Proceedings of the 14th International Conference on Nuclear Engineering
ICONE14
July 17-20, 2006, Miami, Florida, USA

EGME 520: Advanced Viscous Fluids

FINAL PROJECT: LID-DRIVEN CAVITY FLOW ANALYSIS

Andrew Bartels
Graduate Student
Fullerton, California, USAy

ABSTRACT

In this technical memo, a lid-driven cavity flow analysis is done in the MATLAB-using the Laminar Flow Code[1] which makes use of the SIMPLE algorithm to solve a steady two-dimensional laminar incompressible flow inside a square cavity whose top wall moves with a constant velocity *U*, in its own plane. The resulting velocity field for three different Reynolds numbers (100, 400, and 1000) were computed and graphed along with the vorticity and streamlines. These results are then compared to previous studies by U. Ghia et al (1982)., O. Botella et al.(1997) and E. Erturk et al. (2005) Other than difficulties with grid resolution and parameter tuning, the results mimicked the results completed in similar studies using supercomputers. Increase in computing power is helping create more feasibility for

INTRODUCTION

Computational Fluid Dynamics (CFD) is a continually growing field studying fluid flows using numerical analysis and data structures to analyze and solve problems that involve fluid flows. In the results and discussion section below the analysis is compared between

RESULTS AND DISCUSSION

For the primary analysis Reynolds numbers (Re) of 100, 400, and 1000 are looked at. For the 400 and 1000 Re results, issues were found with the grid resolution needed with respect to grid sizing, however parameterization tuning helped solve this issue.

In Figure 1: Lid Driven Cavity flow setup (Erturk et al. 2005), the setup for the problem is see as the boundary layer conditions are all set to zero with a steady velocity in the x-direction setup to flow over the lid.

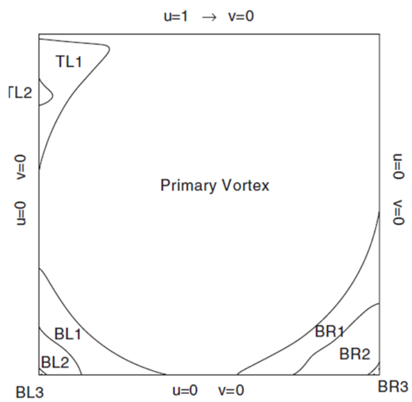


Figure 1: Lid Driven Cavity flow setup (Erturk et al. 2005)

100 REYNOLDS COMPARISON

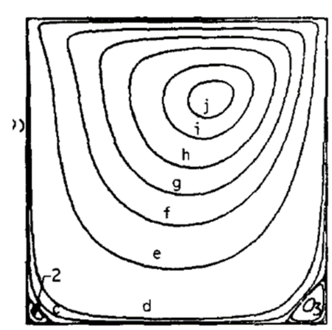


Figure 2: Ghia Re=100; Grid 129 x 129

[1] Mayroal, Salvador. "Laminar_flow.m." updated 29-11-2018

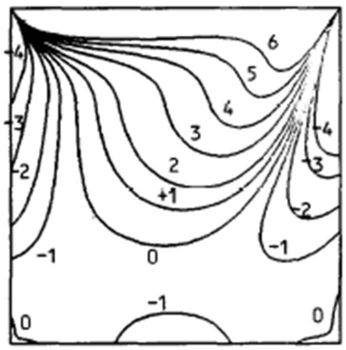


Figure 3: Ghia Vorticity Re=100, Grid 129x129

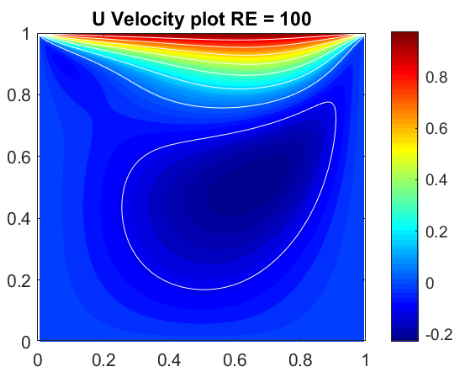


Figure 4: Current Work Velocity Re=100; Grid 100x100

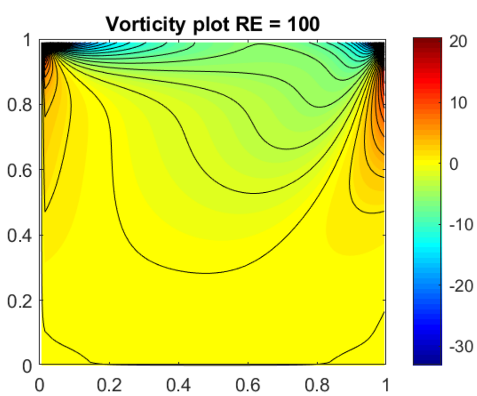


Figure 5: Current Work Vorticity Re=100; Grid 100x100

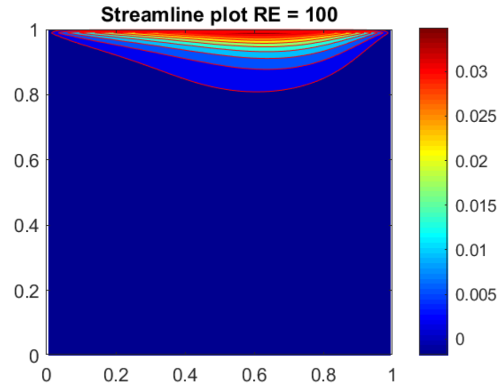


Figure 6: Current Work Streamline Re=100; Grid 100x100

In the 100 Re case, the results were very promising for the comparison between Figures 1 and 2, and Figures 4 and 5. As see, there is little difference if any, and as seen in Figure 7, the residual plot also shows that convergence was reached. These results were repeated for small grid sizes such as 25x25 for the convince of speed as the graphs above took roughly 15 minutes on the higher resolution, but better (i.e. smoother line results) were obtained.

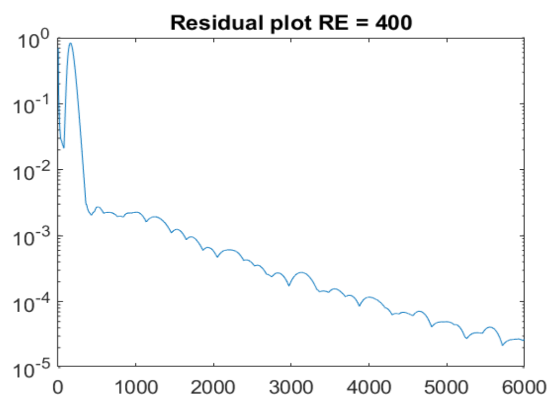


Figure 7: Current Work Residual Plot; Re=100

400 REYNOLDS COMPARISON

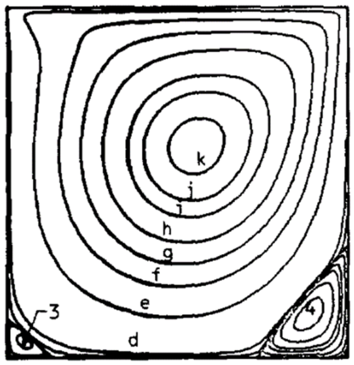


Figure 8: Ghia Streamline Re=400, Grid 129x129

Figure 9: Ghia Vorticity Re=400; Grid 257x257

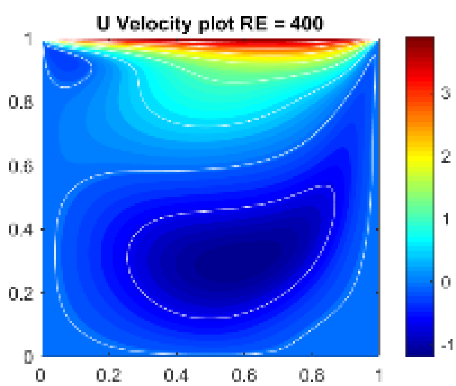


Figure 10: Current Work Velocity Re=400; Grid 185x185

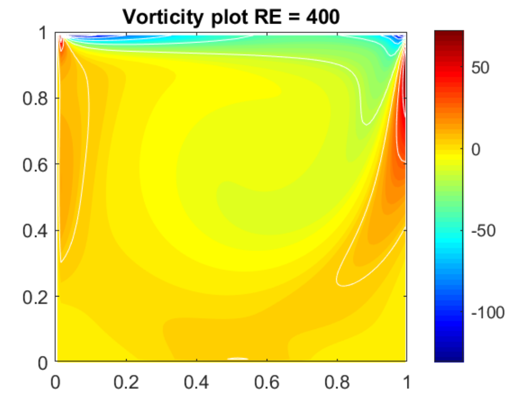


Figure 11: Current Work Vorticity Re=400; Grid 185x185

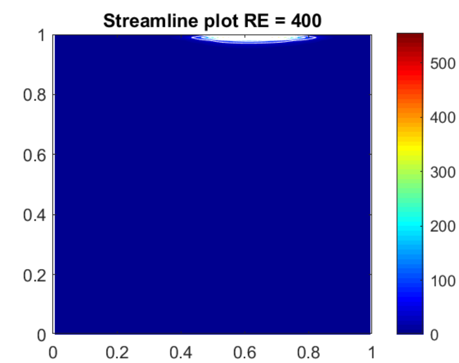


Figure 12: Current Work Streamline Re=400; Grid 185x185

In the comparison of Re=100 and Re=400, it is obvious that the problem isn't fully developed, as all the plots have additional smaller vorticial effects on the corners and bottom of the box. However, there were some issues viewing the streamline plot versus the Re=100 scenario. The residual plot found that the results did converge, but took roughly 10 hours to completely converge. The relaxation parameters of alpha_a and alpha_p remained unchanged at 0.1 and 0.7 respectively. Computationally this scenario used roughly 15 GB of RAM (at peak usage) versus the Re=100 run which consumed a nominal amount of memory.

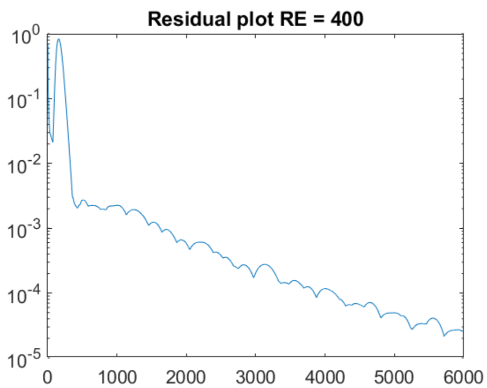


Figure 13: Current Work Residual, Re=400

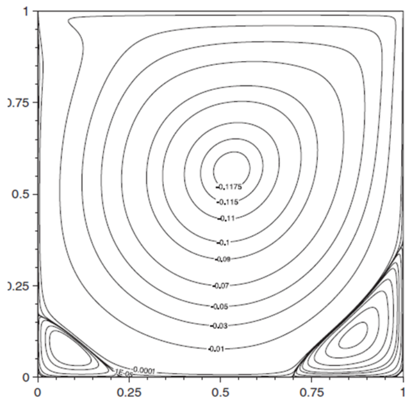


Figure 16: Ertirk Streamline Re=1000; Grid 129x129

1000 REYNOLDS COMPARISON

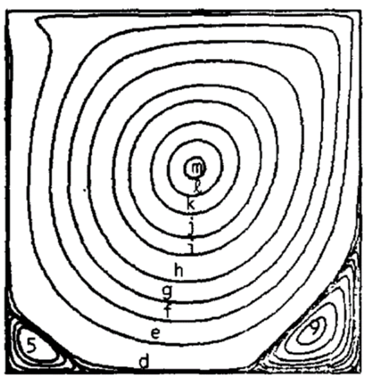


Figure 14: Ghia Streamline Re=1000; Grid 129x129

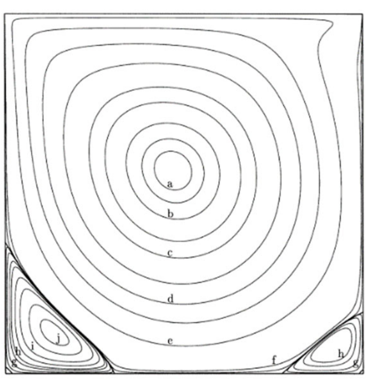


Figure 15: Botella Streamline Re=1000; Grid 128x128

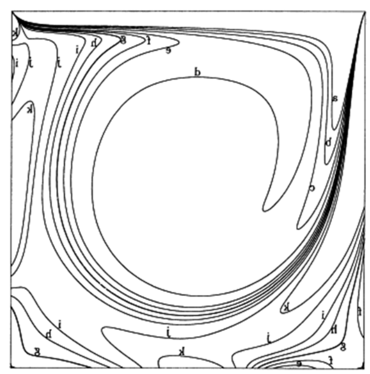


Figure 17: Botella Vorcitity Re=1000; Grid 128x128

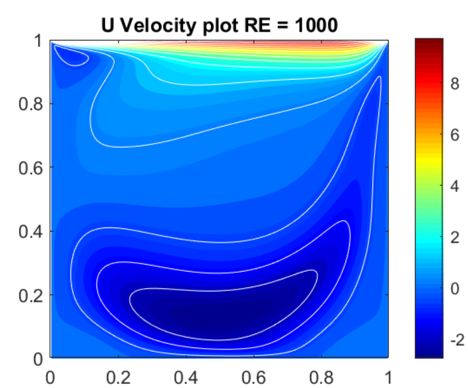


Figure 18: Current Work Velocity Re=1000; Grid 185x185

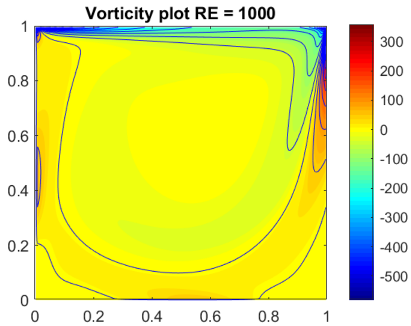


Figure 19: Current Work Vorticity Re=1000; Grid 185x185

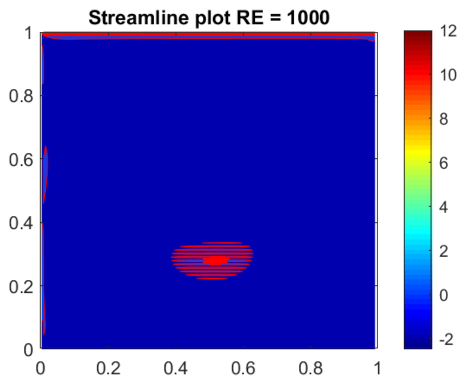


Figure 20: Current Work Streamline Re=1000; Grid 185x185

The most challenging aspect of this project was for the Re=1000, computational issues were encountered. Any grid size for the original relation parameters would diverge within minutes for any grid size below 215x215. However, for any grid larger than 235, the current computer being used for the project (specifications found in Annex A at the top.) would run out of the 32 GB of memory. However, after changing setting alpha_a=0.017 and alpha_p=0.81 the above results were achieved after taking the MATLAB script 41849.989 seconds (roughly 11.6 hours).

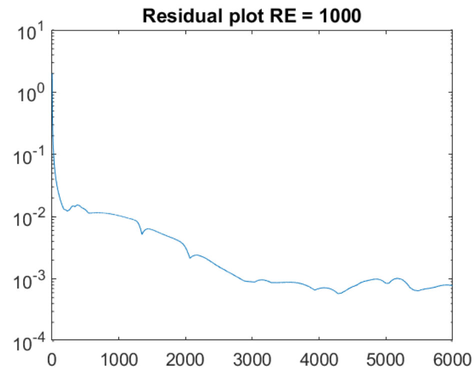


Figure 21: Current work Residual Re=1000

However, after inspecting the 6000 iterations the script performed, further parameter tuning is likely needed to achieve result seen in Figures 14-17.

REFERENCES

O. Botella and R. Peyret, \Benchmark spectral results on the lid-driven cavity flow," Computers and Fluids, vol. 27, pp. 421-433, 1998.

E. Erturk, T. C. Corke, and C. Gorkcol, \Numerical solutions of 2-D steady in-compressible driven cavity flow at high Reynolds numbers," International Journal for Numerical Methods in Fluids, vol. 48, pp. 747-774, 2005.

U. Ghia, K. N. Ghia, and C. T. Shin, \High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method," Journal of Computational Physics, vol. 48,pp. 387{411, 1982.

S.Mayoral, Laminar Flow code implemented in MATLAB using the SIMPLE algorithm, EGME 520: Advanced Viscous Fluid

Appendix A: Nonlinear Grid Attempt

One item that was analyzed while experimenting with getting the results to converge was the attempt to put a squared term on the NX and NY parameters in the script to achieve a finer grid resolution near the boundary conditions to help the model to converge. However misguided, the results were intriguing (found below)

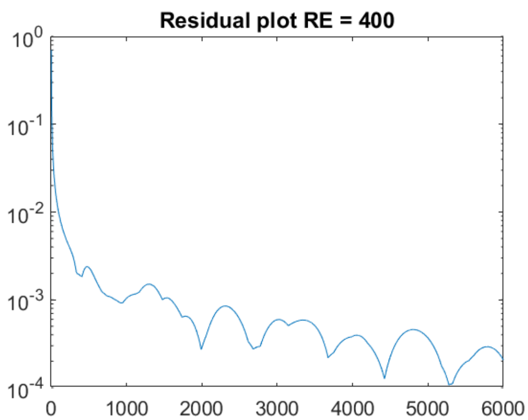
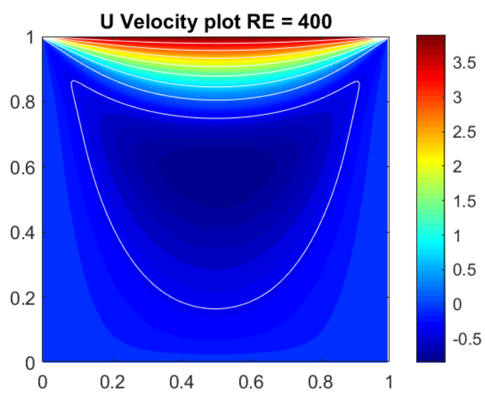
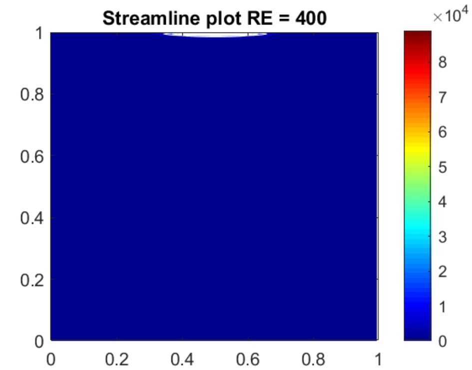
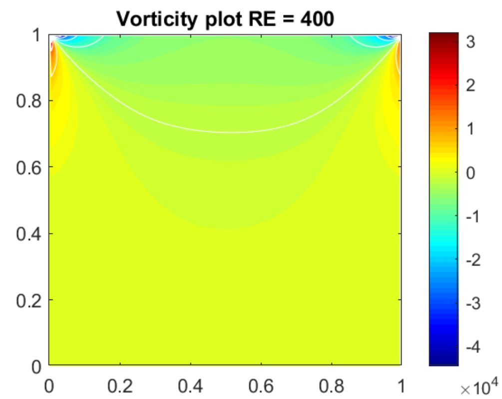


Figure 22: Nonlinear Attempt Re=400; Grid 180x180



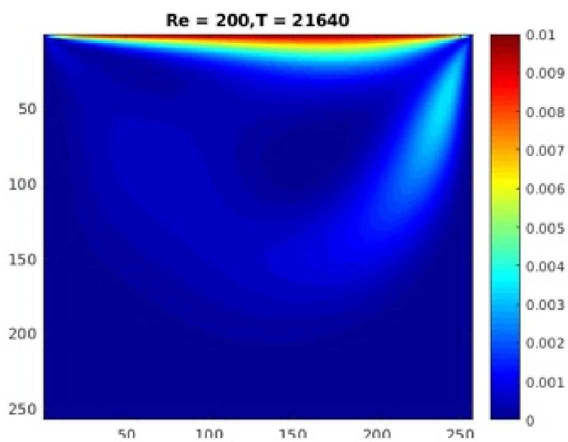


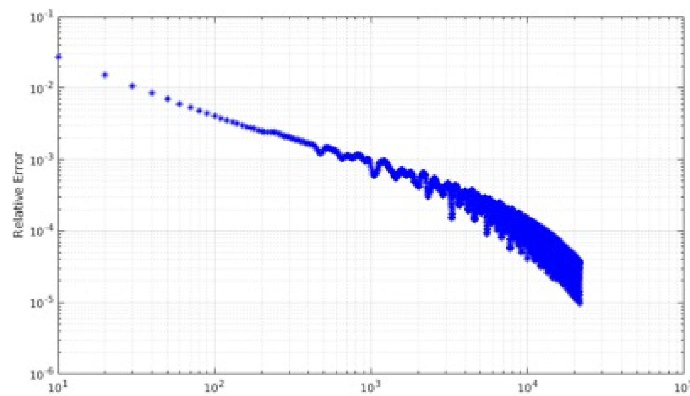


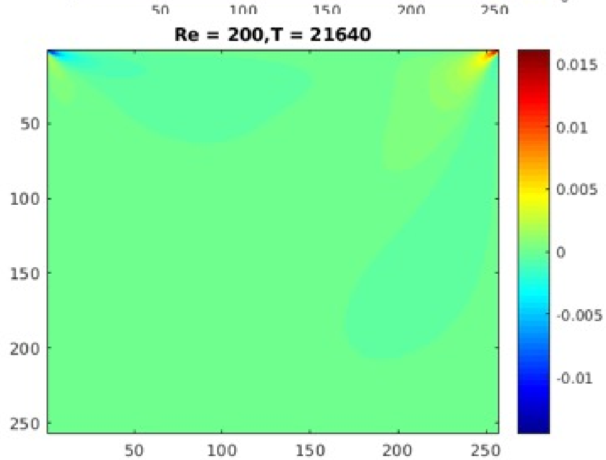
Appendix B: Lattice Boltzmann Method

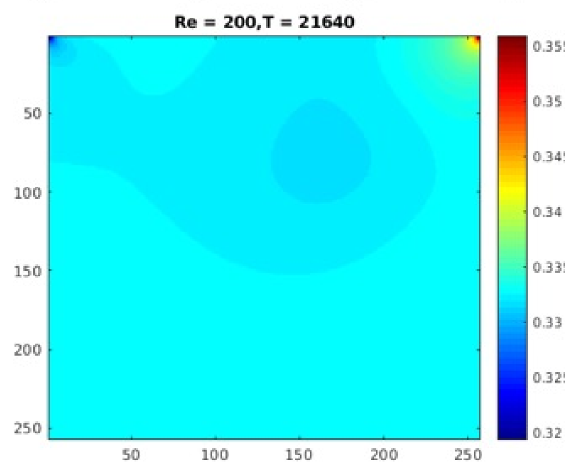
Another attempt to find an alternate solution (although assuredly would work given the time and effort) was the use of a lattice Boltzmann method of solving this problem rather than the SIMPLE algorithm above. The Lattice Boltzmann Method (LBM) uses statistics from particle collisions versus a nodal flow of information through a grid. Below are the result and where the code was found on the MATLAB code exchange website.

Note: This code did not take long to run, but was originally created in 2005 and the author mentions it is not optimized in any way.









Website:
https://github.com/yangyang14641/CavityFlowInLBMInMATLAB

Author: Yang Yang, Ph.D. Candidate in Fluid Mechanics at Peking University. Interested in High-Performance Computing, Computational Geometry, and Advanced Mathematical Physics.

ANNEX A

Below is the MATLAB code and associated function files in order to compute the Reynolds number of 100, 400, and 1000. Relaxation parameters were adjusted according to the best convergence results. (see above)

Laminar Flow Code.m
Run on:

Intel i7-7700K @ 4.2GHz
32.0GB RAM
64-bit Windows 10 OS
Nvidia GTX 1080 8GB GPU

```
%   ------------------------------------------------------------------------
---
%   EGME520 - LAMINAR FLOW CODE
%


%   Case :
%    Lid driven cavity flow
%
%   Descritption :
%    A MATLAB CFD script used to simulate steady 2D incompressible laminar
flow
%    using the SIMPLE algorithm. The code is based on the script by MJ Sarfi.
%
%   Reference :
%     2D Lid Driven Cavity Flow using SIMPLE algorithm by MJ Sarfi
%     https://www.mathworks.com/matlabcentral/fileexchange/68348-2d-lid-
driven-cavity-flow-using-simple-algorithm
%
%   Required m-files (functions):
%    pentaDiag_solve.m - Written by Greg von Winckel 3/15/04
%
%   Code written by :  S.Mayoral
%   Last update     :  29-NOV-2018 by S.Mayoral
%   Used by         :  20-DEC-2018 by A.Bartels

%   ------------------------------------------------------------------------
---
%
%   INITIALIZE CODE
%   ------------------------------------------------------------------------
---
%   Clear memory
tic
clear all
close all
format longG
%
%   Declare global variables
global NX NY NI
global rho mu U dX dY divU
global tolerance residual alpha_u alpha_p
global u u_s u_p u_c
global v v_s v_p v_c
global p p_s p_c p_rhs A_p
%   DECLARE PARAMETERS
%   Flow parameters
rho = 1.0;                              %  density (kg/m3)
mu  = 0.01;                             %  dynamic viscosity (N-s/m2)
U   = 10.0;                             %  lid velocity (m/s)
```

```
%
%   Grid parameters
NX = 180;                                 %  No. grid points in the x-direction
NY = 180;                                 %  No. grid points in the y-direction
%
%   Solver parameters
tolerance = 1e-5;                         %  convergence criteria based on
residual
NI         = 6000;                        %  maximumnumber of iterations
alpha_p    = 0.015;                           %  pressure under-relaxation
alpha_u    = 0.8;                     %  velocity under-relaxation
%
%   Declare arrays for pressure
p       = zeros(NX,NY);                   %  pressure (N/m2)
p_s     = zeros(NX,NY);                   %  staggered pressure (N/m2)
p_c = zeros(NX,NY);                       %  pressure correction (N/m2)
p_rhs   = zeros((NX-2)*(NY-2),1);         %  r.h.s. of pressure correction edn
A_p     = zeros(NX*NY,NX*NY);             %  pressure coefficient matrix
divU    = zeros(NX,NY);                   %  continuity check
%
%   Declare arrays for the y-velocity
u       = zeros(NX+1,NY);                 %  x-velocity (m/s)
u_s     = zeros(NX+1,NY);                 %  staggered x-velocity (m/s)
u_c     = zeros(NX+1,NY);                 %  x-velocity couresstion coefficient
(m/s)
u_p     = zeros(NX+1,NY);                 %  previous x-velocity (m/s)
%
%   Declare arrays for the x-velocity
v       = zeros(NX,NY+1);                 %  y-velocity (m/s)
v_s     = zeros(NX,NY+1);                 %  staggered y-velocity (m/s)
v_c     = zeros(NX,NY+1);                 %  y-velocity couresstion coefficient
(m/s)
v_p     = zeros(NX,NY+1);                 %  previous y-velocity (m/s)
%
%   Declare arrays for post processing steps
psi = zeros(NX,NY);
w = zeros(NX,NY);

%   ------------------------------------------------------------------------
---
%
%   DEFINE DOMAIN & LID BOUNDARY CONDITION
%   ------------------------------------------------------------------------
--
%   Define domain
L  = 1.0;                                 %  edge of square cavity (m)
dX = 1/(NX-1);                            %  cell size along x-direction
dY = 1/(NY-1);                            %  cell size along y-direction
X  = dX/2:dX:L-dX/2;                      %  x-position
Y  = 0:dY:L;                              %  y-position
%
%   Top wall is moving with speed U
u(:,end)      = U;                        %  NOTE: the rest of the domain is
initialized to zero
u_s(:,end)    = U;
%
%   Flow Reynolds number
```

```
Re = rho*U*L/mu;
%   -------------------------------------------------------------------------
---
%
%   SIMPLE ITERATION LOOP
%   -------------------------------------------------------------------------
--
%   Initialize parameters
n           = 1;                    %  initialize iteration counter
residual    = tolerance+1;          %  initial residual, must be greater
than tolerance
convergence = 0;                    %  solution convergance is assumed

while ( (residual > tolerance) && (n<=NI) )
    n=n+1;                          %  increase iteration count
    uMomentum;
    vMomentum;
    buildPressureRHS;
    buildPressureMatrix;
    pressureCorrection;
    updateVelocity;
    checkContinuity;
    residuals;

    disp(['It = ',int2str(n),'; Res = ',num2str(residual)])
    It(1,n) = n;
    Res(1,n) = residual;
    if (residual > 2)           %  check for convergence
        disp('Solution DID NOT converged!');
        break;
    end
end
toc
%   -------------------------------------------------------------------------
---
%
%   POST PROCESSING
%   -------------------------------------------------------------------------
--
%
%   Compute and generate streamlines contours, velocity and vorticity plots.
%   EGME520 - Final prject
%%
%   Calculating Vorticity
for i=2 : NX-1
    for j = 2:NY-1
        uy(i,j) = ( u(i,j) - u(i,j-1))/( Y(j) - Y(j-1));
        vx(i,j) = ( v(i,j) - v(i-1,j))/(X(i) - X(i-1));
        w       = vx - uy;
    end
end

%   Calculating the streamlines
for i = 2:NX-1
    for j = 2:NY-1
        psi(i,j) = u(i,j)*( Y(j+1) - Y(j) + psi(i,j-1));
```

```
    end
end
%%
% U VELOCITY CONTOUR PLOTS
FigHandle_01 = figure('Position', [100, 150, 390, 290]);
contourf(X,Y,u(2:NX,:)',50, 'edgecolor','none');
colormap jet
colorbar;
hold on
contour(X,Y,u(2:NX,:)',8,'edgecolor','w');
axis([0 1 0 1]);
title(sprintf('U Velocity plot RE = %d',Re))
saveas(FigHandle_01,sprintf('U Velocity plot RE_%d',Re),'png');

%% STREAMLINE CONTOUR PLOT
FigHandle_02 = figure('Position', [100, 150, 390, 290]);
contourf(X,Y,psi(2:NX,:)',5, 'edgecolor','w');
colormap jet
colorbar;
hold on
% contour(X,Y,psi(2:NX,:)',8,'edgecolor','w');
axis([0 1 0 1]);
title(sprintf('Streamline plot RE = %d',Re))
saveas(FigHandle_02,sprintf('Streamline plot RE_%d',Re),'png');

%% RESIDUAL PLOT
FigHandle_03 = figure('Position', [100, 150, 390, 290]);
semilogy(It,Res)
%axis([0 1 0 1]);
title(sprintf('Residual plot RE = %d',Re))
saveas(FigHandle_03,sprintf('Residual plot RE_%d',Re),'png');


%% VORTICITY CONTOUR PLOT
FigHandle_04 = figure('Position', [100, 150, 390, 290]);
contourf(X,Y(:,1:NX-1),w(1:NX-1,:)',100, 'edgecolor','none');
colormap jet
colorbar;
hold on
contour(X,Y(:,1:NX-1),w(1:NX-1,:)',8,'edgecolor','w');
axis([0 1 0 1]);
title(sprintf('Vorticity plot RE = %d',Re))
saveas(FigHandle_04,sprintf('Vorticity plot RE_%d',Re),'png');




%   -----------------------------------------------------------------------
---
%
%   END OF LAMINAR FLOW CODE
%   -----------------------------------------------------------------------
---
%   Below is a set of functions that are called by the MAIN Laminar Flow Code
```

```
%   Do not modify them! That changes that you will need to make are with in
the
%   post processing section.
%
%   ------------------------------------------------------------------------
---
%
%   FUNCTIONS - Do not modify these!
%   ------------------------------------------------------------------------
---
%   Solve u-momentum equation for intermediate x-velocity u_s
function uMomentum
global NX dX u u_s u_c u_p
global NY dY v p_s
global rho mu U alpha_u
%
%       convective coefficients
De = mu*dY/dX;
Dw = mu*dY/dX;
Dn = mu*dX/dY;
Ds = mu*dX/dY;
%
A = @(F,D)( max(0, (1-0.1 * abs(F/D))^5 ) );
%
%       Compute u_s
for i = 2:NX
    for j = 2:NY-1
        Fe  = 0.5*rho*dY*(u(i+1,j)+u(i,j));
        Fw  = 0.5*rho*dY*(u(i-1,j)+u(i,j));
        Fn  = 0.5*rho*dX*(v(i,j+1)+v(i-1,j+1));
        Fs  = 0.5*rho*dX*(v(i,j)+v(i-1,j));
        %
        aE = De * A(Fe,De) + max(-Fe,0);
        aW = Dw * A(Fw,Dw) + max(Fw,0);
        aN = Dn * A(Fn,Dn) + max(-Fn,0);
        aS = Ds * A(Fs,Ds) + max(Fs,0);
        aP = aE + aW + aN + aS + (Fe-Fw) + (Fn-Fs);
        %
        p_term   = (p_s(i-1,j)-p_s(i,j)) * dY;
        u_s(i,j) = alpha_u/aP * ( (aE*u(i+1,j)+aW*u(i-1,j)+aN*u(i,j+1)...
            +aS*u(i,j-1)) + p_term ) + (1-alpha_u)*u(i,j);
        u_c(i,j) = alpha_u*dY/aP;
    end
end
%
%       Set u_c for left and right boundary conditions, they will be later
used
%       by the pressure correction equation they should not be zero, or BCs
of
%       pressure correction will get messed up
%
%       Apply bottom boundary condition
j = 1;
for i=2:NX
    Fe  = 0.5*rho*dY*(u(i+1,j)+u(i,j));
    Fw  = 0.5*rho*dY*(u(i-1,j)+u(i,j));
    Fn  = 0.5*rho*dX*(v(i,j+1)+v(i-1,j+1));
```

```
    Fs  = 0;
    %
    aE = De * A(Fe,De) + max(-Fe,0);
    aW = Dw * A(Fw,Dw) + max(Fw,0);
    aN = Dn * A(Fn,Dn) + max(-Fn,0);
    aS = 0;
    aP = aE + aW + aN + aS + (Fe-Fw) + (Fn-Fs);
    %
    u_c(i,j) = alpha_u*dY/aP;
end
%
%       Apply top boundary condition
j = NY;
for i=2:NX
    Fe  = 0.5*rho*dY*(u(i+1,j)+u(i,j));
    Fw  = 0.5*rho*dY*(u(i-1,j)+u(i,j));
    Fn  = 0;
    Fs  = 0.5*rho*dX*(v(i,j)+v(i-1,j));
    %
    aE = De * A(Fe,De) + max(-Fe,0);
    aW = Dw * A(Fw,Dw) + max(Fw,0);
    aN = 0;
    aS = Ds * A(Fs,Ds) + max(Fs,0);
    aP = aE + aW + aN + aS + (Fe-Fw) + (Fn-Fs);
    %
    u_c(i,j) = alpha_u*dY/aP;
end
%
%       Apply boundary conditions
u_s(1,:)   =-u_s(2,:);                      %  left wall
u_s(end,:) =-u_s(NX,:);                     %  right wall
u_s(:,1)   = 0;                             %  bottom wall
u_s(:,end) = U;                             %  top wall

u_p = u;                                    %  store previous solution
end
%
%   -------------------------------------------------------------------------
---
%   Solve v-momentum equation for intermediate y-velocity v_s
function vMomentum
global NX dX u p_s
global NY dY v v_s v_c v_p
global rho mu alpha_u
%
%       convective coefficients
De = mu*dY/dX;
Dw = mu*dY/dX;
Dn = mu*dX/dY;
Ds = mu*dX/dY;
%
A = @(F,D)( max(0, (1-0.1 * abs(F/D))^5 ) );
%
%       Compute u_s
for i = 2:NX-1
    for j = 2:NY
        Fe  = 0.5*rho*dY*(u(i+1,j)+u(i+1,j-1));
```

```
        Fw  = 0.5*rho*dY*(u(i,j)+u(i,j-1));
        Fn  = 0.5*rho*dX*(v(i,j)+v(i,j+1));
        Fs  = 0.5*rho*dX*(v(i,j-1)+v(i,j));
        %
        aE = De * A(Fe,De) + max(-Fe,0);
        aW = Dw * A(Fw,Dw) + max(Fw,0);
        aN = Dn * A(Fn,Dn) + max(-Fn,0);
        aS = Ds * A(Fs,Ds) + max(Fs,0);
        aP = aE + aW + aN + aS + (Fe-Fw) + (Fn-Fs);
        %
        p_term   = (p_s(i,j-1)-p_s(i,j)) * dX;
        v_s(i,j) = alpha_u/aP * ( (aE*v(i+1,j)+aW*v(i-1,j)+aN*v(i,j+1) ...
            +aS*v(i,j-1)) + p_term ) + (1-alpha_u)*v(i,j);
        v_c(i,j) = alpha_u*dX/aP;
    end
end
%
%       Set v_c for left and right boundary conditions, they will be later
used
%       by the pressure correction equation they should not be zero, or BCs
of
%       pressure correction will get messed up
%
%       Apply left boundary condition
i = 1;
for j=2:NY
    Fe  = 0.5*rho*dY*(u(i+1,j)+u(i+1,j-1));
    Fw  = 0;
    Fn  = 0.5*rho*dX*(v(i,j)+v(i,j+1));
    Fs  = 0.5*rho*dX*(v(i,j-1)+v(i,j));
    %
    aE = De * A(Fe,De) + max(-Fe,0);
    aW = 0;
    aN = Dn * A(Fn,Dn) + max(-Fn,0);
    aS = Ds * A(Fs,Ds) + max(Fs,0);
    aP = aE + aW + aN + aS + (Fe-Fw) + (Fn-Fs);
    %
    v_c(i,j) = alpha_u*dX/aP;
end
%
%       Apply right boundary condition
i = NX;
for j=2:NY
    Fe  = 0;
    Fw  = 0.5*rho*dY*(u(i,j)+u(i,j-1));
    Fn  = 0.5*rho*dX*(v(i,j)+v(i,j+1));
    Fs  = 0.5*rho*dX*(v(i,j-1)+v(i,j));
    %
    aE = 0;
    aW = Dw * A(Fw,Dw) + max(Fw,0);
    aN = Dn * A(Fn,Dn) + max(-Fn,0);
    aS = Ds * A(Fs,Ds) + max(Fs,0);
    aP = aE + aW + aN + aS + (Fe-Fw) + (Fn-Fs);
    %
    v_c(i,j) = alpha_u*dX/aP;
end
%
```

```
%       Apply boundary conditions
v_s(1,:)   = 0.0;                         %  left wall
v_s(end,:) = 0.0;                         %  right wall
v_s(:,1)   =-v_s(:, 2);                   %  bottom wall
v_s(:,end) =-v_s(:,NY);                   %  top wall

v_p = v;                                  %  store previous solution
end
%
%   -------------------------------------------------------------------------
---
%   Compute RHS vector of the Pressure Poisson matrix
function buildPressureRHS
global NX dX u_s
global NY dY v_s
global p_rhs rho
%
%       RHS is the same for all nodes except the p_c(1,1) sincs p(1,1) is
zero
for i=1:NX
    for j=1:NY
        k = i + (j-1)*NY;
        p_rhs(k)  = rho*(u_s(i,j)*dY - u_s(i+1,j)*dY + v_s(i,j)*dX -
v_s(i,j+1)*dX);
    end
end
p_rhs(1)=0;
end
%
%   -------------------------------------------------------------------------
---
%   Build the Pressure Poisson coefficient matrix
function buildPressureMatrix
global NX dX u_c
global NY dY v_c
global A_p rho
%
%       p_c for boundary nodes is set to zero and interior nodes imax-2*jmax-
2 solved implicitly for p_c
for j=1:NY
    for i=1:NX
        k = i + (j-1)*NY;

        aE = 0;
        aW = 0;
        aN = 0;
        aS = 0;

        %               Set boundary conditions for four corners
        if (i == 1 && j == 1)
            A_p(k,k)  = 1;                %  pressure correction at the first
node is zero
            continue;
        end

        if (i == NX && j == 1)
```

```
    A_p(k,k-1)  =-rho*u_c(i,j)*dY;
    aW          =-A_p(k,k-1);
    A_p(k,k+NY) =-rho*v_c(i,j+1)*dX;
    aN          =-A_p(k,k+NY);
    aP          = aE + aN + aW + aS;
    A_p(k,k)    = aP;
    continue;
end

if (i == 1 && j == NY)
    A_p(k,k+1)  =-rho*u_c(i+1,j)*dY;
    aE          =-A_p(k,k+1);
    A_p(k,k-NY) =-rho*v_c(i,j)*dX;
    aS          =-A_p(k,k-NY);
    aP          = aE + aN + aW + aS;
    A_p(k,k)    = aP;
    continue;
end

if (i == NX && j == NY)
    A_p(k,k-1)  =-rho*u_c(i,j)*dY;
    aW          =-A_p(k,k-1);
    A_p(k,k-NY) =-rho*v_c(i,j)*dX;
    aS          =-A_p(k,k-NY);
    aP          = aE + aN + aW + aS;
    A_p(k,k)    = aP;
    continue;
end
%
%               Set  boundary conditions for four boundaries
if (i == 1)
    A_p(k,k+1)  =-rho*u_c(i+1,j)*dY;
    aE          =-A_p(k,k+1);
    A_p(k,k+NY)  =-rho*v_c(i,j+1)*dX;
    aN          =-A_p(k,k+NY);
    A_p(k,k-NY) =-rho*v_c(i,j)*dX;
    aS          =-A_p(k,k-NY);
    aP          = aE + aN + aW + aS;
    A_p(k,k)    = aP;
    continue;
end

if (j == 1)
    A_p(k,k+1)  =-rho*u_c(i+1,j)*dY;
    aE          =-A_p(k,k+1);
    A_p(k,k+NY) =-rho*v_c(i,j+1)*dX;
    aN          =-A_p(k,k+NY);
    A_p(k,k-1)  =-rho*u_c(i,j)*dY;
    aW          =-A_p(k,k-1);
    aP          = aE + aN + aW + aS;
    A_p(k,k)    = aP;
    continue;
end

if (i == NX)
    A_p(k,k+NY) =-rho*v_c(i,j+1)*dX;
```

```
            aN           =-A_p(k,k+NY);
            A_p(k,k-NY) =-rho*v_c(i,j)*dX;
            aS           =-A_p(k,k-NY);
            A_p(k,k-1)  =-rho*u_c(i,j)*dY;
            aW           =-A_p(k,k-1);
            aP           = aE + aN + aW + aS;
            A_p(k,k)    = aP;
            continue;
        end

        if (j == NY)
            A_p(k,k+1)  =-rho*u_c(i+1,j)*dY;
            aE           =-A_p(k,k+1);
            A_p(k,k-NY) =-rho*v_c(i,j)*dX;
            aS           =-A_p(k,k-NY);
            A_p(k,k-1)  =-rho*u_c(i,j)*dY;
            aW           =-A_p(k,k-1);
            aP           = aE + aN + aW + aS;
            A_p(k,k)    = aP;
            continue;
        end

        %                Interior nodes
        A_p(k,k-1)  =-rho*u_c(i,j)*dY;          %  sub diagonal
        aW           =-A_p(k,k-1);
        A_p(k,k+1)  =-rho*u_c(i+1,j)*dY;        %  upper diagonal
        aE           =-A_p(k,k+1);
        A_p(k,k-NY) =-rho*v_c(i,j)*dX;          %  sub sub diagonal
        aS           =-A_p(k,k-NY);
        A_p(k,k+NY) =-rho*v_c(i,j+1)*dX;        %  upper upper diagonal
        aN           =-A_p(k,k+NY);
        aP           = aE + aN + aW + aS;
        A_p(k,k)    = aP;
    end
end
end
%
%   -----------------------------------------------------------------------
---
%   Solve pressure correction implicitly and update pressure
function pressureCorrection
global NX NY
global p p_c p_s p_rhs A_p alpha_p
%
%       Solve a pentadiagonal system with pentaDiag_solve.m (must be in the
same directory)
p_ci = pentaDiag_solve(A_p,p_rhs);
%
%       Convert pressure correction in to a matrix
k=0;
for j=1:NY
    for i=1:NX
        k=k+1;
        p_c(i,j)=p_ci(k);
        p(i,j) = p_s(i,j) + alpha_p*p_c(i,j);   %  update pressure values
    end
```

```
end
%
p(1,1)=0;
p_s = p;
end
%
%   ------------------------------------------------------------------------
---
%   Update velocity based on pressure correction
function updateVelocity
global p_c U
global u u_s u_c NX
global v v_s v_c NY
%
%       Update interior u-nodes
for i=2:NX
    for j=2:NY-1
        u(i,j) = u_s(i,j) + u_c(i,j)*(p_c(i-1,j)-p_c(i,j));
    end
end
%
%       Update interior u-nodes
for i=2:NX-1
    for j=2:NY
        v(i,j) = v_s(i,j) + v_c(i,j)*(p_c(i,j-1)-p_c(i,j));
    end
end
%
%       Update boundary conditions, x-velocity
u(1,:)   =-u(2,:);                        %  left wall
u(end,:) =-u(NX,:);                       %  right wall
u(:,1)   = 0.0;                           %  bottom wall
u(:,end) = U;                             %  top wall
%
%       Update boundary conditions, y-velocity
v(1,:)   = 0.0;                           %  left wall
v(end,:) = 0.0;                           %  right wall
v(:,1)   =-v(:,2);                        %  bottom wall
v(:,end) =-v(:,NY);                       %  top wall
end
%   ------------------------------------------------------------------------
---
%   Check if velocity field satisfies the continuity equation
function checkContinuity
global divU
global NX dX u
global NY dY v
%
for i=1:NX
    for j=1:NY
        divU(i,j) = (u(i,j)-u(i+1,j))/dX + (v(i,j)-v(i,j+1))/dY;
    end
end
end
%
%   ------------------------------------------------------------------------
---
```

```
%   Determine the maximum residual
function residuals
global residual
global u u_p
global v v_p
%
uRes = abs(u - u_p);                    %  local x-velocity residual
vRes = abs(v - v_p);                    %  local y-velocity residual
%
uRes_max = max(max(uRes));              %  maximum x-velocity residual
vRes_max = max(max(vRes));              %  maximum y-velocity residual
%
residual = max(uRes_max, vRes_max);     %  maximum residual
end


%   -------------------------------------------------------------------------
---
%
%   END OF FILE
%   -------------------------------------------------------------------------
---
```

pentaDiag_solve.m

```
function x=pentaDiag_solve(A,b)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%
% pentsolve.m
%
% Solve a pentadiagonal system Ax=b where A is a strongly nonsingular matrix
%
% If A is not a pentadiagonal matrix, results will be wrong
%
% Reference: G. Engeln-Muellges, F. Uhlig, "Numerical Algorithms with C"
%                Chapter 4. Springer-Verlag Berlin (1996)
%
% Written by Greg von Winckel 3/15/04
% Contact: gregvw@chtm.unm.edu
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%

[M,N]=size(A);

% Check dimensions
if M~=N
    error('Matrix must be square');
    return;
end

if length(b)~=M
    error('Matrix and vector must have the same number of rows');
    return;
end

x=zeros(N,1);

% Check for symmetry
if A==A'    % Symmetric Matrix Scheme

    % Extract bands
    d=diag(A);
    f=diag(A,1);
    e=diag(A,2);

    alpha=zeros(N,1);
    gamma=zeros(N-1,1);
    delta=zeros(N-2,1);
    c=zeros(N,1);
    z=zeros(N,1);

    % Factor A=LDL'
    alpha(1)=d(1);
    gamma(1)=f(1)/alpha(1);
    delta(1)=e(1)/alpha(1);
```

```
    alpha(2)=d(2)-f(1)*gamma(1);
    gamma(2)=(f(2)-e(1)*gamma(1))/alpha(2);
    delta(2)=e(2)/alpha(2);

    for k=3:N-2
        alpha(k)=d(k)-e(k-2)*delta(k-2)-alpha(k-1)*gamma(k-1)^2;
        gamma(k)=(f(k)-e(k-1)*gamma(k-1))/alpha(k);
        delta(k)=e(k)/alpha(k);
    end

    alpha(N-1)=d(N-1)-e(N-3)*delta(N-3)-alpha(N-2)*gamma(N-2)^2;
    gamma(N-1)=(f(N-1)-e(N-2)*gamma(N-2))/alpha(N-1);
    alpha(N)=d(N)-e(N-2)*delta(N-2)-alpha(N-1)*gamma(N-1)^2;

    % Update Lx=b, Dc=z

    z(1)=b(1);
    z(2)=b(2)-gamma(1)*z(1);

    for k=3:N
        z(k)=b(k)-gamma(k-1)*z(k-1)-delta(k-2)*z(k-2);
    end

    c=z./alpha;

    % Backsubstitution L'x=c
    x(N)=c(N);
    x(N-1)=c(N-1)-gamma(N-1)*x(N);

    for k=N-2:-1:1
        x(k)=c(k)-gamma(k)*x(k+1)-delta(k)*x(k+2);
    end

else        % Non-symmetric Matrix Scheme

    % Extract bands
    d=diag(A);
    e=diag(A,1);
    f=diag(A,2);
    h=[0;diag(A,-1)];
    g=[0;0;diag(A,-2)];

    alpha=zeros(N,1);
    gam=zeros(N-1,1);
    delta=zeros(N-2,1);
    bet=zeros(N,1);

    c=zeros(N,1);
    z=zeros(N,1);

    % Factor A=LR
    alpha(1)=d(1);
    gam(1)=e(1)/alpha(1);
    delta(1)=f(1)/alpha(1);
```

```matlab
    bet(2)=h(2);
    alpha(2)=d(2)-bet(2)*gam(1);
    gam(2)=( e(2)-bet(2)*delta(1) )/alpha(2);
    delta(2)=f(2)/alpha(2);

    for k=3:N-2
        bet(k)=h(k)-g(k)*gam(k-2);
        alpha(k)=d(k)-g(k)*delta(k-2)-bet(k)*gam(k-1);
        gam(k)=( e(k)-bet(k)*delta(k-1) )/alpha(k);
        delta(k)=f(k)/alpha(k);
    end

    bet(N-1)=h(N-1)-g(N-1)*gam(N-3);
    alpha(N-1)=d(N-1)-g(N-1)*delta(N-3)-bet(N-1)*gam(N-2);
    gam(N-1)=( e(N-1)-bet(N-1)*delta(N-2) )/alpha(N-1);
    bet(N)=h(N)-g(N)*gam(N-2);
    alpha(N)=d(N)-g(N)*delta(N-2)-bet(N)*gam(N-1);

    % Update b=Lc
    c(1)=b(1)/alpha(1);
    c(2)=(b(2)-bet(2)*c(1))/alpha(2);

    for k=3:N
        c(k)=( b(k)-g(k)*c(k-2)-bet(k)*c(k-1) )/alpha(k);
    end


    % Back substitution Rx=c
    x(N)=c(N);
    x(N-1)=c(N-1)-gam(N-1)*x(N);

    for k=N-2:-1:1
        x(k)=c(k)-gam(k)*x(k+1)-delta(k)*x(k+2);
    end

end
```