

Scilab Textbook Companion for
Numerical Methods For Scientific And
Engineering Computation
by M. K. Jain, S. R. K. Iyengar And R. K.
Jain¹

Created by
M Vamsi Krishan
B.Tech (pursuing)
Electronics Engineering
Visvesvaraya National Institute of Technology
College Teacher
Mr. Nagarajun
Cross-Checked by
Santosh Kumar, IITB

July 31, 2019

¹Funded by a grant from the National Mission on Education through ICT, <http://spoken-tutorial.org/NMEICT-Intro>. This Textbook Companion and Scilab codes written in it can be downloaded from the "Textbook Companion Project" section at the website <http://scilab.in>

Book Description

Title: Numerical Methods For Scientific And Engineering Computation

Author: M. K. Jain, S. R. K. Iyengar And R. K. Jain

Publisher: New Age International (P) Limited

Edition: 5

Year: 2007

ISBN: 8122420012

Scilab numbering policy used in this document and the relation to the above book.

Exa Example (Solved example)

Eqn Equation (Particular equation of the above book)

AP Appendix to Example(Scilab Code that is an Appednix to a particular Example of the above book)

For example, Exa 3.51 means solved example 3.51 of this book. Sec 2.3 means a scilab code whose theory is explained in Section 2.3 of the book.

Contents

List of Scilab Codes	4
2 TRANSCENDENTAL AND POLINOMIAL EQUATIONS	5
3 SYSTEM OF LINEAR ALGEBRIC EQUATIONS AND EIGENVALUE PROBLEMS	52
4 INTERPOLATION AND APPROXIMATION	65
5 DIFFERENTIATION AND INTEGRATION	83
6 ORDINARY DIFFERENTIAL EQUATIONS INNITAL VALUE PROBLEMS	97
7 ORDINARY DIFFERENTIAL EQUATIONS BOUNDARY VALUE PROBLEM	108

List of Scilab Codes

Exa 2.1	intervals containing the roots of the equation	5
Exa 2.2	interval containing the roots	7
Exa 2.3	solution to the eq by bisection method . . .	9
Exa 2.4	solution to the eq by bisection method . . .	11
Exa 2.5	solution to the given equation	14
Exa 2.6	solution by secant and regula falsi	16
Exa 2.7	solution to the equation by newton raphson method	19
Exa 2.8	solution to the equation by newton raphson method	21
Exa 2.9	solution to the equation by newton raphson method	23
Exa 2.11	solution to the given equation by muller method	25
Exa 2.12	solution by five iterations of muller method .	26
Exa 2.13	solution by chebeshev method	28
Exa 2.14	solution by chebeshev method	30
Exa 2.15	solution by chebeshev method	32
Exa 2.16	multipoint iteration	34
Exa 2.17	multipoint iteration	36
Exa 2.23	general iteration	38
Exa 2.24.1	solution by general iteration and aitken method	39
Exa 2.24.2	solution by general iteration and aitken method	42
Exa 2.25	solution by general iteration and aitken method	44
Exa 2.26	solution to the eq with multiple roots	46
Exa 2.27	solution to the given transcendental equation	49
Exa 3.1	determinant	52
Exa 3.2	property A in the book	53
Exa 3.4	solution to the system of equations	53

Exa 3.5	solution by gauss elimination method	54
Exa 3.6	solution by pivoted gauss elimination method	55
Exa 3.8	solution by pivoted gauss elimination method	56
Exa 3.9	solution using the inverse of the matrix	56
Exa 3.10	decomposition method	57
Exa 3.11	inverse using LU decoposition	57
Exa 3.12	solution by decomposition method	58
Exa 3.13	LU decomposition	59
Exa 3.14	cholesky method	60
Exa 3.15	cholesky method	61
Exa 3.21	jacobi iteration method	62
Exa 3.22	solution by gauss siedal method	63
Exa 3.27	eigen vale and eigen vector	63
Exa 4.3	linear interpolation polinomial	65
Exa 4.4	linear interpolation polinomial	66
Exa 4.6	legrange linear interpolation polinomial	67
Exa 4.7	polynomial of degree two	67
Exa 4.8	solution by quadratic interpolation	68
Exa 4.9	polynomial of degree two	69
Exa 4.15	forward and backward difference polynomial	69
Exa 4.20	hermite interpolation	70
Exa 4.21	piecewise linear interpolating polinomial	71
Exa 4.22	piecewise quadratic interpolating polinomial	72
Exa 4.23	piecewise cubical interpolating polinomial	73
Exa 4.31	linear approximation	74
Exa 4.32	linear polinomial approximation	75
Exa 4.34	least square straight fit	76
Exa 4.35	least square approximation	77
Exa 4.36	least square fit	77
Exa 4.37	least square fit	78
Exa 4.38	gram schmidt orthogonalisation	79
Exa 4.39	gram schmidt orthogonalisation	80
Exa 4.41	chebishev polinomial	82
Exa 5.1	linear interpolation	83
Exa 5.2	quadratic interpolation	84
Exa 5.10	jacobian matrix of the given system	84
Exa 5.11	solution by trapizoidal and simpsons	85

Exa 5.12	integral approximation by mid point and two point	85
Exa 5.13	integral approximation by simpson three eight rule	86
Exa 5.15	quadrature formula	87
Exa 5.16	gauss legendary three point method	88
Exa 5.17	gauss legendary method	88
Exa 5.18	integral approximation by gauss chebishev	89
Exa 5.20	integral approximation by gauss legurre method	90
Exa 5.21	integral approximation by gauss legurre method	91
Exa 5.22	integral approximation by gauss legurre method	91
Exa 5.26	composite trapizoidal and composite simpson	92
Exa 5.27	integral approximation by gauss legurre method	93
Exa 5.29	double integral using simpson rule	94
Exa 5.30	double integral using simpson rule	95
Exa 6.3	solution to the system of equations	97
Exa 6.4	solution ti the IVP	98
Exa 6.9	euler method to solve the IVP	99
Exa 6.12	solution ti IVP by back euler method	99
Exa 6.13	solution ti IVP by euler mid point method	100
Exa 6.15	solution ti IVP by taylor expansion	100
Exa 6.17	solution ti IVP by modified euler cauchy and heun	101
Exa 6.18	solution ti IVP by fourth order range kutta method	102
Exa 6.20	solution to the IVP systems	102
Exa 6.21	solution ti IVP by second order range kutta method	103
Exa 6.25	solution ti IVP by third order adamsbashfort meth	104
Exa 6.27	solution ti IVP by third order adams moult method	105
Exa 6.32	solution by numerov method	106
Exa 7.1	solution to the BVP by shooting method	108
Exa 7.3	solution to the BVP by shooting method	109
Exa 7.4	solution to the BVP by shooting method	110
Exa 7.5	solution to the BVP	111
Exa 7.6	solution to the BVP by finite differences	113

Exa 7.11 solution to the BVP by finite differences . . . 114

Chapter 2

TRANSCENDENTAL AND POLINOMIAL EQUATIONS

Scilab code Exa 2.1 intervals containing the roots of the equation

```
1                                     // The equation
                                     8*x^3-12*x
                                     ^2-2*x+3==0
                                     has three real
                                     roots.
2                                     // the graph of
                                     this function
                                     can be
                                     observed here.
3 xset('window',0);
4 x=-1:.01:2.5;
                                     //
   defining the range of x.
5 deff('[y]=f(x)', 'y=8*x^3-12*x^2-2*x+3');
                                     //defining the cunction
6 y=feval(x,f);
7
```

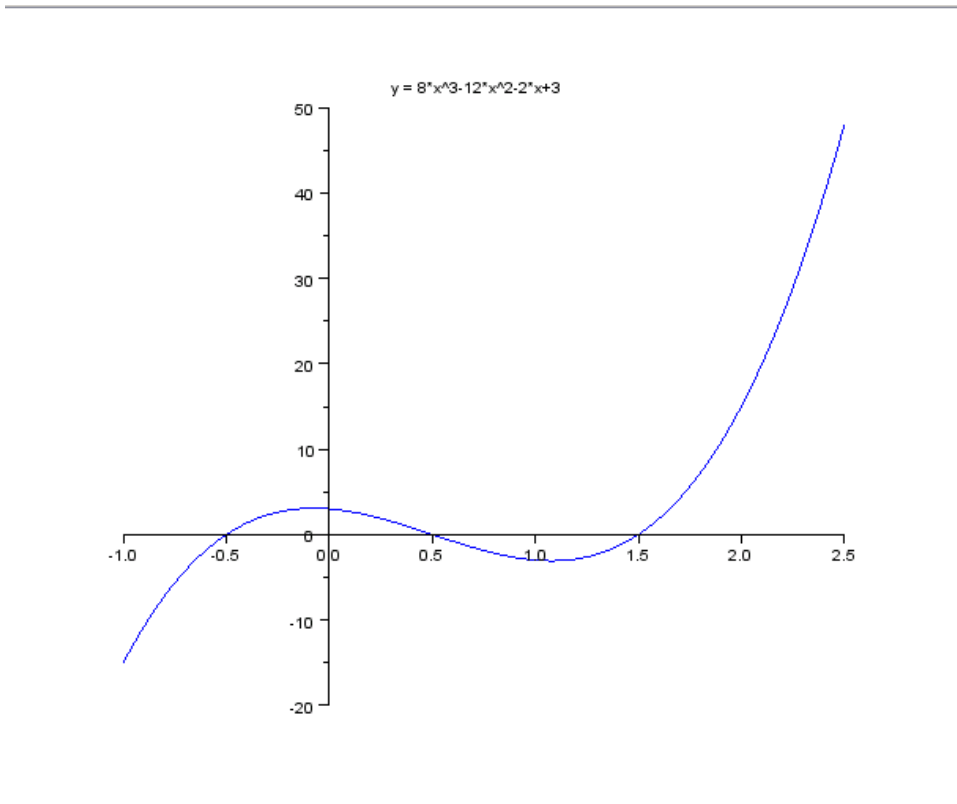


Figure 2.1: intervals containing the roots of the equation

```

8 a=gca();
9
10 a.y_location = "origin";
11
12 a.x_location = "origin";
13 plot(x,y)

    // instruction to plot the graph
14
15 title(' y = 8*x^3-12*x^2-2*x+3')
16
17 // from the above plot we can infre that the
    function has roots between
18 // the intervals (-1,0),(0,1),(1,2).

```

Scilab code Exa 2.2 interval containing the roots

```

1                                     // The equation
                                     cos(x)-x*%e^x
                                     ==0 has real
                                     roots.
2                                     // the graph of
                                     this function
                                     can be
                                     observed here.

3 xset('window',1);
4 x=0:.01:2;

                                     //
    defining the range of x.
5 deff(' [y]=f(x)', 'y=cos(x)-x*%e^x');
                                     //defining the cunction.
6 y=feval(x,f);
7

```

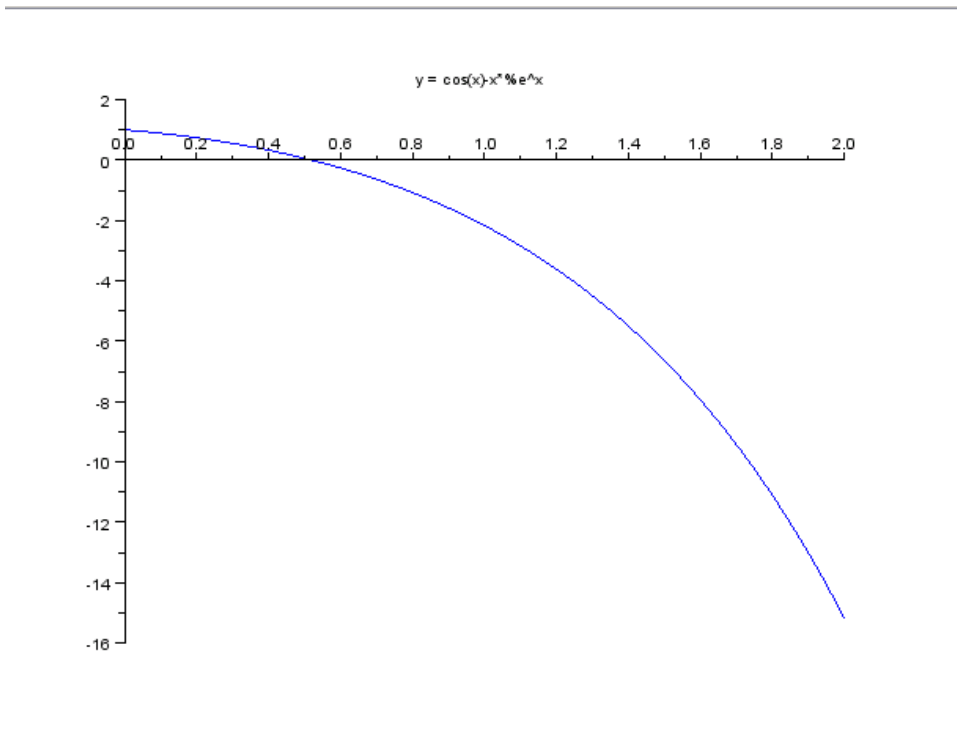


Figure 2.2: interval containing the roots

```

8 a=gca();
9
10 a.y_location = "origin";
11
12 a.x_location = "origin";
13 plot(x,y)

    // instruction to plot the graph
14 title(' y = cos(x)-x*%e^x')
15
16 // from the above plot we can infre that the
    function has root between
17 // the interval (0,1)

```

check Appendix [AP 49](#) for dependency:

Vbisection.sce

check Appendix [AP 48](#) for dependency:

Vbisection5.sce

Scilab code Exa 2.3 solution to the eq by bisection method

```

1                                     // The equation x
                                     ^3-5*x+1==0
                                     has real
                                     roots.
2                                     // the graph of
                                     this function
                                     can be
                                     observed here.

3 xset('window',2);
4 x=-2:.01:4;

                                     //
    defining the range of x.

```

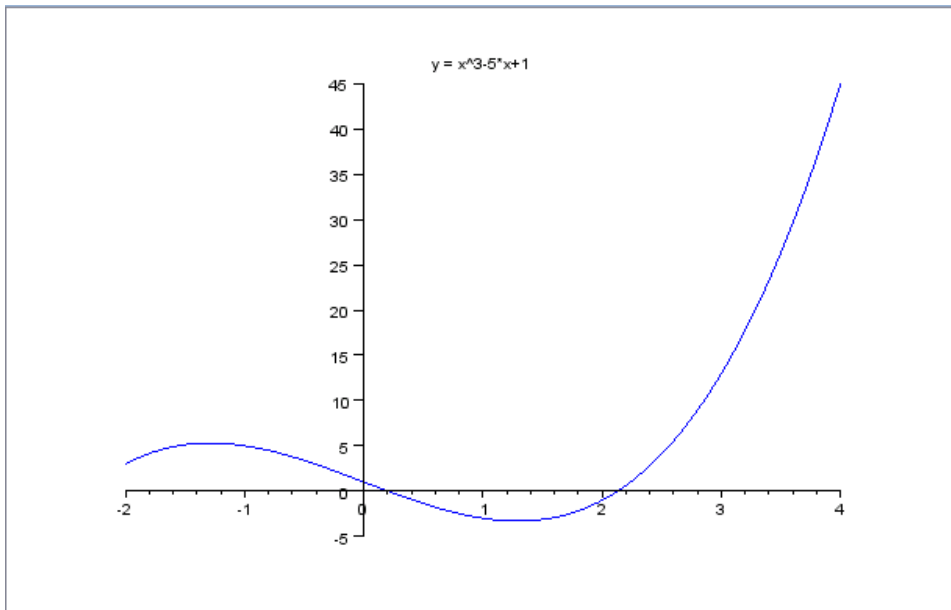


Figure 2.3: solution to the eq by bisection method

```

5 deff(' [y]=f(x) ', 'y=x^3-5*x+1'); //
   defining the function.
6 y=feval(x,f);
7
8 a=gca();
9
10 a.y_location = "origin";
11
12 a.x_location = "origin";
13 plot(x,y)

   // instruction to plot the graph
14 title(' y = x^3-5*x+1')
15
16 // from the above plot we can infre that the
   function has roots between
17 // the intervals (0,1) ,(2,3).
18 // since we have been asked for the smallest

```

```

    positive root of the equation ,
19 // we are intrested on the interval (0,1)
20 // a=0;b=1,
21
22 // we call a user-defined function 'bisection' so as
    to find the approximate
23 // root of the equation with a defined permissible
    error .
24
25 bisection(0,1,f)
26
27 // since in the example 2.3 we have been asked to
    perform 5 itterations
28 // the approximate root after 5 iterations can be
    observed .
29
30
31
32 bisection5(0,1,f)
33
34
35 // hence the approximate root after 5 iterations is
    0.203125 witin the permissible error of  $10^{-4}$ ,

```

check Appendix [AP 49](#) for dependency:

Vbisection.sce

check Appendix [AP 48](#) for dependency:

Vbisection5.sce

Scilab code Exa 2.4 solution to the eq by bisection method

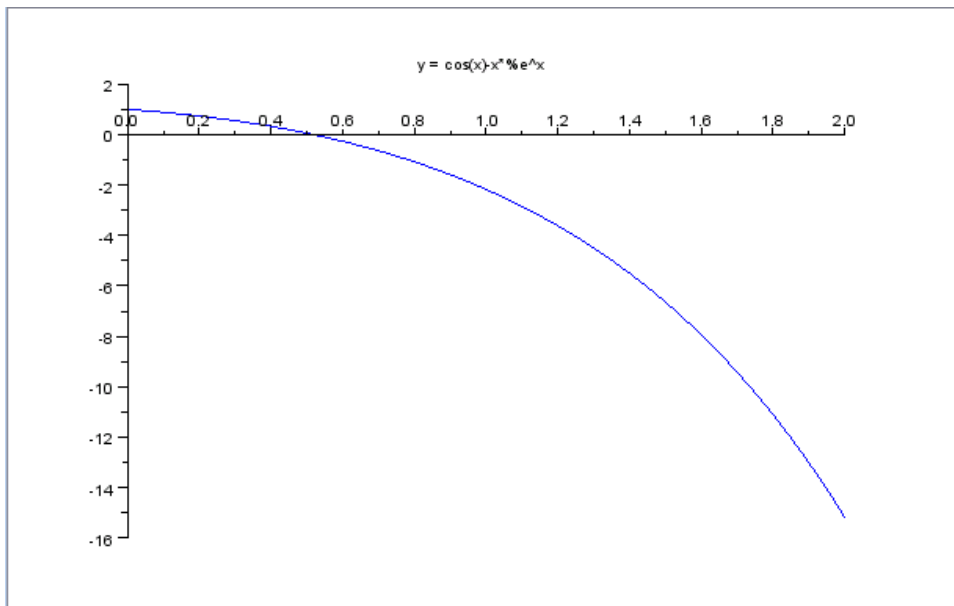


Figure 2.4: solution to the eq by bisection method

```

1                                     // The equation
                                     // cos(x)-x*%e^x
                                     // ==0 has real
                                     // roots.
2                                     // the graph of
                                     // this function
                                     // can be
                                     // observed here.

3 xset('window',3);
4 x=0:.01:2;

                                     //
                                     // defining the range of x.
5 deff(' [y]=f(x)', 'y=cos(x)-x*%e^x');
                                     //defining the cunction.
6 y=feval(x,f);
7
8 a=gca();
9
10 a.y_location = "origin";

```



```

11
12 a.x_location = "origin";
13 plot(x,y)

    // instruction to plot the graph
14 title(' y = cos(x)-x*%e^x')
15
16 // from the above plot we can infre that the
    function has root between
17 // the interval (0,1)
18
19
20 // a=0;b=1,
21
22 // we call a user-defined function 'bisection' so as
    to find the approximate
23 // root of the equation with a defined permissible
    error.
24
25 bisection(0,1,f)
26
27 // since in the example 2.4 we have been asked to
    perform 5 itterations ,
28
29 bisection5(0,1,f)
30
31
32 // hence the approximate root after 5 iterations is
    0.515625 witin the permissible error of 10^-4,

```

check Appendix [AP 46](#) for dependency:

regulafalsi4.sce

check Appendix [AP 47](#) for dependency:

secant4.sce

Scilab code Exa 2.5 solution to the given equation

```
1 // The equation  $x^3-5x+1=0$ 
// has real
// roots.
2 // the graph of
// this function
// can be
// observed here.
3 xset('window',4);
4 x=-2:.01:4;
//
// defining the range of x.
5 def('f(x)', 'y=x^3-5*x+1'); //
// defining the function.
6 y=fval(x,f);
7
8 a=gca();
9
10 a.y_location = "origin";
11
12 a.x_location = "origin";
13 plot(x,y)
// instruction to plot the graph
14 title('y = x^3-5*x+1')
15
16 // from the above plot we can infer that the
// function has roots between
17 // the intervals (0,1),(2,3).
18 // since we have been given the interval to be
// considered as (0,1)
19 // a=0;b=1,
20
21
22 // Solution by
// secant method
```

```

23
24
25
26
27
28 // since in the example 2.5 we have been asked to
    perform 4 itterations ,
29 secant4(0,1,f)           // we call a user-defined
    function 'bisection' so as to find the
    approximate
30 // root of the equation with a defined permissible
    error.
31
32
33
34 // hence the approximate root occurred in secant
    method after 4 iterations is 0.201640 witin the
    permissible error of  $10^{-4}$ ,
35
36
37
38 // solution by regular
    falsi method
39
40
41 // since in the example 2.5 we have been asked to
    perform 4 itterations ,
42
43 regulafalsi4(0,1,f)      // we call a user-
    defined function 'regularfalsi4' so as to find
    the approximate
44 // root of the equation with a defined permissible
    error.
45
46
47
48 // hence the approximate root occurred in
    regularfalsi method after 4 iterations is

```

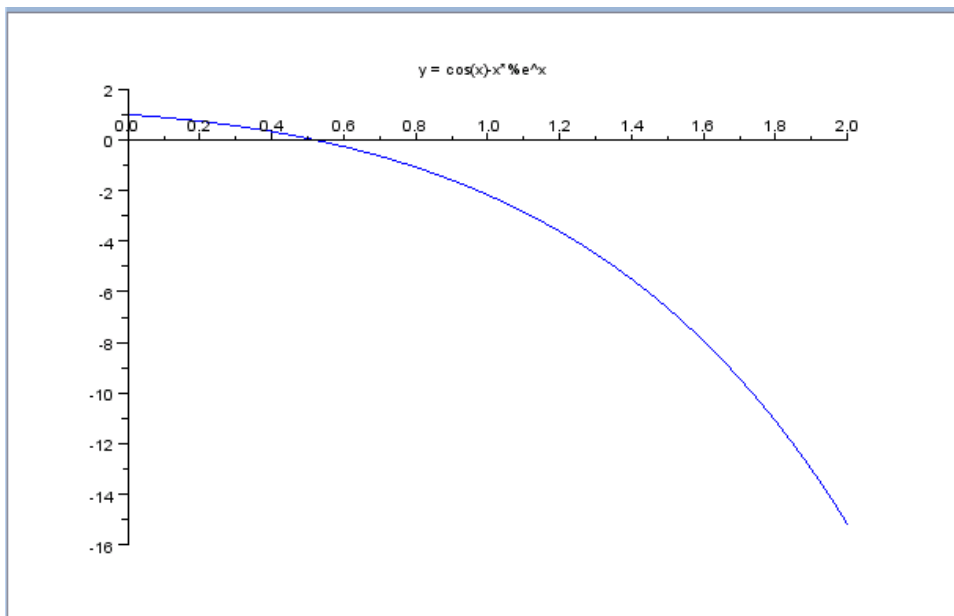


Figure 2.5: solution by secant and regula falsi

0.201640 within the permissible error of 10^{-4} ,

check Appendix AP 44 for dependency:

Vsecant.sce

check Appendix AP 45 for dependency:

regulafalsi.sce

Scilab code Exa 2.6 solution by secant and regula falsi

```
1 // The equation
   // cos(x) - x * e^x
   // == 0 has real
   // roots.
```

```

2                                     // the graph of
                                     // this function
                                     // can be
                                     // observed here.

3 xset('window',3);
4 x=0:.01:2;

                                     //
                                     // defining the range of x.
5 deff(' [y]=f(x) ', 'y=cos(x)-x*%e^x');
                                     //defining the cunction.
6 y=feval(x,f);
7
8 a=gca();
9
10 a.y_location = "origin";
11
12 a.x_location = "origin";
13 plot(x,y)

                                     // instruction to plot the graph
14 title(' y = cos(x)-x*%e^x')
15
16 // from the above plot we can infre that the
    // function has root between
17 // the interval (0,1)
18
19
20 // a=0;b=1,
21
22
23                                     // Solution by
                                     // secant method

24
25
26
27
28
29 // since in the example 2.6 we have no specification

```

```

    of the no. of itterations ,
30 // we define a function 'secant' and execute it.
31
32
33
34 secant(0,1,f)           // we call a user-defined
    function 'secant' so as to find the approximate
35 // root of the equation with a defined permissible
    error.
36
37
38
39 // hence the approximate root occured in secant
    method witin the permissible error of  $10^{-5}$  is ,
40
41
42
43
    // solution by regular
    falsi method
44
45
46
47
48
49 // since in the example 2.6 we have no specification
    of the no. of itterations ,
50
51
52 regulafalsi(0,1,f)     // we call a user-
    defined function 'regularfalsi' so as to find the
    approximate
53 // root of the equation with a defined permissible
    error.

```

check Appendix [AP 43](#) for dependency:

Vnewton4.sce

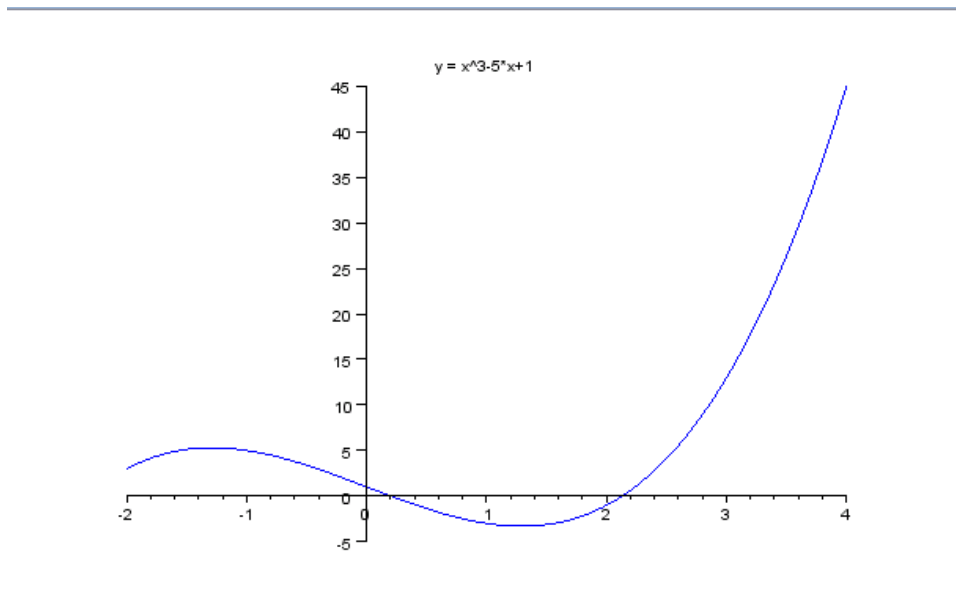


Figure 2.6: solution to the equation by newton raphson method

Scilab code Exa 2.7 solution to the equation by newton raphson method

```

1           // The equation x
           // ^3-5*x+1==0
           // has real
           // roots.
2           // the graph of
           // this function
           // can be
           // observed here.

3  xset('window',6);
4  x=-2:.01:4;
           //

```

```

    defining the range of x.
5  deff(' [y]=f(x) ', 'y=x^3-5*x+1'); //
    defining the function.
6  deff(' [y]=fp(x) ', 'y=3*x^2-5');
7  y=feval(x,f);
8
9  a=gca();
10
11 a.y_location = "origin";
12
13 a.x_location = "origin";
14 plot(x,y)

    // instruction to plot the graph
15 title(' y = x^3-5*x+1')
16
17 // from the above plot we can infre that the
    function has roots between
18 // the intervals (0,1),(2,3).
19 // since we have been asked for the smallest
    positive root of the equation,
20 // we are intrested on the interval (0,1)
21 // a=0;b=1,
22
23 // since in the example 2.7 we have been asked to
    perform 4 itterations ,
24 // the approximate root after 4 iterations can be
    observed.
25
26
27 newton4(0.5,f,fp)
28
29
30 // hence the approximate root after 4 iterations is
    0.201640 witin the permissible error of  $10^{-15}$ ,

```

check Appendix AP 43 for dependency:

Vnewton4.sce

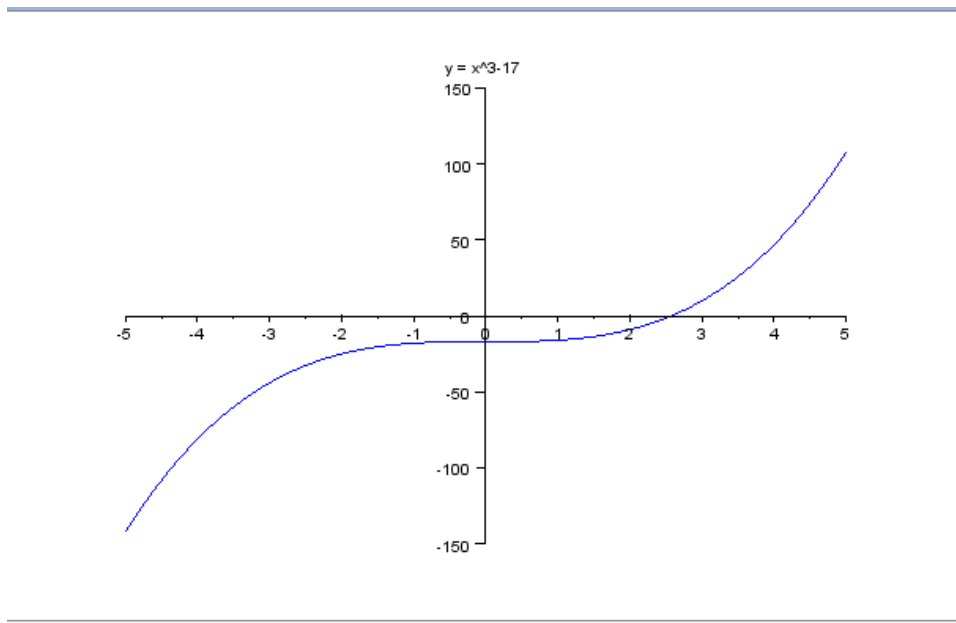


Figure 2.7: solution to the equation by newton raphson method

Scilab code Exa 2.8 solution to the equation by newton raphson method

```

1           // The equation  $x^3 - 17 = 0$  has
           // three real
           // roots.
2           // the graph of
           // this function
           // can be
           // observed here.

3  xset('window',7);
4  x=-5:.001:5;
           //

```

```

    defining the range of x.
5  deff(' [y]=f(x) ', 'y=x^3-17');           //
    defining the cunction.
6  deff(' [y]=fp(x) ', 'y=3*x^2');
7  y=feval(x,f);
8
9  a=gca();
10
11 a.y_location = "origin";
12
13 a.x_location = "origin";
14 plot(x,y)

    // instruction to plot the graph
15 title(' y = x^3-17')
16
17 // from the above plot we can infre that the
    function has root between
18 // the interval (2,3).
19
20
21
22         //solution by newton raphson 's method
23
24
25
26 // since in example no.2.8 we have been asked to
    perform 4 iterations ,we define a fuction
    newton4'' which does newton raphson's method of
    finding approximate root upto 4 iterations ,
27
28
29
30 newton4(2,f,fp)           //calling the pre-
    defined function 'newton4'.

```

check Appendix [AP 42](#) for dependency:

Vnewton.sce

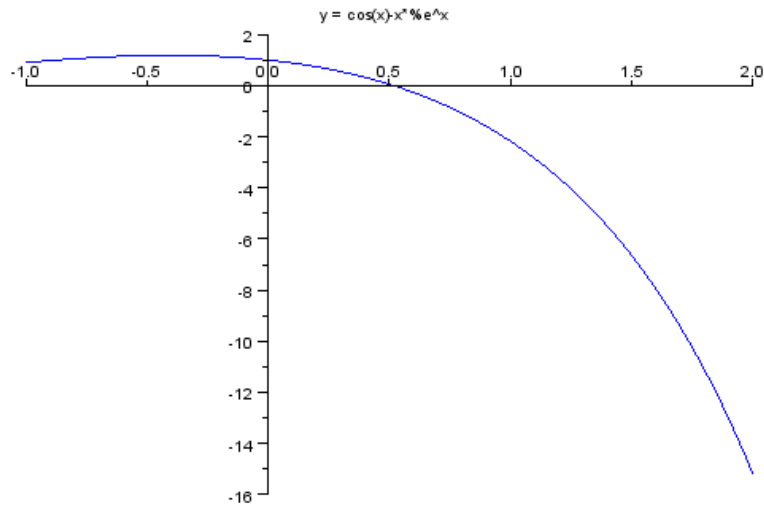


Figure 2.8: solution to the equation by newton raphson method

Scilab code Exa 2.9 solution to the equation by newton raphson method

```

1                                     // The equation
                                     // cos(x)-x*%e^x
                                     // ==0 has real
                                     // roots.
2                                     // the graph of
                                     // this function
                                     // can be
                                     // observed here.

3 xset('window',8);
4 x=-1:.001:2;
                                     //

```

```

    defining the range of x.
5  deff( '[y]=f(x) ', 'y=cos(x)-x*%e^x ');
    //defining the function.
6  deff( '[y]=fp(x) ', 'y=-sin(x)-x*%e^x-%e^x ');
7  y=feval(x,f);
8
9  a=gca();
10
11 a.y_location = "origin";
12
13 a.x_location = "origin";
14 plot(x,y)

    // instruction to plot the graph
15 title(' y = cos(x)-x*%e^x ')
16
17 // from the above plot we can infre that the
    function has root between
18 // the interval (0,1)
19
20
21 // a=0;b=1,
22
23
24
25         // solution by newton raphson's method
            with a permissible error of  $10^{-8}$ .
26
27
28 // we call a user-defined function 'newton' so as to
    find the approximate
29 // root of the equation within the defined
    permissible error limit.
30
31 newton(1,f,fp)
32
33
34

```

```

35
36
37 // hence the approximate root within the permissible
    error of  $10^{-8}$  is 0.5177574.

```

check Appendix AP 61 for dependency:

muller3.sce

Scilab code Exa 2.11 solution to the given equation by muller method

```

1           // The equation  $x^3 - 5x + 1 = 0$  has
                real roots.
2           // the graph of this
                function can be
                observed here.
3  xset('window',10);
4  x=-2:.01:4;                                     //
                defining the range of x.
5  deff('[y]=f(x)', 'y=x^3-5*x+1');               //
                defining the function.
6  y=feval(x,f);
7
8  a=gca();
9
10 a.y_location = "origin";
11
12 a.x_location = "origin";
13 plot(x,y)                                       //
                instruction to plot the graph
14 title(' y = x^3-5*x+1')
15
16 // from the above plot we can infer that the
                function has roots between
17 // the intervals (0,1),(2,3).

```

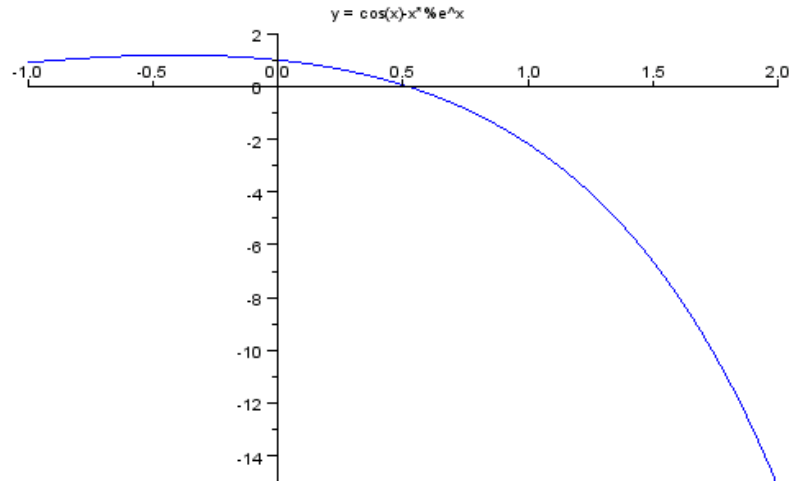


Figure 2.9: solution by five iterations of muller method

```

18 // since we have been asked for the smallest
    positive root of the equation ,
19 // we are intrested on the interval (0,1)
20
21
22 //sollution by muller method to 3 iterations
    .
23
24 muller3(0,.5,1,f)

```

check Appendix [AP 60](#) for dependency:

muller5.sce

Scilab code Exa 2.12 solution by five itrations of muller method

```

1                                     // The equation
                                     cos(x)-x*%e^x
                                     ==0 has real
                                     roots.
2                                     // the graph of
                                     this function
                                     can be
                                     observed here.

3 xset('window',8);
4 x=-1:.001:2;

                                     //
                                     defining the range of x.
5 deff(' [y]=f(x)', 'y=cos(x)-x*%e^x');
                                     //defining the function.
6 deff(' [y]=fp(x)', 'y=-sin(x)-x*%e^x-%e^x');
7 y=feval(x,f);
8
9 a=gca();
10
11 a.y_location = "origin";
12
13 a.x_location = "origin";
14 plot(x,y)

                                     // instruction to plot the graph
15 title(' y = cos(x)-x*%e^x')
16
17 // from the above plot we can infre that the
                                     function has root between
18 // the interval (0,1)
19
20
21                                     //sollution by muller method to 5 iterations
22
23
24 muller5(-1,0,1,f)

```

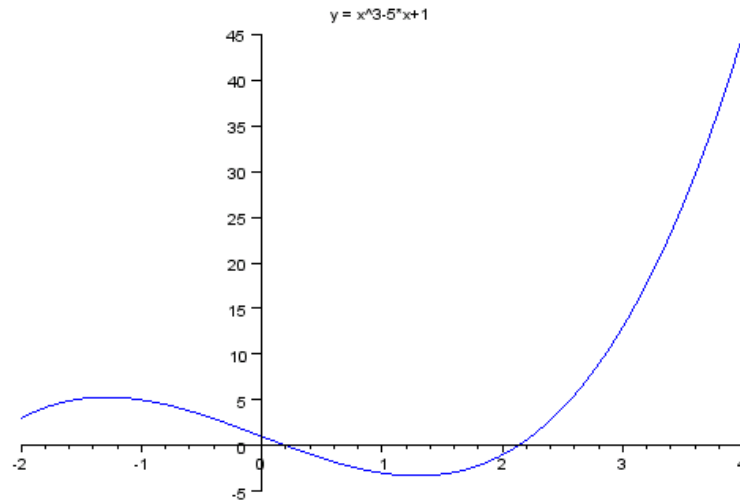


Figure 2.10: solution by chebeshev method

check Appendix [AP 59](#) for dependency:

chebyshev.sce

Scilab code Exa 2.13 solution by chebeshev method

```

1           // The equation x
           // ^3-5*x+1==0
           // has real
           // roots.
2           // the graph of
           // this function
           // can be
           // observed here.
3  xset('window',12);
```



```

4 x=-2:.01:4;
//
    defining the range of x.
5 deff(' [y]=f(x) ', 'y=x^3-5*x+1'); //
    defining the function.
6 deff(' [y]=fp(x) ', 'y=3*x^2-5');
7 deff(' [y]=fpp(x) ', 'y=6*x');
8 y=feval(x,f);
9
10 a=gca();
11
12 a.y_location = "origin";
13
14 a.x_location = "origin";
15 plot(x,y)

    // instruction to plot the graph
16 title(' y = x^3-5*x+1')
17
18 // from the above plot we can infre that the
    function has roots between
19 // the intervals (0,1) ,(2,3).
20 // since we have been asked for the smallest
    positive root of the equation ,
21 // we are intrested on the interval (0,1)
22 // a=0;b=1,
23
24
25 //          solution by chebyshev method
26
27 // the approximate root after 4 iterations can be
    observed.
28
29
30 chebyshev(0.5,f,fp)
31
32
33 // hence the approximate root witin the permissible

```

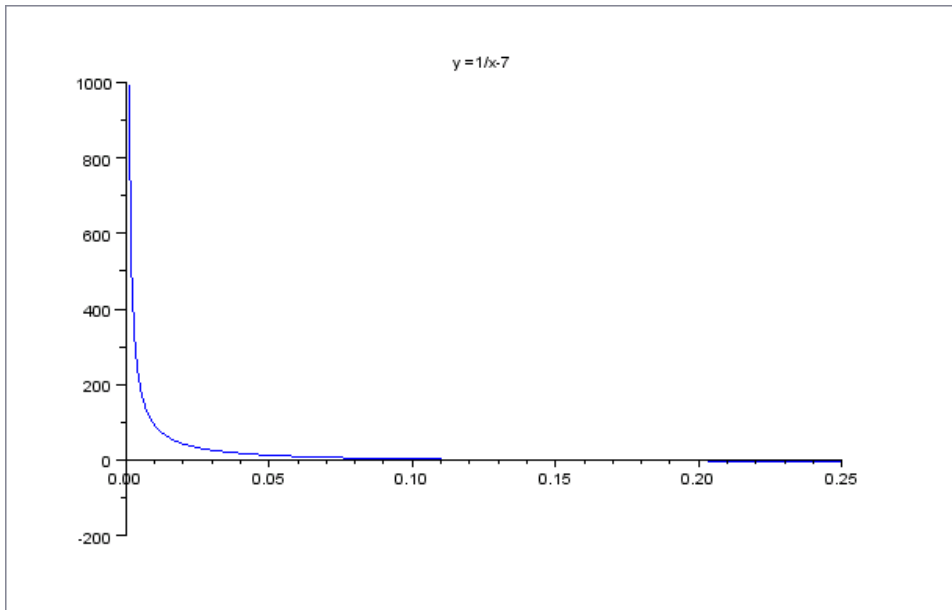


Figure 2.11: solution by chebeshev method

error of 10^{-15} is .2016402,

check Appendix [AP 59](#) for dependency:

chebyshev.sce

Scilab code Exa 2.14 solution by chebeshev method

```

1
2
3                                     // The equation
                                     // 1/x-7==0 has a
                                     // real root.
4                                     // the graph of
                                     // this function

```

can be
observed here.

```
5 xset('window',13);
6 x=0.001:.001:.25;

    defining the range of x.
7 deff(' [y]=f(x) ', 'y=1/x-7');
    defining the function.
8 deff(' [y]=fp(x) ', 'y=-1/x^2');
9 y=feval(x,f);
10
11 a=gca();
12
13 a.y_location = "origin";
14
15 a.x_location = "origin";
16 plot(x,y)

    // instruction to plot the graph
17 title(' y =1/x-7')
18
19 // from the above plot we can infre that the
    function has roots between
20 // the interval (0,2/7)
21
22
23         //solution by chebyshev method
24
25
26 chebyshev(0.1,f,fp) //calling the
    pre-defined function 'chebyshev' to find the
    approximate root in the range of (0,2/7).
```

check Appendix [AP 59](#) for dependency:

chebyshev.sce

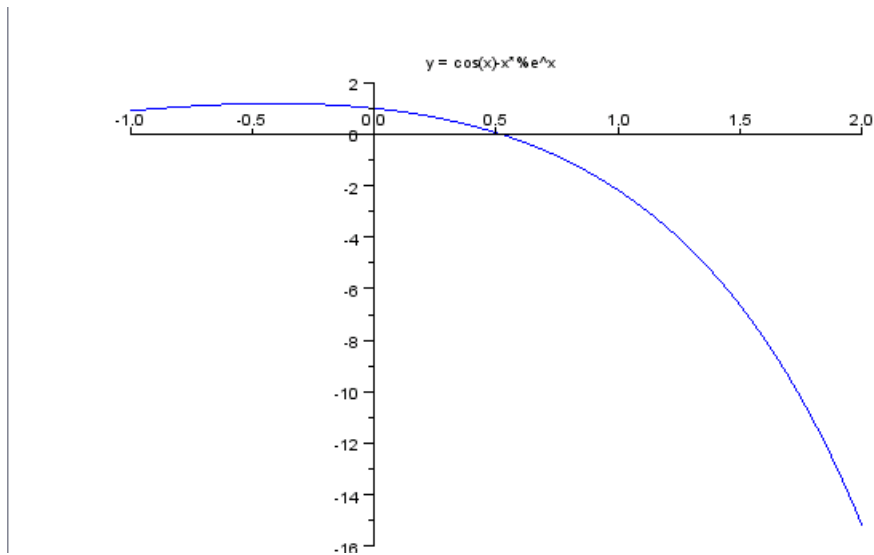


Figure 2.12: solution by chebeshev method

Scilab code Exa 2.15 solution by chebeshev method

```

1                                     // The equation
                                     // cos(x)-x*%e^x
                                     // ==0 has real
                                     // roots.
2                                     // the graph of
                                     // this function
                                     // can be
                                     // observed here.

3 xset('window',8);
4 x=-1:.001:2;

                                     //
                                     // defining the range of x.
5 deff(' [y]=f(x) ', 'y=cos(x)-x*%e^x ');
                                     //defining the function.
6 deff(' [y]=fp(x) ', 'y=-sin(x)-x*%e^x-%e^x ');

```

```

7  deff(' [y]=fpp(x)', 'y=-cos(x)-x*%e^x-2*%e^x');
8  y=feval(x,f);
9
10 a=gca();
11
12 a.y_location = "origin";
13
14 a.x_location = "origin";
15 plot(x,y)

    // instruction to plot the graph
16 title(' y = cos(x)-x*%e^x ')
17
18 // from the above plot we can infre that the
    function has root between
19 // the interval (0,1)
20
21
22 // a=0;b=1,
23
24
25
26          // solution by chebyshev with a
    permissible error of 10^-15.
27
28 // we call a user-defined function 'chebyshev' so as
    to find the approximate
29 // root of the equation within the defined
    permissible error limit.
30
31 chebyshev(1,f,fp)
32
33
34
35 // hence the approximate root witin the permissible
    error of 10^-15 is

```

check Appendix AP 58 for dependency:

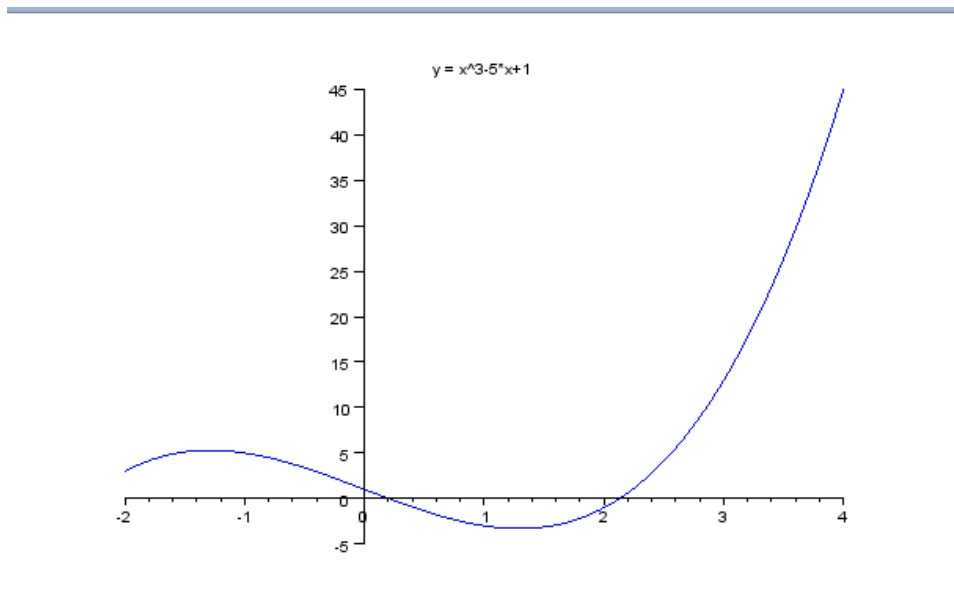


Figure 2.13: multipoint iteration

`multipoint_iteration31.sce`

check Appendix [AP 57](#) for dependency:

`multipoint_iteration33.sce`

Scilab code Exa 2.16 multipoint iteration

```

1           // The equation  $x^3 - 5x + 1 = 0$ 
           // has real
           // roots.
2           // the graph of
           // this function
           // can be
           // observed here.

```

```

3  xset('window',15);
4  x=-2:.01:4;

                                     //
      defining the range of x.
5  deff(' [y]=f(x)', 'y=x^3-5*x+1');      //
      defining the function.
6  deff(' [y]=fp(x)', 'y=3*x^2-5');
7  deff(' [y]=fpp(x)', 'y=6*x');
8  y=feval(x,f);
9
10 a=gca();
11
12 a.y_location = "origin";
13
14 a.x_location = "origin";
15 plot(x,y)

      // instruction to plot the graph
16 title(' y = x^3-5*x+1')
17
18 // from the above plot we can infre that the
      function has roots between
19 // the intervals (0,1),(2,3).
20 // since we have been asked for the smallest
      positive root of the equation ,
21 // we are intrested on the interval (0,1)
22 // a=0;b=1,
23
24
25 //          solution by multipoint iteration
      method
26
27 // the approximate root after 3 iterations can be
      observed.
28
29
30 multipoint_iteration31(0.5,f,fp)
31

```

```

32 // hence the approximate root witin the permissible
    error of  $10^{-15}$  is .201640,
33
34
35
36 multipoint_iteration33(0.5,f,fp)
37
38 // hence the approximate root witin the permissible
    error of  $10^{-15}$  is .201640,

```

check Appendix AP 57 for dependency:

multipoint_iteration33.sce

Scilab code Exa 2.17 multipoint iteration

```

1                                     // The equation
                                     cos(x)-x*%e^x
                                     ==0 has real
                                     roots.
2                                     // the graph of
                                     this function
                                     can be
                                     observed here.

3 xset('window',8);
4 x=-1:.001:2;

                                     //
    defining the range of x.
5 deff(' [y]=f(x) ', 'y=cos(x)-x*%e^x ');
                                     //defining the function.
6 deff(' [y]=fp(x) ', 'y=-sin(x)-x*%e^x-%e^x ');
7 deff(' [y]=fpp(x) ', 'y=-cos(x)-x*%e^x-2*%e^x ');
8 y=feval(x,f);
9
10 a=gca();
11

```



```

12 a.y_location = "origin";
13
14 a.x_location = "origin";
15 plot(x,y)

    // instruction to plot the graph
16 title(' y = cos(x)-x*%e^x ')
17
18 // from the above plot we can infre that the
    function has root between
19 // the interval (0,1)
20
21
22 // a=0;b=1,
23
24
25
26          // solution by multipoint_iteration
    method using the formula given in
    equation no.2.33.

27
28 // we call a user-defined function '
    multipoint_iteration33 ' so as to find the
    approximate
29 // root of the equation within the defined
    permissible error limit.
30
31 multipoint_iteration33(1,f,fp)
32
33
34 // hence the approximate root witin the permissible
    error of  $10^{-5}$  is 0.5177574.

```

check Appendix [AP 56](#) for dependency:

generaliteration.sce

Scilab code Exa 2.23 general iteration

```
1 // The equation
// 3*x^3+4*x^2+4*
// x+1==0 has
// three real
// roots.
2 // the graph of
// this function
// can be
// observed here.

3 xset('window',22);
4 x=-1.5:.001:1.5;

//
// defining the range of x.
5 def(' [y]=f(x) ', 'y=3*x^3+4*x^2+4*x+1');
//defining the cunction
6 y=feval(x,f);
7
8 a=gca();
9
10 a.y_location = "origin";
11
12 a.x_location = "origin";
13 plot(x,y)

// instruction to plot the graph
14
15 title(' y =3*x^3+4*x^2+4*x+1')
16
17 // from the above plot we can infre that the
// function has root between
18 // the interval (-1,0),
19
20 x0=-.5; // initial approximation
21
22
23 // let the iterative function g(x) be x+A*(3*x^3+4*x
```

```

      ^2+4*x+1) =g(x);
24
25 // gp(x)=(1+A*(9*x^2+8*x+4) )
26 // we need to choose a value for A , which makes abs
      (gp(x0))<1
27
28 // hence abs(gp(x0))=abs(1+9*A/4)
29
30 A=-1:.1:1;
31
32 abs(1+9*A/4)           // tryin to check the values
      of abs(gp(x0)) for different values of A.
33
34
35 // from the above values of 'A' and the values of '
      abs(gp(x0))',
36 // we can infer that for the vales of 'A 'in the
      range (-.8,0) g(x ) will be giving a converging
      solution ,
37
38 // hence deliberatele we choose a to be -0.5,
39
40 A=-0.5;
41
42 deff(' [y]=g(x) ', 'y= x-0.5*(3*x^3+4*x^2+4*x+1) ');
43 deff(' [y]=gp(x) ', 'y= 1-0.5*(9*x^2+8*x+4) '); //
      hence defining g(x) and gp(x) ,
44 generaliteration(x0,g,gp)

```

check Appendix [AP 55](#) for dependency:

aitken.sce

Scilab code Exa 2.24.1 solution by general iteration and aitken method

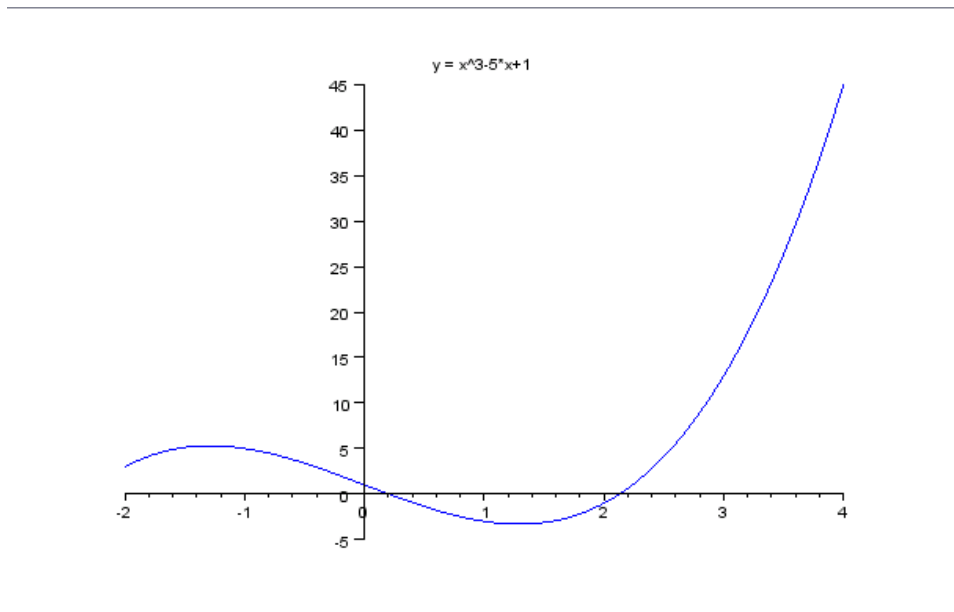


Figure 2.14: solution by general iteration and aitken method

```

1                                     // The equation  $x^3 - 5x + 1 = 0$  has
                                     // real roots.
2                                     // the graph of this
                                     // function can be
                                     // observed here.
3 xset('window',2);
4 x = -2:.01:4;                                     //
   // defining the range of x.
5 def ('[y]=f(x)', 'y=x^3-5*x+1');                 //
   // defining the function.
6 y = feval(x,f);
7
8 a = gca();
9
10 a.y_location = "origin";
11
12 a.x_location = "origin";

```

```

13 plot(x,y) //
    instruction to plot the graph
14 title(' y = x^3-5*x+1')
15
16 // from the above plot we can infre that the
    function has roots between
17 // the intervals (0,1),(2,3).
18 // since we have been asked for the smallest
    positive root of the equation,
19 // we are intrested on the interval (0,1)
20
21 x0=.5;
22
23 //solution using linear iteration method
    for the first two iterations and aitken's
    process two times for the third
    iteration.
24
25 deff(' [y]=g(x) ', 'y=1/5*(x^3+1) ');
26 deff(' [y]=gp(x) ', 'y=1/5*(3*x^2) ');
27
28
29 generaliteration2(x0,g,gp)
30
31
32 // from the above iterations performed we can infer
    that-
33 x1=0.225;
34 x2=0.202278;
35
36
37
38
39 aitken(x0,x1,x2,g) // calling the aitken
    method for one iteration

```

check Appendix [AP 54](#) for dependency:

generaliteration2.sce

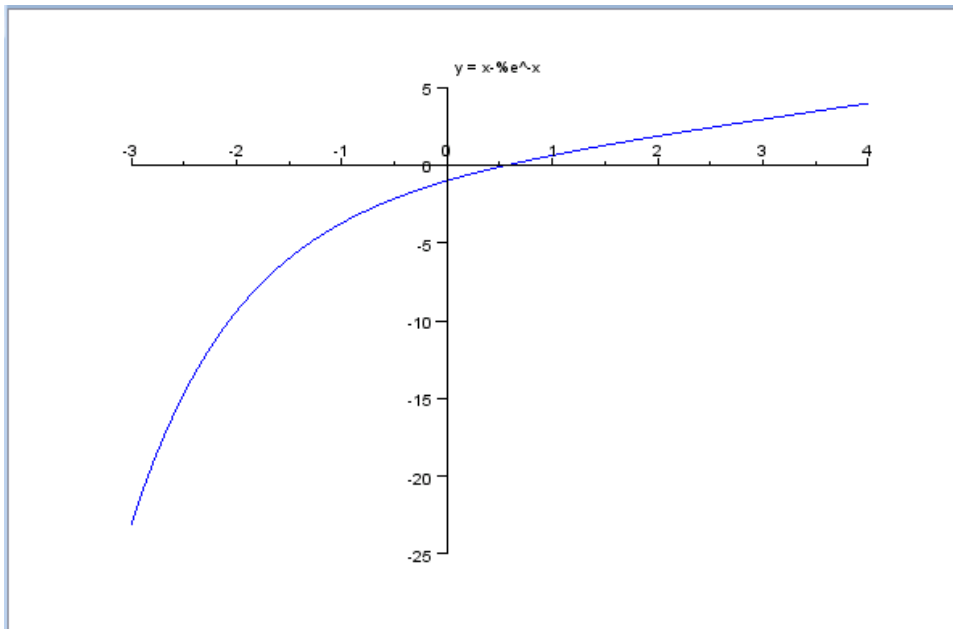


Figure 2.15: solution by general iteration and aitken method

check Appendix [AP 55](#) for dependency:

aitken.sce

Scilab code Exa 2.24.2 solution by general iteration and aitken method

```

1           // The equation  $x - e^{-x} = 0$  has real roots.
2           // the graph of this
              function can be
              observed here.
3  xset('window', 24);
```

```

4 x=-3:.01:4;
//
// defining the range of x.
5 deff(' [y]=f(x) ', 'y=x-%e^-x');
//
// defining the cunction.
6 y=feval(x,f);
7
8 a=gca();
9
10 a.y_location = "origin";
11
12 a.x_location = "origin";
13 plot(x,y)
// instruction to plot the graph
14 title(' y = x-%e^-x ')
15
16 // from the above plot we can infre that the
// function has root between
17 // the interval (0,1)
18
19 x0=1;
20
21 //solution using linear iteration method
// for the first two iterations and aitken
// 's process two times for the third
// iteration.
22
23
24
25 deff(' [y]=g(x) ', 'y=%e^-x ');
26 deff(' [y]=gp(x) ', 'y=-%e^-x ');
27
28
29 generaliteration2(x0,g,gp)
30
31
32 // from the above iterations performed we can infer

```

```

    that -
33 x1=0.367879;
34 x2=0.692201;
35
36
37
38
39 aitken(x0,x1,x2,g) // calling the aitken
    method for one iteration

```

check Appendix [AP 54](#) for dependency:

generaliteration2.sce

check Appendix [AP 55](#) for dependency:

aitken.sce

check Appendix [AP 54](#) for dependency:

generaliteration2.sce

Scilab code Exa 2.25 solution by general iteration and aitken method

```

1 // The equation
    cos(x)-x*%e^x
    ==0 has real
    roots.
2 // the graph of
    this function
    can be
    observed here.
3 xset('window',25);
4 x=0:.01:2;
    //
    defining the range of x.

```

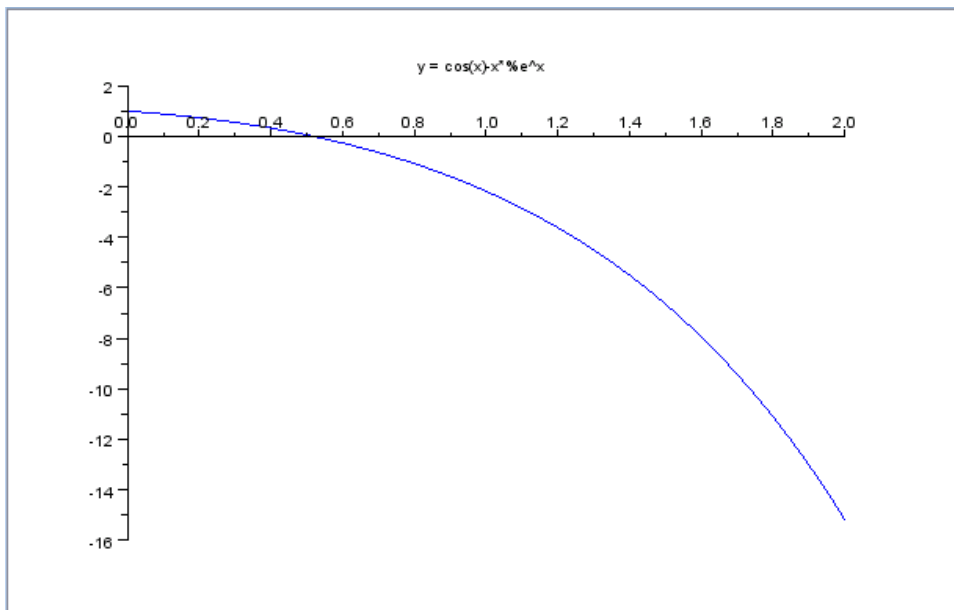



Figure 2.16: solution by general iteration and aitken method

```

5  deff ( ' [y]=f(x) ', 'y=cos(x)-x*%e^x' );
                                     //defining the function.
6  y=feval(x,f);
7
8  a=gca();
9
10 a.y_location = "origin";
11
12 a.x_location = "origin";
13 plot(x,y)

    // instruction to plot the graph
14 title(' y = cos(x)-x*%e^x ')
15
16 // from the above plot we can infre that the
    function has root between
17 // the interval (0,1)
18

```

```

19 x0=0;
20
21 //solution using linear iteration method
    for the first two iterations and aitken'
    s process two times for the third
    iteration.
22
23 def f(' [y]=g(x) ', 'y=x+1/2*(cos(x)-x*%e^x) ');
24 def f(' [y]=gp(x) ', 'y=1+1/2*(-sin(x)-x*%e^x-
    %e^x) ');
25
26
27 generaliteration2(x0,g,gp)
28
29
30 // from the above iterations performed we can infer
    that-
31 x1=0.50000000;
32 x2=0.5266110;
33
34
35
36 aitken(x0,x1,x2,g) // calling the aitken
    method for one iteration

```

check Appendix [AP 42](#) for dependency:

Vnewton.sce

check Appendix [AP 53](#) for dependency:

modified_newton.sce

Scilab code Exa 2.26 solution to the eq with multiple roots

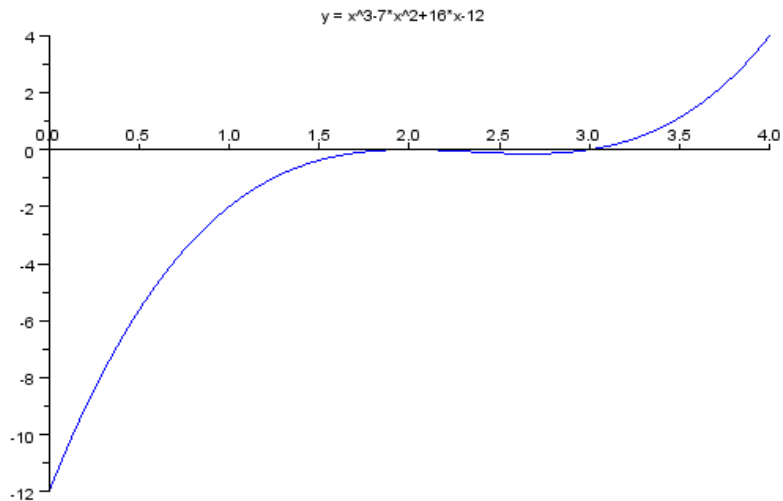


Figure 2.17: solution to the eq with multiple roots

```

1                                     // The equation  $x^3 - 7x^2 + 16x - 12 = 0$  has
                                     // real roots.
2                                     // the graph of
                                     // this function
                                     // can be
                                     // observed here.

3 xset('window', 25);
4 x=0:.001:4;
                                     //
                                     // defining the range of x.
5 def ('[y]=f(x)', 'y=x^3-7*x^2+16*x-12');
                                     //defining the function.
6 def ('[y]=fp(x)', 'y=3*x^2-14*x+16');
7 y=feval(x,f);
8
9 a=gca();

```

```

10
11 a.y_location = "origin";
12
13 a.x_location = "origin";
14 plot(x,y)

    // instruction to plot the graph
15 title(' y = x^3-7*x^2+16*x-12 ')
16
17
18
19
20 // given that the equation has double roots at x=2
    hence m=2;
21
22 m=2;
23
24                // solution by newton raphson
                    method
25
26
27 newton(1,f,fp)                // calling the user
    defined function
28
29
30
31
32                //solution by modified
                    newton raphsons method
33
34
35
36 modified_newton(1,f,fp)

```

check Appendix [AP 42](#) for dependency:

Vnewton.sce

check Appendix [AP 43](#) for dependency:

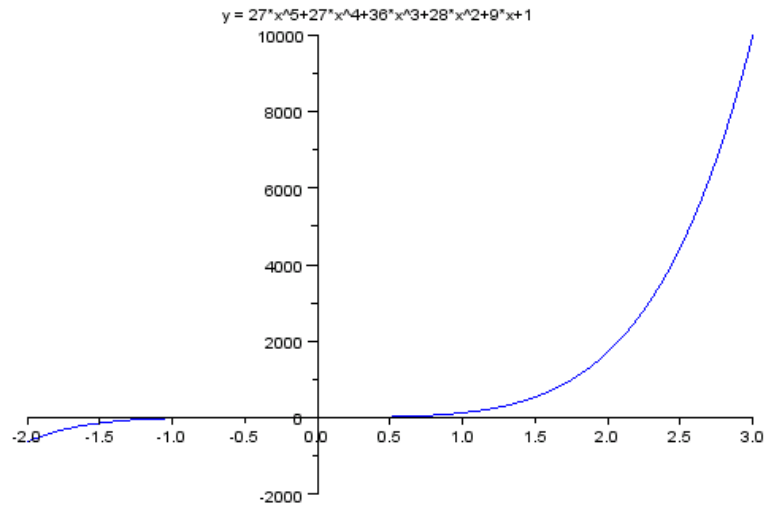


Figure 2.18: solution to the given transcendental equation

Vnewton4.sce

check Appendix [AP 50](#) for dependency:

newton63.sce

check Appendix [AP 51](#) for dependency:

secant64.sce

check Appendix [AP 52](#) for dependency:

secant65.sce

Scilab code Exa 2.27 solution to the given transcendental equation

```

1                                     // The equation
                                     27*x^5+27*x
                                     ^4+36*x^3+28*x
                                     ^2+9*x+1==0
                                     has real
                                     roots.
2                                     // the graph of
                                     this function
                                     can be
                                     observed here.

3 xset('window',26);
4 x=-2:.001:3;

                                     //
                                     defining the range of x.
5 deff(' [y]=f(x)', 'y=27*x^5+27*x^4+36*x^3+28*x^2+9*x+1
   ');                                     //defining the function.
6 deff(' [y]=fp(x)', 'y=27*5*x^4+27*4*x^3+36*3*x^2+28*2*
   x+9');
7 deff(' [y]=fpp(x)', 'y=27*5*4*x^3+27*4*3*x^2+36*3*2*x
   +28*2');
8 y=feval(x,f);
9
10 a=gca();
11
12 a.y_location = "origin";
13
14 a.x_location = "origin";
15 plot(x,y)

                                     // instruction to plot the graph
16 title(' y = 27*x^5+27*x^4+36*x^3+28*x^2+9*x+1')
17
18
19
20
21                                     // solution by newton raphson
                                     method as per the equation no.
                                     2.14

```

```

22
23
24 newton(-1,f,fp)           // calling the user
    defined function
25
26
27 newton4(-1,f,fp)
28
29
30           // solution by newton
            raphson method as per the
            equation no. 2.63
31
32 newton63(-1,f,fp,fpp)    // calling
    the user defined function
33           // solution by the secant
            method defined to
            satisfy the equation
            no.2.64.
34
35
36 secant64(0,-1,f,fp)
37
38
39
40
41
42
43           // solution by the secant
            method defined to
            satisfy the equation
            no.2.65.
44
45
46 secant65(0,-.5,f)

```

Chapter 3

SYSTEM OF LINEAR ALGEBRIC EQUATIONS AND EIGENVALUE PROBLEMS

Scilab code Exa 3.1 determinant

```
1 // example 3.1
2 //Positive definite
3
4 A=[12 4 -1;4 7 1;-1 1 6];
5 //check if the determinant of the leading minors is
   a positive value-
6
7 A1=A(1);
8 det(A1)
9 A2=A(1:2,1:2);
10 det(A2)
11 A3=A(1:3,1:3);
12 det(A3)
13
14 //we observe that the determinant of the leading
```


minors is a positive ,hence the given matrix A is
a positive definite.

Scilab code Exa 3.2 property A in the book

```
1 //example no.3.2
2 // show if a given matrix 'A' possesses 'property A'
3
4 A=[2 -1 0 -1;-1 2 -1 0;0 -1 2 0;0 0 -1 2 ]
5
6 P=[0 0 0 1;0 1 0 0;0 0 1 0;1 0 0 0] //
   let us take the pemutation matrix as P
7
8 // then we find that
9
10 P*A*P'
11
12 // is of the form (3.2) . hence the matrix 'A' has
   property A
```

Scilab code Exa 3.4 solution to the system of equations

```
1 // example no.3.4
2 // solve the system of equations
3
4 //(a) . by using cramer's rule ,
5
6 A=[1 2 -1;3 6 1;3 3 2]
7
8 B1=[2 2 -1;1 6 1;3 3 2]
9
10 B2=[1 2 -1;3 1 1;3 3 2]
11
```

```

12 B3=[1 2 2;3 6 1; 3 3 3]
13
14
15 //we know;
16
17 X1=det(B1)/det(A)
18 X2=det(B2)/det(A)
19 X3=det(B3)/det(A)
20
21
22 //(b). by detemining the inverse of the coefficient
    matrix
23
24 A=[1 2 -1;3 6 1;3 3 2]
25
26 b=[2 ;1 ;3]
27
28 //we know;
29
30 X=inv(A)*b

```

check Appendix [AP 34](#) for dependency:

Vgausselim.sce

Scilab code Exa 3.5 solution by gauss elimination method

```

1 //example-3.5
2 //caption-solution by gauss elimibnation method
3
4 A=[10 -1 2;1 10 -1;2 3 20] //matrices
    A and b from the above
5
//
system
of

```

equations

```
6 b=[4;3;7]
7
8 gausselim(A,b) //call gauss elimination
  function to solve the
9 //matrices A and b
```

check Appendix [AP 33](#) for dependency:

pivotgausselim.sci

Scilab code Exa 3.6 solution by pivoted gauss elimination method

```
1 //example-3.6
2 //caption-solution by gauss elimibnation method
3
4 A=[1 1 1;3 3 4;2 1 3] //matrices A and
  b from the above
5 //
  system
  of
  equations
```

```
6
7 b=[6;20;13]
8
9 pivotgausselim(A,b) //call gauss
  elimination function to solve the
10 //matrices A and b
```

check Appendix [AP 33](#) for dependency:

pivotgausselim.sci

Scilab code Exa 3.8 solution by pivoted gauss elimination method

```
1 //example no. 3.8
2 // solving the matrix equation with partial pivoting
   in gauss elimination
3
4 A=[2 1 1 -2;4 0 2 1;3 2 2 0;1 3 2 -1]
5
6 b=[-10;8;7;-5]
7
8 pivotgausselim(A,b)
```

check Appendix [AP 32](#) for dependency:

jordan.sce

Scilab code Exa 3.9 solution using the inverse of the matrix

```
1 //example no.3.9
2 //solving the system using inverse of the coefficient
   matrix
3
4 A=[1 1 1;4 3 -1;3 5 3]
5
6 I=[1 0 0;0 1 0;0 0 1]
7
8 b=[1 ;6 ;4]
9
10 M=jorden(A,I)
11
12 IA=M(1:3,4:6)
13
14 X=IA*b
```

check Appendix [AP 41](#) for dependency:

LandU.sce

check Appendix [AP 38](#) for dependency:

back.sce

check Appendix [AP 40](#) for dependency:

fore.sce

Scilab code Exa 3.10 decomposition method

```
1 //example no. 3.10
2 //solve system by decomposition method
3
4 A=[1 1 1;4 3 -1;3 5 3]
5 n=3;
6
7 b=[1;6;4]
8
9 [U,L]=LandU(A,3)
10
11 Z=fore(L,b)
12
13 X=back(U,Z)
```

check Appendix [AP 41](#) for dependency:

LandU.sce

Scilab code Exa 3.11 inverse using LU decomposition

```

1 //example no.3.11
2 //caption: Inverse using LU decomposition
3
4 A=[3 2 1;2 3 2;1 2 2]
5
6 [U,L]=LandU(A,3) // call LandU function to
   evaluate U,L of A,
7
8 //since A=L*U ,
9 // inv(A)=inv(U)*inv(L)
10 // let inv(A)=AI
11
12 AI=U^-1*L^-1

```

check Appendix [AP 41](#) for dependency:

LandU.sce

check Appendix [AP 38](#) for dependency:

back.sce

check Appendix [AP 40](#) for dependency:

fore.sce

Scilab code Exa 3.12 solution by decomposition method

```

1 //example no. 3.12
2 //solve system by decomposition method
3
4 A=[1 1 -1;2 2 5;3 2 -3]
5
6 b=[2;-3;6]
7
8
9

```

```

10      // hence we can observe that LU decomposition
        method fails to solve this system since
        the pivot  $L(2,2)=0$ ;
11
12
13      //we note that the coefficient matrix is not
        a positive definite matrix and hence its
        LU decomposition is not guaranteed ,
14
15
16      //if we interchange the rows of A as shown
        below the LU decomposition would work ,
17
18      A=[3 2 -3;2 2 5;1 1 -1]
19
20      b=[6; -3;2]
21
22      [U,L]=LandU(A,3)           // call LandU
        function to evaluate U,L of A,
23
24      n=3;
25      Z=fore(L,b);
26
27      X=back(U,Z)

```

check Appendix [AP 41](#) for dependency:

LandU.sce

check Appendix [AP 38](#) for dependency:

back.sce

check Appendix [AP 40](#) for dependency:

fore.sce

Scilab code Exa 3.13 LU decomposition

```

1 //example no. 3.13
2 //solve system by LU decomposition method
3
4 A=[2 1 1 -2;4 0 2 1;3 2 2 0;1 3 2 -1]
5
6 b=[-10;8;7;-5]
7
8 [U,L]=LandU(A,4)
9 n=4;
10 Z=fore(L,b);
11
12 X=back(U,Z)
13
14 //since A=L*U ,
15 // inv(A)=inv(U)*inv(L)
16 // let inv(A)=AI
17
18 AI=U^-1*L^-1

```

check Appendix [AP 38](#) for dependency:

back.sce

check Appendix [AP 39](#) for dependency:

cholesky.sce

check Appendix [AP 12](#) for dependency:

fact.sci

Scilab code Exa 3.14 cholesky method

```

1 //example no. 3.14
2 //solve system by cholesky method
3
4 A=[1 2 3;2 8 22;3 22 82]

```



```

5
6 b=[5;6;-10]
7
8 L=cholesky (A,3) //call cholesky function to
    evaluate the root of the system
9 n=3;
10 Z=fore(L,b);
11
12 X=back(L',Z)

```

check Appendix [AP 38](#) for dependency:

back.sce

check Appendix [AP 39](#) for dependency:

cholesky.sce

check Appendix [AP 40](#) for dependency:

fore.sce

Scilab code Exa 3.15 cholesky method

```

1 //example no. 3.15
2 //solve system by cholesky method
3
4 A=[4 -1 0 0;-1 4 -1 0;0 -1 4 -1;0 0 -1 4]
5
6 b=[1;0;0;0]
7
8 L=cholesky (A,4) //call cholesky function to
    evaluate the root of the system
9
10 n=4;
11 Z=fore(L,b);
12

```

```

13 X=back(L',Z)
14
15 //since A=L*L',
16 // inv(A)=inv(L')*inv(L)
17 // let inv(A)=AI
18
19 AI=L' ^ -1*L ^ -1

```

check Appendix [AP 37](#) for dependency:

`jacobiiteration.sce`

Scilab code Exa 3.21 jacobi iteration method

```

1 //example no. 3.21
2 //solve the system by jacobi iteration method
3
4 A=[4 1 1;1 5 2;1 2 3]
5
6 b=[2 ; -6; -4]
7
8 N=3;           //no. of iterations
9 n=3;           // order of the matrix is n*n
10
11 X=[.5; -.5; -.5]           //initial approximation
12
13
14 jacobiiteration(A,n,N,X,b)           //call the
    function which performs jacobi iteration method
    to solve the system

```

check Appendix [AP 36](#) for dependency:

`Vgaussseidel.sce`

Scilab code Exa 3.22 solution by gauss siedal method

```
1 //example no. 3.22
2 //solve the system by gauss seidel method
3
4 A=[2 -1 0;-1 2 -1;0 -1 2]
5
6 b=[7;1;1]
7
8 N=3;           //no. of ierations
9 n=3;           // order of the matrix is n*n
10
11 X=[0;0;0]      //initial approximation
12
13
14 gaussseidel(A,n,N,X,b)           //call the
    function which performs gauss seidel method to
    solve the system
```

check Appendix [AP 35](#) for dependency:

geigenvectors.sci

Scilab code Exa 3.27 eigen vale and eigen vector

```
1 // example 3.27
2 // a) find eigenvalue and eigen vector;
3 // b) verify  $\text{inv}(S)*A*S$  is a diagonal matrix;
4
5 // 1)
6 A=[1 2 -2 ;1 1 1;1 3 -1];
7
8 B=[1 0 0;0 1 0; 0 0 1];
9
10 [x,lam] = geigenvectors(A,B);
11
```

```
12  inv(x)*A*x
13
14  // 2)
15  A=[3 2 2;2 5 2;2 2 3];
16
17  B=[1 0 0;0 1 0; 0 0 1];
18
19
20  [x,lam] = geigenvectors(A,B);
21
22  inv(x)*A*x
```

Chapter 4

INTERPOLATION AND APPROXIMATION

check Appendix [AP 29](#) for dependency:

```
NDDinterpol.sci
```

check Appendix [AP 28](#) for dependency:

```
aitkeninterpol.sci
```

check Appendix [AP 25](#) for dependency:

```
legrangeinterpol.sci
```

Scilab code Exa 4.3 linear interpolation polinomial

```
1 // example 4.3
2 // find the linear interpolation polinomial
3 // using
4
5 disp('f(2)=4');
6 disp('f(2.5)=5.5');
7 // 1)lagrange interpolation ,
8
```

```

9  P1=legrangeinterpol (2,2.5,4,5.5)
10 // 2)aitken's iterated interpolation ,
11
12 P1=aitkeninterpol (2,2.5,4,5.5)
13
14 // 3) newton divided difference interpolation ,
15
16 P1=NDDinterpol (2,2.5,4,5.5)
17
18 // hence approximate value of f(2.2)= 4.6;

```

check Appendix [AP 25](#) for dependency:

legrangeinterpol.sci

Scilab code Exa 4.4 linear interpolation polinomial

```

1  // example 4.4
2  // find the linear interpolation polinomial
3  // using lagrange interpolation ,
4
5  disp('sin (.1) =.09983; sin (.2) =.19867 ');
6
7
8  P1=legrangeinterpol (.1,.2,.09983,.19867)
9
10 // hence;
11 disp('P(.15) =.00099+.9884*.15 ')
12 disp('P(0.15) =0.14925 ');

```

check Appendix [AP 25](#) for dependency:

legrangeinterpol.sci

Scilab code Exa 4.6 legrange linear interpolation polinomial

```
1 // example :4.6
2 // caption : obtain the legrange linear
  interpolating polinomial
3
4
5 // 1) obtain the legrange linear interpolating
  polinomial in the interval [1,3] and obtain
  approximate value of f(1.5),f(2.5);
6 x0=1;x1=2;x2=3;f0=.8415;f1=.9093;f2=.1411;
7
8 P13=legrangeinterpol (x0,x2,f0,f2)           //in
  the range [1 ,3]
9
10
11 P12=legrangeinterpol (x0,x1,f0,f1)         //
  in the range [1,2]
12
13 P23=legrangeinterpol (x1,x2,f1,f2)
  // inthe range [2,3]
14
15 // from P23 we find that ; where as exact value is
  sin(2.5)=0.5985;
16 disp('P(1.5)=0.8754 ');
17 disp('exact value of sin(1.5)=.9975 ');
18 disp('P(2.5)=0.5252 ');
```

check Appendix [AP 24](#) for dependency:

lagrangefundamentalpoly.sci

Scilab code Exa 4.7 polynomial of degree two

```
1 // example 4.7
2 // polinomial of degree 2;
```

```

3
4 // f(0)=1;f(1)=3;f(3)=55;
5
6 // using legrange fundamental polinomial rule ,
7
8 x=[0 1 3]; // arrainging the
    inputs of the function as elements of a row,
9 f=[1 3 55]; // arrainging the
    outputs of the function as elements of a row,
10 n=2; // degree of the
    polinomial;
11
12
13 P2=lagrangefundamentalpoly(x,f,n)

```

check Appendix [AP 24](#) for dependency:

lagrangefundamentalpoly.sci

Scilab code Exa 4.8 solution by quadratic interpolation

```

1
2 // example 4.8
3 // caption: solution by quadratic interpolation;
4
5 // x-degrees:[10 20 30]
6 // hence x in radians is
7 x=[3.14/18 3.14/9 3.14/6];
8 f=[1.1585 1.2817 1.3660];
9 n=2;
10
11
12 P2=lagrangefundamentalpoly(x,f,n)
13
14 // hence from P2 ,the exact value of f(3.14/12) is
    1.2246;

```



```
15 // where as exact value is 1.2247;
```

check Appendix [AP 23](#) for dependency:

```
NDDinterpol2.sci
```

check Appendix [AP 22](#) for dependency:

```
iteratedinterpol.sci
```

Scilab code Exa 4.9 polinomial of degree two

```
1 // example 4.9
2 // caption :obtain the polinomial of degree 2
3
4 x=[0 1 3];
5 f=[1 3 55];
6 n=2;
7
8 // 1) iterated interpolation;
9
10
11 [L012,L02,L01]=iteratedinterpol (x,f,n)
12
13 // 2) newton divided differences interpolation;
14
15
16 P2=NDDinterpol2 (x,f)
```

check Appendix [AP 31](#) for dependency:

```
NBDP.sci
```

Scilab code Exa 4.15 forward and backward difference polynomial

```

1 // example 4.15:
2 // obtain the interpolate using backward differences
   polinomial
3
4
5 xL=[.1 .2 .3 .4 .5 ]'
6
7 f=[1.4 1.56 1.76 2 2.28]'
8 n=2;
9
10
11 // hence;
12 disp('P=1.4+(x-.1)*(.16/.1)+(x-.5)*(x-.4)*(.04/.02)')
   )
13 disp('P=2x^2+x+1.28');
14
15 // 1) obtain the interpolate at x=0.25;
16 x=0.25;
17 [P]=NBBDP(x,n,xL,f);
18 P
19 disp('f(.25)=1.655');
20
21
22 // 2) obtain the interpolate at x=0.35;
23 x=0.35;
24 [P]=NBBDP(x,n,xL,f);
25 P
26 disp('f(.35)=1.875');

```

check Appendix [AP 30](#) for dependency:

hermiteinterpol.sci

Scilab code Exa 4.20 hermite interpolation

```

1 // example 4.20;

```

```

2 // hermite interpolation:
3
4 x=[-1 0 1];
5
6 f=[1 1 3];
7
8 fp=[-5 1 7];
9
10 P= hermiteinterpol(x,f,fp);
11
12 // hence;
13 disp('f(-0.5)=3/8');
14 disp('f(0.5)=11/8');

```

check Appendix [AP 25](#) for dependency:

legrangeinterpol.sci

Scilab code Exa 4.21 piecewise linear interpolating polinomial

```

1 // example: 4.21;
2 // piecewise linear interpolating polinomials:
3
4 x1=1;x2=2; x3=4;x4=8;
5 f1=3; f2=7; f3=21; f4=73;
6 // we need to apply legranges interpolation in sub-
   ranges [1,2];[2,4],[4,8];
7
8 x=poly(0,"x");
9
10 P1=legrangeinterpol (x1,x2,f1,f2);           // in the
   range [1,2]
11 P1
12
13 P1=legrangeinterpol (x2,x3,f2,f3);           // in the
   range [2,4]

```

```

14 P1
15
16 P1=legrangeinterpol (x3,x4,f3,f4);           // in the
    range [4,8]
17 P1

```

check Appendix [AP 24](#) for dependency:

lagrangefundamentalpoly.sci

Scilab code Exa 4.22 piecewise quadratic interpolating polinomial

```

1 // example: 4.22;
2 // piecewise quadratic interpolating polinomials:
3
4 X=[-3 -2 -1 1 3 6 7];
5 F=[369 222 171 165 207 990 1779];
6 // we need to apply legranges interpolation in sub-
    ranges [-3 , -1];[-1,3],[3,7];
7
8 x=poly(0,"x");
9
10 // 1) in the range [-3,-1]
11 x=[-3 -2 -1];
12 f=[369 222 171];
13 n=2;
14 P2=lagrangefundamentalpoly(x,f,n);
15
16 // 2) in the range [-1,3]
17 x=[-1 1 3];
18 f=[171 165 207];
19 n=2;
20 P2=lagrangefundamentalpoly(x,f,n)
21
22 // 3) in the range [3,7]
23 x=[3 6 7];

```

```

24 f=[207 990 1779];
25 n=2;
26 P2=lagrangefundamentalpoly(x,f,n)
27
28
29
30 // hence, we obtain the values of f(-2.5)=48; f
    (6.5)=1351.5;

```

check Appendix [AP 24](#) for dependency:

lagrangefundamentalpoly.sci

Scilab code Exa 4.23 piecewise cubical interpolating polinomial

```

1 // example: 4.23;
2 // piecewise cubical interpolating polinomials:
3
4 X=[-3 -2 -1 1 3 6 7];
5 F=[369 222 171 165 207 990 1779];
6 // we need to apply legranges interpolation in sub-
    ranges [-3 ,1];[1 ,7];
7
8
9 x=poly(0,"x");
10
11 // 1) in the range [-3,1]
12 x=[-3 -2 -1 1];
13 f=[369 222 171 165];
14 n=3;
15 P2=lagrangefundamentalpoly(x,f,n);
16
17 // 2) in the range [1,7]
18 x=[1 3 6 7];
19 f=[165 207 990 1779];
20 n=3;

```

```

21 P2=lagrangefundamentalpoly(x,f,n)
22
23
24
25 // hence ,
26 disp('f(6.5)=1339.25');

```

Scilab code Exa 4.31 linear approximation

```

1 // example 4.31
2 // obtain the linear polinomial approximation to the
  function f(x)=x^3
3
4 // let P(x)=a0*x+a1
5
6 // hence I(a0,a1)= integral (x^3-(a0*x+a1))^2 in the
  interval [0,1]
7
8 printf('I=1/7-2*(a0/5+a1/4)+a0^2/3+a0*a1+a1^2')
9 printf('dI/da0 = -2/5+2/3*a0+a1=0')
10
11 printf('dI/da1 = -1/2+a0+2*a1=0')
12
13 // hence
14
15 printf('[2/3 1;1 2]*[a0 ;a1]=[2/5; 1/2]')
16
17 // solving for a0 and a1;
18
19 a0=9/10;
20 a1=-1/5;
21 // hence considering the polinomial with intercept
  P1(x)=(9*x-2)/10;
22
23 // considering the polinomial approximation through

```

```
origin P2(x)=3*x/5;
```

Scilab code Exa 4.32 linear polinomial approximation

```
1 // example 4.32
2 // obtain the linear polinomial approximation to the
  function f(x)=x^1/2
3
4 // let P(x)=a0*x+a1
5
6
7 // for n=1;
8 // hence I(c0,c1)= integral (x^1/2-(c1*x+c0))^2 in
  the interval [0,1]
9
10
11 printf('dI/dc0 = -2*(2/3-c0-c1/2)=0')
12
13 printf('dI/dc1 =-2*(2/5-c0/2-c1/3) =0')
14
15 // hence
16
17 printf('[1 1/2;1/2 1/3]*[c0 ;c1]=[-4/3; -4/5]')
18
19 // hence solving for c0 and c1;
20
21
22 // the first degree square approximation P(x)
  =4*(1+3*x)/15;
23
24 // for n=2;
25
26 // hence I(c0,c1,c2)= integral (x^1/2-(c2*x^2+c1*x+
  c0))^2 in the interval [0,1]
27
```

```

28
29 printf('dI/dc0 = (2/3-c0-c1/2-c2/2)=0')
30
31 printf('dI/dc1 =(2/5-c0/2-c1/3-c2/4) =0')
32
33 printf('dI/dc2 =(2/7-c0/3-c1/4-c2/5) =0')
34
35
36 // hence
37
38 printf('[1 1/2 1/2;1/2 1/3 1/4;1/3 1/4 1/5]*[c0 ;c1;
        c2]=[-2/3; -2/5;-2/7]')
39
40 // hence solving for c0,c1 and c2;
41
42
43 // the first degree square approximation P(x)=(6+48*
        x-20*x^2)/35;

```

check Appendix [AP 27](#) for dependency:

straightlineapprox.sce

Scilab code Exa 4.34 least square straight fit

```

1 // example 4.34
2
3 // obtain least square straight line fit
4
5 x=[.2 .4 .6 .8 1];
6 f=[.447 .632 .775 .894 1];
7
8
9 [P]=straightlineapprox(x,f) // call of the
        function to get the desired solution

```

check Appendix [AP 26](#) for dependency:

quadraticapprox.sci

Scilab code Exa 4.35 least square approximation

```
1 // example 4.35
2
3 // obtain least square approximation of second
  degree;
4 x=[-2 -1 0 1 2];
5 f=[15 1 1 3 19];
6
7 [P]=quadraticapprox(x,f) // call of the
  function to get the desired solution
```

Scilab code Exa 4.36 least square fit

```
1 // example 4.36
2 // method of least squares to fit the data to the
  curve  $P(x)=c_0*X+c_1/\text{sqrt}(X)$ 
3
4 x=[.2 .3 .5 1 2];
5 f=[16 14 11 6 3];
6
7 //  $I(c_0, c_1) = \text{summation of } (f(x) - (c_0*X + c_1/\text{sqrt}(X)))^2$ 
8
9 // hence on partially derivating the summation,
10
11 n=length(x); m=length(f);
12 if m<>n then
13     error('linreg - Vectors x and f are not of the
      same length.');
```

```

14     abort;
15 end;
16
17 s1=0;           // s1= summation of x(i
    )*f(i)
18 s2=0;           // s2= summation of f(i
    )/sqrt(x(i))
19 s3=0;
20 for i=1:n
21     s1=s1+x(i)*f(i);
22     s2=s2+f(i)/sqrt(x(i));
23     s3=s3+1/x(i);
24 end
25
26 c0=det([s1 sum(sqrt(x));s2 s3])/det([sum(x^2) sum(
    sqrt(x));sum(sqrt(x)) s3])
27
28 c1=det([sum(x^2) s1;sum(sqrt(x)) s2])/det([sum(x^2)
    sum(sqrt(x));sum(sqrt(x)) s3])
29 X=poly(0,"X");
30 P=c0*X+c1/X^1/2
31 // hence considering the polinomial P(x)=7.5961*X
    ^1/2-1.1836*X

```

Scilab code Exa 4.37 least square fit

```

1 // example 4.37
2 // method of least squares to fit the data to the
    curve P(x)=a*%e^(-3*t)+b*%e^(-2*t);
3
4 t=[.1 .2 .3 .4];
5 f=[.76 .58 .44 .35];
6
7 // I(c0,c1)= summation of (f(x)-a*%e^(-3*t)+b*%e
    ^(-2*t))

```

```

8
9 // hence on partially derivating the summation,
10
11 n=length(t);m=length(f);
12 if m<>n then
13     error('linreg - Vectors t and f are not of the
14           same length. ');
15     abort;
16 end;
17 s1=0; // s1= summation of f(i
18     )*%e^(-3*t(i));
19 s2=0; // s2= summation of f(i
20     )*%e^(-2*t(i));
21
22 for i=1:n
23     s1=s1+f(i)*%e^(-3*t(i));
24     s2=s2+f(i)*%e^(-2*t(i));
25
26 end
27
28 a=det([s1 sum(%e^(-5*t)); s2 sum(%e^(-4*t))])/det([
29     sum(%e^(-6*t)) sum(%e^(-5*t)); sum(%e^(-5*t)) sum
30     (%e^(-4*t))])
31
32 b=det([sum(%e^(-6*t)) s1; sum(%e^(-5*t)) s2])/det([
33     sum(%e^(-6*t)) sum(%e^(-5*t)); sum(%e^(-5*t)) sum
34     (%e^(-4*t))])
35
36 // hence considering the polinomial P(t)=.06853*%e
37     ^(-3*t)+0.3058*%e^(-2*t)

```

Scilab code Exa 4.38 gram schmidt orthogonalisation

```
1 // example 4.38
```

```

2 // gram schmidt orthogonalisation
3
4 W=1;
5 x=poly(0,"x");
6 P0=1;
7 phi0=P0;
8 a10=integrate('W*x*phi0','x',0,1)/integrate('W
      *1*phi0','x',0,1)
9 P1=x-a10*phi0
10 phi1=P1;
11
12 a20=integrate('W*x^2*phi0','x',0,1)/integrate(
      'W*1*phi0','x',0,1)
13
14 a21=integrate('(x^2)*(x-1/2)','x',0,1)/integrate
      ('(x-1/2)^2','x',0,1)
15
16 P2=x^2-a20*x-a21*phi1
17
18 // since ,I= intgral [x^(1/2)-c0*P0-c1*P1-c2*P2]^2
      inthe range [0,1]
19
20 // hence partially derivating I
21
22 c0=integrate('x^(1/2)','x',0,1)/integrate('1','x'
      ,0,1)
23 c1=integrate('(x^(1/2))*(x-(1/2))','x',0,1)/
      integrate('(x-(1/2))^2','x',0,1)
24 c1=integrate('(x^(1/2))*(x^2-4*x/3+1/2)','x',0,1)/
      integrate('(x^2-4*x/3+1/2)^2','x',0,1)

```

Scilab code Exa 4.39 gram schmidt orthogonalisation

```

1 // example 4.39
2 // gram schmidt orthogonalisation

```

```

3
4 // 1)
5 W=1;
6 x=poly(0,"x");
7 P0=1
8 phi0=P0;
9 a10=0;
10 P1=x-a10*phi0
11 phi1=P1;
12
13 a20=integrate('x^2','x',-1,1)/integrate('W*1*
      phi0','x',-1,1);
14
15 a21=integrate('(x^3)','x',-1,1)/integrate('(x)^2
      ','x',-1,1);
16
17 P2=x^2-a20*x-a21*phi1
18
19
20 // 2)
21 disp(' W=1/(1-x^2)^(1/2) ');
22 x=poly(0,"x");
23 P0=1
24 phi0=P0;
25 a10=0;
26 P1=x-a10*phi0
27 phi1=P1;
28
29 a20=integrate('x^2/(1-x^2)^(1/2)','x',-1,1)/
      integrate('1/(1-x^2)^(1/2)','x',-1,1);
30
31 a21=0; // since x^3 is
      an odd function;
32
33 P2=x^2-a20*x-a21*phi1

```

Scilab code Exa 4.41 chebishev polinomial

```
1 // example 4.41
2 // using chebyshev polinomials obtain least squares
   approximation of second degree;
3
4 // the chebeshev polinomials;
5 x=poly(0,"x");
6 T0=1;
7 T1=x;
8 T2=2*x^2-1;
9
10
11 // I=integrate ('1/(1-x^2)^(1/2)*(x^4-c0*T0-c1*T1-c2
   *T2)^2','x',-1,1)
12
13 // since;
14 c0=integrate(' (1/3.14)*(x^4)/(1-x^2)^(1/2)', 'x',
   ,-1,1)
15
16 c1=integrate(' (2/3.14)*(x^5)/(1-x^2)^(1/2)', 'x',
   ,-1,1)
17
18 c2=integrate(' (2/3.14)*(x^4)*(2*x^2-1)/(1-x^2)^(1/2)
   ', 'x', -1,1)
19
20 f=(3/8)*T0+(1/2)*T2;
```

Chapter 5

DIFFERENTIATION AND INTEGRATION

check Appendix [AP 21](#) for dependency:

```
linearinterpol.sci
```

Scilab code Exa 5.1 linear interpolation

```
1 // example: 5.1
2 // linear and quadratic interpolation:
3
4 // f(x)=ln x;
5
6 xL=[2 2.2 2.6];
7 f=[.69315 .78846 .95551];
8
9 // 1) fp(2) with linear interpolation;
10
11 fp=linearinterpol(xL,f);
12 disp(fp);
```

Scilab code Exa 5.2 quadratic interpolation

```
1 // example 5.2
2 // evaluate fp(.8) and fpp(.8) with quadratic
  interpolation;
3
4 xL=[.4 .6 .8];
5 f=[.0256 .1296 .4096];
6 h=.2;
7
8 fp=(1/2*h)*(f(1)-4*f(2)+3*f(3))
9 fpp=(1/h^2)*(f(1)-2*f(2)+f(3))
   // from equation 5.22c
   and 5.24c in the book;
```

check Appendix [AP 20](#) for dependency:

`jacobianmat.sci`

Scilab code Exa 5.10 jacobian matrix of the given system

```
1 // example 5.10;
2 // find the jacobian matrix;
3
4
5 // given two functions in x,y;
6 // and the point at which the jacobian has to be
  found out;
7
8 def f(' [w]=f1(x,y)', 'w=x^2+y^2-x');
9
10 def f(' [q]=f2(x,y)', 'q=x^2-y^2-y');
11
12 h=1;k=1;
13
14 J= jacobianmat (f1,f2,h,k);
```


15 `disp(J);`

check Appendix [AP 18](#) for dependency:

`simpson.sci`

check Appendix [AP 19](#) for dependency:

`trapezoidal.sci`

Scilab code Exa 5.11 solution by trapezoidal and simpsons

```
1 // example : 5.11
2 // solve the definite integral by 1) trapezoidal
  rule , 2) simpsons rule
3 // exact value of the integral is  $\ln 2 = 0.693147$ ,
4
5 def ('[y]=F(x)', 'y=1/(1+x)')
6
7 //1) trapezoidal rule ,
8
9 a=0;
10 b=1;
11 I =trapezoidal(0,1,F)
12 disp(error =.75-.693147)
13
14 // simpson 's rule
15
16 I=simpson(a,b,F)
17
18 disp(error =.694444-.693147)
```

Scilab code Exa 5.12 integral approximation by mid point and two point

```

1 // example 5.12
2 // caption: solve the integral by 1)mid-point rule
   ,2)two-point open type rule
3
4
5 // let integration of  $f(x)=\sin(x)/(x)$  in the range
    $[0,1]$  is equal to I1 and I2
6 // 1)mid -point rule;
7 a=0;b=1;
8 h=(b-a)/2;
9
10 x=0:h:1;
11 deff ('[y]=f(x)', 'y=sin(x)/x')
12 I1=2*h*f(x(1)+h)
13
14
15 //2) two-point open type rule
16 h=(b-a)/3;
17 I2=(3/2)*h*(f(x(1)+h)+f(x(1)+2*h))

```

check Appendix [AP 17](#) for dependency:

simpson38.sci

Scilab code Exa 5.13 integral approximation by simpson three eight rule

```

1 // example 5.13
2 // caption: simpson 3-8 rule
3
4
5 // let integration of  $f(x)=1/(1+x)$  in the range
    $[0,1]$  by simpson 3-8 rule is equal to I
6
7 x=0:1/3:1;
8 deff ('[y]=f(x)', 'y=1/(1+x)')
9

```

```
10 [I] = simpson38(x,f)
```

Scilab code Exa 5.15 quadrature formula

```
1 // example :5.15
2 // find the quadrature formula of
3 // integral of  $f(x)*(1/\sqrt{x(1+x)})$  in the range
4 // [0,1]=  $a1*f(0)+a2*f(1/2)+a3*f(1)=I$ 
5 // hence find integral  $1/\sqrt{x-x^3}$  in the range
6 // [0,1]
7 // making the method exact for polynomials of degree
8 // upto 2,
9 //  $I=I1=a1+a2+a3$ 
10 //  $I=I2=(1/2)*a2+a3$ 
11 //  $I=I3=(1/4)*a2+a3$ 
12 //  $A=[a1 \ a2 \ a3]'$ 
13  $I1=\text{integrate}('1/\sqrt{x*(1-x)}', 'x', 0, 1)$ 
14  $I2=\text{integrate}('x/\sqrt{x*(1-x)}', 'x', 0, 1)$ 
15  $I3=\text{integrate}('x^2/\sqrt{x*(1-x)}', 'x', 0, 1)$ 
16 //hence
17 //  $[1 \ 1 \ 1; 0 \ 1/2 \ 1; 0 \ 1/4 \ 1]*A=[I1 \ I2 \ I3]'$ 
18 //  $A=\text{inv}([1 \ 1 \ 1; 0 \ 1/2 \ 1; 0 \ 1/4 \ 1])*[I1 \ I2 \ I3]'$ 
19 //  $I=(3.14/4)*(f(0)+2*f(1/2)+f(1))$ ;
20 // hence, for solving the integral  $1/\sqrt{x-x^3}$ 
21 // in the range  $[0,1]=I$ 
22 //
23 //  $\text{def}f(' [y]=f(x)', 'y=1/\sqrt{1+x}')$ ;
24 //  $I=(3.14/4)*[1+2*\text{sqrt}(2/3)+\text{sqrt}(2)/2]$ 
```

Scilab code Exa 5.16 gauss legendary three point method

```
1 // example 5.16
2 // caption: gauss-legendre three point method
3 // I= integral 1/(1+x) in the range [0,1];
4 // first we need ti transform the interval [0,1 ] to
   [-1,1], since gauss-legendre three point method
   is applicable in the range[-1,1],
5
6 // let t=ax+b;
7 // solving for a,b from the two ranges , we get a=2;
   b=-1; t=2x-1;
8
9 // hence I=integral 1/(1+x) in the range [0,1]=
   integral 1/(t+3) in the range [-1,1];
10
11
12 deff ('[y]=f(t)', 'y=1/(t+3)');
13 // since , from gauss legendre three point rule (n=2)
   ;
14 I=(1/9)*(5*f(-sqrt(3/5))+8*f(0)+5*f(sqrt(3/5)))
15
16 // we know , exact solution is ln 2=0.693147;
```

Scilab code Exa 5.17 gauss legendary method

```
1 // example 5.17
2 // caption: gauss-legendre method
3 // I= integral 2*x/(1+x^4) in the range [1,2];
4 // first we need ti transform the interval [1,2 ] to
   [-1,1], since gauss-legendre three point method
   is applicable in the range[-1,1],
```

```

5
6 // let t=ax+b;
7 // solving for a,b from the two ranges , we get a
  =1/2; b=3/2; x=(t+3)/2;
8
9 // hence I=integral 2*x/(1+x^4) in the range [0,1]=
  integral 8*(t+3)/16+(t+3)^4 in the range [-1,1];
10
11
12 def f(' [y]=f(t) ', 'y=8*(t+3)/(16+(t+3)^4) ');
13
14 // 1) since , from gauss legendre one point rule;
15 I1=2*f(0)
16
17 // 2) since , from gauss legendre two point rule;
18 I2=f(-1/sqrt(3))+f(1/sqrt(3))
19
20 // 3) since , from gauss legendre three point rule;
21 I=(1/9)*(5*f(-sqrt(3/5))+8*f(0)+5*f(sqrt(3/5)))
22
23
24 // we know , exact solution is 0.5404;

```

Scilab code Exa 5.18 integral approximation by gauss chebishev

```

1 // example 5.18
2 // caption: gauss-chebyshev method
3
4 // we write the integral as I=integral f(x)/sqrt(1-x
  ^2) in the range [-1,1];
5 // where f(x)=(1-x^2)^2*cos(x)
6
7 def f(' [y]=f(x) ', 'y=(1-x^2)^2*cos(x) ');
8
9 // 1) since , from gauss chebyshev one point rule;

```

```

10 I1=(3.14)*f(0)
11
12 // 2) since , from gauss chebyshev two point rule;
13 I2=(3.14/2)*f(-1/sqrt(2))+f(1/sqrt(2))
14
15 // 3) since , from gauss chebyshev three point rule;
16 I=(3.14/3)*(f(-sqrt(3)/2)+f(0)+f(sqrt(3)/2))
17
18
19 // and 4) since , from gauss legendre three point
    rule;
20 I=(1/9)*(5*f(-sqrt(3/5))+8*f(0)+5*f(sqrt(3/5)))

```

Scilab code Exa 5.20 integral approximation by gauss legurre method

```

1 // example 5.20
2 // caption: gauss-leguerre method
3 // I= integral e^-x/(1+x^2) in the range [0,~];
4
5 // observing the integral we can inffer that f(x)
    =1/(1+x^2)
6
7 def f (' [y]=f(x) ', 'y=1/(1+x^2) ');
8
9
10 // 1) since , from gauss leguerre two point rule;
11 I2=(1/4)*[(2+sqrt(2))*f(2-sqrt(2))+(2-sqrt(2))*f(2+
    sqrt(2))]
12
13 // 3) since , from gauss leguerre three point rule;
14 I=(0.71109*f(0.41577)+0.27852*f(2.29428)+0.01039*f
    (6.28995))

```

Scilab code Exa 5.21 integral approximation by gauss legurre method

```
1 // example 5.21
2 // caption: gauss-leguerre method
3 // I= integral e-x*(3*x3-5*x+1) in the range
  [0,~];
4
5 // observing the integral we can inffer that f(x)
  =(3*x3-5*x+1)
6
7 def f (' [y]=f(x) ', 'y=(3*x3-5*x+1) ');
8
9
10 // 1) since , from gauss leguerre two point rule;
11 I2=(1/4)*[(2+sqrt(2))*f(2-sqrt(2))+(2-sqrt(2))*f(2+
  sqrt(2))]
12
13 // 3) since , from gauss leguerre three point rule;
14 I3=(0.71109*f(0.41577)+0.27852*f(2.29428)+0.01039*f
  (6.28995))
```

Scilab code Exa 5.22 integral approximation by gauss legurre method

```
1 // example 5.22
2 // caption: gauss-leguerre method
3 // I= integral 1/(x2+2*x+2) in the range [0,~];
4
5 // since in the gauss-leguerre method the integral
  would be of the form ex*f(x);
6
7 // observing the integral we can inffer that f(x)=%e
  ^x/(x2+2*x+2)
8 def f (' [y]=f(x) ', 'y=%ex/(x2+2*x+2) ');
9
10
```

```

11 // 1) since , from gauss leguerre two point rule;
12 I2=(1/4)*[(2+sqrt(2))*f(2-sqrt(2))+(2-sqrt(2))*f(2+
    sqrt(2))]
13
14 // 3) since , from gauss leguerre three point rule;
15 I=(0.71109*f(0.41577)+0.27852*f(2.29428)+0.01039*f
    (6.28995))
16
17
18 // the exact solution is given by,
19
20 I=integrate('1/((x+1)^2+1)', 'x', 0, 1000) // 1000
    ~infinite;

```

check Appendix [AP 15](#) for dependency:

comp_trapezoidal.sci

check Appendix [AP 16](#) for dependency:

simpson13.sci

Scilab code Exa 5.26 composite trapizoidal and composite simpson

```

1 // Example 5.26
2 // caption: 1) composite trapizoidal rule , 2)
    composite simpsons rule with 2,4 ,8 equal sub-
    intervals ,
3
4 // I=integral 1/(1+x) in the range [0,1]
5
6 def f (' [y]=f(x)', 'y=1/(1+x)')
7
8 // when N=2;
9 // 1) composite trapizoidal rule
10 h=1/2;
11 x=0:h:1;

```



```

12
13 IT=comptrapezoidal(x,h,f)
14
15 // 2) composite simpsons rule
16
17 [I] = simpson13(x,h,f)
18
19
20 // when N=4
21 // 1) composite trapizoidal rule
22 h=1/4;
23 x=0:h:1;
24
25 IT=comptrapezoidal(x,h,f)
26
27 // 2) composite simpsons rule
28
29 [I] = simpson13(x,h,f)
30
31
32
33 // when N=8
34 // 1) composite trapizoidal rule
35 h=1/8;
36 x=0:h:1;
37
38 IT=comptrapezoidal(x,h,f)
39
40 // 2) composite simpsons rule
41
42 [I] = simpson13(x,h,f)

```

Scilab code Exa 5.27 integral approximation by gauss legurre method

```
1 // example 5.27
```

```

2 // caption: gauss-legendre three point method
3 // I= integral 1/(1+x) in the range [0,1];
4
5 // we are asked to subdivide the range into two,
6 // first we need to sub-divide the interval [0,1 ]
   to [0,1/2] and [1/2,1] and then transform both to
   [-1,1], since gauss-legendre three point method
   is applicable in the range[-1,1],
7
8 // t=4x-1 and y=4x-3;
9
10 // hence I=integral 1/(1+x) in the range [0,1]=
   integral 1/(t+5) in the range [-1,1]+ integral
   1/(t+7) in the range [-1,1]
11
12
13 def f1(' [y1]=f1(t) ', 'y1=1/(t+5) ');
14 // since , from gauss legendre three point rule(n=2)
   ;
15 I1=(1/9)*(5*f1(-sqrt(3/5))+8*f1(0)+5*f1(sqrt(3/5)))
16
17 def f2(' [y2]=f2(t) ', 'y2=1/(t+7) ');
18 // since , from gauss legendre three point rule(n=2)
   ;
19 I2=(1/9)*(5*f2(-sqrt(3/5))+8*f2(0)+5*f2(sqrt(3/5)))
20
21 I=I1+I2
22
23 // we know , exact solution is .693147;

```

Scilab code Exa 5.29 double integral using simpson rule

```

1 // example 5.29
2 // evaluate the given double integral using the
   simpsons rule;

```

```

3
4 // I= double integral f(x)=1/(x+y) in the range x
   =[1,2],y=[1,1.5];
5
6 h=.5;
7 k=.25;
8 def f(' [w]=f(x,y)', 'w=1/(x+y)')
9
10 I=(.125/9)*[{f(1,1)+f(2,1)+f(1,1.5)+f(2,1.5)}+4*{f
   (1.5,1)+f(1,1.25)+f(1.5,1.5)+f(2,1.25)}+16*f
   (1.5,1.25)];
11 disp(I);

```

Scilab code Exa 5.30 double integral using simpson rule

```

1 // example 5.30
2 // evaluate the given double integral using the
   simpsons rule;
3
4 // I= double integral f(x)=1/(x+y) in the range x
   =[1,2],y=[1,2];
5 // 1)
6 h=.5;
7 k=.5;
8 def f(' [w]=f(x,y)', 'w=1/(x+y)')
9
10 I=(1/16)*[{f(1,1)+f(2,1)+f(1,2)+f(2,2)}+2*{f(1.5,1)+
   f(1,1.5)+f(2,1.5)+f(1.5,2)}+4*f(1.5,1.5)]
11
12 // 2)
13 h=.25;
14 k=.25;
15 def f(' [w]=f(x,y)', 'w=1/(x+y)')
16
17 I=(1/64)*[{f(1,1)+f(2,1)+f(1,2)+f(2,2)}+2*{f(5/4,1)+

```

$$\begin{aligned} & f(3/2, 1) + f(7/4, 1) + f(1, 5/4) + f(1, 3/2) + f(1, 7/4) + f \\ & (2, 5/4) + f(2, 3/4) + f(2, 7/4) + f(5/4, 2) + f(3/2, 2) + f \\ & (7/4, 2) \} + 4 * \{ f(5/4, 5/4) + f(5/4, 3/2) + f(5/4, 7/4) + f \\ & (3/2, 5/4) + f(3/2, 3/2) + f(3/2, 7/4) + f(7/4, 5/4) + f \\ & (7/4, 3/2) + f(7/4, 7/4) \} \end{aligned}$$

Chapter 6

ORDINARY DIFFERENTIAL EQUATIONS INNITIAL VALUE PROBLEMS

check Appendix [AP 3](#) for dependency:

```
eigenvectors.sci
```

Scilab code Exa 6.3 solution to the system of equations

```
1 // example 6.3
2 // solution to the given IVP
3
4 disp('du/dt= A*u');
5 // u=[u1 u2]';
6 A=[-3 4 ; -2 3]; // given
7 B=[1 0; 0 1]; // identity
   matrix;
8
9
10
11
12 [x,lam] = geigenvectors(A,B);
```

```

13
14 // hence;
15 disp('u=c1*%e^t*x(:,1)+c2*%e^-t*x(:,2)');
16 disp('u1=c1*%e^t+c2*%e^-t*2');
17 disp('u2=c1*%e^t+c2*%e^-t');

```

check Appendix [AP 3](#) for dependency:

eigenvectors.sci

Scilab code Exa 6.4 solution to the IVP

```

1 // example 6.4
2 // solution to the given IVP
3
4 disp('du/dt= A*u');
5 // u=[u1 u2]';
6 A=[-2 1;1 -20]; // given
7 B=[1 0;0 1]; // identity
   matrix;
8
9
10
11
12
13 [x,lam] = geigenvectors(A,B);
14
15 // hence;
16 disp('u=c1*%e^(lam(1)*t)*x(:,1)+c2*%e^-(lam(2)*t)*x
   ((:,2)')');

```

check Appendix [AP 2](#) for dependency:

Euler1.sce

Scilab code Exa 6.9 euler method to solve the IVP

```
1 // example 6.9
2 // solve the IVP by euler method,
3 // with h=0.2, 0.1, 0.05;
4 // u'=f(t,u)
5 // u'=-2tu^2
6 deff(' [z]=f(t,u)', 'z=-2*t*u^2');
7
8
9 [u,t] = Euler1(1,0,1,.2,f) // h=0.2;
10
11 [u,t] = Euler1(1,0,1,0.1,f) // h=0.1;
12
13
14 [u,t] = Euler1(1,0,1,0.05,f) // h=0.05;
```

check Appendix [AP 14](#) for dependency:

backeuler.sci

Scilab code Exa 6.12 solution ti IVP by back euler method

```
1 // example 6.12,
2 // caption: solve the IVP by backward euler method,
3 // with h=0.2,
4 // u'=f(t,u)
5 deff(' [z]=f(t,u)', 'z=-2*t*u^2');
6
7
8 [u] = backeuler(1,0,0.4,.2,f) // h=0.2;
```

check Appendix [AP 13](#) for dependency:

eulermidpoint.sci

Scilab code Exa 6.13 solution ti IVP by euler mid point method

```
1 // example 6.13 ,
2 // caption: solve the IVP by euler midpoint method ,
3 // with h=0.2 ,
4 // u'=f(t , u)
5 def f (' [z]=f(t , u) ' , ' z=-2*t*u^2 ' ) ;
6 def fp (' [w]=fp(t , u) ' , ' w=-2*u^2-4*u*t ' ) ;
7
8
9
10 [u] = eulermidpoint(1,0,1,.2,f,fp) // h=0.2;
```

check Appendix [AP 12](#) for dependency:

fact.sci

check Appendix [AP 11](#) for dependency:

taylor.sci

Scilab code Exa 6.15 solution ti IVP by taylor expansion

```
1 // example 6.15
2 //caption: solving ODE by tailor series method
3 // u'=t^2+ u^2, u0=0;
4
5 t=0;U=0; //at t=0, the value of u
   is 0
6 U1=0; // u1 is the 1st derivatove
   of the funtion u
7 U2=2*t+2*U*U1 // U2 —— 2nd
   derivative
```



```

8 U3=2+2*(U*U2+U1^2)
9 U4=2*(U*U3+3*U1*U2)
10 U5=2*(U*U4+4*U1*U3+3*U2^2)
11 U6=2*(U*U5+5*U1*U4+10*U2*U3)
12 U7=2*(U*U6+6*U1*U5+15*U2*U4+10*U3^2)
13 U8=0;
14 U9=0;
15 U10=0;
16 U11=2*(U*U10+10*U1*U9+45*U2*U8+120*U3*U7+210*U4*U6
      +126*U5^2)
17                                     // U11 is the 11th
                                     derivative of u
18
19
20 taylor(1)

```

check Appendix [AP 10](#) for dependency:

heun.sci

check Appendix [AP 9](#) for dependency:

modifiedeuler.sci

Scilab code Exa 6.17 solution to IVP by modified euler cauchy and heun

```

1 // example 6.17,
2 // caption: solve by 1) modified euler cauchy, 2) heun
  method
3
4 // h=0.2
5 // 1) modified euler cauchy method,
6
7 // u'=f(t,u)
8 // u'=-2tu^2
9 def f(' [z]=f(t,u)', 'z=-2*t*u^2');
10

```

```

11 modifiedeuler(1,0,.4,.2,f)          // calling the
    function ,
12
13 // 2) heun method ,
14 deff ( ' [z]=f (t , u) ' , ' z=-2*t*u^2 ' ) ;
15
16
17 heun(1,0,.4,.2,f)          // calling the function ,

```

check Appendix [AP 8](#) for dependency:

RK4.sci

Scilab code Exa 6.18 solution ti IVP by fourth order range kutta method

```

1 // example 6.18 ,
2 // caption: use of 4th order runge kutta method ,
3
4 // u'=f(t,u)
5 // u'=-2tu^2
6 deff ( ' [z]=f (t , u) ' , ' z=-2*t*u^2 ' ) ;
7
8 RK4(1,0,.4,.2,f)          // calling the function ,

```

check Appendix [AP 6](#) for dependency:

Vsim_eulercauchy.sce

check Appendix [AP 7](#) for dependency:

simRK4.sci

Scilab code Exa 6.20 solution to the IVP systems

```

1 // example no. 6.20,
2 // caption: solve the system of equations
3
4 //1) eulercauchy method solving simultaneous ODE
5
6 def f1(t,u,v) = -3*u+2*v;
7 def f2(t,u,v) = 3*u-4*v;
8
9
10 [u,v,t] = simeulercauchy(0,.5,0,.4,.2,f1,f2)
11
12 // 2) RK4 method solving simultaneous ODE
13
14
15
16 [u,v,t]=simRK4(0,.5,0,.4,.2,f1,f2)

```

check Appendix AP 5 for dependency:

newtonrap.sce

Scilab code Exa 6.21 solution to IVP by second order range kutta method

```

1 // example no. 6.21,
2 // caption: solving the IVP by implicit RK2 method
3
4 // u'=f(t,u)
5 // u'=-2tu^2
6 //u(0)=1,h=0.2;
7 t0=0;h=0.2;tn=.4;u0=1;
8 def f(t,u) = -2*t*u^2;
9 umaxAllowed = 1e+100;
10
11 t = [t0:h:tn]; u = zeros(t); n = length(u); u(1) =
    u0;
12

```

```

13 for j = 1:n-1
14     // k1=h*f(t(j)+h/2,u(j)+k1/2);
15     // considering the IVP we can infer that the
16     // above expression is non linear in k1,
17     // hence we use newton raphson method to solve for k1
18     ;
19     def f(' [w]=F(u2) ', 'w=k1+h*(2*t(j)+h)*(u(j)+k1/2)^2')
20     // u2=u(2)
21     def f(' [x]=Fp(u2) ', 'x=1+h*(2*t(j)+h)*(u(j)+k1/2)')
22     ;
23     k1=h*f(t(j),u(j));
24     newton(k1,F,Fp);
25     u(j+1) = u(j) +k1
26     disp(u(j+1))
27 end;

```

check Appendix [AP 4](#) for dependency:

adamsbashforth3.sci

Scilab code Exa 6.25 solution to IVP by third order adamsbashfort meth

```

1 // example 6.25
2 // caption: solving the IVP by adams-bashforth 3rd
3 // order method.
4 // u'=f(t,u)
5 // u'=-2tu^2
6 // u(0)=1,h=0.2;
7 def f(' [z]=f(t,u) ', 'z=-2*t*u^2');
8 adamsbashforth3(1,0,1,.2,f) // calling the
9 function ,

```

Scilab code Exa 6.27 solution to IVP by third order adams moult method

```

1 // example 6.27
2 // solving IVP by 3rd order adams moulton
3 // u'=t^2+u^2,    u(1)=2,
4 // h=0.1,        [1,1.2]
5 def f(' [z]=f(t,u)', 'z=t^2+u^2');
6 t0=1;u0=2;h=0.1;tn=1.2;
7 // third order adams moulton method,
8 // u(j+2)=u(j+1)+(h/12)*(5*f(t(j+2),u(j+2))+8*f(t(j
    +1),u(j+1))-f(t(j),u(j)));- is the expression
    for adamsbas-moulton3
9
10
11 // on observing the IVP we can infer that this
    would be a non linear equation,
12 // u(j+2)=u(j+1)+(h/12)*(5*((t(j+2))^2+(u(j+2))^2)
    +8*((t(j+1))^2+(u(j+1))^2)-((t(j))^2+(u(j))^2))
13
14 t = [t0:h:tn]; u = zeros(t); n = length(u); u(1) =
    u0;
15 for j = 1:n-2
16     if j==1 then
17         k1=h*f(t(j),u(j));
18         k2=h*f(t(j)+h,u(j)+k1);
19         u(j+1) = u(j) + (k2+k1)/2;
20         disp(u(j+1))
21     end;
22 end;
23
24 // hence the third order adams moulton expression
    turns to be,
25 // u(2)= 0.041667*(u(2))^2+3.194629
26 // let us use newton raphsom method to solve this,

```

```

27 def f(' [w]=F(u2)', 'w=-u2+ 0.041667*(u2)^2+3.194629')
    // u2=u(2)
28 def f(' [x]=Fp(u2)', 'x=-1+ 0.041667*2*u2')
29
30 // let us assume the initial guess of u(2)=u(1);
31
32 newton(2.633333, F, Fp)

```

Scilab code Exa 6.32 solution by numerov method

```

1 // example 6.32
2 // caption: solving the IVP by numerov method
3 // u''=(1+t^2)*u
4 // u(0)=1, u'(0)=0, [0,1]
5 // h=0.2,
6
7 // expression for numerov method is
8 // u(j+1)-2*u(j)+u(j-1)=(h^2/12)*(u''(j+1)+10*u''(j)+
    u''(j-1));
9
10 // observing the IVP we can reduce the numerov
    method to
11 // u(2)=2*u(1)-u(0)+(.2^2/12)*(1.16*u(2)+10.4*u(1)+1)
    ; for j=1
12 // u(3)=2*u(2)-u(1)+(.2^2/12)*(1.36*u(3)+11.6*u(2)
    +1.04*u(1)); for j=2
13 // u(4)=2*u(3)-u(2)+(.2^2/12)*(1.64*u(4)+13.6*u(3)
    +1.16*u(2)); for j=3
14 // u(5)=2*u(4)-u(3)+(.2^2/12)*(2*u(5)+16.4*u(4)
    +1.36*u(3)); for j=4
15
16 // from taylor series expansion we observe that
17 u1=1.0202; u0=1;
18 // u2-(.2^2/12)*(1.16*u2)=2*u1-u0+(.2^2/12)*(10.4*u1
    +1);

```

```
19 u2=(1/.9961333)*2*u1-u0+(.2^2/12)*(10.4*u1+1)
20
21 u3=(1/.995467)*(2.038667*u2-.996533*u1)
22
23 u4=(1/.994533)*(2.045333*u3-.996133*u2)
24
25 u5=(1/.993333)*(2.054667*u4-.995467*u3)
```

Chapter 7

ORDINARY DIFFERENTIAL EQUATIONS BOUNDARY VALUE PROBLEM

check Appendix [AP 1](#) for dependency:

```
shooting.sci
```

Scilab code Exa 7.1 solution to the BVP by shooting method

```
1 // example 7.1
2 // solve by shooting method;
3
4 // u' = u + 1;
5 // u(0) = 0; u(1) = %e - 1;
6
7 // let -> U1(x) = du/dx;
8 // U2(x) = d2u/dx2;
9
10 // U(x) = [U1(x); U2(x)]
11
12 // hence ;
13 // dU/dx = f(x, U);
```



```

14
15
16
17 def f(' [w]=f(x,U) ', 'w=[U(2); U(1)+1] ')
18
19 h=0.25;
20 x=[0:h:1];
21 ub=[0,%e-1];
22 up=[0:1:10];
23
24
25 [U] = shooting(ub,up,x,f);
26
27 // the solution obtained would show the values of u
    and their derivatives at various x taken in
    regular intervals of h;

```

check Appendix AP 1 for dependency:

shooting.sci

Scilab code Exa 7.3 solution to the BVP by shooting method

```

1 // example 7.3
2 // solve by shooting method;
3
4 // u' = 2*u*u';
5 // u(0) = 0.5; u(1) = 1;
6
7 // let -> U1(x) = du/dx;
8 // U2(x) = d2u/dx2;
9
10 // U(x) = [U1(x); U2(x)]
11
12 // hence ;
13 // dU/dx = f(x,U);

```

```

14
15 h=.25;
16
17 ub=[.5,1];
18
19 up=[0:.1:1];
20
21 x=0:h:1;
22
23 def f(' [w]=f(x,U) ', 'w=[U(2); 2*U(1)*U(2)] ')
24
25
26
27 [U] = shooting(ub,up,x,f);
28
29 // the solution obtained would show the values of u
    in the first collumn and their corresponding
    derivatives in the second collumn ;

```

check Appendix [AP 1](#) for dependency:

shooting.sci

Scilab code Exa 7.4 solution to the BVP by shooting method

```

1 // example 7.4
2 // solve by shooting method;
3
4 // u''=2*u*u';
5 // u(0)=0.5; u(1)=1;
6
7 // let -> U1(x)=du/dx;
8 // U2(x)=d2u/dx2;
9
10 // U(x)=[U1(x);U2(x)]
11

```

```

12 // hence ;
13 // dU/dx=f(x,U);
14
15 h=.25;
16
17 ub=[.5,1];
18
19 up=[0:.1:1];
20
21 x=0:h:1;
22
23 def f('w)=f(x,U)', 'w=[U(2); 2*U(1)*U(2)]')
24
25
26 [U] = shooting(ub,up,x,f);
27
28 // the solution obtained would show the values of u
    in the first column and their corresponding
    derivatives in the second column ;

```

Scilab code Exa 7.5 solution to the BVP

```

1 // example 7.5
2 // solve the boundary value problem      u''=u+x;
3 // u(x=0)=u(0)=0;    u(x=1)=u(4)=0;      h=1/4;
4
5
6 // we know;      u''=(u(j-1)-2*u(j)+u(j+1))/h^2;
7
8 // 1) second order method;
9 x=0:1/4:1;
10 u0=0;
11 u4=0;
12 u1_3 = rand(1,3)
13 u=[u0 u1_3 u4];

```

```

14 // hence;
15 disp(' (u(j-1)-2*u(j)+u(j+1))/h^2=u(j)+x(j) ')
    // for j=1,2,3;
16
17 disp(' for j=1                -16*u0+33*u1-16*u2=-.25 ')
18
19 disp(' for j=2                -16*u1+33*u2-16*u3=-.50 ')
20
21 disp(' for j=3                -16*u2+33*u3-16*u4=-.75 ')
22
23 // hence solving for u1,u2,u3)    ,
24 u1=-.034885;
25 u2=-.056326;
26 u3=-.050037;
27
28 disp(x);
29 disp(u);
30
31 // 2) numerov method;
32 x=0:1/4:1;
33 u0=0;
34 u4=0;
35 u=[u0 u1 u2 u3 u4];
36 // since according to numerov method we get the
    following system of equations;
37 disp(' (191*u(j-1)-394*u(j)+191*u(j+1)=x(j-1)+10*x(j)
    +x(j+1) ') // for j=1,2,3;
38
39 disp(' for j=1                191*u0-394*u1+191*u2=3 ')
40
41 disp(' for j=2                191*u1-394*u2+191*u3=6 ')
42
43 disp(' for j=3                191*u2-394*u3+191*u4=9 ')
44
45 // hence solving for u1,u2,u3    ,
46 u1=-.034885
47 u2=-.056326
48 u3=-.050037

```

```

49
50
51 disp(x);
52 disp(u);

```

Scilab code Exa 7.6 solution to the BVP by finite differences

```

1 // example 7.6
2 // solve the boundary value problem      u''=u*x;
3 // u(0)+u'(0)=1;   u(x=1)=0;           h=1/3;
4
5
6 // we know;      u''=(u(j-1)-2*u(j)+u(j+1))/h^2;
7
8 // 1) second order method;
9 x=0:1/3:1;
10 u1_2 = rand(1,3)
11 u3=1;
12 u=[u1_2 u3];
13 // hence;
14 disp(' (u(j-1)-2*u(j)+u(j+1))/h^2=u(j)*x(j) ')
    // for j=0,1,2,3;
15
16 disp(' for j=0          u1'-2*u0+u1=0 ')
    // u1'=u(-1)
17
18 disp(' for j=1          u0-2*u1+u2=(1/27)u1 ')
19
20 disp(' for j=2          u1-2*u2+u3=(2/27)u2 ')
21
22 // we know;      u'=(u(j+1)-u(j-1))/2h
23 // hence eliminating u1!
24 // solving for u0,u1,u2,u3 ,
25 u0=-.9879518;
26 u1=-.3253012;

```

```

27 u2=-.3253012;
28
29 disp(x);
30 disp(u);

```

Scilab code Exa 7.11 solution to the BVP by finite differences

```

1 // example 7.11
2 // solve the boundary value problem      u''=u'+1;
3 // u(0)=1;    u(x=1)=2(%e-1);          h=1/3;
4
5
6 // we know;      u''=(u(j-1)-2*u(j)+u(j+1))/h^2;
7 // we know;      u'=(u(j+1)-u(j-1))/2h;
8
9 // 1) second order method;
10 x=0:1/3:1;
11
12 u= rand(1,4);;
13 // hence;
14 disp(' (u(j-1)-2*u(j)+u(j+1))/h^2=((u(j+1)-u(j-1))/2h
      )+1')          // for j=1,2;
15
16
17 disp(' for j=1          (7/6)*u0-2*u1+(5/6)*u2
      =(1/9)')
18
19 disp(' for j=2          (7/6)*u1-2*u2+(5/6)*u3
      =(1/9)')
20
21
22 // hence eliminating u1!
23 // solving for u1,u2,
24 u0=1;
25 u3=2*(%e-1);

```

```
26 u1=1.454869;  
27 u2=2.225019;  
28  
29 disp(x);  
30 disp(u);
```

Appendix

Scilab code AP 1 shooting method for solving BVP

```
1 function [U] = shooting(ub,up,x,f)
2
3 //Shooting method for a second order
4 //boundary value problem
5 //ub = [u0 u1] -> boundary conditions
6 //x = a vector showing the range of x
7 //f = function defining ODE, i.e.,
8 //    du/dx = f(x,u), u = [u(1);u(2)].
9 //up = vector with range of du/dx at x=x0
10 //xuTable = table for interpolating derivatives
11 //uderiv = derivative boundary condition
12
13 n = length(up);
14 m = length(x);
15 y1 = zeros(up);
16
17 for j = 1:n
18     u0 = [ub(1);up(j)];
19     uu = ode(u0,x(1),x,f);
20     u1(j) = uu(1,m);
21 end;
22
23 xuTable = [u1';up];
24 uderiv = interpoln(xuTable,ub(2));
25 u0 = [ub(1);uderiv];
26 u = ode(u0,x(1),x,f);
```



```
27 U=u';
28
29 endfunction
```

Scilab code AP 2 euler method

```
1 function [u,t] = Euler1(u0,t0,tn,h,f)
2
3 //Euler 1st order method solving ODE
4 // du/dt = f(u,t), with initial
5 //conditions u=u0 at t=t0. The
6 //solution is obtained for t = [t0:h:tn]
7 //and returned in u
8
9 umaxAllowed = 1e+100;
10
11 t = [t0:h:tn]; u = zeros(t); n = length(u); u(1) =
    u0;
12
13 for j = 1:n-1
14     u(j+1) = u(j) + h*f(t(j),u(j));
15     if u(j+1) > umaxAllowed then
16         disp('Euler 1 - WARNING: underflow or
            overflow ');
17         disp('Solution sought in the following range:
            ');
18         disp([t0 h tn]);
19         disp('Solution evaluated in the following
            range: ');
20         disp([t0 h t(j)]);
21         n = j; t = t(1,1:n); u = u(1,1:n);
22         break;
23     end;
24 end;
25
26 endfunction
```

Scilab code AP 3 eigen vectors and eigen values

```

1 function [x,lam] = geigenvectors(A,B)
2
3 //Calculates unit eigenvectors of matrix A
4 //returning a matrix x whose columns are
5 //the eigenvectors. The function also
6 //returns the eigenvalues of the matrix.
7
8 [nA,mA] = size(A);
9 [nB,mB] = size(B);
10
11 if (mA<>nA | mB<>nB) then
12     error('geigenvectors - matrix A or B not square'
13         );
14     abort;
15 end;
16 if nA<>nB then
17     error('geigenvectors - matrix A and B have
18         different dimensions');
19     abort;
20 end;
21 lam = poly(0,'lam'); //Define variable "lam
22     "
23 chPoly = det(A-B*lam); //Characteristic
24     polynomial
25 lam = roots(chPoly)'; //Eigenvalues of
26     matrix A
27
28 x = []; n = nA;
29
30 for k = 1:n
31     BB = A - lam(k)*B; //Characteristic matrix
32     CC = BB(1:n-1,1:n-1); //Coeff. matrix for
33         reduced system
34     bb = -BB(1:n-1,n); //RHS vector for
35         reduced system
36     y = CC\b; //Solution for reduced system

```

```

32     y = [y;1];           //Complete eigenvector
33
34     x = [x y];           //Add eigenvector to matrix
35 end;
36
37 endfunction

```

Scilab code AP 4 adams bashforth third order method

```

1 function [u,t] = adamsbashforth3(u0,t0,tn,h,f)
2
3 //adamsbashforth3 3rd order method solving ODE
4 // du/dt = f(u,t), with initial
5 //conditions u=u0 at t=t0. The
6 //solution is obtained for t = [t0:h:tn]
7 //and returned in u
8
9 umaxAllowed = 1e+100;
10
11 t = [t0:h:tn]; u = zeros(t); n = length(u); u(1) =
    u0;
12 for j = 1:n-2
13 if j<3 then
14     k1=h*f(t(j),u(j));
15     k2=h*f(t(j)+h,u(j)+k1);
16     u(j+1) = u(j) + (k2+k1)/2;
17 end;
18
19 if j>=2 then
20     u(j+2) = u(j+1) + (h/12)*(23*f(t(j+1),u(j+1))
        -16*f(t(j),u(j))+5*f(t(j-1),u(j-1)));
21 end;
22 end;
23 endfunction

```

Scilab code AP 5 newton raphson method

```

1

```

```

2 function x=newton(x,f,fp)
3     R=5;
4     PE=10^-15;
5     maxval=10^4;
6
7     for n=1:1:R
8         x=x-f(x)/fp(x);
9
10        if abs(f(x))<=PE then break
11        end
12        if (abs(f(x))>maxval) then error('Solution
            diverges ');
13            abort
14            break
15        end
16    end
17    disp(n," no. of iterations =")
18 endfunction

```

Scilab code AP 6 euler cauchy solution to the simultaniouys equations

```

1 function [u,v,t] = simeulercauchy(u0,v0,t0,tn,h,f1,
    f2)
2
3
4 // du/dt = f1(t,u,v), dv/dt = f2(t,u,v) with
    initial
5 //conditions u=u0,v=v0 at t=t0. The
6 //solution is obtained for t = [t0:h:tn]
7 //and returned in u,v
8
9
10 umaxAllowed = 1e+100;
11
12 t = [t0:h:tn]; u = zeros(t);v= zeros(t); n = length(
    u); u(1) = u0;v(1)=v0;
13
14 for j = 1:n-1

```

```

15     k11=h*f1(t(j),u(j),v(j));
16     k21=h*f2(t(j),u(j),v(j));
17     k12=h*f1(t(j)+h,u(j)+k11,v(j)+k21);
18     k22=h*f2(t(j)+h,u(j)+k11,v(j)+k21);
19     u(j+1) = u(j) + (k11+k12)/2;
20     v(j+1) = v(j) + (k21+k22)/2;
21
22 end;
23
24 endfunction

```

Scilab code AP 7 simultaneous fourth order range kutta

```

1 function [u,v,t] = simRK4(u0,v0,t0,tn,h,f1,f2)
2
3 // RK4 method solving simultanious ODE
4 // du/dt = f1(t,u,v), dv/dt = f2(t,u,v) with
   initial
5 //conditions u=u0,v=v0 at t=t0. The
6 //solution is obtained for t = [t0:h:tn]
7 //and returned in u,v
8
9 umaxAllowed = 1e+100;
10
11 t = [t0:h:tn]; u = zeros(t);v=zeros(t) ;n = length(u
   ); u(1) = u0;v(1)=v0
12
13 for j = 1:n-1
14     k11=h*f1(t(j),u(j),v(j));
15     k21=h*f2(t(j),u(j),v(j));
16     k12=h*f1(t(j)+h/2,u(j)+k11/2,v(j)+k21/2);
17     k22=h*f2(t(j)+h/2,u(j)+k11/2,v(j)+k21/2);
18     k13=h*f1(t(j)+h/2,u(j)+k12/2,v(j)+k22/2);
19     k23=h*f2(t(j)+h/2,u(j)+k12/2,v(j)+k22/2);
20     k14=h*f1(t(j)+h,u(j)+k13,v(j)+k23);
21     k24=h*f2(t(j)+h,u(j)+k13,v(j)+k23);
22     u(j+1) = u(j) + (1/6)*(k11+2*k12+2*k13+k14);
23     v(j+1) = v(j) + (1/6)*(k21+2*k22+2*k23+k24);

```

```

24
25 end;
26
27 endfunction

```

Scilab code AP 8 fourth order range kutta method

```

1 function [u,t] = RK4(u0,t0,tn,h,f)
2
3 // RK4 method solving ODE
4 // du/dt = f(u,t), with initial
5 //conditions u=u0 at t=t0. The
6 //solution is obtained for t = [t0:h:tn]
7 //and returned in u
8
9 umaxAllowed = 1e+100;
10
11 t = [t0:h:tn]; u = zeros(t); n = length(u); u(1) =
    u0;
12
13 for j = 1:n-1
14     k1=h*f(t(j),u(j));
15     k2=h*f(t(j)+h/2,u(j)+k1/2);
16     k3=h*f(t(j)+h/2,u(j)+k2/2);
17     k4=h*f(t(j)+h,u(j)+k3);
18     u(j+1) = u(j) + (1/6)*(k1+2*k2+2*k3+k4);
19     if u(j+1) > umaxAllowed then
20         disp('Euler 1 - WARNING: underflow or
                overflow ');
21         disp('Solution sought in the following range:
                ');
22         disp([t0 h tn]);
23         disp('Solution evaluated in the following
                range: ');
24         disp([t0 h t(j)]);
25         n = j; t = t(1,1:n); u = u(1,1:n);
26         break;
27     end;

```

```

28 end;
29
30 endfunction

```

Scilab code AP 9 modified euler method

```

1 function [u,t] = modifiedeuler(u0,t0,tn,h,f)
2
3 //modifiedeuler 1st order method solving ODE
4 // du/dt = f(u,t), with initial
5 //conditions u=u0 at t=t0. The
6 //solution is obtained for t = [t0:h:tn]
7 //and returned in u
8
9 umaxAllowed = 1e+100;
10
11 t = [t0:h:tn]; u = zeros(t); n = length(u); u(1) =
    u0;
12
13 for j = 1:n-1
14     k1=h*f(t(j),u(j));
15     k2=h*f(t(j)+h/2,u(j)+k1/2);
16     u(j+1) = u(j) + k2;
17     if u(j+1) > umaxAllowed then
18         disp('Euler 1 - WARNING: underflow or
                overflow');
19         disp('Solution sought in the following range:
                ');
20         disp([t0 h tn]);
21         disp('Solution evaluated in the following
                range: ');
22         disp([t0 h t(j)]);
23         n = j; t = t(1,1:n); u = u(1,1:n);
24         break;
25     end;
26 end;
27
28 endfunction

```

Scilab code AP 10 euler cauchy or heun

```
1 function [u,t] = heun(u0,t0,tn,h,f)
2
3 //heun method solving ODE
4 // du/dt = f(u,t), with initial
5 //conditions u=u0 at t=t0. The
6 //solution is obtained for t = [t0:h:tn]
7 //and returned in u
8
9 umaxAllowed = 1e+100;
10
11 t = [t0:h:tn]; u = zeros(t); n = length(u); u(1) =
    u0;
12
13 for j = 1:n-1
14     k1=h*f(t(j),u(j));
15     k2=h*f(t(j)+h,u(j)+k1);
16     u(j+1) = u(j) + (k2+k1)/2;
17     if u(j+1) > umaxAllowed then
18         disp('Euler 1 - WARNING: underflow or
                overflow ');
19         disp('Solution sought in the following range:
                ');
20         disp([t0 h tn]);
21         disp('Solution evaluated in the following
                range: ');
22         disp([t0 h t(j)]);
23         n = j; t = t(1,1:n); u = u(1,1:n);
24         break;
25     end;
26 end;
27
28 endfunction
```

Scilab code AP 11 taylor series


```

1 function u=taylor(t)
2     u=(t^1*U1)/fact(1)+(t^2*U2)/fact(2)+(t^3*U3)/fact
      (3)+(t^4*U4)/fact(4)+(t^5*U5)/fact(5)+(t^6*U6)
      /fact(6)+(t^7*U7)/fact(7)+(t^8*U8)/fact(8)+(t
      ^9*U9)/fact(9)+(t^10*U10)/fact(10)+(t^11*U11)/
      fact(11)
3 endfunction

```

Scilab code AP 12 factorial

```

1 function x=fact(n)
2     x=1;
3     for i=2:1:n
4         x=x*i;
5     end;
6 endfunction

```

Scilab code AP 13 mid point nmethod

```

1 function [u] = eulermidpoint(u0,t0,tn,h,f,fp)
2
3 //midpoint 1st order method solving ODE
4 // du/dt = f(u,t), with initial
5 //conditions u=u0 at t=t0. The
6 //solution is obtained for t = [t0:h:tn]
7 //and returned in u
8
9 umaxAllowed = 1e+100;
10
11 t = [t0:h:tn]; u = zeros(t); n = length(u); u(1) =
      u0;
12 u(2)=u(1)+h*f(t(1),u(1))+(h^2/2)*fp(t(1),u(1));
13 for j = 2:n-1
14     u(j+1) = u(j-1) + 2*h*f(t(j),u(j));
15     if u(j+1) > umaxAllowed then
16         disp('Euler 1 - WARNING: underflow or
              overflow');

```

```

17         disp('Solution sought in the following range:
           ');
18         disp([t0 h tn]);
19         disp('Solution evaluated in the following
           range: ');
20         disp([t0 h t(j)]);
21         n = j; t = t(1,1:n); u = u(1,1:n);
22         break;
23     end;
24 end;
25
26 endfunction

```

Scilab code AP 14 back euler method

```

1 function [u] = backeuler(u0,t0,tn,h,f)
2
3 //backeuler 1st order method solving ODE
4 // du/dt = f(u,t), with initial
5 //conditions u=u0 at t=t0. The
6 //solution is obtained for t = [t0:h:tn]
7 //and returned in u
8
9 umaxAllowed = 1e+100;
10
11 t = [t0:h:tn]; u = zeros(t); n = length(u); u(1) =
    u0;
12
13 for j=1:n-1
14     u(j+1)=u(j);
15     for i = 0:5
16         u(j+1) = u(j) + h*f(t(j+1),u(j+1));
17         i=i+1;
18     end;
19 end;
20
21
22 endfunction

```

Scilab code AP 15 composite trapezoidal rule

```
1 function I=comptrapezoidal(x,h,f)
2     //This function calculates the numerical
3     //integration of f(x)dx
4     //between limits x(1) and x(n) using composite
5     //trapezoidal rule
6     //Check that x and y have the same size (which must
7     //be an odd number)
8     //Also, the values of x must be equally spaced with
9     //spacing h
10    y=feval(x,f);
11    [nrx,ncx]=size(x)
12    [nrf,ncf]=size(y)
13    if ((nrx<>1)|(nrf<>1)) then
14        error('x or f, or both, not column vector(s)');
15        abort;
16    end;
17    if ((ncx<>ncf)) then
18        error('x and f are not of the same length');
19        abort;
20    end;
21    //check that the size of the lists xL and f is odd
22    if (modulo(ncx,2)==0) then
23        disp(ncx,"list size =")
24        error('list size must be an odd number');
25        abort
26    end;
27    n = ncx;
28
29    I = f(x(1)) + f(x(n));
30    for j = 2:n-1
31        if(modulo(j,2)==0) then
32            I = I + 2*f(x(j));
33        else
34            I = I + 2*f(x(j));
35        end;
36    end;
```

```

31     end;
32 end;
33 I = (h/2.0)*I
34 endfunction

```

Scilab code AP 16 simpson rule

```

1 function [I] = simpson13(x,h,f)
2 //This function calculates the numerical integration
  of f(x)dx
3 //between limits x(1) and x(n) using Simpson's 1/3
  rule
4 //Check that x and y have the same size (which must
  be an odd number)
5 //Also, the values of x must be equally spaced with
  spacing h
6 y=feval(x,f);
7 [nrx,ncx]=size(x)
8 [nrf,ncf]=size(y)
9 if ((nrx<>1)|(nrf<>1)) then
10     error('x or f, or both, not column vector(s)');
11     abort;
12 end;
13 if ((ncx<>ncf)) then
14     error('x and f are not of the same length');
15     abort;
16 end;
17 //check that the size of the lists xL and f is odd
18 if (modulo(ncx,2)==0) then
19     disp(ncx,"list size =")
20     error('list size must be an odd number');
21     abort
22 end;
23 n = ncx;
24
25 I = f(x(1)) + f(x(n));
26 for j = 2:n-1
27     if(modulo(j,2)==0) then

```

```

28         I = I + 4*f(x(j));
29     else
30         I = I + 2*f(x(j));
31     end;
32 end;
33 I = (h/3.0)*I
34 endfunction

```

Scilab code AP 17 simpson rule

```

1 function [I] = simpson38(x,f)
2 //This function calculates the numerical integration
   of f(x)dx
3 //between limits x(1) and x(n) using Simpson's 3/8
   rule
4 //Check that x and f have the same size (which must
   be of the form 3*i+1,
5 //where i is an integer number)
6 //Also, the values of x must be equally spaced with
   spacing h
7
8 y=feval(x,f);
9 [nrx,ncx]=size(x)
10 [nrf,ncf]=size(y)
11 if ((nrx<>1)|(nrf<>1)) then
12     error('x or f, or both, not column vector(s)');
13     abort;
14 end;
15 if ((ncx<>ncf)) then
16     error('x and f are not of the same length');
17     abort;
18 end;
19 //check that the size of the lists xL and f is odd
20 if (modulo(ncx-1,3)<>0) then
21     disp(ncx,"list size =")
22     error('list size must be of the form 3*i+1,
   where i=integer');
23     abort

```

```

24 end;
25 n = ncx;
26 xdifff = mtlb_diff(x);
27 h = xdifff(1,1);
28 I = f(x(1)) + f(x(n));
29 for j = 2:n-1
30     if(modulo(j-1,3)==0) then
31         I = I + 2*f(x(j));
32     else
33         I = I + 3*f(x(j));
34     end;
35 end;
36 I = (3.0/8.0)*h*I
37 endfunction

```

Scilab code AP 18 approximation to the integral by simpson method

```

1 function I=simpson(a,b,f)
2     I=((b-a)/6)*(f(a)+4*f((a+b)/2)+f(b));
3 endfunction

```

Scilab code AP 19 integration by trapizoidal method

```

1 // solves the definite integral by the trapezoidal
  rule ,
2 // given the limits a,b and the function f,
3 // returns the integral value I
4
5 function I =trapezoidal(a,b,f)
6     I=((b-a)/2)*(f(a)+f(b));
7 endfunction

```

Scilab code AP 20 jacobian of a given matrix

```

1 function J= jacobianmat (f1,f2,h,k)
2     J=zeros(2,2);
3 J(1,1)=(f1(1+h,1)-f1(1,1))/2*h;

```

```

4
5 J(1,2)=(f1(1,1+k)-f1(1,1))/2*k;
6 J(2,1)=(f2(1+h,1)-f2(1,1))/2*h;
7 J(2,2)=(f2(1,1+k)-f2(1,1))/2*k;
8 endfunction

```

Scilab code AP 21 linear interpolating polinomial

```

1 function fp=linearinterpol(xL,f)
2     fp=(f(2)-f(1))/(xL(2)-xL(1));
3 endfunction;

```

Scilab code AP 22 iterated interpolation

```

1 function [L012,L02,L01]=iteratedinterpol (x,f,n)
2     X=poly(0,"X");
3     L01=(1/(x(2)-x(1)))*det([f(1) x(1)-X;f(2) x(2)-X
4         ]);
5     L02=(1/(x(3)-x(1)))*det([f(1) x(1)-X;f(3) x(3)-X
6         ]);
7     L012=(1/(x(3)-x(2)))*det([L01 x(2)-X;L02 x(3)-X
8         ]);
9
10 endfunction

```

Scilab code AP 23 newton divided differences interpolation order two

```

1 function P2=NDDinterpol2 (x,f)
2     X=poly(0,"X");
3     f01=(f(2)-f(1))/(x(2)-x(1));
4     f13=(f(3)-f(2))/(x(3)-x(2));
5     f013=(f13-f01)/(x(3)-x(1));
6     P2=f(1)+(X-x(1))*f01+(X-x(1))*(X-x(2))*f013;
7 endfunction

```

Scilab code AP 24 legrange fundamental polynomial

```

1 function P2=lagrangefundamentalpoly(x,f,n)
2     [nrx,ncx]=size(x)
3     [nrf,ncf]=size(f)
4 if ((nrx<>1)|(nrf<>1)) then
5     error('x or f, or both, not column vector(s)');
6     abort;
7 end;
8 if ((ncx<>ncf)) then
9     error('x and f are not of the same length');
10    abort;
11 end;
12
13 X=poly(0,"X");
14 L=zeros(n);
15
16 P2=0;
17 for i=1:n+1
18     L(i)=1;
19     for j=1:n+1
20         if i~=j then
21             L(i)=L(i)*(X-x(j))/(x(i)-x(j))
22         end;
23     end;
24 P2=P2+L(i)*f(i);
25 end;
26
27
28 endfunction

```

Scilab code AP 25 legrange interpolation

```

1 function P1=legrangeinterpol (x0,x1,f0,f1)
2     x=poly(0,"x");
3     L0=(x-x1)/(x0-x1);
4     L1=(x-x0)/(x1-x0);
5     P1=L0*f0+L1*f1;
6 endfunction

```

Scilab code AP 26 quadratic approximation

```
1 function [P]=quadraticapprox(x,f)
2
3 n=length(x);m=length(f);
4 if m<>n then
5     error('linreg - Vectors x and f are not of the
6         same length. ');
7     abort;
8 end;
9 s1=0;
10 s2=0;
11 for i=1:n
12     s1=s1+x(i)*f(i);
13     s2=s2+x(i)^2*f(i);
14 end
15 c0=det([sum(f) sum(x) sum(x^2);s1 sum(x^2) sum(x^3);
16     s2 sum(x^3) sum(x^4)])/det([n sum(x) sum(x^2);
17     sum(x) sum(x^2) sum(x^3); sum(x^2) sum(x^3) sum(x
18     ^4)]);
19
20 c1=det([n sum(f) sum(x^2);sum(x) s1 sum(x^3); sum(x
21     ^2) s2 sum(x^4)])/det([n sum(x) sum(x^2);sum(x)
22     sum(x^2) sum(x^3); sum(x^2) sum(x^3) sum(x^4)]);
23
24 c2=det([n sum(x) sum(f);sum(x) sum(x^2) s1; sum(x^2)
25     sum(x^3) s2])/det([n sum(x) sum(x^2);sum(x) sum(
26     x^2) sum(x^3); sum(x^2) sum(x^3) sum(x^4)]);
27
28 X=poly(0,"X");
29 P=c2*X^2+c1*X+c0;
30 endfunction
```

Scilab code AP 27 straight line approximation

```
1 function [P]=straightlineapprox(x,f)
2
3 n=length(x);m=length(f);
```

```

4  if m<>n then
5      error('linreg - Vectors x and f are not of the
           same length. ');
6      abort;
7  end;
8  s=0;
9  for i=1:n
10     s=s+x(i)*f(i);
11 end
12 c0=det([sum(f) sum(x); s sum(x^2)])/det([n sum(x);
           sum(x) sum(x^2)]);
13 c1=det([ n sum(f); sum(x) s])/det([n sum(x);sum(x)
           sum(x^2)]);
14 X=poly(0,"X");
15 P=c1*X+c0;
16 endfunction

```

Scilab code AP 28 aitken interpolation

```

1  function P1=aitkeninterpol (x0,x1,f0,f1)
2      x=poly(0,"x");
3      P1=(1/(x1-x0))*det([f0 x0-x;f1 x1-x]);
4  endfunction

```

Scilab code AP 29 newton divided differences interpolation

```

1  function P1=NDDinterpol (x0,x1,f0,f1)
2      x=poly(0,"x");
3      f01=(f1-f0)/(x1-x0);
4      P1=f0+(x-x0)*f01;
5  endfunction

```

Scilab code AP 30 hermite interpolation

```

1  function P= hermiteinterpol(x,f,fp)
2      X=poly(0,"X");
3      function f0=L0(X)

```

```

4
5   f0=(X-x(2))*(X-x(3))/((x(1)-x(2))*(x(1)-x(3)))
6   endfunction;
7   a0=[1-2*(X-x(1))*numdiff(L0,x(1))];
8   L0=(X-x(2))*(X-x(3))/((x(1)-x(2))*(x(1)-x(3)));
9   A0=a0*L0*L0;
10  disp(A0)
11  B0=(X-x(1))*L0^2;
12
13  X=poly(0,"X");
14
15  function f1=L1(X)
16
17  f1=(X-x(1))*(X-x(3))/((x(2)-x(1))*(x(2)-x(3)))
18  endfunction;
19  a1=[1-2*(X-x(2))*0];
20  L1=(X-x(1))*(X-x(3))/((x(2)-x(1))*(x(2)-x(3)));
21  A1=a1 *L1*L1;
22  disp(A1)
23  B1=(X-x(2))*L1^2;
24  function f2=L2(X)
25
26  f2=(X-x(1))*(X-x(2))/((x(3)-x(1))*(x(3)-x(2)))
27  endfunction;
28  a2=[1-2*(X-x(3))*numdiff(L2,x(3))];
29  L2=(X-x(1))*(X-x(2))/((x(3)-x(1))*(x(3)-x(2)));
30  A2=a2 *L2*L2;
31  disp(A2)
32  B2=(X-x(3))*L2^2;
33
34
35
36  P=A0*f(1)+A1*f(2)+A2*f(3)+B0*fp(1)+B1*fp(2)+B2*
    fp(3);
37 endfunction

```

Scilab code AP 31 newton backward differences polinomial

```

1 function [P]=NBDP(x,n,xL,f)
2 //This function calculates a Newton Forward-
   Difference Polynomial of
3 //order n, evaluated at x, using column vectors xL,
   f as the reference
4 //table. The first value of xL and of f, represent,
   respectively,
5 //xo and fo in the equation for the polynomial.
6 [m,nc]=size(f)
7 //check that it is indeed a column vector
8 if (nc<>1) then
9     error('f is not a column vector. ');
10    abort
11 end;
12 //check the difference order
13 if (n >= m) then
14     disp(n,"n=");
15     disp(m,"m=");
16     error('n must be less than or equal to m-1');
17    abort
18 end;
19 //
20 xo = xL(m,1);
21 delx = mtlb_diff(xL);
22 h = delx(1,1);
23 s = (x-xo)/h;
24 P = f(m,1);
25 delf = f;
26 disp(delf);
27 for i = 1:n
28     delf = mtlb_diff(delf);
29     [m,nc] = size(delf);
30     disp(delf);
31     P = P + Binomial(s+i-1,i)*delf(m,1)
32 end;
33 endfunction
34
35 function [C]=Binomial(s,i)

```

```

36     C = 1.0;
37     for k = 0:i-1
38         C = C*(s-k);
39     end;
40     C = C/factorial(i)
41 endfunction
42 function [fact]=factorial(nn)
43     fact = 1.0
44     for k = nn:-1:1
45         fact=fact*k
46     end;
47 endfunction

```

Scilab code AP 32 gauss jorden

```

1 function [M]=jorden(A,b)
2     M=[A b];
3     [ra,ca]=size(A);
4     [rb,cb]=size(b);
5     n=ra;
6     for p=1:1:n
7         for k=(p+1):1:n
8             if abs(M(k,p))>abs(M(p,p)) then
9                 M({p,k},:)=M({k,p},:);
10            end
11        end
12        M(p,:)=M(p,+)/M(p,p);
13        for i=1:1:p-1
14            M(i,:)=M(i,)-M(p,)*(M(i,p)/M(p,p));
15        end
16        for i=p+1:1:n
17            M(i,:)=M(i,)-M(p,)*(M(i,p)/M(p,p));
18        end
19    end
20 endfunction

```

Scilab code AP 33 gauss elimination with pivoting

```

1 function [x]=pivotgausselim(A,b)
2     M=[A b];
3     [ra,ca]=size(A);
4     [rb,cb]=size(b);
5     n=ra;
6     for p=1:1:n
7         for k=(p+1):1:n
8             if abs(M(k,p))>abs(M(p,p)) then
9                 M({p,k},:)=M({k,p},:);
10            end
11        end
12        for i=p+1:1:n
13            m(i,p)=M(i,p)/M(p,p);
14            M(i,:)=M(i,:)-M(p,:)*m(i,p);
15
16        end
17    end
18    a=M(1:n,1:n);
19    b=M(:,n+1);
20    for i = n:-1:1
21        sumj=0
22        for j=n:-1:i+1
23            sumj = sumj + a(i,j)*x(j);
24        end;
25        x(i)=(b(i)-sumj)/a(i,i);
26    end
27 endfunction

```

Scilab code AP 34 gauss elimination

```

1 function [x] = gausselim(A,b)
2
3 //This function obtains the solution to the system
  of
4 //linear equations  $A*x = b$ , given the matrix of
  coefficients A
5 //and the right-hand side vector, b
6

```

```

7 [nA,mA] = size(A)
8 [nb,mb] = size(b)
9
10 if nA<>mA then
11     error('gausselim - Matrix A must be square');
12     abort;
13 elseif mA<>nb then
14     error('gausselim - incompatible dimensions
15         between A and b');
16     abort;
17 end;
18 a = [A b];
19
20 //Forward elimination
21
22 n = nA;
23 for k=1:n-1
24     for i=k+1:n
25         for j=k+1:n+1
26             a(i,j)=a(i,j)-a(k,j)*a(i,k)/a(k,k);
27         end;
28     end;
29 end;
30
31 //Backward substitution
32
33 x(n) = a(n,n+1)/a(n,n);
34
35 for i = n-1:-1:1
36     sumk=0
37     for k=i+1:n
38         sumk=sumk+a(i,k)*x(k);
39     end;
40     x(i)=(a(i,n+1)-sumk)/a(i,i);
41 end;
42
43 endfunction

```

Scilab code AP 35 eigen vector and eigen value

```
1 function [x,lam] = geigenvectors(A,B)
2
3 //Calculates unit eigenvectors of matrix A
4 //returning a matrix x whose columns are
5 //the eigenvectors. The function also
6 //returns the eigenvalues of the matrix.
7
8 [nA,mA] = size(A);
9 [nB,mB] = size(B);
10
11 if (mA<>nA | mB<>nB) then
12     error('geigenvectors - matrix A or B not square'
13         );
14     abort;
15 end;
16 if nA<>nB then
17     error('geigenvectors - matrix A and B have
18         different dimensions');
19     abort;
20 end;
21 lam = poly(0,'lam'); //Define variable "lam
22
23 chPoly = det(A-B*lam); //Characteristic
24     polynomial
25 lam = roots(chPoly)'; //Eigenvalues of
26     matrix A
27
28 x = []; n = nA;
29
30 for k = 1:n
31     BB = A - lam(k)*B; //Characteristic matrix
32     CC = BB(1:n-1,1:n-1); //Coeff. matrix for
```



```

        reduced system
30     bb = -BB(1:n-1,n);           //RHS vector for
        reduced system
31     y = CC\bb;                   //Solution for reduced system
32     y = [y;1];                   //Complete eigenvector
33
34     x = [x y];                   //Add eigenvector to matrix
35 end;
36
37 endfunction

```

Scilab code AP 36 gauss siedel method

```

1 function [X]=gaussseidel(A,n,N,X,b)
2     L=A;
3     U=A;
4     D=A;
5     for i=1:1:n
6         for j=1:1:n
7             if j>i then L(i,j)=0;
8                 D(i,j)=0;
9             end
10            if i>j then U(i,j)=0;
11                D(i,j)=0;
12            end
13            if i==j then L(i,j)=0;
14                U(i,j)=0;
15            end
16        end
17
18    end
19    for k=1:1:N
20        X=(D+L)^-1*(-U*X+b);
21        disp(X)
22    end
23
24 endfunction

```

Scilab code AP 37 jacobi iteration method

```
1 function [X]=jacobiiteration(A,n,N,X,b)
2     L=A;
3     U=A;
4     D=A;
5     for i=1:1:n
6         for j=1:1:n
7             if j>i then L(i,j)=0;
8                 D(i,j)=0;
9             end
10            if i>j then U(i,j)=0;
11                D(i,j)=0;
12            end
13            if i==j then L(i,j)=0;
14                U(i,j)=0;
15            end
16        end
17    end
18    for k=1:1:N
19        X=-D^-1*(L+U)*X+D^-1*(b);
20    end
21 endfunction
22
23
```

Scilab code AP 38 back substitution

```
1 function [x] = back(U,Z)
2
3 x=zeros(1,n);
4 for i = n:-1:1
5     sumk=0
6     for j=i+1:n
7         sumk=sumk+U(i,j)*x(j);
8     end;
9     x(i)=(Z(i)-sumk)/U(i,i);
10 end;
```

```
11
12
13
14
15 endfunction
```

Scilab code AP 39 cholesky method

```
1 function L=cholesky (A,n)
2     L=zeros(n,n);
3     for k=1:1:n
4         S=0;
5         P=0;
6         for j=1:1:k-1
7             S=S+(L(k,j)^2);
8             P=P+L(i,j)*L(k,j)
9         end
10        L(k,k)=sqrt(A(k,k)-S);
11        for i=k+1:1:n
12            L(i,k)=(A(i,k)-P)/L(k,k);
13        end
14    end
15
16 endfunction
```

Scilab code AP 40 forward substitution

```
1 function x=fore(L,b)
2
3 for i = 1:1:n
4     sumk=0
5     for j=1:i-1
6         sumk=sumk+L(i,j)*x(j);
7     end;
8     x(i)=(b(i)-sumk)/L(i,i);
9 end;
10
11 endfunction
```

Scilab code AP 41 L and U matrices

```
1 function [U,L]=LandU(A,n)
2     U=A
3     L=eye(n,n)
4     for p=1:1:n-1
5         for i=p+1:1:n
6             m=A(i,p)/A(p,p);
7             L(i,p)=m;
8             A(i,:)=A(i,:)-m*A(p,:);
9             U=A;
10        end
11    end
12 endfunction
```

Scilab code AP 42 newton raphson method

```
1
2 function x=newton(x,f,fp)
3     R=100;
4     PE=10^-8;
5     maxval=10^4;
6
7     for n=1:1:R
8         x=x-f(x)/fp(x);
9         if abs(f(x))<=PE then break
10        end
11        if (abs(f(x))>maxval) then error('Solution
12        diverges');
13        abort
14        break
15    end
16    disp(n," no. of iterations =")
17 endfunction
```

Scilab code AP 43 four iterations of newton raphson method

```
1 function x=newton4(x,f,fp)
2     R=4;
3     PE=10^-15;
4     maxval=10^4;
5     for n=1:1:R
6         if fp(x)==0 then disp("select another
           initial root x0")
7         end
8         x=x-f(x)/fp(x);
9         if abs(f(x))<=PE then break
10        end
11        if (abs(f(x))>maxval) then error('Solution
           diverges ');
12            abort
13            break
14        end
15    end
16    disp(n," no. of iterations =")
17 endfunction
```

Scilab code AP 44 secant method

```
1 function [x]=secant(a,b,f)
2     N=100; // define max. no. iterations
           to be performed
3     PE=10^-4 // define tolerance for
           convergence
4     for n=1:1:N // initiating for loop
5         x=a-(a-b)*f(a)/(f(a)-f(b));
6         if abs(f(x))<=PE then break; //checking for
           the required condition
7         else a=b;
8             b=x;
9         end
10    end
11    disp(n," no. of iterations =") //
```

12 `endfunction`

Scilab code AP 45 regula falsi method

```
1 function [x]=regulafalsi(a,b,f)
2     N=100;
3     PE=10^-5;
4     for n=2:1:N
5         x=a-(a-b)*f(a)/(f(a)-f(b));
6         if abs(f(x))<=PE then break;
7         elseif (f(a)*f(x)<0) then b=x;
8             else a=x;
9     end
10 end
11 disp(n," no. of iterations =")
12 endfunction
```

Scilab code AP 46 four iterations of regula falsi method

```
1 function [x]=regulafalsi4(a,b,f)
2     N=100;
3     PE=10^-5;
4     for n=2:1:N
5         x=a-(a-b)*f(a)/(f(a)-f(b));
6         if abs(f(x))<=PE then break;
7         elseif (f(a)*f(x)<0) then b=x;
8             else a=x;
9     end
10 end
11 disp(n," no. of iterations =")
12 endfunction
```

Scilab code AP 47 four iterations of secant method

```
1 function [x]=secant4(a,b,f)
2     N=4; // define max. no. iterations
           to be performed
```

```

3     PE=10^-4           // define tolerance for
      convergence
4     for n=1:1:N       // initiating for loop
5         x=a-(a-b)*f(a)/(f(a)-f(b));
6         if abs(f(x))<=PE then break; //checking for
          the required condition
7         else a=b;
8             b=x;
9         end
10    end
11    disp(n," no. of iterations =") //
12 endfunction

```

Scilab code AP 48 five iterations by bisection method

```

1 function x=bisection5(a,b,f)
2     N=5; //
      define max. number of iterations
3     PE=10^-4; //
      define tolerance
4     if (f(a)*f(b) > 0) then error('no root possible
      f(a)*f(b) > 0') // checking if the decided
      range is containing a root
5         abort;
6     end;
7     if(abs(f(a)) < PE) then
8         error('solution at a') //
          seeing if there is an approximate root
          at a,
9         abort;
10    end;
11    if(abs(f(b)) < PE) then //
          seeing if there is an approximate root at b,
12    error('solution at b')
13    abort;
14    end;
15    x=(a+b)/2

```

```

16     for n=1:1:N                                     //
17         initialising 'for' loop ,
18         p=f(a)*f(x)
19         if p<0 then b=x ,x=(a+x)/2;
20             //checking for the required conditions( f
21             (x)*f(a)<0),
19         else
20             a=x
21             x=(x+b)/2;
22         end
23         if abs(f(x))<=PE then break
24             // instruction to come out of the loop
25             after the required condition is achived ,
26         end
27     end
28     disp(n," no. of iterations =")
29     // display the no. of iterations took to
30     achive required condition ,
27 endfunction

```

Scilab code AP 49 bisection method

```

1 function x=bisection(a,b,f)
2     N=100;                                           //
3     define max. number of iterations
4     PE=10^-4                                         //
5     define tolerance
6     if (f(a)*f(b) > 0) then
7         error('no root possible f(a)*f(b) > 0')
8         // checking if the decided range is
9         containing a root
10        abort;
11    end;
12    if(abs(f(a)) <PE) then
13        error('solution at a')                       //
14        seeing if there is an approximate root
15        at a,
16        abort;

```



```

11     end;
12     if(abs(f(b)) < PE) then //
        seeing if there is an approximate root at b,
13     error('solution at b')
14     abort;
15     end;
16     x=(a+b)/2
17     for n=1:1:N //
        initialising 'for' loop ,
18         p=f(a)*f(x)
19         if p<0 then b=x ,x=(a+x)/2;
            //checking for the required conditions( f
            (x)*f(a)<0),
20         else
21             a=x
22             x=(x+b)/2;
23         end
24         if abs(f(x))<=PE then break
            // instruction to come out of the loop
            after the required condition is achived ,
25         end
26     end
27     disp(n," no. of iterations =")
        // display the no. of iterations took to
        achive required condition ,
28 endfunction

```

Scilab code AP 50 solution by newton method given in equation 2.63

```

1
2 function x=newton63(x,f,fp,fpp)
3     R=100;
4     PE=10^-15;
5     maxval=10^4;
6
7     for n=1:1:R
8         x=x-(f(x)*fp(x))/(fp(x)^2-f(x)*fpp(x));
9         if abs(f(x))<=PE then break

```

```

10         end
11         if (abs(f(x))>maxval) then error('Solution
           diverges ');
12             abort
13             break
14         end
15     end
16     disp(n," no. of iterations =")
17 endfunction

```

Scilab code AP 51 solution by secant method given in equation 2.64

```

1 function [x]=secant64(a,b,f,fp)
2     N=100;           // define max. no. iterations
                       to be performed
3     PE=10^-15      // define tolerance for
                       convergence
4     for n=1:1:N    // initiating for loop
5         x=(b*f(a)*fp(b)-a*f(b)*fp(a))/(f(a)*fp(b)-f(
                       b)*fp(a));
6         if abs(f(x))<=PE then break; //checking for
                       the required condition
7         else a=b;
8             b=x;
9         end
10    end
11    disp(n," no. of iterations =") //
12 endfunction

```

Scilab code AP 52 solution by secant method given in equation 2.65

```

1 function [x]=secant65(a,b,f)
2     deff(' [y]=g(x) ', 'y=-f(x)^2/(f(x-f(x))-f(x)) ');
3     N=4;           // define max. no. iterations
                       to be performed
4     PE=10^-15      // define tolerance for
                       convergence
5     for n=1:1:N    // initiating for loop

```

```

6         x=a-(b-a)*g(a)/(g(b)-g(a));
7         if abs(f(x))<=PE then break; //checking for
           the required condition
8         else a=b;
9           b=x;
10        end
11       end
12       disp(n," no. of iterations =") //
13 endfunction

```

Scilab code AP 53 solution to the equation having multiple roots

```

1
2 function x=modified_newton(x,f,fp)
3     R=100;
4     PE=10^-8;
5     maxval=10^4;
6
7     for n=1:1:R
8         x=x-m*f(x)/fp(x);
9         if abs(f(x))<=PE then break
10        end
11        if (abs(f(x))>maxval) then error('Solution
           diverges');
12            abort
13            break
14        end
15    end
16    disp(n," no. of iterations =")
17 endfunction

```

Scilab code AP 54 solution by two iterations of general iteration

```

1
2 function x=generaliteration2(x,g,gp)
3     R=2;
4     PE=10^-8;
5     maxval=10^4;

```

```

6     A=[0 0];
7     k=gp(x);
8     if abs(k)>1 then error('function chosen does not
           converge')
9         abort;
10    end
11    for n=1:1:R
12        x=g(x);
13        disp(x);
14        if abs(g(x))<=PE then break
15        end
16        if (abs(g(x))>maxval) then error('Solution
           diverges');
17            abort
18            break
19        end
20    end
21    disp(n," no. of iterations =")
22 endfunction

```

Scilab code AP 55 solution by aitken method

```

1 // this program is exclusively coded to perform one
   iteration of aitken method,
2
3 function x0aa=aitken(x0,x1,x2,g)
4 x0a=x0-(x1-x0)^2/(x2-2*x1+x0);
5 x1a=g(x0a);
6 x2a=g(x1a);
7 x0aa=x0a-(x1a-x0a)^2/(x2a-2*x1a+x0a);
8
9 endfunction

```

Scilab code AP 56 solution by general iteration

```

1
2 function x=generaliteration(x,g,gp)
3     R=5;

```

```

4     PE=10^-8;
5     maxval=10^4;
6     k=gp(x);
7     if abs(k)>1 then error('function chosen does not
           converge')
8         abort;
9     end
10    for n=1:1:R
11        x=g(x);
12        disp(x);
13        if abs(g(x))<=PE then break
14        end
15        if (abs(g(x))>maxval) then error('Solution
           diverges');
16            abort
17            break
18        end
19    end
20    disp(n," no. of iterations =")
21 endfunction

```

Scilab code AP 57 solution by multipoint iteration given in equation 33

```

1 function x=multipoint_iteration33(x,f,fp,R)
2     R=3;
3     PE=10^-5;
4     maxval=10^4;
5     for n=1:1:R
6         x=x-f(x)/fp(x)-f(x-(f(x)/fp(x)))/fp(x);
7         if abs(f(x))<=PE then break;
8         end
9         if (abs(f(x))>maxval) then error('Solution
           diverges');
10            break
11        end
12    end
13    disp(n," no. of iterations =")
14 endfunction

```

Scilab code AP 58 solution by multipoint iteration given in equation 31

```
1 function x=multipoint_iteration31(x,f,fp,R)
2     R=3;
3     PE=10^-5;
4     maxval=10^4;
5     for n=1:1:R
6         x=x-f(x)/fp(x-(1/2)*(f(x)/fp(x)));
7         if abs(f(x))<=PE then break;
8         end
9         if (abs(f(x))>maxval) then error('Solution
            diverges ');
10            break
11        end
12    end
13    disp(n," no. of iterations =")
14 endfunction
```

Scilab code AP 59 solution by chebeshev method

```
1 function x=chebyshev(x,f,fp,fpp)
2     R=100;
3     PE=10^-5;
4     maxval=10^4;
5     if fp(x)==0 then disp("select another
            initial root x0");
6         break;
7     end
8     for n=1:1:R
9         x=x-f(x)/fp(x)-(1/2)*(f(x)/fp(x))^2 *(fpp(x)
            /fp(x));
10        if abs(f(x))<=PE then break;
11        end
12        if (abs(f(x))>maxval) then error('Solution
            diverges ');
13        abort;
```

```

14         break
15     end
16 end
17 disp(n," no. of iterations =")
18 endfunction

```

Scilab code AP 60 solution by five iterations of muller method

```

1 function x=muller5(x0,x1,x2,f)
2     R=5;
3     PE=10^-8;
4     maxval=10^4;
5     for n=1:1:R
6
7         La=(x2-x1)/(x1-x0);
8         Da=1+La;
9         ga=La^2*f(x0)-Da^2*f(x1)+(La+Da)*f(x2);
10        Ca=La*(La*f(x0)-Da*f(x1)+f(x2));
11
12        q=ga^2-4*Da*Ca*f(x2);
13        if q<0 then q=0;
14        end
15        p= sqrt(q);
16        if ga<0 then p=-p;
17        end
18        La=-2*Da*f(x2)/(ga+p);
19        x=x2+(x2-x1)*La;
20        if abs(f(x))<=PE then break
21        end
22        if (abs(f(x))>maxval) then error('Solution
23            diverges ');
24            abort;
25            break
26        else
27            x0=x1;
28            x1=x2;
29            x2=x;
30        end

```

```

30     end
31     disp(n," no. of iterations =")
32 endfunction

```

Scilab code AP 61 solution by three iterations of muller method

```

1  function x=muller3(x0,x1,x2,f)
2      R=3;
3      PE=10^-8;
4      maxval=10^4;
5      for n=1:1:R
6
7          La=(x2-x1)/(x1-x0);
8          Da=1+La;
9          ga=La^2*f(x0)-Da^2*f(x1)+(La+Da)*f(x2);
10         Ca=La*(La*f(x0)-Da*f(x1)+f(x2));
11
12         q=ga^2-4*Da*Ca*f(x2);
13         if q<0 then q=0;
14         end
15         p= sqrt(q);
16         if ga<0 then p=-p;
17         end
18         La=-2*Da*f(x2)/(ga+p);
19         x=x2+(x2-x1)*La;
20         if abs(f(x))<=PE then break
21         end
22         if (abs(f(x))>maxval) then error('Solution
           diverges ');
23             abort;
24             break
25         else
26             x0=x1;
27             x1=x2;
28             x2=x;
29         end
30     end
31     disp(n," no. of iterations =")

```


32 `endfunction`
