2

A One-Dimensional, Linear Partial Differential Equation

This partial differential equation (PDE) problem is considered for the following reasons:

1. The PDE has an exact solution that can be used to assess the accuracy of the numerical method of lines (MOL) solution.
2. Both Dirichlet and Neumann boundary conditions (BCs) are included in the analysis.
3. The use of library routines for the finite-difference (FD) approximation of the spatial (boundary-value) derivative is illustrated.
4. The explicit programming of the FD approximations is included for comparison with the use of the library routines.
5. Some basic methods for assessing the accuracy of the MOL solution are presented.

The PDE is the *one-dimensional (1D) heat conduction equation in Cartesian coordinates:*

$$u_t = u_{xx} \tag{2.1}$$

Here we have used subscript notation for partial derivatives, so

$$u_t \leftrightarrow \frac{\partial u}{\partial t}$$

$$u_{xx} \leftrightarrow \frac{\partial^2 u}{\partial x^2}$$

The initial condition (IC) is

$$u(x, t = 0) = \sin(\pi x/2) \tag{2.2}$$

A *Dirichlet* BC is specified at $x = 0$,

$$u(x = 0, t) = 0 \tag{2.3}$$

and a *Neumann* BC is specified at $x = 1$,

$$u_x(x = 1, t) = 0 \tag{2.4}$$

The analytical solution to Eqs. (2.1)–(2.4) is

$$u(x,t) = e^{-(\pi^2/4)t}\sin(\pi x/2) \tag{2.5}$$

A main program in Matlab for the MOL solution of Eqs. (2.1)–(2.4) with the analytical solution, Eq. (2.5), included for comparison with the MOL solution, is given in Listing 2.1.

```
%
% Clear previous files
  clear all
  clc
%
% Parameters shared with the ODE routine
  global ncall ndss
%
% Initial condition
  n=21;
  for i=1:n
     u0(i)=sin((pi/2.0)*(i-1)/(n-1));
  end
%
% Independent variable for ODE integration
  t0=0.0;
  tf=2.5;
  tout=linspace(t0,tf,n);
  nout=n;
  ncall=0;
%
% ODE integration
  mf=1;
  reltol=1.0e-04; abstol=1.0e-04;
  options=odeset('RelTol',reltol,'AbsTol',abstol);
  if(mf==1) % explicit FDs
    [t,u]=ode15s(@pde_1,tout,u0,options); end
  if(mf==2) ndss=4; % ndss = 2, 4, 6, 8 or 10 required
    [t,u]=ode15s(@pde_2,tout,u0,options); end
  if(mf==3) ndss=44; % ndss = 42, 44, 46, 48 or 50 required
    [t,u]=ode15s(@pde_3,tout,u0,options); end
%
% Store numerical and analytical solutions, errors at x = 1/2
  n2=(n-1)/2.0+1;
  sine=sin(pi/2.0*0.5);
  for i=1:nout
    u_plot(i)=u(i,n2);
    u_anal(i)=exp(-pi^2/4.0*t(i))*sine;
    err_plot(i)=u_plot(i)-u_anal(i);
  end
```

```
%
% Display selected output
  fprintf('\n mf = %2d  abstol = %8.1e  reltol = %8.1e\n', ...
          mf,abstol,reltol);
  fprintf('\n     t       u(0.5,t)  u_anal(0.5,t)
          err u(0.5,t)\n');
  for i=1:5:nout
    fprintf('%6.3f%15.6f%15.6f%15.7f\n', ...
            t(i),u_plot(i),u_anal(i),err_plot(i));
  end
  fprintf('\n ncall = %4d\n',ncall);
%
% Plot numerical solution and errors at x = 1/2
  figure(1);
  subplot(1,2,1)
  plot(t,u_plot); axis tight
  title('u(0.5,t) vs t'); xlabel('t'); ylabel('u(0.5,t)')
  subplot(1,2,2)
  plot(t,err_plot); axis tight
  title('Err u(0.5,t) vs t'); xlabel('t');
        ylabel('Err u(0.5,t)');
  print -deps pde.eps; print -dps pde.ps; print -dpng pde.png
%
% Plot numerical solution in 3D perspective
  figure(2);
  colormap('Gray');
  C=ones(n);
  g=linspace(0,1,n); % For distance x
  h1=waterfall(t,g,u',C);
  axis('tight');
  grid off
  xlabel('t, time')
  ylabel('x, distance')
  zlabel('u(x,t)')
  s1=sprintf('Diffusion Equation - MOL Solution');
  sTmp=sprintf('u(x,0) = sin(\\pi x/2 )');
  s2=sprintf('Initial condition: %s',sTmp);
  title([{s1}, {s2}],'fontsize',12);
  v=[0.8616   -0.5076    0.0000   -0.1770
     0.3712    0.6301    0.6820   -0.8417
     0.3462    0.5876   -0.7313    8.5590
          0         0         0    1.0000];
  view(v);
  rotate3d on;
```

Listing 2.1. Main program `pde_1_main`

We can note the following points about the main program given in Listing 2.1:

1. After declaring some parameters `global` so that they can be shared with other routines called via this main program, IC (2.2) is computed over a 21-point grid in x.

```
%
% Clear previous files
  clear all
  clc
%
% Parameters shared with the ODE routine
  global ncall ndss
%
% Initial condition
  n=21;
  for i=1:n
    u0(i)=sin((pi/2.0)*(i-1)/(n-1));
  end
```

2. The independent variable `t` is defined over the interval $0 \leq t \leq 2.5$; again, a 21-point grid is used.

```
%
% Independent variable for ODE integration
  t0=0.0;
  tf=2.5;
  tout=linspace(t0,tf,n);
  nout=n;
  ncall=0;
```

3. The 21 ordinary differential equations (ODEs) are then integrated by a call to the Matlab integrator `ode15s`.

```
%
% ODE integration
  mf=1;
  reltol=1.0e-04; abstol=1.0e-04;
  options=odeset('RelTol',reltol,'AbsTol',abstol);
  if(mf==1) % explicit FDs
    [t,u]=ode15s(@pde_1,tout,u0,options); end
```

```
if(mf==2) ndss=4; % ndss = 2, 4, 6, 8 or 10 required
  [t,u]=ode15s(@pde_2,tout,u0,options); end
if(mf==3) ndss=44; % ndss = 42, 44, 46, 48 or 50 required
  [t,u]=ode15s(@pde_3,tout,u0,options); end
```

Three cases are programmed corresponding to `mf=1,2,3`, for which three different ODE routines, `pde_1`, `pde_2`, and `pde_3`, are called (these routines are discussed subsequently). The variable `ndss` refers to a library of differentiation routines for use in the MOL solution of PDEs; the use of `ndss` is illustrated in the subsequent discussion. Note that a stiff integrator, `ode15s`, was selected because the 21 ODEs are sufficiently stiff that a nonstiff integrator results in a large number of calls to the ODE routine.

4. Selected numerical results are stored for subsequent tabular and plotted output.

```
%
% Store numerical and analytical solutions, errors at x = 1/2
  n2=(n-1)/2.0+1;
  sine=sin(pi/2.0*0.5);
  for i=1:nout
    u_plot(i)=u(i,n2);
    u_anal(i)=exp(-pi^2/4.0*t(i))*sine;
    err_plot(i)=u_plot(i)-u_anal(i);
  end
```

5. Selected tabular numerical output is displayed.

```
%
% Display selected output
  fprintf('\n mf = %2d   abstol = %8.1e  reltol = %8.1e\n',...
          mf,abstol,reltol);
  fprintf('\n   t   u(0.5,t)  u_anal(0.5,t)  err u(0.5,t)\n');
  for i=1:5:nout
    fprintf('%6.3f%15.6f%15.6f%15.7f\n',...
            t(i),u_plot(i),u_anal(i),err_plot(i));
  end
  fprintf('\n ncall = %4d\n',ncall);
```

The output from this code is given in Table 2.1.

Table 2.1. Output for `mf=1` from `pde_1_main` and `pde_1`

```
mf =  1   abstol = 1.0e-004   reltol = 1.0e-004

    t       u(0.5,t)  u_anal(0.5,t)   err u(0.5,t)
0.000       0.707107       0.707107      0.0000000
0.625       0.151387       0.151268      0.0001182
1.250       0.032370       0.032360      0.0000093
1.875       0.006894       0.006923     -0.0000283
2.500       0.001472       0.001481     -0.0000091

ncall =   85
```

The output displayed in Table 2.1 indicates that the MOL solution agrees with the analytical solution to at least three significant figures. Also, `ode15s` calls the derivative routine only 85 times (in contrast with the nonstiff integrator `ode45`, which requires approximately 5,000–10,000 calls, clearly indicating the advantage of a stiff integrator for this problem).

6. The MOL solution and its error (computed from the analytical solution) are plotted.

```
%
% Plot numerical solution and errors at x = 1/2
  figure(1);
  subplot(1,2,1)
  plot(t,u_plot); axis tight
  title('u(0.5,t) vs t'); xlabel('t'); ylabel('u(0.5,t)')
  subplot(1,2,2)
  plot(t,err_plot); axis tight
  title('Err u(0.5,t) vs t'); xlabel('t'); ...
        ylabel('Err u(0.5,t)')
  print -deps pde.eps; print -dps pde.ps;  print -dpng pde.png
```

The plotted error output shown in Figure 2.1 indicates that the error in the MOL solution varied between approximately -3×10^{-5} and 16×10^{-5}, which is not quite within the error range specified in the program

```
    reltol=1.0e-04; abstol=1.0e-04;
```

The fact that the error tolerances illustrated in Figure 2.1 were not satisfied does not necessarily mean that `ode15s` failed to adjust the integration

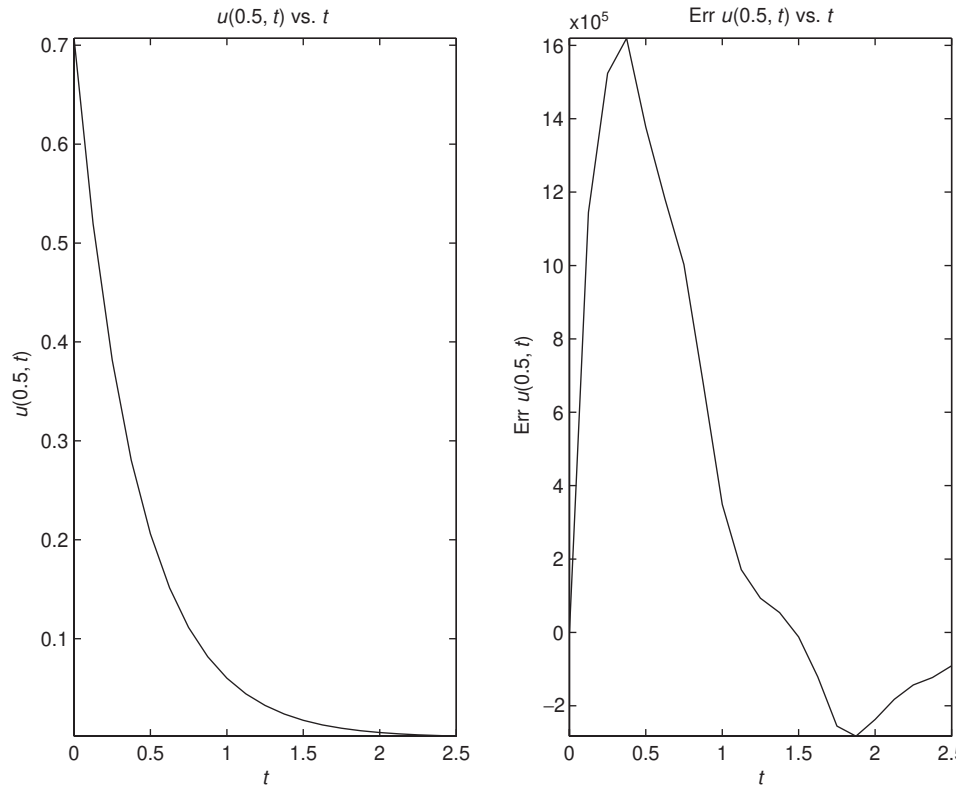

Figure 2.1. Two-dimensional graphical output from `pde_1_main`; `mf=1`

interval to meet these error tolerances. Rather, the error of approximately 1.6×10^{-4} is due to the limited accuracy of the second-order FD approximation of $\partial^2 u/\partial x^2$ programmed in `pde_1`. This conclusion is confirmed when the main program calls `pde_2` (for `mf=2`) or `pde_3` (for `mf=3`), as discussed subsequently; these two routines have FD approximations that are more accurate than in `pde_1`, so the errors fall below the specified tolerances.

This analysis indicates that two sources of errors result from the MOL solution of PDEs such as Eq. (2.1): (1) errors due to the integration in t (by `ode15s`) and (b) errors due to the approximation of the spatial derivatives such as $\partial^2 u/\partial x^2$ programmed in the derivative routine such as `pde_1`. In other words, we have to be attentive to integration errors in the *initial-* and *boundary-value independent variables*.

In summary, a comparison of the numerical and analytical solutions indicates that 21 grid points in x were not sufficient when using the second-order FDs in `pde_1`. However, in general, we will not have an analytical solution such as Eq. (2.5) to determine if the number of spatial grid points is adequate. In this case, some experimentation with the number of grid points, and the observation of the resulting solutions to infer the degree of accuracy or *spatial convergence*, may be required.

7. A 3D plot is also produced.

```
%
% Plot numerical solution in 3D perspective
  figure(2);
  colormap('Gray');
  C=ones(n);
  g=linspace(0,1,n); % For distance x
  h1=waterfall(t,g,u',C);
  axis('tight');
  grid off
  xlabel('t, time')
  ylabel('x, distance')
  zlabel('u(x,t)')
  s1=sprintf('Diffusion Equation - MOL Solution');
  sTmp=sprintf('u(x,0) = sin(\\pi x/2 )');
  s2=sprintf('Initial condition: %s',sTmp);
  title([{s1}, {s2}],'fontsize',12);
  v=[0.8616   -0.5076    0.0000   -0.1770
     0.3712    0.6301    0.6820   -0.8417
     0.3462    0.5876   -0.7313    8.5590
          0         0         0    1.0000];
  view(v);
  rotate3d on;
```

The plotted output shown in Figure 2.2 clearly indicates the origin of the *lines* in the *method of lines* (also discussed in Chapter 1).

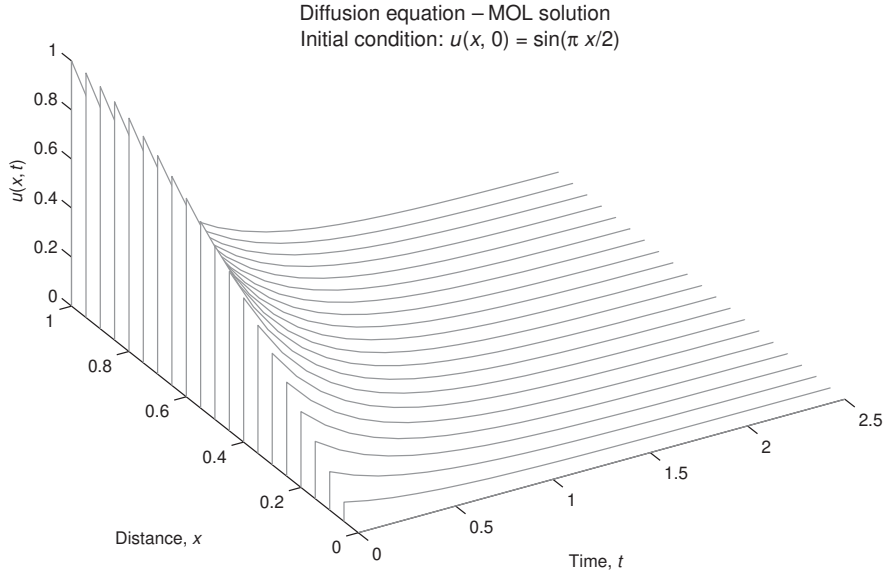

Figure 2.2. Three-dimensional graphical output from `pde_1_main`; `mf=1`

The programming of the approximating MOL/ODEs is in one of the three routines called by ode15s. We now consider each of these routines. For mf=1, ode15s calls function pde_1 (see Listing 2.2).

```
  function ut=pde_1(t,u)
%
% Problem parameters
  global ncall
  xl=0.0;
  xu=1.0;
%
% PDE
  n=length(u);
  dx2=((xu-xl)/(n-1))^2;
  for i=1:n
    if(i==1)     ut(i)=0.0;
    elseif(i==n) ut(i)=2.0*(u(i-1)-u(i))/dx2;
    else         ut(i)=(u(i+1)-2.0*u(i)+u(i-1))/dx2;
    end
  end
  ut=ut';
%
% Increment calls to pde_1
  ncall=ncall+1;
```

Listing 2.2. Routine pde_1

We can note the following points about pde_1:

1. After the call definition of the function, some problem parameters are defined.

```
  function ut=pde_1(t,u)
%
% Problem parameters
  global ncall
  xl=0.0;
  xu=1.0;
```

xl and xu could have also been set in the main program and passed to pde_1 as global variables. The defining statement at the beginning of pde_1 indicates

that the independent variable t and dependent variable vector u are inputs to pde_1, while the output is the vector of t derivatives, ut; in other words, all of the n ODE derivatives in t must be defined in pde_1.

2. The FD approximation of Eq. (2.1) is then programmed.

```
%
% PDE
  n=length(u);
  dx2=((xu-xl)/(n-1))^2;
  for i=1:n
    if(i==1)      ut(i)=0.0;
    elseif(i==n) ut(i)=2.0*(u(i-1)-u(i))/dx2;
    else          ut(i)=(u(i+1)-2.0*u(i)+u(i-1))/dx2;
  end
  end
  ut=ut';
```

The number of ODEs (21) is determined by the length command n=length(u); so that the programming is general (the number of ODEs can easily be changed in the main program). The square of the FD interval, dx2, is then computed.

3. The MOL programming of the 21 ODEs is done in the for loop. For BC (2.3), the coding is

```
    if(i==1) ut(i)=0.0;
```

since the value of $u(x = 0, t) = 0$ does not change after being set as an IC in the main program (and therefore its time derivative is zero).

4. For BC (2.4), the coding is

```
    elseif(i==n) ut(i)=2.0*(u(i-1)-u(i))/dx2;
```

which follows directly from the FD approximation of BC (2.4),

$$u_x \approx \frac{u(i+1) - u(i-1)}{\Delta x} = 0$$

or with $i = n$,

$$u(n+1) = u(n-1)$$

Note that the *fictitious value* $u(n+1)$ can then be replaced in the ODE at $i = n$ by $u(n-1)$.

5. For the remaining interior points, the programming is

```
    else ut(i)=(u(i+1)-2.0*u(i)+u(i-1))/dx2;
```

which follows from the FD approximation of the second derivative

$$u_{xx} \approx \frac{(u(i+1) - 2u(i) + u(i-1))}{\Delta x^2}$$

6. Since the Matlab ODE integrators require a column vector of derivatives, a final transpose of `ut` is required.

```
  ut=ut';
%
% Increment calls to pde_1
  ncall=ncall+1;
```

Finally, the number of calls to `pde_1` is incremented so that at the end of the solution, the value of `ncall` displayed by the main program gives an indication of the computational effort required to produce the entire solution. The numerical and graphical output for this case (`mf=1`) was discussed previously.

For `mf=2`, function `pde_2` is called by `ode15s` (see Listing 2.3).

```
function ut=pde_2(t,u)
%
% Problem parameters
  global ncall ndss
  xl=0.0;
  xu=1.0;
%
% BC at x = 0 (Dirichlet)
  u(1)=0.0;
%
% Calculate ux
  n=length(u);
  if    (ndss== 2) ux=dss002(xl,xu,n,u); % second order
  elseif(ndss== 4) ux=dss004(xl,xu,n,u); % fourth order
```

```
  elseif(ndss== 6) ux=dss006(xl,xu,n,u); % sixth order
  elseif(ndss== 8) ux=dss008(xl,xu,n,u); % eighth order
  elseif(ndss==10) ux=dss010(xl,xu,n,u); % tenth order
  end
%
% BC at x = 1 (Neumann)
  ux(n)=0.0;
%
% Calculate uxx
  if    (ndss== 2) uxx=dss002(xl,xu,n,ux); % second order
  elseif(ndss== 4) uxx=dss004(xl,xu,n,ux); % fourth order
  elseif(ndss== 6) uxx=dss006(xl,xu,n,ux); % sixth order
  elseif(ndss== 8) uxx=dss008(xl,xu,n,ux); % eighth order
  elseif(ndss==10) uxx=dss010(xl,xu,n,ux); % tenth order
  end
%
% PDE
  ut=uxx';
  ut(1)=0.0;
%
% Increment calls to pde_2
  ncall=ncall+1;
```

Listing 2.3. Routine `pde_2`

We can note the following points about `pde_2`:

1. The initial statements are the same as in `pde_1`. Then the Dirichlet BC at $x = 0$ is programmed.

```
%
% BC at x = 0 (Dirichlet)
  u(1)=0.0;
```

   Actually, the statement `u(1)=0.0;` has no effect since the dependent variables can only be changed through their derivatives, that is, `ut(1)`, in the ODE derivative routine. This code was included just to serve as a reminder of the BC at $x = 0$, which is programmed subsequently.
2. The first-order spatial derivative $\partial u/\partial x = u_x$ is then computed.

```
%
% Calculate ux
  n=length(u);
```

```
  if (ndss==2)     ux=dss002(xl,xu,n,u); % second order
  elseif(ndss== 4) ux=dss004(xl,xu,n,u); % fourth order
  elseif(ndss== 6) ux=dss006(xl,xu,n,u); % sixth order
  elseif(ndss== 8) ux=dss008(xl,xu,n,u); % eighth order
  elseif(ndss==10) ux=dss010(xl,xu,n,u); % tenth order
  end
```

Five library routines, `dss002` to `dss010`, are programmed that use second-order to tenth-order FD approximations, respectively. Since `ndss=4` is specified in the main program, `dss004` is used in the calculation of `ux`.

3. BC (2.4) is then applied, followed by the calculation of the second-order spatial derivative from the first-order spatial derivative.

```
%
% BC at x = 1 (Neumann)
  ux(n)=0.0;
%
% Calculate uxx
  if    (ndss== 2) uxx=dss002(xl,xu,n,ux); % second order
  elseif(ndss== 4) uxx=dss004(xl,xu,n,ux); % fourth order
  elseif(ndss== 6) uxx=dss006(xl,xu,n,ux); % sixth order
  elseif(ndss== 8) uxx=dss008(xl,xu,n,ux); % eighth order
  elseif(ndss==10) uxx=dss010(xl,xu,n,ux); % tenth order
  end
```

Again, `dss004` is called, which is the usual procedure (the order of the FD approximation is generally not changed in computing higher-order derivatives from lower-order derivatives, a process termed *stagewise differentiation*).

4. Finally, Eq. (2.1) is programmed and the Dirichlet BC at $x = 0$ (Eq. (2.3)) is applied.

```
%
% PDE
  ut=uxx';
  ut(1)=0.0;
%
% Increment calls to pde_2
  ncall=ncall+1;
```

Note the similarity of the code to the PDE (Eq. (2.1)), and also the transpose required by `ode15s`.

Table 2.2. Output for `mf=2` from `pde_1_main` and `pde_2`

```
mf =  2   abstol = 1.0e-004   reltol = 1.0e-004

    t       u(0.5,t)  u_anal(0.5,t)   err u(0.5,t)
0.000       0.707107       0.707107      0.0000000
0.625       0.151267       0.151268     -0.0000013
1.250       0.032318       0.032360     -0.0000418
1.875       0.006878       0.006923     -0.0000446
2.500       0.001467       0.001481     -0.0000138

ncall =   62
```

The numerical output for this case (`mf=2`) is provided in Table 2.2. The plotted error output given in Figure 2.3 indicates that the error in the MOL solution varied between approximately -5×10^{-5} and 3.2×10^{-5}, which is within the error range specified in the program

```
reltol=1.0e-04; abstol=1.0e-04;
```

Thus, switching from the second-order FDs in `pde_1` to fourth-order FDs in `pde_2` reduced the *spatial truncation error* so that the MOL solution met the specified error tolerances.

For `mf=3`, function `pde_3` is called by `ode15s`, as given in Listing 2.4.

```
  function ut=pde_3(t,u)
%
% Problem parameters
  global ncall ndss
  xl=0.0;
  xu=1.0;
%
% BC at x = 0
  u(1)=0.0;
%
% BC at x = 1
  n=length(u);
  ux(n)=0.0;
%
% Calculate uxx
  nl=1; % Dirichlet
  nu=2; % Neumann
```

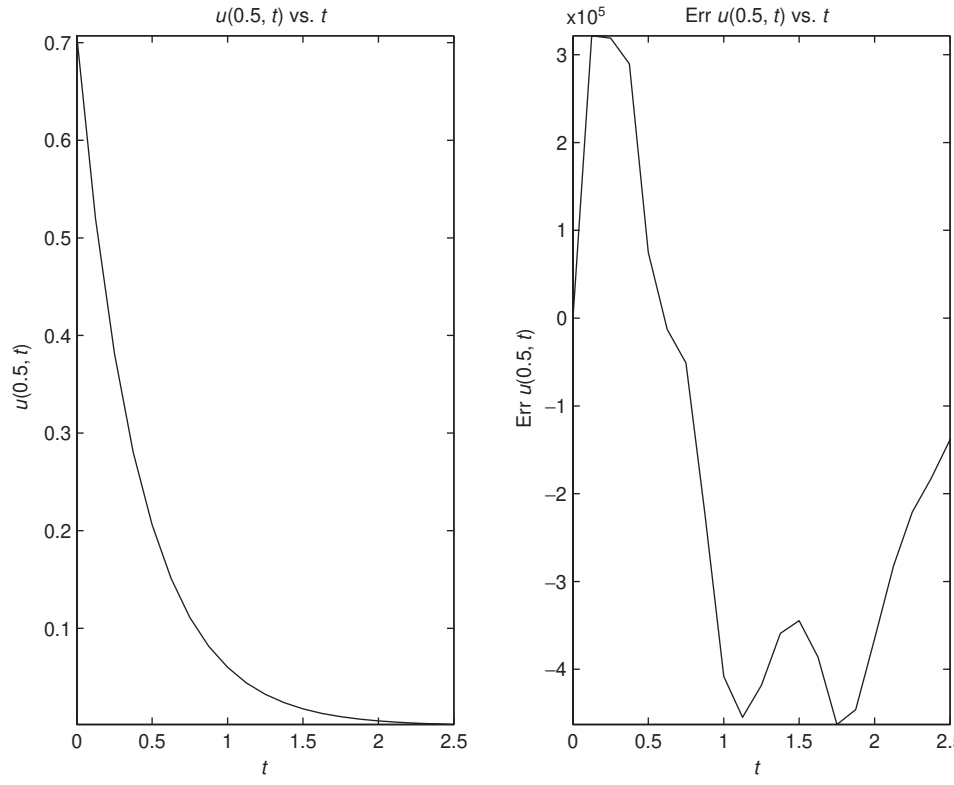

Figure 2.3. Two-dimensional graphical output from `pde_1_main`; `mf=2`

```
  if    (ndss==42) uxx=dss042(xl,xu,n,u,ux,nl,nu);
         % second order
  elseif(ndss==44) uxx=dss044(xl,xu,n,u,ux,nl,nu);
         % fourth order
  elseif(ndss==46) uxx=dss046(xl,xu,n,u,ux,nl,nu);
         % sixth order
  elseif(ndss==48) uxx=dss048(xl,xu,n,u,ux,nl,nu);
         % eighth order
  elseif(ndss==50) uxx=dss050(xl,xu,n,u,ux,nl,nu);
         % tenth order
  end
%
% PDE
  ut=uxx';
  ut(1)=0.0;
%
% Increment calls to pde_3
  ncall=ncall+1;
```

Listing 2.4. Routine `pde_3`

We can note the following points about pde_3:

1. The initial statements are the same as in pde_1. Then the Dirichlet BC at $x = 0$ and the Neumann BC at $x = 1$ are programmed.

```
  function ut=pde_3(t,u)
%
% Problem parameters
  global ncall ndss
  xl=0.0;
  xu=1.0;
%
% BC at x = 0
  u(1)=0.0;
%
% BC at x = 1
  n=length(u);
  ux(n)=0.0;
```

   Again, the statement u(1)=0.0; has no effect (since the dependent variables can only be changed through their derivatives, i.e., ut(1), in the ODE derivative routine). This code was included just to serve as a reminder of the BC at $x = 0$, which is programmed subsequently.
2. The second-order spatial derivative $\partial^2 u/\partial x^2 = u_{xx}$ is then computed.

```
%
% Calculate uxx
  nl=1; % Dirichlet
  nu=2; % Neumann
  if (ndss==42)    uxx=dss042(xl,xu,n,u,ux,nl,nu); % second order
  elseif(ndss==44) uxx=dss044(xl,xu,n,u,ux,nl,nu); % fourth order
  elseif(ndss==46) uxx=dss046(xl,xu,n,u,ux,nl,nu); % sixth order
  elseif(ndss==48) uxx=dss048(xl,xu,n,u,ux,nl,nu); % eighth order
  elseif(ndss==50) uxx=dss050(xl,xu,n,u,ux,nl,nu); % tenth order
  end
```

   Five library routines, dss042 to dss050, are programmed that use second-order to tenth-order FD approximations, respectively, for a second derivative. Since ndss=44 is specified in the main program, dss044 is used in the calculation of uxx. Also, these differentiation routines have two parameters that specify the type of BCs: (a) nl=1 or 2 specifies a Dirichlet or a

Table 2.3. Output for `mf=3` from `pde_1_main` and `pde_3`

```
mf =  3   abstol = 1.0e-004   reltol = 1.0e-004

    t       u(0.5,t)  u_anal(0.5,t)   err u(0.5,t)
0.000       0.707107       0.707107      0.0000000
0.625       0.151267       0.151268     -0.0000017
1.250       0.032318       0.032360     -0.0000420
1.875       0.006878       0.006923     -0.0000447
2.500       0.001467       0.001481     -0.0000138

ncall =   62
```

Neumann BC, respectively, at the lower boundary value of $x = xl(= 0)$; in this case, BC (2.3) is Dirichlet, so `nl=1`; and (b) `nu=1` or 2 specifies a Dirichlet or a Neumann BC, respectively, at the upper boundary value of $x = xu(= 1)$; in this case, BC (2.4) is Neumann, so `nu=2`.

3. Finally, Eq. (2.1) is programmed and the Dirichlet BC at $x = 0$ (Eq. (2.3)) is applied.

```
%
% PDE
  ut=uxx';
  ut(1)=0.0;
%
% Increment calls to pde_3
  ncall=ncall+1;
```

Again, the transpose is required by `ode15s`.

The numerical output for this case (`mf=3`) is given in Table 2.3. The plotted error output shown in Figure 2.4 indicates that the error in the MOL solution varied between approximately -4.8×10^{-5} and 3.2×10^{-5}, which is within the error range specified in the program

```
reltol=1.0e-04; abstol=1.0e-04;
```

We conclude the example given in Figure 2.4 with the following observation: As the solution approaches steady state, $t \to \infty$, $u_t \to 0$, and from Eq. (2.1), $u_{xx} \to 0$.

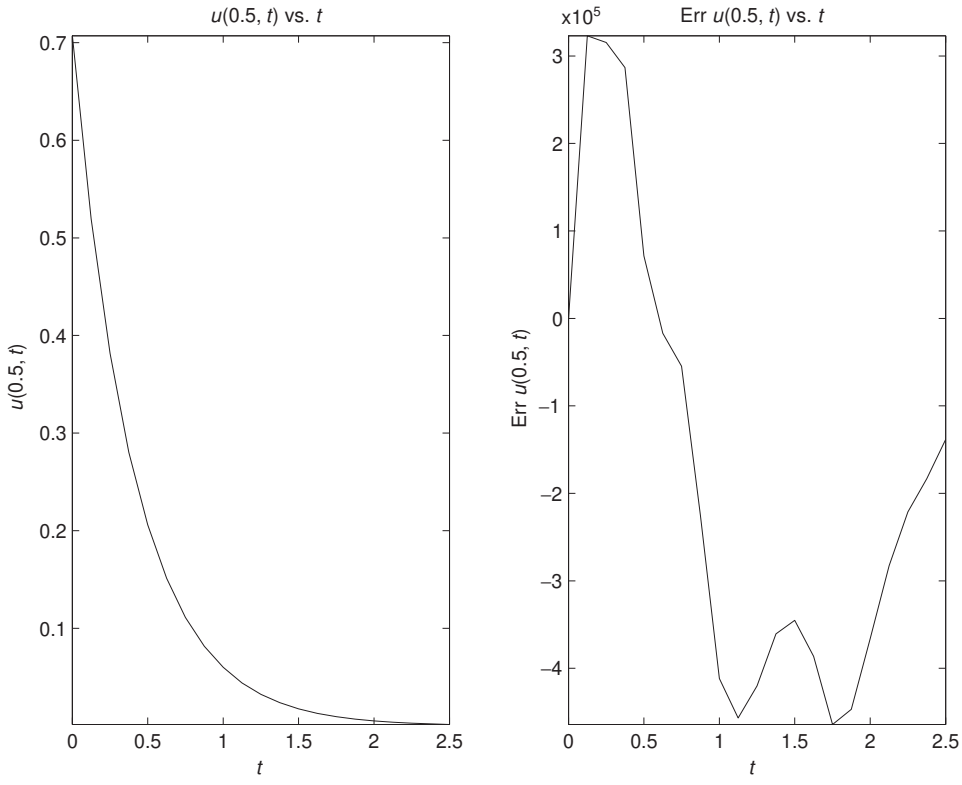

Figure 2.4. Two-dimensional graphical output from `pde_1_main`; `mf=3`

As the second derivative vanishes, the solution becomes

$$u_{xx} = 0$$

$$u_x = c_1$$

$$u = c_1 x + c_2$$

Thus, the steady-state solution is linear in x, which can serve as another check on the numerical solution (for BCs (2.3) and (2.4), $c_1 = c_2 = 0$ and thus at steady state, $u = 0$, which also follows from the analytical solution, Eq. (2.5)). This type of special case analysis is often useful in checking a numerical solution. In addition to mathematical conditions such as the linear dependency on x, physical conditions can frequently be used to check solutions, for example, conservation of mass, momentum, and energy.

Through this example application we have attempted to illustrate the basic steps of MOL/PDE analysis to arrive at a numerical solution of acceptable accuracy. We have also presented some basic ideas for assessing accuracy with respect to time and space (e.g., t and x). More advanced applications (e.g., problems expressed as systems of nonlinear PDEs) are considered in subsequent chapters.