Differential Equations in MATLAB 7

Jaywan Chung
KAIST

January 12, 2012

Contents

1 Basic Commands **1**
1.1 Help and Others 2
1.2 Creating a Matrix 2
1.3 Manipulating Matrices 3
1.4 Control Statements and Others 5
1.5 M-File: Script or Function 6
1.6 An Example: Implementing Finite Difference Method 7

2 Ordinary Differential Equations **8**
2.1 Initial-Value Problems for First Order ODEs 8
2.2 Initial-Value Problems for Higher Order ODEs 9
2.3 Boundary-Value Problems 9

3 Partial Differential Equations in One Spatial Dimension **11**
3.1 Single Parabolic PDEs 11
3.2 System of Parabolic PDEs 12

4 Partial Differential Equations in Two Spatial Dimensions **14**
4.1 GUI of PDE Toolbox 14
4.2 Single Elliptic PDEs 15
4.3 Single Parabolic PDEs 17
4.4 System of Parabolic PDEs 19
4.5 Nonlinear Parabolic PDEs 21
4.6 Creating Domains without using GUI 22

5 Graph and Animation **24**
5.1 2D Graphs 24
5.2 3D Graphs 25
5.3 Animation 25

6 Installation Problems **26**

1 Basic Commands

In this section, basic commands in MATLAB are introduced. You can test the commands by typing in the Command Window.

1.1 Help and Others

- `help`: Display help text in *Command Window*.
  `doc`: Display help text in *Help browser*.

- `clear`: Clear variables and functions from memory. `clear all` removes all variables and functions.

  e.g.
  ```
  >> pi
  ans =
      3.1416
  >> pi = 0.5
  pi =
      0.5000
  >> clear pi
  >> pi
  ans =
      3.1416
  ```

- `clc`: Clear Command Window. You can also use the menu $\boxed{\text{Edit}}$ > $\boxed{\text{Clear Command Window}}$.

- `%`: Comment symbol; ignore whole line following it.

  e.g.
  ```
  >> % This is a comment!
  ```

- When you add a *semicolon* at the end of command, the result of the command does not appear.

  e.g.
  ```
  >> 3 + 5                                   >> 3 + 5;
  ans =                                      >>
       8
  ```

1.2 Creating a Matrix

- To create a matrix, we use square brackets `[` and `]`.

  e.g. `[ 3 5 ]` $= \begin{bmatrix} 3 & 5 \end{bmatrix}$ and `[3; 5]` $= \begin{bmatrix} 3 \\ 5 \end{bmatrix}$.

  Square brackets can also concatenate two matrices.

  e.g.
  ```
  >> a = [1 2];                              >> a = [1; 2];
  >> b = [a; 3 4]                            >> b = [a [3;4]]
  b =                                        b =
       1     2                                    1     3
       3     4                                    2     4
  ```

- `zeros`: Create a matrix of zeros. `zeros(n)` returns an `n`-by-`n` matrix of zeros. `zeros(m,n)` returns an `m`-by-`n` matrix of zeros.

- `ones`: Create a matrix of all ones. `ones(n)` returns an `n`-by-`n` matrix of ones. `ones(m,n)` returns an `m`-by-`n` matrix of ones.

- To replace an entry of a matrix, we use round brackets `(` and `)`.

  e.g. `a(1,2)=3` means we replace first row and second column of a matrix a by '3'.

- `eye`: Create an identity matrix. `eye(n)` returns the `n`-by-`n` identity matrix. `eye(m,n)` returns an `m`-by-`n` matrix with 1's on the diagonal and 0's elsewhere.

- `diag`: Create a diagonal matrix. Position of the diagnal can be specified.

e.g. `diag([3 5])` or `diag([3 5],0)` create $\begin{bmatrix}3 & 0\\0 & 5\end{bmatrix}$. While `diag([3 5],1)` and `diag([3 5],-1)` create

$$\begin{bmatrix}0 & 3 & 0\\0 & 0 & 5\\0 & 0 & 0\end{bmatrix}, \quad \begin{bmatrix}0 & 0 & 0\\3 & 0 & 0\\0 & 5 & 0\end{bmatrix}.$$

respectively.

Remark 1. Creating a matrix by giving its entries one at a time is *undesirable*; it takes much time. Instead, you can use *built-in commands.*

e.g.
```
a=zeros(2,2)
a(1,1)=3
a(2,2)=5
```
can be replaced by `a=diag([3 5])`. Both of them give $a = \begin{bmatrix}3 & 0\\0 & 5\end{bmatrix}$.

- `linspace`: Create a linearly spaced vector. `linspace(a,b,n)` returns a row vector of `n` points linearly spaced between and including `a` and `b`. `linspace(a,b)` is equal to `linspace(a,b,100)`.

  The colon operator '`:`' do similar task; instead we should give *increment* in the operator.

  e.g.
  ```
  >> linspace(1,6,3)
  ans =
      1.0000    3.5000    6.0000
  ```
  e.g.
  ```
  >> 1:2.5:6                          >> 1:6
  ans =                               ans =
      1.0000    3.5000    6.0000          1     2     3     4     5     6
  ```

1.3 Manipulating Matrices

- A *colon* represents a whole row or column. Also you can use it to replace specific row or column.

  e.g. `a(:,1)` returns the first column of matrix a. Also `a(2,:)` returns the second row of a.

  e.g.
  ```
  >> a = zeros(2,2);
  >> a(:,2) = [3; 5]
  a =
       0     3
       0     5
  ```

- `size`: Return size of a matrix. `size(a)` returns the sizes of each dimension of matrix a. `size(a,dim)` returns the size of the `dim`-th dimension of a.

  e.g. When `a=zeros(2,3)`, `size(a)` returns $\begin{bmatrix}2 & 3\end{bmatrix}$. Also `size(a,2)` returns 3.

- `length`: Returns the size of the longest dimension of a matrix.

  e.g. `length([1 2 3])` and `length(zeros(2,3))` return 3.

- `find`: Find indices of *nonzero* elements. Logical or relational operators also can be used to find indices satisfying certain condition.

e.g.

```
>> a = [1 0 1; 1 0 0]
a =
     1     0     1
     1     0     0
>> ind = find(a)
ind =
     1
     2
     5
>> a(ind) = [1 2 4]
a =
     1     0     4
     2     0     0
```

```
>> a = diag([2 4 8])
a =
     2     0     0
     0     4     0
     0     0     8
>> ind = (a >= 4)
ind =
     0     0     0
     0     1     0
     0     0     1
>> a(ind) = 4
a =
     2     0     0
     0     4     0
     0     0     4
```

- `'`: Returns the conjugate transpose of a matrix. (The operator `.'` returns the unconjugated complex transpose of a matrix.)

  e.g. When `a` $= \begin{bmatrix} 1 & 2 \end{bmatrix}$, we have `a'` $= \begin{bmatrix} 1 \\ 2 \end{bmatrix}$.

- `*` vs. `.*`: The operator '`*`' means matrix multiplication. On the other hand, the operator '`.*`' returns multiplication of each entry.

  e.g.

```
>> a = [1 2];
>> a*a
??? Error using ==> mtimes
Inner matrix dimensions must agree.
>> a.*a
ans =
     1     4
```

```
>> a = [1 2];
>> a*a'
ans =
     5
>> a'*a
ans =
     1     2
     2     4
```

- `inv`: Returns the inverse of a matrix.

  e.g.

```
>> a = [1 2; 2 8];
>> inv(a)
ans =
    2.0000   -0.5000
   -0.5000    0.2500
```

```
>> a = [1 2; 2 4];
>> inv(a)
Warning: Matrix is singular to
working precision.
ans =
   Inf   Inf
   Inf   Inf
```

- `\`: Matrix left division. If a is a square matrix, `a\b` $= a^{-1}b$. So the solution of the equation $ax = b$ is $x = a^{-1}b =$ `a\b`.

  e.g.

```
>> a = [1 2; 2 8];
>> b = eye(2);
>> a\b
ans =
    2.0000   -0.5000
   -0.5000    0.2500
```

```
>> a = [1 2; 2 4];
>> b = eye(2);
>> a\b
Warning: Matrix is singular to
working precision.
ans =
  -Inf   Inf
   Inf  -Inf
```

- `/` vs. `./`: The operator '`/`' means matrix right division; `a/b` $= ab^{-1}$. On the other hand, the operator '`./`' returns division of each entry.

e.g.
```
>> a = [1 2; 2 8];
>> b = eye(2);
>> b/a
ans =
    2.0000   -0.5000
   -0.5000    0.2500
```
```
>> a = [1 2; 2 8];
>> b = 2*ones(2);
>> a./b
ans =
    0.5000    1.0000
    1.0000    4.0000
```

- `^` vs. `.^`: The operator '`^`' means matrix power. On the other hand, the operator '`.^`' returns power of each entry.

e.g.
```
>> a = [1 2; 2 4];
>> a^2
ans =
     5    10
    10    20
```
```
>> a = [1 2; 2 4];
>> a.^2
ans =
     1     4
     4    16
```

- Built-in mathematical functions admits matrix as an input; the functions operate element-wise on matrix.

e.g.
```
>> x = linspace(0,pi,5)
x =
         0    0.7854    1.5708    2.3562    3.1416
>> sin(x)
ans =
         0    0.7071    1.0000    0.7071    0.0000
```

1.4 Control Statements and Others

- `disp`: Display text or matrix. `sprintf` command can be used to create a text.

e.g.
```
>> a = [1 2; 4 8];
>> disp(a)
     1     2
     4     8
```
```
>> str = sprintf('The array is %dx%d.',2,3);
>> disp(str)
The array is 2x3.
```

- `for`: Execute statements specified number of times.
  `while`: Repeatedly execute statements *while* condition is true.

e.g.
```
>> for s = [1 5 8 17]
   disp(s)
   end
     1
     5
     8
    17
```
```
>> a = 1;
>> while a < 5
   disp(a)
   a = a + 1;
   end
     1
     2
     3
     4
```

- `if`: Execute statements *if* condition is true.

e.g.
```
>> a = 5;
>> if a == 5
   a = 3
   end
a =
     3
```
```
>> a = 4;
>> if a == 5
   a = 3
   else
   a = 4
   end
a =
     4
```

- `trapz`: Trapezoidal numerical integration.

e.g. Want to compute $\int_0^\pi \sin x \, dx$ (which is exactly two):

```
>> x = linspace(0,pi,100);
>> y = sin(x);
>> trapz(x,y)
ans =
    1.9998
```

- `%{` and `%}`: Multiple line comments. Ignore all lines between them.

  e.g.
  ```
  %{
  Everything is
  ignored in here.
  %}
  ```

1.5 M-File: Script or Function

You can create a text file containing MATLAB code; it is called *M-File* because the filename extension should be '.m'. M-File can be used in two ways: script or function. A *script* is just a series of commands, while a *function* has input and output so that it is a user-defined command.

- To create a M-File script, choose the menu $\boxed{\text{File}}$ > $\boxed{\text{New}}$ > $\boxed{\text{Script}}$ (or use keyboard shortcut Ctrl+N). Then an editor shows up. Type a series of commands in the editor and save it. To execute your script, enter the filename (without extension) in the command window.

  e.g.
  ```
  % example1.m : an approximation of pi      >> example1
  a = 3 + 8/60 + 29/60^2 + 44/60^3;          3.1415926
  str = sprintf('%.7f', a);
  disp(str)
  ```

- To create a function, first create a M-File by choosing the menu $\boxed{\text{File}}$ > $\boxed{\text{New}}$ > $\boxed{\text{Function}}$. Then an editor filled with the following commands shows up:

  ```
  function [ output_args ] = Untitled( input_args )
  %UNTITLED Summary of this function goes here
  %   Detailed explanation goes here

  end
  ```

  Replace `Untitled` by your function name. Also type input/output arguments instead of `input_args` and `output_args` respectively. If you're done, save the file; *filename should be the same as the name of the function.* You can use the function in the command window or other M-Files.

  e.g.
  ```
  % add.m : add two numbers          >> add(3,5)
  function c = add(a,b)              ans =
  c = a + b;                              8
  end
  ```

  A function can return two or more outputs.

  e.g.
  ```
  % swap.m : swap two numbers        >> swap(3,5)
  function [c,d] = swap(a,b)         ans =
  c = b;                                  5
  d = a;                             >> [a,b] = swap(3,5)
  end                                a =
                                          5
                                     b =
                                          3
  ```

- If your function is simple, there is a way to define it without creating a M-File. Using the operator `@`, you can create a function handle.

  e.g.
  ```
  >> sqr = @(x) x.^2;
  ```
  The function defined in this way works in the same way as M-File function does.

  e.g.
  ```
  >> sqr(5)
  ans =
      25
  ```
  Some functions need a function handle as its input. For example, the function `quad`, which integrate a function numerically using adaptive Simpson quadrature, needs a function handle as its first input. To integrate the function `sqr` from 0 to 1, type

  ```
  >> quad(sqr, 0, 1)
  ans =
      0.3333
  ```

  But *when you use a M-File function*, you should use the operator `@`.

  e.g.
  ```
  % cube.m : return the cube of a number
  function y = cube( x )
  y = x.^3;
  end
  ```
  ```
  >> quad(@cube, 0, 1)
  ans =
      0.2500
  ```

1.6 An Example: Implementing Finite Difference Method

In this section, we implement the finite difference method to solve a boundary-value problem

$$y''(x) = 0, \quad 0 < x < 1, \quad y(0) = 0,\ y(1) = 1.$$

1. First we divide the interval $[0, 1]$ into subintervals of size h and let x_i be the mesh points (endpoints of the subintervals). Here we divide the interval with $N = 20$ mesh points:

   ```
   >> N = 20;
   >> x = linspace(0,1,N);
   ```

   Each element of the row vector `x` is a mesh point. Hence the size of subintervals h is

   ```
   >> h = x(2)-x(1);
   ```

2. Now replace the second-order derivative of the problem with the centered-difference formula:

   $$y''(x_i) \simeq \frac{1}{h^2}\big[y(x_{i-1}) - 2y(x_i) + y(x_{i+1})\big].$$

   Then the original problem becomes a system of equations:

   $$\begin{cases} y(x_1) = 0, & \\ -y(x_{i-1}) + 2y(x_i) - y(x_{i+1}) = 0, & 2 \le i \le N-1, \\ y(x_N) = 1. & \end{cases}$$

   The system is expressed in the $(N-2)$-by-$(N-2)$ matrix form $aw = b$ where

   $$a = \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & \ddots & \ddots & \\ & & \ddots & \ddots & -1 \\ & & & -1 & 2 \end{bmatrix}, \quad w = \begin{bmatrix} y(x_2) \\ y(x_3) \\ \vdots \\ y(x_{N-1}) \end{bmatrix}, \quad b = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}.$$

   Using the command `diag`, we can generate the $(N-2)$-by-$(N-2)$ tridiagonal matrix a:

```
>> aupper = diag(-ones(1,N-3),1);     % upper triangular part
>> adiag = diag(2*ones(1,N-2));       % diagonal part
>> alower = diag(-ones(1,N-3),-1);    % lower triangular part
>> a = aupper + adiag + alower;
```

The column vector b has only one nonzero element at the last entry:

```
>> b = zeros(N-2,1);
>> b(end,1) = 1;
```

The command `end` represents last index of row or column.

3. Solve the equation $aw = b$ using the operator `\`:

   ```
   >> w = a\b;
   ```

   Now we add $y(x_1)$ and $y(x_{20})$ to w and find the solution y:

   ```
   >> y = [0; w; 1];
   ```

4. Plot the solution using the `plot` command:

   ```
   >> plot(x,y);
   ```

   Then you can see a straight line. (The analytic solution is $y(t) = t$.)

2 Ordinary Differential Equations

In this section, we will demonstrate how to solve ordinary differential equations. For initial-value problems we will use the function `ode45`; it implements a variable time step Runge-Kutta method of order four to five. Among other ODE solvers in MATLAB, `ode45` is the first solver you try; it fits most of the time. For boundary-value problems we will use the `bvp4c` command.

2.1 Initial-Value Problems for First Order ODEs

Consider the initial-value problem

$$y' = y - t^2 + 1, \quad 0 < t < 2, \quad y(0) = 0.5.$$

To solve the problem, do the followings:

1. Define a function describing the right hand side in terms of `t` and `y`. In our example, the right hand side is simple so that we can define it via function handle:

   ```
   >> odefunc = @(t,y) y - t^2 + 1;
   ```

2. Define a vector describing the time interval.

   ```
   >> tspan = [ 0 2 ];
   ```

3. Define initial condition.

   ```
   >> y0 = 0.5;
   ```

4. Finally use the function `ode45` to solve the equation.

   ```
   >> [t,y] = ode45(odefunc, tspan, y0);
   ```

   The result is contained in two column vectors `t` and `y` of same size.

5. To plot the result, we use the function `plot`.

   ```
   >> plot(t,y)
   ```

2.2 Initial-Value Problems for Higher Order ODEs

Now consider a second order ODE

$$y'' - 2y' + 2y = e^{2t} \sin t, \quad 0 < t < 1, \quad y(0) = -0.4,\ y'(0) = -0.6.$$

To solve the higher order ODEs, we need one more step: convert it to a system of first order equations.

1. Convert the ODE to a system of first order equations.

   Let $y_1 = y$ and $y_2 = y'$. Then we have

$$\begin{cases} y_1' = y_2, \quad y_1(0) = -0.4, \\ y_2' = 2y_2 - 2y_1 + e^{2t} \sin t, \quad y_2(0) = -0.6 \end{cases}$$

   or in a vector form,

$$\frac{d}{dt} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} y_2 \\ 2y_2 - 2y_1 + e^{2t} \sin t \end{bmatrix}, \quad \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} (0) = \begin{bmatrix} -0.4 \\ -0.6 \end{bmatrix}.$$

2. Define a function describing the right hand side in terms of `t` and `y`. Note that the right hand side is a vector and `y` is a *column vector.*

```
% my2ndode.m : defines the RHS of an ODE system
function ydot = my2ndode( t, y )
% Note that y is a column vector
ydot = [ y(2); 2*y(2) - 2*y(1) + exp(2*t) * sin(t) ];
end
```

3. Define a vector describing the time interval.

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

4. Define a *column vector* containing initial conditions.

```
>> y0 = [-0.4; -0.6];
```

5. Finally use the function `ode45` to solve the equation.

```
>> [t,y] = ode45(@my2ndode, tspan, y0);
```

   As in the first order ODE case, `t` is a column vector. But now `y` is a matrix of which columns represent y and y'.

6. To plot the value of y, we use the first column of `y`:

```
>> plot( t, y(:,1) )
```

   Similary, to plot the value of y', we use the second column of `y`:

```
>> plot( t, y(:,2) )
```

Higher order ODEs can be dealt in a similar fashion. The output of your function would have more rows and the result `y` would have more columns.

2.3 Boundary-Value Problems

Consider consider a second-order ODE

$$y'' = 2y^3, \quad -1 < x < 0, \quad y(-1) = \frac{1}{2},\ y(0) = \frac{1}{3}.$$

To solve the boundary value problem, do the followings:

1. Convert the ODE to a system of first order equations.

   Let $y_1 = y$ and $y_2 = y'$. Then we have

$$\frac{d}{dx}\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} y_2 \\ 2y_1^3 \end{bmatrix}, \quad y_1(-1) = \frac{1}{2},\ y_1(0) = \frac{1}{3}.$$

2. Code a function describing the right hand side in terms of `x` and `y`. Note that the right hand side is a vector and `y` is a *column vector.*

```
% mybvp.m : defines the RHS of an ODE system
function dydx = mybvp(x,y)
% Note that y is a column vector
dydx = [ y(2); 2*y(1)^3 ];
end
```

3. Code a function describing the boundary condition. The function should return the residual in the boundary condition. In our case, the boundary condition can be written as

$$\begin{bmatrix} y_1(-1) - \frac{1}{2} \\ y_1(0) - \frac{1}{3} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

   We should give the left hand side of the equation:

```
% mybvpbc.m : defines the BC of an ODE system
function res = mybvpbc(yl,yr)
res = [ yl(1)-1/2; yr(1)-1/3 ];
end
```

   Here `yl` and `yr` represents the values of `y` at the left and right endpoints respectively.

4. Form the initial guess for `bvp4c`. We need to give an initial guess of solution because boundary-value problems might have many solutions.

   Here we give an initial guess $y_1(x) \equiv 1$ and $y_2(x) \equiv 0$ at five equally spaced points in $[-1, 0]$:

   `>> solinit = bvpinit(linspace(-1,0,5),[1 0]);`

5. Finally use the function `bvp4c` to solve the equation.

   `>> sol = bvp4c(@mybvp,@mybvpbc,solinit);` Then `bvp4c` produces a C^1-solution on $[-1, 0]$. Use the function `deval` to evaluate the solution at specific points in the interval:

```
x = linspace(-1,0,20);
y = deval(sol,x);
```

   Then `y` is a matrix of which *rows* represent y and y'.

6. To plot the solution y, we use the first row of `y`:

   `>> plot( x, y(1,:) )`

   Similary, to plot the value of y', we use the second row of `y`:

   `>> plot( x, y(2,:) )`

   Compare the numerical solution with the actual solution $y(x) = 1/(x + 3)$:

```
>> actual_y = 1./(x+3);
>> plot(x,y(1,:), x,actual_y)
```

Higher order ODEs can be dealt in a similar fashion. The output of your function would have more rows and the result `y` would have more rows.

3 Partial Differential Equations in One Spatial Dimension

In this section, we will use MATLAB function `pdepe` to solve initial-boundary value problems for parabolic and elliptic PDEs in one spatial dimension. More precisely, `pdepe` solves PDEs of the form:

$$c(x,t,u,u_x)\,u_t = x^{-m}\big(x^m f(x,t,u,u_x)\big)_x + s(x,t,u,u_x), \quad (x,t) \in (x_l,x_r)\times(t_0,t_f) \tag{1}$$

with initial-boundary conditions

$$u(x,t_0) = u_0(x), \quad x_l \le x \le x_r, \tag{2}$$

$$p(x,t,u) + q(x,t)\,f(x,t,u,u_x) = 0, \quad \text{for } t_0 \le t \le t_f \text{ and } x = x_l, x_r. \tag{3}$$

Specifying constant m and functions c, f, s, p, q, we can solve various types of PDEs as we will see.

3.1 Single Parabolic PDEs

Consider the heat equation

$$\begin{cases} u_t = u_{xx}, \quad (x,t) \in (0,1)\times(0,1), \\ u(x,0) = x^2, \quad 0 \le x \le 1, \\ u(0,t) = 0, \quad 0 \le t \le 1, \\ u(1,t) = 1, \quad 0 \le t \le 1. \end{cases}$$

To solve this initial-boundary value problem, we need following steps:

1. Set the constant m in (1). Usually $m = 0$, but other values of m can be useful. For example, if the solution u is radially symmetric in $\mathbf{R}^d$, the Laplace operator in spherical coordinates is

   $$\Delta u = r^{-m}\big(r^m\, u_r\big)_r, \quad m = d-1.$$

   In our example, $m = 0$.

   ```
   >> m = 0;
   ```

2. Define your PDE by specifying the functions c, f, s in (1).

   In our example, we have

   $$c(x,t,u,u_x) = 1, \quad f(x,t,u,u_x) = u_x, \quad s(x,t,u,u_x) = 0.$$

   Hence a M-File function describing the above is

   ```
   % heateqn.m : defines the heat equation for "pdepe"
   function [ c,f,s ] = heateqn( x,t,u,dudx )
   c = 1;
   f = dudx;
   s = 0;
   end
   ```

3. Define the initial condition u_0 in (2).

   In our example, $u_0(x) = x^2$ so that a M-File function describing it is

   ```
   % icfun1.m : defines an initial condition for "pdepe"
   function u0 = icfun1( x )
   u0 = x^2;
   end
   ```

4. Define the boundary conditions by specifying the functions p, q in (3). Note that the flux function f is already defined.

   In our example, we have

$$p(0,t,u) = u, \quad p(1,t,u) = u-1, \quad q(0,t) = q(1,t) = 0.$$

   Hence a MATLAB function describing the boundary conditions is

```
% bcfun1.m : defines a boundary conditions for "pdepe"
function [ pl,ql,pr,qr ] = bcfun1( xl,ul,xr,ur,t )
pl = ul;
ql = 0;
pr = ur-1;
qr = 0;
end
```

   Note that `pl`, `ql` and `ul` stand for $p(x_l,t,u)$, $q(x_l,t,u)$ and $u(x_l,t)$ respectively. Other variables are defined similarly.

5. Define space mesh and time interval.

```
>> x = linspace(0,1,20);
>> t = linspace(0,1,10);
```

6. Finally use the function `pdepe` to solve the PDE.

```
>> u = pdepe(m, @heateqn, @icfun1, @bcfun1, x, t);
```

   Then `u` is a 10-by-20 matrix; each *row* of `u` represents a solution at a specific time.

7. To observe evolution of the solution, we can use the function `surf`.

```
>> surf( x,t,u )
```

   To plot the solution profile at a specific time, we need to access a row of `u`. For example, we can plot the final profile of the solution as the following:

```
>> plot( x, u(end,:) )
```

3.2 System of Parabolic PDEs

Consider a system of reaction-diffusion equations

$$\begin{cases} u_{1,t} = 2u_{1,xx} + u_1\,(1-u_1-u_2), \quad (x,t) \in (0,1)\times(0,1), \\ u_{2,t} = u_{2,xx} + u_2\,(1-u_1-u_2), \quad (x,t) \in (0,1)\times(0,1), \\ u_1(x,0) = u_2(x,0) = 0.1\chi_{(0,4,0.6)}, \quad 0 \le x \le 1, \\ u_{1,x} = u_{2,x} = 0, \quad \text{for } 0 \le t \le 1 \text{ and } x = 0,1 \end{cases}$$

To solve the system, we replace the functions in (1) by vectors. Detailed procedure is the following:

1. Rewrite the PDE in a expected form by `pdepe`.

   In our example, the vector form of the equation is

$$\begin{bmatrix}1\\1\end{bmatrix} .* \frac{\partial}{\partial t}\begin{bmatrix}u_1\\u_2\end{bmatrix} = x^{-0}\frac{\partial}{\partial x}\begin{bmatrix}x^0\,2u_{1,x}\\x^0\,u_{2,x}\end{bmatrix} + \begin{bmatrix}u_1\,(1-u_1-u_2)\\u_2\,(1-u_1-u_2)\end{bmatrix}.$$

   Hence for a column vector $u = \begin{bmatrix}u_1\\u_2\end{bmatrix}$, we have

$$c(x,t,u,u_x) = \begin{bmatrix}1\\1\end{bmatrix}, \quad f(x,t,u,u_x) = \begin{bmatrix}2\\1\end{bmatrix} .* u_x, \quad s(x,t,u,u_x) = \begin{bmatrix}u_1\,(1-u_1-u_2)\\u_2\,(1-u_1-u_2)\end{bmatrix}$$

   Also $m = 0$. M-File function describing the system of PDEs is

```
% system1.m : defines a system of PDEs for "pdepe"
function [c,f,s] = system1(x,t,u,dudx)
c = [1; 1];
f = [2; 1] .* dudx;
s = [u(1)*(1-u(1)-u(2)); u(2)*(1-u(1)-u(2))];
end
```

2. Define the initial condition u_0.

   In our example, $u_0(x) = 0.1\chi_{(0.4,0.6)} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ so that a M-File function describing it is

```
% icfun2.m : defines an initial condition for "pdepe"
function u0 = icfun2(x)
u0 = [0; 0];
if ((x > 0.4) && (x < 0.6))
    u0 = [0.1; 0.1];
end
end
```

3. Define the boundary conditions by specifying the functions p, q in (3).

   In our example, the boundary condition is

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} .* f = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \text{at } x = 0, 1.$$

   Hence we have

$$p(0,t,u) = p(1,t,u) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad q(0,t,u) = q(1,t,u) = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

   Hence a MATLAB function describing the boundary conditions is

```
% bcfun2.m : defines a boundary conditions for "pdepe"
function [pl,ql,pr,qr] = bcfun2(xl,ul,xr,ur,t)
pl = [0; 0];
ql = [1; 1];
pr = [0; 0];
qr = [1; 1];
end
```

   `pl`, `ql` and `ul` stand for $p(x_l, t, u)$, $q(x_l, t, u)$ and $u(x_l, t)$ respectively. Other variables are defined similarly.

4. Define space mesh and time interval.

```
>> x = linspace(0,1,20);
>> t = linspace(0,1,10);
```

5. Finally use the function `pdepe` to solve the PDE.

   In our example, $m = 0$ so that the first argument is zero.

```
>> u = pdepe(0, @system1, @icfun2, @bcfun2, x, t);
```

   Then `u` is a 10-by-20-by-2 matrix; each dimension determines time, space and solutions u_1, u_2 respectively.

6. To observe evolution of the solutions, we can use the function `surf`. For example, to draw evolution of u_1, type

   ```
   >> surf( x,t,u(:,:,1) )
   ```

   We can also plot the solution profile at a specific time. For example, to plot the final profile of u_2, type

   ```
   >> plot( x, u(end,:,2) )
   ```

4 Partial Differential Equations in Two Spatial Dimensions

MATLAB provides an addon to solve PDEs in two dimensions; it is *Partial Differential Equation Toolbox.* (Because it is an addon, you need to install it.) In the followings, we will learn how to use the PDE Toolbox.

4.1 GUI of PDE Toolbox

PDE Toolbox provides graphic user interface so that we can easily draw domains. To run the toolbox, type `pdetool` in the command windows:

```
>> pdetool
```

Then a new window (PDE Toolbox) shows up. Consider the Poisson's equation

$$\begin{cases} -\Delta u = x^2, & (x,y) \in (0,2)\times(0,1), \\ \quad u = 0 & \text{on the boundary of } (0,2)\times(0,1). \end{cases}$$

To solve the PDE, we need the following steps:

1. Select type of your problem. In $\boxed{\text{Options}}$ > $\boxed{\text{Application}}$ menu, you can choose problem type.

   In our example, we choose the default "Generic Scalar".

2. Select axes and draw the domain.

   First make it sure the axes contains your domain by choosing $\boxed{\text{Options}}$ > $\boxed{\text{Axes Limits}}$. In Axes Limits dialog, you can select X-axis range and Y-axis range. In our example, X-axis range `[-0.5 2.5]` and Y-axis range `[-0.5 1.5]` is enough.

   Using $\boxed{\text{Draw}}$ Menus (or first five icons), we can draw rectangles, ellipses and polygons. For example, in $\boxed{\text{Rectangle/Square}}$ Menu, you can create a rectangle by dragging with left-click. Dragging with right-click allows you to create a square. Also we can modify an objectby double-clicking it.

   Remark 2. Each object you drawed has name, e.g., `R1`, `SQ1`. Using $\boxed{\text{Set formula}}$, you can create sophisticated domains from other objects. For example, you can add a rectangle `R1` and a circle `C1` by giving set formula `R1 + C1`. Also set minus `-` and set intersection `*` can be used.

   You can also draw objects in the command windows using commands `pderect`, `pdecirc`, `pdeellip`, and `pdepoly`.

   In our example, to draw the domain, it suffices to type

   ```
   >> pderect([0 2 0 1])
   ```

   in the command window.

3. Give boundary conditions.

   Choose $\boxed{\text{Boundary}}$ > $\boxed{\text{Boundary Mode}}$ (or $\boxed{\partial\Omega}$ icon). Then the boundary of domains changes into red arrows. Double-clicking one of the red arrow, you can specify boundary condition on it. Also by shift-click you can choose many arrows at once; then red arrows turn into black arrows.

By double-clicking one of black arrow, Boundary Condition dialog shows up and you can choose Neumann or Dirichlet condition.

In our example, you need to choose every arrows at once by shift-click. Also in the Boundary Condition dialog choose Dirichlet condition. It needs the form

$$hu = r.$$

Because we want to specify the boundary condition $u = 0$, the values of `h` and `r` should be `1` and `0` respectively.

Remark 3. In fact, red arrow means Dirichlet condition. If you give Neumann condition, arrow becomes blue.

4. Specify the PDE.

   Choose $\boxed{\text{PDE}}$ > $\boxed{\text{PDE Specification}}$ menu (or $\boxed{\text{PDE}}$ icon). Then PDE Specification dialog shows up and you can choose your PDE. There are four kinds of PDEs you can choose: Elliptic, Parabolic, Hyperbolic and Eigenmodes.

   In our example, we choose "Elliptic". Required form is

   $$-\nabla \cdot (c\nabla u) + au = f.$$

   So we set the values of `c`, `a` and `f` by `1`, `0`, and `x.^2`. Note that x^2 should be written as `x.^2` because `x` is a vector in the toolbox.

5. Create mesh.

   PDE Toolbox uses finite element method so that mesh should be specified. To initialize mesh, choose $\boxed{\text{Mesh}}$ > $\boxed{\text{Initialize Mesh}}$ Menu (or triangle icon). To refine mesh, choose $\boxed{\text{Mesh}}$ > $\boxed{\text{Refine Mesh}}$ Menu (or icon of four triagles).

6. Solve the PDE.

   Choose $\boxed{\text{Solve}}$ > $\boxed{\text{Solve PDE}}$ menu (or $\boxed{=}$ icon). Then the solution is plotted. To change plot options, choose $\boxed{\text{Plot}}$ > $\boxed{\text{Parameters}}$. In Plot Selection dialog, you can select plot type such as Contour and 3-D plot.

You can save the result as a M-File. If you execute the M-File in the command window, PDE Toolbox shows up and the result is recalculated.

But the save file generated by the toolbox heavily depends on the toolbox so it is inconvenient to use the result in other purpose. In the following sections, we will learn how to solve PDEs without using interface of the toolbox.

4.2 Single Elliptic PDEs

In this section, we will program a M-File script solving the Poisson's equation

$$\begin{cases} -\Delta u = x^2, & (x, y) \in (0, 2) \times (0, 1), \\ \quad u = 0 & \text{on the boundary of } (0, 2) \times (0, 1). \end{cases}$$

1. Generate M-Files for specification of geometry and boundary condition.

   (a) Using GUI of the PDE Toolbox, select problem type ("Generic Scalar"), draw the domain and give boundary conditions. (As we did in Section 4.1).

   (b) Now choose $\boxed{\text{Boundary}}$ > $\boxed{\text{Export Decomposed Geometry, Boundary Cond's...}}$ to export two matrices `g` and `b` into workspace. Using commands `wgeom` and `wbound`, you can generate M-Files to specify geometry and boundary condition.

   In our example, type

```
>> wgeom(g, 'myrectangleg');
>> wbound(b, 'myrectangleb');
```

Then you can see `myrectangleg.m` and `myrectangleb.m` are generated.

(c) End the PDE toolbox. After we generated above two files, we don't need GUI anymore.

2. Create triangular mesh.

   To initialize mesh, we use the command `initmesh` with geometry M-File. In our example, type

   ```
   >> [p,e,t] = initmesh('myrectangleg');
   ```

   Here `p`, `e` and `t` mean point, edge and triangle matrix respectively. The first and second rows of `p` contain x- and y-coordinates of the points in the mesh.

   To refine a mesh, use `refinemesh` with geometry M-File. In our example, type

   ```
   >> [p,e,t] = refinemesh('myrectangleg', p,e,t);
   ```

3. Specify the PDE. Rewrite the PDE in the form

   $$-\nabla \cdot (c\nabla u) + au = f.$$

   In our example, `c` and `a` can be easily determined.

   ```
   >> c = 1;
   >> a = 0;
   ```

   But the forcing term x^2 is not constant so that we should be careful to set the term. A simple way is to use a string with symbols. In our example, type

   ```
   >> f = 'x.^2';
   ```

   Here `x` represents x- coordinate.

   Remark 4. For the elliptic case, the toolbox *admits nonlinearity*; we can use u, u_x, u_y in the coefficients or forcing term. `u`, `ux`, `uy` represents them respectively. For example, if your forcing term is u^2, it suffices to type `f = 'u.^2';`.

   Remark 5. A hard way to specify the PDE is to construct the coefficients or forcing terms.

   In our example, because the first row of `p` gives x-coordinate, it seems `f = p(1,:).^2;` would work. But it does NOT work because the coefficients or forcing terms must be a row vector of *values at midpoints* of the triangles. `pdeintrp` helps to interpolate from node data to triangle midpoint data. Hence a proper code is

   ```
   >> ftemp = p(1,:).^2;              % a row vector
   >> f = pdeintrp(p,t, ftemp');      % need a column vector as input
   ```

4. Solve the PDE. `pdenonlin` solves a nonlinear elliptic PDE; its first argument is a M-File specifying boundary condition. In our example, type

   ```
   >> u = pdenonlin('myrectangleb', p,e,t, c,a,f);
   ```

5. Plot the solution. `pdesurf` draws the solution. Type

   ```
   >> pdesurf(p,t,u);
   ```

   `pdecont` allows contour plot:

   ```
   >> pdecont(p,t,u);
   ```

   To plot a triangular mesh, type

   ```
   >> pdemesh(p,e,t);
   ```

Remark 6. We can interpolate the solution from triangular mesh to rectangular grid using `tri2grid`. First set rectangular coordinates in $(0,2) \times (0,1)$:

```
>> x = linspace(0,2,30);
>> y = linspace(0,1,30);
```

Then interpolate the solution to rectangular grid:

```
>> uxy=tri2grid(p,t,u,x,y);
```

No we can use `trapz` to compute integration of the solution:

```
trapz( y', trapz(x,uxy) )
```

4.3 Single Parabolic PDEs

In this section, we will program a M-File script solving the heat equation with source

$$\begin{cases} u_t - \Delta u = t(x^2+y^2), \quad \text{where } x^2+y^2 < 1 \text{ and } 0 < t < 1, \\ u(x,y,0) = 1, \quad \text{where } x^2+y^2 < 1, \\ \mathbf{n} \cdot \nabla u = 0 \quad \text{on the boundary.} \end{cases}$$

Here $\mathbf{n}$ is the outward normal vector; so we are giving Neumann boundary condition.

1. Generate M-Files for specification of geometry and boundary condition.

   (a) Using GUI of the PDE Toolbox, select problem type ("Generic Scalar"), draw the domain and give boundary conditions. (As we did in Section 4.1).

   In our example, we need a circular domain of radius one centered at the origin:

   ```
   >> pdecirc(0,0,1)
   ```

   Also we should give Neumann boundary condition. The toolbox specifies Neumann conditions in the form

   $$\mathbf{n} \cdot (c\nabla u) + qu = g.$$

   Because c will be specified by the PDE (as $c = 1$ in our case), q and g are only variables to be determined. In our case, `q = 0` and `g = 0`.

   (b) Now choose $\boxed{\text{Boundary}}$ > $\boxed{\text{Export Decomposed Geometry, Boundary Cond's...}}$ to export two matrices `g` and `b` into workspace. Using commands `wgeom` and `wbound`, you can generate M-Files to specify geometry and boundary condition.

   In our example, type

   ```
   >> wgeom(g, 'mycircleg');
   >> wbound(b, 'mycircleb');
   ```

   Then you can see `mycircleg.m` and `mycircleb.m` are generated.

   (c) End the PDE toolbox. After we generated above two files, we don't need GUI anymore.

2. Create triangular mesh.

   To initialize mesh, we use the command `initmesh` with geometry M-File. In our example, type

   ```
   >> [p,e,t] = initmesh('mycircleg');
   ```

   Here `p`, `e` and `t` mean point, edge and triangle matrix respectively. The first and second rows of `p` contain x- and y-coordinates of the points in the mesh.

   To refine a mesh, use `refinemesh` with geometry M-File. In our example, type

   ```
   >> [p,e,t] = refinemesh('mycircleg', p,e,t);
   ```

3. Specify the initial data.

   In our example, there are three ways to specify the initial data.

   (a) Because the initial data is constant, the following works:
   ```
   >> u0 = 1;
   ```
   (b) You may create a column vector of ones:
   ```
   >> u0 = ones(size(p,2), 1);
   ```
   Note that `size(p,2)` means number of columns of `p`, i.e., number of points in mesh.

   (c) Using `find`, you can specify initial conditions on specific portion of domain.

   In our example, we should give the value one on a portion of domain satisfying $x^2 + y^2 \leq 1$ (in fact, it is the whole domain.) To do that first initialize a matrix:
   ```
   >> u0 = zeros(size(p,2), 1);
   ```
   Note that `p(1,:)` and `p(2,:)` mean x- and $y-$ coordinates. Now we find indices of the matrix satisfying $x^2 + y^2 \leq 1$:
   ```
   >> ind = find(p(1,:).^2 + p(2,:).^2 <= 1);
   ```
   Then give the value one on the portion we found:
   ```
   >> u0(ind) = ones(size(ind));
   ```

4. Specify the list of times at which the solution would be solved.

   In our example, we will just solve the PDE at the final time:
   ```
   >> tlist = [0 1];
   ```

5. Specify the PDE. Rewrite the PDE in the form

$$du_t - \nabla \cdot (c\nabla u) + au = f.$$

   In our example, `d`, `c` and `a` can be easily determined.
   ```
   >> d = 1;
   >> c = 1;
   >> a = 0;
   ```
   But the forcing term $t(x^2 + y^2)$ is not constant so that we should be careful to set the term. A simple way is to use a string with symbols. In our example, type
   ```
   >> f = 't*(x.^2 + y.^2)';
   ```
   Note that here `t` represents time (not triangles).

6. Solve the PDE. `parabolic` solves parabolic PDEs; its first argument is a M-File specifying boundary condition. In our example, type
   ```
   >> u1 = parabolic(u0,tlist, 'mycircleb', p,e,t, c,a,f,d);
   ```
   Each column in `u1` is the solution at a time in `tlist`.

7. Plot the solution. `pdesurf` draws the solution. To draw the solution at final time, type
   ```
   >> pdesurf(p,t,u1(:,end));
   ```
   `pdecont` allows contour plot:
   ```
   >> pdecont(p,t,u1(:,end));
   ```
   To plot a triangular mesh, type
   ```
   >> pdemesh(p,e,t);
   ```

4.4 System of Parabolic PDEs

In this section, we will program a M-File script solving a system of parabolic PDEs

$$\begin{cases} u_{1,t} - \Delta u_1 = y u_2, \quad (x,y,t) \in (0,1)\times(0,1)\times(0,1), \\ u_{2,t} - \Delta u_2 = x u_1, \quad (x,y,t) \in (0,1)\times(0,1)\times(0,1), \\ u_1(x,y,0) = 1, \quad (x,y) \in (0,1)\times(0,1), \\ u_2(x,y,0) = x+y, \quad (x,y) \in (0,1)\times(0,1), \\ \mathbf{n}\cdot\nabla u_1 = \mathbf{n}\cdot\nabla u_2 = 0 \quad \text{on the boundary.} \end{cases}$$

Here $\mathbf{n}$ is the outward normal vector.

1. Generate M-Files for specification of geometry and boundary condition.

   (a) Using GUI of the PDE Toolbox, select problem type ("Generic System"), draw the domain and give boundary conditions. (As we did in Section 4.1).
   In our example, we need a square domain:
   ```
   >> pderect([0 1 0 1])
   ```
   Also we should give Neumann boundary condition. The toolbox specifies Neumann conditions in the form
   $$\mathbf{n}\cdot(c\otimes\nabla u) + qu = g, \tag{4}$$
   where
   $$u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}, \quad c = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}, \quad q = \begin{bmatrix} q_{11} & q_{12} \\ q_{21} & q_{22} \end{bmatrix}, \quad g = \begin{bmatrix} g_1 \\ g_2 \end{bmatrix}.$$
   The k-th component of $c\otimes\nabla u$ is
   $$\left(c\otimes\nabla u\right)_k = c\nabla u_k = c\begin{bmatrix} u_{k,x} \\ u_{k,y} \end{bmatrix}$$
   and (4) is equivalent to
   $$\begin{bmatrix} \mathbf{n}\cdot(c\nabla u_1) \\ \mathbf{n}\cdot(c\nabla u_2) \end{bmatrix} + qu = g.$$
   The matrix c will be specified by the PDE; in our case, c is the identity matrix. Our boundary condition is
   $$\begin{bmatrix} \mathbf{n}\cdot\nabla u_1 \\ \mathbf{n}\cdot\nabla u_2 \end{bmatrix} = 0$$
   so we set $q = 0$ and $g = 0$.

   (b) Now choose $\boxed{\text{Boundary}}$ > $\boxed{\text{Export Decomposed Geometry, Boundary Cond's...}}$ to export two matrices `g` and `b` into workspace. Using commands `wgeom` and `wbound`, you can generate M-Files to specify geometry and boundary condition.
   In our example, type
   ```
   >> wgeom(g, 'mysquareg');
   >> wbound(b, 'mysquareb');
   ```
   Then you can see `mysquareg.m` and `mysquareb.m` are generated.

   (c) End the PDE toolbox. After we generated above two files, we don't need GUI anymore.

2. Create triangular mesh.

   To initialize mesh, we use the command `initmesh` with geometry M-File. In our example, type
   ```
   >> [p,e,t] = initmesh('mysquareg');
   ```
   Here `p`, `e` and `t` mean point, edge and triangle matrix respectively. The first and second rows of `p` contain x- and y-coordinates of the points in the mesh.

   To refine a mesh, use `refinemesh` with geometry M-File. In our example, type
   ```
   >> [p,e,t] = refinemesh('mysquareg', p,e,t);
   ```

3. Specify the initial data.

   First obtain number of mesh points `np`:

   ```
   >> np = size(p,2);
   ```

   Then initial data should be a column vector with 2`np` rows; first `np` rows are for u_1 and second `np` rows are for u_2. In our case, initial data of u_1 and u_2 are 1 and $x + y$ respectively so

   ```
   >> u10 = ones(np, 1);        % initial data of u1: a column vector
   >> u20 = (p(1,:)+p(2,:))';   % initial data of u2: a column vector
   ```

   Hence initial data for the system is

   ```
   >> u0 = [u10; u20];
   ```

4. Specify the list of times at which the solution would be solved.

   In our example, we will just solve the PDE at the final time:

   ```
   >> tlist = [0 1];
   ```

5. Specify the PDE. Rewrite the PDE in the form

   $$du_t - \nabla \cdot (c \otimes \nabla u) + au = f.$$

   In our example, we have

   $$d = c = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad a = \begin{bmatrix} 0 & -y \\ -x & 0 \end{bmatrix}, \quad f = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

   But we should be careful; values `d`, `c`, `a` and `f` should be a column vector (not a 2-by-2 matrix). Also every rows should be in the *bracketed expression having same number of columns.*

   ```
   >> d = ['1'; '0'; '0'; '1'];
   >> c = ['1'; '0'; '0'; '1'];
   >> % all rows in the bracketed expression must have the same number of columns
   >> a = [' 0'; '-x'; '-y'; ' 0'];
   >> f = ['0'; '0'];
   ```

   Note that in the first and fourth rows of `a`, we intentionally add a space so that every rows have two columns (text with length two).

6. Solve the PDE. `parabolic` solves parabolic PDEs; its first argument is a M-File specifying boundary condition. In our example, type

   ```
   >> u = parabolic(u0,tlist, 'mysquareb', p,e,t, c,a,f,d);
   ```

   Each column in `u` contains information of u_1 and u_2 at a time in `tlist`; first `np` rows are for u_1 and second `np` rows are for u_2.

7. Plot the solution. `pdesurf` draws the solution.

   To draw u_1 at final time, type

   ```
   >> pdesurf(p,t, u(1:np, end));
   ```

   To draw u_2 at final time, type

   ```
   >> pdesurf(p,t, u(np+1:2*np, end));
   ```

4.5 Nonlinear Parabolic PDEs

In this section, we we will solve the nonlinear heat equation

$$\begin{cases} u_t - \Delta u + (u - x^2)u = 0, \quad \text{where } x^2 + y^2 < 1 \text{ and } 0 < t < 1, \\ u(x, y, 0) = 1, \quad \text{where } x^2 + y^2 < 1, \\ \mathbf{n} \cdot \nabla u = 0 \quad \text{on the boundary.} \end{cases}$$

Here $\mathbf{n}$ is the outward normal vector.

The PDE Toolbox does NOT support to solve *nonlinear* parabolic PDEs. So we employ an expedient; solve *linear* problems many times. Let u_0 be the initial data and solve the linear problem

$$u_t - \Delta u + (u_0 - x^2)u = 0$$

up to small time. Then using u as an initial data, solve a linear problem repeatedly. And we assume a solution given in this way would be close to the solution of the nonlinear problem.

1. Generate M-Files for specification of geometry and boundary condition.

   Our domain and boundary conditions are exactly same as those in Section 4.3. We will use `mycircleg.m` and `mycircleb.m` generated in Section 4.3.

2. Create triangular mesh.

```
>> [p,e,t] = initmesh('mycircleg');
>> [p,e,t] = refinemesh('mycircleg', p,e,t);
```

3. Specify the initial data as a column vector.

```
>> u0 = ones(size(p,2), 1);
```

4. Specify the list of times at which the solution would be solved.

   In our example, we will solve 20 linear PDEs:

```
>> tstep = 20;
>> tlist = linspace(0,1,tstep+1);
```

5. Specify the (linear) PDE. Rewrite the PDE in the form

$$du_t - \nabla \cdot (c\nabla u) + au = f.$$

   In our example, `d`, `c` and `f` can be easily determined.

```
>> d = 1;
>> c = 1;
>> f = 0;
```

   But the nonlinear term a would be replaced by linear term $u_0 - x^2$ and it changes at each time step; so we will define `a` later.

6. Solve the (linear) PDE repeatedly. We need a loop to solve linear PDEs iteratively. Also in each step, the term `a` should be renewed. In our example, the following code works:

```
>> u_old = u0;
>> for k=1:tstep
       atemp = u_old' - p(1,:).^2  % a row vector
       a = pdeintrp(p,t, atemp'); % need a column vector as input
       u1 = parabolic(u_old,tlist(k:k+1), 'mycircleb', p,e,t, c,a,f,d);
       u_old = u1(:,2);            % solution at final time
   end
```

Here `u_old` represents updated initial data; the solution at previous time step. Also note that the coefficients or forcing terms must be a row vector of *values at midpoints* of the triangles. So we used `pdeintrp` to interpolate from node data to triangle midpoint data.

7. Plot the solution. To draw the solution at final time, type

```
>> pdesurf(p,t,u1(:,end));
```

Note that *nonlinear system* of PDEs can be similarly dealt; solve single linear PDEs many times.

4.6 Creating Domains without using GUI

In the previous sections, we used GUI of the PDE Toolbox to generate M-Files specifying domain geometry and boundary conditions. But when the domain is unknown, we cannot generate the files in advance. In this section, we demonstrate how to give the information of domain geometry and boundary conditions without using GUI.

The `initmesh` command uses the output of `decsg` to generate an initial mesh.

We will solve the Poisson's equation

$$\begin{cases} -\Delta u = x^2, & (x,y) \in (0,2)\times(0,1), \\ u = 0 & \text{when } y = 0 \text{ or } y = 1, \\ \mathbf{n}\cdot\nabla u = 0 & \text{when } x = 0 \text{ or } x = 2. \end{cases}$$

1. Create the "geometry description matrix".

   The geometry description matrix `gd` describes objects that you draw in the toolbox. Four types of objects (circle, polygon, rectangle, ellipse) are supported but here we will use a polygon only.

   Each column of `gd` contains information of an object. For a polygon object, first row of `gd` contains "2", and the second row contains the number, n, of points consisting of the polygon. The following n rows contain the x-coordinates of the consecutive points, and the following n rows contain the y-coordinates of the consecutive points.

   In our case, the domain is a rectangle with four consecutive points: $(0,0)$, $(2,0)$, $(2,1)$ and $(0,1)$.

   ```
   >> domain = [0 0; 2 0; 2 1; 0 1];
   ```

   Create the geometry description matrix rearranging the points.

   ```
   >> gd = [2; size(domain,1); domain(:,1); domain(:,2)];
   ```

2. Convert `gd` into "decomposed geometry matrix" using the `decsg` command:

   ```
   >> gl = decsg(gd);
   ```

   Then each column of `gl` contains edge information of the domain.

3. Create the "boundary condition matrix".

   Each column of the boundary condition matrix `b` should give boundary conditions on an edge of the domain; for each column in `gl` there must be a corresponding column in `b`. The format of each column is according to the list below:

   (a) First row contains the dimension of the system. For scalar case, first row should be just "1".
   In our case, first row of `b` should have four columns of ones (there are four edges).

   ```
   >> b = ones(1,4);        % first row
   ```

   (b) Second row contains the number of Dirichlet boundary conditions. For scalar case, "1" implies Dirichlet boundary condition and "0" implies Neumann boundary condition.
   In our case, first and third edges have Dirichlet boundary conditions and other edges have Neumann boundary condition:

   ```
   >> b = [b; 1 0 1 0];     % add second row
   ```

(c) Third and Fourth rows contain the lengths for the strings representing `q` and `g` respectively. Recall that the toolbox requires Neumann conditions of the form

$$\mathbf{n} \cdot (c\nabla u) + qu = g.$$

In our case $q = g = 0$. Considering the value "0" as a string, the length of the string `'0'` is just one:

```
>> b = [b; ones(1,4)];    % add third row
>> b = [b; ones(1,4)];    % add fourth row
```

(d) Fifth and sixth rows contain the lengths for the strings representing `h` and `r` respectively. Recall that the toolbox requires Dirichlet conditions of the form

$$hu = r.$$

In our case, $h = 1$ and $r = 0$. Because the lengths of the string '1' and '0' are equal to one we code

```
>> b = [b; ones(1,4)];    % add fifth row
>> b = [b; ones(1,4)];    % add sixth row
```

(e) In the following rows, we give information of `q`, `g`, `h` and `r` one after another. But we should be careful: the information given by strings should be converted to numeric codes using the `double` command.

In our case, the lengths of them are equal to one so each information has one row. And each row contains numeric codes of the strings representing boundary conditions.

```
>> b = [b; double('0')*ones(1,4)];      % q = '0'
>> b = [b; double('0')*ones(1,4)];      % g = '0'
>> b = [b; double('1')*ones(1,4)];      % h = '1'
>> b = [b; double('0')*ones(1,4)];      % r = '0'
```

(f) There is one important step left. In the fifth and sixths rows, we put the lengths of `h` and `r`. But we must give numeric code of `'0'` *when there is no Dirichlet boundary condition.*

In our case, second and fourth edges have Neumann boundary conditions so that we must give numeric code of `'0'` in those columns of `b`:

```
>> b(5:6,2) = double('0')*ones(2,1);
>> b(5:6,4) = double('0')*ones(2,1);
```

Now we have completed creating the decomposed geometry matrix `gl` and boundary condition matrix `b`. Remaining steps are standard.

4. Initialize and refine triangular mesh using the decomposed geometry matrix `gl`.

```
>> [p,e,t] = initmesh(gl);
>> [p,e,t] = refinemesh(gl, p,e,t);
```

5. Specify the PDE in the form $-\nabla \cdot (c\nabla u) + au = f$:

```
>> c = 1;
>> a = 0;
>> f = 'x.^2';
```

6. Solve the PDE; use the boundary condition matrix `b` as the first argument of `pdenonlin`:

```
>> u = pdenonlin(b, p,e,t, c,a,f);
```

7. Plot the solution.

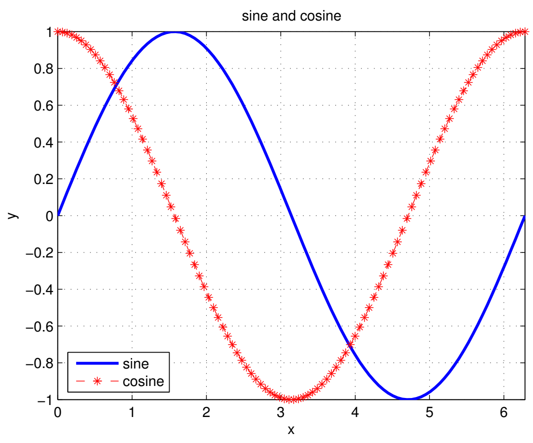
```
>> pdesurf(p,t,u);
```

5 Graph and Animation

In this section, we will learn how to draw 2D and 3D graphs and make an animation movie.

5.1 2D Graphs

In this section, we will draw the following figure:



1. Select mesh points on x-coordinates.

   Here we select 100 mesh points on the interval $[0, 2\pi]$.

   ```
   >> x = linspace(0,2*pi);
   ```

2. Create a figure window. Usually you may skip this step unless you need several figure windows.

   ```
   >> figure(1)
   ```

   The input argument "1" represents index of figure window. The index should be a natural number.

3. Plot the graphs.

   First we draw the sine function:

   ```
   >> plot(x,sin(x), 'b', 'LineWidth',2);
   ```

   The option `'b'` means you draw a blue line. Also the option `'LineWidth'` following a number specifies the width (in points) of the line.

   Next we draw the cosine function. But we should be careful: without specifying, the `plot` command overwrite the graphs. To hold existing graphs on, we type

   ```
   >> hold on
   ```

   Now draw the cosine function and restore the setting.

   ```
   >> plot(x,cos(x), '--r*');
   >> hold off
   ```

   The option `'--r*'` means you draw a dashed red line with asterisk marker.

   To find more options on the line, type `doc LineSpec` in the command window.

4. Display the title:

   ```
   >> title('sine and cosine');
   ```

5. Select axis limits. `axis([xmin xmax ymin ymax])` sets the limits for the x- and y-axis of the current axes.

   ```
   >> axis([0 2*pi -1 1]);
   ```

6. Display the labels:

   ```
   >> xlabel('x')
   >> ylabel('y')
   ```

7. Display the legend on graphs.

   ```
   >> legend('sine','cosine', 'Location','SouthWest')
   ```

   Here the `'Location'` option specifies location of the legend.

8. Add major grid lines to the current axes.

   ```
   >> grid on
   ```

   The `grid off` command removes all grid lines from the current axes.

9. Save the figure as an EPS file: in the Figure window select the $\boxed{\text{File}}$ > $\boxed{\text{Save As...}}$ menu.

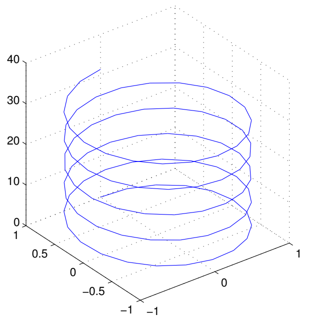
5.2 3D Graphs

MATLAB provides several commands to help drawing 3D graphs. In this section, we give two examples drawing line plot and surface plot.

Example 1. The command `plot3` draws a 3D line.
The following code plots a three-dimensional helix.

```
%% plot3_eg.m
t = linspace(0,10*pi);
plot3(sin(t),cos(t),t)
grid on
axis square
```

The command `axis square` makes the current axes region square (or cubed when three-dimensional).

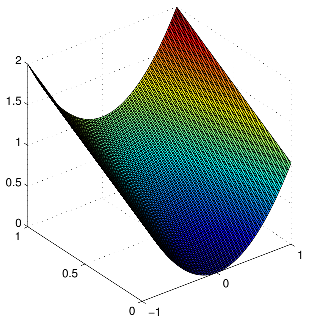


Example 2. The command `surf` draws a shaded surface.
The following code plots the graph of

$$z = x^2 + y, \quad (x, y) \in (-1, 1) \times (0, 1) :$$

```
%% surf_eg.m : 3D shaded surface plot
x = linspace(-1,1);
y = linspace(0,1);
[X,Y] = meshgrid(x,y);
Z = X.^2 + Y;
surf(X,Y,Z)
```

The command `meshgrid` generates `X` and `Y` matrices for three-dimensional plots.



5.3 Animation

In this section, we will create an animated movie and save it as an AVI file.

Consider the function

$$y(t, x) = \sin(\pi t)\sin(x), \quad t \geq 0, \; x \in [0, 2\pi].$$

The profiles of y at time t from $t = 0$ sec to $t = 6$ sec will create an animated movie.

1. Choose fps (frames per second) and create time and space mesh.

   ```
   >> fps = 15;
   >> t = linspace(0,6,6*fps);
   >> x = linspace(0,2*pi);
   ```

2. Get the figure handle and initialize movie variables.

   ```
   >> h = figure(1);
   >> clear M;
   ```

   Then `h` have the handle to the figure object. `M` will contain frames of our movie.

3. Plot and save the frames.

   ```
   >> for i = 1:length(t)
        plot(x, sin(pi*t(i))*sin(x));
        axis([0,2*pi,-1,1]);
        M(i) = getframe(h);
      end
   ```

   `getframe(h)` returns a frame from the figure identified by the handle `h`. Note that if we omit the `axis` command, each frame of the movie would have different axes (no good).

4. Create an AVI file.

   ```
   >> movie2avi(M, 'movie_eg.avi', 'compression','none', 'fps',fps);
   ```

   The option `'compression','none'` is a good choice to avoid the compression codec problem.

6 Installation Problems

When you install old versions of MATLAB 7 in Windows 7, there might be problems. Here are some solutions to fix them.

- Q. When I run the MATLAB, a dialog on java error shows up and MATLAB does not start.

  A. (by Chris Jackson[1]) The problem comes from old versions of java. Find MATLAB.exe and put the file into "Windows 2000" compatibility mode.

- Q. When I run the MATLAB, the program window opens as normal, but then it shutdowns almost immediately.

  A. (by Hans Wurst[2]) If you are using AMD CPU, such problem can occur because MATLAB couldn't find appropriate dll file. Do the followings:

  1. Right click on "My Computer" and select "Properties".
  2. Click on "Advanced system settings" and click on "Advance" tab.
  3. Click on "Environment Variables" button.
  4. Set the variable under "System Variables" : Locate (or add) "Variable" `BLAS_VERSION` and set its "Value" to `c:\matlab7\bin\win32\atlas_Athlon.dll`. (or to where your `blas.spec` is located)

Acknowledgements

This note was prepared for 2012 Korea PDE Winter School; the author is grateful to organizers, especially prof. Yong-Jung Kim. Also the author would like to thank Jung Eun and Seongwon Lee. J. Eun's advice encouraged me to write Section 4.6. S. Lee gave good advice on Section 1.

[1] `http://blogs.msdn.com/b/cjacks/archive/2010/08/12/how-to-fix-older-versions-of-matlab-to-run-on-windows-7.aspx`

[2] `http://www.mathworks.com/matlabcentral/newsreader/view_thread/293427`

References

[1] S. Boettcher, *Solving ODEs and PDEs in MATLAB*, `https://www.math.uni-bremen.de/zetem/cms/media.php/255/SCiE_talk2.pdf`, 2009.

[2] R. L. Burden and J. D. Faires, “Numerical Analysis” (8th ed), Thomson Brooks/Cole, 2005.

[3] D. C. Hanselman and B. L. Littlefield, “Mastering MATLAB 7”, Prentice Hall, 2004.

[4] P. Howard, *Partial Differential Equations in MATLAB 7.0*, `www.math.tamu.edu/~phoward/m401/pdemat.pdf`, 2010.