Tutorial 5

Numerical Analysis with MATLAB

Math 475/CS 475

MATLAB has many tools that make this package well suited for numerical computations. This tutorial deals with the rootfinding, interpolation, numerical differentiation and integration and numerical solutions of the ordinary differential equations. Numerical methods of linear algebra are discussed in Tutorial 4.

5.1 MATLAB functions used in Tutorial 5

Function	Description
abs	Absolute value
dblquad	Numerically evaluate double integral
erf	Error function
feval	Execute function specified by string
fzero	Scalar nonlinear zero finding
gamma	Gamma function
inline	Construct INLINE object
interp1	One-dimensional interpolation
interp2	Two-dimensional interpolation
linspace	Evenly spaced vector
meshgrid	X and Y arrays for 3-D plots
norm	Matrix or vector norm
ode23	Solve non-stiff differential equations
ode45	Solve non-stiff differential equations
ode113	Solve non-stiff differential equations
ode15s	Solve stiff differential equations
ode23s	Solve stiff differential equations
poly	Convert roots to polynomial
polyval	Evaluate polynomial
ppval	Evaluate piecewise polynomial
quad	Numerically evaluate integral, low order method
quad8	Numerically evaluate integral, higher order method
rcond	Reciprocal condition estimator
roots	Find polynomial roots
spline	Cubic spline data interpolation
surf	3-D colored surface
unmkpp	Supply details about piecewise polynomial

5.2 Rootfinding

A central problem discussed in this section is formulated as follows. Given a real-valued function **f: $\mathbb{R}^n \to \mathbb{R}^n$, n ⩾ 1**, find a vector **r** so that **f(r) = 0**. Vector **r** is called the *root* or *zero* of **f**.

5.2.1 Computing roots of the univariate polynomials

Polynomials are represented in MATLAB by their coefficients in the descending order of powers. For instance, the cubic polynomial **$p(x) = 3x^3 + 2x^2 - 1$** is represented as

```
p = [3 2 0 1];
```

Its roots can be found using function **roots**

```
format long

r = roots(p)

r =
 -1.00000000000000
  0.16666666666667 + 0.55277079839257i
  0.16666666666667 - 0.55277079839257i
```

To check correctness of this result we evaluate **p(x)** at **r** using function **polyval**

```
err = polyval(p, r)

err =
  1.0e-014 *
  0.22204460492503
                  0 + 0.01110223024625i
                  0 - 0.01110223024625i
```

To reconstruct a polynomial from its roots one can use function **poly**. Using the roots **r** computed earlier we obtain

```
poly(r)

ans =
   1.00000000000000   0.66666666666667   0.00000000000000
0.33333333333333
```

Let us note that these are the coefficients of **p(x)** all divided by 3. The coefficients of **p(x)** can be recovered easily

```
3*ans

ans =
   3.00000000000000   2.00000000000000   0.00000000000000
1.00000000000000
```

Numerical computation of roots of a polynomial is the *ill-conditioned* problem. Consider the fifth degree polynomial $\mathbf{p(x) = x^5 - 10x^4 + 40x^3 - 80x^2 + 80x - 32}$. Let us note that $\mathbf{p(x) = (x-2)^5}$. Using function **roots** we find

```
format short

p = [1 -10 40 -80 80 -32];

x = roots(p)

x =
   2.0017
   2.0005 + 0.0016i
   2.0005 - 0.0016i
   1.9987 + 0.0010i
   1.9987 - 0.0010i
```

These results are not satisfactory. We will return to the problem of finding the roots of **p(x)** in the next section.

5.2.2 Finding zeros of the univariate functions using MATLAB function fzero

Let now **f** be a transcendental function from $\mathbb{R}$ to $\mathbb{R}$. MATLAB function **fzero** computes a zero of the function **f** using user supplied initial guess of a zero sought.

In the following example let **f(x) = cos(x) – x**. First we define a function **y = f1(x)**

```
function y = f1(x)

% A univariate function with a simple zero.

y = cos(x) - x;
```

To compute its zero we use MATLAB function **fzero**

```
r = fzero('f1', 0.5)

r =
   0.73908513321516
```

Name of the function whose zero is computed is entered as a string. Second argument of function **fzero** is the initial approximation of **r**. One can check last result using function **feval**

```
err = feval('f1', r)

err =
     0
```

In the case when a zero of function is bracketed a user can enter a two-element vector that designates a starting interval. In our example we choose **[0 1]** as a starting interval to obtain

```
r = fzero('f1', [0 1])

r =
   0.73908513321516
```

However, this choice of the designated interval

```
fzero('f1', [1 2])

¨??? Error using ==> fzero
The function values at the interval endpoints must differ in sign.
```

generates the error message.

By adding the third input parameter **tol** you can force MATLAB to compute the zero of a function with the relative error tolerance **tol**. In our example we let **tol = 10^{-3}** to obtain

```
rt = fzero('f1', .5, 1e-3)

rt =
   0.73886572291538
```

A relative error in the computed zero **rt** is

```
rel_err = abs(rt-r)/r

rel_err =
    2.969162630892787e-004
```

Function **fzero** takes fourth optional parameter. If it is set up to 1, then the iteration information is displayed. Using function **f1**, with **x0 = 0.5**, we obtain

```
format short

rt = fzero('f1', .5, eps, 1)

 Func evals      x           f(x)          Procedure
    1             0.5      0.377583        initial
    2        0.485858      0.398417        search
    3        0.514142      0.356573        search
    4            0.48      0.406995        search
    5            0.52      0.347819        search
    6        0.471716      0.419074        search
    7        0.528284      0.335389        search
    8            0.46      0.436052        search
    9            0.54      0.317709        search
   10        0.443431      0.459853        search
   11        0.556569      0.292504        search
   12            0.42      0.493089        search
   13            0.58      0.256463        search
   14        0.386863      0.539234        search
   15        0.613137       0.20471        search
   16            0.34      0.602755        search
   17            0.66      0.129992        search
   18        0.273726      0.689045        search
```

```
   19        0.726274     0.0213797        search
   20            0.18      0.803844        search
   21            0.82     -0.137779        search

   Looking for a zero in the interval [0.18, 0.82]

   22        0.726355     0.0212455        interpolation
   23        0.738866    0.00036719        interpolation
   24        0.739085 -6.04288e-008        interpolation
   25        0.739085  2.92788e-012        interpolation
   26        0.739085             0        interpolation
rt =
    0.7391
```

We have already seen that MATLAB function **roots** had faild to produce satisfactory results when computing roots of the polynomial $\mathbf{p(x) = (x - 2)^5}$. This time we will use function **fzero** to find a multiple root of **p(x)**. We define a new function named **f2**

```
function y = f2(x)

y = (x - 2)^5;
```

and next change format to

```
format long
```

Running function **fzero** we obtain

```
rt = fzero('f2', 1.5)

rt =
   2.00000000000000
```

This time the result is as expected.

Finally, we will apply function **fzero** to compute the multiple root of **p(x)** using an expanded form of the polynomial **p(x)**

```
function y = f3(x)

y = x^5 - 10*x^4 + 40*x^3 -80*x^2 + 80*x - 32;

rt = fzero('f3', 1.5)

rt =
   1.99845515925755
```

Again, the computed approximation of the root of **p(x)** has a few correct digits only.

5.2.3 The Newton-Raphson method for systems of nonlinear equations

This section deals with the problem of computing zeros of the vector-valued function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$, $n \geq 1$. Assume that the first order partial derivatives of $\mathbf{f}$ are continuous on an open domain holding all zeros of $\mathbf{f}$. A method discussed below is called the *Newton-Raphson method*. To present details of this method let us introduce more notation. Using MATLAB's convention for representing vectors we write $\mathbf{f}$ as a column vector $\mathbf{f} = [f_1; \ldots; f_n]$, where each f_k is a function from $\mathbb{R}^n$ to $\mathbb{R}$. Given an initial approximation $\mathbf{x}^{(0)} \in \mathbb{R}^n$ of $\mathbf{r}$ this method generates a sequence of vectors $\{\mathbf{x}^{(k)}\}$ using the iteration

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - J_f(\mathbf{x}^{(k)})^{-1} \mathbf{f}(\mathbf{x}^{(k)}), \qquad k = 0, 1, \ldots .$$

Here J_f stands for the *Jacobian matrix* of $\mathbf{f}$, i.e., $J_f(\mathbf{x}) = [\partial f_i(\mathbf{x})/\partial x_j]$, $1 \leq i, j \leq n$. For more details the reader is referred to [6] and [9].

Function **NR** computes a zero of the system of nonlinear equations.

```
function [r, niter] = NR(f, J, x0, tol, rerror, maxiter)

% Zero r of the nonlinear system of equations f(x) = 0.
% Here J is the Jacobian matrix of f and x0 is the initial
% approximation of the zero r.
% Computations are interrupted either if the norm of
% f at current approximation is less (in magnitude)
% than the number tol,or if the relative error of two
% consecutive approximations is smaller than the prescribed
% accuracy rerror, or if the number of allowed iterations
% maxiter is attained.
% The second output parameter niter stands for the number
% of performed iterations.

Jc = rcond(feval(J,x0));
if Jc < 1e-10
   error('Try a new initial approximation x0')
end
   xold = x0(:);
   xnew = xold - feval(J,xold)\feval(f,xold);
for k=1:maxiter
   xold = xnew;
   niter = k;
   xnew = xold - feval(J,xold)\feval(f,xold);
   if (norm(feval(f,xnew)) < tol) |...
         norm(xold-xnew,'inf')/norm(xnew,'inf') < tol|...
         (niter == maxiter)
      break
   end
end
r = xnew;
```

The following nonlinear system

$$f_1(\mathbf{x}) = x_1 + 2x_2 - 2,$$
$$f_2(\mathbf{x}) = x_1^2 + 4x_2^2 - 4$$

has the exact zeros **r = [0 1]^T** and **r = [2 0]^T** (see [6], p. 166). Functions **fun1** and **J1** define the system of equations and its Jacobian, respectively

```
function z = fun1(x)

z = zeros(2,1);
z(1) = x(1) + 2*x(2) - 2;
z(2) = x(1)^2 + 4*x(2)^2 - 4;


function s = J1(x)

s = [1 2;2*x(1) 8*x(2)];
```

Let

```
x0 = [0 0];
```

Then

```
[r, iter] = NR('fun1', 'J1', x0, eps, eps, 10)

¨??? Error using ==> nr
Try a new initial approximation x0
```

For **x0** as chosen above the associated Jacobian is singular. Let's try another initial guess for **r**

```
x0 = [1 0];

[r, niter] = NR('fun1', 'J1', x0, eps, eps, 10)

r =
   2.00000000000000
  -0.00000000000000
niter =
     5
```

Consider another nonlinear system

$f_1(x) = x_1 + x_2 - 1$
$f_2(x) = \sin(x_1^2 + x_2^2) - x_1.$

The m-files needed for computing its zeros are named **fun2** and **J2**

```
function w = fun2(x);

w(1) = x(1) + x(2) - 1;
w(2) = sin(x(1)^2 + x(2)^2) - x(1);
w = w(:);
```

```
function s = J2(x)

s = [1 1;
       2*x(1)*cos(x(1)^2 + x(2)^2)-1 2*x(2)*cos(x(1)^2 + x(2)^2)];
```

With the initial guess

```
x0 = [0 1];
```

the zero **r** is found to be

```
 [r, niter] = NR('fun2', 'J2', x0, eps, eps, 10)

r =
   0.48011911689839
   0.51988088310161
niter =
     5
```

while the initial guess

```
x0 = [1 1];

[r, iter] = NR('fun2', 'J2', x0, eps, eps, 10)

r =
  -0.85359545600207
   1.85359545600207
iter =
    10
```

gives another solution. The value of function **fun2** at the computed zero **r** is

```
feval('fun2', r)

ans =
  1.0e-015 *
                    0
  -0.11102230246252
```

Implementation of other classical methods for computing the zeros of scalar equations, including the fixed-point iteration, the secant method and the Schroder method are left to the reader (see Problems 3, 6, and 12 at the end of this tutorial).

5.3 One Dimensional Interpolation

Interpolation of functions is one of the classical problems in numerical analysis. A one dimensional interpolation problem is formulated as follows.

Given set of **n+1** points $\{x_k , y_k\}$, $0 \leq k \leq n$, with $x_0 < x_1 < \ldots < x_n$, find a function **f(x)** whose graph interpolates the data points, i.e., $f(x_k) = y_k$, for $k = 0, 1, \ldots, n$.

In this section we will use as the interpolating functions algebraic polynomials and *spline functions*.

5.3.1 MATLAB function interp1

The general form of the function **interp1** is **yi = interp1(x, y, xi, method)**, where the vectors **x** and **y** are the vectors holding the x- and the y- coordinates of points to be interpolated, respectively, **xi** is a vector holding points of evaluation, i.e., **yi = f(xi)** and **method** is an optional string specifying an interpolation method. The following methods work with the function **interp1**

- Nearest neighbor interpolation, method = **'nearest'**. Produces a locally piecewise constant interpolant.
- Linear interpolation method = **'linear'**. Produces a piecewise linear interpolant.
- Cubic spline interpolation, method = **'spline'**. Produces a cubic spline interpolant.
- Cubic interpolation, method = **'cubic'**. Produces a piecewise cubic polynomial.

In this example, the following points $(x_k, y_k) = (k\pi/5, \sin(2x_k))$, $k = 0, 1, \ldots, 5$,

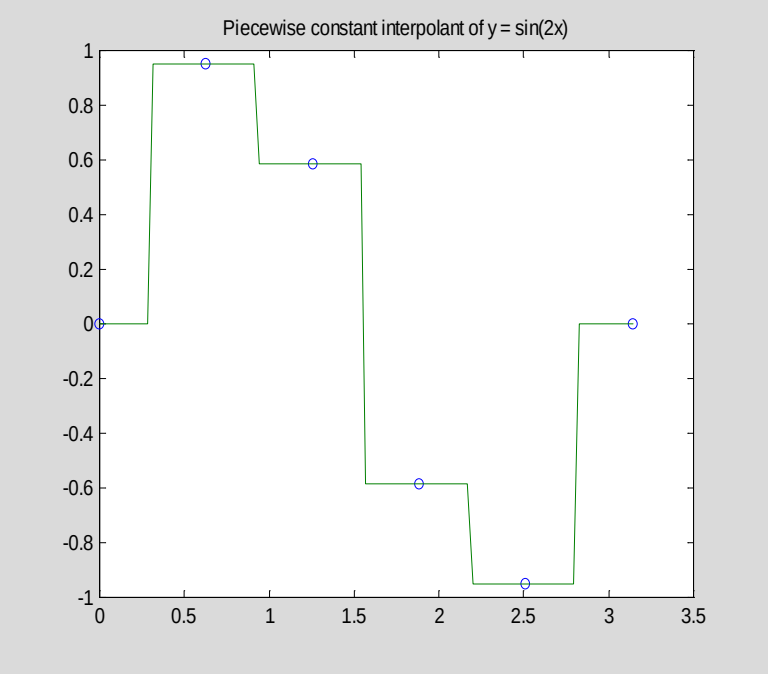
```
x = 0:pi/5:pi;

y = sin(2.*x);
```

are interpolated using two methods of interpolation **'nearest'** and **'cubic'** . The interpolant is evaluated at the following points

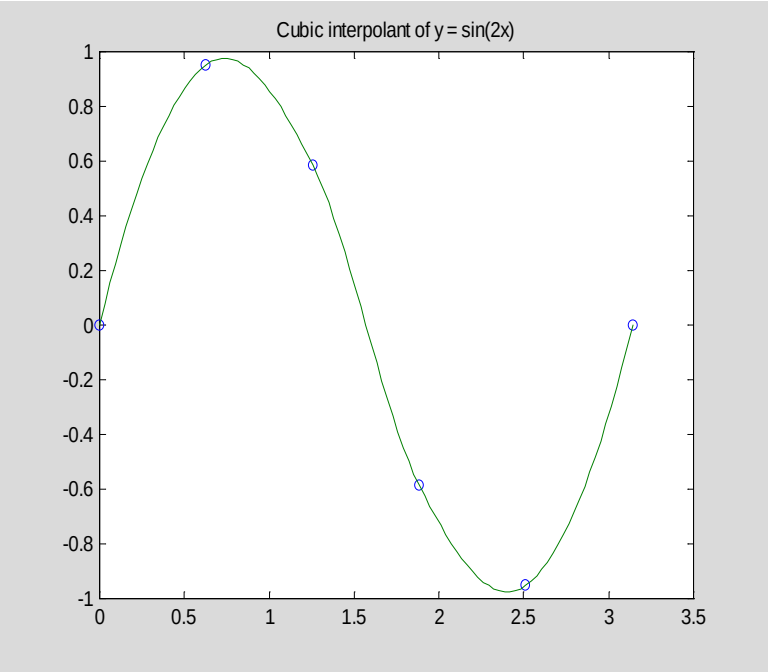
```
xi = 0:pi/100:pi;

yi = interp1(x, y, xi, 'nearest');
```

Points of interpolation together with the resulting interpolant are displayed below

```
plot(x, y, 'o', xi, yi), title('Piecewise constant interpolant of y =
sin(2x)')
```




```
yi = interp1(x, y, xi, 'cubic');
plot(x, y, 'o', xi, yi), title('Cubic interpolant of y = sin(2x)')
```



5.3.2 Interpolation by algebraic polynomials

Assume now that the interpolating function is an algebraic polynomial $\mathbf{p_n(x)}$ of degree at most **n**, where **n** = number of points of interpolation – 1. It is well known that the interpolating polynomial $\mathbf{p_n}$ always exists and is unique (see e.g., [6], [9]). To determine the polynomial interpolant one can use either the Vandermonde's method or Lagrange form or Newton's form or Aitken's method. We shall describe briefly the Newton's method.

We begin writing $\mathbf{p(x)}$ as

(5.3.1) $$\mathbf{p_n(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \ldots + a_n(x - x_0)(x - x_1) \ldots (x - x_{n-1})}$$

Coefficients $\mathbf{a_0}$, $\mathbf{a_1}$, … , $\mathbf{a_n}$ are called the *divided differences* and they can be computed recursively. Representation (5.3.1) of $\mathbf{p_n(x)}$ is called the *Newton's form* of the interpolating polynomial. The k-th order divided difference based on points $\mathbf{x_0}$, … $\mathbf{x_k}$, denoted by $\mathbf{[x_0, \ldots, x_k]}$, is defined recursively as

$$\mathbf{[x_m] = y_m} \quad \text{if } \mathbf{k = 0}$$

$$\mathbf{[x_0, \ldots, x_k] = ([x_1, \ldots, x_k] - [x_0, \ldots, x_{k-1}])/(x_k - x_0)} \quad \text{if } \mathbf{k > 0}.$$

Coefficients $\mathbf{\{a_k\}}$ in representation (5.3.1) and the divided differences are related in the following way

$$\mathbf{a_k = [x_0, \ldots, x_k]}.$$

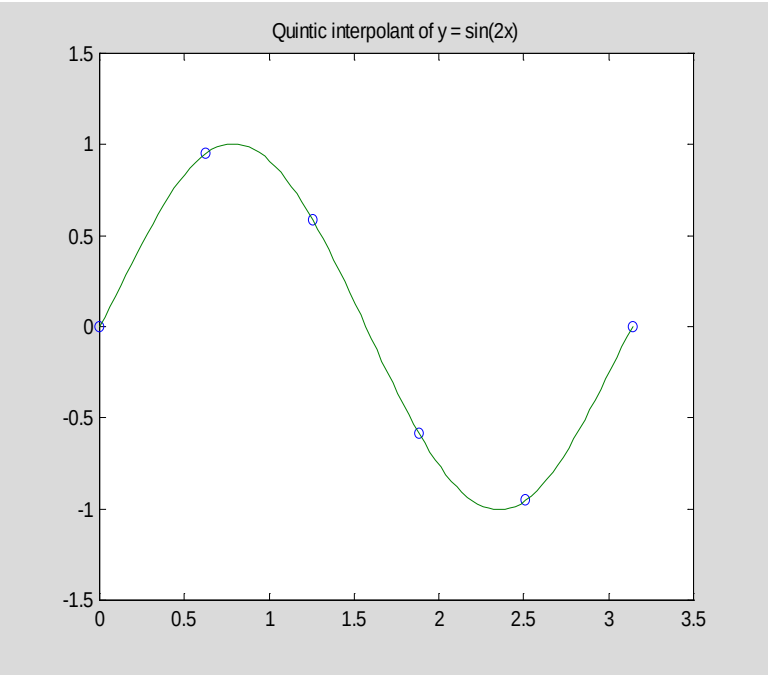
Function **Newtonpol** evaluates an interpolating polynomial at the user supplied points.

```
function [yi, a] = Newtonpol(x, y, xi)

% Values yi of the interpolating polynomial at the points xi.
% Coordinates of the points of interpolation are stored in
% vectors x and y. Horner's method is used to evaluate
% a polynomial. Second output parameter a holds coeeficients
% of the interpolating polynomial in Newton's form.

a = divdiff(x, y);
n = length(a);
val = a(n);
for m = n-1:-1:1
   val = (xi - x(m)).*val + a(m);
end
yi = val(:);


function a = divdiff(x, y)

% Divided differences based on points stored in arrays x and y.

n = length(x);
for k=1:n-1
```

```
   y(k+1:n) = (y(k+1:n) - y(k))./(x(k+1:n) - x(k));
end
a = y(:);
```

For the data of the last example, we will evaluate Newton's interpolating polynomial of degree at most five, using function **Newtonpol**. Also its graph together with the points of interpolation will be plotted.

```
[yi, a] = Newtonpol(x, y, xi);

plot(x, y, 'o', xi, yi), title('Quintic interpolant of y = sin(2x)')
```



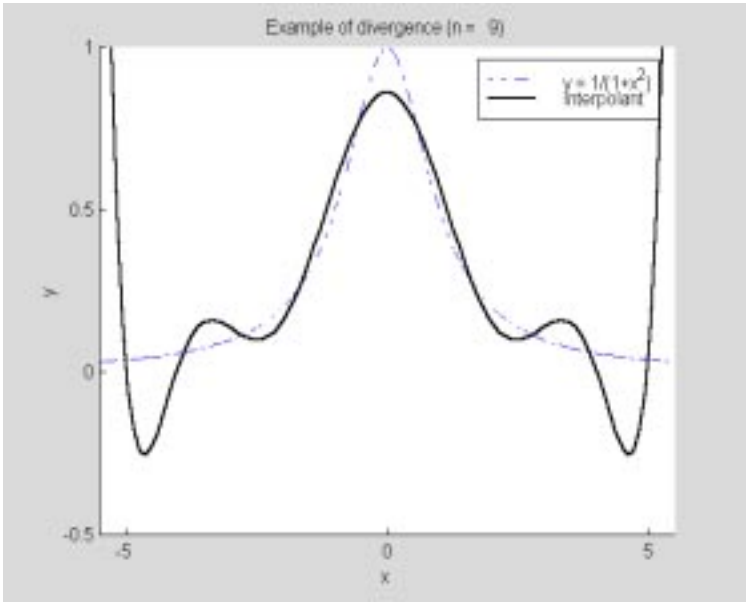
Interpolation process not always produces a sequence of polynomials that converge uniformly to the interpolated function as degree of the interpolating polynomial tends to infinity. A famous example of divergence, due to Runge, illustrates this phenomenon. Let $g(x) = 1/(1 + x^2)$, $-5 \leqslant x \leqslant 5$, be the function that is interpolated at $n + 1$ evenly spaced points $x_k = -5 + 10k/n$, $k = 0, 1, \ldots, n$.

Script file **showint** creates graphs of both, the function $g(x)$ ant its interpolating polynomial $p_n(x)$.

```
% Script showint.m
% Plot of the function 1/(1 + x^2) and its
% interpolating polynomial of degree n.

m = input('Enter number of interpolating polynomials ');
```

```
for k=1:m
   n = input('Enter degree of the interpolating polynomial ');
   hold on
   x = linspace(-5,5,n+1);
   y = 1./(1 + x.*x);
   z = linspace(-5.5,5.5);
   t = 1./(1 + z.^2);
   h1_line = plot(z,t,'-.');
   set(h1_line, 'LineWidth',1.25)
   t = Newtonpol(x,y,z);
   h2_line = plot(z,t,'r');
   set(h2_line,'LineWidth',1.3,'Color',[0 0 0])
   axis([-5.5 5.5 -.5 1])
   title(sprintf('Example of divergence (n =  %2.0f)',n))
   xlabel('x')
   ylabel('y')
   legend('y = 1/(1+x^2)','interpolant')
   hold off
end
```

Typing **showint** in the Command Window you will be prompted to enter value for the parameter **m** = number of interpolating polynomials you wish to generate and also you have to enter value(s) of the degree of the interpolating polynomial(s). In the following example **m = 1** and **n = 9**



Divergence occurs at points that are close enough to the endpoints of the interval of interpolation **[-5, 5]**.

We close this section with the *two-point Hermite interpolaion* problem by cubic polynomials. Assume that a function **y= g(x)** is differentiable on the interval **[a, b]**. We seek a cubic polynomial $\mathbf{p_3(x)}$ that satisfies the following interpolatory conditions

(5.3.2) $p_3(a) = g(a), \quad p_3(b) = g(b), \quad p_3'(a) = g'(a), \quad p_3'(b) = g'(b)$

Interpolating polynomial $p_3(x)$ always exists and is represented as follows

(5.3.3) $$p_3(x) = (1 + 2t)(1 - t)^2 g(a) + (3 - 2t)t^2 g(b) + h[t(1 - t)^2 g'(a) + t^2(t - 1)g'(b)] ,$$

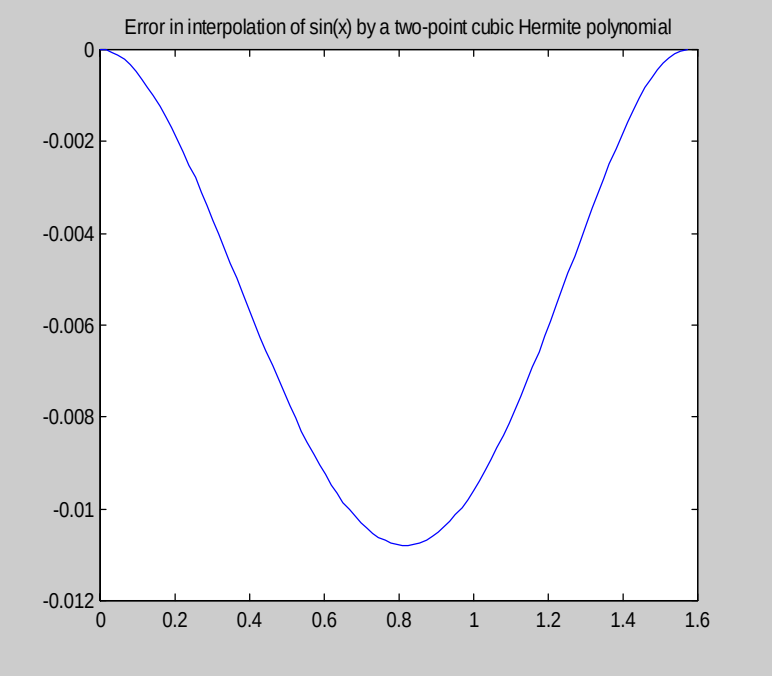
where $t = (x - a)/(b - a)$ and $h = b - a$.

Function **Hermpol** evaluates the Hermite interpolant at the points stored in the vector **xi**.

```
function yi = Hermpol(ga, gb, dga, dgb, a, b, xi)

% Two-point cubic Hermite interpolant. Points of interpolation
% are a and b. Values of the interpolant and its first order
% derivatives at a and b are equal to ga, gb, dga and dgb,
% respectively.
% Vector yi holds values of the interpolant at the points xi.

h = b – a;
t = (xi - a)./h;
t1 = 1 - t;
t2 = t1.*t1;
yi = (1 + 2*t).*t2*ga + (3 - 2*t).*(t.*t)*gb +…
h.*(t.*t2*dga + t.^2.**(t - 1)*dgb);
```

In this example we will interpolate function $g(x) = \sin(x)$ using a two-point cubic Hermite interpolant with $a = 0$ and $b = \pi/2$

```
xi = linspace(0, pi/2);

yi = Hermpol(0, 1, 1, 0, 0, pi/2, xi);

zi = yi – sin(xi);

plot(xi, zi), title('Error in interpolation of sin(x) by a two-point
cubic Hermite polynomial')
```



5.3.3 Interpolation by splines

In this section we will deal with interpolation by polynomial splines. In recent decades splines have attracted attention of both researchers and users who need a versatile approximation tools. We begin with the definition of the polynomial spline functions and the spline space.

Given an interval **[a, b]**. A *partition* **Δ** of the interval **[a, b]** with the *breakpoints* $\{x_l\}_1^m$ is defined as $\Delta = \{a = x_1 < x_2 < \ldots < x_m = b\}$, where $m > 1$. Further, let **k** and **n**, $k < n$, be nonnegative integers. Function **s(x)** is said to be a *spline function of degree* **n** *with smoothness* **k** if the following conditions are satisfied:

(i) On each subinterval $[x_l, x_{l+1}]$ **s(x)** coincides with an algebraic polynomial of degree at most **n**.
(ii) **s(x)** and its derivatives up to order **k** are all continuous on the interval **[a, b]**

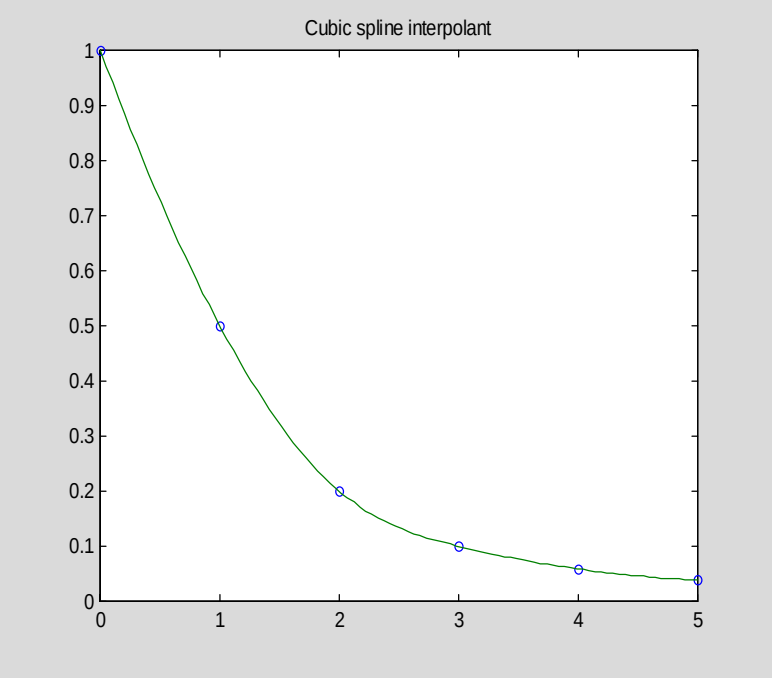
Throughout the sequel the symbol **Sp(n, k, Δ)** will stand for the *space of the polynomial splines* of degree **n** with smoothness **k** , and the breakpoints **Δ**. It is well known that Sp(n, k, Δ) is a linear subspace of dimension $(n + 1)(m - 1) - (k + 1)(m - 2)$. In the case when $k = n - 1$, we will write **Sp(n, Δ)** instead of **Sp(n, n – 1, Δ)**.

MATLAB function **spline** is designed for computations with the cubic splines ($n = 3$) that are twice continuously differentiable ($k = 2$) on the interval $[x_1, x_m]$. Clearly
$\dim Sp(3, \Delta) = m + 2$. The spline interpolant **s(x)** is determined uniquely by the interpolatory conditions $s(x_l) = y_l$, $l = 1, 2, \ldots, m$ and two additional boundary conditions, namely that $s'''(x)$ is continuous at $x = x_2$ and $x = x_{m-1}$. These conditions are commonly referred to as the *not-a-knot* end conditions.

MATLAB's command **yi = spline(x, y, xi)** evaluates cubic spline **s(x)** at points stored in the array **xi**. Vectors **x** and **y** hold coordinates of the points to be interpolated. To obtain the piecewise polynomial representation of the spline interpolant one can execute the command **pp = spline(x, y)**. Command **zi = ppval(pp, xi)** evaluates the piecewise polynomial form of the spline interpolant. Points of evaluation are stored in the array **xi**. If a spline interpolant has to be evaluated for several vectors **xi**, then the use of function **ppval** is strongly recommended.

In this example we will interpolate Runge's function **$g(x) = 1/(1 + x^2)$** on the interval **[0, 5]** using six evenly spaced breakpoints

```
x = 0:5;

y = 1./(1 + x.^2);

xi = linspace(0, 5);

yi = spline(x, y, xi);

plot(x, y, 'o', xi, yi), title('Cubic spline interpolant')
```



The maximum error on the set **xi** in approximating Runge's function by the cubic spline we found is

```
err = norm(abs(yi-1./(1+xi.^2)),'inf')

err =
    0.0859
```

Detailed information about the piecewise polynomial representation of the spline interpolant can be obtained running function **spline** with two input parameters **x** and **y**

```
pp = spline(x, y);
```

and next executing command **unmkpp**

```
[brpts, coeffs, npol, ncoeff] = unmkpp(pp)

brpts =
     0     1     2     3     4     5
coeffs =
    0.0074    0.0777   -0.5852    1.0000
    0.0074    0.1000   -0.4074    0.5000
   -0.0371    0.1223   -0.1852    0.2000
   -0.0002    0.0110   -0.0519    0.1000
   -0.0002    0.0104   -0.0306    0.0588
npol =
     5
ncoeff =
     4
```

The output parameters **brpts**, **coeffs**, **npol**, and **ncoeff** represent the breakpoints of the spline interpolant, coefficients of **s(x)** on successive subintervals, number of polynomial pieces that constitute spline function and number of coefficients that represent each polynomial piece, respectively. On the subinterval $[x_l, x_{l+1}]$ the spline interpolant is represented as

$$s(x) = c_{l1}(x - x_l)^3 + c_{l2}(x - x_l)^2 + c_{l3}(x - x_l) + c_{l4}$$

where $[c_{l1}\ c_{l2}\ c_{l3}\ c_{l4}]$ is the lth row of the matrix **coeffs**. This form is called the *piecewise polynomial form* (*pp–form*) of the spline function.

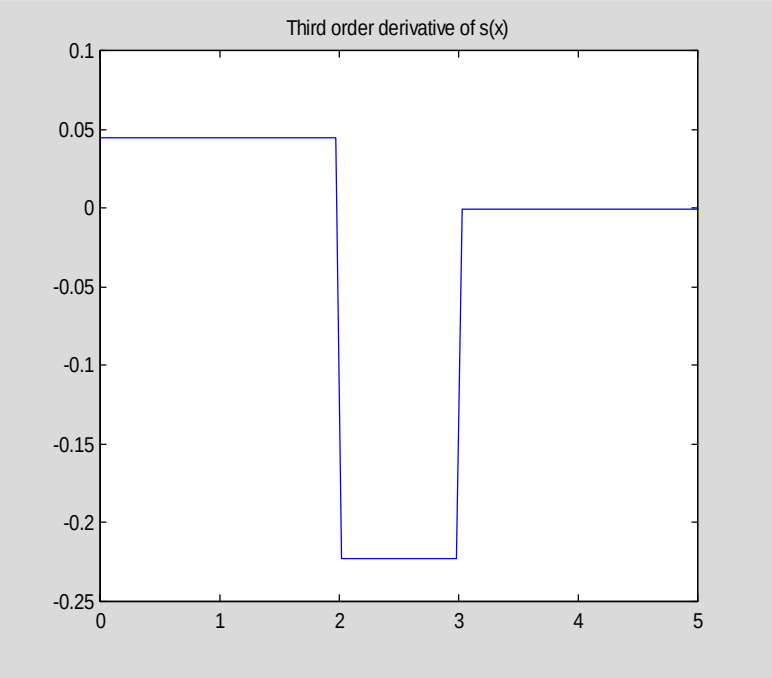
Differentiation of the spline function s**(x)** can be accomplished running function **splder**. In order for this function to work properly another function **pold** (see Problem 19) must be in MATLAB's search path.

```
function p = splder(k, pp, x)

% Piecewise polynomial representation of the derivative
% of order k (0 <= k <= 3) of a cubic spline function in the
% pp form with the breakpoints stored in the vector x.

m = pp(3);
lx4 = length(x) + 4;
n = pp(lx4);
c = pp(1 + lx4:length(pp))';
c = reshape(c, m, n);
b = pold(c, k);
b = b(:)';
```

```
p = pp(1:lx4);
p(lx4) = n - k;
p = [p b];
```

The third order derivative of the spline function of the last example is shown below

```
p = splder(3, pp, x);

yi = ppval(p, xi);

plot(xi, yi), title('Third order derivative of s(x)')
```



Note that **s'''(x)** is continuous at the breakpoints $x_2 = 1$ and $x_5 = 4$. This is due to the fact that the not-a-knot boundary conditions were imposed on the spline interpolant.

Function **evalppf** is the utility tool for evaluating the piecewise polynomial function **s(x)** at the points stored in the vector **xi**. The breakpoints $x = \{x_1 < x_2 < \ldots < x_m\}$ of **s(x)** and the points of evaluation **xi** must be such that $x_1 = xi_1$ and $x_m = xi_p$, where **p** is the index of the largest number in **xi**. Coefficients of the polynomial pieces of **s(x)** are stored in rows of the matrix **A** in the descending order of powers.

```
function [pts, yi] = evalppf(x, xi, A)

% Values yi of the piecewise polynomial function (pp-function)
% evaluated at the points xi. Vector x holds the breakpoints
% of the pp-function and matrix A holds the coefficients of the
% pp-function. They are stored in the consecutive rows in
```

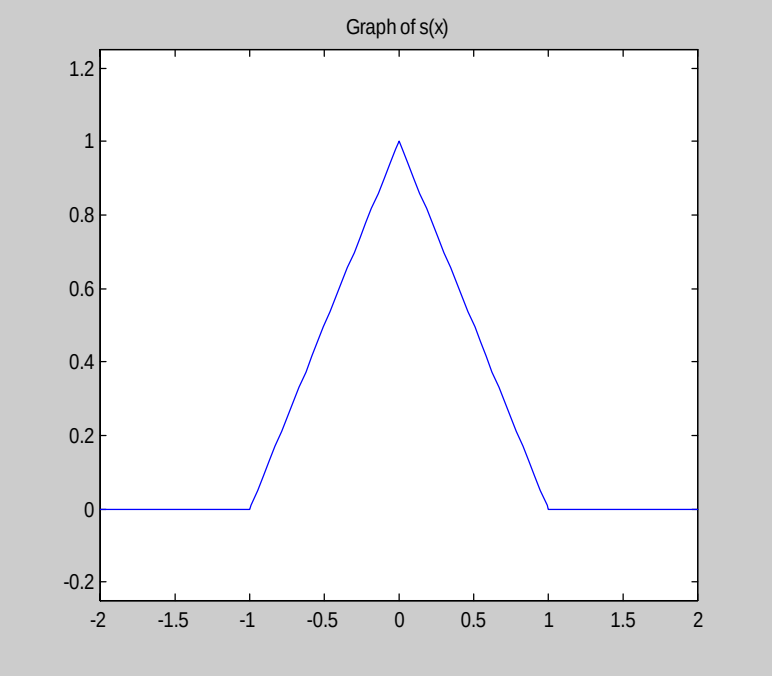
```
% the descending order of powers.The output parameter pts holds
% the points of the union of two sets x and xi.


n = length(x);
[p, q] = size(A);
if n-1 ~= p
   error('Vector t and matrix A must be "compatible"')
end
yi = [];
pts = union(x, xi);
for m=1:p
   l = find(pts == x(m));
   r = find(pts == x(m+1));
   if m < n-1
      yi = [yi polyval(A(m,:), pts(l:r-1))];
   else
      yi = [yi polyval(A(m,:), pts(l:r))];
   end
end
```

In this example we will evaluate and plot the graph of the piecewise linear function **s(x)** that is defined as follows

s(x) = 0, if **|x| ⩾ 1**
s(x) = 1 + x, if **-1 ⩽ x ⩽ 0**
s(x) = 1 – x, if **0 ⩽ x ⩽ 1**

Let

```
x = -2:2;

xi = linspace(-2, 2);

A = [0 0;1 1;1 –1;0 0];

[pts, yi] = evalppf(x, xi, A);

plot(pts, yi), title('Graph of s(x)'), axis([-2 2 -.25 1.25])
```



5.3.4 Two Dimensional Interpolation

The interpolation problem discussed in this section is formulated as follows.

Given a rectangular grid **$\{x_k, y_l\}$** and the associated set of numbers **z_{kl}**, **$1 \leqslant k \leqslant m$**, **$1 \leqslant l \leqslant n$**, find a bivariate function **$z = f(x, y)$** that interpolates the data, i.e., **$f(x_k, y_l) = z_{kl}$** for all values of **k** and **l**. The grid points must be sorted monotonically, i.e. **$x_1 < x_2 < \ldots < x_m$** with a similar ordering of the y-ordinates.

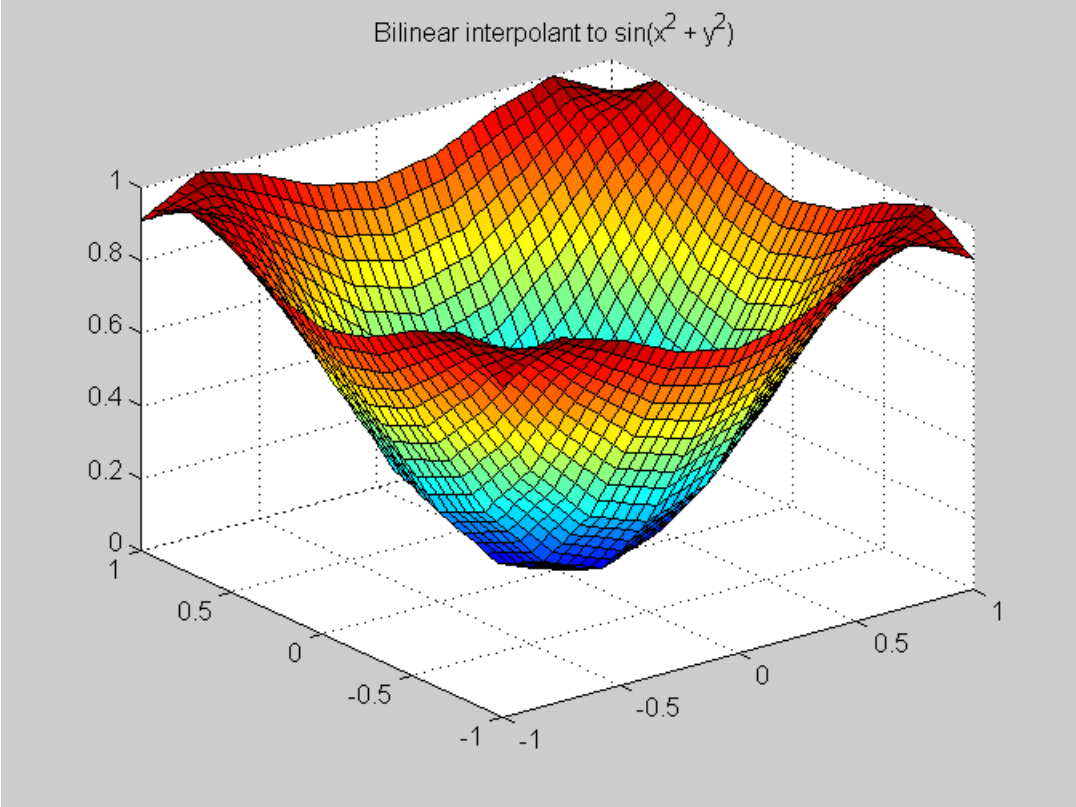
MATLAB's built-in function **zi = interp2(x, y, z, xi, yi, 'method')** generates a bivariate interpolant on the rectangular grids and evaluates it in the points specified in the arrays **xi** and **yi**. Sixth input parameter **'method'** is optional and specifies a method of interpolation. Available methods are:

- **'nearest'** - nearest neighbor interpolation
- **'linear'** - bilinear interpolation
- **'cubic'** - bicubic interpolation
- **'spline'** - spline interpolation

In the following example a bivariate function **$z = \sin(x^2 + y^2)$** is interpolated on the square **$-1 \leqslant x \leqslant 1$**, **$-1 \leqslant y \leqslant 1$** using the 'linear' and the 'cubic' methods.

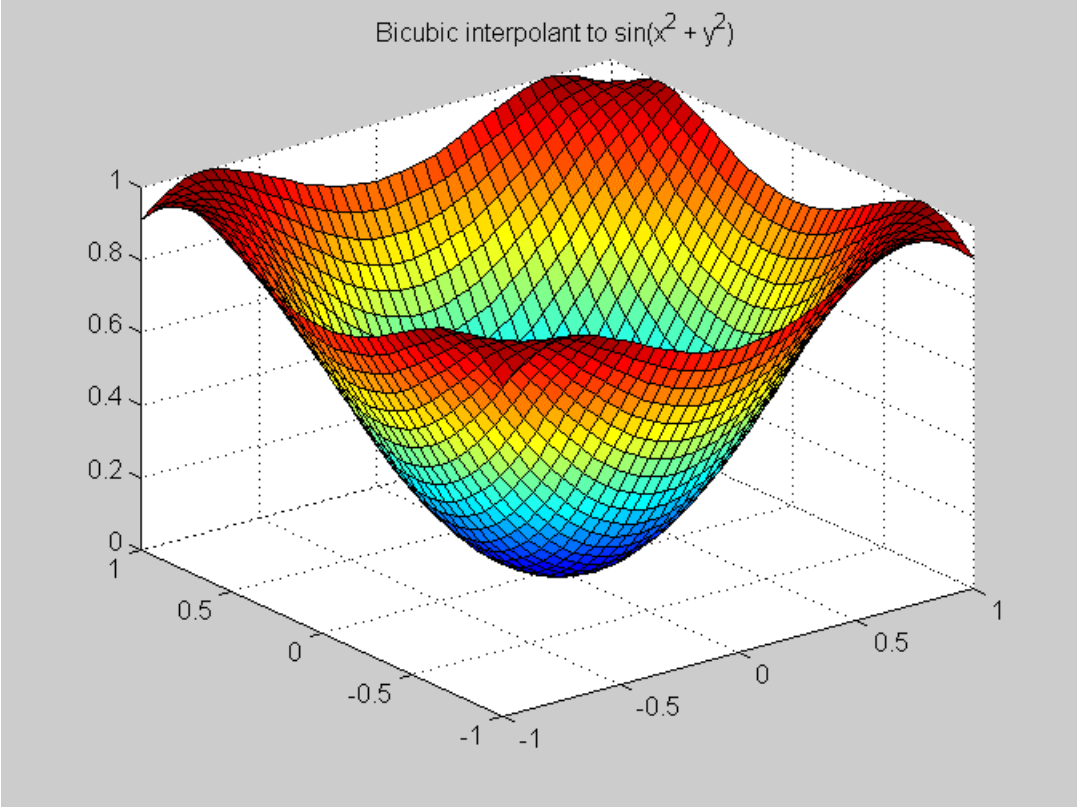
```
[x, y] = meshgrid(-1:.25:1);

z = sin(x.^2 + y.^2);

[xi, yi] = meshgrid(-1:.05:1);
```

```
zi = interp2(x, y, z, xi, yi, 'linear');
surf(xi, yi, zi), title('Bilinear interpolant to sin(x^2 + y^2)')
```



The bicubic interpolant is obtained in a similar fashion

```
zi = interp2(x, y, z, xi, yi, 'cubic');
```



5.4 Numerical Integration and Differentiation

A classical problem of the numerical integration is formulated as follows.

Given a continuous function **f(x)**, **a ⩽ x ⩽ b**, find the coefficients **{w_k}** and the nodes **{x_k}**, **1 ⩽ k ⩽ n**, so that the *quadrature formula*

(5.4.1) $$\int_a^b f(x)dx \approx \sum_{k=1}^{n} w_k f(x_k)$$

is exact for polynomials of a highest possible degree.

For the evenly spaced nodes **{x_k}** the resulting family of the quadrature formulas is called the *Newton-Cotes formulas*. If the coefficients **{w_k}** are assumed to be all equal, then the quadrature formulas are called the *Chebyshev quadrature formulas*. If both, the coefficients **{w_k}** and the nodes **{x_k}** are determined by requiring that the formula (5.4.1) is exact for polynomials of the highest possible degree, then the resulting formulas are called the *Gauss quadrature formulas*.

5.4.1 Numerical integration using MATLAB functions quad and quad8

Two MATLAB functions **quad('f ', a, b, tol, trace, p1, p2, …)** and **quad8('f ', a, b, tol, trace, p1, p2, …)** are designed for numerical integration of the univariate functions. The input parameter **'f'** is a string containing the name of the function to be integrated from **a** to **b**. The fourth input parameter **tol** is optional and specifies user's chosen relative error in the computed integral. Parameter **tol** can hold both the relative and the absolute errors supplied by the user. In this case a two-dimensional vector tol **= [rel_tol, abs_tol]** must be included. Parameter **trace** is optional and traces the function evaluations with a point plot of the integrand. To use default values for **tol** or **trace** one may pass in the empty matrix **[ ]**. Parameters **p1**, **p2**, … are also optional and they are supplied only if the integrand depends on **p1**, **p2**, … .

In this example a simple rational function

$$\mathbf{f(x)} = \frac{\mathbf{a + bx}}{\mathbf{1 + cx^2}}$$

```
function y = rfun(x, a, b, c)

% A simple rational function that depends on three
% parameters a, b and c.

y = (a + b.*x)./(1 + c.*x.^2);
y = y';
```

is integrated numerically from **0** to **1** using both functions **quad** and **quad8**. The assumed relative and absolute errors are stored in the vector **tol**

```
tol = [1e-5 1e-3];

format long

[q, nfev] = quad('rfun', 0, 1, tol, [], 1, 2, 1)

q =
   1.47856630183943
nfev =
     9
```

Using function **quad8** we obtain

```
[q8,nfev] = quad8('rfun', 0, 1, tol, [], 1, 2, 1)

q8 =
   1.47854534395683
nfev =
    33
```

Second output parameter **nfev** gives an information about the number of function evaluations needed in the course of computation of the integral.

The exact value of the integral in question is

```
exact = log(2) + pi/4
```

```
exact =
   1.47854534395739
```

The relative errors in the computed approximations **q** and **q8** are

```
rel_errors = [abs(q – exact)/exact; abs(q8 – exact)/exact]
```

```
rel_errors =
  1.0e-004 *
   0.14174663036002
   0.00000000380400
```

5.4.2 Newton – Cotes quadrature formulas

One of the oldest method for computing the approximate value of the definite integral over the interval **[a, b]** was proposed by Newton and Cotes. The nodes of the Newton – Cotes formulas are chosen to be evenly spaced in the interval of integration. There are two types of the Newton – Cotes formulas the closed and the open formulas. In the first case the endpoints of the interval of integration are included in the sets of nodes whereas in the open formulas they are not. The weights $\{w_k\}$ are determined by requiring that the quadrature formula is exact for polynomials of a highest possible degree.

Let us discuss briefly the Newton – Cotes formulas of the closed type. The nodes of the n – point formula are defined as follows $x_k = a + (k – 1)h$, $k = 1, 2, \dots, n$, where $h = (b – a)/(n – 1)$, $n > 1$. The weights of the quadrature formula are determined from the conditions that the following equations are satisfied for the monomials $f(x) = 1, x, \dots x^{n-1}$

$$\int_a^b f(x)dx = \sum_{k=1}^{n} w_k f(x_k)$$

```
function [s, w, x] = cNCqf(fun, a, b, n, varargin)

% Numerical approximation s of the definite integral of
% f(x). fun is a string containing the name of the integrand f(x).
% Integration is over the interval [a, b].
% Method used:
% n-point closed Newton-Cotes quadrature formula.
% The weights and the nodes of the quadrature formula
% are stored in vectors w and x, respectively.

if n < 2
   error(' Number of nodes must be greater than 1')
end
x = (0:n-1)/(n-1);
```

```
f = 1./(1:n);
V = Vander(x);
V = rot90(V);
w = V\f';
w = (b-a)*w;
x = a + (b-a)*x;
x = x';
s = feval(fun,x,varargin{:});
s = w'*s;
```

In this example the *error function* **Erf(x)** , where

$$\mathbf{Erf(x)} = \frac{2}{\sqrt{\pi}} \int_0^x \mathbf{e}^{-t^2}\,\mathbf{dt}$$

will be approximated at **x = 1** using the closed Newton – Cotes quadrature formulas wit **n = 2** (Trapezoidal Rule), **n = 3** (Simpson's Rule), and **n = 4** (Boole's Rule). The integrand of the last integral is evaluated using function **exp2**

```
function w = exp2(x)

% The weight function w of the Gauss-Hermite quadrarure formula.

w = exp(-x.^2);
```

```
approx_v = [];

for n =2:4
   approx_v = [approx_v; (2/sqrt(pi))*cNCqf('exp2', 0, 1, n)];
end
```

```
approx_v

approx_v =
   0.77174333225805
   0.84310283004298
   0.84289057143172
```

For comparison, using MATLAB's built - in function **erf** we obtain the following approximate value of the error function at **x = 1**

```
exact_v = erf(1)

exact_v =
   0.84270079294971
```

5.4.3 Gauss quadature formulas

This class of numerical integration formulas is constructed by requiring that the formulas are exact for polynomials of the highest possible degree. The Gauss formulas are of the type

$$\int_a^b p(x)f(x)dx \approx \sum_{k=1}^{n} w_k f(x_k)$$

where p(x) denotes the *weight function*. Typical choices of the weight functions together with the associated intervals of integration are listed below

Weight p(x)	Interval [a, b]	Quadrature name
1	**[-1, 1]**	Gauss-Legendre
$1/\sqrt{1-x^2}$	**[-1, 1]**	Gauss-Chebyshev
e^{-x}	**$[0,\infty)$**	Gauss-Laguerre
e^{-x^2}	**$(-\infty,\infty)$**	Gauss-Hermite

It is well known that the weights of the Gauss formulas are all positive and the nodes are the roots of the class of polynomials that are orthogonal, with respect to the given weight function **p(x)**, on the associated interval.

Two functions included below**, Gquad1** and **Gquad2** are designed for numerical computation of the definite integrals using Gauss quadrature formulas. A method used here is described in [3], pp. 93 – 94.

```
function [s, w, x] = Gquad1(fun, a, b, n, type, varargin)

% Numerical integration using either the Gauss-Legendre (type = 'L')
% or the Gauss-Chebyshev (type = 'C') quadrature with n (n > 0) nodes.
% fun is a string representing the name of the function that is
% integrated from a to b. For the Gauss - Chebyshev quadrature
% it is assumed that a = -1 and b = 1.
% The output parameters s, w, and x hold the computed approximation
% of the integral, list of weights, and the list of nodes,
% respectively.

d = zeros(1,n-1);
if type == 'L'
   k = 1:n-1;
   d = k./(2*k - 1).*sqrt((2*k - 1)./(2*k + 1));
   fc = 2;
   J = diag(d,-1) + diag(d,1);
   [u,v] = eig(J);
   [x,j] = sort(diag(v));
   w = (fc*u(1,:).^2)';
   w = w(j)';
```

```
   w = 0.5*(b - a)*w;
   x = 0.5*((b - a)*x + a + b);
else
   x = cos((2*(1:n) - (2*n + 1))*pi/(2*n))';
   w(1:n) = pi/n;
end
f = feval(fun,x,varargin{:});
s = w*f(:);
w = w';
```

In this example we will approximate the error function **Erf(1)** using Gauss-Legendre formulas with **n = 2, 3, … , 8**.

```
approx_v = [];

for n=2:8
   approx_v = [approx_v; (2/sqrt(pi))*Gquad1('exp2', 0, 1, n, 'L')];
end

approx_v
```

```
approx_v =
   0.84244189252255
   0.84269001848451
   0.84270117131620
   0.84270078612733
   0.84270079303742
   0.84270079294882
   0.84270079294972
```

Recall that using MATLAB's function **erf** we have already found that

```
exact_v = erf(1)
```

```
exact_v =
   0.84270079294971
```

If the interval of integration is either semi-infinite or bi-infinite then one may use function **Gquad2**. Details of a method used in this function are discussed in [3], pp. 93 – 94.

```
function [s, w, x] = Gquad2(fun, n, type, varargin)

% Numerical integration using either the Gauss-Laguerre
% (type = 'L') or the Gauss-Hermite (type = 'H') with n (n > 0) nodes.
% fun is a string containing the name of the function that is
% integrated.
% The output parameters s, w, and x hold the computed approximation
% of the integral, list of weights, and the list of nodes,
% respectively.

if type == 'L'
   d = -(1:n-1);
```

```
   f = 1:2:2*n-1;
   fc = 1;
else
   d = sqrt(.5*(1:n-1));
   f = zeros(1,n);
   fc = sqrt(pi);
end
J = diag(d,-1) + diag (f) + diag(d,1);
[u,v] = eig(J);
[x,j] = sort(diag(v));
w = (fc*u(1,:).^2)';
w = w(j);
f = feval(fun,x,varargin{:});
s = w'*f(:);
```

The Euler's gamma function

$$\Gamma(t) = \int_0^\infty e^{-x} x^{t-1} dx \qquad (t > -1)$$

can be approximated using function **Gquad2** with type being set to **'L'** (Gauss-Laguerre quadratures). Let us recall that $\Gamma(n) = (n-1)!$ for $n = 1, 2, \ldots$. Function **mygamma** is designed for computing numerical approximation of the gamma function using Gauss-Laguerre quadratures.

```
function y = mygamma(t)

% Value(s) y of the Euler's gamma function evaluated at t (t > -1).

td = t - fix(t);
if td == 0
   n = ceil(t/2);
else
   n = ceil(abs(t)) + 10;
end
y = Gquad2('pow',n,'L',t-1);
```

The following function

```
function z = pow(x, e)

% Power function z = x^e

z = x.^e;
```

is called from within function **mygamma**.

In this example we will approximate the gamma function for **t = 1, 1.1, … , 2** and compare the results with those obtained by using MATLAB's function **gamma**. A script file **testmyg** computes approximate values of the gamma function using two functions **mygamma** and **gamma**

```
% Script testmyg.m

format long
disp('        t               mygamma              gamma')
disp(sprintf('\n
_____________________________________________________________'))

for t=1:.1:2
   s1 = mygamma(t);
   s2 = gamma(t);
   disp(sprintf('%1.14f   %1.14f   %1.14f',t,s1,s2))
end


testmyg

        t               mygamma              gamma

 _____________________________________________________________
1.00000000000000   1.00000000000000   1.00000000000000
1.10000000000000   0.95470549811706   0.95135076986687
1.20000000000000   0.92244757458893   0.91816874239976
1.30000000000000   0.90150911731168   0.89747069630628
1.40000000000000   0.89058495940663   0.88726381750308
1.50000000000000   0.88871435840715   0.88622692545276
1.60000000000000   0.89522845323377   0.89351534928769
1.70000000000000   0.90971011289336   0.90863873285329
1.80000000000000   0.93196414951082   0.93138377098024
1.90000000000000   0.96199632935381   0.96176583190739
2.00000000000000   1.00000000000000   1.00000000000000
```

5.4.4 Romberg's method

Two functions, namely **quad** and **qauad8**, discussed earlier in this tutorial are based on the adaptive methods. Romberg (see, e.g., [2]), proposed another method, which does not belong to this class of methods. This method is the two-phase method. Phase one generates a sequence of approximations using the *composite trapezoidal rule*. Phase two improves approximations found in phase one using *Richardson's extrapolation*. This process is a recursive one and the number of performed iterations depends on the value of the integral parameter **n**. In many cases a modest value for **n** suffices to obtain a satisfactory approximation.

Function **Romberg(fun, a, b, n, varargin)** implements Romberg's algorithm

```
function [rn, r1] = Romberg(fun, a, b, n, varargin)

% Numerical approximation rn of the definite integral from a to b
% that is obtained with the aid of Romberg's method with n rows
% and n columns. fun is a string that names the integrand.
% If integrand depends on parameters, say p1, p2, ... , then
```

```
% they should be supplied just after the parameter n.
% Second output parameter r1 holds approximate values of the
% computed integral obtained with the aid of the composite
% trapezoidal rule using 1, 2, ... ,n subintervals.

h = b - a;
d = 1;
r = zeros(n,1);
r(1) = .5*h*sum(feval(fun,[a b],varargin{:}));
for i=2:n
   h = .5*h;
   d = 2*d;
   t = a + h*(1:2:d);
   s = feval(fun, t, varargin{:});
   r(i) = .5*r(i-1) + h*sum(s);
end
r1 = r;
d = 4;
for j=2:n
   s = zeros(n-j+1,1);
   s = r(j:n) + diff(r(j-1:n))/(d - 1);
   r(j:n) = s;
   d = 4*d;
end
rn = r(n);
```

We will test function **Romberg** integrating the rational function introduced earlier in this tutorial (see the m-file **rfun**). The interval of integration is **[a, b] = [0, 1]**, **n= 10**, and the values of the parameters **a**, **b**, and **c** are set to **1**, **2**, and **1**, respectively.

```
[rn, r1] = Romberg('rfun', 0 , 1, 10, 1, 2, 1)

rn =
   1.47854534395739
r1 =
   1.25000000000000
   1.42500000000000
   1.46544117647059
   1.47528502049722
   1.47773122353730
   1.47834187356141
   1.47849448008531
   1.47853262822223
   1.47854216503816
   1.47854454922849
```

The absolute and relative errors in **rn** are

```
[abs(exact - rn); abs(rn - exact)/exact]

ans =
     0
```

5.4.4 Numerical integration of the bivariate functions using MATLAB function dblquad

Function **dblquad** computes a numerical approximation of the double integral

$$\iint_D f(x,y)dxdy$$

where **D = {(x, y): a ⩽ x ⩽ b, c ⩽ y ⩽ d}** is the domain of integration. Syntax of the function **dblquad** is **dblquad (fun, a, b, c, d, tol)**, where the parameter **tol** has the same meaning as in the function **quad**.

Let **f(x, y) = e^{-xy} sin(xy)** , **-1 ⩽ x ⩽ 1, 0 ⩽ y ⩽ 1**. The m-file **esin** is used to evaluate function **f**

```
function z = esin(x,y);

z = exp(-x*y).*sin(x*y);
```

Integrating function **f** , with the aid of the function **dblquad**, over the indicated domain we obtain

```
result = dblquad('esin', -1, 1, 0, 1)

result =
  -0.22176646183245
```

5.4.5 Numerical differentiation

Problem discussed in this section is formulated as follows. Given a univariate function **f(x)** find an approximate value of **f '(x)**. The algorithm presented below computes a sequence of the approximate values to derivative in question using the following finite difference approximation of **f '(x)**

$$f'(x) \approx \frac{f(x+h)-f(x-h)}{2h}$$

where **h** is the initial stepsize. Phase one of this method computes a sequence of approximations to f'(x) using several values of **h**. When the next approximation is sought the previous value of **h** is halved. Phase two utilizes Richardson's extrapolation. For more details the reader is referred to [2], pp. 171 – 180.

Function **numder** implements the method introduced in this section.

```
function der = numder(fun, x, h, n, varargin)

% Approximation der of the first order derivative, at the point x,
% of a function named by the string fun. Parameters h and n
% are user supplied values of the initial stepsize and the number
% of performed iterations in the Richardson extrapolation.
% For fuctions that depend on parameters their values must follow
% the parameter n.

d = [];
for i=1:n
   s = (feval(fun,x+h,varargin{:})-feval(fun,x-h,varargin{:}))/(2*h);
   d = [d;s];
   h = .5*h;
end
l = 4;
for j=2:n
   s = zeros(n-j+1,1);
   s = d(j:n) + diff(d(j-1:n))/(l - 1);
   d(j:n) = s;
   l = 4*l;
end
der = d(n);
```

In this example numerical approximations of the first order derivative of the function $\mathbf{f(x) = e^{-x^2}}$ are computed using function **numder** and they are compared against the exact values of **f '(x)** at **x = 0.1, 0.2, … , 1.0**. The values of the input parameters **h** and **n** are **0.01** and **10**, respectively.

```
function testnder(h, n)

% Test file for the function numder. The initial stepsize is h and
% the number of iterations is n. Function to be tested is
% f(x) = exp(-x^2).

format long
disp('        x               numder               exact')
disp(sprintf('\n
_____________________________________________________________'))

for x=.1:.1:1
   s1 = numder('exp2', x, h, n);
   s2 = derexp2(x);
   disp(sprintf('%1.14f   %1.14f   %1.14f',x,s1,s2))
end



function y = derexp2(x)

% First order derivative of f(x) = exp(-x^2).

y = -2*x.*exp(-x.^2);
```

The following results are obtained with the aid of function **testndr**

```
testnder(0.01, 10)
```

```
        x                numder                exact

  ____________________________________________________________

0.10000000000000   -0.19800996675001   -0.19800996674983
0.20000000000000   -0.38431577566308   -0.38431577566093
0.30000000000000   -0.54835871116311   -0.54835871116274
0.40000000000000   -0.68171503117430   -0.68171503117297
0.50000000000000   -0.77880078306967   -0.77880078307140
0.60000000000000   -0.83721159128436   -0.83721159128524
0.70000000000000   -0.85767695185699   -0.85767695185818
0.80000000000000   -0.84366787846708   -0.84366787846888
0.90000000000000   -0.80074451919839   -0.80074451920129
```

5.5 Numerical Methods for the Ordinary Differential Equations

Many problems that arise in science and engineering require a knowledge of a function **y = y(t)** that satisfies the *first order differential equation* **y' = f(t, y)** and the *initial condition* **y(a) = y_0**, where **a** and **y_0** are given real numbers and **f** is a bivariate function that satisfies certain smoothness conditions. A more general problem is formulated as follows. Given function **f** of **n** variables, find a function **y = y(t)** that satisfies the *nth order* ordinary differential equation **$y^{(n)}$ = f(t, y, y', … , $y^{(n-1)}$)** together with the initial conditions **y(a) = y_0**, **y'(a) = y_0'**, … , **$y^{(n-1)}$ (a) = $y_0^{(n-1)}$**. The latter problem is often transformed into the problem of solving a system of the first order differential equations. To this end a term "ordinary differential equations" will be abbreviated as ODEs.

5.5.1 Solving the initial value problems using MATLAB built-in functions

MATLAB has several functions for computing a numerical solution of the initial value problems for the ODEs. They are listed in the following table

Function	Application	Method used
ode23	Nonstiff ODEs	Explicit Runge-Kutta (2, 3) formula
ode45	Nonstiff ODEs	Explicit Runge-Kutta (4, 5) formula
ode113	Nonstiff ODEs	Adams-Bashforth-Moulton solver
ode15s	Stiff ODEs	Solver based on the numerical differentiation formulas
ode23s	Stiff ODEs	Solver based on a modified Rosenbrock formula of order 2

A simplest form of the syntax for the MATLAB ODE solvers is

[t, y] = *solver*(fun, tspan, y0], where **fun** is a string containing name of the ODE m-file that describes the differential equation, **tspan** is the interval of integration, and **y0** is the vector holding the initial value(s). If **tspan** has more than two elements, then solver returns computed values of **y** at these points. The output parameters **t** and **y** are the vectors holding the points of evaluation and the computed values of **y** at these points.

In the following example we will seek a numerical solution **y** at **t = 0, .25, .5, .75, 1** to the following initial value problem $y' = -2ty^2$, with the initial condition **y(0) = 1**. We will use both the **ode23** and the **ode45** solvers. The exact solution to this problem is $y(t) = 1/(1 + t^2)$ (see, e.g., [6], p.289). The ODE m-file needed in these computations is named **eq1**

```
function dy = eq1(t,y)

% The m-file for the ODE y' = -2ty^2.

dy = -2*t.*y(1).^2;
```

```
format long

tspan = [0 .25 .5 .75 1]; y0 = 1;

[t1 y1] = ode23('eq1', tspan, y0);
[t2 y2] = ode45('eq1', tspan, y0);
```

To compare obtained results let us create a three-column table holding the points of evaluation and the y-values obtained with the aid of the **ode23** and the **ode45** solvers

```
[t1 y1 y2]

ans =
                   0   1.00000000000000   1.00000000000000
   0.25000000000000   0.94118221525751   0.94117646765650
   0.50000000000000   0.80002280597122   0.79999999678380
   0.75000000000000   0.64001788410487   0.63999998775736
   1.00000000000000   0.49999658522366   0.50000000471194
```

Next example deals with the system of the *first order* ODEs

$$y_1'(t) = y_1(t) - 4y_2(t),\ y_2'(t) = -y_1(t) + y_2(t),$$
$$y_1(0) = 1;\ y_2(0) = 0.$$

Instead of writing the ODE m – file for this system, we will use MATLAB **inline** function

```
dy = inline('[1 -4;-1 1]*y', 't', 'y')

dy =
     Inline function:
     dy(t,y) = [1 -4;-1 1]*y
```

The inline functions are created in the Command Window. Interval over wich numerical solution is computed and the initial values are stored in the vectors **tspan** and **y0**, respectively

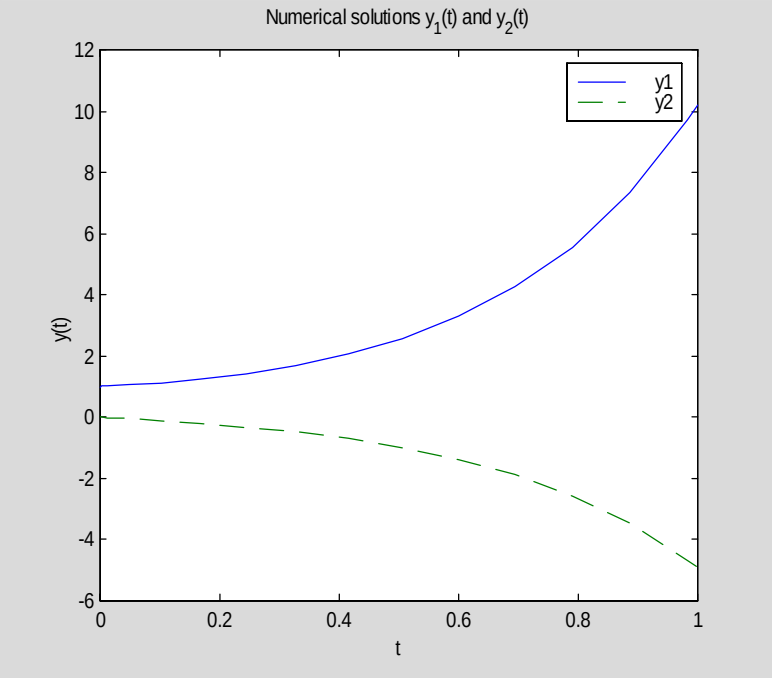
```
tspan = [0 1];  y0 = [1 0];
```

Numerical solution to this system is obtained using the **ode23** function

```
[t,y] = ode23(dy, tspan, y0);
```

Graphs of $y_1(t)$ (solid line) and $y_2(t)$ (dashed line) are shown below

```
plot(t,y(:,1),t,y(:,2),'--'), legend('y1','y2'), xlabel('t'),
ylabel('y(t)'), title('Numerical solutions y_1(t) and y_2(t)')
```



The exact solution $(y_1(t), y_2(t))$ to this system is

```
y1, y2

y1 =
1/2*exp(-t)+1/2*exp(3*t)
y2 =
-1/4*exp(3*t)+1/4*exp(-t)
```

Functions **y1** and **y2** were found using command **dsolve** which is available in the **Symbolic Math Toolbox**.

Last example in this section deals with the *stiff ODE*. Consider

$$y'(t) = -1000(y - \log(1 + t)) + \frac{1}{1+t},$$
$$y(0) = 1.$$

```
dy = inline('-1000*(y - log(1 + t)) + 1/(1 + t)', 't', 'y')

dy =
     Inline function:
     dy(t,y) = -1000*(y - log(1 + t)) + 1/(1 + t)
```

Using the **ode23s** function on the interval

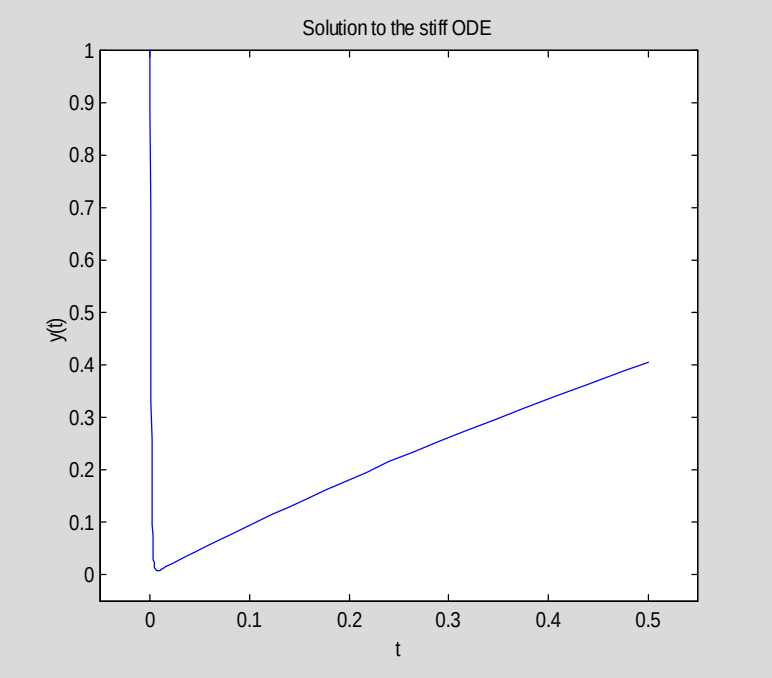
```
tspan = [0 0.5];
```

we obtain

```
[t, y] = ode23s(dy, tspan, 1);
```

To illustrate the effect of stiffness of the differential equation in question, let us plot the graph of the computed solution

```
plot(t, y), axis([-.05 .55 -.05 1] ), xlabel('t'), ylabel('y(t)'),
title('Solution to the stiff ODE')
```



The exact solution to this problem is **y(t) = log(1+t) + exp(-1000*t)**. Try to plot this function on the interval **[-0.05, 0.5]**.

5.5.2 The two – point boundary value problem for the second order ODE's

The purpose of this section is to discuss a numerical method for the two – point boundary value problem for the second order ODE

$$y''(t) = f(t, y, y')$$
$$y(a) = ya,\ \ y(b) = yb.$$

A method in question is the *finite difference method*. Let us assume that the function **f** is of the form $f(t, y, y') = g_0(t) + g_1(t)y + g_2(t)y'$. Thus the function **f** is linear in both **y** and **y'**. Using standard second order approximations for **y'** and y'' one can easily construct a linear system of equations for computing approximate values of the function y on the set of evenly spaced points. Function **bvp2ode** implements this method

```
function [t, y] = bvp2ode(g0, g1, g2, tspan, bc, n)

% Numerical solution y of the boundary value problem
% y'' = g0(t) + g1(t)*y + g2(t)*y', y(a) = ya, y(b) = yb,
% at n+2 evenly spaced points t in the interval tspan = [a b].
% g0, g1, and g2 are strings representing functions g0(t),
% g1(t), and g2(t), respectively. The boundary values
% ya and yb are stored in the vector bc = [ya yb].

a = tspan(1);
b = tspan(2);
t = linspace(a,b,n+2);
t1 = t(2:n+1);
u = feval(g0, t1);
v = feval(g1, t1);
w = feval(g2, t1);
h = (b-a)/(n+1);
d1 = 1+.5*h*w(1:n-1);
d2 = -(2+v(1:n)*h^2);
d3 = 1-.5*h*w(2:n);
A = diag(d1,-1) + diag(d2) + diag(d3,1);
f = zeros(n,1);
f(1) = h^2*u(1) - (1+.5*h*w(1))*bc(1);
f(n) = h^2*u(n) - (1-.5*h*w(n))*bc(2);
f(2:n-1) = h^2*u(2:n-1)';
s = A\f;
y = [bc(1);s;bc(2)];
t = t';
```

In this example we will deal with the two-point boundary value problem

$$y''(t) = 1 + \sin(t)y + \cos(t)y'$$
$$y(0) = y(1) = 1.$$

We define three inline functions

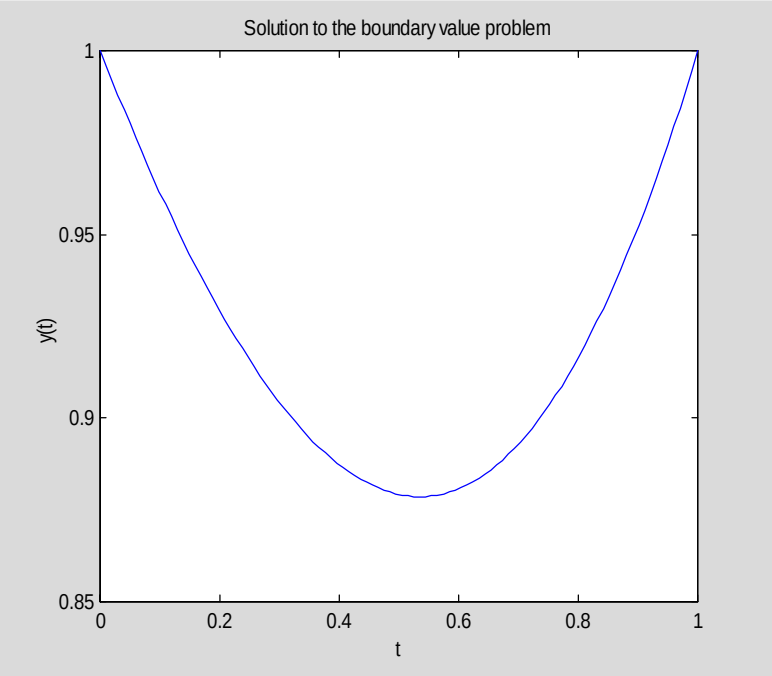
```
g0 = inline('ones(1, length(t))', 't'), g1 = inline('sin(t)', 't'), g2
= inline('cos(t)', 't')

g0 =
     Inline function:
     g0(t) = ones(1, length(t))
g1 =
     Inline function:
     g1(t) = sin(t)
g2 =
     Inline function:
     g2(t) = cos(t)
```

and next run function **bvp2ode** to obtain

```
[t, y] = bvp2ode(g0, g1, g2, [0 1],[1 1],100);
```

Graph of a function generated by **bvp2ode** is shown below

```
plot(t, y), axis([0 1 0.85 1]), title('Solution to the boundary value
problem'), xlabel('t'), ylabel('y(t)')
```



References

[1] B.C. Carlson, Special Functions of Applied Mathematics, Academic Press, New York, 1977.

[2] W. Cheney and D. Kincaid, Numerical Mathematics and Computing, Fourth edition, Brooks/Cole Publishing Company, Pacific Grove, 1999.

[3] P.J. Davis and P. Rabinowitz, Methods of Numerical Integration, Academic Press, New York, 1975.

[4] L.V. Fausett, Applied Numerical Analysis Using MATLAB, Prentice Hall, Upper Saddle River, NJ, 1999.

[4] D. Hanselman and B. Littlefield, Mastering MATLAB 5. A Comprehensive Tutorial and Reference, Prentice Hall, Upper Saddle River, NJ, 1998.

[6] M.T. Heath, Scientific Computing: An Introductory Survey, McGraw-Hill, Boston, MA, 1997.

[7] N.J. Higham, Accuracy and Stability of Numerical Algorithms, SIAM, Philadelphia, PA, 1996.

[8] G. Lindfield and J. Penny, Numerical Methods Using MATLAB, Ellis Horwood, New York, 1995.

[9] J.H. Mathews and K.D. Fink, Numerical Methods Using MATLAB, Third edition, Prentice Hall, Upper Saddle River, NJ, 1999.

[10] MATLAB, The Language of Technical Computing. Using MATLAB, Version 5, The MathWorks, Inc., 1997.

[11] J.C. Polking, Ordinary Differential Equations using MATLAB, Prentice Hall, Upper Saddle River, NJ, 1995.

[12] Ch.F. Van Loan, Introduction to Scientific Computing. A Matrix-Vector Approach Using MATLAB, Prentice Hall, Upper Saddle River, NJ, 1997.

[13] H.B. Wilson and L.H. Turcotte, Advanced Mathematics and Mechanics Applications Using MATLAB, Second edition, CRC Press, Boca Raton, 1997.

Problems

1. Give an example of a polynomial of degree $\mathbf{n \geqslant 3}$ with real roots only for which function **roots** fails to compute a correct type of its roots.

2. All roots of the polynomial $\mathbf{p(x) = a_0 + a_1 x + \ldots + a_{n-1} x^{n-1} + x^n}$, with real coefficients $\mathbf{a_k\ (k = 0, 1, \ldots n - 1)}$, are the eigenvalues of the *companion matrix*

$$\mathbf{A} = \begin{bmatrix} \mathbf{0} & \mathbf{1} & \mathbf{\ldots} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{\ldots} & \mathbf{0} \\ \mathbf{.} & \mathbf{.} & \mathbf{.} & \mathbf{.} & \mathbf{.} \\ \mathbf{-a_0} & \mathbf{-a_1} & \mathbf{-a_2} & \mathbf{\ldots} & \mathbf{-a_{n-1}} \end{bmatrix}$$

   Write MATLAB function **r = polroots(a)** that takes a one-dimensional array **a** of the coefficients of the polynomial **p(x)** in the descending order of powers and returns its roots in the array **r**.
   Organize your work as follows:

   (i) Create a matrix **A**. You may wish to use MATLAB's built-in function **diag** to avoid using loops. Function **diag** takes a second argument that can be used to put a superdiagonal in the desired position.
   (ii) Use MATLAB's function **eig** to compute all eigenvalues of the companion matrix **A**. See Tutorial 4 for more details about the matrix eigenvalue problem.

3. Write MATLAB function **[r, niter] = fpiter(g, x0, maxiter)** that computes a zero **r** of **x = g(x)** using the fixed-point iteration $\mathbf{x_{n+1} = g(x_n)}$, **n = 0, 1, …** with a given initial approximation **x0** of **r**. The input parameter **maxiter** is the maximum number of allowed iterations while the output parameter **niter** stands for the number of iterations performed. Use an appropriate stopping criterion to interrupt computations when current approximation satisfies the exit condition of your choice.

4. In this exercise you are to test function **fpiter** of Problem 3.
   Recall that a convergent sequence $\mathbf{\{x^{(k)}\}}$, with the limit **r**, has the *order of convergence* μ if

$$|\mathbf{x^{(k+1)} - r}| \leqslant \mathbf{C}|\mathbf{x^{(k)} - r}|^{\mu}, \quad \text{for some } \mathbf{C > 0}.$$

   If $\mu = \mathbf{1}$, then $\mathbf{C < 1}$.

   (i) Construct at least one equation of the form **x = g(x)**, with at least one real zero, for which function **fpiter** computes a sequence of approximations $\mathbf{\{x_n\}}$ that converges to the zero of your function. Print out consecutive approximations of the zero **r** and determine the order of convergence.
   (ii) Repeat previous part where this time a sequence of approximations generated by the function **fpiter** does not converge to the zero **r**. Explain why a computed sequence diverges.

5. Derive Newton's iteration for a problem of computing the reciprocal of a nonzero number **a**.
   (i) Does your iteration always converge for any value of the initial guess x_0?
   (ii) Write MATLAB function **r = recp(a, x0)** that computes the reciprocal of **a** using Newton's method with the initial guess **x0**.
   (iii) Run function **recp** for the following following values of **(a, x_0)** : (2, 0.3) and (10, 0.15) and print out consecutive approximations generated by the function **recp** and determine the order of convergence.

6. In this exercise you are to write MATLAB function **[r, niter] = Sch(f, derf, x0, m, tol)** to compute a multiple root **r** of the function **f(x)**. Recall that **r** is a root of multiplicity **m** of **f(x)** if $f(x) = (x - r)^m g(x)$, for some function **g(x)**. Schroder (see [8]) has proposed the following iterative scheme for computing a multiple root **r** of **f(x)**

$$x_{k+1} = x_k - mf(x_k)/f'(x_k), \quad k = 0, 1, \ldots .$$

   When **m = 1**, this method becomes the Newton – Raphson method.
   The input parameters: **f** is the function with a multiple root **r**, **derf** is the first derivative of **f**, **x0** is the initial guess, **m** stands for the multiplicity of **r** and **tol** is the assumed tolerance for the computed root.
   The output parameters: **r** is the computed root and **niter** is the number of performed iterations.

7. In this exercise you are to test function **Sch** of Problem 6.

   (i) Use function **f2** defined in Section 5.2 and write function **derf2** to compute the first order derivative of a function in file **f2**.
   (ii) Use unexpanded form for the derivative. Run function **Sch** with **m = 5** then repeat this experiment letting **m = 1**. In each case choose $x_0 = 0$. Compare number of iterations performed in each case.
   (iii) Repeat the above experiment using function **f3**. You will need a function **derf3** to evaluate the first derivative in the expanded form.

8. Let **p(x)** be a cubic polynomial with three distinct real roots r_k , **k = 1, 2, 3**. Suppose that the exact values of r_1 and r_2 are available. To compute the root r_3 one wants to use function **Sch** of Problem 6 with **m = 1** and $x_0 = (r_1 + r_2)/2$. How many iterations are needed to compute r_3?

9. Based on your observations made during the numerical experiments performed when solving Problem 8 prove that only one step of the Newton-Raphson method is needed to compute the third root of **p(x)**.

10. Given a system of nonlinear equations

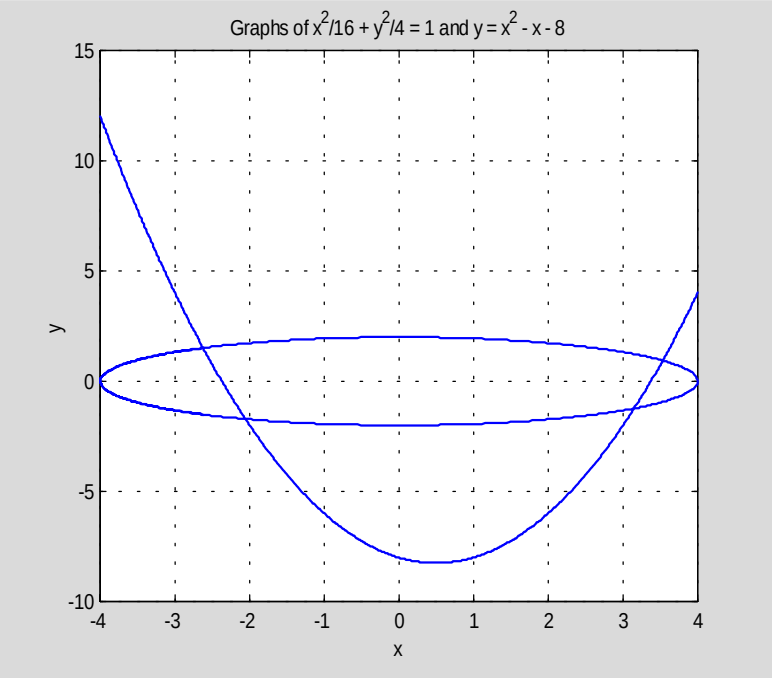
$$x^2/16 + y^2/4 = 1$$
$$x^2 - y^2 = 1$$

    Use function **NR** to compute all the zeros of this system. Compare your results with the exact values **x = ±2** and **y = ±√3**. Evaluate function **f** at the computed zeros and print your results using **format long**.

11. Using function **NR** find all the zeros of the system of nonlinear equations

$$x^2/16 + y^2/4 = 1$$
$$x^2 - x - y - 8 = 0$$

The following graph should help you to choose the initial approximations to the zeros of this system



Evaluate function **f** at the computed zeros and print out your results using **format long**.

12. Another method for computing zeros of the scalar equation **f(x) = 0** is the *secant method*. Given two initial approximations x_0 and x_1 of the zero **r** this method generates a sequence $\{x_k\}$ using the iterative scheme

$$x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}, \quad k = 1, 2, \ldots .$$

Write MATLAB function **[r, niter] = secm(f, x0, x1, tol, maxiter)** that computes the zero **r** of **f(x) = 0**. The input parameters: **f** is the name of a function whose zero is computed, **x0** and **x1** are the initial approximations of **r**, **tol** is the prescribed tolerance and **maxiter** is the maximum number of the allowed iterations. The output parameters: **r** is the computed zero of **f(x)** and **niter** is the number of the performed iterations.

13. Use the function **secm** of Problem 12 to find the smallest positive zero of **f(x)**.

(i) $f(x) = \sin(\tan(x)) - x$
(ii) $f(x) = \sin(x) + 1/(1 + e^{-x}) - 1$
(iii) $f(x) = \cos(x) - e^{-\sin(x)}$

Evaluate each function **f** at the computed zero and print out results using **format long**.

14. Another form of the interpolating polynomial is due to Lagrange and uses as the basis function the so–called *fundamental polynomials* $L_k(x)$, $0 \leq k \leq n$. The kth fundamental polynomial L_k is defined as follows: $L_k(x_k) = 1$, $L_k(x_m) = 0$ for $k \neq m$, and $\deg(L_k) \leq n$. Write MATLAB function **yi = fundpol(k, x, xi)** which evaluates the kth Lagrange fundamental polynomial at points stored in the array **xi**.

15. The Lagrange form of the interpolating polynomial $p_n(x)$ of degree at most **n** which interpolates the data (x_k , y_k), $0 \leq k \leq n$, is

$$p_n(x) = y_0L_0(x) + y_1L_1(x) + \ldots + y_nL_n(x)$$

Write MATLAB function **yi = Lagrpol(x, y, xi)** that evaluates polynomial p_n at points stored in the array **xi**. You may wish to use function **fundpol** of Problem 14.

16. In this exercise you are to interpolate function **g(x)**, $a \leq x \leq b$, using functions **Newtonpol** (see Section 5.3) and **Lagrpol** (see Problem 15). Arrays **x**, **y**, and **xi** are defined as follows $x_k = a + k(b - a)/10$, $y_k = g(x_k)$, $k = 0, 1, \ldots , 10$, and **xi = linspace(a, b).** Run both functions using the following functions **g(x)** and the associated intervals **[a, b]**

(i) $g(x) = \sin(4\pi x)$, $[a, b] = [0, 1]$

(ii) $g(x) = J_0(x)$, $[a, b] = [2, 3]$,

where J_0 stands for the Bessel function of the first kind of order zero. In MATLAB Bessel function $J_0(x)$ can be evaluated using command **besselj(0, x)**.
In each case find the values **yi** of the interpolating polynomial at **xi** and compute the error maximum **err = norm(abs(yi - g(xi)), 'inf ')**. Compare efficiency of two methods used to interpolate function **g(x)**. Which method is more efficient? Explain why.

17. Continuation of Problem 16. Using MATLAB's function **interp1**, with options **'cubic'** and **'spline'**, interpolate both functions **g(x)** of Problem 16 and answer the same questions as stated in this problem. Among four method if interpolation you have used to interpolate functions g(x) which method is the the best one as long as

(i) efficiency is considered?

(ii) accuracy is considered?

18. The *Lebesgue function* $\Lambda(x)$ of the interpolating operator is defined as follows

$$\Lambda(x) = |L_0(x)| + |L_1(x)| + \ldots + |L_n(x)|,$$

where L_k stands for the kth fundamental polynomial introduced in Problem 14. This function was investigated in depth by numerous researchers. It's global maximum over the interval of interpolation provides a useful information about the error of interpolation.
In this exercise you are to graph function $\Lambda(x)$ for various sets of the interpolating abscissa $\{x_k\}$. We will assume that the points of interpolation are symmetric with respect to the origin, i.e., $-x_k = x_{n-k}$, for $k = 0, 1, \ldots , n$. Without loss of generality, we may also assume

that $-x_0 = x_n = 1$. Plot the graph of the Lebesgue function $\Lambda(x)$ for the following choices of the points x_k

(i) $x_k = -1 + 2k/n, \quad k = 0, 1, \dots, n$

(ii) $x_k = -\cos(k\pi/n), \quad k = 0, 1, \dots, n$

In each case put $n = 1, 2, 3$ and estimate the global maximum of $\Lambda(x)$. Which set of the interpolating abscissa provides a smaller value of $\mathbf{Max}\{\Lambda(x) : x_0 \leq x \leq x_n\}$?

19. MATLAB's function **polyder** computes the first order derivative of an algebraic polynomial that is represented by its coefficients in the descending order of powers. In this exercise you are to write MATLAB function **B = pold(A, k)** that computes the kth order derivative of several polynomials of the same degree whose coefficients are stored in the consecutive rows of the matrix **A**. This utility function is useful in manipulations with splines that are represented as the piecewise polynomial functions.
Hint: You may wish to use function **polyder**.

20. The Hermite cubic spline interpolant $s(x)$ with the breakpoints $\Delta = \{x_x < x_2 < \dots < x_m\}$ is a member of $Sp(3, 1, \Delta)$ that is uniquely determined by the interpolatory conditions

(i) $s(x_l) = y_l, \quad l = 1, 2, \dots, m$

(ii) $s'(x_l) = p_l, \quad l = 1, 2, \dots, m$

On the subinterval $[x_l, x_{l+1}]$, $l = 1, 2, \dots, m - 1$, $s(x)$ is represented as follows

$$s(x) = (1 + 2t)(1 - t)^2 y_l + (3 - 2t)t^2 y_{l+1} + h_l[t(1 - t)^2 p_l + t^2(t - 1)p_{l+1}],$$

where $t = (x - x_l)/(x_{l+1} - x_l)$ and $h_l = x_{l+1} - x_l$.
Prove that the Hermite cubic spline interpolant s(x) is convex on the interval $[x_1, x_m]$ if and only if the following inequalities

$$\frac{2p_l + p_{l+1}}{3} \leq \frac{s_{l+1} - s_l}{h_l} \leq \frac{p_l + 2p_{l+1}}{3}$$

are satisfied for all $l = 1, 2, \dots, m - 1$.

21. Write MATLAB function **[pts, yi] = Hermspl(x, y, p)** that computes coefficients of the Hermite cubic spline interpolant **s(x)** described in Problem 20 and evaluates spline interpolant at points stored in the array **xi**. Parameters **x**, **y**, and **p** stand for the breakpoints, values of **s(x)**, and values of **s'(x)** at the breakpoints of **s(x)**, respectively. The output parameter **yi** is the array of values of **s(x)** at points stored in the array **pts** which is defined as the union of the arrays **linspace(x(k), x(k+1))**, **k = 1, 2, … , n – 1**, where **n = length(x)**.
Hint: You may wish to use function **Hermpol** discussed in Section 5.3.

22. The nodes $\{x_k\}$ of the Newton – Cotes formulas of the open type are defined as follows $x_k = a + (k - 1/2)h$, $k = 1, 2, \dots, n - 1$, **where h = (b – a)/(n – 1)**. Write MATLAB function **[s, w, x] = oNCqf(fun, a, b, n, varargin)** that computes an approximate value **s** of

the integral of the function that is represented by the string **fun**. Interval of integration is **[a, b]** and the method used is the n-point open formula whose weights and nodes are are stored in the arrays **w** and **x**, respectively.

23. The Fresnel integral

$$\mathbf{f(x) = \int_0^x \exp\left(\frac{i\pi t^2}{2}\right)dt}$$

is of interest in several areas of applied mathematics. Write MATLAB function **[fr1, fr2] = Fresnel(x, tol, n)** which takes a real array **x**, a two dimensional vector **tol** holding the relative and absolute tolerance for the error of the computed integral (see MATLAB help file for the function **quad8**), and a positive integer **n** used in the function **Romberg** and returns numerical approximations **fr1** and **fr2** of the Fresnel integral using each of the following methods

(i) **quad8** with tolerance **tol = [1e-8 1e-8]**
(ii) **Romberg** with **n = 10**

Compute Fresnel integrals for the following values of **x = 0: 0.1:1**.To compare the approximations **fr1** and **fr2** calculate the number of *decimal places of accuracy* **dpa = –log10(norm(fr1 – fr2, 'inf '))**. For what choices of the input parameters **tol** and **n** the number **dpa** is greater than or equal to 13? The last inequality must be satisfied for all values **x** as defined earlier.

24. Let us assume that the real-valued function **f(x)** has a convergent integral

$$\mathbf{\int_0^\infty f(x)dx}\,.$$

Explain how would you compute an approximate value of this integral using function **Gquad2** developed earlier in this chapter? Extend your idea to convergent integrals of the form

$$\mathbf{\int_{-\infty}^{\infty} f(x)dx.}$$

25. The following integral is discussed in [3], p. 317

$$\mathbf{J = \int_{-1}^{1} \frac{dx}{x^4 + x^2 + 0.9}}\,.$$

To compute an approximate value of the integral **J** use

(i) MATLAB functions **quad** and **quad8** with tolerance **tol = [1e-8 1e-8]**
(ii) functions **Romberg** and **Gquad1** with **n = 8**.

Print out numerical results using **format long**. Which method should be recommended for numerical integration of the integral **J**? Justify your answer.

26. The arc length **s** of the ellipse

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}=1$$

from **(a, 0)** to **(x, y)** in quadrant one is equal to

$$s = b\int_0^{\theta}\sqrt{1-k^2\sin^2 t}\,dt$$

where $k^2 = 1 - (a/b)^2$, $a \le b$, and $\theta = \arccos(x/a) = \arcsin(y/b)$.
In this exercise you are to write MATLAB function **[sR, sq8] = arcell(a, b, x, n, tol)** that takes as the input parameters the semiaxes **a** and **b** and the x – coordinate of the point on the ellipse in quadrant one and returns an approximate value of the arc of ellipse from **(a, 0)** to **(x, y)** using functions **Romberg**, described in this chapter, and the MATLAB function **quad8**. The fourth input parameter **n** will be used by the function **Romberg**. The fifth input parameter **tol** is optional and it holds the user supplied tolerances for the relative and absolute errors in the computed approximation **sq8**. If **tol** is not supplied, the default values for tolerances should be assigned. For more details about using this parameter, type **help quad8** in the **Command Window**. Your program should also work for ellipses whose semiaxes are not restricted to those in the definition of the parameter k^2. Test your function for the following values of the input parameters

(i) **a = 1, b = 2, x = 1: -0.1: 0, n = 10, tol = [ ]**
(ii) **a = 2, b = 1, x = 2: -0.2: 0, n = 10, tol = [ ]**
(iii) **a = 2, b = 1, x = 0: -0.2: -2, n = 10, tol = [ ]**

Note that the terminal points **(x, y)** of the third ellipse lie in quadrant two.

27. Many of the most important special functions can be represented as the *Dirichlet average* **F** of a continuous function **f** (see [1])

$$F(b_1, b_2; a, b) = \frac{1}{B(b_1, b_2)}\int_0^1 t^{b_1-1}(1-t)^{b_2-1} f[ta+(1-t)b]\,dt,$$

where $B(b_1, b_2)$, $(b_1, b_2 > 0)$ stands for the *beta function*. Of special interest are the Dirichlet averages of elementary functions $f(t) = t^{-c}$ and $f(t) = e^t$. Former gives raise to the *hypergeometric functions* such as a celebrated *Gauss hypergeometric function* ${}_2F_1$ while the latter is used to represent the *confluent hypergeometric functions*.
In this exercise you are to implement a method for approximating the Dirichlet integral defined above using $f(t) = t^{-c}$. Write MATLAB function **y = dav(c, b1, b2, a, b)** which computes a numerical approximation of the Dirichlet average of **f**. Use a method of your choice to integrate numerically the Dirichlet integral. MATLAB has a function named **beta** designed for evaluating the beta function. Test your function for the following values of the parameter **c**:

c = 0 (exact value of the Dirichlet average **F** is equal to **1**)

$c = b_1 + b_2$ (exact value of the Dirichlet average is equal to **$1/(a^{b}_{1}\, b^{b}_{2})$**.

28. Gauss hypergeometric function **$_2F_1(a, b; c; x)$** is defined by the infinite power series as follows

$$_2F_1(a, b; c; x) = \sum_{n=0}^{\infty} \frac{(a,n)(b,n)}{(c,n)n!} x^n, \quad |x| \le 1,$$

where **$(a, n) = a(a + 1) \ldots (a + n - 1)$** is the *Appel symbol*. Gauss hypergeometric function can be represented as the Dirichlet average of the power function **$f(t) = t^{-a}$**

$$_2F_1(a, b; c; x) = F(b, c - b; 1 - x, 1) \qquad (c > b > 0,\ |x| < 1).$$

Many of the important elementary functions are special cases of this function. For instance for **$|x| < 1$**, the following formulas

$$\arcsin x = {}_2F_1(0.5, 0.5; 1.5; x^2)$$
$$\ln(1 + x) = x\,{}_2F_1(1, 1; 1.5; x^2)$$
$$\operatorname{arctanh} x = x\,{}_2F_1(0.5, 1; 1.5; x^2)$$

hold true. In this exercise you are to use function **dav** of Problem 27 to evaluate three functions listed above for **x = -0.9 : 0.1 : 0.9**. Compare obtained approximate values with those obtained by using MATLAB functions **asin**, **log**, and **atanh**.

29 Let **a** and **b** be positive numbers. In this exercise you will deal with the four formulas for computing the mean value of **a** and **b**. Among the well – known means the *arithmetic mean* **A(a, b) = (a + b)/2** and the geometric mean **$G(a, b) = \sqrt{ab}$** are the most frequently used ones. Two less known means are the *logarithmic mean* **L** and the *identric mean* **I**

$$L(a, b) = \frac{a - b}{\ln a - \ln b}$$

$$I(a, b) = e^{-1}(a^a/b^b)^{1/(a-b)}$$

The logarithmic and identric means are of interest in some problems that arise in economics, electrostatics, to mention a few areas only. All four means described in this problem can be represented as the Dirichlet averages of some elementary functions. For the means under discussion their b – parameters are both equal to one. Let **M(a, b)** stand for any of these means. Then

$$M(a, b) = f^{-1}(F(1, 1; a, b))$$

where **f^{-1}** stands for the inverse function of **f** and **F** is the Dirichlet average of **f**.
In this exercise you will deal with the means described earlier in this problem.

(i) Prove that the arithmetic, geometric, logarithmic and identric means can be represented as the inverse function of the Dirichlet average of the following functions $\mathbf{f(t) = t}$, $\mathbf{f(t) = t^{-2}}$, $\mathbf{f(t) = t^{-1}}$ and $\mathbf{f(t) = \ln t}$, respectively.

(ii) The logarithmic mean can also be represented as

$$\mathbf{L(a,b) = \int_0^1 a^t b^{1-t} dt.}$$

Establish this formula.

(iii) Use the integral representations you found in the previous part together with the midpoint and the trapezoidal quadrature formulas (with the error terms) to establish the following inequalities: $\mathbf{G \le L \le A}$ and $\mathbf{G \le I \le A}$. For the sake of brevity the arguments **a** and **b** are omitted in these inequalities.

30. A second order approximation of the second derivative of the function **f(x)** is

$$\mathbf{f\,''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2).}$$

Write MATLAB function **der2 = numder2(fun, x, h, n, varargin)** that computes an approximate value of the second order derivative of a function named by the string **fun** at the point **x**. Parameters **h** and **n** are user-supplied values of the initial step size and the number of performed iterations in the Richardson extrapolation. For functions that depend on parameters their values must follow the parameter **n**.
Test your function for **f(x) = tan x** with $\mathbf{x = \pi/4}$ and **h = 0.01**. Experiment with different values for **n** and compare your results with the exact value $\mathbf{f\,''(\pi/4) = 4}$.

31. Consider the following initial – value problem

$$\mathbf{y_1'''(t) = - y_1(t)y_1''(t), \quad y_1(0) = 1, \quad y_1'(0) = -1, \quad y_1''(0) = 1.}$$

(i) Replace the differential equation by the system of three differential equations of order one.

(ii) Write a MATLAB function **dy = order3(t, y)** that evaluates the right – hand sides of the equations you found in the previous part.

(iii) Write a script file **Problem31** to solve the resulting initial – value problem using MATLAB solver **ode45** on the interval **[0 1]**.

(iv) Plot in the same window graphs of function $\mathbf{y_1(t)}$ together with its derivatives up to order two. Add a legend to your graph that clearly describes curves you are plotting.

32. In this exercise you are to deal with the following initial – value problem

$$\mathbf{x\,'(t) = -x(t) - y(t), \quad y\,'(t) = -20x(t) - 2y(t), \quad x(0) = 2, \quad y(0) = 0.}$$

(i) Determine whether or not this system is stiff.

(ii) If it is, use an appropriate MATLAB solver to find a numerical solution on the interval **[0 1]**.

(iii) Plot the graphs of functions **x(t)** and **y(t)** in the same window.

33. The Lotka – Volterra equations describe populations of two species

$$y_1'(t) = y_1(t) - y_1(t)y_2(t), \quad y_2'(t) = -15y_2(t) + y_1(t)y_2(t).$$

Write MATLAB function **LV(y10, y20, tspan)** that takes the initial values **y10** = $y_1(0)$ and **y20** = $y_2(0)$ and plots graphs of the numerical solutions **y1** and **y2** over the interval **tspan**.

34. Numerical evaluation of a definite integral

$$\int_a^b f(t)dt$$

can be accomplished by solving the ODE **y'(t) = f(t)** with the initial condition **y(a) = 0**. Then the integral in question is equal to **y(b)**. Write MATLAB function **yb = integral(a, b, fun)** which implements this method. The input parameter **fun** is the string holding the name of the integrand **f(t)** and **a** and **b** are the limits of integration. To find a numerical solution of the ODE use the MATLAB solver **ode45**. Test your function on integrals of your choice.

35. Given the two – point boundary value problem

$$y'' = -t + t^2 + e^t - ty + ty', \quad y(0) = 1, \quad y(1) = 1 + e.$$

(i) Use function **bvp2ode** included in this tutorial to find the approximate values of the function **y** for the following values of **n = 8, 18**.

(ii) Plot, in the same window, graphs of functions you found in the previous part of the problem. Also, plot the graph of function $y(t) = t + e^t$ which is the exact solution to the boundary value problem in question.

36. Another method for solving the two – point boundary value problem is the *collocation method*. Let

$$y'' = f(t, y, y'), \quad y(a) = ya, \quad y(b) = yb.$$

This method computes a polynomial **p(t)** that approximates a solution **y(t)** using the following conditions

$$p(a) = ya, p(b) = yb, \quad p''(t_k) = f(t_k, p(t_k), p'(t_k))$$

where $k = 2, 3, \ldots, n - 1$ and $a = t_1 < t_2 < \ldots < t_n = b$ are the collocation points that are evenly spaced in the given interval.
In this exercise function **f** is assumed to be of the form $f(t, y, y') = g_0(t) + g_1(t)y + g_2(t)y'$.

(i) Set up a system of linear equations that follows from the collocation conditions.

(ii) Write MATLAB function **p = colloc(g0, g1, g2, a, b, ya, yb, n)** which computes coefficients **p** of the approximating polynomial. They should be stored in the array **p** in the descending order of powers. Note that the approximating polynomial is of degree **n – 1** or less.

37. Test the function **colloc** of Problem 36 using the two – point boundary value problem of Problem 35 and plot the graph of the approximating polynomial.