CS 870 – Numerical Algorithms and Image Processing

Fall 2010

Assignment

Karim Ali

Part a

Figures 1-3 show the curve evolution of the circle at grid sizes **m**=20, 40, and 80 respectively. Table 1 shows the error for each grid size **m**.

To re-generate those plots, you can run the following Matlab function:

evolveCurve('circle', **F**, **tMax**, **m**)

where: **F** = speed (here it's always = 1),
tMax = maximum value for time **t** (here it's always = 0.25)
m = grid size (=20, 40, 80)

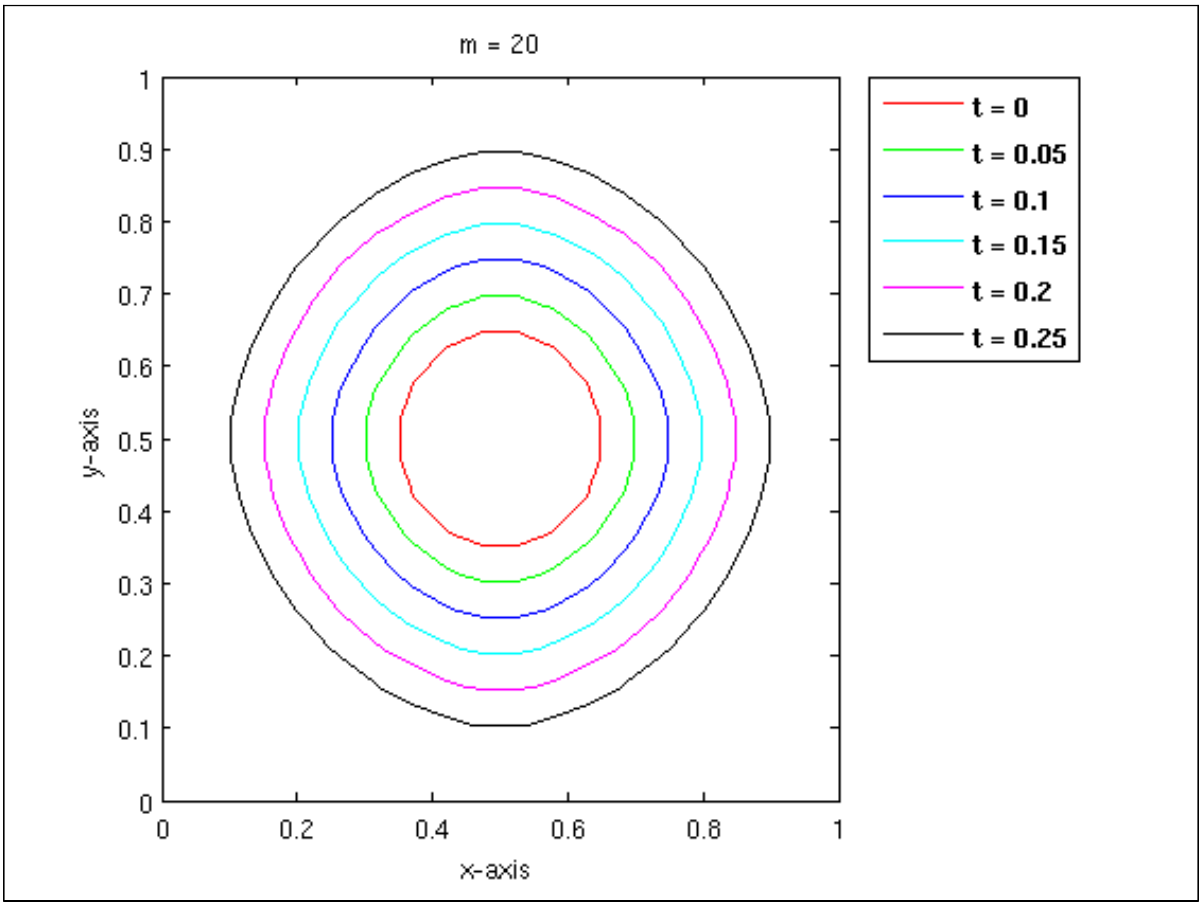

Figure 1

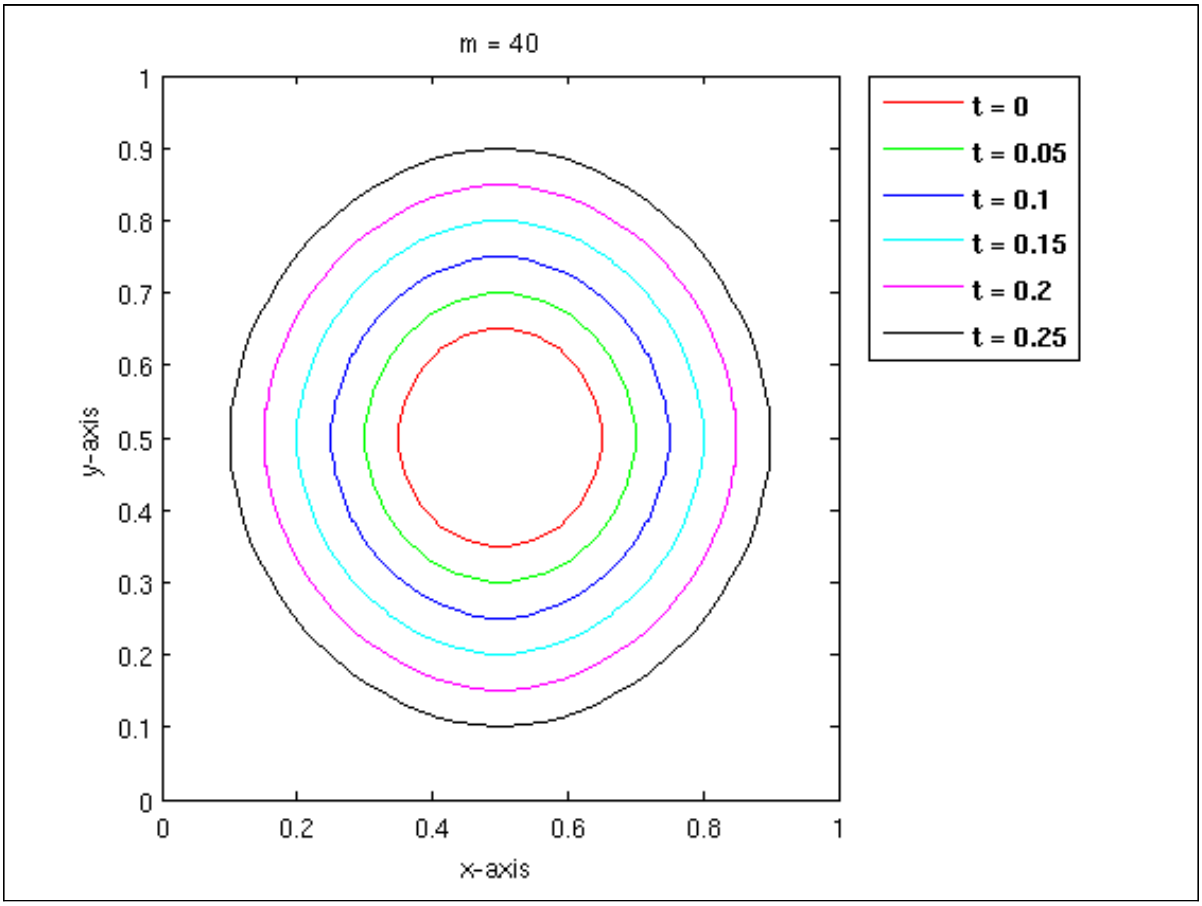
m = 40
y-axis
x-axis
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
0.2
0.4
0.6
0.8
1
t = 0
t = 0.05
t = 0.1
t = 0.15
t = 0.2
t = 0.25

Figure 2

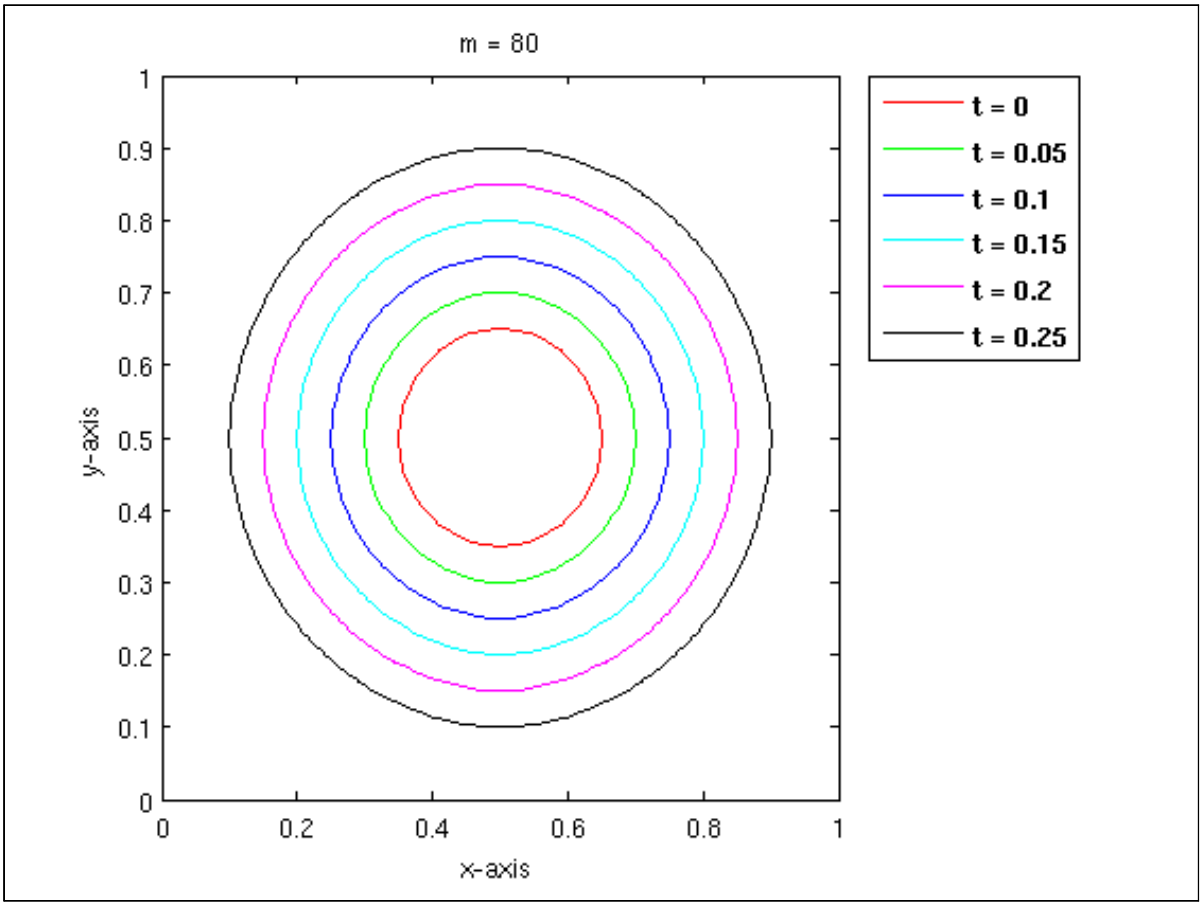
m = 80
y-axis
x-axis
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
0.2
0.4
0.6
0.8
1
t = 0
t = 0.05
t = 0.1
t = 0.15
t = 0.2
t = 0.25

Figure 3

Table 1 shows the error ($A_{exact} - A_{\phi}$) for each grid size **m**.

Grid Size (m)	Error ($A_{exact} - A_{\phi}$)
20	-0.0955004
40	-0.0403150
80	-0.0258936

Table 1

Figure 4 shows a plot for the **error** versus grid size **m**.

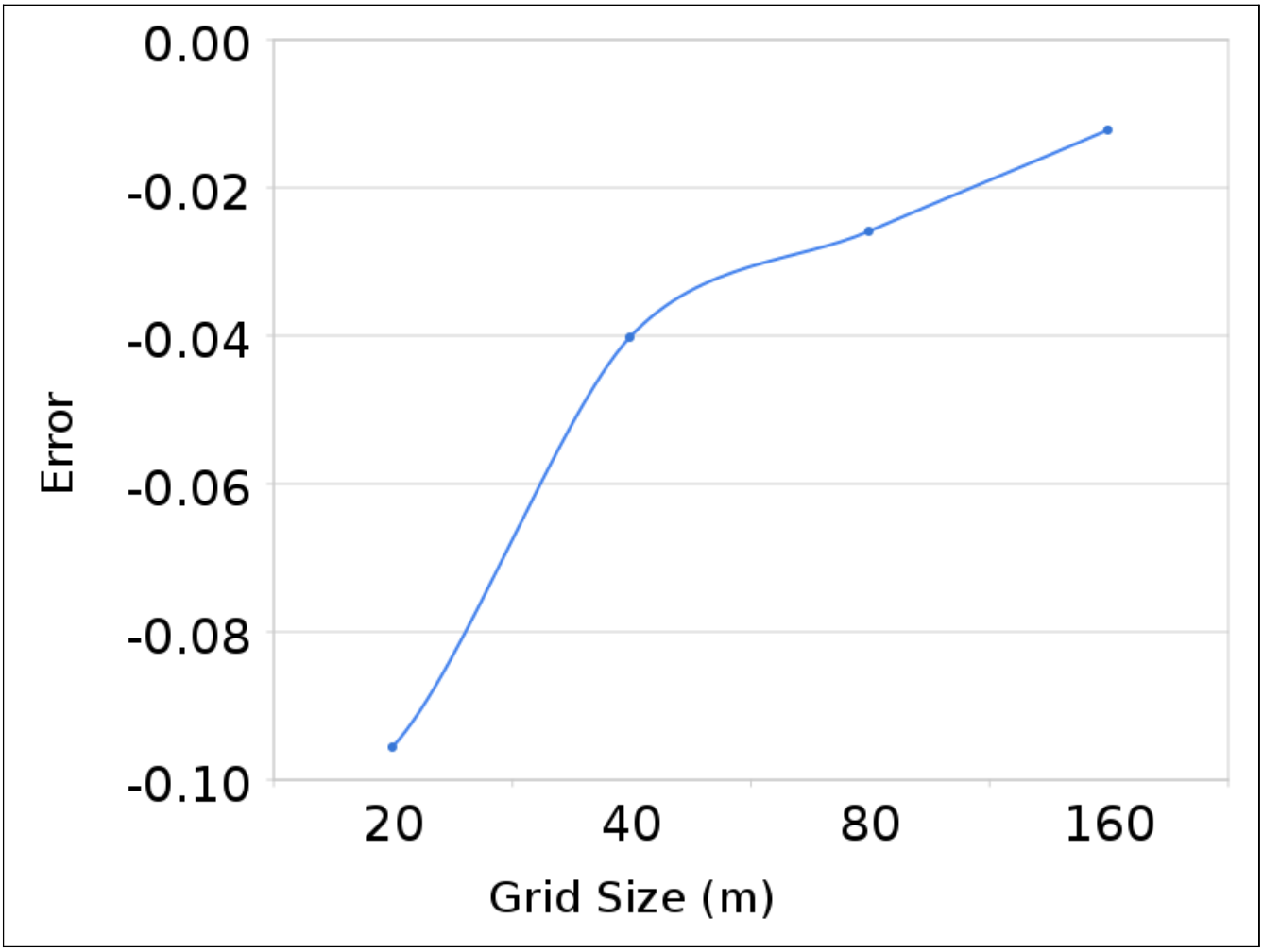

Figure 4

Part b

Figure 5 shows the curve evolution of the dumbbell at grid size **m**=80 . The evolution shows that the dumbbell will split around **t** = 0.1.

To re-generate the plot, you can run the following Matlab function:

evolveCurve('dumbbell', -1, 0.25, 80)

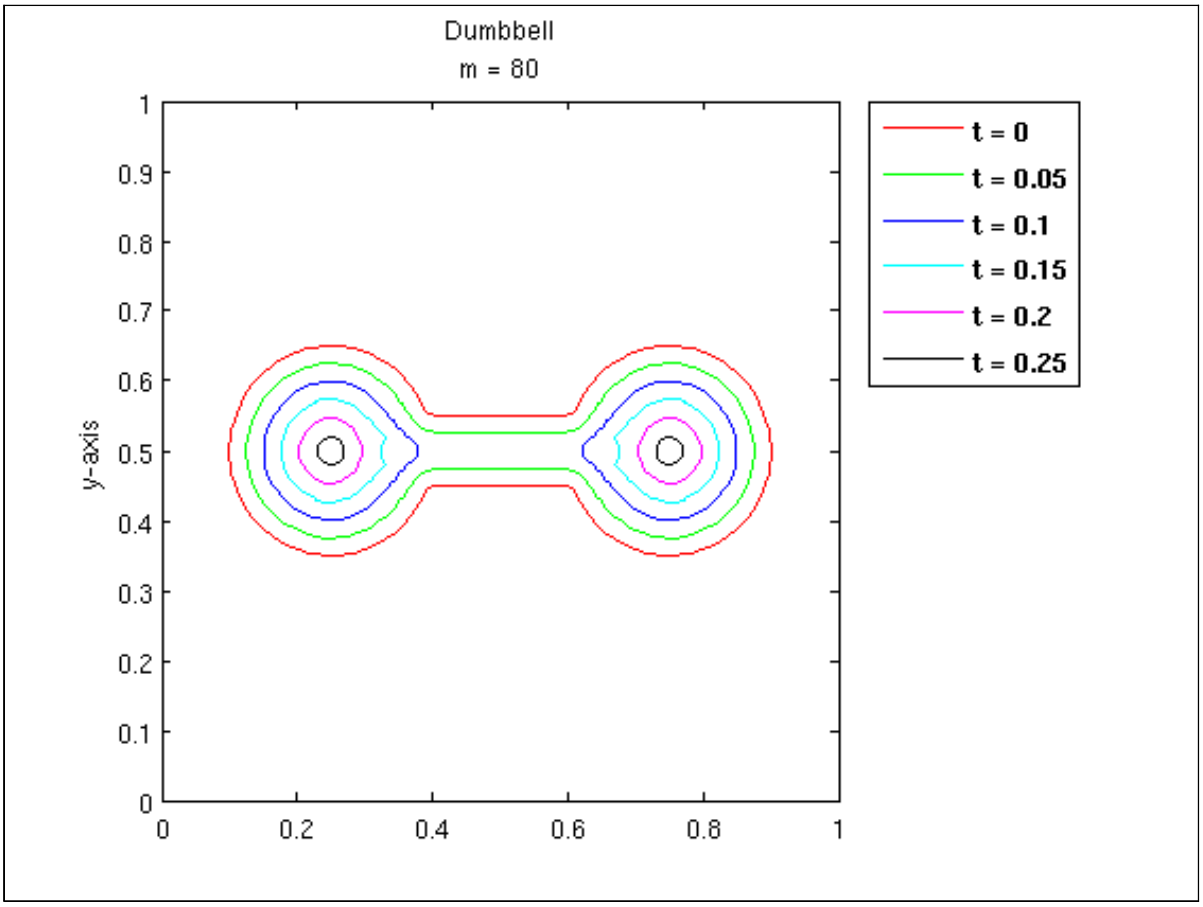

Figure 5

```
% constructGrid(m) : construct a grid of (m+1)x(m+1) cells and populate the
% related values of the grid based on the value of m.
%
% Output parameters:
%   grid = the output grid with all its populated fields
%
% Input parameters:
%   m = the size of the matrix of phi what will be contained in the grid.


function grid = constructGrid(m)

% Where does the grid start and end? This corresponds to the domain of
% omega, i.e (0,0) to (1,1).
grid.lowerLeftCorner = [0 0];
grid.upperRightCorner = [1 1];

% The size of the grid
grid.gridSize = m;

% Calculate dx (step in x direction), and dy (step in y direction)
grid.step = (grid.upperRightCorner - grid.lowerLeftCorner) ./ (grid.gridSize - 1);

% Assign the values along both the x-axis and y-axis. We're getting the
% transpose here so that we get a nice one-column vector instead of a one-row
% vector.
grid.cells = cell(2, 1);
grid.cells{1} = (grid.lowerLeftCorner(1) : grid.step(1) : grid.upperRightCorner(1))';
grid.cells{2} = (grid.lowerLeftCorner(2) : grid.step(2) : grid.upperRightCorner(2))';

% Now generate the x-axis and y-axis. The values in grid.axes will
% determine the values that will be used for the grid lines along both
% axes.
grid.axes = cell(2, 1);
[ grid.axes{:} ] = ndgrid(grid.cells{:});
```

```
% shapeCone: create a level set function phi for a cone. Therefore, when we
% visualize the zero level of phi it will give us a circle.
%
% Output parameters:
%   phi = the level set function for a cirlce given in matrix form
%
% Input parameters:
%   radius = the radius of the circle (i.e. the zero level set of phi).
%   center = [x y] vector for the center of the circle.
%   grid = the grid that will be used for approximation.

function phi = shapeCone(radius, center, grid)

% Since we're allowed to analytically initialize phi, we can use the equation
% given in class to fill out the matrix.
% General formula => phi = sqrt((x − centerX)^2 + (y − centerY)^2) − radius
phi = sqrt( (grid.axes{1} − center(1)).^2 + (grid.axes{2} − center(2)).^2 ) − radius;
```

```
% shapeRectangle: create a level set function phi whose zero level gives
% a rectangle.
%
% Output parameters:
%   initialPhi = mxm matrix that holds the initial values for phi.
%
% Input parameters:
%   grid = the grid that will be used for approximation.
%   lowerLeftCorner = [x y] for the lower left corner of the rectangle.
%   upperRightCorner = [x y] for the upper right corner of the rectangle.

function initialPhi = shapeRectangle(grid, lowerLeftCorner, upperRightCorner)

% A rectangle can be thought of as the intersection of 4 planes.
% Plane 1 = is the plane along the x-axis and right until the upper right corner
% of the rectangle.
%
% Plane 2 = is the plane starting from the lower left corner of the
% rectangle and to the right along the x-axis.
%
% Plane 3 = is the plane along the y-axis and up until the upper right corner of
% the rectangle.
%
% Plane 4 = is the plane starting from the lower left corner of the
% rectangle and up along the y-axis.
plane1 = grid.axes{1} - upperRightCorner(1);
plane2 = lowerLeftCorner(1) - grid.axes{1};
plane3 = grid.axes{2} - upperRightCorner(2);
plane4 = lowerLeftCorner(2) - grid.axes{2};

initialPhi = shapeIntersection(plane1, plane2, plane3, plane4);
```

```
% shapeIntersection : returns the intersection of any number of shapes,
%                      i.e. max(shape1, shape2, ..., etc)
%
% Output parameters:
%   finalShape = the intersection of shape1 with other shapes
%
% Input parameters:
%   shape1 = any shape represented by an mxn matrix
%   varargin = any number of shapes (or zero), each represented by an mxn matrix

function finalShape = shapeIntersection(shape1, varargin)

finalShape = shape1;

for i = 1 : size(varargin, 2)
    finalShape = max(finalShape, varargin{i});
end
```

```
% shapeUnion : returns the union of any number of shapes,
%               i.e. max(shape1, shape2, ..., etc)
%
% Output parameters:
%   finalShape = the union of shape1 with other shapes
%
% Input parameters:
%   shape1 = any shape represented by an mxn matrix
%   varargin = any number of shapes (or zero), each represented by an mxn matrix

function finalShape = shapeUnion(shape1, varargin)

finalShape = shape1;

for i = 1 : size(varargin, 2)
    finalShape = min(finalShape, varargin{i});
end
```

```
% evolveCurve: evlove the given shape
%
% Input parameters:
%   shape = the shape to be evolved. It can be any value of ('circle',
%   'dumbbell').
%   speed = constant speed
%   t0 = initial time
%   tMax = final time
%   m = matrix size
function [phi grid phi0] = evolveCurve(shape, speed, tMax, m)

%-------------------------------------------------------------------------
% Initialize evolution parameters
plotStep = 0.05;            % plot at t = 0, 0.05, 0.1, 0.15, 0.2, and 0.25
t0 = 0;                     % Start at time t = 0

% Don't calculate the error by default
calculateError = false;

% Colors array for plotting
colors = ['r' 'g' 'b' 'c' 'm' 'k'];

% Start time
startTime = cputime;

%-------------------------------------------------------------------------
% Determine which curve to evolve
if(strcmpi(shape, 'circle'))
    grid = constructGrid(m);
    phi0 = shapeCone(0.15, [0.5 0.5], grid);
    calculateError = true;
elseif(strcmpi(shape, 'dumbbell'))
    plotStep = 0.025;
    grid = constructGrid(m);
    circle1 = shapeCone(0.15, [0.25 0.5], grid);
    circle2 = shapeCone(0.15, [0.75 0.5], grid);
    rectangle = shapeRectangle(grid, [0.25 0.45], [0.75 0.55]);
    phi0 = shapeUnion(circle1, rectangle, circle2);
else
    error('??? Error. Unknown shape.');
end

% Initialize phi with phi0
phi = phi0;

%-------------------------------------------------------------------------
% Evolve the curve
t = t0;
deltaT = (grid.upperRightCorner(1) - grid.lowerLeftCorner(1)) / m;

% There's a little round off here, because while testing the code for m =
% 80, the sixth plot was not shown. Therefore, I guessed if I increased the
% terminating condition of the loop, maybe it will be able to plot it. And
% it did!
while(t <= tMax + 100 * eps)
    % plot at t = 0, 0.05, 0.1, 0.15, 0.2, and 0.25
    if(mod(t, plotStep) == 0)
        index = mod(int32(t/plotStep), size(colors, 2)) + 1;
        contour(grid.axes{1}, grid.axes{2}, phi, [0 0], colors(index));
        % Draw on the same plot
        hold on;
    end
    phi = finiteDifference(phi, speed, deltaT, grid);
```

```
    t = t + deltaT;
end

% Create the legend
legendHandle = legend('t = 0', 't = 0.05', 't = 0.1', 't = 0.15', 't = 0.2', 't = 0.25',
'Location', 'NorthEastOutside');
set(legendHandle, 'FontWeight', 'bold');

% Release the hold on the plot
hold off;

if(calculateError)
    % The exact area of the circle if we calculate it analytically.
    % Since F = 1, this means at any given time t, the radius should increase by
    % (t - t0) to keep the ratio dx/dt = 1 (i.e. F = 1). So at tMax the
    % radius would be = 0.15 + tMax - t0.
    exactArea = pi * (0.15 + tMax - t0)^2;

    % The area calcualted using the final phi
    phiArea = dblquad(@(x,y) heaviside(-1*(phi(ceil(x*(m-1)) + 1, ceil(y*(m-1)) + 1))),
grid.lowerLeftCorner(1) , grid.upperRightCorner(1), grid.lowerLeftCorner(2), grid.
upperRightCorner(2));

    fprintf('AreaPhi - AreaExact = %g - %g = %g\n', exactArea, phiArea, exactArea -
phiArea);
end

% End time
endTime = cputime;

% Total execution time
fprintf('Total execution time = %g seconds.\n', endTime - startTime);
```

```
% finiteDifference: calculate the value of phi_n_plus_1 based on the finite
% difference method.
%
% Output parameters:
%   resultingPhi = phi_n_plus_1
%
% Input parameters:
%   phi = phi_n
%   speed = constant speed
%   deltaT = time step
%   grid = the grid used for approximation

function resultingPhi = finiteDifference(phi, speed, deltaT, grid)

% Calculate gplus only if speed > 0, gminus only if speed < 0. Both are
% zero otherwise.
if(speed > 0)
    gplus = (max(DxMinus(phi, grid), 0).^2 + min(DxPlus(phi, grid), 0).^2 + ...
            max(DyMinus(phi, grid), 0).^2 + min(DyPlus(phi, grid), 0).^2).^0.5;
    gminus = 0;
elseif(speed < 0)
    gplus = 0;
    gminus = (max(DxPlus(phi, grid), 0).^2 + min(DxMinus(phi, grid), 0).^2 + ...
             max(DyPlus(phi, grid), 0).^2 + min(DyMinus(phi, grid), 0).^2).^0.5;
else
    gplus = 0;
    gminus = 0;
end

resultingPhi = phi - deltaT * (max(speed, 0) * gplus + min(speed, 0) * gminus);
```

```
% DxMinus: calculate the backward differencing value along the x-axis
%
% Output parameters:
%   dxm = the backward differencing term (DxMinus) as a 2D matrix.
%
% Input parameters:
%   phi_i_j = the original function phi as a 2D matrix.
%   grid = the grid used for approximation

function dxm = DxMinus(phi_i_j, grid)

% To be able to use the matrix form to calculate DxMinus we need to get a
% matrix that is equivalent to phi_i_minus_1_j. This is basically shifting
% all rows in phi_i_j down by 1.
%
% Then, we extrapolate the values for the missing row by calculating the slope
% in the original phi_i_j as follows:
%       phi_i_j(1, y) - phi_i_j(2, y) for all y ranging from 1 to size(phi).
% Then the value of phi_i_minus_1_j(1, y) would be equal to the following:
%       phi_i_j(1, y) + slope(phi_i_j)
% Simplifying this formula gives us the value of phi_i_minus_1_j(1, y) =
%       2 * phi_i_j(1, y) - phi_i_j(2, y) for all y ranging from 1 to size(phi).
%
% Note:
% -----
% We use a similar formula for extrapolation in DxPlus and DyMinus and DyPlus.
phi_i_minus_1_j = circshift(phi_i_j, [1 0]);
phi_i_minus_1_j(1, :) = 2 * phi_i_j(1, :) - phi_i_j(2, :);

dxm = (phi_i_j - phi_i_minus_1_j) ./ grid.step(1);
```

```
% DxPlus: calculate the forward differencing value along the x-axis
%
% Output parameters:
%   dxp = the forward differencing term (DxPlus) as a 2D matrix.
%
% Input parameters:
%   phi_i_j = the original function phi as a 2D matrix.
%   grid = the grid used for approximation

function dxp = DxPlus(phi_i_j, grid)

% To be able to use the matrix form to calculate DxPlus we need to get a
% matrix that is equivalent to phi_i_Plus_1_j. This is basically shifting
% all rows in phi_i_j up by 1.
phi_i_plus_1_j = circshift(phi_i_j, [-1 0]);
phi_i_plus_1_j(end, :) = 2 * phi_i_j(end, :) - phi_i_j(end - 1, :);

dxp = (phi_i_plus_1_j - phi_i_j) ./ grid.step(1);
```

```
% DyMinus: calculate the backward differencing value along the y−axis
%
% Output parameters:
%   dym = the backward differencing term (DyMinus) as a 2D matrix.
%
% Input parameters:
%   phi_i_j = the original function phi as a 2D matrix.
%   grid = the grid used for approximation

function dym = DyMinus(phi_i_j, grid)

% To be able to use the matrix form to calculate DyMinus we need to get a
% matrix that is equivalent to phi_i_j_minus_1. This is basically shifting
% all columns in phi_i_j to the left by 1.
phi_i_j_minus_1 = circshift(phi_i_j, [0 −1]);
phi_i_j_minus_1(:, end) = 2 * phi_i_j(:, end) − phi_i_j(:, end − 1);

dym = (phi_i_j − phi_i_j_minus_1) ./ grid.step(2);
```

```
% DyPlus: calculate the forward differencing value along the y-axis
%
% Output parameters:
%   dyp = the forward differencing term (DyPlus) as a 2D matrix.
%
% Input parameters:
%   phi_i_j = the original function phi as a 2D matrix.
%   grid = the grid used for approximation

function dyp = DyPlus(phi_i_j, grid)

% To be able to use the matrix form to calculate DyPlus we need to get a
% matrix that is equivalent to phi_i_j_plus_1. This is basically shifting
% all columns in phi_i_j to the right by 1.
phi_i_j_plus_1 = circshift(phi_i_j, [0 1]);
phi_i_j_plus_1(:, 1) = 2 * phi_i_j(:, 1) - phi_i_j(:, 2);

dyp = (phi_i_j_plus_1 - phi_i_j) ./ grid.step(2);
```