

Scilab Textbook Companion for  
An Introduction To Numerical Analysis  
by K. E. Atkinson<sup>1</sup>

Created by  
Warsha B  
B.Tech (pursuing)  
Electronics Engineering  
VNIT  
College Teacher  
G.NAGA RAJU  
Cross-Checked by  
Prashant Dave, IITB

July 31, 2019

<sup>1</sup>Funded 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:** An Introduction To Numerical Analysis

**Author:** K. E. Atkinson

**Publisher:** John Wiley And Sons

**Edition:** 2

**Year:** 2001

**ISBN:** 8126518502

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
1 Error Its sources Propagation and Analysis	5
2 Rootfinding for Nonlinear equations	12
3 Interpolation Theory	22
4 Approximation of functions	28
5 Numerical Integration	38
6 Numerical methods for ordinary differential equations	52
7 Linear Algebra	60
8 Numerical solution of systems of linear equations	64
9 The Matrix Eigenvalue Problem	74

# List of Scilab Codes

Exa 1.1	Taylor series . . . . .	5
Exa 1.3	Vector norms . . . . .	6
Exa 1.4	Conversion to decimal . . . . .	6
Exa 1.5	Error and relative error . . . . .	7
Exa 1.6	Errors . . . . .	7
Exa 1.7	Taylor series . . . . .	8
Exa 1.8	Graph of polynomial . . . . .	8
Exa 1.9	Error and Relative error . . . . .	9
Exa 1.10	Loss of significance errors . . . . .	10
Exa 1.11	Loss of significance errors . . . . .	11
Exa 2.1	Bisection method . . . . .	12
Exa 2.2	Newton method . . . . .	13
Exa 2.3	Secant method . . . . .	14
Exa 2.4	Muller method . . . . .	15
Exa 2.6	Muller method . . . . .	16
Exa 2.7	One point iteration method . . . . .	17
Exa 2.8	One point Iteration method . . . . .	18
Exa 2.10	Aitken . . . . .	20
Exa 2.11	Multiple roots . . . . .	20
Exa 3.1	Lagrange formula . . . . .	22
Exa 3.2	Lagrange Formula . . . . .	22
Exa 3.3	Lagrange formula . . . . .	23
Exa 3.4	Divided differences . . . . .	23
Exa 3.6	Bessel Function . . . . .	24
Exa 3.7	Divided differences . . . . .	25
Exa 3.8	Newton forward difference . . . . .	25
Exa 4.1	Error of approximating exponent of x . . . . .	28
Exa 4.2	Minimax Approximation problem . . . . .	28

Exa 4.3	Least squares approximation problem . . . . .	30
Exa 4.4	Weight functions . . . . .	31
Exa 4.5	Formulae . . . . .	32
Exa 4.6	Formulae for laguerre and legendre polynomi- als . . . . .	32
Exa 4.7	Average error in approximation . . . . .	33
Exa 4.8	Chebyshev expansion coefficients . . . . .	34
Exa 4.9	Max errors in cubic chebyshev least squares approx . . . . .	35
Exa 4.10	Near minimax approximation . . . . .	35
Exa 4.11	Forced oscillation of error . . . . .	36
Exa 5.1	Integration . . . . .	38
Exa 5.2	Trapezoidal rule for integration . . . . .	38
Exa 5.3	Corrected trapezoidal rule . . . . .	40
Exa 5.4	Simpson s rule for integration . . . . .	41
Exa 5.5	Trapezoidal and simpson integration . . . . .	42
Exa 5.6	Newton Cotes formulae . . . . .	44
Exa 5.7	Gaussian Quadrature . . . . .	44
Exa 5.8	Gaussian Legendre Quadrature . . . . .	46
Exa 5.9	Integration . . . . .	48
Exa 5.10	Simpson Integration error . . . . .	48
Exa 5.11	Romberg Integration . . . . .	49
Exa 5.12	Adaptive simpson . . . . .	50
Exa 5.13	Integration . . . . .	50
Exa 5.14	Integration . . . . .	51
Exa 6.1	1st order linear differential equation . . . . .	52
Exa 6.4	Stability of solution . . . . .	52
Exa 6.5	Euler method . . . . .	53
Exa 6.6	Euler . . . . .	54
Exa 6.7	Asymptotic error analysis . . . . .	55
Exa 6.9	Midpoint and trapezoidal method . . . . .	55
Exa 6.10	Euler . . . . .	56
Exa 6.11	Trapezoidal method . . . . .	57
Exa 6.16	Adams Moulton method . . . . .	57
Exa 6.21	Euler method . . . . .	58
Exa 6.24	Trapezoidal method . . . . .	58
Exa 6.31	Boundary value problem . . . . .	59
Exa 7.1	Orthonomal basis . . . . .	60

Exa 7.2	Canonical forms . . . . .	60
Exa 7.3	Orthonormal eigen vectors . . . . .	61
Exa 7.4	Vector and matrix norms . . . . .	61
Exa 7.5	Frobenious norm . . . . .	62
Exa 7.6	Norm . . . . .	62
Exa 7.7	Inverse exists . . . . .	63
Exa 8.2	LU decomposition . . . . .	64
Exa 8.4	LU decomposition . . . . .	65
Exa 8.5	Choleski Decomposition . . . . .	65
Exa 8.6	LU decomposition . . . . .	66
Exa 8.7	Error analysis . . . . .	67
Exa 8.8	Residual correction method . . . . .	68
Exa 8.9	Residual correction method . . . . .	70
Exa 8.10	Gauss Jacobi method . . . . .	70
Exa 8.11	Gauss seidel method . . . . .	71
Exa 8.13	Conjugate gradient method . . . . .	72
Exa 9.1	Eigenvalues . . . . .	74
Exa 9.2	Eigen values and matrix norm . . . . .	74
Exa 9.3	Bounds for perturbed eigen values . . . . .	75
Exa 9.4	Eigenvalues of nonsymmetric matrix . . . . .	77
Exa 9.5	Stability of eigenvalues for nonsymmetric matrices . . . . .	78
Exa 9.7	Rate of convergence . . . . .	79
Exa 9.8	Rate of convergence after extrapolation . . . . .	80
Exa 9.9	Householder matrix . . . . .	80
Exa 9.11	QR factorisation . . . . .	81
Exa 9.12	Tridiagonal Matrix . . . . .	82
Exa 9.13	Planner Rotation Orthogonal Matrix . . . . .	82
Exa 9.14	Eigen values of a symmetric tridiagonal Matrix . . . . .	82
Exa 9.15	Sturm Sequence property . . . . .	83
Exa 9.16	QR Method . . . . .	83
Exa 9.18	Calculation of Eigen vectors and Inverse iteration . . . . .	84
Exa 9.19	Inverse Iteration . . . . .	85

# Chapter 1

## Error Its sources Propagation and Analysis

Scilab code Exa 1.1 Taylor series

```
1          //      PG (6)
2
3 //      Taylor series for  $e^{-x^2}$  upto first four
      terms
4
5 deff ('[y]=f(x)', 'y=exp(-x^2)')
6 funcprot(0)
7 deff ('[y]=fp(x)', 'y=-2*x*exp(-x^2)')
8 funcprot(0)
9 deff ('[y]=fpp(x)', 'y=(1-2*x^2)*(-2*exp(-2*x^2))')
10 funcprot(0)
11 deff ('[y]=g(x)', 'y=4*x*exp(-x^2)*(3-2*x^2)')
12 funcprot(0)
13 deff ('[y]=gp(x)', 'y=(32*x^4*exp(-x^2))+(-72*x^2*exp
      (-x^2))+12*exp(-x^2)')
14 funcprot(0)
15 x0=0;
16 x=poly(0, "x");
17 T = f(x0) + (x-x0)*fp(x0)/factorial(1) + (x-x0)^2 *
```



```

    fpp(x0)/factorial(2) + (x-x0)^3 * g(x0)/factorial
    (3) + (x-x0)^4 * gp(x0)/factorial(4)
18
19
20
21 //      Similarly Taylor series for inv(tan(x))

```

---

### Scilab code Exa 1.3 Vector norms

```

1          //      PG (11)
2
3  A = [1 -1;3 2]
4  x = [1;2]
5  y = A*x
6  norm(A, 'inf')
7  norm(x, 'inf')
8  norm(y, 'inf')
9
10 x = [1;1]
11 y = A*x
12 norm(y, 'inf')
13 norm(A, 'inf')*norm(x, 'inf')
14
15 //      norm(y, 'inf') = norm(A, 'inf') * norm(x, 'inf')

```

---

### Scilab code Exa 1.4 Conversion to decimal

```

1          //      PG (12)
2
3  //      11011.01 is a binary number. Its decimal
    equivalent is:
4

```

```

5 1*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 + 0*2^(-1) +
    1*2^(-2)
6
7 //      56C.F is a hexadecimal number. Its decimal
    equivalent is :
8
9 5*16^2 + 6*16^1 + 12*16^0 + 15*16^(-1)

```

---

### Scilab code Exa 1.5 Error and relative error

```

1          //      PG (17)
2
3 xT = exp(1)
4 xA = 19/7
5
6 //      Error(xA)
7
8 xT - xA
9
10 //      Relative error , Rel(xA)
11
12 (xT-xA)/xT

```

---

### Scilab code Exa 1.6 Errors

```

1          //      PG (18)
2
3 xT = 1/3
4 xA = 0.333
5 abs(xT-xA)      //      Error
6
7 //-----
8

```

```

9 xT = 23.496
10 xA = 23.494
11 abs(xT-xA) // Error
12
13 //-----
14
15 xT = 0.02138
16 xA = 0.02144
17 abs(xT-xA) // Error

```

---

#### Scilab code Exa 1.7 Taylor series

```

1 // PG (20)
2
3 // Taylor series for the first two terms
4
5 deff(' [y]=f(x)', 'y=sqrt(1+x)')
6 funcprot(0)
7 deff(' [y]=fp(x)', 'y=0.5*(1+x)^(-1/2)')
8 funcprot(0)
9 x0=0;
10 x=poly(0, 'x');
11 T = f(x0) + (x-x0)*fp(x0)/factorial(1)

```

---

#### Scilab code Exa 1.8 Graph of polynomial

```

1 // PG (21)
2
3 deff(' [y]=f(x)', 'y = x^3-3*x^2+3*x-1')
4 xset('window', 0);
5 x=-0:.01:2;

```

//

defining the range of x.

```

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 = x^3-3*x^2+3*x-1')

```

---

#### Scilab code Exa 1.9 Error and Relative error

```

1          // PG (24)
2
3 xT = %pi
4 xA = 3.1416
5 yT = 22/7
6 yA = 3.1429
7 xT - xA          // Error
8 (xT - xA)/xT    // Relative Error
9 yT - yA          // Error
10 (yT - yA)/yT   // Relative Error
11
12 (xT - yT) - (xA - yA)
13 ((xT - yT) - (xA - yA))/(xT - yT)
14
15 // Although the error in xA - yA is quite small,
16 // the relative error in xA - yA is much larger
    than that in xA or yA alone.

```

---

Scilab code Exa 1.10 Loss of significance errors

```
1          // PG (25)
2
3 //      Consider solving  $ax^2 + b*x + c =$ 
4
5
6 //      Consider a polynomial  $y = x^2 - 26*x + 1 = 0$ 
7
8 x = poly(0,"x");
9 y = x^2 - 26*x + 1
10 p = roots(y)
11 ra1 = p(2,1)
12 ra2 = p(1,1)
13
14 //      Using the standard quadratic formula for
      finding roots ,
15
16 rt1 = (-(-26)+sqrt((-26)^2 - 4*1*1))/(2*1)
17 rt2 = (-(-26)-sqrt((-26)^2 - 4*1*1))/(2*1)
18
19 //      Relative error
20
21 rel1 = (ra1-rt1)/ra1
22 rel2 = (ra2-rt2)/ra2
23
24 //      The significant errors have been lost in the
      subtraction  $ra2 = xa - ya$ .
25 //      The accuracy in ra2 is much less.
26 //      To calculate ra2 accurately , we use:
27
28 rt2 = ((13-sqrt(168))*(13+sqrt(168)))/(1*(13+sqrt
      (168)))
29 //      Now, rt2 is nearly equal to ra2. So, by exact
      calculations , we will now get a much better rel2.
```

---

Scilab code Exa 1.11 Loss of significance errors

```
1          // PG (26)
2
3  x = poly(0,"x");
4  x = 0;
5  deff(' [y]=f(t)', 'y=exp(x*t)')
6  integrate('exp(x*t)', 't', 0, 1)
7
8  // So, for x = 0, f(0) = 1
9  // f(x) is continuous at x = 0.
10
11 // To see that there is a loss of significance
12 // problem when x is small,
13 // we evaluate f(x) at 1.4*10(-9)
14 x = 1.4*10(-9)
15 integrate('exp(x*t)', 't', 0, 1)
16 // When we use a ten-digit hand calculator, the
17 // result is 1.000000001
18 // To avoid the loss of significance error, we
19 // may use a quadratic Taylor approximation to exp(x
20 // ) and then simplify f(x).
```

---

# Chapter 2

## Rootfinding for Nonlinear equations

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

```
bisection1.sce
```

**Scilab code Exa 2.1** Bisection method

```
1 // EXAMPLE (PG 57)
2 // To find largest root, alpha, of  $x^6 - x - 1 = 0$ 
3 // using bisection method
4 // The graph of this function can also be
   observed here.
5
6 def ('[y]=f(x)', 'y=x^6-x-1')
7 // It is straightforward to
   show that  $1 < \alpha < 2$ , and
8 //we will use this as our
   initial interval [a,b]
9
10
11 xset('window',0);
```

```

12 x=-5:.01:5;
//
    defining the range of x.
13 y=feval(x,f);
14
15 a=gca();
16
17 a.y_location = "origin";
18
19 a.x_location = "origin";
20 plot(x,y)

    // instruction to plot the graph
21
22 title(' y = x^6-x-1')
23
24 // execution of the user defined function so as to
    use it in program to find the approximate
    solution.
25
26 // we call a user-defined function 'bisection' so as
    to find the approximate
27 // root of the equation with a defined permissible
    error.
28
29 bisection(1,2,f)

```

---

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

newton.sce

### Scilab code Exa 2.2 Newton method

```

1 // EXAMPLE (PG 60)
2 // To find largest root, alpha, of  $f(x) = x^6$ 
    -  $x - 1 = 0$ 

```



```

3      //      using newton's method
4
5
6  def f(' [y]=f(x) ', 'y=x^6-x-1')
7  def fp(' [y]=fp(x) ', 'y=6*x^5-1') //
      Derivative of f(x)
8  x=(1+2)/2 //      Initial
      appoximation
9
10 //we call a user-defined function 'newton' so as to
      find the approximate
11 // root of the equation with a defined permissible
      error.
12
13
14 newton(x,f,fp)

```

---

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

secant.sce

### Scilab code Exa 2.3 Secant method

```

1      //      EXAMPLE ( PG 66)
2      //      To find largest root, alpha, of  $f(x) = x^6$ 
      -  $x - 1 = 0$ 
3      //      using secant method
4
5  def f(' [y]=f(x) ', 'y=x^6-x-1')
6  a=1
7  b=2 //      Initial approximations
8
9
10 // we call a user-defined function 'secant' so as to
      find the approximate

```

```

11 // root of the equation with a defined permissible
    error.
12
13 secant(a,b,f)

```

---

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

muller.sce

#### Scilab code Exa 2.4 Muller method

```

1 // EXAMPLE1 (PG 76)
2 // f(x) = x^20 - 1
3 // solving using Muller's method
4
5
6 xset('window',1);
7 x=-2:.01:4; //
    defining the range of x.
8 def(' [y]=f(x)', 'y=x^20-1'); //
    defining the function.
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 = x^20-1')
18
19 // from the above plot we can infere that the
    function has roots between
20 // the intervals (0,1),(2,3).
21

```

```

22         //sollution by muller method to 3 iterations
23         .
24 muller(0, .5, 1, f)

```

---

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

muller.sce

### Scilab code Exa 2.6 Muller method

```

1     //     EXAMPLE3 (PG 76)
2     //     f(x) = x^6- 12 * x^5 + 63 * x^4 - 216* x^3
           + 567 * x^2 - 972 * x + 729
3     //     or f(x) = (x^2+9)*(x-3)^4
4     //     solving using Muller 's method
5
6     deff ('[y]=f(x)', 'y=(x^2+9)*(x-3)^4')
7
8     xset('window', 2);
9     x=-10:.1:10;
                                     //
           defining the range of x.
10    y=feval(x, f);
11
12    a=gca();
13
14    a.y_location = "origin";
15
16    a.x_location = "origin";
17    plot(x, y)
                                     // instruction to plot the graph
18
19    title(' y = (x^2+9)*(x-3)^4')
20

```

```
21
22 muller(0,.5,1,f)
```

---

### Scilab code Exa 2.7 One point iteration method

```
1 // EXAMPLE (PG 77)
2 // x^2-a = 0
3
4 // The graph for x^2-3 can also be observed
   here.
5
6 deff(' [y]=f(x) ', 'y=x*x-3')
7 funcprot(0)
8 xset('window',3);
9 x=-2:.01:10;
                                     //
   defining the range of x.
10 y=feval(x,f);
11
12 a=gca();
13
14 a.y_location = "origin";
15
16 a.x_location = "origin";
17 plot(x,y)
                                     // instruction to plot the graph
18
19 title(' y = x^2-3')
20 // CASE 1
21
22 //We have f(x) = x^2-a.
23 //So, we assume g(x) = x^2+x-a and the value of a =
   3
24
```

```

25 deff ('[y]=g(x)', 'y=x^2+x-3')
26 funcprot(0)
27 x=2
28 for n=0:1:3
29     g(x);
30     x=g(x)
31 end
32
33 //          CASE 2
34
35 //We have  $f(x) = x^2 - a$ .
36 //So, we assume  $g(x) = a/x$  and the value of  $a = 3$ 
37
38 deff ('[y]=g(x)', 'y=3/x')
39 funcprot(0)
40 x=2
41 for n=0:1:3
42     g(x);
43     x=g(x)
44 end
45
46 //          CASE 3
47
48 //We have  $f(x) = x^2 - a$ .
49 //So, we assume  $g(x) = 0.5*(x+(a/x))$  and the value
    of  $a = 3$ 
50
51 deff ('[y]=g(x)', 'y=0.5*(x+(3/x))')
52 funcprot(0)
53 x=2
54 for n=0:1:3
55     g(x);
56     x=g(x)
57 end

```

---

## Scilab code Exa 2.8 One point Iteration method

```
1 // EXAMPLE (PG 81)
2
3 //Assume alpha is a solution of  $x = g(x)$ 
4
5 alpha=sqrt(3);
6
7 // case 1
8
9
10 deff(' [y]=g(x) ', 'y=x^2+x-3')
11 deff(' [z]=gp(x) ', 'z=2*x+1') // Derivative
    of g(x)
12 gp(alpha)
13
14 // case 2
15
16 deff(' [y]=g(x) ', 'y=3/x')
17 funcprot(0)
18 deff(' [z]=gp(x) ', 'z=3/x') // Derivative of
    g(x)
19 gp(alpha)
20
21 // case 3
22
23 deff(' [y]=g(x) ', 'y=0.5*(x+(3/x))')
24 funcprot(0)
25 deff(' [z]=gp(x) ', 'z=0.5*(1-(3/(x^2)))') //
    Derivative of g(x)
26 gp(alpha)
```

---

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

aitken1.sce

### Scilab code Exa 2.10 Aitken

```
1 // EXAMPLE (PG 85)
2
3 // x(n+1) = 6.28 + sin(x(n))
4 // True root is alpha = 6.01550307297
5
6 def f ('[y]=f(x)', 'f(x)=6.28+sin(x(n))')
7 // k=6.01550307297
8
9 //x=6.01550307297
10
11 def g ('[y]=g(x)', 'y=cos(x)')
12
13
14 // we call a user-defined function 'aitken' so as to
15 // find the approximate
16 // root of the equation with a defined permissible
17 // error.
18 aitken(0.2,0.5,1,g)
```

---

### Scilab code Exa 2.11 Multiple roots

```
1 // EXAMPLE (PG 87)
2
3 // f(x) = (x-1.1)^3 * (x-2.1)
4
5 c = [2.7951 -8.954 10.56 -5.4 1]
6 p4=poly(c, 'x', 'coeff')
7 roots(p4)
8 def f ('[y]=f(x)', 'y=(x-1.1)^3*(x-2.1)')
9 xset('window',0);
10 x=0:.01:3;
```

```

//
    defining the range of x.
11 y=feval(x,f);
12
13 a=gca();
14
15 a.y_location = "origin";
16
17 a.x_location = "origin";
18 plot(x,y)

    // instruction to plot the graph
19
20 title(' y = (x-1.1)^3*(x-2.1)')
```

---



# Chapter 3

## Interpolation Theory

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

```
lagrange.sce
```

**Scilab code Exa 3.1** Lagrange formula

```
1          // PG (134)
2
3 X = [0, -1, 1]
4 Y = [1, 2, 3]
5 lagrange(X,Y)
```

---

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

```
lagrange.sce
```

**Scilab code Exa 3.2** Lagrange Formula

```
1          // PG (136)
2
3
```

```

4
5 X=[0]
6 Y=[1]
7 def f(' [y]=f(x) ', 'y=log10(x) ')
8 p=lagrange(X,Y)

```

---

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

lagrange.sce

### Scilab code Exa 3.3 Lagrange formula

```

1          // PG (137)
2
3 X=[0, -1]
4 Y=[1, 2]
5 def f(' [y]=f(x) ', 'y=log(x) ')
6 def fp(' [y]=fp(x) ', 'y=1/x ')
7 def fpp(' [y]=fpp(x) ', 'y=-1/(x)^2 ')
8 p = lagrange(X,Y)
9 // E = f(x)-p
10 e = 0.00005 // for a four-place logarithmic
    table

```

---

### Scilab code Exa 3.4 Divided differences

```

1          // PG (140)
2
3 X = [2.0, 2.1, 2.2, 2.3, 2.4]
4 X1 = X(1,1)
5 X2 = X(1,2)
6 X3 = X(1,3)
7 X4 = X(1,4)

```

```

8 X5 = X(1,5)
9 def f(' [y]=f(x) ', 'y=sqrt(x) ')
10 Y = [f(X1) f(X2) f(X3) f(X4) f(X5)]
11 Y1 = Y(1,1)
12 Y2 = Y(1,2)
13 Y3 = Y(1,3)
14 Y4 = Y(1,4)
15 Y5 = Y(1,5)
16
17 //      Difference
18
19 //      f [X1,X2]
20 (f(X2) - f(X1))*10
21 //      f [X2,X3]
22 (f(X3) - f(X2))*10
23 //      f [X3,X4]
24 (f(X4) - f(X3))*10
25 //      f [X4,X5]
26 (f(X5) - f(X4))*10
27
28 //      D^2 * f [Xi]
29
30 ((f(X3)-f(X2)) - (f(X2)-f(X1))) * 50
31 ((f(X4)-f(X3)) - (f(X3)-f(X2))) * 50
32 ((f(X5)-f(X4)) - (f(X4)-f(X3))) * 50

```

---

### Scilab code Exa 3.6 Bessel Function

```

1          //      PG (142)
2
3 //      Values of Bessel Function Jo(x)
4
5 //      x                Jo(x)
6
7 //      2.0            0.2238907791

```

```

8 //          2.1          0.1666069803
9 //          2.2          0.1103622669
10 //         2.3          0.0555397844
11 //         2.4          0.0025076832
12 //         2.5         -0.0483837764
13 //         2.6         -0.0968049544
14 //         2.7         -0.1424493700
15 //         2.8         -0.1850360334
16 //         2.9         -0.2243115458
17
18 //          Calculate the value of x for which Jo(x) = 0.1

```

---

#### Scilab code Exa 3.7 Divided differences

```

1 //          PG (144)
2
3 deff ( '[y]=f(x)', 'y=sqrt(x)' )
4 funcprot(0)
5 deff ( '[y]=fp(x)', 'y=0.5/sqrt(x)' )
6 funcprot(0)
7 deff ( '[y]=fpp(x)', 'y=-0.25*x^(-3/2)' )
8 funcprot(0)
9 deff ( '[y]=fppp(x)', 'y=3*x^(-2.5)/8' )
10 deff ( '[y]=fpppp(x)', 'y=-15*x^(-7/2)/16' )
11
12 //          f[2.0,2.1,.....2.4] = -0.002084
13
14 fpppp(2.3103)/factorial(4)

```

---

#### Scilab code Exa 3.8 Newton forward difference

```

1 //          PG (150)
2

```

```

3 X = [2.0,2.1,2.2,2.3,2.4]
4 X1 = X(1,1)
5 X2 = X(1,2)
6 X3 = X(1,3)
7 X4 = X(1,4)
8 X5 = X(1,5)
9 def f(' [y]=f(x) ', 'y=sqrt(x) ')
10 Y = [f(X1) f(X2) f(X3) f(X4) f(X5)]
11 Y1 = Y(1,1)
12 Y2 = Y(1,2)
13 Y3 = Y(1,3)
14 Y4 = Y(1,4)
15 Y5 = Y(1,5)
16
17 //      Difference
18
19 //      f [X1,X2]
20 (f(X2) - f(X1))
21 //      f [X2,X3]
22 (f(X3) - f(X2))
23 //      f [X3,X4]
24 (f(X4) - f(X3))
25 //      f [X4,X5]
26 (f(X5) - f(X4))
27
28 //      D^2 * f [Xi]
29
30 ((f(X3)-f(X2)) - (f(X2)-f(X1)))
31 ((f(X4)-f(X3)) - (f(X3)-f(X2)))
32 ((f(X5)-f(X4)) - (f(X4)-f(X3)))
33
34 //      D^3 * f [Xi]
35
36 ((f(X4)-f(X3)) - (f(X3)-f(X2))) - ((f(X3)-f(X2)) - (
    f(X2)-f(X1)))
37 ((f(X5)-f(X4)) - (f(X4)-f(X3))) - ((f(X4)-f(X3)) - (
    f(X3)-f(X2)))
38

```

```

39 //      D^4 * f[Xi]
40
41 (((f(X5)-f(X4)) - (f(X4)-f(X3))) - ((f(X4)-f(X3)) -
      (f(X3)-f(X2)))) - (((f(X4)-f(X3)) - (f(X3)-f(X2))
      ) - ((f(X3)-f(X2)) - (f(X2)-f(X1))))
42
43 mu = 1.5;
44 x = 2.15;
45
46 p1 = f(X1) + mu * (f(X2) - f(X1))
47 p2 = p1 + mu*(mu-1)*((f(X3)-f(X2)) - (f(X2)-f(X1)))
      /2
48
49 //      Similarly , p3 = 1.466288
50 //      p4 = 1.466288

```

---

# Chapter 4

## Approximation of functions

Scilab code Exa 4.1 Error of approximating exponent of x

```
1          // PG (199)
2
3 x = poly(0,"x");
4 p3 = 1 + x + (1/2)*x^(2) + (1/6)*x^3
5 deff(' [y]=f(x) ', 'y=exp(x) ')
6 funcprot(0)
7 x = -1:0.01:1;
8 f(x) - p3
```

---

Scilab code Exa 4.2 Minimax Approximation problem

```
1          // PG (200)
2
3 deff(' [y]=f(x) ', 'y=exp(x) ')
4
5 xset('window',0);
6 x=-1:.01:1;          // defining the range of
   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
16
17
18 // possible approximation
19 // y = q1(x)
20
21 // Let e(x) = exp(x) - [a0+a1*x]
22 // q1(x) & exp(x) must be equal at two points in
    [-1,1], say at x1 & x2
23 // sigma1 = max(abs(e(x)))
24 // e(x1) = e(x2) = 0.
25 // By another argument based on shifting the
    graph of y = q1(x),
26 // we conclude that the maximum error sigma1 is
    attained at exactly 3 points.
27 // e(-1) = sigma1
28 // e(1) = sigma1
29 // e(x3) = -sigma1
30 // x1 < x3 < x2
31 // Since e(x) has a relative minimum at x3, we
    have e'(x) = 0
32 // Combining these 4 equations, we have..
33 // exp(-1) - [a0-a1] = sigma1 -----(
    i)
34 // exp(1) - [a0+a1] = p1 -----(
    ii)
35 // exp(x3) - [a0+a1*x3] = -sigma1 -----(
    iii)
36 // exp(x3) - a1 = 0 -----(

```



```

        iv)
37
38 //      These have the solution
39
40 a1 = (exp(1) - exp(-1))/2
41 x3 = log(a1)
42 sigma1 = 0.5*exp(-1) + x3*(exp(1) - exp(-1))/4
43 a0 = sigma1 + (1-x3)*a1
44
45 x = poly(0,"x");
46 //      Thus,
47 q1 = a0 + a1*x
48
49 deff(' [y1]=f(x) ', 'y1=1.2643+1.1752*x ')
50
51 xset('window',0);
52 x=-1:.01:1;           // defining the range of
        x.
53 y=feval(x,f);
54
55 a=gca();
56
57 a.y_location = "origin";
58
59 a.x_location = "origin";
60 plot(x,y)           // instruction to plot the
        graph

```

---

### Scilab code Exa 4.3 Least squares approximation problem

```

1           //      PG (205)
2
3 deff(' [y]=f(x) ', 'y=exp(x) ')
4
5 x=-1:.01:1;           // defining the range of

```

```

        x
6
7 //      Let r1(x) = b0 + b1(x)
8 //      Minimize
9 //      ||f-r1||^2 = integrate('(exp(x)-b0-b1*x)
^2','x',-1,1) = F(b0,b1)
10 //      F = integrate('exp(2*x) + b0^2 + (b1^2)*(x^2)
- 2*b0*x*exp(x) + 2*b0*b1*x','x',b0,b1)
11 //      To find a minimum, we set
12
13 //      df/db0 = 0
14 //      df/db1 = 0-----necessary conditions
      at a minimal point
15 //      On solving, we get the values of b0 & b1
16
17 b0 = 0.5*integrate('exp(x)','x',-1,1)
18 b1 = 1.5*integrate('x*exp(x)','x',-1,1)
19 r1 = b0+b1*x;
20 norm(exp(x)-r1,'inf') //      least squares
      approximation
21
22 r3 = 0.996294 + 0.997955*x + 0.536722*x^2 +
      0.176139*x^3
23 norm(exp(x)-r3,'inf') //      cubic least squares
      approximation

```

---

#### Scilab code Exa 4.4 Weight functions

```

1 //      PG (206)
2
3 //      The following are the weight functions of most
      interest in the
4 //      developments of this text:
5
6 //      w(x)=1          a < = x < = b

```

```

7
8 //      w(x)=1/sqrt(1-x^2)      -1 < = x < = 1
9
10 //      w(x)=exp(-x)           0 < = x < infinity
11
12 //      w(x)=exp(-x^2)         -infinity < x <
    infinity

```

---

**Scilab code Exa 4.5** Formulae

```

1 //      PG (215)
2
3 //      for laguerre polynomials ,
4 //      L(n+1)=1*[2*n+1-x]*L(n)/(n+1) - n*L(n-1)/(n+1)
5
6 //      for Legendre polynomials ,
7
8 //      P(n+1)= (2*n +1)*x*P(n)/(n+1) - n*P(n-1)/(n+1)

```

---

**Scilab code Exa 4.6** Formulae for laguerre and legendre polynomials

```

1 //      PG (215)
2
3 //      for laguerre polynomials ,
4 //      L(n+1)=1*[2*n+1-x]*L(n)/(n+1) - n*L(n-1)/(n+1)
5
6 //      for Legendre polynomials ,
7
8 //      P(n+1)= (2*n +1)*x*P(n)/(n+1) - n*P(n-1)/(n+1)

```

---

Scilab code Exa 4.7 Average error in approximation

```

1           // PG (219)
2
3  deff ('[y]=f(x)', 'y=exp(x)')
4
5  x=-1:.01:1;           // defining the range of
   x
6
7  // Let r1(x) = b0 + b1(x)
8  // Minimize
9  // ||f-r1||^2 = integrate('(exp(x)-b0-b1*x)
   ^2','x',-1,1) = F(b0,b1)
10 // F = integrate('exp(2*x) + b0^2 + (b1^2)*(x^2)
   - 2*b0*x*exp(x) + 2*b0*b1*x','x',b0,b1)
11 // To find a minimum, we set
12
13 // df/db0 = 0
14 // df/db1 = 0-----necessary conditions
   at a minimal point
15 // On solving, we get the values of b0 & b1
16
17 b0 = 0.5*integrate('exp(x)','x',-1,1);
18 b1 = 1.5*integrate('x*exp(x)','x',-1,1);
19 r1 = b0+b1*x;
20 norm(exp(x)-r1,'inf'); // least squares
   approximation
21
22 r3 = 0.996294 + 0.997955*x + 0.536722*x^2 +
   0.176139*x^3;
23 norm(exp(x)-r3,'inf'); // cubic least squares
   approximation
24
25 // average error E
26
27 E = norm(exp(x)-r3,2)/sqrt(2)

```

---

### Scilab code Exa 4.8 Chebyshev expansion coefficients

```
1 // PG (220)
2
3 deff ('[y]=f(x)', 'y=exp(x)')
4
5 // Chebyshev expansion coefficients for exp(x)
6 // j = 0
7 C0=2*(integrate('exp(cos(x))', 'x', 0, 3.14))/(3.14)
8
9 // j = 1
10 C1=2*(integrate('exp(cos(x))*cos(x)', 'x', 0, 3.14))
    /(3.14)
11
12 // j = 2
13 C2=2*(integrate('exp(cos(x))*cos(2*x)', 'x', 0, 3.14))
    /(3.14)
14
15 // j = 3
16 C3=2*(integrate('exp(cos(x))*cos(3*x)', 'x', 0, 3.14))
    /(3.14)
17
18 // j = 4
19 C4=2*(integrate('exp(cos(x))*cos(4*x)', 'x', 0, 3.14))
    /(3.14)
20
21 // j = 5
22 C5=2*(integrate('exp(cos(x))*cos(5*x)', 'x', 0, 3.14))
    /(3.14)
23
24 // we obtain
25 x=0:.01:%pi; // defining the range of
    x
26
```

```

27 c1=1.266+1.130*x;
28 c3=0.994571+0.997308*x+0.542991*x.^2+0.177347*x.^3;
29 norm(exp(x)-c1,'inf')
30 norm(exp(x)-c3,'inf')

```

---

Scilab code Exa 4.9 Max errors in cubic chebyshev least squares approx

```

1          // PG (223)
2
3 deff(' [y]=f(x)', 'y=exp(x)')
4
5 x=[-1.0 -0.6919 0.0310 0.7229 1.0];
   // defining x
6
7 r3 = 0.996294 + 0.997955*x + 0.536722*x^2 +
   0.176139*x^3;
8 norm(exp(x)-r3,'inf'); // cubic least squares
   approximation
9 deff(' [y]=g(x)', 'y=0.994571+0.997308*x+0.542991*x
   ^2+0.177347*x^3')
10 // c3=g(x);
11 x1=x(1,1);
12 (exp(x1)-g(x1))
13 x2=x(1,2);
14 (exp(x2)-g(x2))
15 x3=x(1,3);
16 (exp(x3)-g(x3))
17 x4=x(1,4);
18 (exp(x4)-g(x4))
19 x5=x(1,5);
20 (exp(x5)-g(x5))

```

---

Scilab code Exa 4.10 Near minimax approximation

```

1 // PG (227)
2
3 deff(' [y]=f(x) ', 'y=exp(x) ')
4 x = -1:0.01:1;
5 c3=0.994571+0.997308*x+0.542991*x.^2+0.177347*x.^3;
6 norm(exp(x)-c3, 'inf')
7
8 // as obtained in the example 6, c4 = 0.00547, T4
9 // (x) = (-1)
10 // c4*T4(x) = 0.00547 * (-1)
11 // norm(exp(x)-q3, 'inf') = 0.00553

```

---

#### Scilab code Exa 4.11 Forced oscillation of error

```

1 // PG (234)
2
3 deff(' [y]=f(x) ', 'y=exp(x) ')
4 x = -1:0.01:1;
5 // For
6 n = 1;
7 x = [-1 0 1];
8 E1 = 0.272;
9 F1 = 1.2715 + 1.1752*x;
10
11 // Relative errors
12
13 x = -1.0;
14 exp(x) - F1;
15 r1 = ans(1,1)
16 x = 0.1614;
17 exp(x) - F1;
18 r2 = ans(1,2)
19 x = 1.0;
20 exp(x) - F1;
21 r3 = ans(1,3)

```

```
22
23 F3 = 0.994526 + 0.995682*x + 0.543981*x*x +
      0.179519*x*x*x;
24 x = [-1.0 -0.6832 0.0493 0.7324 1.0]
25 exp(x) - F3 // relative errors
```

---



# Chapter 5

## Numerical Integration

Scilab code Exa 5.1 Integration

```
1          // PG (250)
2
3  defff(' [y]=f(x) ', 'y=(exp(x)-1)/x ')
4  x0=0;
5  x1=1;
6  integrate('(exp(x)-1)/x ', 'x ', x0, x1)
```

---

Scilab code Exa 5.2 Trapezoidal rule for integration

```
1          // PG (254)
2
3  defff(' [y]=f(x) ', 'y=exp(x)*cos(x) ')
4  defff(' [y]=fp(x) ', 'y=exp(x)*(cos(x)-sin(x)) ')
5  defff(' [y]=fpp(x) ', 'y=-2*exp(x)*sin(x) ')
6  x0=0;
7  x1=%pi;
8
9
```

```

10 //      True value
11 integrate('exp(x)*cos(x)', 'x', x0, x1)
12
13 //      Using Trapezoidal rule
14
15 n=2;
16 h=(x1-x0)/n;
17 I1 = (x1-x0) * (f(x0)+f(x1)) /4
18 E1 = -h^2 * (fp(x1)-fp(x0)) /12
19
20 n=4;
21 h=(x1-x0)/n;
22 I2 = (x1-x0) * (f(x0)+f(x1)) /4
23 E2 = -h^2 * (fp(x1)-fp(x0)) /12
24
25 n=8;
26 h=(x1-x0)/n;
27 I3 = (x1-x0) * (f(x0)+f(x1)) /4
28 E3 = -h^2 * (fp(x1)-fp(x0)) /12
29
30 n=16;
31 h=(x1-x0)/n;
32 I4 = (x1-x0) * (f(x0)+f(x1)) /4
33 E4 = -h^2 * (fp(x1)-fp(x0)) /12
34
35 n=32;
36 h=(x1-x0)/n;
37 I5 = (x1-x0) * (f(x0)+f(x1)) /4
38 E5 = -h^2 * (fp(x1)-fp(x0)) /12
39
40 n=64;
41 h=(x1-x0)/n;
42 I6 = (x1-x0) * (f(x0)+f(x1)) /4
43 E6 = -h^2 * (fp(x1)-fp(x0)) /12
44
45 n=128;
46 h=(x1-x0)/n;
47 I7 = (x1-x0) * (f(x0)+f(x1)) /4

```

```
48 E7 = -h^2 * (fp(x1)-fp(x0)) /12
```

---

### Scilab code Exa 5.3 Corrected trapezoidal rule

```
1          // PG (255)
2
3 deff(' [y]=f(x) ', 'y=exp(x)*cos(x) ')
4 deff(' [y]=fp(x) ', 'y=exp(x)*(cos(x)-sin(x)) ')
5 deff(' [y]=fpp(x) ', 'y=-2*exp(x)*sin(x) ')
6 x0=0;
7 x1=%pi;
8
9
10 // True value
11 integrate('exp(x)*cos(x)', 'x', x0, x1)
12
13 // Using Corrected Trapezoidal rule
14
15 n=2;
16 h=(x1-x0)/n;
17 I1 = ((x1-x0)/2) * (f(x0)+f(x1)) /2
18 E1 = -h^2 * (fp(x1)-fp(x0)) /12
19 C1 = I1 + E1
20
21 n=4;
22 h=(x1-x0)/n;
23 I2 = ((x1-x0)/2) * (f(x0)+f(x1)) /2
24 E2 = -h^2 * (fp(x1)-fp(x0)) /12
25 C2 = I2 + E2
26
27 n=8;
28 h=(x1-x0)/n;
29 I3 = ((x1-x0)/2) * (f(x0)+f(x1)) /2
30 E3 = -h^2 * (fp(x1)-fp(x0)) /12
31 C3 = I3 + E3
```

```

32
33 n=16;
34 h=(x1-x0)/n;
35 I4 = ((x1-x0)/2) * (f(x0)+f(x1)) /2
36 E4 = -h^2 * (fp(x1)-fp(x0)) /12
37 C4 = I4 + E4
38
39 n=32;
40 h=(x1-x0)/n;
41 I5 = ((x1-x0)/2) * (f(x0)+f(x1)) /2
42 E5 = -h^2 * (fp(x1)-fp(x0)) /12
43 C5 = I5 + E5
44
45 n=64;
46 h=(x1-x0)/n;
47 I6 = ((x1-x0)/2) * (f(x0)+f(x1)) /2
48 E6 = -h^2 * (fp(x1)-fp(x0)) /12
49 C6 = I6 + E6

```

---

#### Scilab code Exa 5.4 Simpson s rule for integration

```

1 // PG (258)
2
3 def f(' [y]=f(x) ', 'y=exp(x)*cos(x) ')
4 x0=0;
5 xn=%pi;
6 x=x0:xn;
7
8 // True value
9
10 I = integrate('exp(x)*cos(x) ', 'x ', x0, xn)
11
12 // Using Simpson's rule
13
14 N=2;

```

```

15 h=(xn-x0)/N;
16 x1=x0+h;
17 x2=x0+2*h;
18     I1 = h*(f(x0)+4*f(x1)+f(x2))/3
19
20 N=4;
21 h=(xn-x0)/N;
22 x1=x0+h;
23 x2=x0+2*h;
24 x3=x0+3*h;
25 x4=x0+4*h;
26     I2 = h*(f(x0)+4*f(x1)+2*f(x2)+4*f(x3)+f(x4))/3
27
28 N=8;
29 h=(xn-x0)/N;
30 x1=x0+h;
31 x2=x0+2*h;
32 x3=x0+3*h;
33 x4=x0+4*h;
34 x5=x0+5*h;
35 x6=x0+6*h;
36 x7=x0+7*h;
37 x8=x0+8*h;
38     I3 = h*(f(x0)+4*f(x1)+2*f(x2)+4*f(x3)+2*f(x4)+4*
           f(x5)+2*f(x6)+4*f(x7)+f(x8))/3

```

---

### Scilab code Exa 5.5 Trapezoidal and simpson integration

```

1 // PG (261)
2
3 // Example 1
4
5 def f(' [y]=f(x) ', 'y=x^(7/2) ')
6 def fp(' [y]=fp(x) ', 'y=3.5*x^(5/2) ')
7 def fpp(' [y]=fpp(x) ', 'y=8.75*x^(3/2) ')

```

```

8 def f(' [y]=fppp(x)', 'y=(105*sqrt(x))/8')
9 def g(' [y]=fpppp(x)', 'y=(105*x^(-0.5))/16')
10
11 x0=0;
12 xn=1;
13 x=x0:xn;
14
15 // True value
16 I = integrate('x^(7/2)', 'x', x0, xn)
17
18 // Using Trapezoidal rule
19
20 n=2;
21 h=(xn-x0)/n;
22 I1 = (xn-x0) * (f(x0)+f(xn)) /4;
23 E1 = -h^2 * (fp(xn)-fp(x0)) /12 // Error
24
25 n=4;
26 h=(xn-x0)/n;
27 I2 = (xn-x0) * (f(x0)+f(xn)) /4;
28 E2 = -h^2 * (fp(xn)-fp(x0)) /12 // Error
29
30 // Using Simpson's rule
31
32 N=2;
33 h=(xn-x0)/N;
34 x1=x0+h;
35 x2=x0+2*h;
36 I1 = h*(f(x0)+4*f(x1)+f(x2))/3
37 E1 = -h^4*(xn-x0)*fpppp(0.5)/180
38
39 N=4;
40 h=(xn-x0)/N;
41 x1=x0+h;
42 x2=x0+2*h;
43 x3=x0+3*h;
44 x4=x0+4*h;
45 I2 = h*(f(x0)+4*f(x1)+2*f(x2)+4*f(x3)+f(x4))/3

```

46  $E2 = -h^4*(x_n - x_0)*f_{pppp}(0.5)/180$

---

**Scilab code Exa 5.6 Newton Cotes formulae**

```

1          // PG (266)
2
3 //      Commonly used Newton Cotes formulae:-
4
5 //      n=1
6
7 //      h/2 * [f(a)+f(b)] - (h^3)*f''(e)/12-----
      Trapezoidal rule
8
9 //      n=2
10
11 //      h/3 * [f(a)+4*f((a+b)/2)+f(b)] - (h^5)*f^(4)(e
      )/90-----Simpson's rule
12
13 //      n=3
14
15 //      3*h/8 * [f(a)+3*f(a+h)+3*f(b-h)+f(b)] - (3*h
      ^5)*f^(4)(e)/80
16
17 //      n=4
18
19 //      2*h/45 * [7*f(a)+32*f(a+h)+12*f((a+b)/2)+32*f(
      b-h)+7*f(b)] - (8*h^7)*f^(7)(e)/945

```

---

check Appendix AP 6 for dependency:

legendrepol.sce

**Scilab code Exa 5.7 Gaussian Quadrature**

```

1 function [pL] = legendrepol(n,var)
2
3 //      Generates the Legendre polynomial
4 //      of order n in variable var
5
6 if n == 0 then
7     cc = [1];
8 elseif n == 1 then
9     cc = [0 1];
10 else
11     if modulo(n,2) == 0 then
12         M = n/2
13     else
14         M = (n-1)/2
15     end;
16
17     cc = zeros(1,M+1);
18     for m = 0:M
19         k = n-2*m;
20         cc(k+1)=...
21             (-1)^m*gamma(2*n-2*m+1)/(2^n*gamma(m+1)*
22                 gamma(n-m+1)*gamma(n-2*m+1));
23     end;
24 end;
25 pL = poly(cc,var,'coeff');
26 endfunction
27
28 //      End function legendrepol
29
30         //      PG (277)
31
32 deff(' [y]=f(x) ', 'y=exp(x)*cos(x) ')
33 x0=0;
34 x1=%pi;
35
36
37 //      True value

```



```

38 I = integrate('exp(x)*cos(x)', 'x', x0, x1)
39
40 //      Using Gaussian Quadrature
41
42 //      For n=2, w=1
43
44 n=2;
45 p = legendrepol(n, 'x')
46 xr = roots(p);
47 A = [];
48
49 for j = 1:2
50     pd = derivat(p)
51     A = [A 2/((1-xr(j))^2)*(horner(pd, xr(j)))^2)]
52 end
53
54 tr = ((x1-x0)/2.*xr)+((x1+x0)/2)

```

---

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

legendrepol.sce

### Scilab code Exa 5.8 Gaussian Legendre Quadrature

```

1 function [pL] = legendrepol(n, var)
2
3 //      Generates the Legendre polynomial
4 //      of order n in variable var
5
6 if n == 0 then
7     cc = [1];
8 elseif n == 1 then
9     cc = [0 1];
10 else
11     if modulo(n,2) == 0 then
12         M = n/2

```

```

13     else
14         M = (n-1)/2
15     end;
16
17     cc = zeros(1,M+1);
18     for m = 0:M
19         k = n-2*m;
20         cc(k+1)=...
21             (-1)^m*gamma(2*n-2*m+1)/(2^n*gamma(m+1)*
                gamma(n-m+1)*gamma(n-2*m+1));
22     end;
23 end;
24
25 pL = poly(cc,var,'coeff');
26 endfunction
27
28 //     End function legendrepol
29
30         //     PG (278)
31
32 deff(' [y]=f(x) ', 'y=exp(-x^2) ')
33 x0=0;
34 x1=1;
35
36
37 //     True value
38 I = integrate('exp(-x^2)', 'x', x0, x1)
39
40 //     Using Gaussian Quadrature
41
42 //     For n=2, w=1
43
44 n=2;
45 p = legendrepol(n, 'x')
46 xr = roots(p);
47 A = [];
48
49 for j = 1:2

```

```

50     pd = derivat(p)
51     A = [A 2/((1-xr(j)^2)*(horner(pd,xr(j)))^2)]
52 end
53
54 tr = ((x1-x0)./2.*xr)+((x1+x0)./2);
55
56 s = ((x1-x0)./2).*f(tr)
57 I = s*A

```

---

### Scilab code Exa 5.9 Integration

```

1         // PG (280)
2
3 I1 = integrate('sqrt(x)', 'x', 0, 1)
4
5 I2 = integrate('1/(1+(x-%pi)^2)', 'x', 0, 5)
6
7 I3 = integrate('exp(-x)*sin(50*x)', 'x', 0, 2*pi)

```

---

### Scilab code Exa 5.10 Simpson Integration error

```

1         // PG (292)
2
3 deff(' [y]=f(x)', 'y=x^(3/2)')
4 x0=0;
5 xn=1;
6 x=x0:xn;
7
8 // True value
9
10 I = integrate('x^(3/2)', 'x', 0, 1)
11
12 // Using Simpson's rule

```

```

13
14 N=2;
15 h=(xn-x0)/N;
16 x1=x0+h;
17 x2=x0+2*h;
18     I1 = h*(f(x0)+4*f(x1)+f(x2))/3
19     I-I1
20
21 N=4;
22 h=(xn-x0)/N;
23 x1=x0+h;
24 x2=x0+2*h;
25 x3=x0+3*h;
26 x4=x0+4*h;
27     I2 = h*(f(x0)+4*f(x1)+2*f(x2)+4*f(x3)+f(x4))/3
28     I-I2
29
30 N=8;
31 h=(xn-x0)/N;
32 x1=x0+h;
33 x2=x0+2*h;
34 x3=x0+3*h;
35 x4=x0+4*h;
36 x5=x0+5*h;
37 x6=x0+6*h;
38 x7=x0+7*h;
39 x8=x0+8*h;
40     I3 = h*(f(x0)+4*f(x1)+2*f(x2)+4*f(x3)+2*f(x4)+4*
41           f(x5)+2*f(x6)+4*f(x7)+f(x8))/3

```

---

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

romberg.sce

**Scilab code Exa 5.11 Romberg Integration**

```

1           // PG (297)
2
3  deff ('[y]=f(x)', 'y=exp(x)*cos(x)')
4  a=0;
5  b=%pi;
6  h=1;
7
8  // True value
9
10 I = integrate('exp(x)*cos(x)', 'x', a, b)
11
12 // Using Romberg integration
13
14 Romberg(a, b, f, h)

```

---

#### Scilab code Exa 5.12 Adaptive simpson

```

1           // PG (302)
2
3  deff ('[y]=f(x)', 'y=sqrt(x)')
4  funcprot(0)
5  a=0;
6  b=1;
7
8  // True value
9
10 I = integrate('sqrt(x)', 'x', a, b)

```

---

#### Scilab code Exa 5.13 Integration

```

1           // PG (307)
2
3  deff ('[y]=f(x)', 'y=sqrt(-log(x))')

```

```
4 funcprot(0)
5 a=0;
6 b=1;
7
8 // True value
9
10 I = integrate('sqrt(-log(x))', 'x', a, b)
```

---

#### Scilab code Exa 5.14 Integration

```
1 // PG (313)
2
3 def(' [y]=f(x)', 'y=(log(x))/(x+2)')
4 funcprot(0)
5 a=0;
6 b=1;
7
8 // True value
9
10 I = integrate('(log(x))/(x+2)', 'x', a, b)
```

---

# Chapter 6

## Numerical methods for ordinary differential equations

Scilab code Exa 6.1 1st order linear differential equation

```
1          // PG (334)
2
3 //      dy/dt=-y
4 function ydot=f(y,t),ydot=-y,
5 endfunction
6 y0=0;t0=0;t=0:1:%pi;
7 y=ode(y0,t0,t,f)
8 plot(t,y)
```

---

Scilab code Exa 6.4 Stability of solution

```
1          // PG (339)
2
3 //      dy/dx=100y -101*(%e)^(-x)
4 function ydox=f(x,y),ydox=100*y -101*(%e)^(-x),
5 endfunction
```

```

5 funcprot(0)
6 y0=1;
7 x0=0;
8 x=0:5;
9 y=ode(y0,x0,x,f)
10
11 //      Solution will be  $Y(x) = \exp(-x)$ 
12
13 //      For the perturbed problem,  $dy/dx = 100*y -$ 
14 //       $101*\exp(-x)$ ,  $y(0) = 1+e$ 
15 //      Solution will be  $Y(x;e) = \exp(-x) + e*\exp(100*$ 
16 //       $x)$ 
17 //      This rapidly departs from the true solution.

```

---

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

euler.sce

### Scilab code Exa 6.5 Euler method

```

1          //      PG (344)
2
3 //       $dy/dx = y$ 
4
5 //  $y'=f(x, t)$ 
6 def f(' [z]=f(x, y) ', ' z=y ');
7
8 // execute the function euler1 , so as to call it to
9 // evaluate the value of y,
10
11 [y,x] = Euler1(0.40,1,2.00,0.2,f) // h=0.2;
12
13 [y,x] = Euler1(0.40,1,2.00,0.1,f) // h=0.1;
14
15 [y,x] = Euler1(0.40,1,2.00,0.05,f) // h=0.05;

```



```

16
17 // True solution is
18 Y = exp(x)
19
20
21 // dy/dx = (1/(1+x^2)) - (2*y^2)
22
23 // y'=f(x,t)
24 def f(' [z]=f(x,y)', 'z=(1/(1+x^2)) - (2*y^2)');
25
26 // execute the function euler1 , so as to call it to
    evaluate the value of y,
27
28
29
30 [y,x] = Euler1(0,0,2,0.2,f) // h=0.2;
31
32 [y,x] = Euler1(0,0,2,0.1,f) // h=0.1;
33
34 [y,x] = Euler1(0,0,2,0.05,f) // h=0.05;
35
36 // True solution is
37 Y = x/(1+x^2)

```

---

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

euler.sce

### Scilab code Exa 6.6 Euler

```

1 // PG (351)
2
3 // dy/dx = -y + 2 * cos(x)
4
5 def f(' [y]=g(x,y)', 'y=-y+2*cos(x)')
6 y0=1;

```

```

7 x0=0;
8 xn=5;
9
10 // execute the function euler1 , so as to call it to
    evaluate the value of y,
11
12 [y,x] = Euler1(y0,x0,xn,0.04,g) // h = 0.04
13
14 [y,x] = Euler1(y0,x0,xn,0.02,g) // h = 0.02
15
16 [y,x] = Euler1(y0,x0,xn,0.01,g) // h = 0.01

```

---

**Scilab code Exa 6.7** Asymptotic error analysis

```

1 // PG (354)
2
3 // dy/dx = -y
4
5 def f( ' [z]=f(x,y) ', ' z=-y ' )
6 y0=1;
7
8 // True solution is
9 Y = exp(-x)
10 // The equation for D(x) is
11 // D'(x) = -D(x) + 0.5*exp(-x)
12 // D(0) = 0
13 // The solution is
14 // D(x) = 0.5*x*exp(-x)

```

---

**Scilab code Exa 6.9** Midpoint and trapezoidal method

```

1 // PG (357)
2

```

```

3 //      1. The mid-point method is defined by
4
5 //       $y(n+1) = y(n-1) + 2*h*f(xn,yn)$ -----n>=1
6
7 //      It is an explicit two-step method.
8
9
10 //      The trapezoidal method is defined by
11
12 //       $y(n+1) = yn + h*[f(xn,yn) + f(x(n+1),y(n+1))$ 
13 //      ]-----n>=0
14 //      It is an implicit one-step method.

```

---

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

euler.sce

### Scilab code Exa 6.10 Euler

```

1 //      PG (365)
2
3 deff ( ' [z]=g(x,y) ', ' z=-y ' )
4 [y,x] = Euler1(0.25,1,2.25,0.25,g)
5
6 //-----
7
8 deff ( ' [z]=f(x,y) ', ' z=x-y^2 ' )
9 [y,x] = Euler1(0.25,0,3.25,0.25,f)

```

---

check Appendix [AP 5](#) for dependency:

trapezoidal.sce

### Scilab code Exa 6.11 Trapezoidal method

```
1          //    PG (372)
2
3  deff(' [y]=f(x,y) ', 'y=-y^2 ')
4  [x,y] = trapezoidal(1,1,5,1,f)
```

---

### Scilab code Exa 6.16 Adams Moulton method

```
1          //    PG (389)
2
3  //    Using Adams–Moulton Formula
4
5  deff(' [z]=f(x,y) ', 'z=(1/(1+x^2))-2*y^2 ')
6  y0 = 0;
7
8  //    Solution is  $Y(x) = x/(1+x^2)$ 
9
10 function [y,x] = adamsmoulton4(y0,x0,xn,h,f)
11
12 //adamsmoulton4 4th order method solving ODE
13 // dy/dx = f(y,x), with initial
14 //conditions y=y0 at x=x0. The
15 //solution is obtained for x = [x0:h:xn]
16 //and returned in y
17
18 umaxAllowed = 1e+100;
19
20 x = [x0:h:xn]; y = zeros(x); n = length(y); y(1) =
    y0;
21 for j = 1:n-1
22 if j<3 then
23     k1=h*f(x(j),y(j));
24     k2=h*f(x(j)+h,y(j)+k1);
25     y(j+1) = y(j) + (k2+k1)/2;
```

```

26 end;
27
28 if j>=2 then
29     y(j+2) = y(j+1) + (h/12)*(23*f(x(j+1),y(j+1))
        )-16*f(x(j),y(j))+5*f(x(j-1),y(j-1)));
30 end;
31 end;
32 endfunction
33
34 adamsmoulton4(0,2.0,10.0,2.0,f)

```

---

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

euler.sce

#### Scilab code Exa 6.21 Euler method

```

1 // PG (405)
2
3 def f(' [y]=f(x,y)', 'y=lamda*y+(1-lamda)*cos(x)-(1+
    lamda)*sin(x)')
4 lamda = -1;
5 [x,y]=Euler1(1,1,5,0.5,f)
6 lamda = -10;
7 [x,y]=Euler1(1,1,5,0.1,f)
8 lamda = -50;
9 [x,y]=Euler1(1,1,5,0.01,f)

```

---

check Appendix [AP 5](#) for dependency:

trapezoidal.sce

#### Scilab code Exa 6.24 Trapezoidal method

```

1          //    PG (409)
2
3  deff( '[y]=f(x,y)', 'y=lamda*y+(1-lamda)*cos(x)-(1+
      lamda)*sin(x)')
4  lamda = -1;
5  [x,y]=trapezoidal(1,1,5,0.5,f)
6  lamda = -10;
7  [x,y]=trapezoidal(1,1,5,0.5,f)
8  lamda = -50;
9  [x,y]=trapezoidal(1,1,5,0.5,f)

```

---

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

bvpeigen.sce

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

eigenvectors.sce

### Scilab code Exa 6.31 Boundary value problem

```

1          //    PG (434)
2
3  //    2-point linear Boundary value problem
4
5
6  //    Boundary value problems with eigenvalues -
      case:  $d^2y/dx^2 + \lambda y = 0$ 
7  //    subject to  $y(0) = 0, y(1) = 0$ , where  $\lambda$  is
      unknown.
8  //    The finite-difference approximation is:
9  //     $(y(i-1) - 2y(i) + y(i+1))) = -\lambda \Delta x^2 y(i), i =$ 
      2, 3, ..., n-1
10
11
12 [x,y,lam] = BVPeigen1(1,5)

```

---

# Chapter 7

## Linear Algebra

Scilab code Exa 7.1 Orthonormal basis

```
1          // PG (470)
2
3 u1 = [1/2, sqrt(3)/2]
4 u2 = [-sqrt(3)/2, 1/2]
5
6 // For a given vector x = (x1,x2), it can be
   written as
7 // x = alpha1*u1 + alpha2*u2
8 // alpha1 = (x1+x2*sqrt(3))/2
9 // alpha2 = (x2-x1*sqrt(3))/2
10
11 // (1,0) = (1/2)*u1 - (sqrt(3)/2)*u2
```

---

Scilab code Exa 7.2 Canonical forms

```
1          // PG (476)
2
3 A = [0.2 0.6 0; 1.6 -0.2 0; -1.6 1.2 3.0]
```

```

4 U = [0.6 0 -0.8;0.8 0 0.6;0 1.0 0]
5 Ustar = inv(U)
6 T = Ustar*A*U
7 trace(A)
8 lam =spec(A)'
9 lam1 = lam(1,1)
10 lam2 = lam(1,2)
11 lam3 = lam(1,3)
12 lam1 + lam2 + lam3
13
14 //      trace(A) = lam1 + lam2 + lam3
15
16 det(A)
17 lam1*lam2*lam3
18
19 //      det(A) = lam1 * lam2 * lam3

```

---

### Scilab code Exa 7.3 Orthonormal eigen vectors

```

1 //      PG (477)
2
3 A = [2 1 0;1 3 1;0 1 2]
4 lam = spec(A)'
5 lam1 = lam(1,1)
6 lam2 = lam(1,2)
7 lam3 = lam(1,3)
8 //      Orthonormal Eigen vectors
9
10 u1 = (1/sqrt(3))*[1;-1;1]
11 u2 = (1/sqrt(2))*[1;0;-1]
12 u3 = (1/sqrt(6))*[1;2;1]

```

---

### Scilab code Exa 7.4 Vector and matrix norms



```

1          // PG (481)
2
3 x = [1,0,-1,2]
4     // 1-norm
5 norm(x,1)
6     // 2-norm
7 norm(x,2)
8     // infinity norm
9 norm(x,'inf')
```

---

#### Scilab code Exa 7.5 Frobenious norm

```

1          // PG (484)
2
3 // A be n * n
4 // norm(A*x,2)
5 // norm(A*x,2) <= norm(A,'fro') * norm(x,2)
6 // norm(A*B,'fro') = norm(A,'fro') * norm(B,'fro
   ')
```

---

#### Scilab code Exa 7.6 Norm

```

1          // PG (489)
2
3 A = [1 -2;-3 4]
4 norm(A,1)
5 norm(A,2)
6 norm(A,'inf')
7 lam = spec(A)
8 r = max(abs(lam))
9     //r <= norm(A,2)
```

---

Scilab code Exa 7.7 Inverse exists

```
1           // PG (494)
2
3 A = [4 1 0 0;1 4 1 0;0 1 4 1;0 0 1 4]
4 B = A/4 - eye()
5 norm(B, 'inf')
6 // Let (I+B = C)
7 C = eye() + B
8 inv(C)
9 // Inverse of (I + B) exists
10 norm(C, 'inf')
11 // Inverse of A exists.
```

---

# Chapter 8

## Numerical solution of systems of linear equations

Scilab code Exa 8.2 LU decomposition

```
1      //      EXAMPLE (PG 512)
2
3      A = [1 2 1;2 2 3;-1 -3 0]           //
         Coefficient matrix
4      b = [0 3 2]'                       //      Right
         hand matrix
5      [l,u] = lu(A)
6      //      l is lower triangular matrix & u is upper
         triangular matrix
7      l*u
8      if(A==l*u)
9          disp('A = LU is verified')
10     end
11     det(A)
12     det(u)
13     if(det(A)==det(u))
14         disp('Determinant of A is equal to that of its
             upper triangular matrix')
15
```

16 // Product rule of determinants is verified

---

#### Scilab code Exa 8.4 LU decomposition

```
1 // EXAMPLE (PG 518)
2
3 // Row interchanges on A can be represented
  by premultiplication of A
4 // by an appropriate matrix E, to get EA.
5 // Then, Gaussian Elimination leads to LU =
  PA
6
7 A = [0.729 0.81 0.9;1 1 1;1.331 1.21 1.1] //
  Coefficient Matrix
8 b = [0.6867 0.8338 1.000]' //
  Right Hand Matrix
9 [L,U,E] = lu(A)
10 // L is lower triangular matrix(mxn)
11 // U is upper triangular matrix(mxmin(m,n))
12 // E is permutation matrix(min(m,n)xn)
13 Z=L*U
14
15 disp("LU = EA")
16 E
17
18 // The result EA is the matrix A with first ,
  rows 1 & 3 interchanged ,
19 // and then rows 2 & 3 interchanged.
20
21 // NOTE:-According to the book, P is replaced
  by E here.
```

---

#### Scilab code Exa 8.5 Choleski Decomposition

```

1      //      EXAMPLE (PG 526)
2
3      disp("Consider Hilbert matrix of order three")
4
5      n=3;          //      Order of the matrix
6      A=zeros(n,n); //      a symmetric positive definite
          real or complex matrix.
7      for i=1:n    //      Initializing 'for' loop
8          for j=1:n
9              A(i,j)=1/(i+j-1);
10         end
11     end          //End of 'for' loop
12     A
13     chol(A)      //      Choleski
          Decomposition
14     L=[chol(A)]' //      Lower Triangular
          Matrix
15
16     //      The square roots obtained here can be
          avoided using a slight modification.
17     //      We find a diagonal matrix D & a lower
          triangular matrix (L^~),
18     //      with 1s on the diagonal such that A = (L
          ^~) * D * (L^~)',
19
20
21     //      chol(A) uses only the diagonal and upper
          triangle of A.
22     //      The lower triangular is assumed to be the
          (complex conjugate) transpose of the upper
23     //

```

---

**Scilab code Exa 8.6 LU decomposition**

```

1      //      EXAMPLE (PG 529)

```

```

2
3 // Consider the coefficient matrix for spline
  interpolation
4
5
6 A = [2 1 0 0;1 4 1 0;0 1 4 1;0 0 1 2]
7 [l,u] = lu(A); // LU Decomposition
8 U = l' // Lower Triangular matrix
9 L = u' // Upper triangular matrix

```

---

#### Scilab code Exa 8.7 Error analysis

```

1 // EXAMPLE (PG 531)
2
3 // Consider the linear system
4
5 // 7*x1 + 10*x2 = b1
6 // 5*x1 + 7*x2 = b2
7
8 A = [7 10;5 7] // Coefficient matrix
9 inv(A) // Inverse matrix
10
11 // cond(A)1 // Condition matrix
12
13 norm(A,1)*norm(inv(A),1)
14
15 // cond(A)2 // Condition matrix
16
17 norm(A,2)*norm(inv(A),2)
18
19 // These condition numbers all suggest that
  the above system
20 // may be sensitive to changes in the right
  side b.
21

```

```

22     //      Consider the particular case
23
24     b = [1 0.7]';           //      Right hand matrix
25     x = A\b;               //      Solution matrix
26
27     //      Solution matrix
28
29     x1 = x(1,:);
30     x2 = x(2,:);
31
32     //      For the perturbed system, we solve for:
33
34     b = [1.01 0.69]';     //      Right hand matrix
35     x = A\b;               //      Solution matrix
36
37     //      Solution matrix
38
39     x1 = x(1,:);
40     x2 = x(2,:);
41
42     //      The relative changes in x are quite large
43     //      when compared with
44     //      the size of the relative changes in the
45     //      right side b.

```

---

#### Scilab code Exa 8.8 Residual correction method

```

1     //      EXAMPLE (PG 541)
2
3     //      Consider a Hilbert matrix of order 3
4
5     n=3;                   //      Order of the matrix
6     A=zeros(n,n);         //      a symmetric positive definite
7     //      real or complex matrix.
8     for i=1:n              //      Initializing 'for' loop

```

```

8     for j=1:n
9         A(i,j)=1/(i+j-1);
10    end
11 end          //      End of 'for' loop
12 A
13
14    //      Rounding off to 4 decimal places
15 A = A*10^4;
16 A = int(A);
17 A = A*10^(-4);
18 disp(A)          //      Final Solution
19
20 H = A          //      Here H denoted H bar as denoted
    in the text
21
22 b = [1 0 0]';
23 x = H\b
24
25    //      Rounding off to 3 decimal places
26 x = x*10^3;
27 x = int(x);
28 x = x*10^(-3);
29 disp(x)          //      Final Solution
30
31 //Now, using elimination with Partial Pivoting, we
    get the following answers
32
33 x0 = [8.968 -35.77 29.77]';
34
35    //      ro is Residual correction
36
37 r0 = b - A*x0
38
39    //      A*e0 = r0
40
41 e0 = inv(A)*r0
42
43 x1 = x0 + e0

```



```

44
45 //           Repeating the above operations , we can
           get the values of r1 , x2 , e1 ...
46 //           The vector x2 is accurate to 4 decimal
           digits .
47 //           Note that  $x_1 - x_0 = e_0$  is an accurate
           predictor of the error  $e_0$  in  $x_0$ .

```

---

**Scilab code Exa 8.9** Residual correction method

```

1 //EXAMPLE (PG 544)
2
3 //A(e) = A0 + eB
4
5 A0=[2 1 0;1 2 1;0 1 2]
6 B=[0 1 1;-1 0 1;-1 -1 0]
7 //inv(A(e)) = C = inv(A0)
8 C=inv(A0)
9 b=[0 1 2]'
10 x=A0\b
11 r=b-A0*x

```

---

**Scilab code Exa 8.10** Gauss Jacobi method

```

1 //           EXAMPLE (PG 547)
2
3 //           Gauss Jacobi Method
4
5 A = [10 3 1;2 -10 3;1 3 10] //
           Coefficient Matrix
6 b = [14 -5 14]' //           Right
           hand matrix
7

```

```

8 x = [0 0 0] ' // Initial
      Gauss
9 d = diag(A) //
      Diagonal elements of matrix A
10 a11 = d(1,1)
11 a22 = d(2,1)
12 a33 = d(3,1)
13 D = [a11 0 0;0 a22 0;0 0 a33] //
      Diagonal matrix of A
14 [L,U] = lu(A) // L is lower triangular matrix, U
      is upper triangular matrix
15 H = -inv(D)*(L+U)
16 C = inv(D)*b
17
18 for(m=0:6) // Initialising 'for' loop for
      setting no of iterations to 6
19     x = H*x+C;
20     disp(x)
21     m=m+1;
22     x; // Solution
23     // Rounding off to 4 decimal places
24     x = x*10^4;
25     x = int(x);
26     x = x*10^(-4);
27     disp(x) // Final Solution
28
29 end

```

---

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

gaussseidel.sce

### Scilab code Exa 8.11 Gauss seidel method

```

1 //EXAMPLE (PG 549)
2

```

```

3      //Gauss Seidel Method
4
5  exec gaussseidel.sce
6  A = [10 3 1;2 -10 3;1 3 10]      //      Coefficient
      matrix
7  b = [14 -5 14]'                //      Right hand
      matrix
8  x0 = [0 0 0]'                  //      Initial Gauss
9  gaussseidel(A,b,x0)            //      Calling
      function
10
11      //      End the problem

```

---

#### Scilab code Exa 8.13 Conjugate gradient method

```

1      //      EXAMPLE (PG 568)
2
3  A= [5 4 3 2 1;4 5 4 3 2;3 4 5 4 3;2 3 4 5 4;1 2 3 4
      5]      //      Matrix of order 5
4      //      Getting the eigenvalues
5
6  lam = spec(A)                  //      lamda = spectral
      radius of matrix A
7
8  max(lam)                       //      Largest eigenvalue
9  min(lam)                       //      Smallest eigen
      value
10
11      //      For the error bound given earlier on
      Pg 567
12
13  c = min(lam)/max(lam)
14
15  (1-sqrt(c))/(1+sqrt(c))
16

```

```
17     // For linear system, choose the following
        values of b
18
19 b = [7.9380 12.9763 17.3057 19.4332 18.4196]';
20
21 x = A\b; // Solution matrix
22
23     // Rounding off to 4 decimal places
24 x = x*10^4;
25 x = int(x);
26 x = x*10^(-4)
27 disp(x) // Final Solution
```

---

# Chapter 9

## The Matrix Eigenvalue Problem

Scilab code Exa 9.1 Eigenvalues

```
1           // EXAMPLE 590
2
3 A = [4 1 0;1 0 -1;1 1 -4]
4 [n,m] = size(A);
5
6 if m<>n then
7     error('eigenvectors - matrix A is not square');
8     abort;
9 end;
10
11 lam = spec(A) '           //Eigenvalues of
    matrix A
```

---

Scilab code Exa 9.2 Eigen values and matrix norm

```
1           // PG 591
```

```

2
3 n = 4
4 A = [4 1 0 0;1 4 1 0;0 1 4 1;0 0 1 4]
5 lam = spec(A)
6
7 // Since A is symmetric, all eigen values are
  real.
8 // The radii are all 1 or 2.
9 // The centers of all the circles are 4.
10 // All eigen values must all lie in the interval
    [2,6]
11 // Since the eigen values of inv(A) are the
    reciprocals of those of A,
12 // 1/6 <= mu <= 1/2
13
14 // Let inv(A) = B
15
16 B=inv(A);
17 norm(B,2)
18 n
19 i = 1:n;
20 j = 1:n;
21
22 // for j~i
23 // r = sum(abs(B(i,j)))
24
25 // norm(B,2) = r(B) <= 0.5

```

---

### Scilab code Exa 9.3 Bounds for perturbed eigen values

```

1 // PG 593
2
3 disp("Consider Hilbert matrix of order three")
4
5 n=3; // Order of the matrix

```

```

6 A=zeros(n,n);// a symmetric positive definite
  real or complex matrix.
7 for i=1:n // Initializing 'for' loop
8   for j=1:n
9     A(i,j)=1/(i+j-1);
10  end
11 end //End of 'for' loop
12 A
13
14 [n,m] = size(A)
15
16 if m<>n then
17   error('eigenvectors - matrix A is not square');
18   abort;
19 end;
20
21 lam = spec(A)' //Eigenvalues of
  matrix A
22
23 lam1 = lam(1,1)
24 lam2 = lam(1,2)
25 lam3 = lam(1,3)
26
27 // Rounding off to 4 decimal places
28
29 A = A*10^4;
30 A = int(A);
31 A = A*10^(-4);
32 disp(A) // Final Solution
33
34 lamr = spec(A)'
35
36 lamr1 = lamr(1,1)
37 lamr2 = lamr(1,2)
38 lamr3 = lamr(1,3)
39
40 // Errors
41

```

```

42 lam = lamr
43
44     //     Relative Errors
45
46 R1 = (lam1-lamr1)/lam1
47 R2 = (lam2-lamr2)/lam2
48 R3 = (lam3-lamr3)/lam3

```

---

#### Scilab code Exa 9.4 Eigenvalues of nonsymmetric matrix

```

1         //     PG 594
2
3 A = [101 -90;110 -98]
4 [n,m] = size(A)
5
6 if m<>n then
7     error('eigenvectors - matrix A is not square');
8     abort;
9 end;
10
11 lam = spec(A) '           //Eigenvalues of
12     matrix A
13
14     //     A+E = [101-e -90-e;110 -98]
15     //     Let e = 0.001
16 e = 0.001;
17     //     Let A+E = D
18 D = [101-e -90-e;110 -98]
19
20 [n,m] = size(D)
21
22 if m<>n then
23     error('eigenvectors - matrix D is not square');
24     abort;

```



```

25 end;
26 lam = spec(D)' //Eigenvalues of
    matrix A

```

---

Scilab code Exa 9.5 Stability of eigenvalues for nonsymmetric matrices

```

1 // PG 599
2
3 // e = 0.001
4 // From earlier example :
5 // eigen values of matrix A are 1 and 2. So
6 // ...
7 // inv(P)*A*P = [1 0;0 2]
8
9 A = [101 -90;110 -98]
10 B = [-1 -1;0 0]
11 // From the above equation , we get:
12
13 P = [9/sqrt(181) -10/sqrt(221);10/sqrt(181) -11/sqrt
    (221)]
14 inv(P)
15 K = norm(P)*norm(inv(P)) // K is condition
    number
16 u1 = P(:,1)
17 u2 = P(:,2)
18 Q = inv(P)
19 R = Q'
20 w1 = R(:,1)
21 w2 = R(:,2)
22 s1 = 1/norm(w1,2)
23 norm(B)
24
25 // abs(lam1(e) - lam1) <= sqrt(2)*e/0.005 + O(e
    ^2) = 283*e + O(e^2)

```

---

Scilab code Exa 9.7 Rate of convergence

```
1          //      (PG 607)
2
3  A = [1 2 3;2 3 4;3 4 5]
4  lam = spec(A) '
5  lam1 = lam(1,3)
6  lam2 = lam(1,1)
7  lam3 = lam(1,2)
8
9          //      Theoretical ratio of convergence
10
11 lam2/lam1
12
13 b = 0.5*(lam2+lam3)
14 B = A-b*eye(3,3)
15
16          //      Eigen values of A-bI are:
17
18 lamb = spec(B) '
19 lamb1 = lamb(1,3)
20 lamb2 = lamb(1,2)
21 lamb3 = lamb(1,1)
22
23          //      Ratio of convergence for the power method
                applied to A-bI will be:
24
25 lamb2/lamb1
26
27          //      This is less than half the magnitude of
                the original ratio.
```

---

### Scilab code Exa 9.8 Rate of convergence after extrapolation

```
1          // PG (608)
2
3 A = [1 2 3;2 3 4;3 4 5]
4 lam = spec(A)' // Eigen values of A
5 lam1 = lam(1,3)
6 lam2 = lam(1,1)
7 lam3 = lam(1,2)
8
9 // Theoretical ratio of convergence
10
11 lam2/lam1
12
13 // After extrapolating, we get
14     lame1 = 9.6234814
15
16 // Error:
17 lam1-lame1
```

---

### Scilab code Exa 9.9 Householder matrix

```
1          // PG (610)
2
3 w = [1/3 2/3 2/3]'
4 w1 = w(1,1)
5 w2 = w(2,1)
6 w3 = w(3,1)
7
8 U = [1-2*abs(w1)^2 -2*w1*w2' -2*w1*w3'; -2*w1'*w2
      1-2*abs(w2)^2 -2*w2*w3'; -2*w1'*w3 -2*w2'*w3 1-2*
      abs(w3)^2]
9 U
10 inv(U)
11 // U = inv(U)-----Hence, U is Hermitian
```

```
12 U*U
13 // U*U = I—————Hence, U is orthogonal
```

---

### Scilab code Exa 9.11 QR factorisation

```
1 // PG (613)
2
3 A = [4 1 1;1 4 1;1 1 4]
4 w1 = [0.985599 0.119573 0.119573] '
5 P1 = eye() - 2*w1*w1 '
6 A2 = P1*A
7 w2 = [0 0.996393 0.0848572] '
8 P2 = eye() - 2*w2*w2 '
9 R = P2*A2
10 Q = P1*P2
11 Q*R
12
13 // A = Q * R
14
15 abs(det(A))
16 abs(det(Q)*det(R))
17
18 // |det(A)| = |det(Q)*det(R)| = |det(R)| = 54 (
    approx)
19
20 lam = spec(A) '
21 lam1 = lam(1,1)
22 lam2 = lam(1,2)
23 lam3 = lam(1,3)
24 lam1 * lam2 * lam3
25
26 // Product of eigen values also comes out to be
    54
```

---

### Scilab code Exa 9.12 Tridiagonal Matrix

```
1          // PG (617)
2
3 A = [1 3 4;3 1 2;4 2 1]
4 w2 = [0 2/sqrt(5) 1/sqrt(5)]'
5 P1 = eye() - 2*w2*w2'
6 T = P1' * A * P1 // Tridiagonal matrix
```

---

### Scilab code Exa 9.13 Planner Rotation Orthogonal Matrix

```
1          // PG (619)
2
3 x = %pi/4
4 R = [cos(x) 0 sin(x);0 1 0;-sin(x) 0 cos(x)]
5
6 // Planner Rotation Orthogonal Matrix
```

---

### Scilab code Exa 9.14 Eigen values of a symmetric tridiagonal Matrix

```
1          // PG (620)
2
3 T = [2 1 0 0 0 0;1 2 1 0 0 0;0 1 2 1 0 0;0 0 1 2 1
      0;0 0 0 1 2 1;0 0 0 0 1 2]
4 lam = spec(T)'
5 lam1 = lam(1,1)
6 B = [2-lam1 1 0 0 0 0;1 2-lam1 1 0 0 0;0 1 2-lam1 1
      0 0;0 0 1 2-lam1 1 0;0 0 0 1 2-lam1 1;0 0 0 0 1
      2]
```

```
7 f0 = abs(det(B))
8 f1 = 2-lam1
```

---

### Scilab code Exa 9.15 Sturm Sequence property

```
1          // PG (621)
2
3 // For the previous example, consider the
  sequence f0, f1....f6
4
5 // For lam = 3,
6
7 // (f0 ,..... f6) = (1, -1, 0, 1, -1, 0, 1)
8
9 // The corresponding sequence of signs is
10
11 // (+, -, +, +, -, +, +)
12
13 // and s(3) = 2
```

---

### Scilab code Exa 9.16 QR Method

```
1          // PG (624)
2
3 A1 = [2 1 0; 1 3 1; 0 1 4]
4 lam = spec(A1)'
5 [Q1, R1] = qr(A1);
6 A2 = R1 * Q1
7 [Q2, R2] = qr(A2);
8 A3 = R2 * Q2
9 [Q3, R3] = qr(A3);
10 A4 = R3 * Q3
11 [Q4, R4] = qr(A4);
```

```

12 A5 = R4 * Q4
13 [Q5,R5] = qr(A5);
14 A6 = R5 * Q5
15 [Q6,R6] = qr(A6);
16 A7 = R6 * Q6
17 [Q7,R7] = qr(A7);
18 A8 = R7 * Q7
19 [Q8,R8] = qr(A8);
20 A9 = R8 * Q8
21 [Q9,R9] = qr(A9);
22 A10 = R9 * Q9
23 [Q10,R10] = qr(A10);

```

---

Scilab code Exa 9.18 Calculation of Eigen vectors and Inverse iteration

```

1          // PG (631)
2
3 A = [2 1 0;1 3 1;0 1 4]
4 lam = spec(A)
5 [L,U] = lu(A)
6 y1 = [1 1 1]'
7 w1 = [3385.2 -2477.3 908.20]'
8 z1 = [w1/norm(w1,'inf')]
9 w2 = [20345 -14894 5451.9]'
10 z2 = [w2/norm(w2,'inf')]
11 z3 = z2
12
13 // The true answer is
14
15 x3 = [1 1-sqrt(3) 2-sqrt(3)]
16
17 // z2 equals x3 to within the limits of rounding
    error accumulations.

```

---

### Scilab code Exa 9.19 Inverse Iteration

```
1          // PG (633)
2
3 A = [2 1 0;1 3 1;0 1 4]
4 lam = spec(A)
5 [L,U] = lu(A)
6 y1 = [1 1 1]'
7 w1 = [3385.2 -2477.3 908.20]'
8 z1 = [w1/norm(w1,'inf')]
9 w2 = [20345 -14894 5451.9]'
10 z2 = [w2/norm(w2,'inf')]
11 z3 = z2
12
13 // The true answer is
14
15 x3 = [1 1-sqrt(3) 2-sqrt(3)]
16
17 // z2 equals x3 to within the limits of rounding
    error accumulations.
18
19 // Consider lam = 1.2679
20
21 // 0.7321*x1 + x2 = 0
22 // x1 + 1.7321*x2 + x3 = 0
23 // Taking x1= 1.0, we have the approximate
    eigenvector
24
25 // x = [1.0000 -0.73210 0.26807]
26
27
28 // Compared with the true answer obtained above,
    this is a slightly poorer
29 // result obtained by inverse iteration.
```

---



# Appendix

## Scilab code AP 1 Gauss seidel method

```
1 function [x]=gaussseidel(A,b,x0)
2 [nA,mA]=size(A)
3 n=nA
4 [L,U] = lu(A)
5 d = diag(A)
6 a11 = d(1,1)
7 a22 = d(2,1)
8 a33 = d(3,1)
9 D = [a11 0 0;0 a22 0;0 0 a33]
10 H = -inv(L+D)*U
11 C = inv(L+D)*b
12 for m=0:3
13     x = -inv(D)*(L+U)*x + inv(D)*b
14     m=m+1
15     disp(x)
16 end
17
18 endfunction
```

---

## Scilab code AP 2 Euler method

```
1 function [x,y] = Euler1(x0,y0,xn,h,g)
2
3 //Euler 1st order method solving ODE
4 // dy/dx = g(x,y), with initial
5 // conditions y=y0 at x = x0. The
```

```

6 //solution is obtained for x = [x0:h:xn]
7 //and returned in y
8
9 ymaxAllowed = 1e+100
10
11 x = [x0:h:xn];
12 y = zeros(x);
13 n = length(y);
14 y(1) = y0;
15
16 for j = 1:n-1
17     y(j+1) = y(j) + h*g(x(j),y(j));
18     if y(j+1) > ymaxAllowed then
19         disp('Euler 1 - WARNING: underflow or
20             overflow ');
21         disp('Solution sought in the following range:
22             ');
23         disp([x0 h xn]);
24         disp('Solution evaluated in the following
25             range: ');
26         disp([x0 h x(j)]);
27         n = j;
28         x = x(1,1:n); y = y(1,1:n);
29         break;
30     end;
31 end;
32 endfunction
33
34 //End function Euler1

```

---

### Scilab code AP 3 Eigen vectors

```

1 function [x,lam] = eigenvectors(A)
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 [n,m] = size(A);
9
10 if m<>n then
11     error('eigenvectors - matrix A is not square');
12     abort;
13 end;
14
15 lam = spec(A)'; //Eigenvalues of
    matrix A
16
17 x = [];
18
19 for k = 1:n
20     B = A - lam(k)*eye(n,n); //Characteristic matrix
21     C = B(1:n-1,1:n-1); //Coeff. matrix for
        reduced system
22     b = -B(1:n-1,n); //RHS vector for
        reduced system
23     y = C\b; //Solution for reduced system
24     y = [y;1]; //Complete eigenvector
25     y = y/norm(y); //Make unit eigenvector
26     x = [x y]; //Add eigenvector to matrix
27 end;
28
29 endfunction
30 //End of function

```

---

#### Scilab code AP 4 Boundary value problem

```

1 function [x,y,lam] = BVPeigen1(L,n)
2
3 Dx = L/(n-1);
4 x=[0:Dx:L];
5 a = 1/Dx^2;
6 k = n-2;
7

```

```

8 A = zeros(k,k);
9 for j = 1:k
10     A(j,j) = 2*a;
11 end;
12 for j = 1:k-1
13     A(j,j+1) = -a;
14     A(j+1,j) = -a;
15 end;
16
17 exec eigenvectors.sce
18
19 [yy,lam]=eigenvectors(A);
20 //disp('yy');disp(yy);
21
22 y = [zeros(1,k);yy;zeros(1,k)];
23 //disp('y');disp(y);
24
25
26 xmin=min(x);xmax=max(x);ymin=min(y);ymax=max(y);
27 rect = [xmin ymin xmax ymax];
28
29 if k>=5 then
30     m = 5;
31 else
32     m = k;
33 end
34
35
36 endfunction

```

---

#### Scilab code AP 5 Trapezoidal method

```

1 function [x,y] = trapezoidal(x0,y0,xn,h,g)
2
3 //Trapezoidal method solving ODE
4 // dy/dx = g(x,y), with initial
5 //conditions y=y0 at x = x0. The
6 //solution is obtained for x = [x0:h:xn]

```

```

7 //and returned in y
8
9 ymaxAllowed = 1e+100
10
11 x = [x0:h:xn];
12 y = zeros(x);
13 n = length(y);
14 y(1) = y0;
15
16 for j = 1:n-1
17     y(j+1) = y(j) + h*(g(x(j),y(j))+g(x(j+1),y(j+1)))
18         )/2;
19     if y(j+1) > ymaxAllowed then
20         disp('Euler 1 - WARNING: underflow or
21             overflow ');
22         disp('Solution sought in the following range:
23             ');
24         disp([x0 h xn]);
25         disp('Solution evaluated in the following
26             range: ');
27         disp([x0 h x(j)]);
28         n = j;
29         x = x(1,1:n); y = y(1,1:n);
30         break;
31     end;
32 end;
33 endfunction
34
35 //End function trapezoidal

```

---

#### Scilab code AP 6 Legendre Polynomial

```

1
2 function [pL] = legendrepol(n,var)
3
4 // Generates the Legendre polynomial
5 // of order n in variable var

```

```

6
7 if n == 0 then
8     cc = [1];
9 elseif n == 1 then
10    cc = [0 1];
11 else
12     if modulo(n,2) == 0 then
13         M = n/2
14     else
15         M = (n-1)/2
16     end;
17
18     cc = zeros(1,M+1);
19     for m = 0:M
20         k = n-2*m;
21         cc(k+1) = ...
22             (-1)^m*gamma(2*n-2*m+1)/(2^n*gamma(m+1)*
                gamma(n-m+1)*gamma(n-2*m+1));
23     end;
24 end;
25
26 pL = poly(cc,var,'coeff');
27
28 // End function legendrepol

```

---

#### Scilab code AP 7 Romberg Integration

```

1 function [I]=Romberg(a,b,f,h)
2
3 // This function calculates the numerical
  integral of f(x) between
4 // x = a and x = b, with intervals h.
  Intermediate results are obtained
5 // by using SCILAB's own intrap function
6
7 x=(a:h:b)
8 x1=x(1,1)
9 x2=x(1,2)

```

```

10 x3=x(1,3)
11 x4=x(1,4)
12 y1=f(x1)
13 y2=f(x2)
14 y3=f(x3)
15 y4=f(x4)
16 y=[y1 y2 y3 y4]
17 I1 = inttrap(x,y)
18 x=(a:h/2:b)
19 x1=x(1,1)
20 x2=x(1,2)
21 x3=x(1,3)
22 x4=x(1,4)
23 x5=x(1,5)
24 x6=x(1,6)
25 x7=x(1,7)
26 y1=f(x1)
27 y2=f(x2)
28 y3=f(x3)
29 y4=f(x4)
30 y5=f(x5)
31 y6=f(x6)
32 y7=f(x7)
33 y=[y1 y2 y3 y4 y5 y6 y7]
34 I2 = inttrap(x,y)
35 I = I2 + (1.0/3.0)*(I2-I1)
36
37 endfunction
38 //end function Romberg

```

---

#### Scilab code AP 8 Lagrange

```

1 function [P]=lagrange(X,Y)
2
3     // X nodes ,Y values
4     // P is the numerical Lagrange polynomial
5     // interpolation
6     n=length(X)

```

```

6      //      n is the number of nodes. (n-1) is the
           degree
7  x=poly(0,"x")
8  P=0
9  for i=1:n, L=1
10     for j=[1:i-1,i+1:n] L=L*(x-X(j))/(X(i)-X(j))
11         end
12 P=P+L*Y(i)
13 end
14 endfunction

```

---

#### Scilab code AP 9 Muller method

```

1  function x=muller(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

```

---

#### Scilab code AP 10 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 11 Newton

```

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

```

```

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

```

---

### Scilab code AP 12 Aitken1

```

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 13 Bisection method

```

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

```

```

        at a,
10         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 achieved,
25         end
26     end
27     disp(n," no. of iterations =")
        // display the no. of iterations took to
        achive required condition,
28 endfunction

```

---