading Python commands from the library

Python has a very extensive library of commands, documented in the *Python Library Reference Manual*. These commands are organized into *modules*. One of the available

modules that is especially useful for us is the math module. We can use it as shown below

```
>>> from math import sqrt , exp
>>> exp(-1)
0.36787944117144233
>>> sqrt(2)
1.4142135623730951
```

We first import the **sqrt** and **exp** functions from the math module, then use them to compute e^{-1}= 1/e and $\sqrt{2}$.

Once we have loaded a function from a module, it is available for the rest of that session. When we start a new session, we have to reload the function if we need it.

Note that we can import all the functions in the math module by using a *:

```
>>> from math import *
```

15.3.3 Defining own functions in Python

In Python we can define and use our own functions. If we need to do a calculation only once, then we use the interpreter. But when we need to perform a certain type of calculation many times, then we can define a function. For example,

```
>>> def square(x):
...   return x*x
...
```

defines the squaring function square(x) =x^2, which is a popular example used in elementary math courses. We can use the function to calculate the square of any given input:

```
>>> def square(x):
...   return x*x
...
>>> square (3)
9
```

```
>>> square (2.5)
6.25
```

A function name is not completely arbitrary, since we are not allowed to use a reserved word as a function name.



CHECK YOUR PROGRESS

Q 1: What is an interpreter mode in Python?

.....

.....

Q 2: How to import all the functions in the math module?

.....

.....

15.4 PYTHON PROGRAMMING

Python program files are saved with **.py** extensions. Say we have a Python program file with name *my_first.py*, the program can be run using the command "python" in interpreter mode.

```
python my_first.py
```

The program is executed and it continues until the program is finished executing.

15.4.1 Identifiers

An identifier is a name given to a variable, function, class, module or other objects. An identifier starts with an underscore (_) or an alphabet followed by more alphabets, underscores or digits.

15.4.2 Keywords

There are certain keywords in Python which are reserved and cannot be used as a constant or an identifier names. These keywords contain lowercase letters only. The keywords are:

and, exec, not, assert, finally, or, break, for, pass, class, from,

print, continue, global, raise, def, if, return, del, import, try, elif, in, while, else, is, with, except, lambda, yield.

15.4.3 Standard Data Types

There are five standard data types in Python. They are described below:

- **Numbers –** Python supports four different numerical types : *int*, *long*, *float* and *complex* Example, *var1 = 1*.
- **Strings –** Strings are contiguous set of characters represented in quotation marks. Example, str = 'Hello World!'.
- **Lists –** Lists contains items separated by commas and enclosed with square brackets ([ ]). These items can be of different data types. Example, list = ['abcd', 123, 3.14, 'tom'].
- **Tuples –** Tuples are similar to lists containing items separated by commas but enclosed in parenthesis (( )) and whose items cannot be updated. Example, tuple = ('abcd', 123, 3.14, 'tom').
- **Dictionary –** Dictionaries are kind of hash tables where a dictionary key can be almost any Python type and values can be any arbitrary Python object which are enclosed by curly braces ({ }) and values are assigned and accessed using square braces ([ ]). Example, dict = { 'name' : 'tom', 'code' : 123, 'value' : 3.14}.

15.4.4 Operators

Python has many different types of operators. These functions of these operators are described in detail below.

- **Arithmetic Operators**

Operator	Operations
+	Addition
-	Subtraction
*	Multiplication

/	Division
%	Modulus
**	Exponent
//	Floor Division

- **Comparison Operators**

Operators	Operations
==	Equal
!=	Not equal
<>	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

- **Assignment Operators**

Operators	Operations
=	Assign values right operand to left operand
+=	Add right operand to left operand and assign result to left operand
-=	Subtract right operand from left operand and assign result to left operand
*=	Multiply right operand with left operand and assign result to left operand
/=	Divide left operand with right operand and assign result to left operand
%=	Take modulus using two operands and assign result to left operand
**=	Exponential (power) calculation on operands and assign result to left operand
//=	Floor division on operands and assign result to left operand

- **Bitwise Operators**

Operators	Operations
&	Binary AND
|	Binary OR

^	Binary XOR
~	Binary Ones Complement
<<	Binary left shift
>>	Binary right shift

- **Logical Operators**

Operators	Operations
and	Logical AND
or	Logical OR
Not	Logical NOT

- **Membership Operators**

Operators	Operations
=	Assign values right operan
in	True if it finds a variable in the specified sequence and false otherwise.
not in	True if it does not finds a variable in the specified sequence and false otherwise.

- **Identity Operators –** is, is not.

Operators	Operations
is	True if the variables on either side of the operator point to the same object and false otherwise.
is not	False if the variables on either side of the operator point to the same object and true otherwise.

15.4.5 Decision Making

Decision making are actions taken considering some conditions. Some of the common decision making statements are – *if* statement, *if else* statement and *nested if* statements. We already know how to use these statements in programming languages like C, now let us look how these can be used in Python.

***if* statement**

```
if ( value == 100)
```

```
     print ("value is equal to 100.")
```

***if else* statement**

```
if ( value == 100)
     print ("value is equal to 100.")
else
     print ("value is not equal to 100.")
```

***nested if* statement**

```
if ( value > 1)
     if ( value < 100 )
          print ("value is greater than 1 and
          less 100.")
```

15.4.6 Loops

A loop statement allows us to execute a statement or group of statements. We have already come across different types of loops like *while* loop, *for* loop and *nested* loops in other programming languages like C, C++ etc. Let us look at some examples on how to use these different loops in Python.

***while* loop**

```
     i = 1
     while ( i <= 10)
          print ( i )
          i++
```

***for* loop**

```
     mylist = [ "Hello", "World"]
     for i in mylist:
          print ( i )
```

***nested* loop**

```
     mylist = [ 1, 2, 3 ]
     for i in mylist:
```

```
for j in mylist:
    print ( i*j )
```

Loop control statements are used to change execution of a loop's normal sequence.

Control Statement	Description
break statement	Terminates a loop statement and executes the statement after the loop
continue statement	Skips the remainder of loop statements and retest its condition
pass statement	Used when a loop statement is required but not to be executed

15.4.7 Functions

A function is a block of reusable code that is used to perform a single related action. The syntax for defining a function is given below.

```
def function_name( parameters ):
    "function_docstring"
    function_suite
    return [expression]
```

For example, let us look at a python function below:

```
def print_str( str ):
    "This prints a passed string into
    this function"
    print str
    return
```

A function is called by just calling it with necessary parameters. For example,

```
print_str("Calling function print_str!")\
```

15.5 WRITING PYTHON PROGRAMS

In this section, we will write Python programs using the various

programming concepts that we have learned in the earlier sections. Let us now write our first Python program for adding two numbers.

Program 15.1: Sum of two numbers

```
a = int(input("enter first number: "))
b = int(input("enter second number: "))
sum = a + b
print ("sum:", sum)
```

The above Python program 15.1 adds two integer numbers. Variables *a* and *b* are two integers to be added and its values are to be input by the user. The addition of the two numbers is stored in the variable *sum* and printed out using **print** function which is a in-built function.

Figure 15.1: Output of program 15.1



Program 15.2: Check if a number is even or odd

```
def is_even( num ):
    if (num % 2) == 0:
        print("{0} is Even".format(num))
    else:
        print("{0} is Odd".format(num))
    return
num = int(input("Enter a number: "))
is_even( num );
```

The above Python program checks if a number is an even or an odd number. Variable *num* is an integer variable and its value is inputted by the user. The value of *num* is passed to function *is_even* which using *if else* statement checks if *num* is even or odd and prints out the result.

Figure 15.2: Output of program 15.2



Program 15.3: Print nth Fibonacci number

```
def Fibonacci(n):
    if n<0:
        print("Incorrect input")
    elif n==1:
        return 0
    elif n==2:
        return 1
    else:
        return Fibonacci(n-1)+Fibonacci(n-2)
num = int(input("Enter value of n: "))
print(Fibonacci(num))
```

The above Python program finds the n-th Fibonacci number. The value of *n* is input by the user. Using the function *Fibonacci(n)* recursively the n-th Fibonacci number is found out.

Figure 15.3: Output of program 15.3

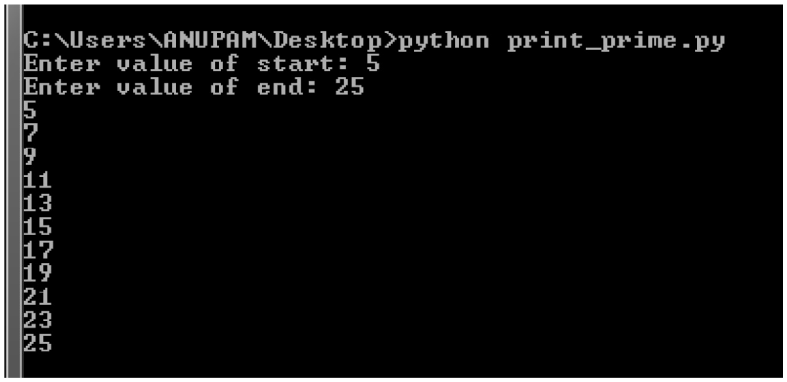
```
C:\Users\ANUPAM\Desktop>python nth_fibonnaci.py
Enter value of n: 10
34
```

Program 15.4: Prime numbers in an interval

```
start = int(input("Enter value of start: "))
end = int(input("Enter value of end: "))
for val in range(start, end + 1):
    if val > 1:
        for n in range(2, val):
            if (val % n) == 0:
                break
            else:
                print(val)
                break
```

The above program finds the prime numbers in an interval. The *start* and the *end* values are inputted by the user. Checking if the number is divisible by any number between 2 and the number, we find the prime numbers which are divided only by it.

Figure 15.4: Output of program 15.4



Program 15.5: Reversing a List

```
def Reverse(mylist):
     return [ele for ele in reversed(mylist)]


mylist = [10, 11, 12, 13, 14, 15]
print(Reverse(mylist))
```

The above program reverses the contents of the list *mylist*. In this program inbuilt function ***reversed()*** is used to reverse the list.

Figure 15.5: Output of program 15.5



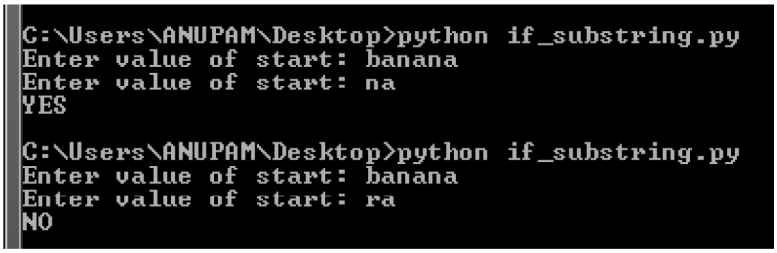
Program 15.6: Check if a substring is present in a given string

```
def check(string, sub_str):
     if (string.find(sub_str) == -1):
          print("NO")
     else:
          print("YES")


string = str(input("Enter value of string: "))
sub_str = str(input("Enter value of sub string:
"))
check(string, sub_str)
```

The above program checks if substring *sub_str* is present in string *string*. In this program inbuilt function ***find()*** is used to check if the substring is present in the string.

Figure 15.6: Output of program 15.6



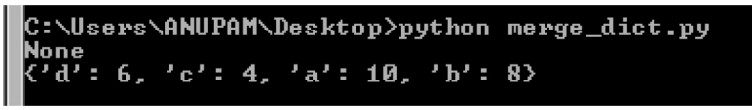
Program 15.7: Merge two dictionaries

```
def Merge(dict1, dict2):
    return(dict2.update(dict1))

dict1 = {'a': 10, 'b': 8}
dict2 = {'d': 6, 'c': 4}
print(Merge(dict1, dict2))
print(dict2)
```

The above program merges two dictionaries. The dictionaries *dict1* and *dict2* are merged using the inbuilt function ***update()***.

Figure 15.7: Output of program 15.7





CHECK YOUR PROGRESS

Q 3: What is the extension of a python file?

.....

.....

Q 4: Name three keywords in Python.

.....

.....



15.6 LET US SUM UP

- Python has an interpreter mode.
- In the interpreter, we just enter a Python command and it gives the answer.

- Python interpreter can be used as a calculator.
- Interpreter mode can be quit by CTRL-D.
- If the interpreter gets stuck in an infinite loop, you can quit the current execution by CTRL-C.
- In Python commands are organized into modules.
- Python functions can be loaded from modules.
- In Python we can define and use our own functions.
- When we need to perform a certain type of calculation many times, then we can define a function.
- A function name is not completely arbitrary.
- Python program files are saved with .py extensions.
- An identifier is a name given to a variable, function, class, module or other objects.
- Python have reserved keywords which cannot be used as a constant or an identifier names.
- Python has five standard data types - numbers, strings, lists, tuples and dictionaries.
- Python has seven types of operators - arithmetic, comparison, assignment, bitwise, logical, membership and identity operators.
- A loop statement allows us to execute a statement or group of statements.
- Python has three types of loops - while, for and nested loops.
- Python has three control statements - break, continue and pass statement.
- A function is a block of reusable code that is used to perform a single related action.
- A function is called by just calling it with necessary parameters.



15.7 FURTHER READING

1) Barry, P. (2016). Head First Python: A Brain-Friendly Guide. " O'Reilly Media, Inc.".

2) Matthes, E. (2015). Python crash course: a hands-on, project-based introduction to programming. No Starch Press.

3) Zelle, J. M. (2004). Python programming: an introduction to computer science. Franklin, Beedle & Associates, Inc..



15.8 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No 1: Interpreter mode in Python is where we just enter a command and Python gives the answer.

Ans to Q No 2: By running the command "*from math import **" in interpreter mode.

Ans to Q No 3: The extension is *.py.*

Ans to Q No 4: Three keywords in Python are: and, exec, not.



15.9 MODEL QUESTIONS

Q 1: What is a Python interpreter mode?

Q 2: How can we quit a Python interpreter mode?

Q 3: How to quit a Python interpreter mode when it gets stuck in infinite loop?

Q 4: From where can we load Python commands?

Q 5: Give an example of how to load a Python command.

Q 6: How can we define our own function in Python?

Q 7: What is an identifier?

Q 8: What are standard data types in Python?

Q 9: What are types of operators in Python?

Q 10: What are the different loops in Python?

Q 11: What are the control statements in Python?

Q 12: What is a function?

Q 13: How do we call a Python function?

Q 14: How to write a function in Python?

Q 15: Write a Python program to find greatest of two numbers.

Q 16: Write a Python program to print Fibonacci numbers.

Q 17: Write a Python program to find smallest number in a list.

Q 18: Write a Python program to count positive and negative numbers in a list.

Q 19: Write a Python program to split and join a string.

Q 20: Write a Python program to remove a key from a dictionary.

*** ***** ***



REFERENCES (For All Units of the Course)

Books:

1) Mittelbach, F., Goossens M.,The LaTeX Companion, 2nd ed. Addison-Wesley(2004), ISBN 0-201-36299-6.
2) Lamport L.: LATEX, A Document Preparation System, User's Guide and Reference Manual, 2nd ed., Addison-Wesley (1994), ISBN 0-201-52983-1.
3) Knuth, D. E., & Bibby, D. (1984). The texbook (Vol. 3). Reading: Addison-Wesley.
4) Heck, A. (2005). Learning LATEX by Doing. AMSTEL Institute.
5) Goossens, M., Rahtz, S. P., Rahtz, S., &Mittelbach, F. (1997). The LATEX graphics companion: illustrating documents with TEX and PostScript (Vol. 1). Addison-Wesley Professional.
6) Kottwitz, S. (2011). LaTeX beginner's guide. Packt Publishing Ltd.
7) Mittelbach, F., Goossens, M., Braams, J., Carlisle, D., & Rowley, C. (2004). The LATEX companion. Addison-Wesley Professional.
8) Kopka, H., & Daly, P. W. (1999). A Guide to La T E X: document preparation for beginners and advanced users.
9) Jackson, A. L., &Geckeis, C. K. (2003). How to prepare your curriculum vitae. McGraw-Hill.
10) Grätzer, G. (2013). Math into LATEX. Springer Science & Business Media.
11) Verma, A. K., (2018). SCILAB: A beginner's Approach.
12) Nagar, S., (2018). Introduction to Scilab: For Engineers and Scientists.
13) Barry, P. (2016). Head First Python: A Brain-Friendly Guide. " O'Reilly Media, Inc.".
14) Matthes, E. (2015). Python crash course: a hands-on, project-based introduction to programming. No Starch Press.
15) Zelle, J. M. (2004). Python programming: an introduction to computer science. Franklin, Beedle & Associates, Inc..

Name of the Paper: .....
Course Code: .....

Centre for Internal Quality Assurance (CIQA)
Krishna Kanta Handiqui State Open University
City Office: Housefed Complex, Guwahati-781006

Learner's Feedback on Course

Dear Learner,

Regarding the course as mentioned above, we would like to know your opinions and comments so as to improve the quality of self learning materials in future. Please respond to the following statements by ticking the number you feel most reflect your opinion. After completion of the additional comments, please detach the page and send/mail the same to us at the address given below.

The Director, Centre for Internal Quality Assurance, KKHandiqui State Open University
Housefed Complex, Dispur, Guwahati-781006
(E-mail id: ciqa@kkhsou.in)

1) Approximately how many hours did you spend for studying the units in the course? ☐☐☐

2) Please give your opinions (by ✓ mark) to the following items based on your reading of the block:

Sl. No.	Statements	Strongly Agree	Agree	Neither Agree nor Disagree	Disagree	Strongly Disagree
I)	The SLMs of the course fulfil the learning objectives set out in the beginning of each unit					
II)	The units of the SLM could satisfy my academic needs and expectations					
III)	The Unit writers have excellent knowledge about the course contents					
IV)	Language and contents of the units were presented at a level which I could readily understand					
V)	Ample opportunity for participation in the activities provided in the units					
VI)	Used enough Illustrations (Diagrams, tables etc.) for conceptual clarity					
VII)	Quality of content is engaging, relevant, and up-to-date					
VIII)	The self check questions are very helpful					
IX)	The Possible/model questions and the answers to check my progress have benefited me a lot					

Additional Comments: *(Please feel free to provide your open comments)*

1) Which aspects of the SLM, according to you, worked well?

.....

.....

2) What sort of changes/improvements do you feel KKHSOU could implement to improve the overall quality of the SLM?

.....

.....

Thank you for taking the time to complete this form.

******

(To be inserted as last page after the end of the last unit of the course)