10

Elliptic Partial Differential Equations: Laplace's Equation

This partial differential equation (PDE) application introduces the following mathematical concepts and computational methods:

1. Laplace's equation is
   (a) a classical PDE with many physical applications;
   (b) a PDE in two dimensions (2D), and therefore the following analysis demonstrates the method of lines (MOL) solution of 2D PDEs;
   (c) an elliptic PDE so that the following analysis demonstrates the application of the MOL to elliptic PDEs;
   (d) a basic PDE with a variety of extensions, for example, Poisson's equation and Helmholtz's equation.
2. Dirichlet and Neumann boundary conditions implemented on a 2D domain.
3. Evaluation of the accuracy of the numerical solution, including a comparison between the numerical and analytical solutions.
4. Application of *h*- and *p*-refinement to achieve spatial convergence of the numerical solution.
5. Introduction to the concept of continuation for the solution of elliptic PDEs and other important classes of problems.

Laplace's equation is

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{10.1}$$

where x and y are boundary-value (spatial) independent variables. In subscript notation, Eq. (10.1) is

$$u_{xx} + u_{yy} = 0$$

Equation (10.1) is classified as an *elliptic* PDE; it is a *boundary-value problem only* (since it does not have an initial-value independent variable).

Since the MOL generally involves the integration of a system of initial-value ODEs, it cannot be applied directly to Eq. (10.1) (again, Eq. (10.1) does not have an initial-value independent variable). Thus, to perform a MOL solution of Eq. (10.1), we add a derivative with respect to the initial-value variable t to form a *parabolic PDE*:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \tag{10.2}$$

Equation (10.2) is the *heat conduction equation*. It can be integrated to a *steady state* or *equilibrium condition*, corresponding to $t \to \infty$ for which $\partial u/\partial t \approx 0$, and then Eq. (10.2) reverts to Eq. (10.1).

Equation (10.2) requires one *"pseudo" initial condition* (IC) in t (since it is first order in t) and two boundary conditions (BCs) in x and y (since it is second order in x and y). We use the term "pseudo" because t in Eq. (10.2) is not part of the original problem of Eq. (10.1). Thus, we select the IC arbitrarily with the expectation that for $t \to \infty$, this IC will not determine the final equilibrium solution of Eq. (10.1); multiple solutions may be possible for different ICs, but we choose the IC as close as possible to the final solution of interest (admittedly, a rather imprecise procedure that can be tested by observing how the solution approaches the final equilibrium solution). For this purpose, we choose just a piecewise constant function, as explained subsequently.

$$u(x, y, t = 0) = u_0 \tag{10.3}$$

For the BCs in x, we use the *homogeneous Neumann conditions*

$$\frac{\partial u(x = 0, y, t)}{\partial x} = 0 \tag{10.4}$$

$$\frac{\partial u(x = 1, y, t)}{\partial x} = 0 \tag{10.5}$$

For the BCs in y, we use the *Dirichlet conditions*

$$u(x, y = 0, t) = 0 \tag{10.6}$$

$$u(x, y = 1, t) = 1 \tag{10.7}$$

Equations (10.2)–(10.7) for $t \to \infty$ have the analytical solution

$$u(x, y, t = \infty) = y \tag{10.8}$$

which will be used to evaluate the accuracy of the solution to Eq. (10.1) (subject to BCs (10.4)–(10.7).

A main program for the MOL solution of Eqs. (10.2)–(10.7) is given in Listing 10.1.

```
%
% Clear previous files
  clear all
  clc
%
% Parameters shared with the ODE routine
  global ncall nx ny ndss mmf

% Initial condition (which also is consistent with the
% boundary conditions)
  nx=11;
  ny=11;
  for i=1:nx
  for j=1:ny
    if(j==1)     u0(i,j)=0.0;
    elseif(j==ny)u0(i,j)=1.0;
    else         u0(i,j)=0.5;
    end
  end
  end
%
% Matrix conversion method flag
%
%   mmf = 1 - explicit subscripting for matrix conversion
%
%   mmf = 2 - Matlab reshape function
%
  mmf=2;
%
% 2D to 1D matrix conversion
%
  if(mmf==1)
    for i=1:nx
    for j=1:ny
      y0((i-1)*ny+j)=u0(i,j);
    end
    end
  end
%
  if(mmf==2)
    y0=reshape(u0',1,nx*ny);
  end
%
% Independent variable for ODE integration
  t0=0.0;
```

```
  tf=0.15;
  tout=[t0:0.03:tf]';
  nout=6;
  ncall=0;
%
% ODE integration
  mf=2;
  reltol=1.0e-04; abstol=1.0e-04;
  options=odeset('RelTol',reltol,'AbsTol',abstol);
  if(mf==1) % explicit FDs
    [t,y]=ode15s(@pde_1,tout,y0,options); end
  if(mf==2) ndss=4; % ndss = 2, 4, 6, 8 or 10 required
    [t,y]=ode15s(@pde_2,tout,y0,options); end
  if(mf==3) ndss=44; % ndss = 42, 44, 46, 48 or 50 required
    [t,y]=ode15s(@pde_3,tout,y0,options); end
%
% 1D to 2D matrix conversion (plus t), with 3D plotting at
% t = 0, 0.03, ..., 0.15
%
% Composite 3D plots showing evolution of solution
  figure(1);
  for it=1:nout
%
% 2D to 3D matrix conversion
    u(it,:,:)=reshape(y(it,:),ny,nx)';
    subplot(3,2,it)
    uview(:,:)=u(it,:,:);
    surf(uview);
  end
%
% 3D plot of initial condition
  figure(2);
  uview(:,:)=u(1,:,:);
  surf(uview);
  xlabel('y grid number');
  ylabel('x grid number');
  zlabel('u(x,y)');
  s1=sprintf('Laplace Equation - MOL Solution');
  s2=sprintf('Parametric plot at t=0 - initial condition');
  title([{s1}, {s2}], 'fontsize', 12);
  rotate3d on;
%
% 3D plot of final solution
  figure(3);
  uview(:,:)=u(nout,:,:);
  surf(uview);
  xlabel('y grid number');
```

```
  ylabel('x grid number');
  zlabel('u(x,y)');
  s1=sprintf('Laplace Equation - MOL Solution');
  s2=sprintf('Parametric plot at t=%4.2f - final solution', ...
             tout(nout));
  title([{s1}, {s2}], 'fontsize', 12);
  rotate3d on;
%
% Display selected output
  for it=1:nout
    fprintf('\n\n t = %5.2f',t(it));
    fprintf('\n x across (x=0,0.2,...,1.0)');
    fprintf('\n y  down  (y=1.0,0.8,...,0)');
  for j=ny:-2:1
    fprintf('\n%10.3f%10.3f%10.3f%10.3f%10.3f%10.3f', ...
            u(it,1:2:nx,j));
  end
  end
  fprintf('\n\n ncall = %4d\n',ncall);
```

Listing 10.1. Main program `pde_1_main.m`

We can note the following points about this program:

1. After some variables and parameters are declared *global* so that they can be shared with the MOL ordinary differential equation (ODE) routine, IC Eq. (10.3) is coded as $u(x, y, t = 0) = 0.5$ over a 11×11 grid in x and y; note, however, that along the boundary $y = 0$, $u(x, y = 0, t) = 0$, and along the boundary $y = 1$, $u(x, y = 1, t) = 1$, which is consistent with the BCs (Eqs. (10.6) and (10.7). In other words, we program the IC to be consistent with the BCs; the reason for this will be explained in the subsequent discussion.

```
%
% Clear previous files
  clear all
  clc
%
% Parameters shared with the ODE routine
  global ncall nx ny ndss mmf

% Initial condition (which also is consistent with the
% boundary conditions)
  nx=11;
  ny=11;
  for i=1:nx
```

```
for j=1:ny
  if(j==1)      u0(i,j)=0.0;
  elseif(j==ny)u0(i,j)=1.0;
  else          u0(i,j)=0.5;
  end
end
end
```

The choice of `nx=11` and `ny=11` and grid points in x and y was made rather arbitrarily. Thus we have the requirement to determine if these numbers are large enough to give sufficient accuracy in the MOL solution of Eq. (10.1). Of course, we should not choose these numbers of grid points to be larger than necessary to achieve acceptable accuracy since this would merely result in longer computer runs to achieve unnecessary accuracy in the numerical solution of Eq. (10.1). We can ascertain the adequacy of `nx=11` and `ny=1` in at least three ways:

(a) The numerical solution can be compared with the analytical solution (Eq. (10.8)) (although in general we will not have an analytical solution to evaluate the numerical solution).

(b) The numbers of grid points can be changed and the effect on the numerical solution can be observed (since the grid spacing in numerical approximations is often given the symbol "h," this is referred to as *h-refinement*).

(c) The order of the approximations of the derivatives in x and y in Eqs. (10.1) and (10.2) can be changed and the effect on the numerical solution observed (since the order of numerical approximations is often given the symbol "p," this is referred to as *p-refinement*).

2. Since library ODE integrators, such as the Matlab integrators, generally require all of the dependent variables to be arranged in a single 1D vector, we must convert the 2D IC matrix `u0(i,j)` ($= u(x, y, t = 0)$) of Eq. (10.2) into a 1D vector, in this case `y0`. This is done in two equivalent ways: (a) the explicit programming of the matrix subscripting (`mmf=1`) and (b) the use of a Matlab utility, `reshape`, (`mmf=2`).

```
%
% Matrix conversion method flag
%
%   mmf = 1 - explicit subscripting for matrix conversion
%
%   mmf = 2 - Matlab reshape function
%
  mmf=2;
%
% 2D to 1D matrix conversion
%
  if(mmf==1)
```

```
    for i=1:nx
    for j=1:ny
      y0((i-1)*ny+j)=u0(i,j);
    end
    end
  end
%
  if(mmf==2)
    y0=reshape(u0',1,nx*ny);
  end
```

The details of this conversion in a pair of nested `for` loops (`mmf=1`) should be studied since this is an essential operation for the solution of 2D PDEs (with straightforward extensions to 3D PDEs). This can be done by considering the successive passes through the two `for` loops: for example, for `i=1, j=1,2,...,ny; i=2, j=1,2,...,ny; ...;i=nx, j=1,2,...,ny`, so that there are $nx \times ny = (11)(11) = 121$ passes through the two `for` loops. In other words, for each increasing value of `i` in the outer `for` loop, `ny` values are assigned to `y0` through the inner `for` loop; thus, for `nx` passes through the outer `for` loop, 121 values are assigned to `y0`.

The Matlab `reshape` function allows the user to manipulate an array in order to change the number of rows and columns, but without changing the overall number of elements. Here we use the `reshape` function to change the 2D matrix `u0` to the 1D matrix (or vector) `y0`, which is then an input to the Matlab integrator `ode15s` (as discussed subsequently).

The relative merits of the two approaches for the 2D to 1D conversion are as follows:

(a) For `mmf=1`, the approach is general in the sense that it illustrates the basic operations for the 2D to 1D conversion, and can therefore be used in any procedural programming language. Also, this coding clarifies what takes place in `reshape`.
(b) For `mmf=1`, the execution is modestly more efficient (faster) than for `mmf=2`. This becomes clear when running the two cases (`mmf=1,2`).
(c) For `mmf=2`, the code is more compact (one line vs. six lines for `mmf=1`).

3. The total interval in t is defined ($0 \leq t \leq 0.15$). The solution is to be displayed six times, so the output interval is 0.03.

```
%
% Independent variable for ODE integration
  t0=0.0;
  tf=0.15;
  tout=[t0:0.03:tf]';
  nout=6;
  ncall=0;
```

4. The ODE integration (a total of 11×11 ODEs) is integrated by Matlab integrator `ode15s`; we chose a stiff integrator only because the MOL approximating ODEs are frequently stiff (although we did not test this idea by, for example, also using a nonstiff integrator and comparing the performance through the final value of `ncall`).

```
%
% ODE integration
  mf=2;
  reltol=1.0e-04; abstol=1.0e-04;
  options=odeset('RelTol',reltol,'AbsTol',abstol);
  if(mf==1) % explicit FDs
    [t,y]=ode15s(@pde_1,tout,y0,options); end
  if(mf==2) ndss=4; % ndss = 2, 4, 6, 8 or 10 required
    [t,y]=ode15s(@pde_2,tout,y0,options); end
  if(mf==3) ndss=44; % ndss = 42, 44, 46, 48 or 50 required
    [t,y]=ode15s(@pde_3,tout,y0,options); end
```

Three cases are programmed:

(a) `mf = 1`: Routine `pde_1` is called by `ode15s` with the explicit programming of the finite-difference (FDs) approximations of the derivatives $\partial^2 u/\partial x^2$ and $\partial^2 u/\partial y^2$ in Eqs. (10.1) and (10.2). The details of `pde_1` are discussed subsequently.

(b) `mf = 2`: Routine `pde_2` is called by `ode15s` with the calculation of FD approximations of the derivatives $\partial^2 u/\partial x^2$ and $\partial^2 u/\partial y^2$ in Eqs. (10.1) and (10.2) by a library routine for first-order derivatives, for example, `DSS004`. The second-order derivatives are computed by successive calculation of first-order derivatives (termed *stagewise differentiation*). The details of `pde_2` are discussed subsequently.

(c) `mf = 3`: Routine `pde_3` is called by `ode15s` with the calculation of FD approximations of the derivatives $\partial^2 u/\partial x^2$ and $\partial^2 u/\partial y^2$ in Eqs. (10.1) and (10.2) by a library routine for second-order derivatives, for example, `DSS044`. The details of `pde_3` are discussed subsequently.

5. After integration of the ODEs by `ode15s`, the solution is returned as a 2D matrix `y`; the two dimensions of `y` are for t with a first dimension of 6 and a combination of x and y with a second dimension of 121. This matrix is then expressed as a 2D matrix in x and y at a series of `nout=6` values of t through the index `it`, so that a parametric series of surface plots can be produced through the Matlab routine `surf`.

```
%
% 1D to 2D matrix conversion (plus t), with 3D plotting at
% t = 0, 0.03, ..., 0.15
```

```
%
% Composite 3D plots showing evolution of solution
  figure(1);
  for it=1:nout
%
% 2D to 3D matrix conversion
    u(it,:,:)=reshape(y(it,:),ny,nx)';
    subplot(3,2,it)
    uview(:,:)=u(it,:,:);
    surf(uview);
  end
%
% 3D plot of initial condition
  figure(2);
  uview(:,:)=u(1,:,:);
  surf(uview);
  xlabel('y grid number');
  ylabel('x grid number');
  zlabel('u(x,y)');
  s1=sprintf('Laplace Equation - MOL Solution');
  s2=sprintf('Parametric plot at t=0 - initial condition');
  title([{s1}, {s2}], 'fontsize', 12);
  rotate3d on;
%
% 3D plot of final solution
  figure(3);
  uview(:,:)=u(nout,:,:);
  surf(uview);
  xlabel('y grid number');
  ylabel('x grid number');
  zlabel('u(x,y)');
  s1=sprintf('Laplace Equation - MOL Solution');
  s2=sprintf('Parametric plot at t=%4.2f-final solution', ...
             tout(nout));
  title([{s1}, {s2}], 'fontsize', 12);
  rotate3d on;
```

6. Additionally, tabular output is displayed so that the numerical solution can be compared with the analytical solution (Eq. (10.8)).

```
%
% Display selected output
  for it=1:nout
    fprintf('\n\n t = %5.2f',t(it));
```

```
  fprintf('\n x across (x=0,0.2,...,1.0)');
  fprintf('\n y  down  (y=1.0,0.8,...,0)');
for j=ny:-2:1
  fprintf('\n%10.3f%10.3f%10.3f%10.3f%10.3f%10.3f',...
          u(it,1:2:nx,j));
end
end
fprintf('\n\n ncall = %4d\n',ncall);
```

The three ODE routines called by `ode15s`, `pde_1.m`, `pde_2.m`, `pde_3.m`, are now discussed. A listing of `pde_1.m` is given in Listing 10.2.

```
  function yt=pde_1(t,y)
%
% Function pde_1 computes the temporal derivative in the
% pseudo transient solution of Laplace's equation by explicit
% finite differences
%
% Problem parameters
  global ncall nx ny mmf
  xl=0.0;
  xu=1.0;
  yl=0.0;
  yu=1.0;
%
% Initially zero derivatives in x, y, t
  uxx=zeros(nx,ny); uyy=zeros(nx,ny); ut=zeros(nx,ny);
%
% 1D to 2D matrix conversion
  if(mmf==1)
    for i=1:nx
    for j=1:ny
      u(i,j)=y((i-1)*ny+j);
    end
    end
  end
  if(mmf==2)
    u=reshape(y,ny,nx)';
  end
%
% PDE
  dx2=((xu-xl)/(nx-1))^2;
  dy2=((yu-yl)/(ny-1))^2;
  for i=1:nx
```

```
  for j=1:ny
%
%   uxx
    if(i==1)      uxx(i,j)=2.0*(u(i+1,j)-u(i,j))/dx2;
    elseif(i==nx) uxx(i,j)=2.0*(u(i-1,j)-u(i,j))/dx2;
    else          uxx(i,j)=(u(i+1,j)-2.0*u(i,j)+u(i-1,j))/dx2;
    end
%
%   uyy
    if(j~=1)&(j~=ny)
       uyy(i,j)=(u(i,j+1)-2.0*u(i,j)+u(i,j-1))/dy2;
    end
%
%   ut = uxx + uyy
    if(j==1)|(j==ny)ut(i,j)=0.0;
    else ut(i,j)=uxx(i,j)+uyy(i,j);
    end
  end
  end
%
% 2D to 1D matrix conversion
  if(mmf==1)
    for i=1:nx
    for j=1:ny
      yt((i-1)*ny+j)=ut(i,j);
    end
    end
    yt=yt';
  end
  if(mmf==2)
    yt=reshape(ut',nx*ny,1);
  end
%
% Increment calls to pde_1
  ncall=ncall+1;
```

Listing 10.2. ODE routine `pde_1.m`

We can note the following points about `pde_1.m`:

1. The definition of `pde_1`, `function yt=pde_1(t,y)`, indicates two important points:
   (a) The input arguments, `t,y`, the ODE system independent variable, and dependent-variable vector, `t` and `y` respectively, are available for the programming in `pde_1`.
   (b) The output argument, `yt`, is the vector of dependent-variable derivatives (with respect to `t`) that must be defined numerically before the execution of `pde_1` is completed. Thus, if `y` is of length `n`, then all of the `n` elements of

yt must be defined (failing to evaluate even one of these derivatives will generally produce an incorrect solution; this may seem obvious, but since in this case there are $n = 121$ dependent variables, failure to evaluate all of the derivatives in yt can easily happen). Some problem parameters are then declared as *global* so that they can be shared with pde_1_main in Listing 10.1, or other routines. In this case, the dimensions of the 2D $x - y$ domain are defined. We also set all of the PDE derivatives to zero (via the Matlab utility zeros) to minimize the effect of failing to set any of these derivatives.

```
  function yt=pde_1(t,y)
%
% Function pde_1 computes the temporal derivative in the
% pseudo transient solution of Laplace's equation by explicit
% finite differences
%
% Problem parameters
  global ncall nx ny mmf
  xl=0.0;
  xu=1.0;
  yl=0.0;
  yu=1.0;
%
% Initially zero derivatives in x, y, t
  uxx=zeros(nx,ny); uyy=zeros(nx,ny); ut=zeros(nx,ny);
```

2. A conversion of the 1D vector, y, to the 2D matrix u is made so that the subsequent programming can be done in terms of the problem-oriented variable u in Eqs. (10.1) and (10.2). Although this conversion is, in principle, not required, programming in terms of problem-oriented variables, for example, u, is a major convenience. Again, we follow two approaches: (a) explicit programming of the conversion and (b) use of the Matlab utility reshape, depending on the value of mmf set in the main program of Listing 10.1.

```
%
% 1D to 2D matrix conversion
  if(mmf==1)
    for i=1:nx
    for j=1:ny
      u(i,j)=y((i-1)*ny+j);
    end
    end
```

```
  end
  if(mmf==2)
    u=reshape(y,ny,nx)';
  end
```

3. The programming of the MOL ODEs is next, starting with the square of the increments for the FD approximations in x and y.

```
%
% PDE
  dx2=((xu-xl)/(nx-1))^2;
  dy2=((yu-yl)/(ny-1))^2;
```

4. We then move throughout the $x - y$ domain using a pair of nested `for` loops. Each pass through this pair of `for` loops executes the programming for another ODE (a total of 121 passes or ODEs).

```
  for i=1:nx
  for j=1:ny
%
%   uxx
    if(i==1)      uxx(i,j)=2.0*(u(i+1,j)-u(i,j))/dx2;
    elseif(i==nx) uxx(i,j)=2.0*(u(i-1,j)-u(i,j))/dx2;
    else          uxx(i,j)=(u(i+1,j)-2.0*u(i,j)+u(i-1,j))/dx2;
    end
%
%   uyy
    if(j~=1)&(j~=ny)
       uyy(i,j)=(u(i,j+1)-2.0*u(i,j)+u(i,j-1))/dy2;
    end
%
%   ut = uxx + uyy
    if(j==1)|(j==ny)ut(i,j)=0.0;
    else ut(i,j)=uxx(i,j)+uyy(i,j);
    end
  end
  end
```

The coding of the FD approximations requires some explanation. For the derivative $\partial^2 u/\partial x^2$ = uxx, three cases are required:

(a) For $x = 0$, BC (10.4) is applied, so the FD approximation of $\partial^2 u(x = 0, y)/\partial x^2$ is

```
if(i==1)uxx(i,j)=2.0*(u(i+1,j)-u(i,j))/dx2;
```

In other words, the BC is applied as `u(0,j) = u(2,j)` to eliminate the fictitious value `u(0,j)`.

(b) For $x = 1$, BC (10.5) is applied, so the FD approximation of $\partial^2 u(x = 1, y)/\partial x^2$ is

```
elseif(i==nx)uxx(i,j)=2.0*(u(i-1,j)-u(i,j))/dx2;
```

The BC is applied as `u(nx+1,j) = u(nx-1,j)` to eliminate the fictitious value `u(nx+1,j)`.

(c) For the interior points `i = 2` to `i = nx-1`, the FD approximation of the second derivative $\partial^2 u(x, y)/\partial x^2$ is

```
else uxx(i,j)=(u(i+1,j)-2.0*u(i,j)+u(i-1,j))/dx2;
```

Similarly, the coding of the FD approximation of $\partial^2 u/\partial y^2$ = `uyy` requires some explanation.

(a) For $\partial^2 u/\partial y^2$ at $y \neq 0, 1$, the coding is

```
if(j~=1)&(j~=ny)uyy(i,j)=(u(i,j+1)-2.0*u(i,j)+u(i,j-1))/dy2; end
```

that is, the usual FD approximation for a second-order derivative.

(b) For $y = 0, 1$, BCs (10.6) and (10.7) apply. The constant values $u(x, y = 0, t) = 0$, $u(x, y = 1, t) = 1$ are implemented as $\partial u(x, y = 0, t)/\partial t = 0$, $\partial u(x, y = 1, t)/\partial t = 0$ or `if(j==1)|(j==ny)ut(i,j)=0.0;`

(c) Thus, the values of $u(x, y = 0, t) = 0$, $u(x, y = 1, t) = 1$ from BCs (10.6) and (10.7) set in the coding of IC (10.3) (see this coding in `pde_1_main` in Listing 10.1) remain at their prescribed values in `pde_1` (as reflected in the zero derivatives in t). This is the rationalization for making the ICs and BCs consistent. To explain a bit further, ODE integrator `ode15s` does not permit setting the dependent variables in `pde_1`, for example,

```
if(j==1) u(i,j)=0.0;
if(j==ny)u(i,j)=1.0;
```

If this coding were used, *it would have no effect*. In other words, the *dependent variables can be defined only in terms of their derivatives* in `pde_1`, for example, `if(j==1)|(j==ny)ut(i,j)=0.0;`

(d) For the remaining grid points for which special conditions are not imposed, Eq. (10.2) is programmed as `else ut(i,j)=uxx(i,j)+uyy(i,j);`. The resemblance of this code to Eq. (10.2) is clear.

5. Finally, the 2D matrix `ut` is converted to a 1D vector required by `ode15s`, and since `ode15s` requires a column derivative vector, a transpose of `ut` is added at the end for `mmf=1`. The counter for the calls to `pde_1`, `ncall` is incremented (the final value of this counter at the end of the solution is displayed by `pde_1_main` and gives an indication of the computational effort required to produce the solution).

```
%
% 2D to 1D matrix conversion
  if(mmf==1)
    for i=1:nx
    for j=1:ny
      yt((i-1)*ny+j)=ut(i,j);
    end
    end
    yt=yt';
  end
  if(mmf==2)
    yt=reshape(ut',nx*ny,1);
  end
%
% Increment calls to pde_1
  ncall=ncall+1;
```

The numerical output from `pde_1_main` of Listing 10.1 and `pde_1` of Listing 10.2 for `mf=1`, mmf=1 is given in Table 10.1. The approach to the analytical solution of Eq. (10.8) for large t is clear (if the code runs to a larger final time (`tf > 0.15`), the numerical solution agrees with Eq. (10.8) to four figures).

Figure 10.1 produced by `surf` indicates how the solution evolves with each iteration. The intermediate figures at $t = 0.03, 0.06, \ldots, 0.012$ show clearly the transition from IC (10.3) (the piecewise constant function) to the final solution of Eq. (10.8) as the integration proceeds.

In addition, for clarity, the first and last figures produced by `surf` are indicated in Figure 10.2 (corresponding to `t = 0` and `t = 0.15`). The IC of Figure 10.2a programmed in `pde_1_main` of Listing 10.1, is in Figure 10.2a and the final solution of Eq. (10.8) is in Figure 10.2b.

Table 10.1. Numerical output for the solution of Eqs. (10.2)–(10.7) from `pde_1`, with `mf=1`, `mmf=1`

```
t =  0.00
x across (x=0,0.2,...,1.0)
y  down  (y=1.0,0.8,...,0)
    1.000     1.000     1.000     1.000     1.000     1.000
    0.500     0.500     0.500     0.500     0.500     0.500
    0.500     0.500     0.500     0.500     0.500     0.500
    0.500     0.500     0.500     0.500     0.500     0.500
    0.500     0.500     0.500     0.500     0.500     0.500
    0.000     0.000     0.000     0.000     0.000     0.000

t =  0.03
x across (x=0,0.2,...,1.0)
y  down  (y=1.0,0.8,...,0)
    1.000     1.000     1.000     1.000     1.000     1.000
    0.706     0.706     0.706     0.706     0.706     0.706
    0.545     0.545     0.545     0.545     0.545     0.545
    0.455     0.455     0.455     0.455     0.455     0.455
    0.294     0.294     0.294     0.294     0.294     0.294
    0.000     0.000     0.000     0.000     0.000     0.000

t =  0.06
x across (x=0,0.2,...,1.0)
y  down  (y=1.0,0.8,...,0)
    1.000     1.000     1.000     1.000     1.000     1.000
    0.770     0.770     0.770     0.770     0.770     0.770
    0.582     0.582     0.582     0.582     0.582     0.582
    0.418     0.418     0.418     0.418     0.418     0.418
    0.230     0.230     0.230     0.230     0.230     0.230
    0.000     0.000     0.000     0.000     0.000     0.000

t =  0.09
x across (x=0,0.2,...,1.0)
y  down  (y=1.0,0.8,...,0)
    1.000     1.000     1.000     1.000     1.000     1.000
    0.791     0.791     0.791     0.791     0.791     0.791
    0.594     0.594     0.594     0.594     0.594     0.594
    0.406     0.406     0.406     0.406     0.406     0.406
    0.209     0.209     0.209     0.209     0.209     0.209
    0.000     0.000     0.000     0.000     0.000     0.000

t =  0.12
x across (x=0,0.2,...,1.0)
y  down  (y=1.0,0.8,...,0)
    1.000     1.000     1.000     1.000     1.000     1.000
```

```
    0.797     0.797     0.797     0.797     0.797     0.797
    0.598     0.598     0.598     0.598     0.598     0.598
    0.402     0.402     0.402     0.402     0.402     0.402
    0.203     0.203     0.203     0.203     0.203     0.203
    0.000     0.000     0.000     0.000     0.000     0.000

t =  0.15
x across (x=0,0.2,...,1.0)
y  down  (y=1.0,0.8,...,0)
    1.000     1.000     1.000     1.000     1.000     1.000
    0.799     0.799     0.799     0.799     0.799     0.799
    0.599     0.599     0.599     0.599     0.599     0.599
    0.401     0.401     0.401     0.401     0.401     0.401
    0.201     0.201     0.201     0.201     0.201     0.201
    0.000     0.000     0.000     0.000     0.000     0.000

ncall =  203
```

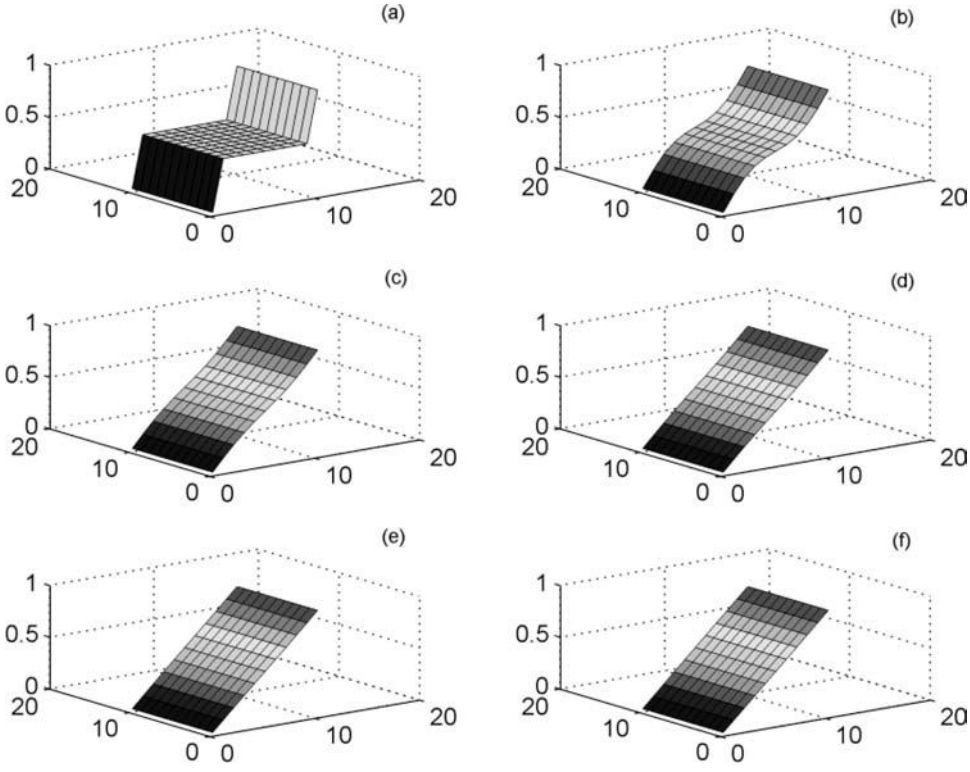
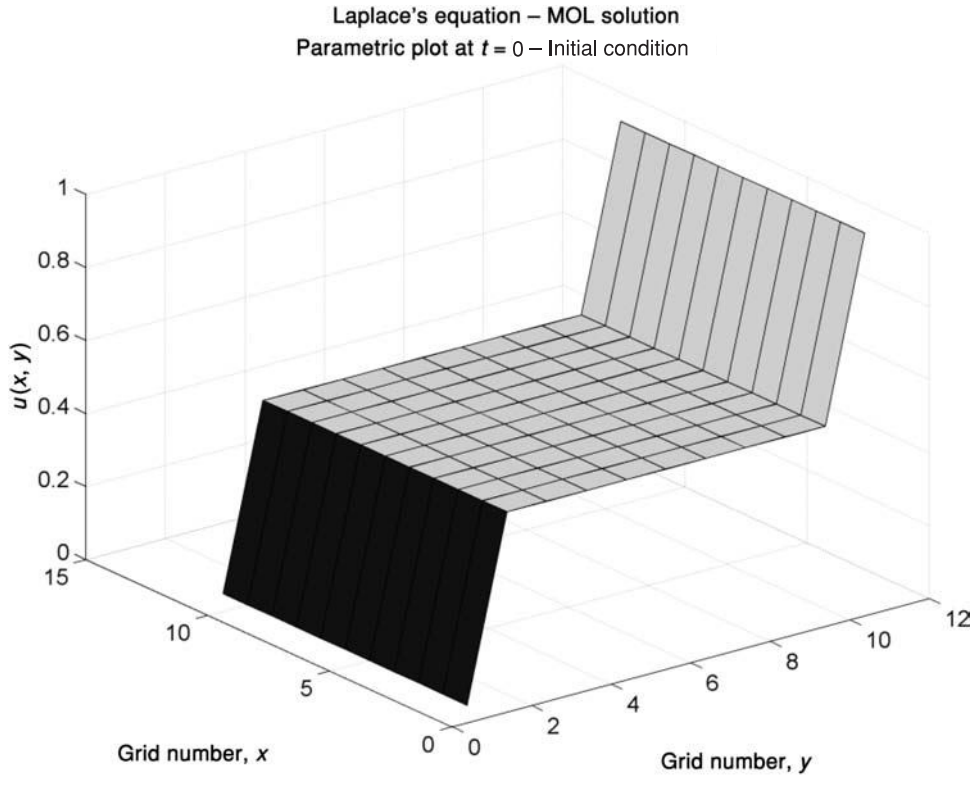
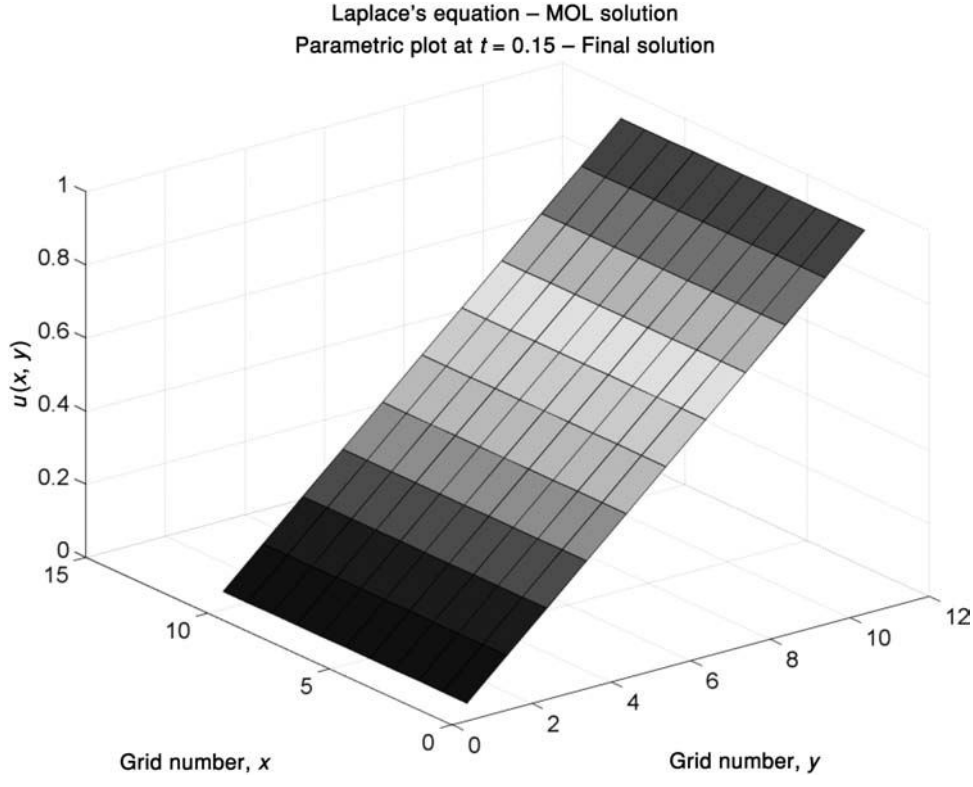


Figure 10.1. Output of pde_1_main and pde1_1 as the solution evolves. The individual plots correspond to (a) $t = 0$; (b) $t = 0.03$; (c) $t = 0.06$; (d) $t = 0.09$; (e) $t = 0.0.12$; (f) $t = 0.15$



(a)



(b)

Figure 10.2. (a) Output of pde_1_main and pde1_1 at $t = 0$; (b) output of pde_1_main and pde1_1 at $t = 0.15$

For mf=2 in pde_1_main, pde_2 is the ODE routine called by ode15s (See Listing 10.3).

```
  function yt=pde_2(t,y)
%
% Function pde_2 computes the temporal derivative in the
% pseudo transient solution of Laplace's equation by
% stagewise differentiation
%
%
% Problem parameters
  global ncall nx ny ndss mmf
  xl=0.0;
  xu=1.0;
  yl=0.0;
  yu=1.0;
%
% 1D to 2D matrix conversion
  if(mmf==1)
    for i=1:nx
    for j=1:ny
      u(i,j)=y((i-1)*ny+j);
    end
    end
  end
  if(mmf==2)
    u=reshape(y,ny,nx)';
  end
%
% PDE
%
% ux
  for j=1:ny
    u1d=u(:,j);
    if    (ndss== 2) ux1d=dss002(xl,xu,nx,u1d); % second order
    elseif(ndss== 4) ux1d=dss004(xl,xu,nx,u1d); % fourth order
    elseif(ndss== 6) ux1d=dss006(xl,xu,nx,u1d); % sixth order
    elseif(ndss== 8) ux1d=dss008(xl,xu,nx,u1d); % eighth order
    elseif(ndss==10) ux1d=dss010(xl,xu,nx,u1d); % tenth order
    end
%
% uxx
    ux1d(1) =0.0;
    ux1d(nx)=0.0;
    if    (ndss== 2) uxx1d=dss002(xl,xu,nx,ux1d); % second order
    elseif(ndss== 4) uxx1d=dss004(xl,xu,nx,ux1d); % fourth order
```

```
    elseif(ndss== 6) uxx1d=dss006(xl,xu,nx,ux1d); % sixth order
    elseif(ndss== 8) uxx1d=dss008(xl,xu,nx,ux1d); % eighth order
    elseif(ndss==10) uxx1d=dss010(xl,xu,nx,ux1d); % tenth order
    end
%
% 1D to 2D
    uxx(:,j)=uxx1d(:);
  end
%
% uy
  for i=1:nx
    u1d=u(i,:);
    if    (ndss== 2) uy1d=dss002(yl,yu,ny,u1d); % second order
    elseif(ndss== 4) uy1d=dss004(yl,yu,ny,u1d); % fourth order
    elseif(ndss== 6) uy1d=dss006(yl,yu,ny,u1d); % sixth order
    elseif(ndss== 8) uy1d=dss008(yl,yu,ny,u1d); % eighth order
    elseif(ndss==10) uy1d=dss010(yl,yu,ny,u1d); % tenth order
    end
%
% uyy
    if    (ndss== 2) uyy1d=dss002(yl,yu,ny,uy1d); % second order
    elseif(ndss== 4) uyy1d=dss004(yl,yu,ny,uy1d); % fourth order
    elseif(ndss== 6) uyy1d=dss006(yl,yu,ny,uy1d); % sixth order
    elseif(ndss== 8) uyy1d=dss008(yl,yu,ny,uy1d); % eighth order
    elseif(ndss==10) uyy1d=dss010(yl,yu,ny,uy1d); % tenth order
    end
%
% 1D to 2D
    uyy(i,:)=uyy1d(:);
  end
%
% ut = uxx + uyy
  ut=uxx+uyy;
  ut(:,1) =0.0;
  ut(:,ny)=0.0;
%
% 2D to 1D matrix conversion
  if(mmf==1)
    for i=1:nx
    for j=1:ny
      yt((i-1)*ny+j)=ut(i,j);
    end
    end
    yt=yt';
  end
  if(mmf==2)
```

```
    yt=reshape(ut',nx*ny,1);
  end
%
% Increment calls to pde_2
  ncall=ncall+1;
```

Listing 10.3. ODE routine `pde_2.m`

We can note the following details about `pde_2`:

1. The function is defined, selected parameters and variables are declared as global, the spatial dimensions are defined, and the conversion of the 1D matrix `y` to the 2D matrix `u` is made so that programming in terms of problem-oriented variables is set up.

```
  function yt=pde_2(t,y)
%
% Function pde_2 computes the temporal derivative in the
% pseudo transient solution of Laplace's equation by stagewise
% differentiation
%
%
% Problem parameters
  global ncall nx ny ndss mmf
  xl=0.0;
  xu=1.0;
  yl=0.0;
  yu=1.0;
```

2. A 1D to 2D conversion allows programming in terms of the problem-oriented dependent variable, `u`.

```
%
% 1D to 2D matrix conversion
  if(mmf==1)
    for i=1:nx
    for j=1:ny
      u(i,j)=y((i-1)*ny+j);
    end
    end
  end
  if(mmf==2)
    u=reshape(y,ny,nx)';
  end
```

3. The calculation of the partial derivative in x, u_{xx}, is computed by calling one of a group of five differentiation in space (DSS) routines (which compute first-order derivatives). In this case, dss004 is selected by setting ndss=4 in pde_1_main (with mf=2).

```
%
% PDE
%
% ux
  for j=1:ny
    u1d=u(:,j);
    if    (ndss== 2) ux1d=dss002(xl,xu,nx,u1d); % second order
    elseif(ndss== 4) ux1d=dss004(xl,xu,nx,u1d); % fourth order
    elseif(ndss== 6) ux1d=dss006(xl,xu,nx,u1d); % sixth order
    elseif(ndss== 8) ux1d=dss008(xl,xu,nx,u1d); % eighth order
    elseif(ndss==10) ux1d=dss010(xl,xu,nx,u1d); % tenth order
    end
%
% uxx
    ux1d(1) =0.0;
    ux1d(nx)=0.0;
    if    (ndss== 2) uxx1d=dss002(xl,xu,nx,ux1d); % second order
    elseif(ndss== 4) uxx1d=dss004(xl,xu,nx,ux1d); % fourth order
    elseif(ndss== 6) uxx1d=dss006(xl,xu,nx,ux1d); % sixth order
    elseif(ndss== 8) uxx1d=dss008(xl,xu,nx,ux1d); % eighth order
    elseif(ndss==10) uxx1d=dss010(xl,xu,nx,ux1d); % tenth order
    end
%
% 1D to 2D
    uxx(:,j)=uxx1d(:);
  end
```

Since these DSS routines compute numerical derivatives of 1D arrays, it is necessary before calling them to temporarily store the 2D array u in a 1D array, u1d, which is easily done using the subscripting utilities of Matlab (in this case, the : operator). After the first derivative of u1d is computed (by a call to dss004), the boundary values are reset in accordance with Eqs. (10.4) and (10.5).

Then a second call to dss004 computes the second derivative uxx1d, again a 1D array. uxx1d is returned to a 2D array, uxx, for subsequent MOL programming of Eq. (10.2). Note in particular the use of *stagewise differentiation* using $u \rightarrow u_x \rightarrow u_{xx}$ by two successive calls to dss004.

4. The same procedure is then repeated for the derivative in y, u_{yy}, which eventually is stored in uyy.

```
%
% uy
  for i=1:nx
    u1d=u(i,:);
    if    (ndss== 2) uy1d=dss002(yl,yu,ny,u1d); % second order
    elseif(ndss== 4) uy1d=dss004(yl,yu,ny,u1d); % fourth order
    elseif(ndss== 6) uy1d=dss006(yl,yu,ny,u1d); % sixth order
    elseif(ndss== 8) uy1d=dss008(yl,yu,ny,u1d); % eighth order
    elseif(ndss==10) uy1d=dss010(yl,yu,ny,u1d); % tenth order
    end
%
% uyy
    if    (ndss== 2) uyy1d=dss002(yl,yu,ny,uy1d); % second order
    elseif(ndss== 4) uyy1d=dss004(yl,yu,ny,uy1d); % fourth order
    elseif(ndss== 6) uyy1d=dss006(yl,yu,ny,uy1d); % sixth order
    elseif(ndss== 8) uyy1d=dss008(yl,yu,ny,uy1d); % eighth order
    elseif(ndss==10) uyy1d=dss010(yl,yu,ny,uy1d); % tenth order
    end
%
% 1D to 2D
    uyy(i,:)=uyy1d(:);
  end
```

Table 10.2. Numerical output for the solution of Eqs. (10.2)–(10.7) from `pde_1.m` and `pde_2`, `mf=2`, `mmf=1`

```
t =  0.00
x across (x=0,0.2,...,1.0)
y  down  (y=1.0,0.8,...,0)
    1.000     1.000     1.000     1.000     1.000     1.000
    0.500     0.500     0.500     0.500     0.500     0.500
    0.500     0.500     0.500     0.500     0.500     0.500
    0.500     0.500     0.500     0.500     0.500     0.500
    0.500     0.500     0.500     0.500     0.500     0.500
    0.000     0.000     0.000     0.000     0.000     0.000

t =  0.03
x across (x=0,0.2,...,1.0)
y  down  (y=1.0,0.8,...,0)
    1.000     1.000     1.000     1.000     1.000     1.000
    0.706     0.706     0.706     0.706     0.706     0.706
    0.544     0.544     0.544     0.544     0.544     0.544
```

(*continued*)

Table 10.2 (*continued*)

```
    0.456     0.456     0.456     0.456     0.456     0.456
    0.294     0.294     0.294     0.294     0.294     0.294
    0.000     0.000     0.000     0.000     0.000     0.000

t =  0.06
x across (x=0,0.2,...,1.0)
y  down  (y=1.0,0.8,...,0)
    1.000     1.000     1.000     1.000     1.000     1.000
    0.772     0.772     0.772     0.772     0.772     0.772
    0.582     0.582     0.582     0.582     0.582     0.582
    0.418     0.418     0.418     0.418     0.418     0.418
    0.228     0.228     0.228     0.228     0.228     0.228
    0.000     0.000     0.000     0.000     0.000     0.000

t =  0.09
x across (x=0,0.2,...,1.0)
y  down  (y=1.0,0.8,...,0)
    1.000     1.000     1.000     1.000     1.000     1.000
    0.791     0.791     0.791     0.791     0.791     0.791
    0.595     0.595     0.595     0.595     0.595     0.595
    0.405     0.405     0.405     0.405     0.405     0.405
    0.209     0.209     0.209     0.209     0.209     0.209
    0.000     0.000     0.000     0.000     0.000     0.000

t =  0.12
x across (x=0,0.2,...,1.0)
y  down  (y=1.0,0.8,...,0)
    1.000     1.000     1.000     1.000     1.000     1.000
    0.797     0.797     0.797     0.797     0.797     0.797
    0.598     0.598     0.598     0.598     0.598     0.598
    0.402     0.402     0.402     0.402     0.402     0.402
    0.203     0.203     0.203     0.203     0.203     0.203
    0.000     0.000     0.000     0.000     0.000     0.000

t =  0.15
x across (x=0,0.2,...,1.0)
y  down  (y=1.0,0.8,...,0)
    1.000     1.000     1.000     1.000     1.000     1.000
    0.799     0.799     0.799     0.799     0.799     0.799
    0.599     0.599     0.599     0.599     0.599     0.599
    0.401     0.401     0.401     0.401     0.401     0.401
    0.201     0.201     0.201     0.201     0.201     0.201
    0.000     0.000     0.000     0.000     0.000     0.000

ncall =  170
```

5. With the two partial derivatives in the RHS of Eq. (10.2) now available, this PDE can be programmed.

```
%
% ut = uxx + uyy
  ut=uxx+uyy;
  ut(:,1) =0.0;
  ut(:,ny)=0.0;
```

Note the application of the BCs in y (Eqs. (10.6) and (10.7)).

6. Finally, a 2D to 1D conversion produces the derivative vector `yt`, which is then transposed as required by `ode15s`. The counter for calls to `pde_1` is incremented at the end of `pde_1`.

```
%
% 2D to 1D matrix conversion
  if(mmf==1)
    for i=1:nx
    for j=1:ny
      yt((i-1)*ny+j)=ut(i,j);
    end
    end
    yt=yt';
  end
  if(mmf==2)
    yt=reshape(ut',nx*ny,1);
  end
%
% Increment calls to pde_2
  ncall=ncall+1;
```

The numerical output from `pde_1_main` and `pde_2` with `mf=2`, `mmf=1` is given in Table 10.2. This output is very similar to that for `mf=1` in Table 10.1. Again, the agreement with the analytical solution, Eq. (10.8), is evident. Also, any possible advantage of using the fourth-order FDs in `dss004` (for `mf=2`) over the second-order FDs in `pde_1` (for `mf=1`) is not evident since the final solution is so smooth; specifically, the solution is linear in y and constant in x, so that second- and fourth-order FDs are exact. However, in general, using higher-order FDs for the same gridding produces more accurate solutions (as demonstrated in other PDE chapters).

For `mf=3` in `pde_1_main`, `pde_3` is the ODE routine called by `ode15s` (See Listing 10.4).

```
  function yt=pde_3(t,y)
%
% Function pde_3 computes the temporal derivative in the
% pseudo transient solution of Laplace's equation by library
% routines for the second derivative
%
% Problem parameters
  global ncall nx ny ndss mmf
  xl=0.0;
  xu=1.0;
  yl=0.0;
  yu=1.0;
%
% 1D to 2D matrix conversion
  if(mmf==1)
    for i=1:nx
    for j=1:ny
      u(i,j)=y((i-1)*ny+j);
    end
    end
  end
  if(mmf==2)
    u=reshape(y,ny,nx)';
  end
%
% PDE
%
% uxx
  for j=1:ny
    u1d=u(:,j);
    nl=2; % Neumann
    nu=2; % Neumann
    ux1d(1) =0.0;
    ux1d(nx)=0.0;
    if    (ndss==42) uxx1d=dss042(xl,xu,nx,u1d,ux1d,nl,nu);
    % second order
    elseif(ndss==44) uxx1d=dss044(xl,xu,nx,u1d,ux1d,nl,nu);
    % fourth order
    elseif(ndss==46) uxx1d=dss046(xl,xu,nx,u1d,ux1d,nl,nu);
    % sixth order
    elseif(ndss==48) uxx1d=dss048(xl,xu,nx,u1d,ux1d,nl,nu);
    % eighth order
    elseif(ndss==50) uxx1d=dss050(xl,xu,nx,u1d,ux1d,nl,nu);
    % tenth order
    end
%
% 1D to 2D
    uxx(:,j)=uxx1d(:);
  end
```

```
%
% uyy
  for i=1:nx
    u1d=u(i,:);
    nl=1; % Dirichlet
    nu=1; % Dirichlet
    uy1d(:)=0.0;
    if    (ndss==42) uyy1d=dss042(yl,yu,ny,u1d,uy1d,nl,nu);
    % second order
    elseif(ndss==44) uyy1d=dss044(yl,yu,ny,u1d,uy1d,nl,nu);
    % fourth order
    elseif(ndss==46) uyy1d=dss046(yl,yu,ny,u1d,uy1d,nl,nu);
    % sixth order
    elseif(ndss==48) uyy1d=dss048(yl,yu,ny,u1d,uy1d,nl,nu);
    % eighth order
    elseif(ndss==50) uyy1d=dss050(yl,yu,ny,u1d,uy1d,nl,nu);
    % tenth order
    end
%
% 1D to 2D
    uyy(i,:)=uyy1d(:);
  end
%
% ut = uxx + uyy
  ut=uxx+uyy;
  ut(:,1) =0.0;
  ut(:,ny)=0.0;
%
% 2D to 1D matrix conversion
  if(mmf==1)
    for i=1:nx
    for j=1:ny
      yt((i-1)*ny+j)=ut(i,j);
    end
    end
    yt=yt';
  end
  if(mmf==2)
    yt=reshape(ut',nx*ny,1);
  end
%
% Increment calls to pde_3
  ncall=ncall+1;
```

Listing 10.4. ODE routine `pde_3.m`

We can note the following points about pde_3:

1. The beginning code is to declare variables as global, set the dimensions of the $x - y$ grid, and convert the 1D matrix (vector) y to the 2D matrix u is the same as in pde_1 and pde_2.

```
  function yt=pde_3(t,y)
%
% Function pde_3 computes the temporal derivative in the
% pseudo transient solution of Laplace's equation by library
% routines for the second derivative
%
% Problem parameters
  global ncall nx ny ndss mmf
  xl=0.0;
  xu=1.0;
  yl=0.0;
  yu=1.0;
%
% 1D to 2D matrix conversion
  if(mmf==1)
    for i=1:nx
    for j=1:ny
      u(i,j)=y((i-1)*ny+j);
    end
    end
  end
  if(mmf==2)
    u=reshape(y,ny,nx)';
  end
```

2. The spatial derivative u_{xx} is now calculated by dss044 (ndss=44 with mf=3 in pde_1_main).

```
%
% PDE
%
% uxx
  for j=1:ny
    u1d=u(:,j);
    nl=2; % Neumann
    nu=2; % Neumann
```

```
    ux1d(1) =0.0;
    ux1d(nx)=0.0;
    if    (ndss==42) uxx1d=dss042(xl,xu,nx,u1d,ux1d,nl,nu);
    % second order
    elseif(ndss==44) uxx1d=dss044(xl,xu,nx,u1d,ux1d,nl,nu);
    % fourth order
    elseif(ndss==46) uxx1d=dss046(xl,xu,nx,u1d,ux1d,nl,nu);
    % sixth order
    elseif(ndss==48) uxx1d=dss048(xl,xu,nx,u1d,ux1d,nl,nu);
    % eighth order
    elseif(ndss==50) uxx1d=dss050(xl,xu,nx,u1d,ux1d,nl,nu);
    % tenth order
    end
%
% 1D to 2D
    uxx(:,j)=uxx1d(:);
  end
```

Note that since BCs (10.4) and (10.5) are Neumann, `nl=nu=2` are inputs to `dss044` corresponding to $x = 0, 1$, respectively. Also, BCs (10.4) and (10.5) are set as the boundary derivatives $u_x(x = 0, t) = u_x(x = 1, t) = 0$ as inputs to `dss044`. Then `dss044` returns the second derivative directly in `uxx1d`.

3. The code is then essentially the same for u_{yy}. The exception are the Dirichlet BCs at $y = 0, 1$ for which `nl=nu=1`.

```
%
% uyy
  for i=1:nx
    u1d=u(i,:);
    nl=1; % Dirichlet
    nu=1; % Dirichlet
    uy1d(:)=0.0;
    if    (ndss==42) uyy1d=dss042(yl,yu,ny,u1d,uy1d,nl,nu);
    % second order
    elseif(ndss==44) uyy1d=dss044(yl,yu,ny,u1d,uy1d,nl,nu);
    % fourth order
    elseif(ndss==46) uyy1d=dss046(yl,yu,ny,u1d,uy1d,nl,nu);
    % sixth order
    elseif(ndss==48) uyy1d=dss048(yl,yu,ny,u1d,uy1d,nl,nu);
    % eighth order
    elseif(ndss==50) uyy1d=dss050(yl,yu,ny,u1d,uy1d,nl,nu);
    % tenth order
    end
```

```
%
% 1D to 2D
    uyy(i,:)=uyy1d(:);
  end
```

Also, since Matlab requires that input arguments to a function be assigned a value, we use `uy1d(:)=0.0;` to meet this requirement, even though the first derivative `uy1d` is not used by `dss044` (since the BCs in y are Dirichlet, not Neumann as in x).

4. Finally, with u_{xx} and u_{yy} available, Eq. (10.2) can be programmed (as in `pde_2`).

```
%
% ut = uxx + uyy
  ut=uxx+uyy;
  ut(:,1) =0.0;
  ut(:,ny)=0.0;
%
% 2D to 1D matrix conversion
  if(mmf==1)
    for i=1:nx
    for j=1:ny
      yt((i-1)*ny+j)=ut(i,j);
    end
    end
    yt=yt';
  end
  if(mmf==2)
    yt=reshape(ut',nx*ny,1);
  end
%
% Increment calls to pde_3
  ncall=ncall+1;
```

The numerical solution produced by `pde_1_main` and `pde_3` is essentially the same as that from `pde_1` and `pde_2` in Tables 10.1 and 10.2, so it will not be listed here.

We conclude this discussion of the solution of 2D PDEs, as illustrated with Laplace's equation (Eq. (10.1)), with the following points:

1. This approach to elliptic PDEs by converting them to parabolic PDEs (essentially, by adding an initial-value derivative to each PDE) is a general method for the solution of elliptic problems, which is usually termed the method of

pseudo transients or *false transients* [1, 2]. These names stem from the solution of the parabolic problem with respect to the parameter t; in other words, the solution appears to be transient as it approaches the steady-state solution of the elliptic problem.

2. t in the previous example is a form of a *continuation parameter* that is used to continue a known, initial solution (such as Eq. (10.3)) to the final, desired solution (for which the derivatives in t are essentially zero in the preceding example). A variety of ways to *embed a continuation parameter* in the problem of interest (other than as a derivative in t) are widely used to continue a problem from a known solution to a final, desired solution [3]. We will not discuss further this very important and useful method to the solution of complex problems.
3. When the parameter is embedded, care is required in doing this in such a way that the modified problem is stable. For example, if t is embedded in Eq. (10.1) as $u_t = -u_{xx} - u_{yy}$, that is, the sign of the derivative u_t is inverted, the resulting PDE is actually unstable, so continuation to a final solution for which $u_t \approx 0$ is not possible.
4. To repeat, the embedding method previously illustrated for the solution of Eq. (10.1) is quite general. For example, the previous analysis can easily be extended to

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y) \tag{10.9}$$

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = u \tag{10.10}$$

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = g(u) \tag{10.11}$$

Equations (10.9) and (10.10) are known as *Poisson's equation* and *Helmholtz's equation*, respectively. If $g(u)$ in Eq. (10.11) is nonlinear, for example, $e^{(-1/u)}$, a full range of nonlinear effects can be investigated. All that is required in the solution of Eqs. (10.9)–(10.11) is to add programming for $f(x, y)$, $u(x, y)$, or $g(u)$ to subroutines `pde_1`, `pde_2`, or `pde_3`. For example, for Eq. (10.10), the programming `ut=uxx+uyy;` could be replaced by `ut=uxx+uyy+u`.

5. However, we should keep in mind that the equilibrium solution to which the PDE problem in t converges can be determined by the assumed IC (for $t = 0$), especially for nonlinear problems such as Eq. (10.11).
6. The generality and ease of use of the MOL solution of elliptic problems as illustrated by the preceding example is due in part to the use of library ODE integrators for the integration with respect to the embedded parameter t (e.g., `ode15s`). In other words, we can take advantage of quality initial-value ODE integrators that are widely available.
7. One possible choice of an integrator is the sparse option of `ode15s`, particularly if the ODE system has a *sparse Jacobian matrix*. In the present case,

121 ODEs is a rather modest problem and the sparse option may not provide a significant computational advantage (of course, this could be investigated). Another possibility is to directly use the Matlab sparse matrix utilities; we have investigated this approach and have provided a Matlab code that is available as part of the supplemental software library.

8. Since MOL can be applied to parabolic problems (such as the heat equation $u_t = u_{xx}$) and hyperbolic problems (such as the wave equation $u_{tt} = u_{xx}$), MOL can be applied to *all three major classes of PDEs, elliptic, parabolic, and hyperbolic*. Also, since MOL can be applied to systems of equations of mixed type, such as hyperbolic–parabolic PDEs, it is a general framework for the numerical solution of PDE systems.

REFERENCES

[1] Schiesser, W. E. (1991), *The Numerical Method of Lines*, Academic Press, San Diego, CA

[2] Schiesser, W. E. (1994), *Computational Mathematics in Engineering and Applied Science: ODEs, DAEs and PDEs*, CRC Press, Boca Raton, FL

[3] Lee, E. S. (1968), *Quasilinearization and Invariant Imbedding: With Applications to Chemical Engineering and Adaptive Control*, Academic Press, New York