11

Three-Dimensional Partial Differential Equation

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

1. A PDE in three dimensions (3D), and therefore the following analysis, illustrates the method of lines (MOL) solution of 3D PDEs.
2. An elliptic PDE so that the following analysis demonstrates the application of the MOL to elliptic PDEs.
3. The implementation of Dirichlet, Neumann, and third-type (mixed, Robin) boundary conditions (BCs).
4. A basic PDE with a variety of extensions, for example, inhomogeneous, PDEs with linear and nonlinear source terms.
5. Evaluation of the accuracy of the numerical solution, including a comparison between the numerical and analytical solutions.
6. Application of *h*- and *p*-refinement to achieve spatial convergence of the numerical solution.
7. The concept of continuation for the solution of elliptic PDEs and other important classes of problems.

The 3D PDE is

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

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

$$0 = u_{xx} + u_{yy} + u_{yy} \tag{11.1}$$

Equation (11.1) is classified as an *elliptic* PDE; it is a *boundary-value problem only* (since it does not have an initial-value independent variable). We follow the general approach of false transients, as discussed in Chapter 10.

Since the MOL generally involves the integration of a system of initial-value ordinary differential equations (ODEs), it cannot be applied directly to Eq. (11.1) (again, Eq. (11.1) does not have an initial-value independent variable). Thus, to

perform a MOL solution of Eq. (11.1), we add a derivative with respect to the initial-value variable t to form a *parabolic PDE*:

$$u_t = u_{xx} + u_{yy} + u_{yy} \tag{11.2}$$

Equation (11.2) can be integrated to a *steady state* or *equilibrium condition*, corresponding to $t \to \infty$ for which $u_t \approx 0$, and then Eq. (11.2) reverts to Eq. (11.1).

For the BCs in x, we use the *nonhomogeneous or inhomogeneous (nonzero) Dirichlet conditions*

$$u(x = 0, y, z, t) = u(x = 1, y, z, t) = 1 \tag{11.3)(11.4}$$

For the BCs in y, we use the *homogeneous (zero) Neumann conditions*

$$u_y(x, y = 0, z, t) = u_y(x, y = 1, z, t) = 0 \tag{11.5)(11.6}$$

For the BCs in z, we use the *third-type (mixed, Robin) conditions*

$$u_z(x, y, z = 0, t) + u(x, y, z = 0, t) = 1 \tag{11.7}$$

$$u_z(x, y, z = 1, t) + u(x, y, z = 1, t) = 1 \tag{11.8}$$

Equation (11.2) requires one *"pseudo" initial condition* (IC) in t (since it is first order in t). We use the term "pseudo" because t in Eq. (11.2) is not part of the original problem of Eq. (11.1). 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. (11.1); 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:

$$u(x = 0, y, z, t = 0) = u(x = 1, y, z, t = 0) = 1 \tag{11.9a}$$

Otherwise

$$u(x, y, z, t = 0) = 0 \quad (x \neq 0, 1) \tag{11.9b}$$

Equations (11.2)–(11.9) for $t \to \infty$ have the analytical solution

$$u(x, y, z, t \to \infty) = 1 \tag{11.10}$$

which will be used to evaluate the accuracy of the solution to Eq. (11.1) (subject to BCs (11.3)–(11.8)).

A main program for the solution of Eqs. (11.2)–(11.9) is given in Listing 11.1.

```
%
% Clear previous files
  clear all
  clc
%
% Parameters shared with the ODE routine
```

```
  global ncall nx ny nz ndss

% Initial condition (which also is consistent with the boundary
% conditions)
  nx=11;
  ny=11;
  nz=11;
  for i=1:nx
  for j=1:ny
  for k=1:nz
    if(i==1)     u0(i,j,k)=1.0;
    elseif(i==nx)u0(i,j,k)=1.0;
    else         u0(i,j,k)=0.0;
    end
  end
  end
  end
%
% 3D to 1D matrix conversion
  for i=1:nx
  for j=1:ny
  for k=1:nz
     y0((i-1)*ny*nz+(j-1)*nz+k)=u0(i,j,k);
  end
  end
  end
%
% Independent variable for ODE integration
  t0=0.0;
  tf=1.0;
  tout=[t0:0.1:tf]';
  nout=11;
  ncall=0;
%
% ODE integration
  mf=3;
  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 3D matrix conversion (plus t)
  for it=1:nout
```

```
  for i=1:nx
  for j=1:ny
  for k=1:nz
    u(it,i,j,k)=y(it,(i-1)*ny*nz+(j-1)*nz+k);
  end
  end
  end
  end
%
% Display selected output as t,
%   u(t,x=0,y=0.5,z=0.5)
%   u(t,x=1,y=0.5,z=0.5)
%   u(t,x=0.5,y=0,z=0.5)
%   u(t,x=0.5,y=1,z=0.5)
%   u(t,x=0.5,y=0.5,z=0)
%   u(t,x=0.5,y=0.5,z=1)
%   u(t,x=0.5,y=0.5,z=0.5)
  fprintf('\n mf = %2d\n',mf);
  for it=1:2:nout
    fprintf('\n t = %5.3f\n%10.3f%10.3f\n%10.3f%10.3f
            \n%10.3f%10.3f\n%10.3f\n',t(it), ...
            u(it,1,6,6),u(it,nx,6,6), ...
            u(it,6,1,6),u(it,6,ny,6),u(it,6,6,1), ...
            u(it,6,6,nz),u(it,6,6,6));
  end
  fprintf('\n ncall = %4d\n',ncall);
```

Listing 11.1. 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 ODE routine, IC Eqs. (11.9a) and (11.9b) are coded over a $11 \times 11 \times 11 = 1{,}331$-point grid in x, y, and z; note that along the boundaries $x = 0$ and $x = 1$, $u(x, y, z, t = 0) = 1$, and everywhere else $u(x, y, z, t = 0) = 0$. This consistency between IC (11.9) and BCs (11.3) and (11.4) facilitates the numerical solution (by avoiding an abrupt change at these boundaries for $t > 0$); in other words, choosing the IC to provide a smooth transition from $t = 0$ to $t > 0$ as much as possible minimizes numerical problems.

```
%
% Clear previous files
  clear all
  clc
%
```

```
% Parameters shared with the ODE routine
  global ncall nx ny nz ndss

% Initial condition (which also is consistent with the boundary
% conditions)
  nx=11;
  ny=11;
  nz=11;
  for i=1:nx
  for j=1:ny
  for k=1:nz
    if(i==1)     u0(i,j,k)=1.0;
    elseif(i==nx)u0(i,j,k)=1.0;
    else         u0(i,j,k)=0.0;
    end
  end
  end
  end
```

The choice of `nx=11, ny=11, nz=11` grid points in x, y, and z 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. (11.1). 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. (11.1). This is especially true for 3D problems for which the total number of grid points increases very rapidly with increases in grid points in each spatial dimension (the so-called *curse of dimensionality*).

2. Since library ODE integrators, such as the Matlab integrators, generally require all of the dependent variables to be arranged in a single ID vector, we must convert the 3D IC matrix `u0(i,j,k)` ($= u(x, y, z, t = 0)$) of Eq. (11.9) into a 1D vector, in this case, `y0`.

```
%
% 3D to 1D matrix conversion
  for i=1:nx
  for j=1:ny
  for k=1:nz
     y0((i-1)*ny*nz+(j-1)*nz+k)=u0(i,j,k);
  end
  end
  end
```

The details of this conversion in three nested `for` loops should be studied since this is an essential operation for the solution of 3D PDEs. We choose

here to explicitly code the conversion to clarify what has actually been done. This conversion could also be coded more compactly using the vector/matrix operations in Matlab, as illustrated in Chapter 10 where the `reshape` function has been used.

3. The total interval in t is defined ($0 \leq t \leq 1$). The solution is to be displayed 11 times, so the output interval is 0.1.

```
%
% Independent variable for ODE integration
  t0=0.0;
  tf=1.0;
  tout=[t0:0.1:tf]';
  nout=11;
  ncall=0;
```

4. The ODE integration (for a total of 1,331 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`). We did try the sparse matrix form of `ode15s` since the Jacobian matrix of this 1,331-ODE system is no doubt sparse, but this did not offer a significant reduction in computer run times.

```
%
% ODE integration
  mf=3;
  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 (FD) approximations of the derivatives u_{xx}, u_{yy}, u_{zz} in Eqs. (11.1) and (11.2). 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 u_{xx}, u_{yy}, u_{zz} in Eqs. (11.1) and (11.2) by a library routine for first-order derivatives, for example, `DSS004`. The second-order derivatives are computed by successive differentiation 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 u_{xx}, u_{yy}, u_{zz} in Eqs. (11.1) and (11.2) 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 11 and a combination of x, y, and z with a second dimension of 1,331. This matrix is then expressed as a 4D matrix in x, y, and z (with indices `i,j,k`, respectively) at a series of `nout=11` values of t through the index `it`.

```
%
% 1D to 3D matrix conversion (plus t)
  for it=1:nout
  for i=1:nx
  for j=1:ny
  for k=1:nz
    u(it,i,j,k)=y(it,(i-1)*ny*nz+(j-1)*nz+k);
  end
  end
  end
  end
```

6. Tabular output is displayed so that the numerical solution can be compared with the analytical solution (Eq. 11.10.)

```
%
% Display selected output as t,
%   u(t,x=0,y=0.5,z=0.5)
%   u(t,x=1,y=0.5,z=0.5)
%   u(t,x=0.5,y=0,z=0.5)
%   u(t,x=0.5,y=1,z=0.5)
%   u(t,x=0.5,y=0.5,z=0)
%   u(t,x=0.5,y=0.5,z=1)
%   u(t,x=0.5,y=0.5,z=0.5)
  fprintf('\n mf = %2d\n',mf);
  for it=1:2:nout
    fprintf('\n t = %5.3f\n%10.3f%10.3f\n%10.3f%10.3f
           \n%10.3f%10.3f\n%10.3f\n',t(it),u(it,1,6,6), ...
            u(it,nx,6,6),u(it,6,1,6),u(it,6,ny,6), ...
            u(it,6,6,1),u(it,6,6,nz),u(it,6,6,6));
  end
  fprintf('\n ncall = %4d\n',ncall);
```

The three ODE routines called by `ode15s`, `pde_1.m`, `pde_2.m`, `pde_3.m`, are now discussed. Refer to Listing 11.2 for `pde_1.m`.

```
  function yt=pde_1(t,y)
%
% Problem parameters
  global ncall nx ny nz
  xl=0.0;
  xu=1.0;
  yl=0.0;
  yu=1.0;
  zl=0.0;
  zu=1.0;
%
% 1D to 3D matrix conversion
  for i=1:nx
  for j=1:ny
  for k=1:nz
    u(i,j,k)=y((i-1)*ny*nz+(j-1)*nz+k);
  end
  end
  end
%
% PDE
  dx=(xu-xl)/(nx-1);
  dy=(yu-yl)/(ny-1);
  dz=(zu-zl)/(nz-1);
  dx2=dx^2;
  dy2=dy^2;
  dz2=dz^2;
  for i=1:nx
  for j=1:ny
  for k=1:nz
%
%   uxx
    if(i~=1)&(i~=nx)uxx(i,j,k)=(u(i+1,j,k)-2.0*u(i,j,k) ...
                               +u(i-1,j,k))/dx2; end
%
%   uyy
    if(j==1)        uyy(i,j,k)=2.0*(u(i,j+1,k)-u(i,j,k))/dy2;
    elseif(j==ny)   uyy(i,j,k)=2.0*(u(i,j-1,k)-u(i,j,k))/dy2;
    else            uyy(i,j,k)=(u(i,j+1,k)-2.0*u(i,j,k)...
                               +u(i,j-1,k))/dy2;
    end
%
%   uzz
```

```
    if(k==1)        u0=u(i,j,k+1)-2.0*dz*(1.0-u(i,j,k));
                    uzz(i,j,k)=(u(i,j,k+1)-2.0*u(i,j,k)+u0)/dz2;
    elseif(k==nz)   u1=u(i,j,k-1)+2.0*dz*(1.0-u(i,j,k));
                    uzz(i,j,k)=(u1-2.0*u(i,j,k)+u(i,j,k-1))/dz2;
    else            uzz(i,j,k)=(u(i,j,k+1)-2.0*u(i,j,k) ...
                              +u(i,j,k-1))/dz2;
    end
%
%   ut = uxx + uyy + uzz
    if(i==1)|(i==nx)ut(i,j,k)=0.0;
    else ut(i,j,k)=uxx(i,j,k)+uyy(i,j,k)+uzz(i,j,k);
    end
  end
  end
  end
%
% 3D to 1D matrix conversion
  for i=1:nx
  for j=1:ny
  for k=1:nz
    yt((i-1)*ny*nz+(j-1)*nz+k)=ut(i,j,k);
  end
  end
  end
  yt=yt';
%
% increment calls to pde_1
  ncall=ncall+1;
```

Listing 11.2. 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 $n = 1{,}331$ dependent variables, failure to evaluate all of the derivatives in `ut` can easily happen).

```
  function yt=pde_1(t,y)
%
% Problem parameters
  global ncall nx ny nz
  xl=0.0;
  xu=1.0;
  yl=0.0;
  yu=1.0;
  zl=0.0;
  zu=1.0;
```

Some problem parameters are then declared as *global* so that they can be shared with `pde_1_main` in Listing 11.1 or other routines. In this case, the dimensions of the 3D $x - y - z$ domain are defined (and, of course, these dimensions can be different in the three directions).

2. A conversion of the 1D vector, `y`, to the 3D matrix `u` is made so that the subsequent programming can be done in terms of the problem-oriented variable `u` in Eqs. (11.1)–(11.9). Although this conversion is, in principle, not required, programming in terms of problem-oriented variables, for example, `u`, is a major convenience.

```
%
% 1D to 3D matrix conversion
  for i=1:nx
  for j=1:ny
  for k=1:nz
    u(i,j,k)=y((i-1)*ny*nz+(j-1)*nz+k);
  end
  end
  end
```

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

```
%
% PDE
  dx=(xu-xl)/(nx-1);
  dy=(yu-yl)/(ny-1);
  dz=(zu-zl)/(nz-1);
  dx2=dx^2;
  dy2=dy^2;
  dz2=dz^2;
```

4. We then move throughout the $x - y - z$ domain using three nested `for` loops. Each pass through this set of `for` loops executes the programming for another ODE (a total of 1,331 passes or ODEs).

```
  for i=1:nx
  for j=1:ny
  for k=1:nz
%
%   uxx
    if(i~=1)&(i~=nx)uxx(i,j,k)=(u(i+1,j,k)-2.0*u(i,j,k) ...
                               +u(i-1,j,k))/dx2; end
%
%   uyy
    if(j==1)        uyy(i,j,k)=2.0*(u(i,j+1,k)-u(i,j,k))/dy2;
    elseif(j==ny)   uyy(i,j,k)=2.0*(u(i,j-1,k)-u(i,j,k))/dy2;
    else            uyy(i,j,k)=(u(i,j+1,k)-2.0*u(i,j,k) ...
                               +u(i,j-1,k))/dy2;
    end
%
%   uzz
    if(k==1)        u0=u(i,j,k+1)-2.0*dz*(1.0-u(i,j,k));
                    uzz(i,j,k)=(u(i,j,k+1)-2.0*u(i,j,k)+u0)/dz2;
    elseif(k==nz)   u1=u(i,j,k-1)+2.0*dz*(1.0-u(i,j,k));
                    uzz(i,j,k)=(u1-2.0*u(i,j,k)+u(i,j,k-1))/dz2;
    else            uzz(i,j,k)=(u(i,j,k+1)-2.0*u(i,j,k) ...
                               +u(i,j,k-1))/dz2;
    end
%
%   ut = uxx + uyy + uzz
    if(i==1)|(i==nx)ut(i,j,k)=0.0;
    else ut(i,j,k)=uxx(i,j,k)+uyy(i,j,k)+uzz(i,j,k);
    end
  end
  end
  end
```

The coding of the FD approximations requires some explanation. For the derivative u_{xx}, two cases are required:
(a) For $x \neq 0, 1$, $u_{xx}(x, y, z, t)$ is programmed as the FD approximation

```
%
%   uxx
    if(i~=1)&(i~=nx)uxx(i,j,k)=(u(i+1,j,k)-2.0*u(i,j,k) ...
                               +u(i-1,j,k))/dx2; end
```

which is based on

$$u_{xx} \approx \frac{u(x+\Delta x, y, z, t) - 2u(x, y, z, t) + u(x-\Delta x, y, z, t)}{\Delta x^2} \tag{11.11}$$

(b) For $x = 0, 1$, BCs (11.3) and (11.4) are programmed toward the end of `pde_1` as

```
if(i==1)|(i==nx)ut(i,j,k)=0.0;
```

In other words, since the boundary values of Eqs. (11.3) and (11.4) are constant, their derivatives in t are zero.

For the derivative u_{yy}, two cases are again required:

(a) For BCs (11.5) and (11.6),

```
%
%   uyy
    if(j==1)        uyy(i,j,k)=2.0*(u(i,j+1,k)-u(i,j,k))/dy2;
    elseif(j==ny)   uyy(i,j,k)=2.0*(u(i,j-1,k)-u(i,j,k))/dy2;
```

This coding is based on the approximation of a second derivative (see Eq. (11.11)), including the approximation of homogeneous Neumann BCs (11.5) and (11.6) (i.e., $u(x, y = 0 - \Delta y, z, t) = u(x, y = 0 + \Delta y, z, t)$, $u(x, y = 1 + \Delta y, z, t) = u(x, y = 1 - \Delta y, z, t)$):

$$u_{yy}(x, y = 0, z, t) \approx 2\frac{u(x, y = 0 + \Delta y, z, t) - u(x, y = 0, z, t)}{\Delta y^2} \tag{11.12a}$$

$$u_{yy}(x, y = 1, z, t) \approx 2\frac{u(x, y = 1 - \Delta y, z, t) - u(x, y = 1, z, t)}{\Delta y^2} \tag{11.12b}$$

(b) For $y \neq 0, 1$, the approximation of uyy is programmed as

```
else  uyy(i,j,k)=(u(i,j+1,k)-2.0*u(i,j,k)+u(i,j-1,k))/dy2;
```

corresponding to the FD

$$u_{yy} \approx \frac{u(x, y+\Delta y, z, t) - 2u(x, y, z, t) + u(x, y-\Delta y, z, t)}{\Delta y^2} \tag{11.12c}$$

For the derivative u_{zz}, two cases are also required:

(a) For BCs (11.7) and (11.8),

```
if(k==1)          u0=u(i,j,k+1)-2.0*dz*(1.0-u(i,j,k));
                  uzz(i,j,k)=(u(i,j,k+1)-2.0*u(i,j,k)+u0)/dz2;
elseif(k==nz)     u1=u(i,j,k-1)+2.0*dz*(1.0-u(i,j,k));
                  uzz(i,j,k)=(u1-2.0*u(i,j,k)+u(i,j,k-1))/dz2;
```

This coding is based on the approximation of a second derivative (see Eq. (11.11)), including the approximation of third-type BCs (11.7) and (11.8). For Eq. (11.7) we have (with the fictitious point u_0)

$$u_z(x, y, z = 0, t) \approx \frac{u(x, y, z = \Delta z, t) - u_0}{2\Delta z} = 1 - u(x, y, z = 0, t)$$

or

$$u_0 = (u(x, y, z = \Delta z, t) - 2\Delta z(1 - u(x, y, z = 0, t)) \tag{11.13a}$$

u_0 from Eq. (11.13a) can then be used in the FD approximation of u_{zz} (see Eq. (11.11)).

$$u_{zz} = \frac{u(x, y, z = \Delta z, t) - 2u(x, y, z = 0, t) + u_0}{\Delta z^2} \tag{11.13b}$$

(b) Similarly, for BC (11.8) (with fictitious point u_1),

$$u_z(x, y, z = 1, t) \approx \frac{u_1 - u(x, y, z = 1 - \Delta z, t)}{2\Delta z} = 1 - u(x, y, z = 1, t)$$

or

$$u_1 = (u(x, y, z = 1 - \Delta z, t) + 2\Delta z(1 - u(x, y, z = 1, t)) \tag{11.13c}$$

u_1 from Eq. (11.13c) can then be used in the FD approximation of u_{zz} (see Eq. (11.11)).

$$u_{zz} = \frac{u_1 - 2u(x, y, z = 1, t) + u(x, y, z = 1 - \Delta z, t)}{\Delta z^2} \tag{11.13d}$$

(c) For $z \neq 0, 1$, the approximation of u_{zz} is

```
else uzz(i,j,k)=(u(i,j,k+1)-2.0*u(i,j,k)+u(i,j,k-1))/dz2;
```

corresponding to the FD

$$u_{zz} \approx \frac{u(x, y, z + \Delta z, t) - 2u(x, y, z, t) + u(x, y, z - \Delta z, t)}{\Delta z^2} \tag{11.13e}$$

With the three derivatives u_{xx}, u_{yy}, u_{zz} computed, PDE (11.2) is programmed as

```
if(i==1)|(i==nx)ut(i,j,k)=0.0;
else ut(i,j,k)=uxx(i,j,k)+uyy(i,j,k)+uzz(i,j,k);
```

The numerical output from `pde_1_main` of Listing 11.1 and `pde_1` of Listing 11.2 for `mf=1` is given in Table 11.1.

Table 11.1. Numerical output for the solution of Eqs. (11.2)–(11.9) from `pde_1_main` and `pde_1`, `mf=1`

```
mf =  1

t = 0.000
    1.000     1.000
    0.000     0.000
    0.000     0.000
    0.000

t = 0.200
    1.000     1.000
    0.804     0.804
    0.692     0.874
    0.804

t = 0.400
    1.000     1.000
    0.967     0.967
    0.945     0.980
    0.967

t = 0.600
    1.000     1.000
    0.994     0.994
    0.990     0.996
    0.994

t = 0.800
    1.000     1.000
    0.999     0.999
    0.998     0.999
    0.999

t = 1.000
    1.000     1.000
    1.000     1.000
    1.000     1.000
    1.000

ncall = 1422
```

We can note the following points:

1. This output (which is quite abbreviated since the solution at 1,331 points is available) indicates the approach of the numerical solution to the analytical solution of Eq. (11.10). A more complete picture of the solution might be available from a series of 3D plots, although this would be rather challenging since there are four independent variables, x, y, z, t. Some subplots might be a better way to proceed, for example, plots at specific values of x, y, z as a function of t. We did not attempt this approach to displaying the solution graphically in the preceding Matlab; some further discussion is given in Appendix 6.
2. IC (11.9) introduces discontinuities at $x = 0, 1$ for which the numerical solution appears to remain quite smooth, which is a consequence of parabolic PDE (11.2); that is, parabolic PDEs tend to "diffuse" or smooth numerical solutions (the same is not true for hyperbolic PDEs that tend to propagate discontinuities and subsequently produce severe numerical distortions).
3. The computational effort is rather substantial (1,422 calls to `pde_1`, which is rather typical for 3D problems).

For `mf=2` in `pde_1_main`, `pde_2` is the ODE routine called by `ode15s` (see Listing 11.3).

```
  function yt=pde_2(t,y)
%
% Problem parameters
  global ncall nx ny nz ndss
  xl=0.0;
  xu=1.0;
  yl=0.0;
  yu=1.0;
  zl=0.0;
  zu=1.0;
%
% 1D to 3D matrix conversion
  for i=1:nx
  for j=1:ny
  for k=1:nz
    u(i,j,k)=y((i-1)*ny*nz+(j-1)*nz+k);
  end
  end
  end
%
% PDE
%
% ux
  for j=1:ny
  for k=1:nz
    u1d=u(:,j,k);
```

```
    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
    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 3D
    uxx(:,j,k)=uxx1d(:);
  end
  end
%
% uy
  for i=1:nx
  for k=1:nz
    u1d=u(i,:,k);
    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
    uy1d(1) =0.0;
    uy1d(ny)=0.0;
    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 3D
    uyy(i,:,k)=uyy1d(:);
  end
  end
%
% uz
  for i=1:nx
  for j=1:ny
    u1d=u(i,j,:);
    if    (ndss== 2) uz1d=dss002(zl,zu,nz,u1d); % second order
    elseif(ndss== 4) uz1d=dss004(zl,zu,nz,u1d); % fourth order
    elseif(ndss== 6) uz1d=dss006(zl,zu,nz,u1d); % sixth order
    elseif(ndss== 8) uz1d=dss008(zl,zu,nz,u1d); % eighth order
    elseif(ndss==10) uz1d=dss010(zl,zu,nz,u1d); % tenth order
    end
%
% uzz
    uz1d(1) =1.0-u1d(1);
    uz1d(nz)=1.0-u1d(nz);
    if    (ndss== 2) uzz1d=dss002(zl,zu,nz,uz1d);
    % second order
    elseif(ndss== 4) uzz1d=dss004(zl,zu,nz,uz1d);
    % fourth order
    elseif(ndss== 6) uzz1d=dss006(zl,zu,nz,uz1d);
    % sixth order
    elseif(ndss== 8) uzz1d=dss008(zl,zu,nz,uz1d);
    % eighth order
    elseif(ndss==10) uzz1d=dss010(zl,zu,nz,uz1d);
    % tenth order
    end
%
% 1D to 3D
    uzz(i,j,:)=uzz1d(:);
  end
  end
%
% ut = uxx + uyy + uzz
  ut=uxx+uyy+uzz;
  ut(1,:,:) =0.0;
  ut(nx,:,:)=0.0;
%
% 3D to 1D matrix conversion
  for i=1:nx
```

```
  for j=1:ny
  for k=1:nz
    yt((i-1)*ny*nz+(j-1)*nz+k)=ut(i,j,k);
  end
  end
  end
  yt=yt';
%
% increment calls to pde_2
  ncall=ncall+1;
```

Listing 11.3. 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 3D matrix `u` is made so that programming in terms of problem-oriented variables is set up.

```
  function yt=pde_2(t,y)
%
% Problem parameters
  global ncall nx ny nz ndss
  xl=0.0;
  xu=1.0;
  yl=0.0;
  yu=1.0;
  zl=0.0;
  zu=1.0;
%
% 1D to 3D matrix conversion
  for i=1:nx
  for j=1:ny
  for k=1:nz
    u(i,j,k)=y((i-1)*ny*nz+(j-1)*nz+k);
  end
  end
  end
```

2. 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
  for k=1:nz
    u1d=u(:,j,k);
    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
    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 3D
    uxx(:,j,k)=uxx1d(:);
  end
  end
```

Since these `DSS` routines compute numerical derivatives of 1D arrays, it is necessary before calling them to temporarily store the 3D array `u` in a 1D array, `u1d`, which is easily done using the subscripting utilities of Matlab (in this case, the `:` operator).

Then a second call to `dss004` computes the second derivative `uxx1d`, again a 1D array. `uxx1d` is returned to a 3D array, `uxx`, for subsequent MOL programming of Eq. (11.2). Note, in particular, the use of *stagewise differentiation* using $u \rightarrow u_x \rightarrow u_{xx}$ by two successive calls to `dss004`.

Also, Dirichlet BCs (11.3) and (11.4) are not used in the calculation of u_{xx} at this point. In other words, it would seem reasonable that these BCs would be programmed before the first call to `dss004` using something like

```
u1d(1 ,j,k)=1.0;
u1d(nx,j,k)=1.0;
```

but these statements would have no effect since dependent variables cannot be set in the ODE routine (a property of the Matlab integrators such as `ode15s`). Rather, these boundary values are set in the IC of Eq. (11.9) in `pde_1_main` and passed to `pde_1` through its second argument `y`. The derivatives of these boundary values are set to zero later in `pde_1` to ensure the constant boundary values (of Eqs. (11.3) and (11.4)) are maintained.

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

```
%
% uy
  for i=1:nx
  for k=1:nz
    u1d=u(i,:,k);
    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
    uy1d(1) =0.0;
    uy1d(ny)=0.0;
    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 3D
    uyy(i,:,k)=uyy1d(:);
  end
  end
```

Note that Neumann BCs (11.5) and (11.6) are implemented as

```
    uy1d(1) =0.0;
    uy1d(ny)=0.0;
```

4. The derivative in z, u_{zz}, is computed and eventually is stored in uzz.

```
%
% uz
  for i=1:nx
  for j=1:ny
    u1d=u(i,j,:);
    if    (ndss== 2) uz1d=dss002(zl,zu,nz,u1d); % second order
    elseif(ndss== 4) uz1d=dss004(zl,zu,nz,u1d); % fourth order
    elseif(ndss== 6) uz1d=dss006(zl,zu,nz,u1d); % sixth order
    elseif(ndss== 8) uz1d=dss008(zl,zu,nz,u1d); % eighth order
    elseif(ndss==10) uz1d=dss010(zl,zu,nz,u1d); % tenth order
    end
%
% uzz
    uz1d(1) =1.0-u1d(1);
    uz1d(nz)=1.0-u1d(nz);
    if    (ndss== 2) uzz1d=dss002(zl,zu,nz,uz1d);
    % second order
    elseif(ndss== 4) uzz1d=dss004(zl,zu,nz,uz1d);
    % fourth order
    elseif(ndss== 6) uzz1d=dss006(zl,zu,nz,uz1d);
    % sixth order
    elseif(ndss== 8) uzz1d=dss008(zl,zu,nz,uz1d);
    % eighth order
    elseif(ndss==10) uzz1d=dss010(zl,zu,nz,uz1d);
    % tenth order
    end
%
```

```
% 1D to 3D
    uzz(i,j,:)=uzz1d(:);
  end
  end
```

Note that the third-type BCs (11.7) and (11.8) are implemented as

```
    uz1d(1) =1.0-u1d(1);
    uz1d(nz)=1.0-u1d(nz);
```

5. With the three partial derivatives in the RHS of Eq. (11.2) now available, this PDE can be programmed.

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

Note the application of the BCs in x, Eqs. (11.3) and (11.4), by setting the derivatives in t to zero to maintain these constant values.

6. Finally, a 3D 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`.

```
%
% 3D to 1D matrix conversion
  for i=1:nx
  for j=1:ny
  for k=1:nz
    yt((i-1)*ny*nz+(j-1)*nz+k)=ut(i,j,k);
  end
  end
  end
  yt=yt';
%
% increment calls to pde_2
  ncall=ncall+1;
```

The numerical output from `pde_1_main` and `pde_2` is given in Table 11.2.

Table 11.2. Numerical output for the solution of Eqs. (11.2)–(11.9) from `pde_1_main` and `pde_2`, `mf=2`

```
mf =  2

t = 0.000
    1.000     1.000
    0.000     0.000
    0.000     0.000
    0.000

t = 0.200
    1.000     1.000
    0.806     0.806
    0.693     0.875
    0.806

t = 0.400
    1.000     1.000
    0.967     0.967
    0.946     0.980
    0.967

t = 0.600
    1.000     1.000
    0.994     0.994
    0.991     0.997
    0.994

t = 0.800
    1.000     1.000
    0.999     0.999
    0.998     0.999
    0.999

t = 1.000
    1.000     1.000
    1.000     1.000
    1.000     1.000
    1.000

ncall = 1420
```

This output is very similar to that for `mf=1` in Table 11.1 Again, the agreement with the analytical solution, Eq. (11.10), 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, Eq. (11.10), is so smooth; specifically, the final solution is constant 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 11.4).

```
  function yt=pde_3(t,y)
%
% Problem parameters
  global ncall nx ny nz ndss
  xl=0.0;
  xu=1.0;
  yl=0.0;
  yu=1.0;
  zl=0.0;
  zu=1.0;
%
% 1D to 3D matrix conversion
  for i=1:nx
  for j=1:ny
  for k=1:nz
    u(i,j,k)=y((i-1)*ny*nz+(j-1)*nz+k);
  end
  end
  end
%
% PDE
%
% uxx
  for j=1:ny
  for k=1:nz
    u1d=u(:,j,k);
    nl=1; % Dirichlet
    nu=1; % Dirichlet
    ux1d(:)=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 3D
    uxx(:,j,k)=uxx1d(:);
  end
  end
%
% uyy
  for i=1:nx
  for k=1:nz
    u1d=u(i,:,k);
    nl=2; % Neumann
    nu=2; % Neumann
    uy1d(1) =0.0;
    uy1d(ny)=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 3D
    uyy(i,:,k)=uyy1d(:);
  end
  end
%
% uzz
  for i=1:nx
  for j=1:ny
    u1d=u(i,j,:);
    nl=2; % Neumann
    nu=2; % Neumann
    uz1d(1) =1.0-u1d(1);
    uz1d(nz)=1.0-u1d(nz);
    if    (ndss==42) uzz1d=dss042(zl,zu,nz,u1d,uz1d,nl,nu);
    % second order
    elseif(ndss==44) uzz1d=dss044(zl,zu,nz,u1d,uz1d,nl,nu);
```

```
    % fourth order
    elseif(ndss==46) uzz1d=dss046(zl,zu,nz,u1d,uz1d,nl,nu);
    % sixth order
    elseif(ndss==48) uzz1d=dss048(zl,zu,nz,u1d,uz1d,nl,nu);
    % eighth order
    elseif(ndss==50) uzz1d=dss050(zl,zu,nz,u1d,uz1d,nl,nu);
    % tenth order
    end
%
% 1D to 3D
    uzz(i,j,:)=uzz1d(:);
  end
  end
%
% ut = uxx + uyy + uzz
  ut=uxx+uyy+uzz;
  ut(1,:,:) =0.0;
  ut(nx,:,:)=0.0;
%
% 3D to 1D matrix conversion
  for i=1:nx
  for j=1:ny
  for k=1:nz
    yt((i-1)*ny*nz+(j-1)*nz+k)=ut(i,j,k);
  end
  end
  end
  yt=yt';
%
% increment calls to pde_3
  ncall=ncall+1;
```

Listing 11.4. ODE routine `pde_3.m`

`pde_3` closely parallels `pde_2` (Listing 11.3). The essential differences are as follows:

1. The use of `dss044` for the direct calculation of the derivatives u_{xx}, u_{yy}, u_{zz} rather than `dss004` for the calculation of these second derivatives via the corresponding first derivatives (through stagewise differentiation).
2. The programming of the Dirichlet BCs of Eqs. (11.3) and (11.4) is

```
    nl=1; % Dirichlet
    nu=1; % Dirichlet
    ux1d(:)=0.0;
```

The setting of the first derivative `ux1d` is required, even though this first derivative is not used, because it is an input argument of `dss044`; Matlab has the requirement that each input argument of a function must have at least one assigned value.

The programming of the Neumann BCs of Eqs. (11.5) and (11.6), that is, the derivatives $u_y(x, y = 0, z, t)$ = `uy1d(1)` and $u_y(x, y = 1, z, t)$ = `uy1d(ny)`, is

```
nl=2; % Neumann
nu=2; % Neumann
uy1d(1) =0.0;
uy1d(ny)=0.0;
```

3. The programming of the third-type BCs of Eqs. (11.7) and (11.8) is done as two Neumann BCs by defining the derivatives $u_z(x, y, z = 0, t)$ = `uz1d(1)` and $u_z(x, y, z = 1, t)$ = `uz1d(nz)`:

```
nl=2; % Neumann
nu=2; % Neumann
uz1d(1) =1.0-u1d(1);
uz1d(nz)=1.0-u1d(nz);
```

The output from `pde_1_main` and `tt pde_3` is given in Table 11.3. This output is very similar to the output in Tables 11.1 and 11.2 for `pde_1` and `pde_2`.

We conclude this discussion of the solution of 3D PDEs, as illustrated by Eqs. (11.1) and (11.2), with the following points:

1. We could investigate the accuracy of the numerical solution (in addition to comparing it with the analytical solution, Eq. (11.10)) by
   (a) Varying the number of spatial grid points, but with modest increases because of the increase in total ODEs, which is particularly limiting for 3D problems. If the solution remains essentially unchanged to a reasonable number of figures, for example, 3–4, with changes in the number of grid points, we have some assurance that the numerical solution is accurate to that number of figures. Since the grid spacing is often given the symbol h in the numerical analysis literature, varying the number of grid points is generally termed *h-refinement*.
   (b) Varying the order of the FD approximations. This is easily done in the present case by changing the calls to the differentiation routines, for example, `dss004` to `dss006` or `dss044` to `dss046`. Only the name of the routine changes; the arguments can remain the same. Then we can observe the effect of changing the order of the FD approximations on the numerical solution. Since the order of the FD approximations is often given the symbol p, this is termed *p-refinement*.

Table 11.3. Numerical output for the solution of Eqs. (11.2)–(11.9) from `pde_1_main` and `pde_3`, `mf=3`

```
mf =  3

t = 0.000
    1.000     1.000
    0.000     0.000
    0.000     0.000
    0.000

t = 0.200
    1.000     1.000
    0.805     0.805
    0.693     0.875
    0.805

t = 0.400
    1.000     1.000
    0.967     0.967
    0.946     0.980
    0.967

t = 0.600
    1.000     1.000
    0.994     0.994
    0.991     0.997
    0.994

t = 0.800
    1.000     1.000
    0.999     0.999
    0.998     0.999
    0.999

t = 1.000
    1.000     1.000
    1.000     1.000
    1.000     1.000
    1.000
ncall = 1427
```

(c) Using the third commonly used method to assess solution accuracy, which we have not used here, that is, to adaptively refine the grid. This is usually termed *adaptive mesh refinement (AMR)* or *r-refinement*.

2. The preceding 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*. 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.
3. t in the previous example is a form of a *continuation parameter* that is used to continue a known, initial solution (such as Eq. (11.9)) 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. We will not discuss further this very important and useful method to the solution of complex problems.
4. 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. (11.1) as $u_t = -u_{xx} - u_{yy} - u_{yy}$, 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.
5. To repeat, the embedding method previously illustrated for the solution of Eq. (11.1) is quite general. For example, the previous analysis can easily be extended to

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = f(x, y, z) \tag{11.14a}$$

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = u \tag{11.14b}$$

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = g(u) \tag{11.14c}$$

If $g(u)$ in Eq. (11.14c) 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. (11.14a)–(11.14c) is to add programming for $f(x, y, z)$, $u(x, y, z)$, or $g(u)$ to subroutines `pde_1`, `pde_2`, or `pde_3`. For example, for Eq. (11.14b), the programming `ut=uxx+uyy+yzz;` could be replaced by `ut=uxx+uyy+uzz+u`.
6. Nonlinear BCs can also easily be included. For example, if the third-type BCs of Eqs. (11.7) and (11.8) have the nonlinear form

$$u_z(x, y, z = 0, t) + u^4(x, y, z = 0, t) = 1 \tag{11.15a}$$

$$u_z(x, y, z = 1, t) + u^4(x, y, z = 1, t) = 1 \tag{11.15b}$$

the preceding code for BCs (11.7) and (11.8) changes simply to

```
nl=2; % Neumann
nu=2; % Neumann
uz1d(1) =1.0-u1d(1)^4;
uz1d(nz)=1.0-u1d(nz)^4;
```

This discussion of Eqs. (11.14) and (11.15) demonstrates one of the important advantages of the numerical approach to PDEs (rather than analytical), namely, the ease with which nonlinearities can be included and how easily the form of the nonlinearities can be changed.

7. 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.
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, in principle, 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.
9. Additionally, this generality for PDEs can be readily extended to systems of ODEs and 1D, 2D, and 3D PDEs (essentially by replicating the methods discussed here for each of the equations). Thus, we can, in principle, investigate numerically a broad class of differential equation systems.