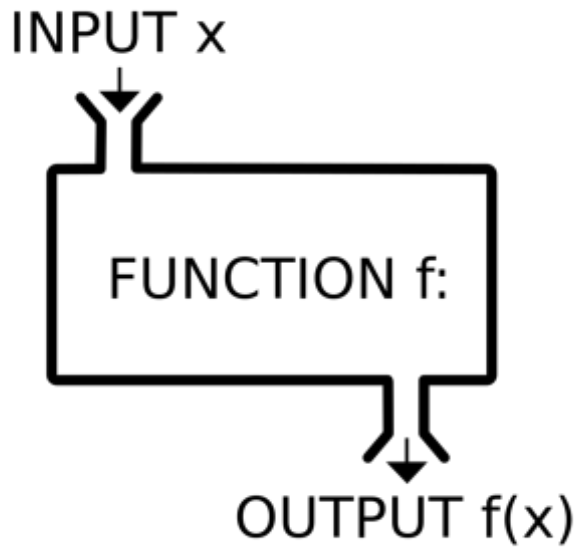
UNIT 2

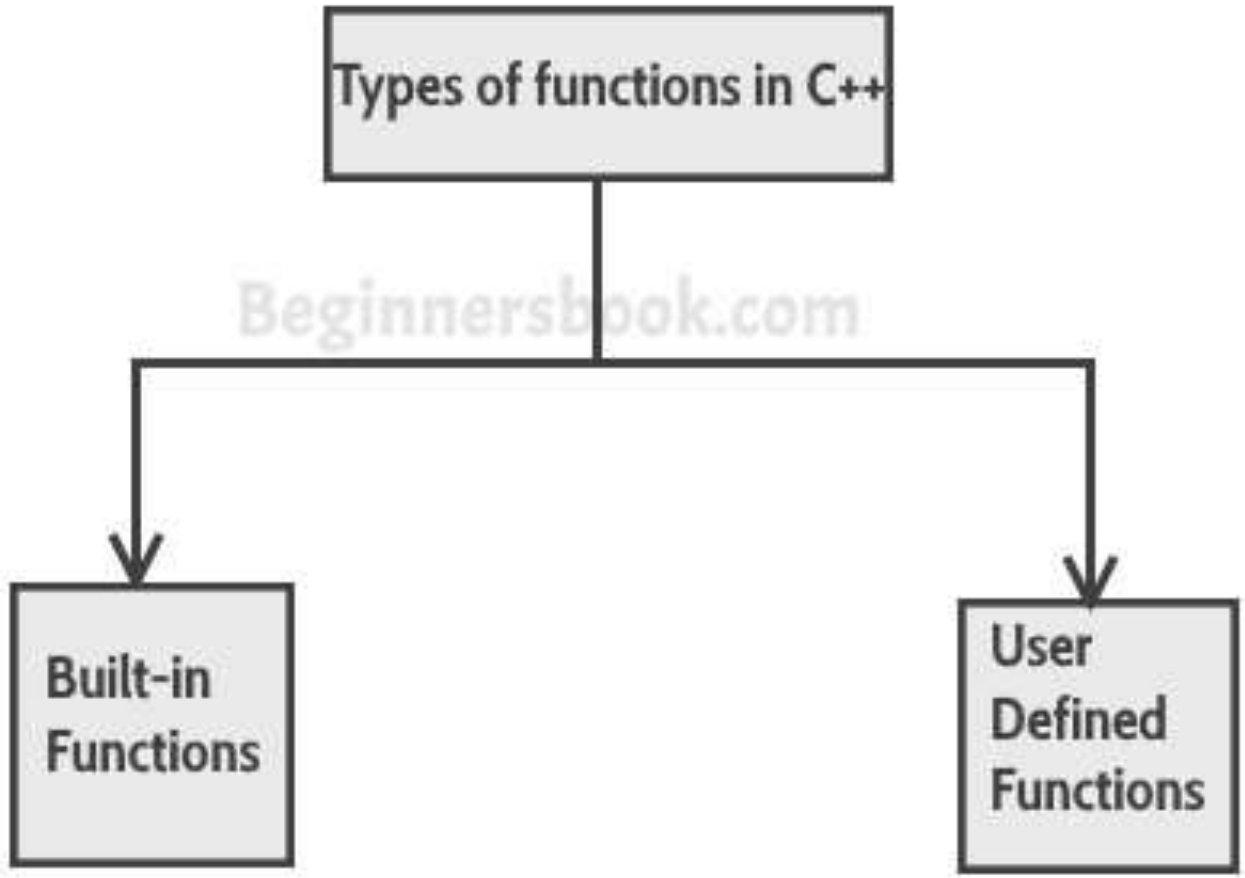
Functions

- Functions are the basic building blocks of any non-trivial program.
- Sequence of instructions repeated again and again
- The first line must look like this:
- function result = thisfunc(arg1, arg2)

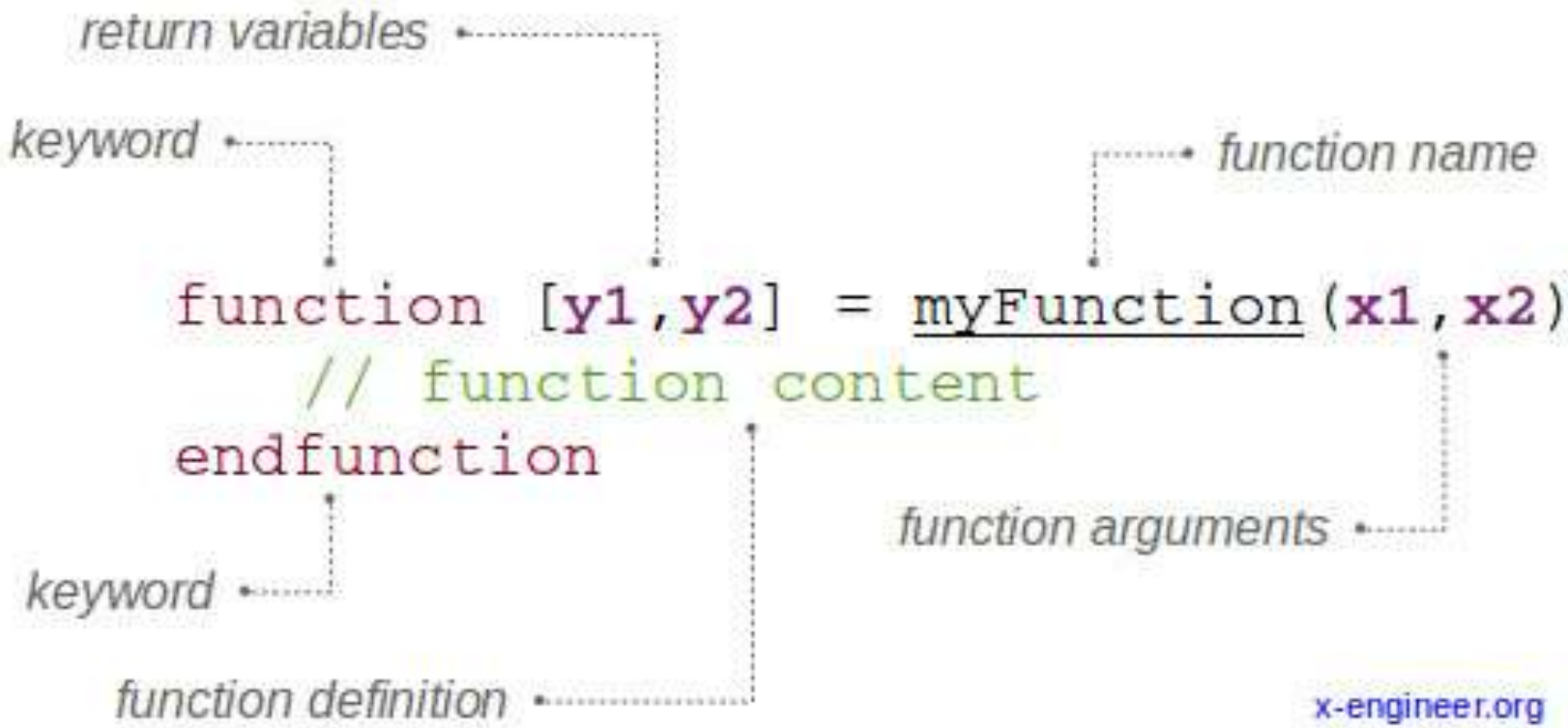

INPUT x
FUNCTION f:
OUTPUT f(x)

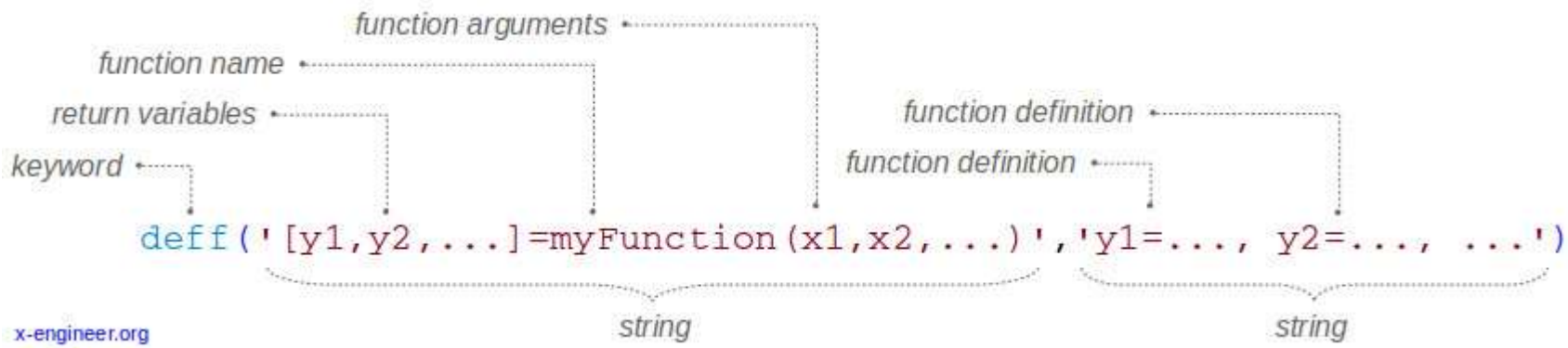
Features of Functions

- Functions can be defined online or offline
- Arguments can be any Scilab objects
- More than one output arguments
- Input/output arguments can be functions
- Functions can be nested


Types of functions in C++
Beginnersbook.com
Built-in Functions
User Defined Functions

Function Syntax


return variables
keyword
function name
function [y1,y2] = myFunction(x1,x2)
// function content
endfunction
function arguments
keyword
function definition
x-engineer.org


function arguments
function name
return variables
keyword
function definition
function definition
deff('[y1,y2,...]=myFunction(x1,x2,...)','y1=..., y2=..., ...')
string
string
x-engineer.org

Function with one output Parameter

```
function a=myfunc(x, y)
    a=x*y;
 endfunction
```

Conditioning

- conditional statement we understand the execution of a statement (instruction) only if a condition or a set of conditions is true.

Examples of conditional statements:

- if a person’s age is below 18 years, it’s forbidden to drive a car
- if the air temperature is above 30 °C, the weather is very hot

IF ELSE conditional statements

```
if condition_1 then
    instructions_1
else
    defaultInstructions
end
```

```
currentHour = 14;
if (currentHour > 7) & (currentHour < 20) then
   outsideIs = 'DAY';
else
   outsideIs = 'NOT DAY';
end
disp(currentHour)
disp(outsideIs)
```

IF ELSEIF conditional statements

```
if condition_1 then
    instructions_1
elseif condition_2 then
    instructions_2
...
elseif condition_N then
    instructions_N
else
    defaultInstructions
end
```

Program

- 8:00 – 10:00 Project Meetings
- 10:00 – 12:00 Training
- 12:00 – 12:30 Lunch Break
- 12:30 – 16:00 Project Development
- 16:00 – 8:00 Free Time

```
currentHour = 9;
if (currentHour > 8) & (currentHour <= 10) then
  disp("Activity: Project Meetings")
elseif (currentHour > 10) & (currentHour <= 12) then
  disp("Activity: Training")
elseif (currentHour > 12) & (currentHour <= 12.3) then
  disp("Activity: Lunch break")
elseif (currentHour > 12.3) & (currentHour <= 16) then
  disp("Activity: Project Development")
 else
  disp("Activity: Free Time!")
 end
```

Select case

- If the control variable has multiple states, for a better structure and readability of the code, it is better to use SELECT CASE instead of IF ELSEIF conditional statements.

```
select controlVariable
    case value_1 then instructions_1
    case value_2 then instructions_2
     ...
    case value_N then instructions_N
        else defaultInstructions
end
```

```
sHex = 'A'
select sHex
   case 'A'
            disp(10)
   case 'B'
            disp(11)
   case 'C'
            disp(12)
   case 'D'
            disp(13)
   case 'E'
            disp(14)
   case 'F'
            disp(15)
   else
            disp('Not a HEX alpha numeric character!')
end
```

Case with then

```
sHex = 'A'
select sHex
   case 'A' then disp(10)
   case 'B' then disp(11)
   case 'C' then disp(12)
   case 'D' then disp(13)
   case 'E' then disp(14)
   case 'F' then disp(15)
else disp('Not a HEX alpha numeric character!')
 end
```

Case with comma

```
sHex = 'A'
select sHex
   case 'A', disp(10)
   case 'B', disp(11)
    case 'C', disp(12)
   case 'D', disp(13)
   case 'E', disp(14)
   case 'F', disp(15)
else disp('Not a HEX alpha numeric character!')
end
```

Case in single line

```
sHex = 'A'
select sHex
   case 'A' then disp(10), case 'B' then disp(11)
else
   disp('Not a HEX alpha numeric character!')
end
```

Case with different options

```
select 1.1
  case 1.1
        disp('ok')
  else
        disp('nok')
  end
```

```
select [1 2]
        case [1 2]
        disp('ok')
Else
        disp('nok')
end
```

Functions with condition

```
function a=f(x, var);
  [lhs,rhs]=argn(0);
  if rhs==2 then
        a=x+var;
  end if rhs==1 then
        a=x*2;
  end
endfunction
```

LOOPS

- To learn conditional and unconditional flow control structures, in MATLAB®
- Looping control structures
  - `for` loop and `nested for` loops
  - `while` loop
- Branching control structures
  - `If` control structure and its different forms, i.e., `if-end`, `if-else-end`, `if-elseif-else-end` and `nested if` structures.
  - `switch-case` structure
- Special MATLAB® statements which control program execution sequence
  - `break` statement
  - `continue` statement
  - `error` statement
- `try-catch` structure to control the way MATLAB responds to errors.

for Loop

for Loop

The for loop control structure is used for unconditional looping. In this control structure, a group of statements, say '*n*' in number, are executed repeatedly for a fixed specified number of times, which is known before the loop starts and is decided by an **index**. The syntax of for statement is given as follows:

```
for index = initial value : increment : last value
   statement 1;
   statement 2;
   .
   .
   .
   statement n;
end;
next statement
```

The index is assigned an initial value and then a group of statements is executed till end statement. After completion of one loop, the index is incremented by increment that is specified and then the group of statements is executed again, provided the new value of index does not exceed the last value. The process is repeated till the index exceeds the last value. After this, for loop is terminated and the control is passed to the statement next to the end statement corresponding to a particular for loop.

In more generalized form, the for loop syntax is expressed as follows:

```
for index = expression,
   statement 1;
   statement 2;
   :
   :
   statement n;
end;
```

where index is the loop variable and expression is the loop control expression.

Example 7.1

Write a MATLAB Program to obtain the sum of all even numbers from 0 to 20, using `for` loop statement.

Solution:

The following commands are used:

```
sum = 0;                          %Initializing variable 'sum'
for k = 0: 2 : 20                 %start of 'for' loop, k is the index
  sum = sum + k;                  %add 'sum' to 'k'
end;                              %end of 'for' loop
```

Here, index value k is incremented each time by the increment value 2 till it exceeds 20. The value of index k is added to the variable sum in each pass. When k exceeds its last value, i.e. 20, the program control terminates the `for` loop statement and exits the loop.

The result is obtained by entering the variable sum in the command window and is as follows:

```
sum =

     110
```

PROGRAMMING TIP 7.1

When the increment value of the index is not mentioned, it is taken as 1 by default.

Example 7.2

Write a MATLAB Program to obtain the sum of all integers from 0 to 20, using `for` loop statement, where the increment value is 1 only.

Solution:

The commands used are as follows:

```
sum = 0;                          %Initializing variable 'sum'
for j = 0:20                      %start of for loop,j is index-count
  sum = sum + j;                  %add 'sum' to 'j'
end;                              %end of for loop
sum
```

This example is similar to Example 7.1 except that the increment value for index `j` is not given in the `for` statement. Here, the increment value of the index is not mentioned and, thus, is taken as 1 by default.

The result is obtained by entering the variable sum in the command window and is as follows:

```
sum =
      210
```

Example 7.3

Program to illustrate the use of `for` loop statement for discrete arbitrary integer values of the index, given as a vector.

Solution:

Let us assume that the index values are: [1 5 7 6]
The program is as follows:

```
sum = 0;
for k = [1 5 7 6]                    %The index 'k' and its values
   sum = sum + k;
end;
```

In this example, the values of the index are given as (1 × 4) array. As k has only four values, the loop will be executed four times only, with value of index k assigned as 1 in first pass, 5 in second pass, 7 in third pass and 6 in the last pass.

The result obtained is

```
sum =

    19
```

Example 7.4

Program to illustrate the use of `for` loop statement that includes a decrement value.

Solution:

The program is as follows:

```
result = 0;
for k = 10 : -2 : 0
  result = result + k;
end;
```

In this example, the index count k is initialized to 10 and decremented by 2 in each iteration till 0 is reached in the last iteration. The decrement value is indicated by the negative increment in the `for` statement.

The result obtained is

```
result =
          30
```

PROGRAMMING TIP 7.2

In `for` loop, it is possible to assign negative increment, i.e. there can be decrement also. In such a case, the last value becomes the minimum value for the index variable.

Example 7.5

Write a program to find the average of any given 10 numbers using `for` statement.
Check the program with following values: 3, 5, 24, 5, 6, 4, 10, 23, 45, 2

Solution:

The program is as follows:

```
%This program illustrates the use of 'for' loop for finding
%average of given 10 numbers.
%i is index value
%sum, avg. are sum and average variables respectively
sum = 0;
for i =1:10
   n = input('Enter the value:');  %feed the values of ten numbers one by one
   sum = sum + n;                  %Find sum
end;
avg = sum/10;                      %Calculate average of numbers
%display the result in floating point format.
fprintf ('The average is :%f ', avg);
```

On executing the above program, the response obtained in the command window is as follows:

```
Enter the value: 3
Enter the value: 5
Enter the value: 24
Enter the value: 5
Enter the value: 6
```

```
Enter the value: 4
Enter the value: 10
Enter the value: 23
Enter the value: 45
Enter the value: 2
The average is: 12.700000
```

Nested for Loop

nested for Loop

When one `for` loop is embedded in another `for` loop, the two loops are called nested `for` loops. The necessary condition is that one loop is completely nested within the other.

Example 7.6

Program to illustrate the use of `nested for` loop structure.

Solution:

The program is as follows:

```
%Program to illustrate the use of nested for loops.
for a = 0 : 2 :4,                    %'a' is index for outer loop
 b=0;
 value=0;
 for b = 1 : 3,                      %'b' is index for inner loop
  value = value+ a + b;              %expression in inner loop
  fprintf('Inner loop:\n');
  fprintf('a = %d \t b = %d \n',a,b);
end;
  fprintf('The outer loop\n');
  fprintf('a = %d \t b = %d\t value = %d\n\n',a,b,value );
end;
```

The result obtained is as follows:

```
Inner loop:
a = 0       b = 1
Inner loop:
a = 0       b = 2
Inner loop:
a = 0       b = 3
The outer loop
a = 0       b = 3          value = 6
Inner loop:
a = 2       b = 1
Inner loop:
a = 2       b = 2
Inner loop:
a = 2       b = 3
The outer loop
a = 2       b = 3          value = 12
Inner loop:
a = 4       b = 1
```

```
Inner loop:
a = 4         b = 2
Inner loop:
a = 4         b = 3
The outer loop
a = 4         b = 3          value = 18
```

PROGRAMMING TIP 7.5

The nested `for` loops should use independent loop index variables.

while Loop

while Loop

A while loop is used for executing a group of statements repeatedly on the basis of a condition. In this control structure, the group of statements associated with the while statement are executed repeatedly or indefinitely until the given condition is true (or non-zero). The general form of while loop is

```
while <condition>,
  statement 1;
  statement 2;
  :
  :
  statement n;
end;
```

The group of statements may include some statements which may alter the result of the condition. When the condition becomes false ('0'), the program will exit from the loop and control will be transferred to the next statement which appears after the end statement in the program.

Example 7.7

Write a program, using `while` loop, for finding squares of integers less than 5.

Solution:

The program is as follows:

```
%Program to find squares of integers less than 5 using while loop
k = 0;                  %Initialize variable
while k < 5,
  k = k + 1;
  ksq = k ^ 2;
  fprintf ('Square of %d is %d\n', k, ksq);
end;
```

The result obtained is as follows:

```
Square of 1 is 1
Square of 2 is 4
Square of 3 is 9
Square of 4 is 16
Square of 5 is 25
```

This example finds squares of integers less than 5. First, value of variable `k` is initialized to 0. Then the condition ($k < 5$) is checked, and if the condition is found `true`, the statements in the `while` loop are executed. The loop is executed again and again till condition $k < 5$ is `true`. Once it becomes `false`, the control will come out of the `while` loop.

Example 7.8

Write a program to add the first 10 digits using `while` loop statement.

Solution:

The program is given as follows:

```
%Program to add first 10 digits using while loop
n = 0;
sum = 0;                          %initialize the variables
while n < 11                      %start the while loop
     sum = sum + n;               %find sum of the numbers
     n = n + 1;
end;
fprintf ('The sum of first ten numbers is \t %d \n', sum);
```

The result obtained is

```
The sum of first ten numbers is 55
```

Example 7.9

Write a program to add the given numbers till the number entered is 0, using `while` loop. Check the program using data given as: 34, 45, 6, 5, 6, 5, 60, 0, 8, 7

Solution:

The program is given as follows:

```
%Program to add given numbers till the number entered is 0
sum = 0;                %Initialize the variable sum to zero get the first
value
num = input('Enter the first number ');
fprintf('Enter 0 to end the summation otherwise \n')
while (num ~= 0),
   sum = sum + num;              %add the value
   num = input('Enter the next number ');
end;
fprintf('The sum of numbers till '0' is entered is\t%d\n', sum);
```

The result obtained is

```
Enter the first number 34
Enter 0 to end the summation otherwise
Enter the next number 45
Enter the next number 6
Enter the next number 5
Enter the next number 6
Enter the next number 5
Enter the next number 60
Enter the next number 0
The sum of numbers till '0' is entered is 161
```

It may be noted that the numbers 8 and 7, which appears after 0, cannot be fed in the program.

Branches Control Structures

The branch control structures allow selecting and executing specific blocks of statements out of many statement blocks depending upon the condition. These include different types of `if` structures, `switch` control structure and so on. The `if` structure is a conditional control structure which allows some group of statements to be executed only if the given condition is `true`.

if Control Structures

`if` Control Structures

In `if` control structure, a group of statements are executed only if the condition is `true`. The following are the different forms of `if` control structures:

(a) `if` structure
(b) `if-else` structure
(c) `if-elseif-else` structure
(d) `nested if` structure

if Structure

if Structure

This is the simplest form of `if` control structure. It has only one group of statements which is executed if the condition is `true`, otherwise the program exits the `if` control structure. The syntax of `if` structure is given as follows:

```
if expression
  statement 1  ┐
  statement 2  │
  .            │ group of statements
  .            │
  .            ┘
end
next statement
```

where expression is a logical expression and gives the result either `true` or `false` (non-zero or 0).

If the expression returns `true` result, then the group of statements after `if` statement is executed and the control goes to the next statement after the `end` statement. If the expression is `false`, then this group of statements is skipped and control goes to next statement which appears after the `end` statement.

if-else Structure

if-else Structure

In this form, `if` control structure has two groups of statements, i.e. one for `true` and the other for `false` condition. The syntax is given as follows:

```
if expression
   statement 1
   statement 2
   .                  group1 of statements
   .
   .
else
   statement 1
   statement 2
   .                  group2 of statements
   .
   .
end
next statement
```

If the expression is `true`, then group1 statements are executed, group2 statements are skipped, and the program control shifts to next executable statement after `end` statement.

If the expression is `false`, then group1 statements are skipped, the control goes to the statements following `else` statement, i.e., group2 statements are executed and the program control shifts to next executable statement after the `end` statement.

if-elseif-else Structure

7.3.1.3 if-elseif-else Structure

This is the most useful form of `if` control structure. It provides a user the option to check a large number of different conditions. The syntax is given as follows:

```
if expression1
  statement 1
  statement 2
  .                 group1 of statements
  .
  .
elseif expression2
  statement 1
  statement 2       group2 of statements
  .
else
  statement 1
  statement 2       group3 of statements
  .
  .
end
next statement
```

If expression1 is `true`, the group1 statements are executed and other groups of statements are skipped, and the control shifts to next executable statement after the `end` statement.

If expression1 is `false`, group1 statements are skipped and expression2 is checked. If it is `true`, group2 statements are executed and other statement group i.e., group3 is skipped, and the program control shifts to next executable statement after the `end` statement.

In case expression2 is `false`, then the group3 statements are executed and control shifts to the next executable statement after the `end` statement.

nested if structure

nested if Structure

Similar to other control structures, `if` structure can also be nested. `If` structures are called `nested`, when one of the `if` structure lies entirely within the domain of the other `if` structure. The syntax is given as follows:

```
if (condition1)
  statement 1
  statement 2

  if (condition2)
        statement 1
        statement 2
    end
    statement 3
    statement 4
end
```

PROGRAMMING TIP 7.6

There can be any number of `nested if` structures. The different `end` statements are associated with the respective closest `if` statement. It is important to take care that for each `if` structure there should be corresponding `end` statement.

PROGRAMMING TIP 7.7

The multiple `elseif` control structure is preferred over `nested if` control structure to avoid syntax error.

Example 7.10

Write a program to check whether a given number is greater than or equal to 50, using `if-elseif-else` statement.

Check the program with the given numbers as: (i) 23 (ii) 50

Solution:

The program is as follows:

```
%Program to check whether the number is greater, equal or smaller than 50.
disp('To check whether the given number is greater, equal or smaller than
50');
a = input ('Enter a number: ');
if a > 50,
  disp('The number is greater than 50');
elseif (a==50)
  disp('The number is equal to 50');
else
  disp('The number is smaller than 50');
end;
```

(i) When 23 is taken as a given number. The result obtained is as follows:

```
To check whether the given number is greater equal or smaller than 50
Enter a number: 23
The number is smaller than 50
```

(ii) When 50 is taken as a given number. The result obtained is as follows:

```
To check whether the given number is greater than 50 or not
Enter a number: 50
The number is equal to 50
```

Example 7.11

Write a program to evaluate $y = 2x + 1$ if $x < 5$, otherwise $y = 2x$. Use the `if-else` statement. Check the program with value of x as: (i) 7 and (ii) 2.

Solution:

The program is given below:

```
%Program to illustrate 'if-else' structure
x = input('Enter the value of x: ');
if x < 5,
     y = 2*x + 1;     %executed if condition is 'true'
else
     y = 2*x;         %executed if condition is 'not true'
end;
fprintf('The value of x is %d\n' , x);
fprintf('The corresponding value of y is %d\n',y)
```

(i) When $x = 7$. The result obtained is as follows:

```
Enter the value of x: 7
The value of x is 7
The corresponding value of y is 14
```

(ii) When $x = 2$. The following result is obtained:

```
Enter the value of x: 2
The value of x is 2
The corresponding value of y is 5
```

Example 7.12

Write a program to find the largest of the three numbers using `if-elseif-else` structure. Check the program the given values as: (i) 45, 98, 34 (ii) 45, 34, 8

Solution:

The program is given below:

```
%This program finds largest of three numbers
x1 = input('Enter first number: ');
x2 = input('Enter second number: ');
x3 = input('Enter third number: ');
if (x1 > x2),                              %the outer 'if'
  if (x1 > x3),
     fprintf('The largest number is %d\n', x1); %the inner 'if'
  end;                                     %end of inner 'if' structure
elseif (x2 > x3),
  fprintf ('The largest number is %d\n', x2);
else
  fprintf('The largest number is %d\n', x3);
end;
```

(i) When the given numbers are: 45, 98, 34.

The result obtained is:

```
Enter first number: 45
Enter second number: 98
Enter third number: 34
The largest number is 98
```

(ii) When the given numbers are: 45, 34, 8. The following result is obtained:

```
Enter first number: 45
Enter second number: 34
Enter third number: 8
The largest number is 45
```

switch Statement

switch Statement

The switch-case-otherwise structure is used for multi-way branching. It allows selection of a particular block of statements from several available blocks. A flag (any variable or expression) is used as a switch and based on the current value of the flag, particular block of statements is selected. This structure is used for logical branching. A flag can be a single integer, character or expression.

The general syntax of switch control structure is

```
switch flag,
case value1,
   block1 statements;
-------
case value2,
   block2 statements;
----------
otherwise,
   last block statements;
end;
next statement
```

Here, depending upon the value of the flag, the particular block of statement is executed. If flag = value1, block1 statements are executed and the program control will exit the switch structure; if flag = value2, block2 is executed and the program control will exit the switch structure; if the flag does not match any case value then the last block associated with otherwise is executed.

The matching end statement indicates the end of the switch structure.

In some cases, it is desirable to execute the same block of statements for a set of values of flag variable. Then `switch-case` control structure is used with modification. The modified syntax is:

```
switch flag,
  case (value1 value2 value3)
     block1    statements;
  -------
  case (value4 value5 value6)
     block2    statements;
  ----------
otherwise,
     last block statements;
end;
```

Example 7.13

Write a program, using `switch` structure involving string variables, to identify the profession of a person if its code is given, such as D for Doctor, E for Engineer, etc.

Solution:

The program is given below:

```
%This program tells the profession of a person using switch case structure.
code = input ('Choose the profession code (D for...
Doctor,E for Engineer, M for Manager,P for Professor): \n', 's');
%Get the value of flag in a variable (code)
%variable name (code) is used as flag of switch structure
fprintf('Selected profession is :\n')
switch code
        case 'D'
  fprintf ('Doctor \n');
        case 'E'
  fprintf ('Engineer \n');
        case 'M'
  fprintf ('Manager \n');
        case 'P'
  fprintf ('Professor \n');
otherwise
  fprintf ('Invalid choice of profession \n');
end;
```

The response obtained is

```
Choose the profession code (D for Doctor,E for Engineer, M
for Manager,P for Professor):
M
Selected profession is :
Manager
```

Example 7.14

Write a program, using `switch` structure, to make a choice of colour from given colour codes i.e., R, Y, G, W for Red, Yellow, Green and White respectively.

Solution:

The program is given below:

```
%Program to make a choice of color.
color = input('Select the color code (R,Y,G,W): ' ,'s');
```

```
switch color,
  case 'R',
  fprintf('Color selected is Red \n');
  case 'Y',
  fprintf ('Color selected is Yellow\n');
  case 'G',
  fprintf ('Color selected is Green \n');
  case 'W',
  fprintf ('Color selected is White\n ');
  otherwise,
  disp('Invalid entry, select amongst the given codes');
end;
```

The result obtained is as follows:

```
Select the color code (R,Y,G,W): Y
Color selected is Yellow
```

In case wrong entry is given, the response obtained is

```
Select the color code (R,Y,G,W): t
Invalid entry, select amongst the given codes
```

Example 7.15

Write a program to tell the day of the week corresponding to given day number. Assume Monday is the first day of the week. Check the program with values given as: (i) 8 (ii) 5

Solution:

The program is given as follows:

```
%This program tells the day of the week corresponding to given day number.
day = input('Enter the no. of the day of the week(1-7) : ','s');
switch day,
  case '1',     %day is used as flag variable
        fprintf ('Day selected is Monday\n');
  case '2',
        fprintf ('Day selected is Tuesday\n');
  case '3',
        fprintf ('Day selected is Wednesday\n');
  case '4',
        fprintf ('Day selected is Thursday\n');
  case '5',
        fprintf ('Day selected is Friday\n');
  case '6',
        fprintf ('Day selected is Saturday\n');
  case '7',
        fprintf ('Day selected is Sunday\n');
  otherwise
        disp('Day no. selected is not valid ')
end;
```

(i) When the given value is 8. The result obtained is

```
Enter the no. of the day of the week(1-7) : 8
Day no. selected is not valid
```

(ii) Executing the program again with given value 5, the result obtained is as follows:

```
Enter the no. of the day of the week(1-7) : 5
Day selected is Friday
```

Example 7.16

Write a program to check whether a given number is a multiple of 2 and 3 from first 15 numbers using `switch` statement.

Check the program with given number as: (i) 11 (ii) 6

Solution:

The program is as given below:

```
%Program to find multiple of 2 and 3 from first 15 numbers
%using switch-case statement'
number = input('Enter a number between 1 and 15: ', );
switch number,
  case {6 12 }
        fprintf ('The selected no. is multiple of 2 and 3\n');
  case { 2 4 6 8 10 12 14},
        fprintf ('The selected no. is multiple of 2\n');
  case {3 6 9 12 15}
        fprintf ('The selected no. is multiple of 3\n')
  otherwise
        fprintf('The number selected is neither multiple of 2 nor of 3\n')
end;
```

(i) When the given number is 11. The result obtained is as follows:

```
Enter a number between 1 and 15: 11
The number selected is neither multiple of 2 nor of 3
```

(ii) When the given number is 6. The result obtained is as follows:

```
Enter a number between 1 and 15: 6
The selected no. is multiple of 2 and 3
```

break Statement

break Statement

The break statement inside a loop is used to terminate the execution of the loop and transfer the control to the next statement after the end statement of the loop. The control is shifted even if the condition for execution of the loop is true.

Example 7.17

Write a program to add the first 10 numbers but stop the program when the sum exceeds 20 using `break` statement.

Solution:

The program is given below:

```
%Program to illustrate the use of break statement
%Add the first 10 numbers but stop the loop when the sum exceeds 20.
sum = 0;
```

```
    for k = 1 : 10,
        sum = sum + k;
        if (sum>20)
       'break;       %Use the break statement to terminate the 'for' loop.
        end;
        fprintf( 'sum = %d\n', sum);
    end;
    disp('The program stops in-between as the sum becomes more than 20');
```

The result obtained is

```
    sum = 1
    sum = 3
    sum = 6
    sum = 10
    sum = 15
    The program stops in-between as the sum becomes more than 20
```

Here, when the program is executed, the sum is calculated till its value remains less than 20, but when it exceeds 20, `break` statement is executed and message is displayed as follows:

```
    The program stops in-between as the sum becomes more than 20
```

Example 7.18

Illustrate the use of break statement inside the nested for loop.

Solution:

The program is as follows:

```
%Program illustrates use of break statement in the 'nested for'
%loop. In this program product and sum of two variables
%is obtained and product is displayed if its value is less than 5.
  prod=0;          %initialize the variables
  sum=0;
  for i=1:3
       fprintf('The value of I is  %d\n',i);
       for j=1:3
          fprintf('The value of J is  %d\t',j);
          prod= i*j;
          if(prod>5)
             fprintf('Product is >5 so not displayed');
             break;      %break command inside inner 'for' loop
          end;
          fprintf ('prod = %d\n', prod);
       end;
       sum= sum+i+j;
       fprintf('\nsum=%d\t\n',sum);
end;
```

The response obtained is as follows:

```
The value of I is 1
The value of j is   1 prod = 1
The value of j is   2 prod = 2
The value of j is   3 prod = 3
sum=4
The value of I is 2
The value of j is   1 prod = 2
The value of j is   2 prod = 4
The value of j is   3 Product is >5 so not displayed.
sum=9
The value of I is 3
The value of j is   1 prod = 3
The value of j is   2 Product is >5 so not displayed
sum=14
```

continue Statement

continue Statement

The `continue` command is used to skip the statements in the current pass in loop but will not exit from the loop. The remaining iterations are continued for remaining values of the index or decision variable.

PROGRAMMING TIP 7.12

The `continue` statement passes control to the next iteration of `for` or `while` loop containing it.

Example 7.19

Write a program to display the value of variable k from 1 to 10 but skip the values $2 < k < 7$ using `continue` statement.

Solution:

The program is given as follows:

```
%Program to illustrate the use of continue statement
%In this program value of k is shown from 1 to 10 but skipped for 2<k<7.
for k=1:10,
  if ((k > 2) & (k < 7 )),
     continue;     % control is shifted to the beginning of the 'for' loop
end;
  fprintf ('k=%d\n', k);
end;
```

The result obtained is

```
k=1
k=2
k=7
k=8
k=9
k=10
```

error Statement

error Statement

Sometimes, on getting incorrect input value the program needs to be terminated and it is desired to display a message. This can be done using `error` command. The syntax is given as

```
error('message to be displayed on encountering error')
```

Example 7.20

Write a program to illustrate the use of `error` statement when an error is encountered during the addition of two row vectors. Check the program for following data:

(i) A = [2 3], B = [23 45]

(ii) A = [12 37], B= [23 2 56 7]

Solution:

The program is given as follows:

```
%Program to illustrate the use of error command.
%In this program two user defined row vectors are added and
%error is generated in case of wrong entry.
fprintf('SUMMATION OF TWO ROW VECTORS\n');
a = input('Enter first row vector in square bracket ');
b = input('Enter second row vector in square bracket ');
if length (a) == length (b),
  sum =a+b
else
  error('Both row vectors should be of same length');
end;
```

(i) When given data is : A = [2 3], B = [23 45]
The result obtained is given as:

```
SUMMATION OF TWO ROW VECTORS
Enter first row vector in square bracket [2 3]
Enter second row vector in square bracket [23 45]
sum =
  25    48
```

(ii) When given data is : A = [12 37], B = [23 2 56 7]
The result obtained is given as:

```
SUMMATION OF TWO ROW VECTORS
Enter first row vector in square bracket  [12 37]
Enter second row vector in square bracket [23 2 56 7]
??? Error using ==> ex721
Both row vectors should be of same length
Error in ==> ex721 at 8
  error('Both row vectors should be of same length');
```

Example 7.21

Write a program to find the largest of given '*n*' numbers using `for` loop and `if` structure. Check the program with data given as: 23, 46, 78, 32 and 56.

Solution:

The program is given below:

```
%Program to find largest of n given numbers
largest =0;              %Initialize a variable
n = input('Enter the count of numbers ');
for i =1: n,
    num(i) = input('Enter the number ');
end;
for i =1:n,
    if (largest < (num (i)))
          largest= num(i);
    end;
end;
    fprintf ('The largest number is: %d\n', largest);
```

The result obtained is

```
Enter the count of numbers 5
Enter the number 23
Enter the number 46
Enter the number 78
Enter the number 32
Enter the number 56
The largest number is: 78
```

Example 7.22

Write a program to find the number of values lying between 50 and 100 from a given data of '*n*' numbers.
Use `for` loop and `if` structure.
Check the program with data given as: 65, 45, 32, 76 and 58.

Solution:

The program is given as follows:

```
%Program to illustrate the use of 'for' loop and 'if' structure
%Program to find number of values lying between 50 and 100
%from a given n numbers,
n = input('Enter the count. of numbers ');
for i =1: n,
    num(i) = input ('Enter the number ');
end;
k=0;
for i =1: n,
    a = num(i);
    if (a >= 50),
        if(a < 100),
            k =k + 1;
        end;
    end;
end;
fprintf('The count of numbers >50 and <100 is %d\t\n', k);
```

The result obtained is

```
Enter the count. of numbers 5
Enter the number 65
Enter the number 45
Enter the number 32
Enter the number 76
Enter the number 58
The count of numbers >50 and <100 is 3
```

Example 7.23

Write a program to find the count of even values in given 'n' numbers. Use `for` loop and `if` structure. Check the program with data given as: 23, 42, 54, 98 and 78 and $n = 5$.

Solution:

The program is given as follows:

```
%Program to find number of even values in given n numbers
n =input ('Enter the total numbers ');
for i = 1: n,
  num(i) = input('Enter the value ');
end;
total=0;
for i = 1 : n,
      if(rem(num(i),2) == 0)
         total = total + 1;
      end;
end;
fprintf('The count of even numbers in given data is %d\n', total);
```

The result obtained is

```
Enter the total numbers 5
Enter the value 23
Enter the value 42
Enter the value 54
Enter the value 98
Enter the value 78
The count of even numbers in given data is 4
```

try-catch Structure

try-catch Structure

Generally, while executing a program if an error occurs, MATLAB aborts the program showing the error message. However, by using `try-catch` structure, errors found by MATLAB are captured, giving the user ability to control the way MATLAB responds to errors. This allows the user to handle errors within the program without stopping it. The syntax for `try-catch` structure is given as follows:

```
try
    Statement 1
    Statement 2      group 1
    ---------
```

```
catch
    Statement 1
    Statement 2        group 2
    ---------
end
```

Here, firstly, the statements in `try` block are executed, then one of the following two cases is possible:

- If no error occurs, then statements in the `catch` block are skipped and program control shifts to statement after `try-catch` structure.
- In case an error occurs in any of the statements of `try` block, then the program stops executing the remaining statements in `try` block and executes the statements in the `catch` block and finally exits from the `try-catch` structure.

Example 7.24

Write a program to add two given row vectors. Illustrate the use of `try-catch` structure.
Check the program with following data:

(i) [3 6 8] and [12 34]

(ii) [2 4 8] and [5 89 34]

Solution:

The program is given as below:

```
%Program to add two row vectors, illustrate the use of try-catch structure
fprintf('SUMMATION OF TWO ROW VECTORS\n');
a = input('Enter first row vector in square bracket ');
b = input('Enter second row vector in square bracket ');
try
    sum =a+b
catch
    disp('ERROR: Both row vectors should be of same length');
end
```

(i) When the data is [3 6 8] and [12 34]. The result obtained is

```
SUMMATION OF TWO ROW VECTORS
Enter the first row vector in square bracket      [3 6 8]
Enter the second row vector in square bracket     [12 34]
ERROR: Both row vectors should be of same length
```

(ii) When the data is [2 4 8] and [5 89 34]. The result obtained is

```
SUMMATION OF TWO ROW VECTORS
Enter the first row vector in square bracket      [2 4 8]
Enter the second row vector in square bracket     [5 89 34]
sum =
   7   93   42
```

List of Commands

Control Structure/Command	Description
`for`	Repeat the statements specific number of times
`while`	Repeat the statements if the condition is true or non-zero
`if`	Conditionally execute the statements
`switch`	Switch between cases depending upon the condition
`break`	Terminate the execution of `for` or `while` loop
`continue`	Pass the control to next iteration of `for` or `while` loop
`error`	Display message and abort function
`try-catch`	Catch the error generated by MATLAB

Writing Programs and Functions

- Study of MATLAB® Editor and different features of Editor Menu bar and Toolbar.
- Learn MATLAB programming fundamentals and types of M-files, i.e., script files and function files.
- Learn how to write a function M-file.
- Learn about the concepts of argument passing, function workspace and types of functions.
- Use of Function Handle using `@` operator and `strfunc` function.

MATLAB EDITOR

MATLAB Editor not only contains basic text editing features but also provides the user with additional facilities such as debugging tools. The Editor/Debugger has proper graphical user interface for ease in programming. It also supports automatic indenting and highlighting to indicate special parts of the program such as keywords, text strings and comments. It also allows a user to observe the values of the variables/constants in the program. Features like setting/clearing breakpoints and executing programs line by line are also available to aid the programmer in effectively debugging the program.

MATLAB EDITOR

1.Opening the Editor.

2.Editor Main Menu.

3.Tool Bar.

Opening the Editor

The Editor can be opened in a number of ways, which are as follows:

1. Selecting `New` from the `File` menu in the MATLAB desktop main menu.
2. Clicking on the new file icon on the toolbar as shown in Figure 8.1.
3. Typing the command `edit` in the Command Window at the command prompt.

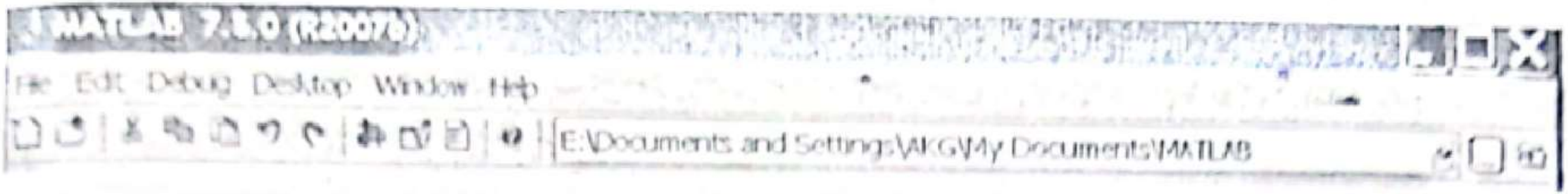


Figure 8.1 MATLAB Main Menu and Toolbar

If a file already exists and is to be modified, the Editor may be started by

1. Selecting open from the `File` menu in the MATLAB desktop main menu.
2. Clicking on the open file icon on the toolbar as shown in Figure 8.1.
3. Typing the command `edit` *`filename`* in the Command Window at the command prompt.

Editor Main Menu

The appearance of the Editor window is similar to normal windows present in Windows® XP operating system. It contains a Title Bar that displays the name of the file being edited. If the changes made in the file have not been saved, it is indicated by an `asterisk (*)` that appears after the name of the file in the Title Bar. The Editor window also contains a main menu and a toolbar. The main menu is composed of File menu, Edit menu, Text menu, Debug menu, Tools menu, etc. It also offers Help menu and choice to have multiple files opened for editing simultaneously. The Editor window is shown in Figure 8.2.

The **File menu** contains options to open, save and print files. It also contains choice to quit/exit from Editor/Debugger or MATLAB, and has choice for importing data from outside or saving the workspace in a *.mat file.

The **Edit menu** contains items that are found in normal text editor like `undo`, `redo`, `cut`, `copy`, `paste`, `find`, `replace text`, `select all`, `delete` and so on. It also has features for clearing the Command Window, Command History and Workspace.

The **Text menu** contains the item `Evaluate Selection` which is used to evaluate an expression and display the answer in the Command Window. There is provision for commenting or uncommenting the

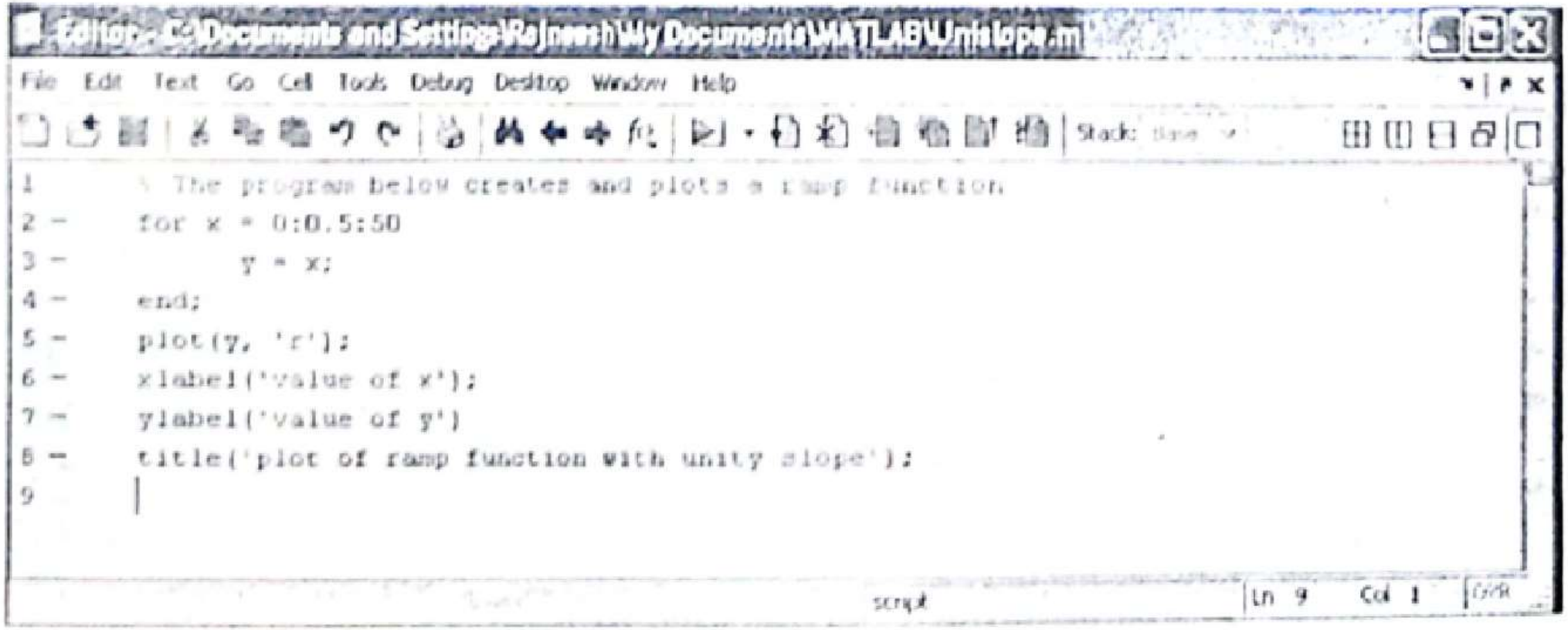


Figure 8.2 The Editor Window

selected text in the program file. This is very helpful when a large section of the program needs to be commented. It also contains option for decreasing or increasing the indent in the selected program text. Further, option for changing the text case is also available. Another very good feature that is included in the Text menu is `Code Folding`. When large programs are to be viewed, for convenience of the programmer the functions in the program are folded i.e. their statements are hidden and only the function definition line is displayed. This function is just like the Windows Explorer which hides the folder contents or displays them when desired by the user. The Editor window shown in Figure 8.3 illustrates `Code Folding` option where the code of the function `divs` is folded.

The **Go menu** includes options for moving `forward` or `back` in the program text. It also includes option for moving to a particular line in the text files that are open in the Editor. It opens a separate dialog box which prompts the user to enter the line number and select the program text file from the ones that are currently opened in the Editor. It also includes feature of `bookmark` to move to selected points in the program text.

Program files are often very long and contain multiple sections. Each section of the program can be described as a single M-file cell. The **Cell menu** deals with such sections i.e. cells of the program code.

The **Tools menu** contains option called `M-Lint`. `M-lint` displays inconsistencies and suspicious constructs in program files. This menu option can be utilized by the user to find such inconsistencies. Another tool available in the Editor tools menu is `Profiler`. This option opens Profiler window and identifies where a program file spends the most time, and therefore helps the user to identify the part of the code needed to be looked into for improving the performance of the code.

The **Debug menu** contains the various features embedded in the Editor for debugging the programs. It also contains option to run the program in the current text window.

The **Desktop menu** allows the user to dock or undock the Editor window and organize the desktop to the satisfaction of the user.

The **Window menu** provides choice of how the different text files in the Editor Window should appear on the desktop i.e. in tiled manner, or the two files viewed in horizontally or vertically split screen etc. It also allows the user to move to Command window, Command History window, Help window or Workspace window directly.

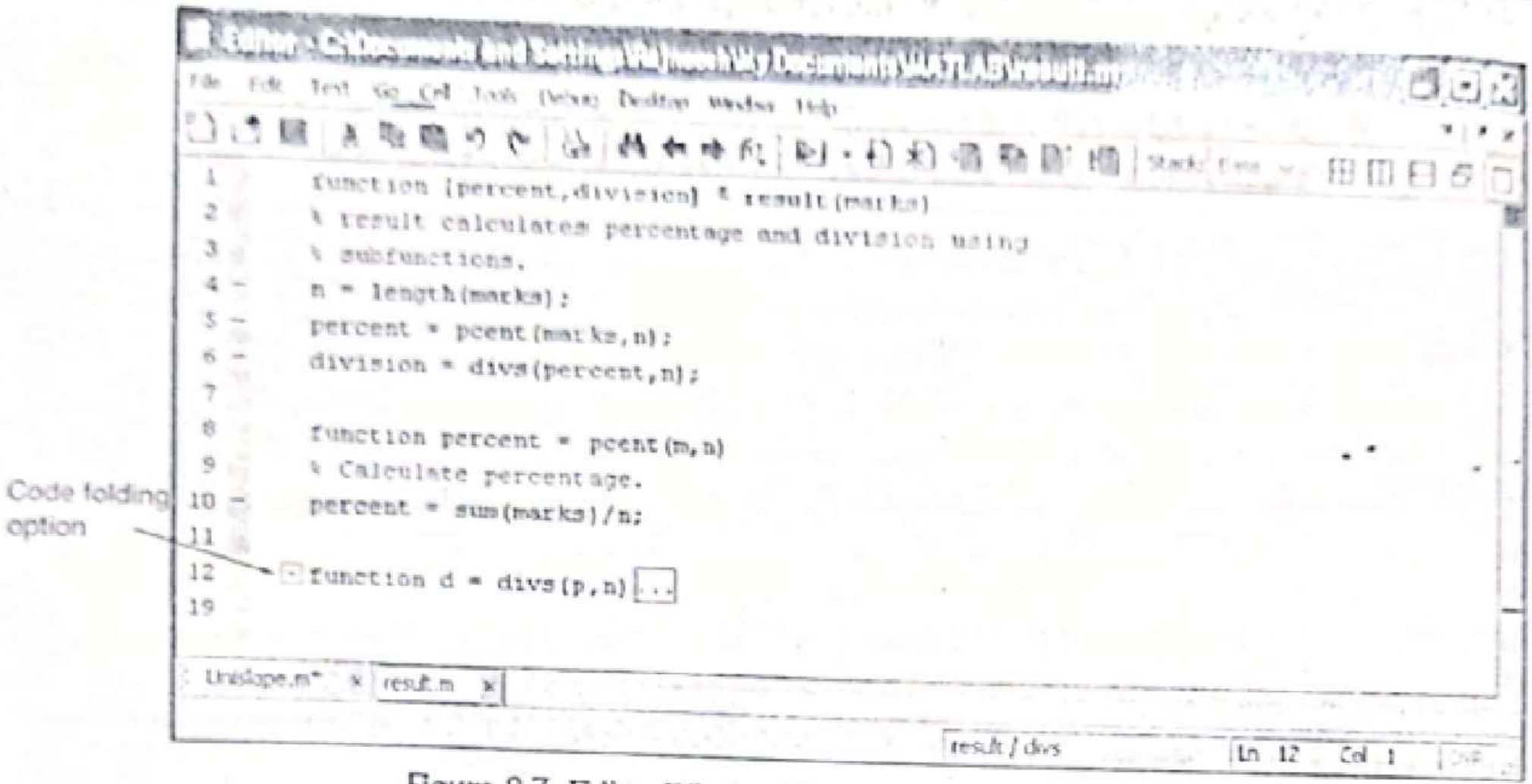


Figure 8.3 Editor Window Illustrating Code Folding

Toolbar

The Editor window in addition to the main menu contains a complete functional toolbar as shown in Figure 8.4. The features found in other text editors like `new file`, `open file`, `save`, `cut`, `copy`, `paste`, `undo`, `redo`, `print` are available on the toolbar and can be invoked by simply clicking on the relevant icon on the toolbar. The toolbar also provides graphical interface for finding a particular text, moving forward or backward in the code and another icon for listing the functions in the text file and moving to any of them by simply selecting the appropriate one from the list.

Further, the toolbar also contains icons for the complete set of debugging tools i.e. setting or clearing breakpoints, executing programs line by line, stepping into the functions, stepping out, continuing execution after stopping at the statements indicated by the breakpoints and quitting debugging. Provision for selecting the program workspace or stack is also available. The last set of icons is for viewing or editing multiple program files in tiled view and so on.

MATLAB Programming

Programming in MATLAB is very similar to that found in C/C++. In previous chapters, it is observed that MATLAB contains most of the basic programming constructs available in other programming languages. It also has option to define classes and objects as done in other object-oriented languages. But, the most important of all is the ease of programming and the tools available in MATLAB. These enable the programmer to develop large scientific applications with shorter code and in far less time.

Program files in MATLAB are called M-files. These files contain a series of MATLAB statements along with the programming constructs or features like `for`, `while` loops or `switch`, `break` and `if` Control statements. As program files in C language have names with extension '.c', MATLAB program files are named with extension '.m'. The program files are created or edited using MATLAB Editor and saved by specifying the filename without the extension. The file extension '.m' is appended to the end of the filename automatically.

MATLAB Programming

1.Creating M-Files.

2.Types of M-files.

Creating M-Files

M-files are ordinary text files and can be created using any text editor whether it is Notepad, Wordpad or MATLAB Editor program. The editor available in MATLAB can be invoked in a number of ways listed as under:

1. Choose new file or open file from the File submenu in MATLAB desktop.
2. Click on the icons for new file or open file on the toolbar available in the MATLAB desktop.
3. Enter the command `edit` or `edit filename` at the command prompt in the MATLAB Command Window.

When `edit` command is given, the Editor is started with the filename `untitled` and the file can be named while storing or saving it in the appropriate folder.

The commands given from the Command Window are saved in the Command History window. M-file can be created using these statements by selecting them in the Command History window, right-clicking and selecting `Create M-File` option. The Editor opens a new file that includes these statements. The statements can also be copied and pasted into an existing M-file.

Types of M-Files

There are two types of M-files:

1. Script files
2. Functions

Both types of files contain MATLAB statements, but have the following major differences:

- The Script files are the simplest of M-files that basically do not accept input arguments or return output arguments. Function files, on the other hand, accept input arguments and return output arguments. These can be 'called' or 'contained' within a Script file or another Function file.
- The Script files operate on data in the workspace. Any variable, that script creates, remains in the workspace even after the script finishes and can be used for further computations. On the other hand, Function files treat the internal variables as local to that function and have their own workspace.
- Function file starts with keyword `function`.

Example 8.1

Write a Script file to plot a ramp function with unity slope.

Solution:

The script file is written & saved in the Editor window and the program to plot the desired ramp function is given below:

```
%The program below creates and plots a ramp function
for x = 0:0.5:50
    y = x;
end;
plot(y, 'r');
xlabel('value of x');
ylabel('value of y');
title('plot of ramp function with unity slope');
```

The file can be executed from the main menu in the MATLAB desktop by typing the name of the file at the command prompt in the Command Window. When execution completes, the variables (x and y) remain in the workspace. The variable list can be displayed using whos command at the command prompt.

Function Subprograms

A large and complex task can be easily executed if divided into smaller sub-tasks or modules. Function subprograms are program segments that are written for implementing a sub-task or a module or a sub-module. These subprograms are developed separately and can be executed independently. These can also be used by other programs. Functions subprograms can be written for the common tasks that are frequently required by many users so that the users do not need to waste time writing code for the common tasks. All users can simply use the functions in their programs. MATLAB also has in its store numerous function subprograms for common tasks related to many fields such as mathematical functions, matrices, graphs, solving differential equations and Fourier Transforms, etc. Besides such predefined functions, MATLAB also contains function subprograms which may be needed frequently for solving many Scientific/Engineering problems.

Functions are M-files that accept input arguments and return output arguments. The structure of the function subprogram is similar to that of any other program, except for the function definition statement that

should be included at the start of the function. The function subprogram includes commands and functions and the function file is normally named after the function name.

Function Subprograms

1.Parts of Function M-file.

2.Function Names.

3.Passing Function Arguments.

4.Function Workspace.

1.Parts of Function M-file.

Parts of Function M-file

The basic parts of a function M-file are given as follows:

1. Function definition line
2. H1 line
3. Help text
4. Function body
5. Comments

Before describing the basic parts of a function M-file, an illustrative example is given below for better understanding:

Example 8.2

Write a function to calculate the roots of a quadratic equation $ax^2 + bx + c = 0$, where a, b and c are constants.

Solution:

A file named 'roots1.m' containing the function roots is created and executed as follows:

```
function [r1, r2] = roots1 (a,b,c)
%This function calculates roots of a quadratic equation
%a, b and c are input arguments
%r1 and r2 are output arguments
disc = b^2-4*a*c;
disc1 = sqrt (disc);
r1 = (-b+disc1)/(2*a)
r2 = (-b-disc1)/(2*a)
```

The above function 'roots1' can be called by giving the following command in the Command Window to calculate the roots of a quadratic equation, $ax^2 + bx + c = 0$.
Let $a = 1$, $b = 4$ and $c = 4$, then the commands used in the Command Window are:

```
a = 1;
b = 4;
c = 4;
[r1, r2] = roots1 (a, b, c);
```

The following answer will be displayed, which gives the two roots:

```
ans =
        r1=-2
        r2=-2
```

Function Definition Line

The function definition line informs MATLAB compiler that the M-file defines a function subprogram. The general form of the function definition line is

```
function[output arguments]=function_name(input arguments)
```

where

1. 'function' is the keyword that indicates that the program following this statement is a function subprogram.

2. 'function_name' is the name of the function and is defined just as other variable names in a program. Normally, it is chosen same as the name of the function subprogram filename.
3. The list of input arguments, if present, is enclosed in parentheses separated by commas.
4. The function output arguments are enclosed in square brackets separated by commas as indicated in the syntax of the function definition line.

Some valid function definition lines and their corresponding file names are as follows:

Function definition Line	File name	Remarks
function [ ] = circle_area (r);	circle_area.m	No output arguments
function rect_area (l,b)	rect_area.m	No output arguments
function [r_1, r_2] = roots (a,b,c)	roots.m	r_1 and r_2 are output arguments

H1 Line

H1 line is the first line, following the function definition line, written as a comment and includes important help statements related to the function subprogram like the details of input and output arguments and the task performed by the function. Because it consists of comment text, the H1 line begins with a percent sign, '%'.

When the user types `'help function_filename'` at the command prompt in the Command Window, this text H1 line is displayed as the first line of help text. The `lookfor` command displays this H1 line only.

For the function given in Example 8.2 for calculating roots of a quadratic equation, H1 line appears as

```
%This function calculates the roots of a quadratic equation
```

Help Text

Help for M-files can be obtained from the command line and can be created by entering help text on one or more consecutive comment lines immediately following the H1 line.

When the user types '`help function_filename`' at the command prompt in the Command Window these comment lines are displayed as the help text between the function definition line and the first executable statement.

The help text for the function given in Example 8.2 for calculating roots of a quadratic equation is

```
%This function calculates roots of a quad. equation
%a, b and c are input arguments
%r1 and r2 are output arguments
```

Function Body

The body of the function contains all the MATLAB statements that perform the required sub-task computations. These statements can consist of programming constructs, such as flow control, interactive input/output and assignment statements.

The body of the function given in Example 8.2 for calculating roots of a quadratic equation is given as

```
disc = b^2-4*a*c;
discl= sqrt (disc);
r1 = (-b+discl)/(2*a)
r2 = (-b-discl)/(2*a)
```

Comments

As mentioned earlier, comment lines begin with a percent sign (%). Comment lines can appear anywhere in an M-file, and one can append comments at the end of a line of code. For example, adding comments to the body of the function given in Example 8.2 for calculating roots of a quadratic equation is given below.

The following comments are added at the end of each statement:

```
disc = b^2-4*a*c;          %calculating discriminant
disc1= sqrt (disc);        %finds square root of discriminant
r1 = (-b+disc1)/(2*a)      %calculates the first root
r2 = (-b-disc1)/(2*a)      %calculates the second
```

2.Function Names.

The rules applied for naming the variable names also apply to function names. Function names must begin with a letter, the remaining characters can be any combination of letters, numbers and underscores. MATLAB uses the first 31 characters of names. The name of the function file and the function name may not be the same but in that case MATLAB ignores the function name. The function is called by the function filename.

When a function is called, MATLAB searches for the relevant file in the following order:

1. Finds if the name is a variable.
2. Checks to see if the name is a subfunction, a MATLAB function that resides in the same M-file as the calling function.
3. Checks to see if the name is a private function, a MATLAB function that resides in a private directory, a directory accessible only to M-files in the directory immediately above it.
4. Checks to see if the name is a function on the MATLAB search path.

MATLAB uses the first file it encounters with the specified name. If the names of variable and many functions available in different folders are same, MATLAB executes the one found first using the above-mentioned rules. It is also possible to overload function names.

Function M-files can be called from either the MATLAB command line or from within other M-files. Let the function definition line of a function `step` be given by

```
function[y,t] = step(num,den)
```

The following are valid call/execution statements:

```
[y, t] = step(n, d)
```

The values of the input variables n and d have to be specified before calling the function. Otherwise, the statement will give error that 'the values of the input arguments are not defined'. The variables y and t are the output variables and some values (outputs) must be assigned to them in the function that is called.

Let the step function defined earlier be called as

```
[Df, x] = step([0 1],[1, 1])
```

It is noted that the input variables are assigned their values directly. Also the output variables, Df and x, have different names than defined in the function subprogram. This indicates that the names given in the function definition line are only **dummy names**. Different names can be specified at the time of call.

3.Passing Function Arguments.

Passing Function Arguments

MATLAB appears to pass all function arguments by value. In reality, however, MATLAB passes by value only those arguments that are modified by the function. If a function does not alter the value of an argument and simply uses it in a computation, then MATLAB passes the argument by reference to optimize memory use.

MATLAB functions can be defined to use varying number of arguments. In such a case when varying number of arguments are used, MATLAB packs all of the specified input or output into a cell array.

4.Function Workspace.

Function Workspace

The function M-file variables take separate memory space from MATLAB Workspace. This memory space is called the **function workspace**. In fact, each function has its own workspace context. Therefore, each MATLAB function has its own local variables, which are separate from those of other functions, and from those of the base workspace.

The variables that are passed to a function must be in the program workspace of the calling program. The called function returns its output arguments to the function workspace of the calling program/function. However, if several functions, and possibly the base workspace, all declare a particular variable as **global**, then they all share a single copy of that variable. Any assignment to that variable, in any function, is available to all the other functions declaring it global.

The following command is used to declare the variables *x*, *y* and *z* as global:

```
global x y z;
```

Further, to declare and work with global variables the following rules are applicable:

- The variable must be declared as global in every function that requires to access that variable.
- If base workspace also needs to access the global variable, it should be declared as global from the command line also.
- The `global` command must appear before the first occurrence of the variable name in each function.

Some variables are required to retain their value between calls to the function. It is possible in MATLAB by declaring them as **persistent variable**. The following command is used to declare variables as persistent:

```
persistent x y z
```

The command defines `x, y` and `z` as persistent variables.

Persistent variables are local to the function in which they are declared and also their values are retained in memory between calls to the function. These variables are similar to global variables because MATLAB creates permanent storage for both. They differ from global variables in the sense that persistent variables are known only to the function in which they are declared. This prevents persistent variables from being changed by other functions or from the MATLAB command prompt. Persistent variable is exactly like global variable, except that the variable name is not in the global workspace, and the value is reset if the M-file is changed or cleared.

Example 8.3

Write a function subprogram to calculate the sum of the following series:

$$S = 1 + r + r^2 + \cdots r^n$$

Solution:

The function subprogram is given as follows:

```
function [s] = seriessum(r,n)     %function definition line
%This function calculates the sum of a series - H1 line
%r is the common rates & n is the no. of terms - Help Text
%Function text body follows
nvector = 0:n;
series = r.^nvector;
s = sum(series)
```

This function series can be called as follows:

(a)

```
seriessum(2, 4)
```

The function will be called and executed. Its output will not be stored. However, it is displayed in the Command Window as there is no semicolon at the end of the function call statement.

(b)
```
r = 2; n = 4;
s = seriessum(r, n)
```

Here, the function will be called and executed. Its result will be stored in the variable `s` and will also be displayed in the Command Window as there is no semicolon at the end of the function call statement.

(c)
```
r = 2; n = 4;
seriessum(r, n);
```

Here, the function will be called and executed. Its output will not be stored and will not be displayed in the Command Window also as the function call is terminated by a semicolon.

Example 8.4

Write a function subprogram to calculate the compound interest, given the initial amount, time period of deposit, rate of interest and time of compounding.

Solution:

The function subprogram is given as:

```
function[cap,interest]=comp(cap,yrs,rate,tcomp)%function def.line
%This function calculates the compound interest - H1 line
%capital, no. of years, rate of interest, period of
%compounding is to be given                              - Help text
a=cap;
n=yrs;
r=rate ;
k = tcomp;
if r>1
   disp('check interest rate. For 7% enter 0.07, not 7.0')
end
cap = a*(1+r/k)^(k*n)
interest = cap-a
```

The function named comp can be called by following methods:

(a) The function comp is called with input arguments alone:

```
comp(1000,6,0.12,1)
cap =
      1.9738e+003
interest =
      973.8227
ans =
      1.9738e+003
```

(b) The function comp is called with input and output arguments alone:

```
[capital,interest]=comp(1000,6,0.08,1)
cap =
      1.5869e+003
```

```
interest =
      586.8743
capital =
      1.5869e+003
interest =
      586.8743
```

As no semicolon is used, the value of variable `cap` and `interest` are displayed first and then as the function call is without semicolon, the values of output arguments compound and interest are also displayed.

(c) The function `comp` is called with input arguments alone but the call is terminated by semicolon:

```
comp(1000,6,0.12,2);
cap =
      2.0122e+003
interest =
      1.0122e+003
```

The variables `cap` and `interest` displayed as the statements in the function are not terminated by semicolon, but as the function call is terminated by semicolon, the result of the function call is not displayed.

Example 8.5

Write a function to check the division scored by a number of students.

Solution:

The program divisioncheck finds the division scored by a number of students:

```
function divisioncheck
%The function finds out the number of students in first
%division among the given no. of total students.
n = input('Give the total number of students\n');
disp('Enter the marks of the students(out of 100) ');
for i=1:n
     k(i)=input('');
end;
x=0;
for i=1:n
     if((k(i)<0)|(k(i)>100))
          fprintf('%d number is invalid \n',i);
     elseif(k(i)>60)
          x=x+1;
     end;
end;
fprintf('The no. of students securing first division is %d\n', x);
```

To execute the function, enter `divisioncheck` at the command prompt in the Command Window. The response of MATLAB is as:

```
Give the total number of students
5
Enter the marks of the students(out of 100)
23
54
```

```
75
456
76
4 number is invalid
The no. of students securing first division is 2
```

To seek help about the function `divisioncheck`, enter the following command:

```
help divisioncheck
```

The MATLAB displays the following help text in the Command Window:

```
The function finds out the number of students in first
division among the given no. of total students.
```

Example 8.6

Write a function to ask for given number of lines and report if they are between 10 and 100 or not.

Solution:

The `inrange` function inputs the number and reports if it is between 10 and 100 or not:

```
function inrange
%function finds whether the given number lies between 10
%and 100 or not.
a= input(' Enter the number\n ');
if (( a>10) & (a<100))
     disp (' The number lies between 10 and 100');
else
     disp('The number does not lie between 10 and 100');
end;
```

If one seeks help about the function `inrange`, the following is the MATLAB response:

```
function finds whether the given number lies between 10 and 100 or not.
```

Functions

TYPES OF FUNCTIONS

Functions can be classified into different categories based on their features and the manner in which they are applied. Some of them are as follows:

- Subfunctions
- Private function
- Nested function
- Inline function

Subfunctions

Function files can contain code for more than one function. The first function in the file is called primary function. The primary function is called by the name of the function M-file. Additional functions defined within the same file are called subfunctions. These functions are visible only to the primary function or other subfunctions defined in the same M-file. Each subfunction begins with its function definition line. These subfunctions follow each other and may occur in any order, as long as the primary function appears first.

Example 8.7

Write a function to calculate percentage and division from a set of marks obtained by a student.

Solution:

The primary function named as `result` calculates the percentage of marks and reports whether the student has scored first division or not.

```
function [percent,division] = result(marks)
%result calculates percentage and division using subfunctions.
n = length(marks);
percent = pcent(marks,n);
division = divs(percent,n);
function percent = pcent(marks,n);
%Calculate percentage.
percent = sum(marks)/n;
function d = divs(p,n)
%Calculate if division is first or not.
If p > 60
     d = 'first'
else
     d = 'not_first'
end
```

Here, `result` is the primary function which contains two subfunctions `pcent` and `divs`. The subfunction `pcent` calculates the percentage of marks and subfunction `divs` finds if the student has scored first division or not. Marks have to be given as input data.

The following points about functions are very significant:

- Functions within the same M-file cannot access the same variables as all subfunctions have their own function workspace. The variables can be shared by declaring them as global variables or pass them as arguments.
- Further, the help command for the function file displays the help text associated with the primary function only.
- As pointed out in the earlier section, when a function is called from within an M-file, MATLAB first checks the file to see if the function is a subfunction. This enables the user to supersede existing M-files using subfunctions with the same name. However, function names must be unique within an M-file.

Private Functions

Private functions are functions that are stored in subfolders with the special name 'private'. These functions are visible only to the functions in their parent folder. For example, let the folder 'student' be on the MATLAB search path. A subfolder called 'private' can be created in the folder 'student'. The folder 'private' can contain functions that can be called only by the functions in its parent folder 'student'. Private functions are invisible outside their parent folder. Therefore, they can have the same names as functions in other folders. This is useful if the user wants to create a different version of a particular function while retaining the original in another folder.

Nested Functions

Just as a `for` loop may be nested within another `for` loop, a function may also be nested within another function. The nested functions provide a way to pass information, without passing it through input/output arguments and without using global variables.

Basic format of a simple nested function is as follows:

```
F1 = primary function (.....)
   ------------------------
   ------------------------          %Code of function F1
   ------------------------
   F2 = nested function (.....)
   ------------------------
   ------------------------          %Code of nested function F2
   ------------------------
   end                               %end of nested function F2
   ------------------------
   ------------------------          %remaining code of F1
end                                  %end of primary function F1
```

A primary function can contain several nested functions within itself. The end statements mark the end of each nested function and the primary function.

Inline Functions

Mathematical expressions can be evaluated by creating functions defining the expressions as follows:

```
function y = mexp(x)
y = x^3 + 4*x^2 + 5*x + 6;
```

Another way to represent the mathematical function at the command prompt is to create an inline object from a string expression. For example, an inline object of the function `mexp` mentioned earlier can be defined as:

```
fx = inline('x^3 + 4*x^2 + 5* + 6', 'x')
```

General form of an `inline` function is

```
Function name = inline('character string expression', 'list of variables')
```

where `character string expression` and `list of variables` are enclosed within single quotes.

Example 8.8

Use `inline` function to evaluate $F(x) = x^3 + 4x^2 + 5x + 6$ for $x = 4$

Solution:

The following MATLAB command can be used:

```
fx = inline ('x^3 + 4 * x^ 2 + 5 * x + 6', 'x' )
```

The following statement will be displayed:

```
fx = inline function:
fx(x) = x^3 + 4*x^2 + 5*x + 6
```

Now type `fx(4)` in the Command Window i.e. x has been given the value equal to 4. The following answer will be displayed:

```
ans
      =  158
```

Example 8.9

Obtain the value of the function given in Example 8.8, for `x = 2`, use built-in function `feval`.

Solution:

The following command can be used in continuation with commands of Example 8.8:

```
y = feval(fx, 2)
```

On pressing the enter key, following answer will be displayed:

```
y
        =      40
```

Functions having more than one argument can also be created with `inline` statement by specifying the names of the input arguments along with the string expression.

Example 8.10

Evaluate the function $f = yx^2 + xy^2$ at $x = 2, y = 3$, using `inline` function

Solution:

The commands used are as follows:

```
f = inline (' y*x^2 + x*y^2', 'x', 'y')
```

To calculate the value of f at x = 2 and y = 3, the following command is used:

```
f(2, 3)
```

The following answer is displayed:

```
ans =
      30
```

FUNCTION HANDLES

As about Handle Graphics® a `function handle` stores all the information about the function which is needed for its execution later. A `function handle` can be created by any of the following two ways:

(a) Using `@` operator.

(b) Using `str2func` function.

Using @ Operator

Suppose it is desired to create a function $f(t) = 4e^{-2t}$, for t in the range of 0 to 5. This function can be defined as follows:

```
Function timres = func(t)
timres = 4 * exp(-2 * t)
end
```

`Function handle` can be created by the following line:

```
handl = @func
```

where `handl` is the name of the `function handle` and `func` is the name of the function. The function `func(t)` can now be executed by just typing the name of the `function handle` and enclosing the arguments in the parentheses. The following MATLAB command will evaluate the function `func(t)`:

```
handle = @func
```

Type the function and save it in the Edit window with the name `func` and give the following command in the Command Window:

```
handle(0 : 5)
```

The following result will be displayed:

```
ans
        = 4.000    0.5413    0.0733    0.0099    0.0013    0.002
```

which gives the values of $4e^{-2t}$, for different values of `t`.

Anonymous Function

Function handle created using @ operator can also be used to create functions called **anonymous functions.** The syntax of general statement for creation of an anonymous function is given as

```
Function Name = @(list of variables/arguments in function) function
expression
```

where @ indicates that the function name mentioned on the left-hand side is a function handle and the list of variables appearing in the expression is enclosed in brackets followed by expression without any punctuation marks.

Example 8.11

Use anonymous function to evaluate:

$$f(x) = x^3 + 4x^2 + 5x + 6\text{, for } x = -2, 2, 4, 6$$

Solution:

Let the function $f(x)$ be defined as an anonymous function, named as `fx`

```
fx = @(x) x^3 + 4*x^2 + 5*x+6
```

Table 8.1 provides the commands and the corresponding answer given by MATLAB for evaluating this function for different given values of x.

Table 8.1 Commands and Results Obtained for Example 8.10

Command	Result Obtained
`fx(-2)`	4
`fx(2)`	40
`fx(4)`	154
`fx(6)`	396

Using `str2func` Function

Function handle can be created and used by typing the following line in the Command Window:

```
handle = str2func('func')
handle (0 : 1 : 5)
```

The following answer will be displayed for the function `func` given in Section 8.7.1.

```
ans
        = 4.000 0.5413 0.0733 0.0099 0.0013 0.002
```

which gives the value of $4e^{-2t}$, for different values of t.

Line and Marker Options

LineSpec (Line Specification)

Line specification

Description

Plotting functions accept line specifications as arguments and modify the graph generated accordingly. You can specify these three components:

- Line style
- Marker symbol
- Color

For example:

```
plot(x,y,'-.or')
```

6.3.3 Using `line` Command

The `line` command is used along with the `plot` command to generate multiple plots. Once a plot is generated in the graphics window, more plots can be added using the `line` command directly. The syntax for the `line` command is

```
line(xdata, ydata, parameter_name, parameter_value)
```

where `xdata` and `ydata` are vectors containing x- and y-coordinates of points on the graph, and parameter name/parameter_values are used to give line style options.

Example 6.10

Illustrate the use of `line` command to generate multiple plots.

Solution:

The following program illustrates use of `line` command to generate multiple plots:

```
% Program to illustrate the use of line command
x = 1:0.1:100;
y1 = sqrt(x .^ 2 + 1);
y2 = 5 * x + 20;
y3 = 10 * x + 3;
plot(x, y1);
gtext('y1=sqrt(x^2+1)');
line(x, y2);
gtext('y2=5x+20');
line(x, y3);
gtext('y3=10x+3');
title('Multiple plots using line command');
grid on;
```

The `plot` command plots the first curve. The `line` command follows to add more curves to the current plot. The graph with multiple plots thus generated is shown in Figure 6.12 (see Figure 6.12 in coloured figure section).

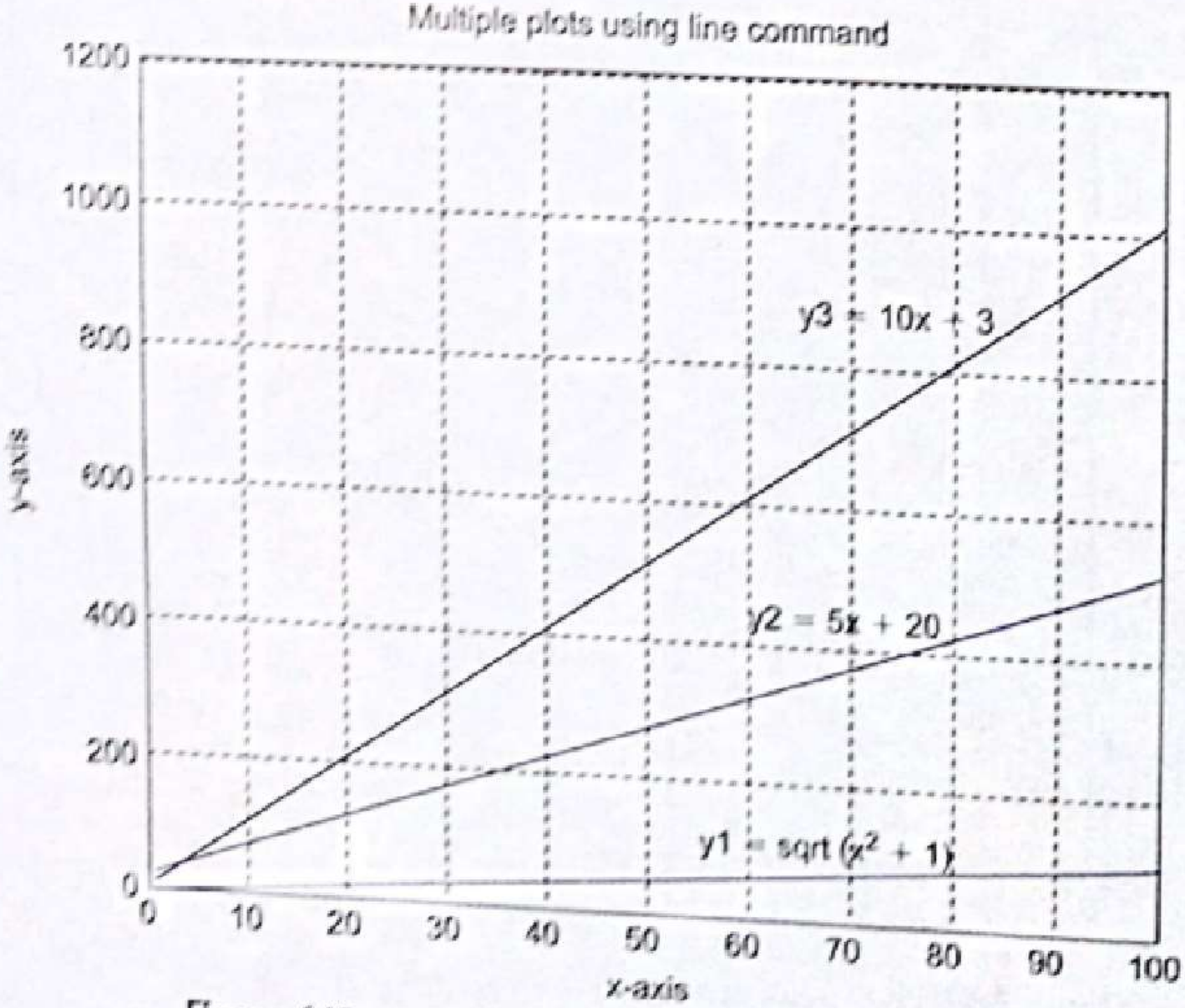


Figure 6.12 Multiple Plots Using `line` Command

STYLE OPTIONS

6.4 STYLE OPTIONS

As seen in the previous sections, for multiple plots many plots on the same graphic window may result in confusion until, or unless, these are properly highlighted and labelled. Hence, for better illustration and outlook of plots different types of style options can be added. These are optional parameters that can be given in the `plot` command as follows:

```
plot (xvalues, yvalues, 'style options');
```

where `xvalues` and `yvalues` are vectors for `x-` and `y-coordinates`, whereas `style-options` is an optional argument, a string which can contain one to four characters enclosed in single quotation marks.

These are used to set the colour, the line style and the marker type for the curves to be drawn, such as:

```
-color(e.g. red, green, yellow, blue etc)
-line style (e.g. solid, dashed, dotted etc.)
-point marker style (e.g. +,*,o, ., etc)
```

The default option of MATLAB *plot is a blue solid line.*

The aforementioned three options in the `plot` command can be used collectively as well as singly or as a combination of the two.

For example,

```
plot(x, y)
%plot y v/s x with default solid line, blue in colour

plot(x, y, 'r')
%plot y v/s x with red coloured line

plot (x, y,'y:')
%plot y v/s x with yellow dotted line

plot (x, y, 'g-')
%plot y v/s x with green dashed line

plot(x, y, '*')
%plot y v/s x as unconnected points marked using (*)

plot (x, y, 'ro')
%plot y v/s x as red lin0e with circles as marker style
```

The different colour, line style and marker style options are given as in Table 6.1.

Example 6.11

Illustrate the different `style options` as used in plots.

Solution:

The following program illustrates the different `style options` as used in plots:

```
% Program to illustrate the use of style options
t=0 : 0.1 : 10;
```

Table 6.1 Colours, Line Style and Marker Style of Style Options

Colour Style Options		Marker Style Options		Line Style Options	
y	yellow	.	point	-	solid
m	magenta	o	circle	:	dotted
c	cyan	x	x-mark	-.	dashdot
r	red	+	plus	--	dashed
g	green	*	star		
b	blue	s	square		
w	white	d	diamond		
k	black	v	triangle(down)		
		^	triangle (up)		
		<	triangle(left)		
		>	triangle(right)		
		p	pentagram		
		h	hexagram		

```
y1 = 3*t.^2 + 5*t + 6;
y2 = t.^3 + 4;
plot (t, y1, 'r', t, y2, 'mo');
xlabel ('x-axis');
ylabel('y-axis');
title ('Plots illustrating use of style options');
grid on;
```

The plot statement in this program plots two curves. The style for the first curve is undefined. Therefore, solid line is plotted. The colour is, however, specified as red instead of default blue. The style of second curve is using circle as marker style and the colour is specified as magenta. The plots obtained are plotted as shown in Figure 6.13 (see Figure 6.13 in coloured figure section).

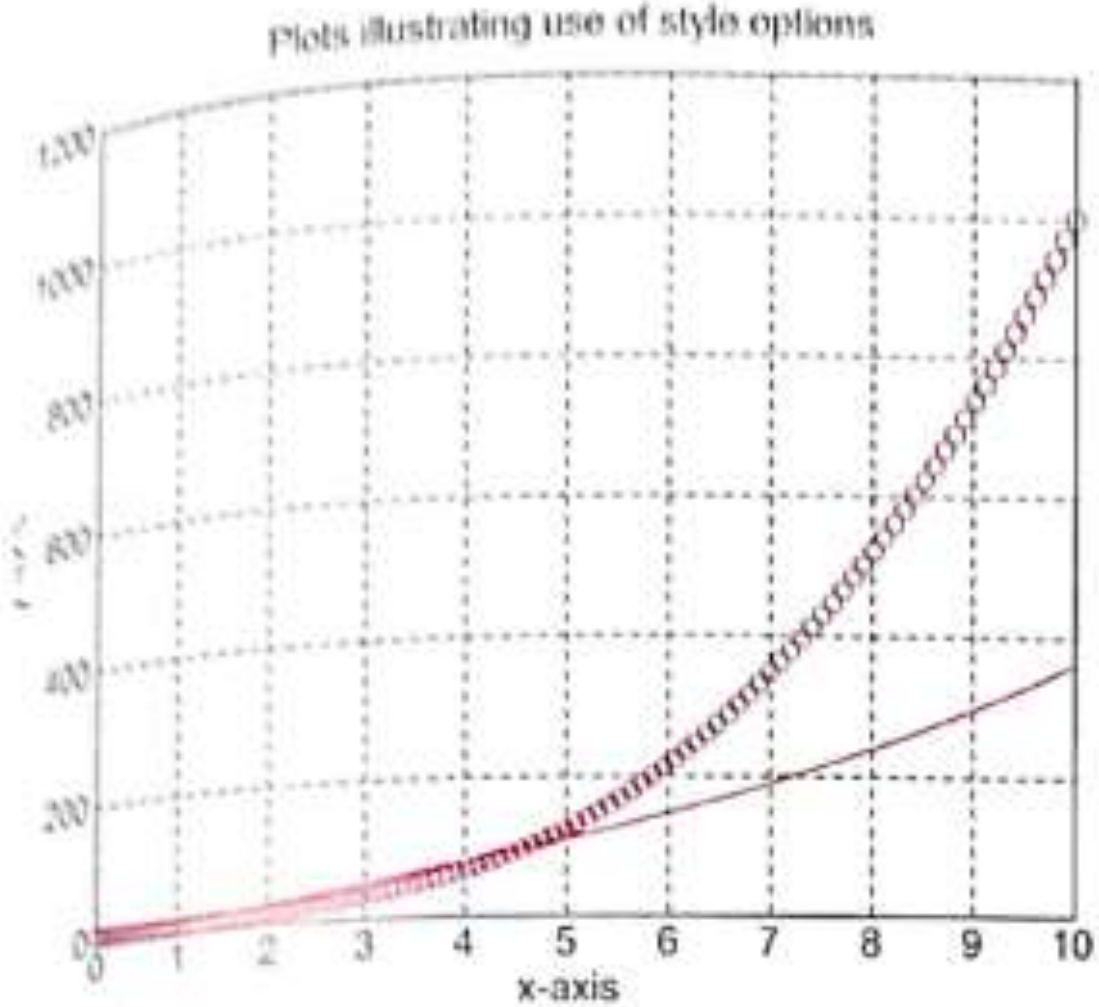


Figure 6.13 Plot Showing the Use of Style Options

THREE DIMENSIONAL PLOTS (3D Plots)

6.8 THREE-DIMENSIONAL PLOTS

For visualization of data in three dimensions, MATLAB provides a number of functions. Most of the functions are the extension of 2-D functions such as `plot3`, `bar3`, `bar3h` and `pie3`. The 3-D graphs are generally used to display two types of data. First, when two variables are the function of independent variable and secondly, when single variable is a function of two independent variables. To display the data which is the function of two independent variables `surface`, `mesh` and `contour` plots are used.

Some of the built-in 3-D functions are discussed as follows:

6.8.1 plot3

plot3 function is an extension of `plot` function used in 2-D data display. In case of `plot` function, two variables are used whereas in `plot3` function three variables or 3-D function of a single variable is displayed. The syntax for this function is as follows:

```
plot3(x, y, t, 'style_options')
```

where

`x`, `y` and `t` are arrays specifying location of data points, and
`style_options` specifies the style options as used in 2-D plot.

Example 6.34

Given is $x = t^2$ and $y = 4t$ for $-4 < t < 4$. Obtain a 3-D plot showing the matrix in (x, y) space as a factors of time.

Solution:

The commands used are as

```
%Program to illustrate the plot3 function
t=-4:0.1:4;
x=t.^2;
y=4*t;
plot3(x,y,t);
grid on;
xlabel('x-axis');
ylabel('y-axis');
zlabel('z-axis');
title('Illustration of plot3 function');
```

The plot obtained is as shown in Figure 6.36 (see Figure 6.36 in coloured figure section).

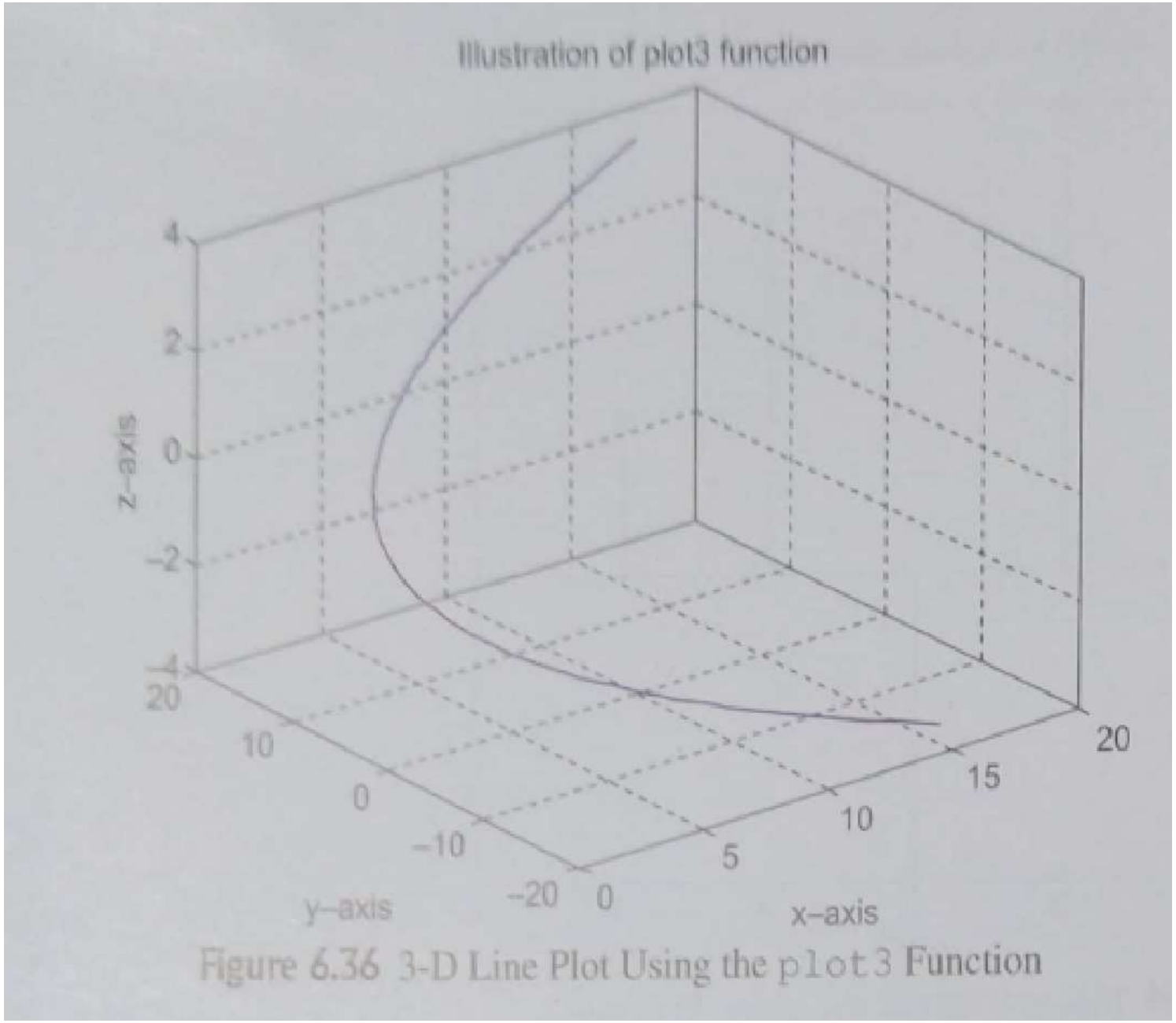


Figure 6.36 3-D Line Plot Using the `plot3` Function

6.8.2 `bar3`

`bar3` function is similar to `bar` function in 2-D plots except that it gives 3-D plot. All features of `bar` function are applicable in `bar3` function.

Example 6.35

Draw 3-D bar graph of the following data:

$x = [0\ 1\ 2\ 3\ 4\ 5\ 6]$

$y = [5\ 17\ 25\ 21\ 31\ 27\ 19]$

Solution:

The program is given as follows:

```
%Program to plot 3-D graph using bar3 function
x=[ 0 1 2 3 4 5 6];
y=[5 17 25 21 31 27 19 ];
bar3(x,y,'g');
xlabel('x-axis');
ylabel('y-axis');
title('Graph to show bar3 function');
```

The graph obtained is as shown in Figure 6.37 (see Figure 6.37 in coloured figure section).

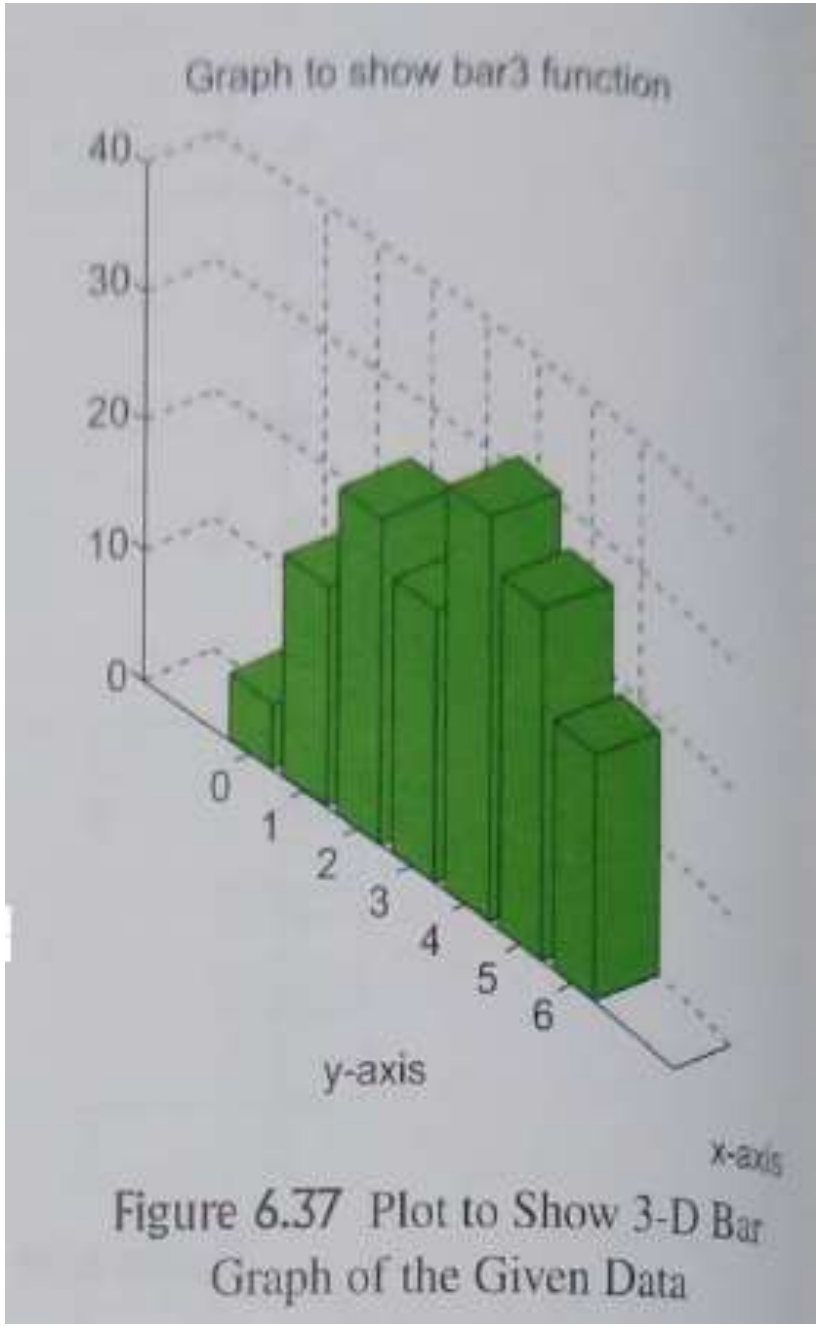


Figure 6.37 Plot to Show 3-D Bar Graph of the Given Data

6.8.3 `bar3h`

The function of **`bar3h`** is similar to `bar3` function except that it plots horizontal plot.

Example 6.36

Draw a 3-D horizontal bar to show the growth of a state as per the following data:

Year = [1992 1993 1994 1995 1996 1997 1998]

Growth (in % age) = [12 15 10 8 6.5 9 11]

Solution:

The program is given as

```
%Program to illustrate use of bar3h function
year=[1992 1993 1994 1995 1996 1997 1998];
growth=[12 15 10 8 6.5 9 11];
bar3h(year,growth,'c');
title('Yearly growth using bar3h function');
zlabel('Year');
ylabel('Percentage growth');
```

The response obtained is shown in Figure 6.38 (see Figure 6.38 in coloured figure section).

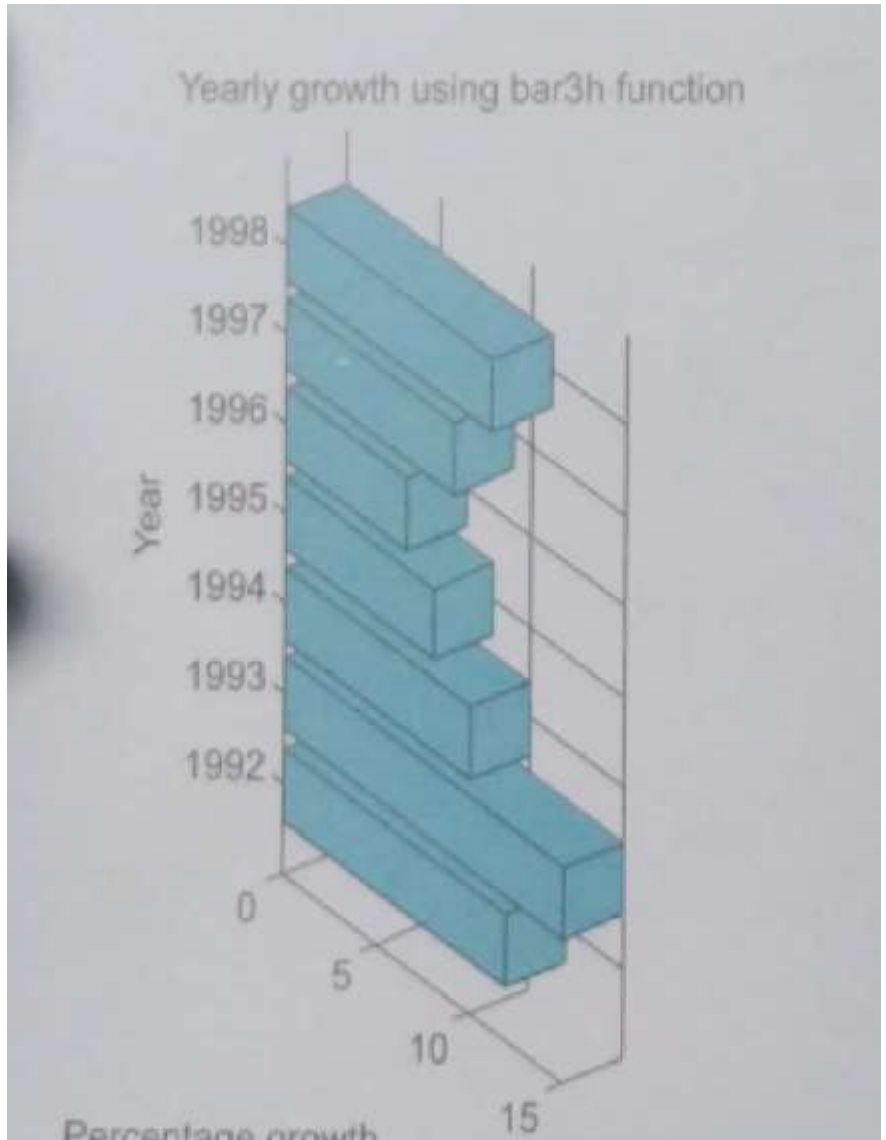


Figure 6.38 Plot to Show Growth of a State Using 3-D Horizontal Bar Graph

6.8.4 pie3

`pie3` function draws a 3-D plot of the given data. The syntax is given as

```
pie3(x)
pie3(x,k)
pie3(x,k, labels)
```

where x, k and `labels` have same meaning as defined in `pie` function earlier.

Example 6.37

Illustrate the use of `pie3` function to show different profession people in a group as per the following data:

Profession	No. of Persons
Manager	12
Engineer	15
Professor	13
Doctor	11
Architect	6
Designer	10

Solution:

The program is given as follows:

```
%Program to illustrate use of pie3 function
person={'Manager','Engineer','Professor','Doctor','Architect','Designer'};
strength=[12 15 13 11 6 10];
explode= [0 0 1 0 0 1];
pie3(strength,explode,person);
title('Persons in a group using pie3 function');
```

The response obtained is shown in Figure 6.39 (see Figure 6.39 in coloured figure section).

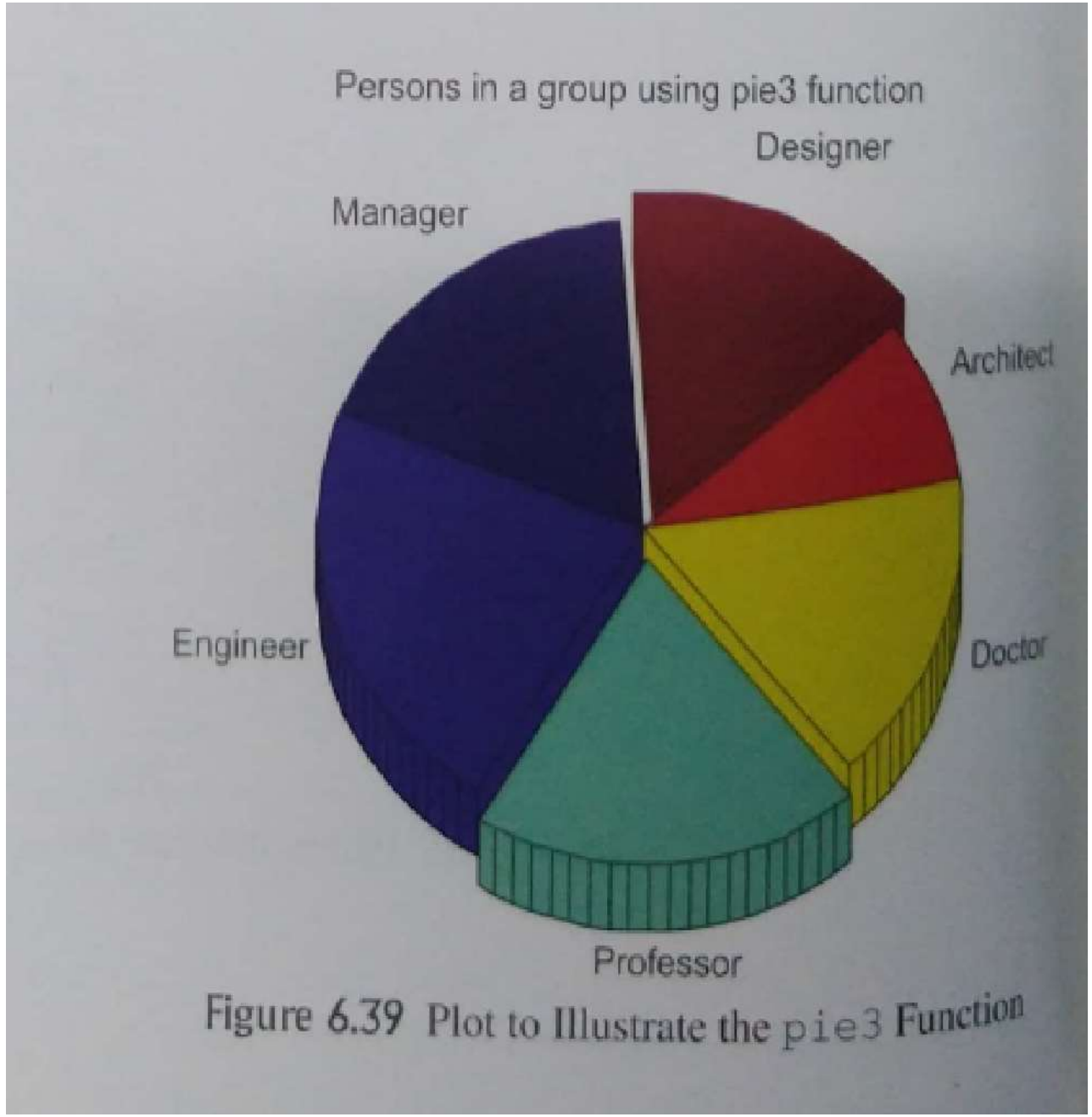


Figure 6.39 Plot to Illustrate the `pie3` Function

6.8.5 `stem3`

The syntax is given as follows:

```
stem3(a)
stem3(x, y, z)
stem3(…,'filled')
```

The function `stem3(a)` draws the discrete surface a as stems from the xy-plane terminated with circles for the data value.

The function `stem3(x, y, z)` plots the surface z at the values specified in *x* and *y*.

The function `stem3(…,'filled')` produces a stem plot with filled markers.

Example 6.38

Write a program to illustrate the use of `stem3` function to plot the following function:

$y = x\cos(x)$; $z = \exp(x/5)\cos(x) + 1$ for $0 \leq x \leq 6\pi$.

Solution:

The program is given as

```
%Program to illustrate the stem3 command
x=linspace(0,6*pi,200);
y=x.*cos(x);
z=exp(x/5).*cos(x)+1;
stem3(x,y,z);
title('Graph to illustrate stem3 function');
xlabel('x-axis');
ylabel('y-axis');
zlabel('z-axis');
```

The response is given in Figure 6.40 (see Figure 6.40 in coloured figure section).

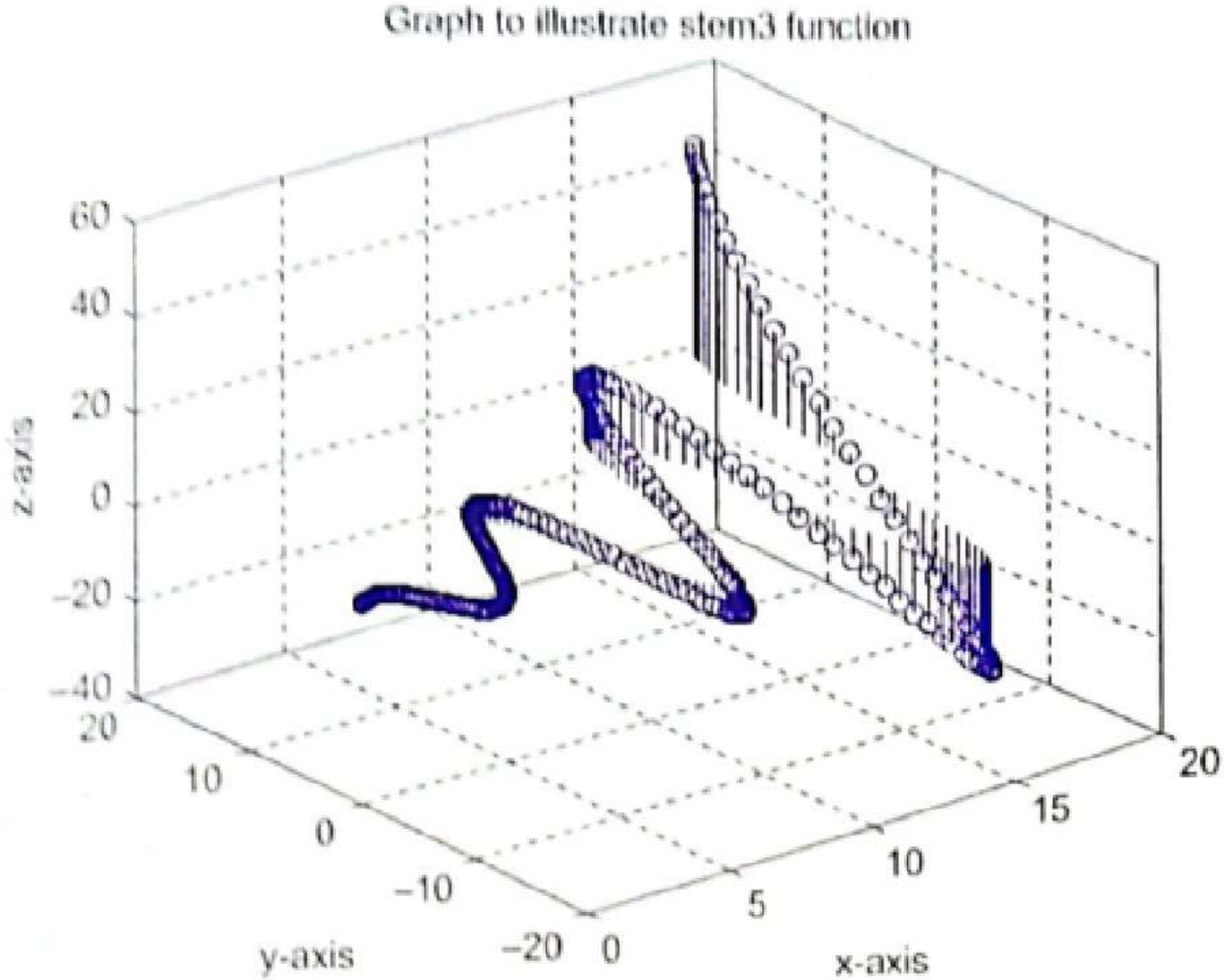


Figure 6.40 Plot to Illustrate the `stem3` Function

6.8.6 meshgrid

The **meshgrid** function transforms the domain specified by vectors x and y into arrays *x* and *y* that can be used for evaluation of the functions of two variables and 3-D surface plots. The syntax is given as

```
[x,y] = meshgrid(xbegin: xinc: xend, ybegin:yinr:yend)
```

where

xbegin, ybegin are starting value of x and y variable,
xinr, yinr are increments of x and y variable and
xend, yend are final values of x and y variable, respectively.

After creating arrays of x and y values, the given function is evaluated and plotted using mesh, surf or contour plots.

Example 6.39

Create `meshgrid` for the data $-1 \leq x \leq 1$ and $-2 \leq y \leq 2$.

Solution:

The command used is given as

```
[x,y]= meshgrid(-1:0.5:1, -2:1:2)
```

This will create x and y array and the following is result obtained:

```
x =
    -1.0000   -0.5000         0    0.5000    1.0000
    -1.0000   -0.5000         0    0.5000    1.0000
    -1.0000   -0.5000         0    0.5000    1.0000
    -1.0000   -0.5000         0    0.5000    1.0000
    -1.0000   -0.5000         0    0.5000    1.0000
```

```
y =
    -2    -2    -2    -2    -2
    -1    -1    -1    -1    -1
     0     0     0     0     0
     1     1     1     1     1
     2     2     2     2     2
```

6.8.7 mesh

mesh function draws wireframe mesh defined by the given arguments.

```
mesh (x, y, z, c)
mesh (x, y, z)
```

where

x is an array containing *x*-values,
y is an array containing *y*-values,
z is an array containing *z*-values and
c argument determines the colour scaling. In case *c* is not mentioned, it is taken to be equal to *z*.

- **Example 6.40**

Create a mesh plot of the function $z = 2/(x^2 + y^2 + 1)$ over the interval $-5 \leq x \leq 5$ and $-5 \leq y \leq 5$.

Solution:

The commands used are as

```
%Program to illustrate the mesh function
[x,y]=meshgrid(-5:0.1:5);
z=2./(x.^2+y.^2+1);%Function to be plotted
mesh(x,y,z);
grid on;
title('Program to illustrate the mesh function');
xlabel('x-axis');
ylabel ('y-axis');
zlabel('z-axis');
```

The response obtained is shown in Figure 6.41 (see Figure 6.41 in coloured figure section).

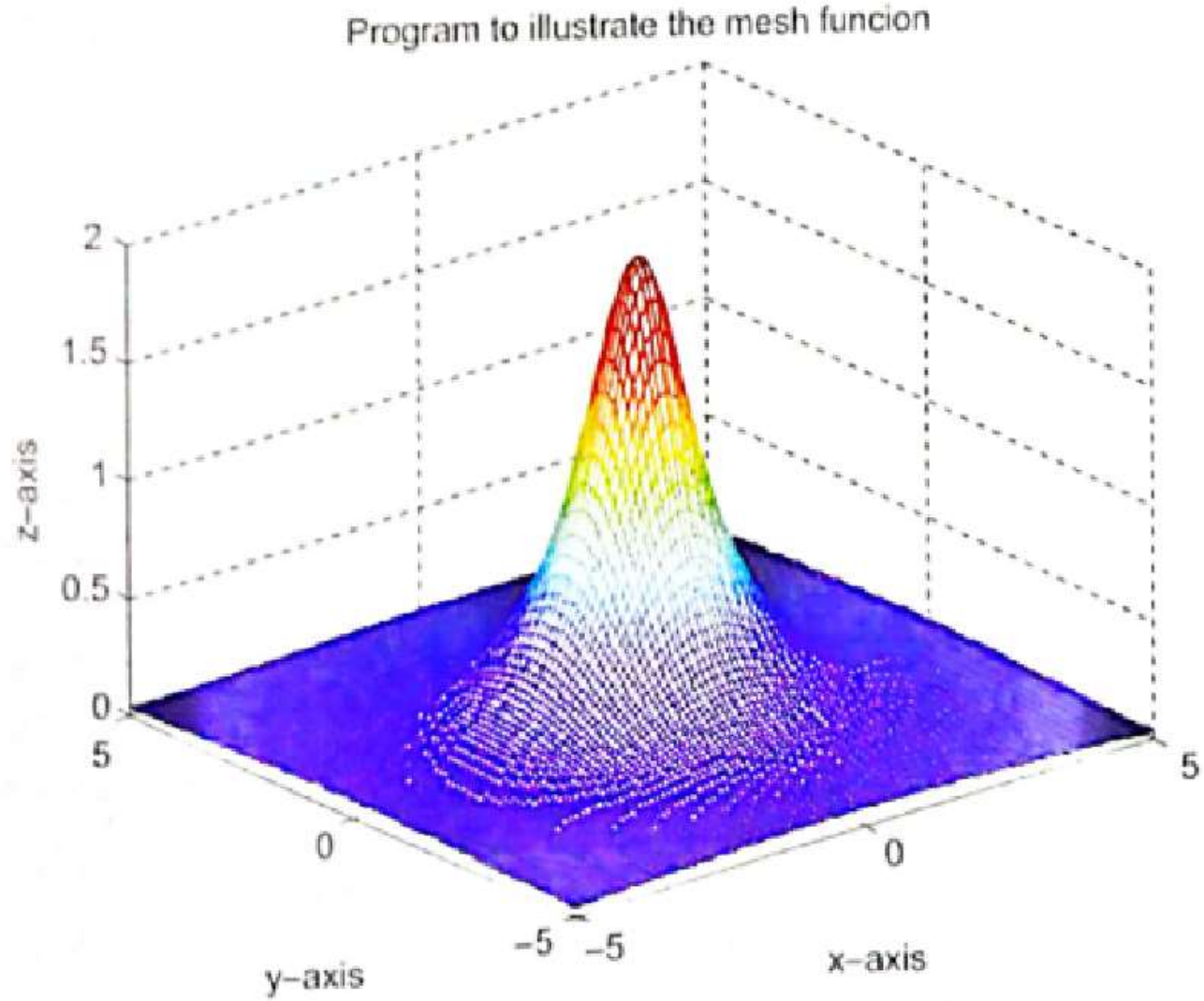


Figure 6.41 Plot to Illustrate mesh Function

Axis Modes

Axis Control

A number of options are available for setting the scaling, orientation and aspect ratio of plots, using the `axis` command. In the plots obtained in MATLAB, the default axis length depends upon the maximum and minimum values specified in x and y vectors and is selected automatically. However, the axes length can be changed as desired with the help of `axis` command. The syntax is

```
axis([xmin xmax ymin ymax])
```

where

`xmin` and `xmax` gives minimum and maximum limits of x-axis, respectively
`ymin` and `ymax` gives minimum and maximum limits of y-axis, respectively.

For example, the MATLAB statement

```
axis([-3 3 0 10])
```

when executed, will set x-axis limits from -3 to 3 and y-axis limits from 0 to 10. The limits can also be set by defining vectors for x-axis and y-axis and then using these vectors in `axis` command. For example,

```
a = [-3 3 0 10];
axis (a);
```

or

```
c = [-3 3];
d = [0 10];
axis([c d]);
```

These produce same result as `axis` command containing the x and y minimum and maximum values.

The axes limits can also be allowed to set partially by writing **`inf`** as the value of a limit. Consider the following statement:

```
axis( [-5 5 -inf 10] );
```

This command will set x-axis limits at -5 and 5, and lower limit of y-axis is set automatically based on the values of data to be plotted and upper y-axis limit as 10.

Similarly, consider the following statement:

```
axis([-5 inf -inf 10]);
```

This command will set lower x-axis limit as -5 and upper y-axis limit as 10, whereas upper x-axis and lower y-axis limits are allowed to be set automatically depending on the data points to be displayed in the figure.

The command `axis auto` can be used to re-enable automatic limit selection.

Axis Aspect Ratio

There are some pre-defined string arguments for the `axis` command for setting the axis aspect ratio given as follows:

```
axis('equal')
```

It sets equal scale on both axes.

```
axis('square')
```

It sets the default rectangular frame to a square frame.

```
axis('normal')
```

It resets the axis to default values.

```
axis('axis')
```

It freezes the current axes limits.

```
axis('off')
```

It turns off all axis lines, tick marks and labels.

The `axis` command should be used after the `plot` command.

Example 6.7

Illustrate the use of `axis` command with specified limits for plot of function $y = 3x^2 + 2x + 5$ for $x = -5$ to 10.

Solution:

The following program illustrates the use of `axis` command with specified limits for plot of given function:

```
x = -5:0.1:10;
y =3 * x.^2 + 2 * x + 5;
plot (x, y);
axis ([-5 5 0 20]);
xlabel ('x-axis');
ylabel('y-axis');
title ('Plot of function y = 3x^2 + 2x + 5');
grid on;
```

The response obtained is shown in Figure 6.9.

The plot can be obtained with equal x- and y-axis by replacing the existing `axis` command by the following command:

```
axis ('equal');
```

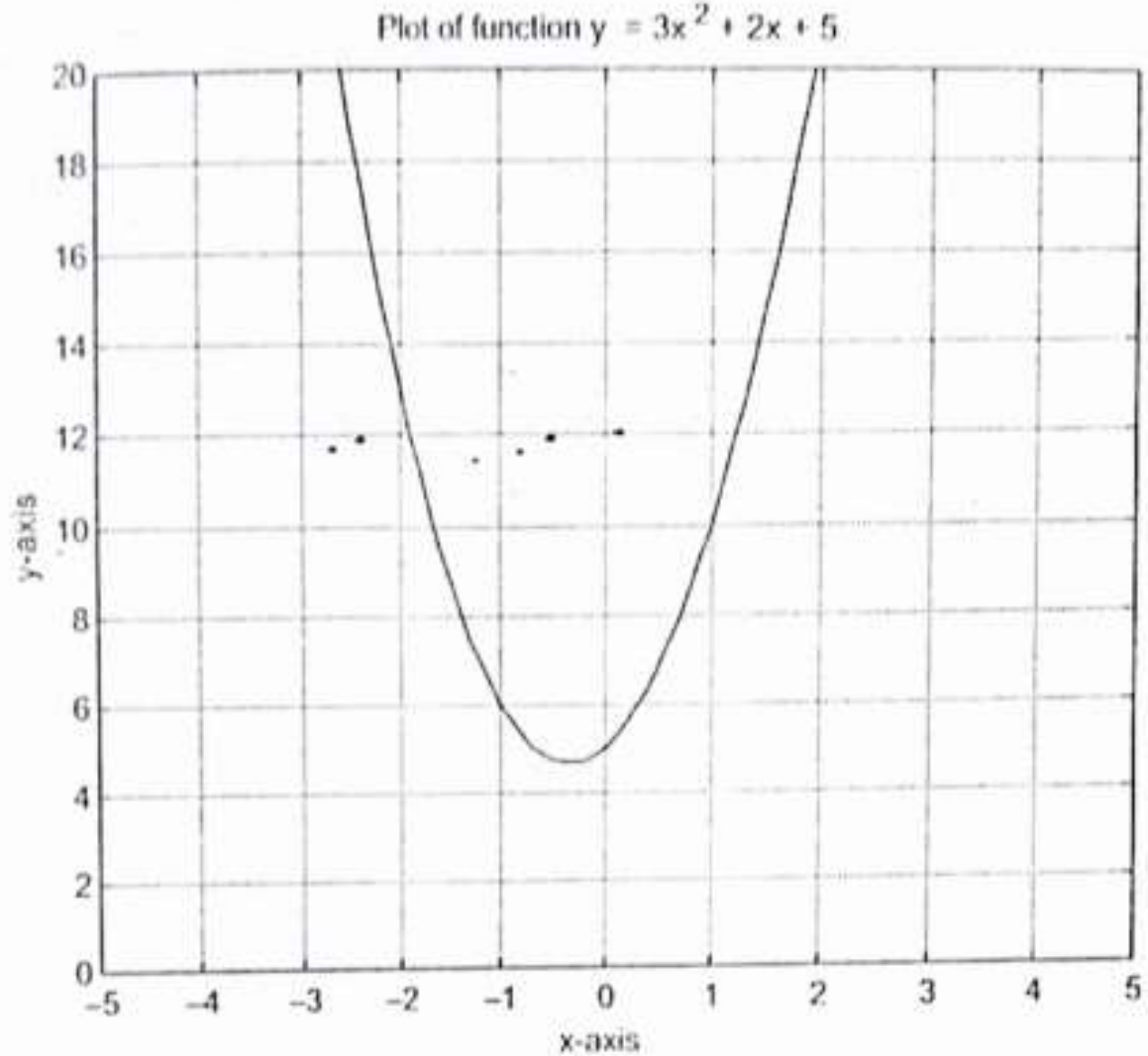


Figure 6.9 Plot to Illustrate the Use of Axis Control

Multi plots in one figure

MULTIPLE PLOTTING

6.3 MULTIPLE PLOTS

There are three different methods of drawing multiple plots in MATLAB:

1. Using `plot` command
2. Using `hold` command
3. Using `line` command

6.3.1 Using `plot` Command

In this method, the earlier explained `plot` command in modified by providing more arguments to generate more plots in the same graphic window.

For example, let there be three plots to be drawn for three data sets, that is, $(x1, y1)$, $(x2, y2)$ and $(x3, y3)$; then the `plot` command will be

```
plot(x1, y1, x2, y2, x3, y3);
```

This will generate three graphs in the same graphics window.

Example 6.8

Illustrate the use of `plot` command for generating multiple plots.

Solution:

The following program illustrates the use of `plot` command for generating multiple plots:

```
% Program to illustrate the use of plot command
x1 = 0:0.01:20;
y1 = exp(0.1*x1).*sin(x1);
x2 = 0:0.1:20;
y2 = sin(x2);
x3 = 0:0.1:20;
y3 = cos(x3);
plot(x1, y1, x2, y2, x3, y3);
gtext ('Curve y1'); gtext ('Curve y2'); gtext ('Curve y3');
title('Multiple plots using plot command');
xlabel('x-axis');
ylabel ('y-axis');
grid on;
```

The first six statements in the program generate x-y coordinates for the three curves. The `plot` command plots the three curves using single statement. *In this method, vector (xi, yi) must have pair-wise same length.* The response obtained is shown in Fig. 6.10 (see Figure 6.10 in coloured figure section).

The `plot` command can also be used with matrix arguments. In such a case, each column of second argument matrix is plotted against the corresponding column of first argument matrix. For example,

```
x = [x1 x2 x3];
y = [y1 y2 y3];
```

where $x1$, $x2$ and $x3$, and $y1$, $y2$ and $y3$ are column vectors of same length, then multiple plots can be generated using the following command:

```
plot(x, y)
```

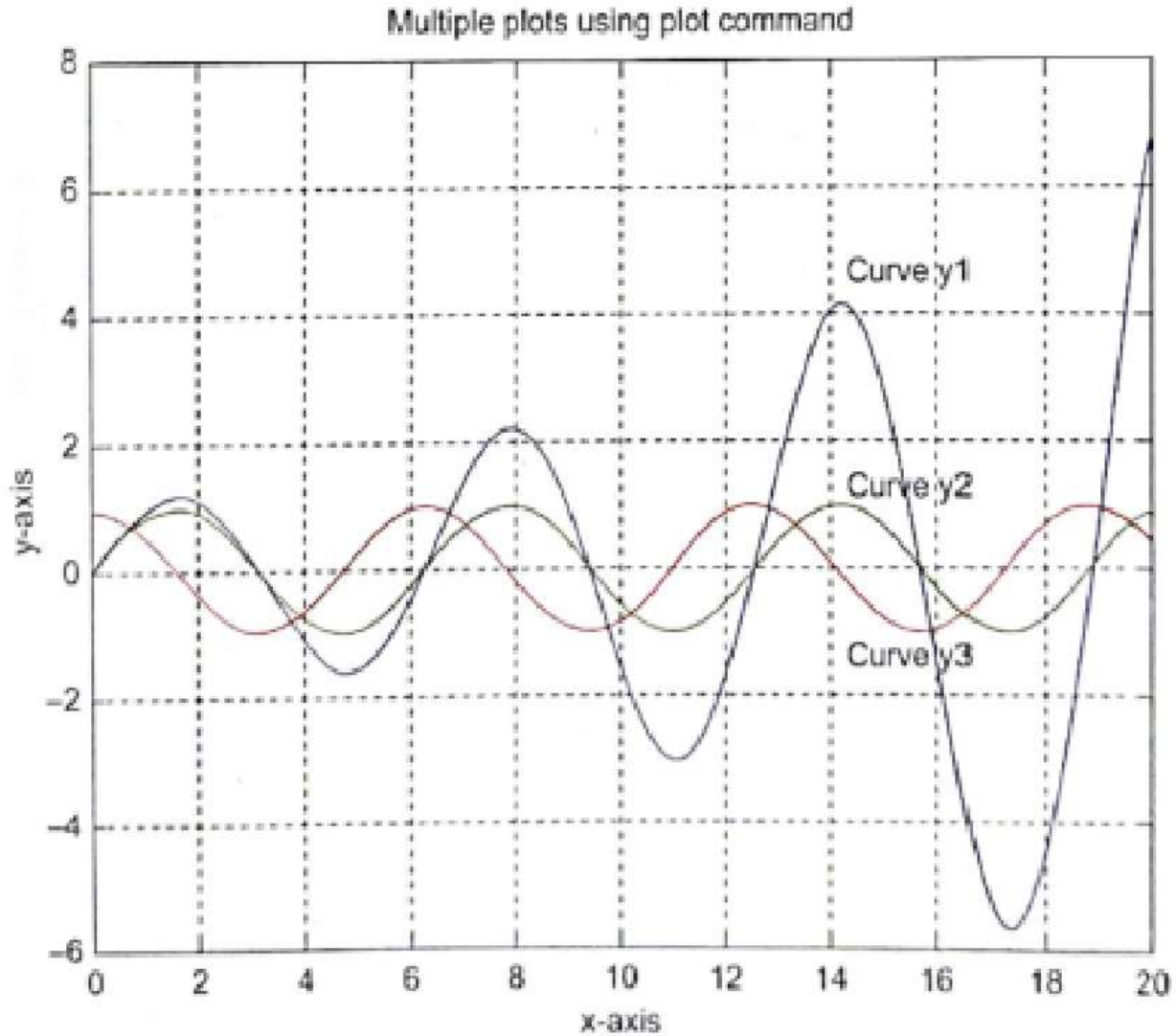


Figure 6.10 Plot to Illustrate the `plot` Command for Multiple Plots

6.3.2 Using `hold` Command

The `hold` command is used to freeze the current plot in the graphics window. The syntax is

```
hold on;
```

The `hold` command holds the cursor on the current plot in the graphics window and subsequent plots generated in the program are added to the same graph.

The **`hold off`** command is used to clear the `hold on` command.
`hold` used repetitively toggles the hold state.

Example 6.9

Illustrate the use of hold command for generating multiple plots.

Solution:

The following program illustrates use of hold command for generating multiple plots:

```
%Program to illustrate use of hold command
x= 0:0.5:10;
y1 = 5 + 4x;
plot(x, y1);
hold on;
y2 = x .^ 2;
plot(x, y2);
hold off;
gtext('y1 = 5 + 4x');
gtext('y2 = x^ 2');
grid on;
title('Multiple plots using hold command');
x label ('x - axis');
y label ('y - axis');
```

The response obtained is shown in Figure 6.11 (see Figure 6.11 in coloured figure section).

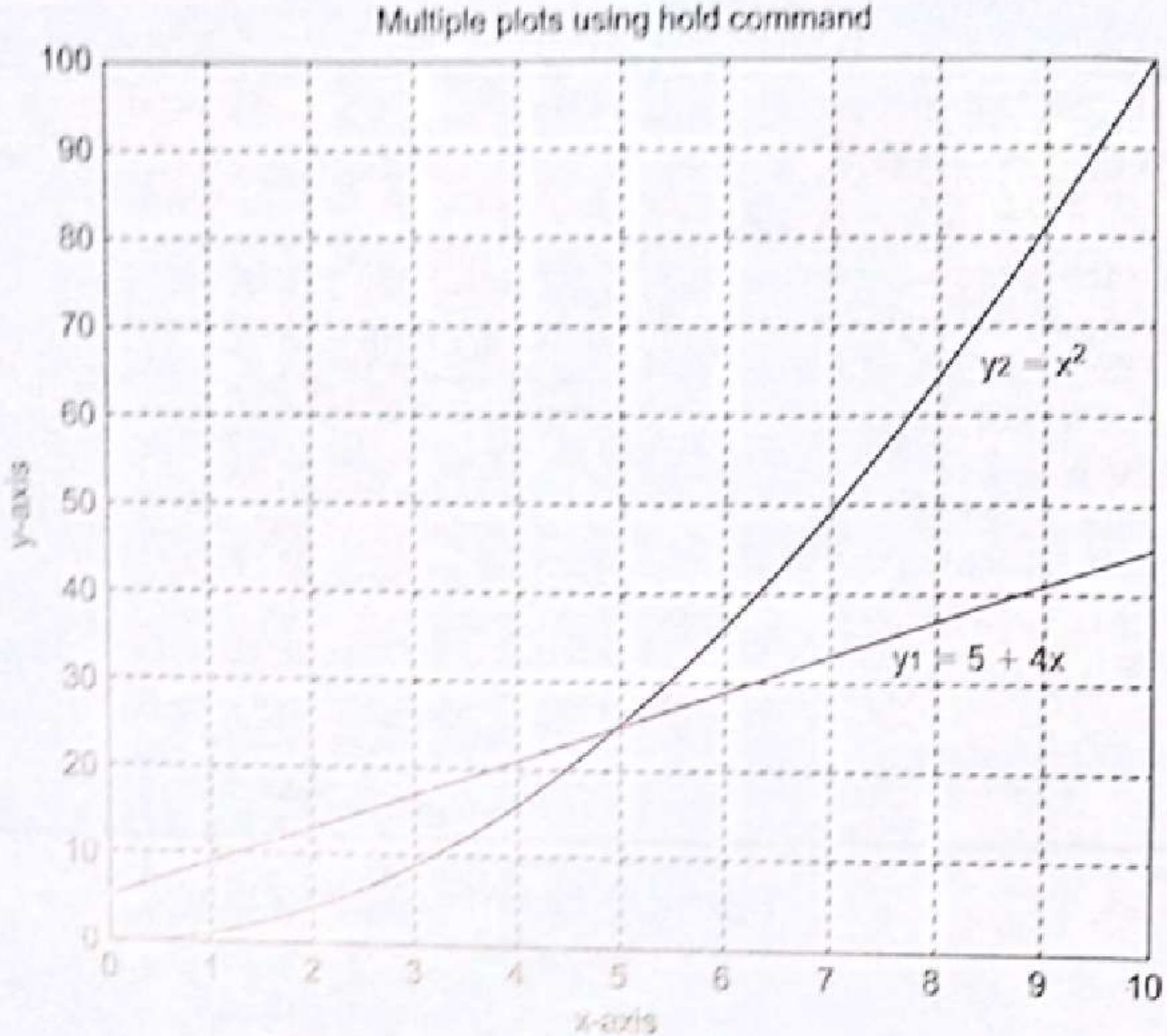


Figure 6.11 Plot to Illustrate the Use of `hold` Command

6.3.3 Using `line` Command

The `line` command is used along with the `plot` command to generate multiple plots. Once a plot is generated in the graphics window, more plots can be added using the `line` command directly. The syntax for the `line` command is

```
line(xdata, ydata, parameter_name, parameter_value)
```

where `xdata` and `ydata` are vectors containing x- and y-coordinates of points on the graph, and parameter name/parameter_values are used to give line style options.

Example 6.10

Illustrate the use of `line` command to generate multiple plots.

Solution:

The following program illustrates use of `line` command to generate multiple plots:

```
% Program to illustrate the use of line command
x = 1:0.1:100;
y1 = sqrt(x .^ 2 + 1);
y2 = 5 * x + 20;
y3 = 10 * x + 3;
plot(x, y1);
gtext('y1=sqrt(x^2+1)');
line(x, y2);
gtext('y2=5x+20');
line(x, y3);
gtext('y3=10x+3');
title('Multiple plots using line command');
grid on;
```

The `plot` command plots the first curve. The `line` command follows to add more curves to the current plot. The graph with multiple plots thus generated is shown in Figure 6.12 (see Figure 6.12 in coloured figure section).

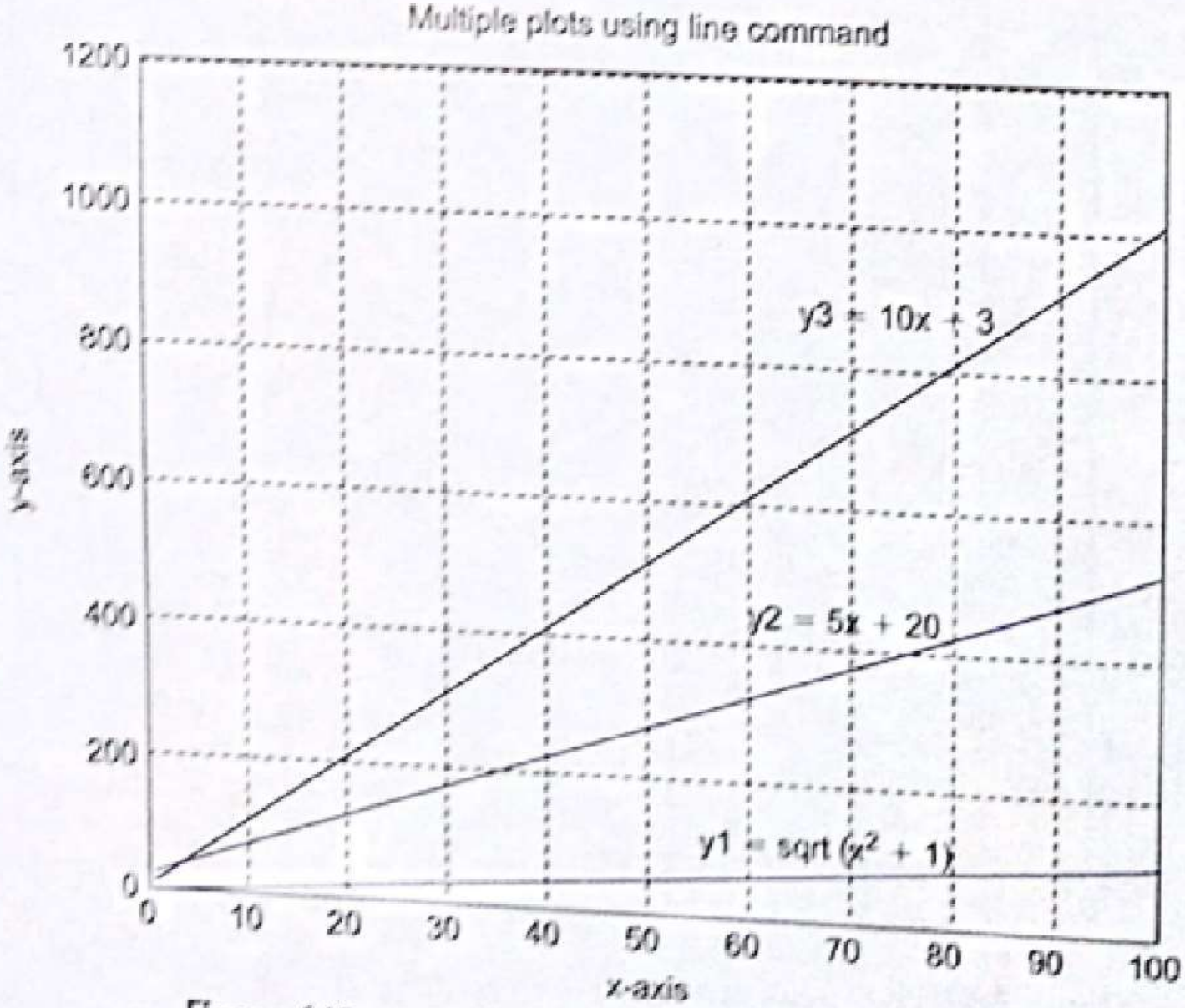


Figure 6.12 Multiple Plots Using `line` Command

Visualising Matrices

spy

Visualize sparsity pattern of matrix

coll

Syntax

```
spy(S)
spy(S,LineSpec)
spy(__,MarkerSize)
```

Description

`spy(S)` plots the sparsity pattern of matrix `S`. Nonzero values are colored while zero values are white. The plot displays the number of nonzeros in the matrix, `nz = nnz(S)`.

`spy(S,LineSpec)` additionally specifies `LineSpec` to give the marker symbol and color to use in the plot. For example, `spy(A,'r*')` uses red asterisks for nonzeros.

`spy(__,MarkerSize)` specifies `MarkerSize` to give the size of the markers using either of the previous input argument combinations.

Color Maps

colormap

View and set current colormap

Syntax

```
colormap map
colormap(map)
colormap(target,map)
```

```
cmap = colormap
cmap = colormap(target)
```

Description

`colormap map` sets the colormap for the current figure to one of the predefined colormaps. If you set the colormap for the figure, then axes and charts in the figure use the same colormap. The new colormap is the same length (number of colors) as the current colormap. When you use this syntax, you cannot specify a custom length for the colormap. To learn more about colormaps, see What Is a Colormap?

`colormap(map)` sets the colormap for the current figure to the colormap specified by `map`.

`colormap(target,map)` sets the colormap for the figure, axes, or chart specified by `target`, instead of for the current figure.

`cmap = colormap` returns the colormap for the current figure as a three-column matrix of RGB triplets.

`cmap = colormap(target)` returns the colormap for the figure, axes, or chart specified by `target`.

Surface Plots & Surf Options

6.8.8 `surf`

`surf` function is similar to `mesh` function with the difference that instead of wireframe plot it is a surface plot. The syntax is given as

```
surf(x, y, z, c)
surf(x, y, z)
```

where

x is an array containing *x*-values,
y is an array containing *y*-values,
z is an array containing *z*-values and
c argument determines the colour scaling.
In case c is not mentioned, it is taken to be equal to z.

Example 6.41

Create a surface plot of the function $f(x, y) = 8 - x^2 - 9y^2$ over the interval $-2 \leq x \leq 2$ and $-3 \leq y \leq 3$.

Solution:

The program is given as follows:

```
%Program to illustrate the surf function
[x,y]= meshgrid (-2:0.1:2, -3:0.5:3);
z = 8-x.^2-9*y.^2;      %Function to be plotted
surf (x,y,z);
grid on;
title ('Program to illustrate the surf function');
xlabel('x-axis');
ylabel('y-axis');
zlabel('z-axis');
```

The response obtained is shown in Figure 6.42. (See Figure 6.42 in coloured figure section.)

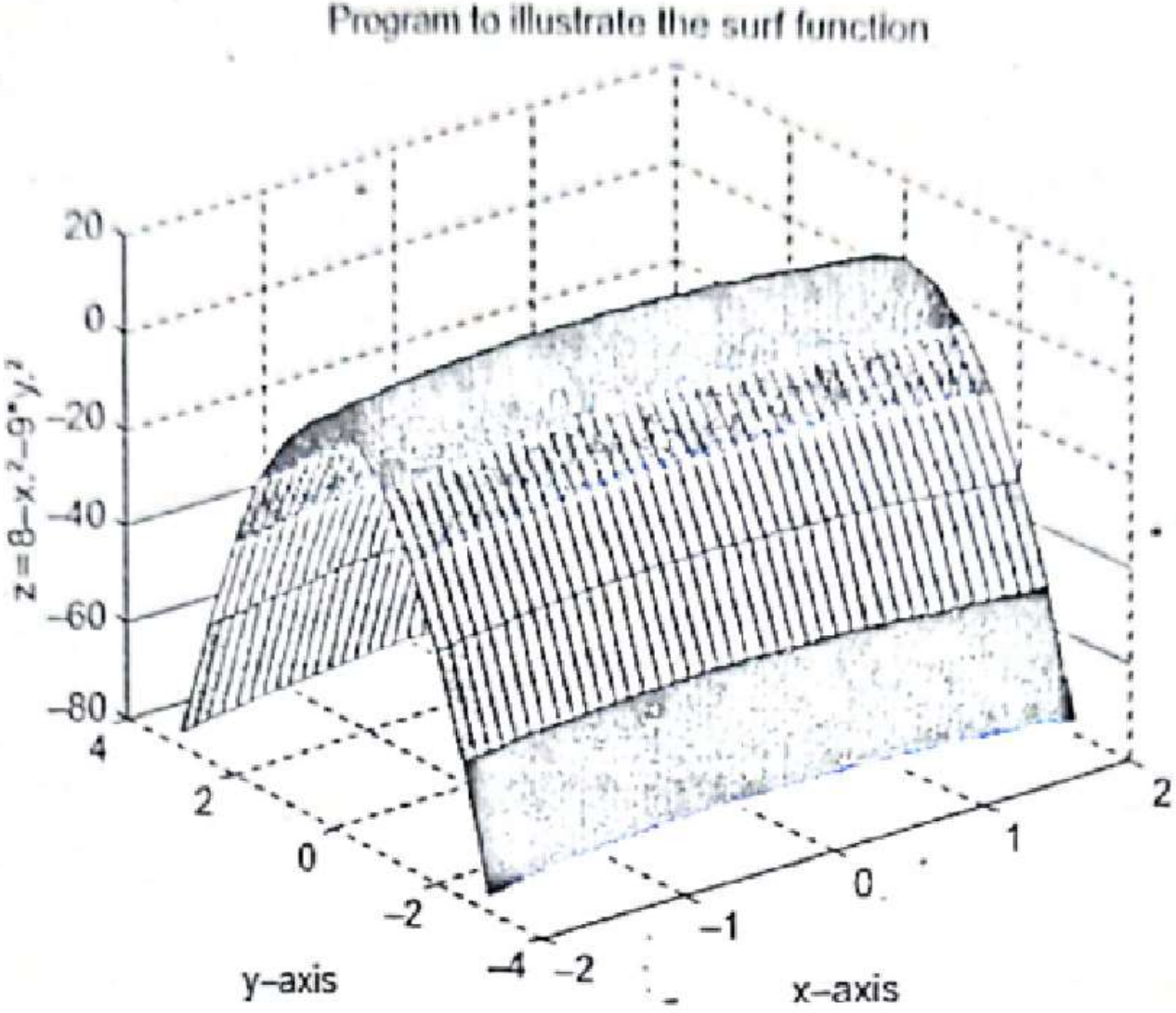


Figure 6.42 Plot to Illustrate `surface` Function

surf

Surface plot

Syntax

```
surf(X,Y,Z)
surf(X,Y,Z,C)
```

```
surf(Z)
surf(Z,C)
```

```
surf(ax, __ )
surf( __ ,Name,Value)
s = surf( __ )
```

Description

`surf(X,Y,Z)` creates a three-dimensional surface plot, which is a three-dimensional surface that has solid edge colors and solid face colors. The function plots the values in matrix `Z` as heights above a grid in the x-y plane defined by `X` and `Y`. The color of the surface varies according to the heights specified by `Z`.

`surf(X,Y,Z,C)` additionally specifies the surface color.

`surf(Z)` creates a surface plot and uses the column and row indices of the elements in `Z` as the x- and y-coordinates.

`surf(Z,C)` additionally specifies the surface color.

`surf(ax, __ )` plots into the axes specified by `ax` instead of the current axes. Specify the axes as the first input argument.

`surf( __ ,Name,Value)` specifies surface properties using one or more name-value pair arguments. For example, `'FaceAlpha',0.5` creates a semitransparent surface.

`s = surf( __ )` returns the chart surface object. Use `s` to modify the surface after it is created. For a list of properties, see Surface Properties.

Contour

6.8.9 `contour` and `contour3`

The `contour(z)` function gives the contour plot of the specified matrix treating the values given in the matrix as the heights above the plane.

The `contour (x, y, z)` is similar to `contour (z)` with the difference that `x` and `y` specifies `(x,y)` coordinates of the surface as for `surf` function.

The `contour3(x, y, z)` is the same as `contour` function except that the contours are drawn at their corresponding z level.

Example 6.42

Create a `contour` 3-D plot of the function `z=sqrt(y.^2-x.^2)` over the interval $-3 \leq x \leq 3$ and $-3 \leq y \leq 3$.

Solution:

The commands used are as follows:

```
%Program to illustrate the contour3 function
[x,y]=meshgrid(-3:0.5:3, -3:0.5:3);
z=sqrt(y.^2-x.^2);
contour3(x,y,z);
grid on;
title('Program to illustrate the contour3 function');
xlabel('x-axis');
ylabel ('y-axis');
zlabel('z=sqrt(y.^2-x.^2)'));
```

The response obtained is as shown in Figure 6.43. (See Figure 6.43 in coloured figure section.)

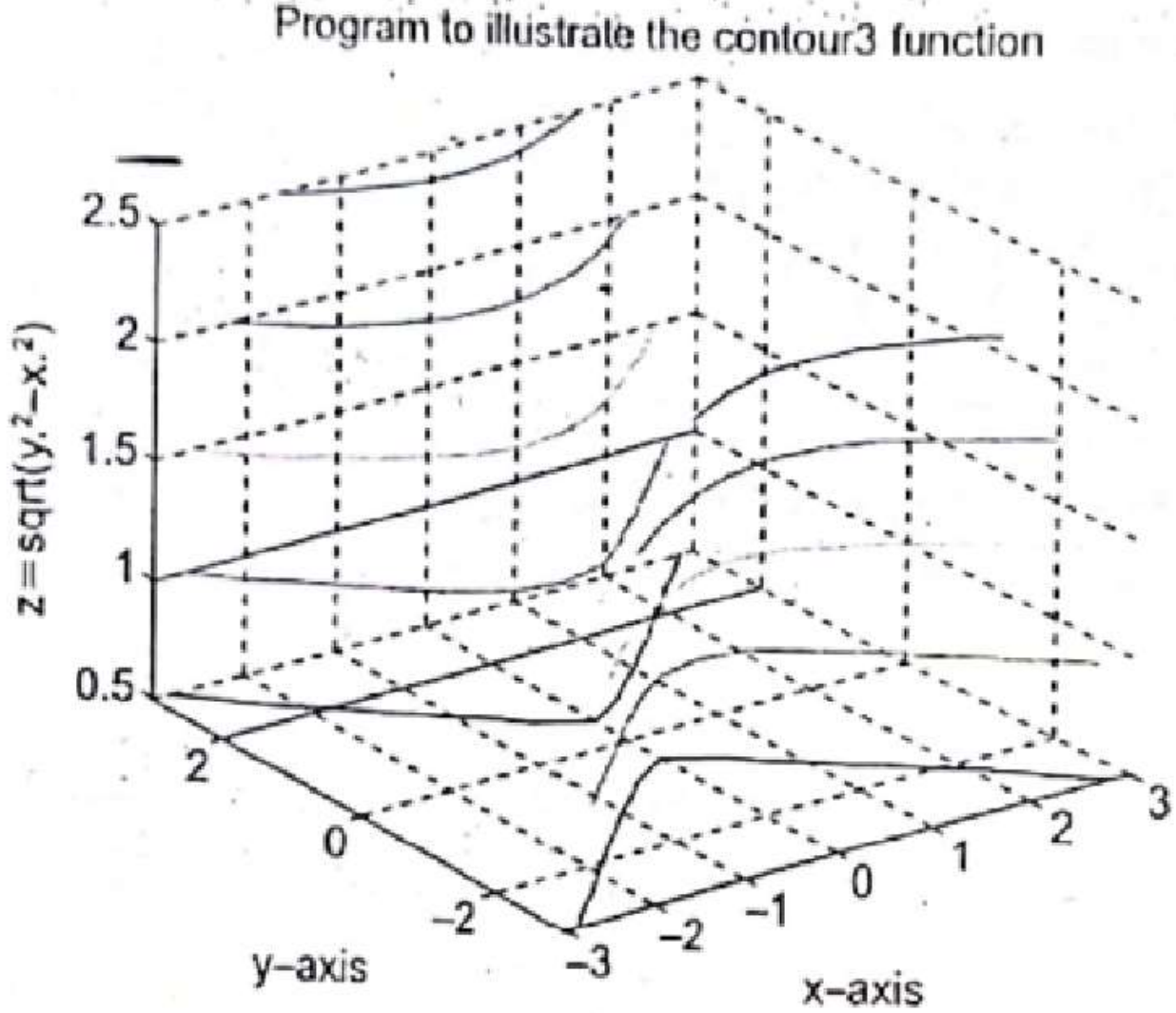


Figure 6.43 `contour3` Plot

Find

find

Find indices and values of nonzero elements

Syntax

```
k = find(X)
k = find(X,n)
k = find(X,n,direction)

[row,col] = find( __ )
[row,col,v] = find( __ )
```

Description

`k = find(X)` returns a vector containing the linear indices of each nonzero element in array `X`.

- If `X` is a vector, then `find` returns a vector with the same orientation as `X`.
- If `X` is a multidimensional array, then `find` returns a column vector of the linear indices of the result.

`k = find(X,n)` returns the first `n` indices corresponding to the nonzero elements in `X`.

`k = find(X,n,direction)`, where `direction` is `'last'`, finds the last `n` indices corresponding to nonzero elements in `X`. The default for `direction` is `'first'`, which finds the first `n` indices corresponding to nonzero elements.

`[row,col] = find( __ )` returns the row and column subscripts of each nonzero element in array `X` using any of the input arguments in previous syntaxes.

`[row,col,v] = find( __ )` also returns vector `v`, which contains the nonzero elements of `X`.

Vectorization

Vectorization

R2020b

Using Vectorization

MATLAB® is optimized for operations involving matrices and vectors. The process of revising loop-based, scalar-oriented code to use MATLAB matrix and vector operations is called *vectorization*. Vectorizing your code is worthwhile for several reasons:

- *Appearance*: Vectorized mathematical code appears more like the mathematical expressions found in textbooks, making the code easier to understand.
- *Less Error Prone*: Without loops, vectorized code is often shorter. Fewer lines of code mean fewer opportunities to introduce programming errors.
- *Performance*: Vectorized code often runs much faster than the corresponding code containing loops.

Vectorizing Code for General Computing

This code computes the sine of 1,001 values ranging from 0 to 10:

```
i = 0;
for t = 0:.01:10
    i = i + 1;
    y(i) = sin(t);
end
```

This is a vectorized version of the same code:

```
t = 0:.01:10;
y = sin(t);
```

Preallocation

Preallocation

R2020b

`for` and `while` loops that incrementally increase the size of a data structure each time through the loop can adversely affect performance and memory use. Repeatedly resizing arrays often requires MATLAB® to spend extra time looking for larger contiguous blocks of memory, and then moving the array into those blocks. Often, you can improve code execution time by preallocating the maximum amount of space required for the array.

The following code displays the amount of time needed to create a scalar variable, `x`, and then to gradually increase the size of `x` in a `for` loop.

```
tic
x = 0;
for k = 2:1000000
   x(k) = x(k-1) + 5;
end
toc

Elapsed time is 0.301528 seconds.
```

If you preallocate a 1-by-1,000,000 block of memory for `x` and initialize it to zero, then the code runs much faster because there is no need to repeatedly reallocate memory for the growing data structure.

```
tic
x = zeros(1, 1000000);
for k = 2:1000000
   x(k) = x(k-1) + 5;
end
toc

Elapsed time is 0.011938 seconds.
```