Scientific Computing Homework: Computer Problems

李阳 11935018

June 9, 2020

3.8. To demonstrate how results from the normal equations method and QR factorization can differ numerically, we need a least squares problem that is ill-conditioned and also has a small residual. We can generate such a problem as follows. We will fit a polynomial of degree $n-1$,

$$p_{n-1}(t) = x_1 + x_2 t + x_3 t^2 + \cdots + x_n t^{n-1},$$

to m data points $(t_i, y_i), m > n$. We choose $t_i = (i-1)/(m-1), i = 1, \ldots, m$, so that the data points are equally spaced on the interval $[0,1]$. We will generate the corresponding values y_i by first choosing values for the x_j, say, $x_j = 1, j = 1, \ldots, n$, and evaluating the resulting polynomial to obtain $y_i = p_{n-1}(t_i), i = 1, \ldots, m$. We could now see whether we can recover the x_j that we used to generate the y_i, but to make it more interesting, we first randomly perturb the y_i values to simulate the data error typical of least squares problems. Specifically, we take $y_i = y_i + (2u_i - 1) * \epsilon, i = 1, \ldots, m$, where each u_i is a random number uniformly distributed on the interval $[0, 1)$ and ϵ is a small positive number that determines the maximum perturbation. If you are using IEEE double precision, reasonable parameters for this problem are $m = 21, n = 12$, and $\epsilon = 10^{-10}$.
Having generated the data set (t_i, y_i) as just outlined, we will now compare the two methods for computing the least squares solution to this polynomial data-fitting problem. First, form the system of normal equations for this problem and solve it using a library routine for Cholesky factorization. Next, solve the least squares system using a library routine for QR factorization. Compare the two resulting solution vectors $\boldsymbol{x}$. For which method is the solution more sensitive to the perturbation we introduced into the data? Which method comes closer to recovering the $\boldsymbol{x}$ that we used to generate the data? Does the fact that the solutions differ affect our ability to fit the data points (t_i, y_i) closely by the polynomial?

Solution. The code is shown as follows.

```
m = 21; % number of data points
n = 11; % degree of the interpolating polynomial
epsilon = 1e-10; % perturbation parameter
t = linspace(0, 1, m)'; % equally spaced data points on
    [0,1]
y = zeros(size(t)); % polynomial values at t_i's
for k=0:n
    y = y + t.^k;
end
u = rand(m,1); % random numbers distributed in (0,1)
y = y + (2*u-1)*epsilon;
A = zeros(m, n+1);
for i=1:n+1
    A(:,i) = t.^(i-1);
end

% Normal equations approach
R = chol(A'*A); % Cholesky factorization
rhs = A'*y;
z = R'\rhs; % forward substitution
x = R\z; % backward substitution
fprintf('Normal equations approach: the inf-norm of the
    error is: \n%d\n', norm(x-ones(n+1,1), inf));

% QR factorization approach
[Q, R] = qr(A); % QR factorization
rhs = Q'*y;
x = R(1:n+1,:)\rhs(1:n+1); % backward substitution
fprintf('QR factorization approach: the inf-norm of the
    error is: \n%d\n', norm(x-ones(n+1,1), inf));
```

Running the code, we obtain the following numerical result:

```
Normal equations approach:
the inf-norm of the error is:
     9.817536e-01
QR factorization approach:
the inf-norm of the error is:
     8.737004e-05
```

From which we see that

- For the normal equations approach, the solution is more sensitive to the perturbation we introduced into the data, which is aligned with our theory that In solving the normal equation
$$A^T A\boldsymbol{x} = A^T\boldsymbol{y},$$
the condition number of $A^T A$ is the square of that of A.
- The QR factorization approach comes closer to recovering the $\boldsymbol{x}$ that we used to generate the data.
- The fact that the solutions differ **does not** affect our ablility to fit the data points (t_i, y_i) closely by the polynomial.

□

4.3

(a) Implement inverse iteration with a shift to compute the eigenvalues nearest to 2, and a corresponding normalized eigenvector, of the matrix

$$A = \begin{bmatrix} 6 & 2 & 1 \\ 2 & 3 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

You may use an arbitrary starting vector.

(b) Use a real symmetric eigensystem library routine to compute all of the eigenvalues and eigenvectors of the matrix, and compare the results with those obtained in part (a).

Solution. (a) The code is shown as follows.

```
A = [6 2 1;
     2 3 1;
     1 1 1];
mu = 2;
M = 5; % number of iterations
n = 3; % order of the matrix
v = zeros(n,1); v(1) = 1;
for k=1:M
     w = (A-mu*eye(n,n))\v; % apply (A-mu*I)
       ^-1
     v = w/norm(w); % normalize
     lambda = v'*A*v; % Rayleight quotient
end
fprintf('The eigenvalue nearest to 2 is:\n %.4d\n
       ', lambda);
fprintf('a corresponding eigenvector is:\n');
v
```

The numerical result obtained is

```
The eigenvalue nearest to 2 is:
  2.1331e+00
a corresponding eigenvector is:

v =

    0.4974
   -0.8196
   -0.2843
```

(b) We use the `matlab` function `eig` to compute the eigenvalue decomposition of A.

$$AV = VD,$$

where the columns of V are the eigenvectors of A and the diagonal entries of D are the corresponding eigenvalues.

```
>> [V, D] = eig(A); diag(D)

ans =

    0.5789
    2.1331
    7.2880

>> V

V =

   -0.0432   -0.4974   -0.8664
   -0.3507    0.8196   -0.4531
    0.9355    0.2843   -0.2098
```

From the above two numerical results, we see that the two results agree with each other up to fourth digit.

□

5.9 In celestial mechanics, *Kepler's equation*

$$M = E - e\sin(E)$$

relates the mean anomaly M to the eccentric anomaly E of an elliptical orbit of eccentricity e, where $0 < e < 1$.

(a) Prove that the fixed-point iteration using the iteration function

$$g(E) = M + e\sin(E)$$

is locally convergent.

(b) Use the fixed-point iteration scheme in part (a) to solve Kepler's equation for the eccentric anomaly E corresponding to a mean anomaly of $M = 1$ (radians) and an eccentricity of $e = 0.5$.

(c) Use Newton's method to solve the same problem.

(d) Use a library zero finder to solve the same problem.

Solution. (a) It sufficies to show that g is locally a contractive mapping, which is true since

$$|g'(E)| = e|\cos(E)| \leq e < 1.$$

Therefore, the fixed-point iteration using the iteration function

$$g(E) = M + e\sin(E)$$

is locally convergent.

(b) The fixed-point iteration formula in this case is given by

$$E_{n+1} = M + e\sin(E_n) = 1 + \frac{\sin(E_n)}{2}$$

```
E = 0; % initial  guess
tol  = 1e−6; % tolerance
while abs(E−1−sin(E)/2)>tol
      E = 1 + sin(E)/2; % fixed−point iteration
        formula
end
fprintf ('E: %d\n', E)
```

The numerical result obtained is

```
E: 1.498701e+00
```

(c) Using Newton's method, we have

$$\begin{aligned} E_{n+1} &= E_n - \frac{E_n - e\sin(E_n) - M}{1 - e\cos(E_n)} \\ &= E_n - \frac{E_n - \sin(E_n)/2 - 1}{1 - \cos(E_n)/2}. \end{aligned}$$

```
E = 1; epsilon  = 1e−6; M = 20;
for  k=0:M
      u = E − sin(E)/2 − 1;
      if  abs(u)<epsilon
            break;
      end
      E = E − u/(1 − cos(E)/2);
end
fprintf ('E: %d\n', E)
```

The numerical result obtained is

```
E: 1.498702e+00
```

(d)
```
>> M = 1; e = 0.5;
>> E = fzero(@(x) M+e*sin(x)-x, 1)
```
The numerical result obtained is

```
E = 1.4987
```

□

7.6 Interpolating the data points

t	0	1	4	9	16	25	36	49	64
y	0	1	2	3	4	5	6	7	8

should give an approximation to the square root function.

(a) Compute the polynomial of degree eight that interpolates these nine data points. Plot the resulting polynomial as well as the corresponding values given by the built-in `sqrt` function over the domain $[0, 64]$.

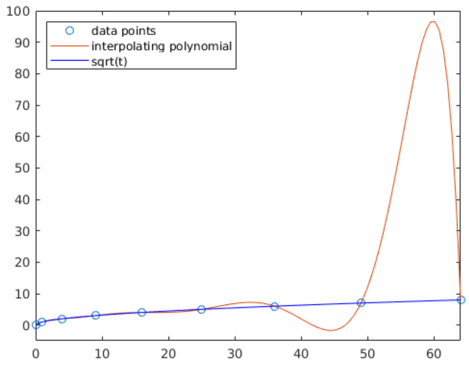
(b) Use a cubic spline routine to interpolate the same data and again plot the resulting curve along with the built-in `sqrt` function.

(c) Which of the two interpolants is more accurate over most of the domain?

(d) Which of the two interpolants is more accurate between 0 and 1?

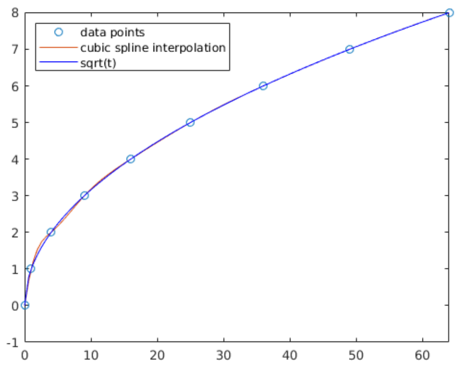
Solution. (a) The code and the plot are shown as follows.

```
% data points for interpolation
y = (0:1:8)';
t = y.^2;
u = linspace(0,64)'; % for plotting
% Perform Lagrange polynomial interpolation
n = length(t)
v = zeros(size(u));
for k = 1:n
    w = ones(size(u));
    for j=[1:k-1 k+1:n]
        w = (u-t(j))./(t(k)-t(j)).*w;
    end
    v = v + w*y(k);
end
plot(t,y,'o',u,v,'-')
axis([0 64 -5 100])
hold on
plot(u, sqrt(u), 'b')
hold off
legend('data points', 'interpolating polynomial', '
    sqrt(t)', 'Location', 'NorthWest')
```



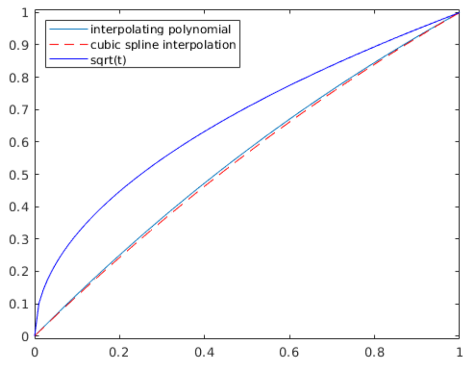
(b) The code and the plot are shown as follows.

```
% data points for interpolation
y = (0:1:8)';
t = y.^2;
u = linspace(0,64)'; % for plotting
% Perform cubic spline interpolation
v = spline(t,y,u);
plot(t,y,'o',u,v,'-')
axis([0 64 -01 8.01])
hold on
plot(u, sqrt(u), 'b')
hold off
legend('data points', 'cubic spline interpolation',
    'sqrt(t)', 'Location', 'NorthWest')
```



(c) From the above two figures, we see that the cubic spline interpolant is more accurate over most of the domain.

(d) The polynomial interpolant is slightly more accurate between 0 and 1, as illustrated by the following figure.



□

8.6 Evaluate the following quantities using each of the given methods:

(a) Use an adaptive quadrature routine to evaluate each of the integrals

$$I_k = e^{-1}\int_0^1 x^k e^x \, \mathrm{d}x$$

for $k = 0, 1, \ldots, 20$.

(b) Verify that the integrals just defined satisfy the recurrence

$$I_k = 1 - kI_{k-1},$$

and use it to generate the same quantities, starting with $I_0 = 1 - e^{-1}$.

(c) Generate the same quantities using the backward recurrence

$$I_{k-1} = (1 - I_k)/k,$$

beginning with $I_n = 0$ for some chosen value $n > 20$. Experiment with different values of n to see the effect on the accuracy of the values generated.

(d) Compare the three methods with respect to accuracy, stability, and execution time. Can you explain these results?

Solution. (a) We utilize the `matlab` routine `quad`, which uses adaptive Simpson quadrature to numerically evaluate an integral. The code is shown as follows.

```
% Evaluate the integral
%   I_k = exp(-1)\int_0^1 x^k*exp(x)dx
k = 21;
I = zeros(k,1);
fprintf(' k I_k\n');
for i=1:k
    I(i) = quad(@(x) x.^(i-1).*exp(x-1), 0, 1);
    fprintf('%2d %.4f\n', i-1, I(i));
end
```

The numerical result obtained is as follows.

```
 k I_k
 0 0.6321
 1 0.3679
 2 0.2642
 3 0.2073
 4 0.1709
 5 0.1455
 6 0.1268
 7 0.1124
 8 0.1009
 9 0.0916
10 0.0839
11 0.0774
12 0.0718
13 0.0669
14 0.0627
15 0.0590
16 0.0557
17 0.0528
18 0.0501
19 0.0477
20 0.0455
```

(b)

$$I_k = e^{-1}\int_0^1 x^k e^x \, \mathrm{d}x = e^{-1}\left(x^k e^x|_0^1 - \int_0^1 kx^{k-1}e^x \, \mathrm{d}x \right)$$
$$= 1 - kI_{k-1}.$$

The code is shown as follows.

```
% Evaluate the integral
%   I_k = exp(-1)\int_0^1 x^k*exp(x)dx
% using the recurrence (unstable)
%   I_k = 1 - k*I_{k-1}
k = 21;
I = zeros(k,1);
fprintf(' k I_k\n');
I(1) = 1 - exp(-1);
for i=2:k
    I(i) = 1 - i*I(i-1);
    fprintf('%2d %.2e\n', i-1, I(i));
end
```

The numerical result obtained is as follows.

```
 k I_k
 1 -2.64e-01
 2 1.79e+00
 3 -6.17e+00
 4 3.19e+01
 5 -1.90e+02
 6 1.33e+03
 7 -1.07e+04
 8 9.59e+04
 9 -9.59e+05
10 1.05e+07
11 -1.27e+08
12 1.65e+09
13 -2.30e+10
14 3.46e+11
15 -5.53e+12
16 9.40e+13
17 -1.69e+15
18 3.21e+16
```

```
19 -6.43e+17
20 1.35e+19
```

(c) The code is shown as follows.

```
% Evaluate the integral
%   I_k = exp(-1)\int_0^1 x^k*exp(x)dx
% using the recurrence (unstable)
%   I_{k-1} = (1-I_k)/k
n = 22; % try different n
k = 21;
I = zeros(n,1);
fprintf(' k I_k\n');
for i=n:-1:2
    I(i-1) = (1 - I(i))/i;
end
I = [1-I(1); I];
% output the result
for i=1:k
    fprintf('%2d %.4f\n', i-1, I(i));
end
```

For $n = 22$, the numerical result obtained is the same with that of (a) up to fourth digit.

(d) From the above numerical experiment, we see that the adaptive quadrature routine and the backward recurrence are accurate and stable. However, the recurrence used in (b) is unstable, which is of course inaccurate due to the error propagation. The backward recurrence is clearly more efficient than the adaptive quadrature routine since we have used more information.
With the recurrence

$$I_k = 1 - kI_{k-1},$$

it's easy to show via induction that

$$I_k = -k! \cdot I_0 - p_k,$$

where p_k is some number (independent of I_0). Therefore,

$$\left|\frac{I_k^* - I_k}{I_k}\right| \approx k! \left|\frac{I_0^* - I_0}{I_0}\right|,$$

i.e., in obtaining I_k, a relatively small perturbation in I_0 (for example, rounding to a machine number) is amplified by a factor of $k!$, which leads to the catastrophic phenomenon that we see in the above numerical experiment. Analyzing the backward recurrence shows that

$$\left|\frac{I_k^* - I_k}{I_k}\right| \approx \frac{1}{k!} \left|\frac{I_0^* - I_0}{I_0}\right|,$$

which shows the stability of the numerical scheme. □

11.2

(a) Use the method of lines and an ODE solver of your choice to solve the heat equation

$$u_t = u_{xx}, \quad 0 \le x \le 1, \quad t \ge 0,$$

with initial condition

$$u(0,x) = \sin(\pi x), \quad 0 \le x \le 1,$$

and Dirichlet boundary conditions

$$u(t,0) = 0, \quad u(t,1) = 0, \quad t \ge 0.$$

Integrate from $t = 0$ to $t = 0.1$. Plot the computed solution, preferably as a three-dimensional surface over the (t,x) plane. If you do not have three-dimensional plotting capability, plot the solution as a function of x for a few values of t, including the initial and final times. Determine the maximum error in the computed solution by comparing with the exact solution

$$u(t,x) = \exp(-\pi^2 t)\sin(\pi x).$$

Experiment with various spatial mesh sizes Δx, and try to characterize the error as a function of Δx. On a log-log scale, plot the maximum error as a function of Δx.

(b) Repeat part (a), but this time with initial condition

$$u(0,x) = \cos(\pi x), \quad 0 \le x \le 1,$$

and Neumann boundary conditions

$$u_x(t,0) = 0, \quad u_x(t,1) = 0, \quad t \ge 0,$$

and compare with the exact solution.

(a) Discretize the spatial domain

$$x_i = i\Delta x, \quad i = 0, 1, \ldots, m+1,$$

where the spatial mesh size $\Delta x = \frac{1}{m+1}$. Discretizing the heat equation with respect to spatial variables only, we obtain

$$\frac{\mathrm{d}u_i(t)}{\mathrm{d}t} = \frac{u_{i-1}(t) - 2u_i(t) + u_{i+1}(t)}{(\Delta x)^2}, \quad i = 1, \ldots, m$$

and

$$u_i(0) = \sin(\pi i \Delta x), \quad i = 1, \ldots, m,$$

the Dirichlet boundary conditions are discretized into

$$u_0(t) = 0, \quad u_{m+1}(t) = 0, \quad t \ge 0.$$

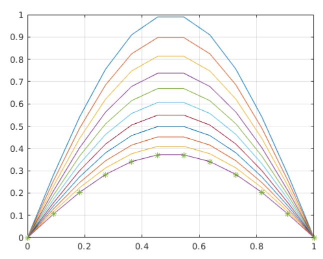
The code is shown as follows.

`func_mol.m`

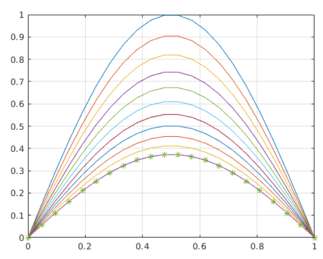
```
function up = func_mol(t, u)
global m h
up = zeros(size(u));
up(1) = (-2*u(1)+u(2))/h^2;
for i=2:m-1
    up(i) = (u(i-1)-2*u(i)+u(i+1))/h^2;
end
up(m) = (u(m-1)-2*u(m))/h^2;
end
```

`mol.m`

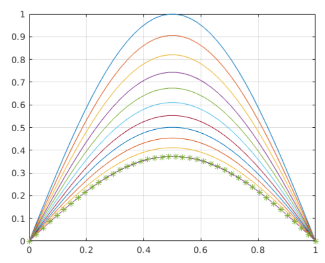
```
% Solve the heat equation using
% method-of-lines
global m h
a = 0; b = 1;
m = 5; h = (b-a)/(m+1);
x = linspace(a,b,m+2)';
t0 = 0; tfinal = 0.1;
y0 = sin(pi*x(2:end-1));
options = odeset('RelTol',1e-9,'AbsTol',1e-12);
[t,u] = ode23s('func_mol',[t0, tfinal ],y0,options);
plot(x,[0 u(1,:) 0]);
hold on
grid
for i=2:length(t)
    plot(x, [0 u(i,:) 0]);
end
uTrue = exp(-pi^2*tfinal)*sin(pi*x);
plot(x, uTrue, '*');
hold off
err = norm([0 u(end,:) 0]'-uTrue, inf)
```



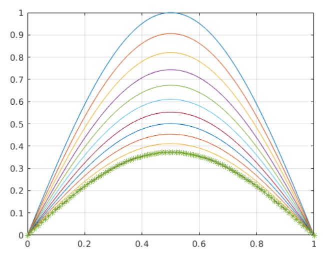
(a) $m = 10$



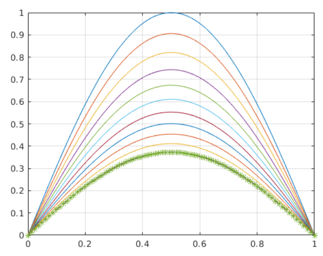
(b) $m = 20$



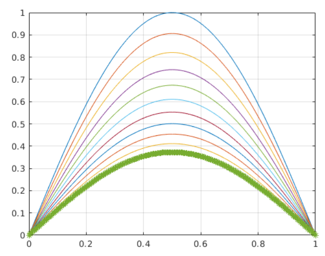
(c) $m = 40$



(d) $m = 80$



(e) $m = 160$

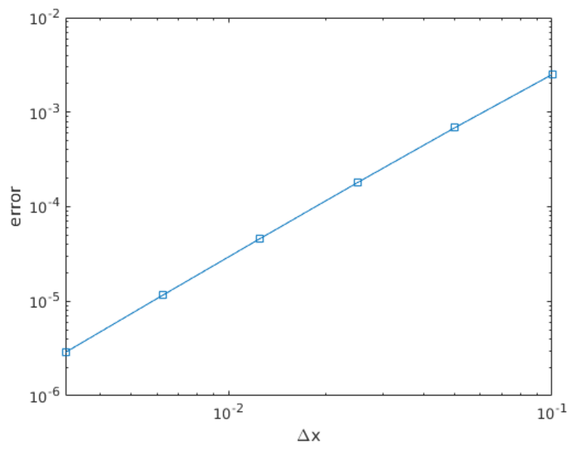


(f) $m = 320$

To characterize the error as a function of Δx, we modify the ODE solver used in `mol.m` as follows.

```
options = odeset('RelTol',1e-9,'AbsTol',1e-12);
[t,u] = ode23s('func_mol',[t0, tfinal],y0,options);
```

On a log-log scale, the maximum error as a function of Δx is as illustrated by the following figure.



(b) In this case, the discretization of the heat equation is

$$
\begin{aligned}
\frac{\mathrm{d}u_0(t)}{\mathrm{d}t} &= \frac{-2u_0(t) + 2u_1(t)}{(\Delta x)^2}, \\
\frac{\mathrm{d}u_i(t)}{\mathrm{d}t} &= \frac{u_{i-1}(t) - 2u_i(t) + u_{i+1}(t)}{(\Delta x)^2}, i = 1, \ldots, \\
\frac{\mathrm{d}u_{m+1}(t)}{\mathrm{d}t} &= \frac{2u_m(t) - 2u_{m+1}(t)}{(\Delta x)^2},
\end{aligned}
$$

with initial condition

$$u_i(0) = \cos(\pi i \Delta x), \quad i = 1, \ldots, m.$$

The code is shown as follows.

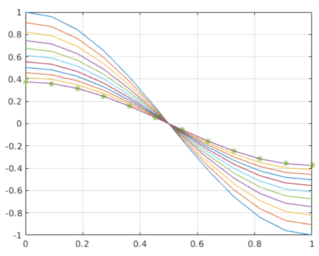
`func_mol2.m`

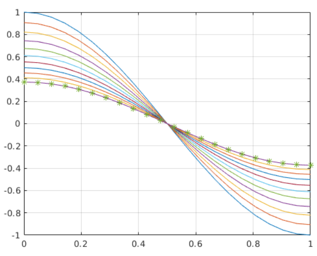
```
function up = func_mol2(t, u)
global m h
up = zeros(size(u));
up(1) = (-2*u(1)+2*u(2))/h^2;
for i=2:m+1
    up(i) = (u(i-1)-2*u(i)+u(i+1))/h^2;
end
up(m+2) = (2*u(m+1)-2*u(m+2))/h^2;
end
```

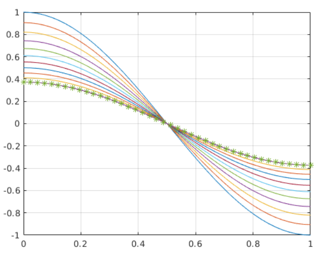
`mol2.m`

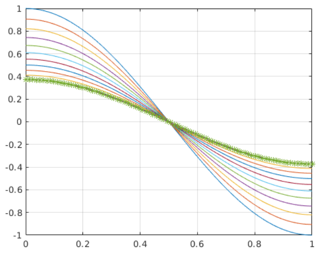
```
global m h
a = 0; b = 1;
m = 320; h = (b-a)/(m+1);
x = linspace(a,b,m+2)';
t0 = 0; tfinal = 0.1;
y0 = cos(pi*x);
options = odeset('RelTol',1e-9,'AbsTol',1e-12);
```

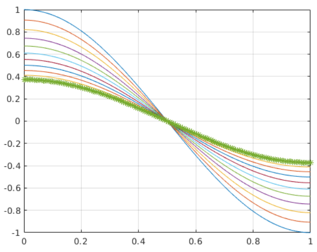
```
[t,u] = ode23s('func_mol2',[t0, tfinal ], y0,options);
plot(x,u(1,:));
hold on
grid
for i=2:length(t)
    plot(x, u(i,:));
end
uTrue = exp(−pi^2*tfinal)*cos(pi*x);
plot(x, uTrue, '*');
hold off
err = norm(u(end,:)'−uTrue, inf)
```

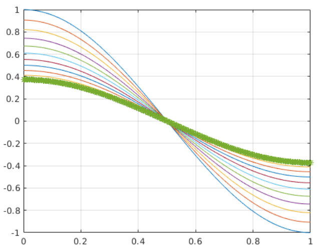

(g) $m = 10$


(h) $m = 20$


(i) $m = 40$


(j) $m = 80$

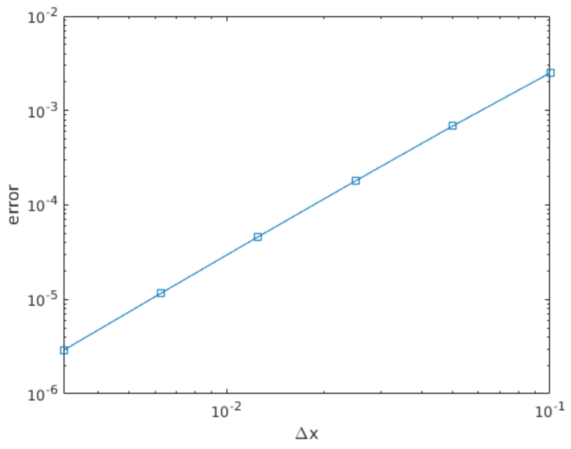

(k) $m = 160$


(l) $m = 320$

To characterize the error as a function of Δx, we modify the ODE solver used in `mol2.m` as follows.

```
options = odeset('RelTol',1e-9,'AbsTol',1e-12);
[t,u] = ode23s('func_mol',[t0, tfinal],y0,options);
```

On a log-log scale, the maximum error as a function of Δx is as illustrated by the following figure.


error
Δx
10^-2
10^-3
10^-4
10^-5
10^-6
10^-2
10^-1