M3N10: Final Project

John Paul Alexander CID:00824301

May 3, 2016

Abstract

Extending cavity flow from a rectangular domain to an L shaped domain.

This is my own unaided work unless otherwise specified.

Contents

1 Advection

The equation governing the motion of advection is the given below (1) where a_x and a_y are parameters controlling the speed and direction of the solution.

$$\frac{\partial q}{\partial t} + a_x \frac{\partial q}{\partial x} + a_y \frac{\partial q}{\partial y} = 0 \tag{1}$$

The code given to us can be easily extended by editing 'SemiLagrAdvect.m' to accept rectangles of different sizes and calling the function twice giving the boundary conditions of the edge where it is so and an average of rows or columns of the previous solution where the boundary falls inside the L-shape. Then the two new partial solutions are stitched together to form the solution at the next step.

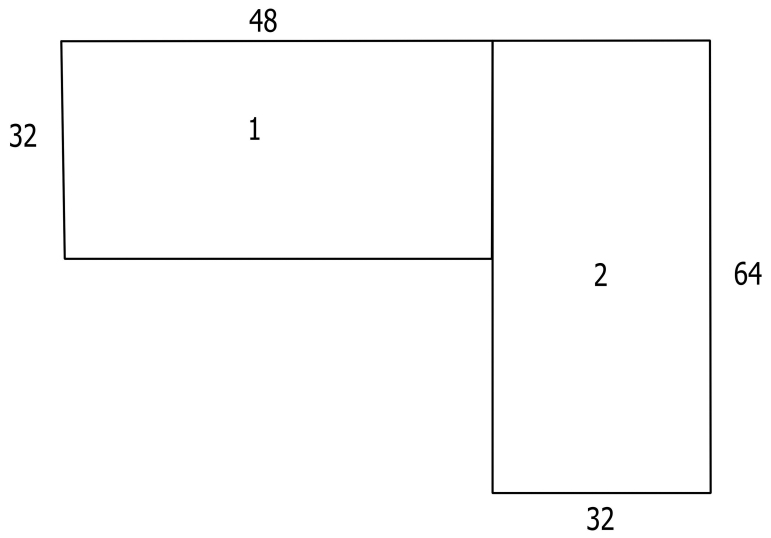


Figure 1: The Domain Decomposition

The decomposition of the L-shape is shown above and so for the first domain the code uses the given boundary conditions for the north, south and west boundaries and an average of the top halves of the 48th and 49th column on the east.

To verify the error and reflection free passing of an initial condition through the interface we compare the solution to one which is computed on a rectangle covering the whole L-shaped domain. Table (1) details the conditions and results of the test with an initial condition of zeros except from a rectangle of ones from $x = 16$ to 32, $y = 48$ to 52 run for 50 time steps of 0.00667.

Variable	Value
a_x	1
a_y	0.1
Error	4.2067e-16

Table 1: Results of test

The results show that the difference in the two solutions is less than the accuracy of Matlab. This shows that our algorithm is error- and reflection-free across the interface between the decomposed domains. The plots given below show the initial condition for the test and also the solution at the end of the test. The second plot shows the smooth transition between domains as there are no obvious jumps.

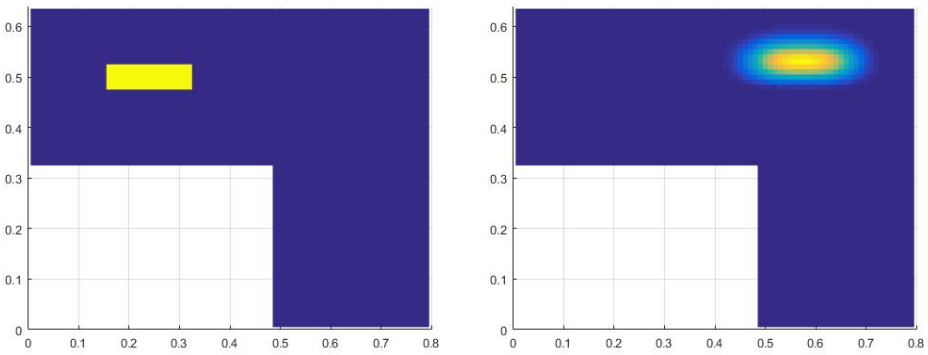


Figure 2: Initial Condition and Final Plot of the Test

2 Diffusion

The two-dimensional diffusion equation takes the form given in equation (2) and the question imposes inhomogeneous Dirichlet boundary conditions. D controls the rate at which the solution diffuses.

$$\frac{\partial q}{\partial t} = D\left(\frac{\partial^2 q}{\partial x^2} + \frac{\partial^2 q}{\partial y^2}\right) \tag{2}$$

The L-shape domain is decomposed into two overlapping rectangles in order for the information to be passed from one to the other. The decomposition is given below in figure (3). Using a new script, 'DiffusionSolve.m' is called successively on each domain updating the boundary conditions at the overlap region boundaries as specified by the multiplicative alternating Schwarz algorithm. The code uses a while loop to run the alternating Schwarz until difference in solution between the two domains is below a specified tolerance.

'DiffusionSolve.m' uses the multigrid method to solve the diffusion equation, which means A,R and P must be given. Fortunately both decomposed domains have the same dimensions and so as diffusion is orientation-less, transposing one of the domains means only one set of A,R and P need to be specified. A,R and P are given such that a Dirichlet condition is specified on the boundaries and A is adjusted to fit the given equation.

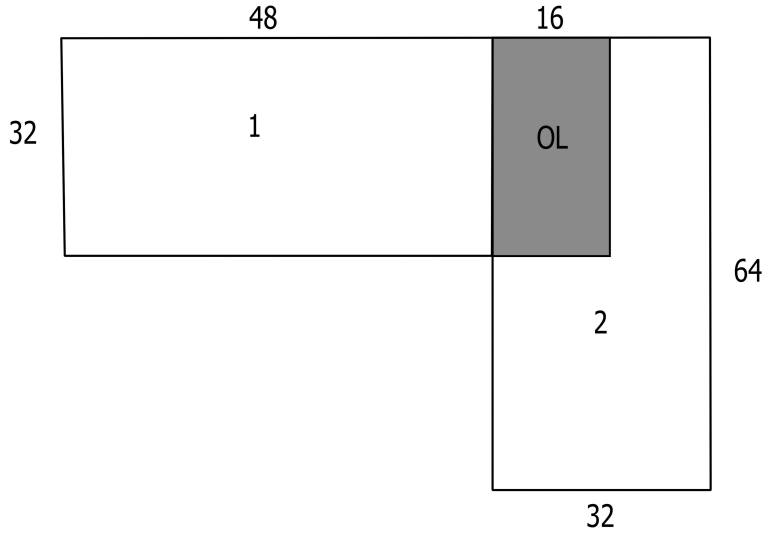


Figure 3: The Domain Decomposition for Diffusion

To verify the numerical solution, we monitor the residual on the interface having selected a D of 1/500 (which corresponds to a Reynolds number of 500) and final time of 1 second, meaning 150 time steps of size 0.01/1.5. The boundary conditions were given as one on the north boundary and

zero elsewhere. The plots created at the end of running the script can be found below in figure (4) and figure (5) is plot showing the decrease in the maximum difference between the solutions on the decomposed regions at each full iteration of the alternating Schwarz for the final time step.

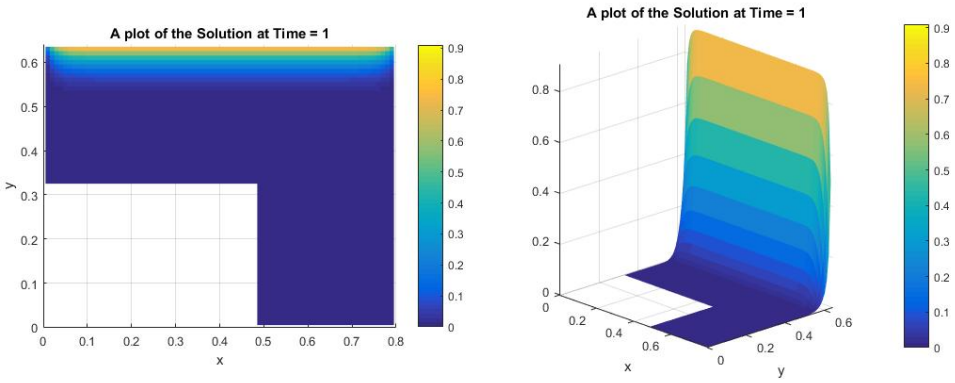


Figure 4: The Solution at the end of the Test

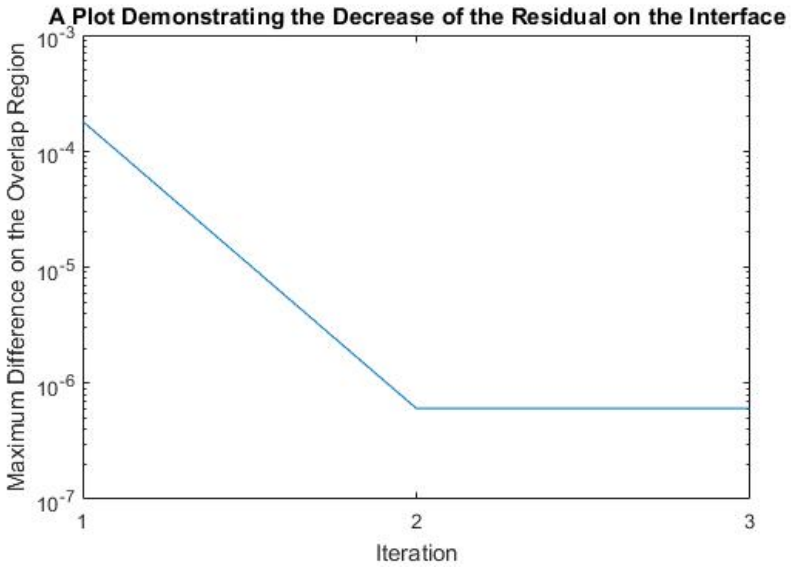


Figure 5:

The plot above shows that after only two full iterations we reach the limit of accuracy that this method can achieve with machine error. Table (2) also shows this very clearly and the fast rate which is achieved.

Iteration	Error	Rate
1	0.000180977	-
2	6.04982e-07	299.145
3	6.04982e-07	1

Table 2: Results of test

This test demonstrates that my code is running as expected and the solution plots coincide with our intuition of diffusion. After this test I included a condition to stop the while loop when the rate of converges has decreased significantly, this means that time is not wasted running the alternating Schwarz when very little increase in accuracy can be achieved.

3 Poisson

The Poisson equation is an eliptic equation and unlike the first two, does not depend on time. For this section the boundary conditions around the L-shaped domain are homogeneous Neumann boundary conditions and the governing equation is the Poisson equation given below.

$$\frac{\partial^2 q}{\partial x^2} + \frac{\partial^2 q}{\partial x^2} = f(x, y) \tag{3}$$

Again the overlap region is used and the two decomposed domains are as in the diffusion section. However for this problem we have Neumann boundary conditions on the boundary of the L-shape and Dirichlet on the boundary of the overlap region. This means that A,P and R must be changed to account for the different types of boundary condition. For P, I splice together sections of a Dirichlet P and a Neumann P in order to get Dirichlet and Neumann on the same side of the rectangle and introduce a second set of A, P and R for the second rectangle. For A, I adjust the matrix after it has been created for Neumann BC. The Poisson equation with full Neumann boundary conditions is underdetermined and so to ensure the uniqueness of the solution we impose a single Dirichlet point on the boundary.

The multiplicative alternating Schwarz algorithm is implemented in exactly the same way as before, looping until the solutions coincide.

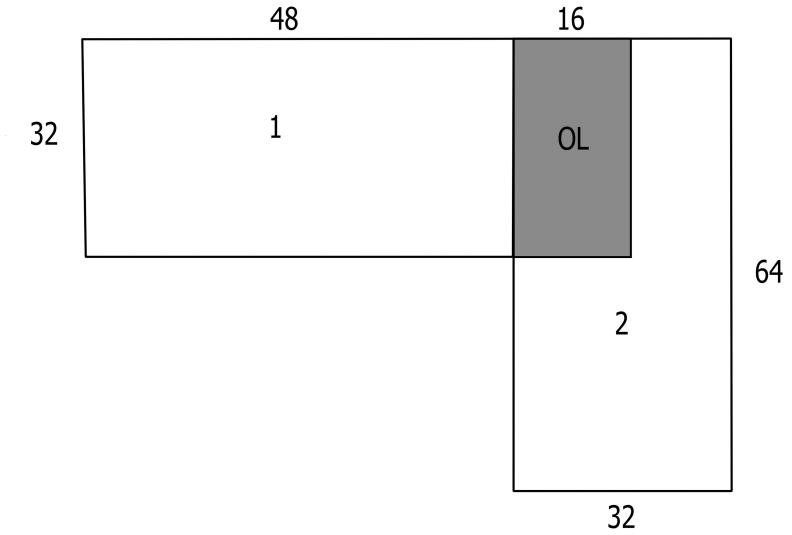


Figure 6: The Domain Decomposition for Poisson

Again we verify the solution by monitoring the residual on the interface of a test problem. For this test we impose homogeous Neumann boundary conditions round the L-shaped domain (except for the single Dirichlet point), a right hand side in the Poisson equation of 1 and initial guess of zeros. Plots of the numerical solution are given in figure (7), the spike is at the position of the imposed Dirichlet point.

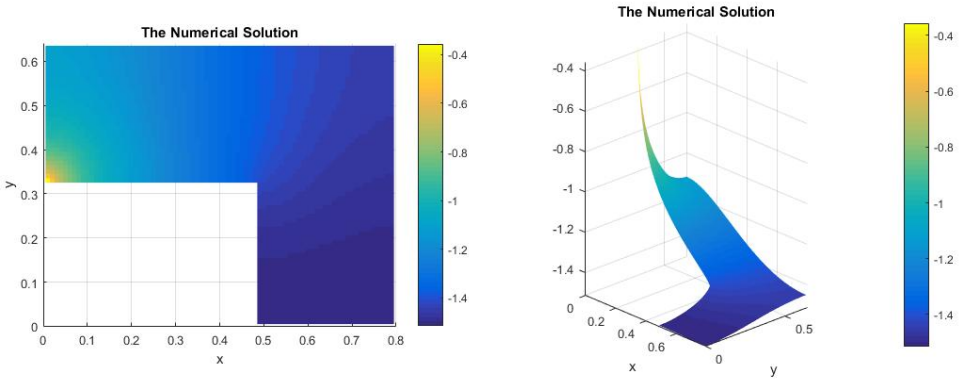


Figure 7: The solution at the end of the test 1

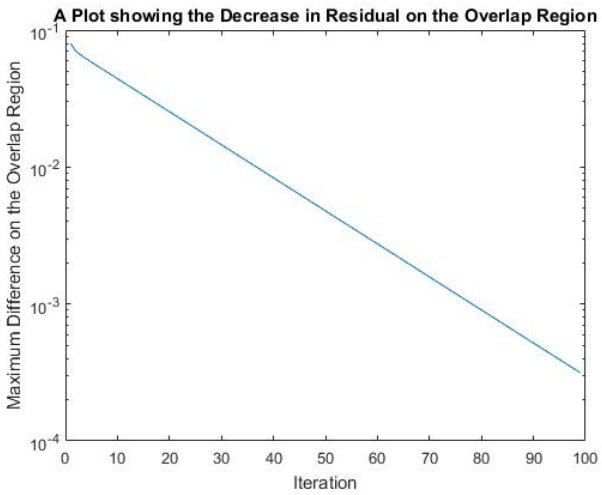


Figure 8: Test 1

Above is a plot showing the steady decrease in the maximum difference of the two solutions on the overlap region. For all the tests, including those not included here, the rate of reduction settled to a constant 1.05714 per iteration. This means that the initial guess of the solution is a big factor for number of iterations until the desired accuracy is achieved.

To show this, the test was run again with a new initial condition of -1.4 everywhere. The original test resulted in a final maximum difference of 0.000314344 compared to 9.06051e-06 for the second test, the ratio of which is 34.6939. The plot for the second test is given below in figure (9).

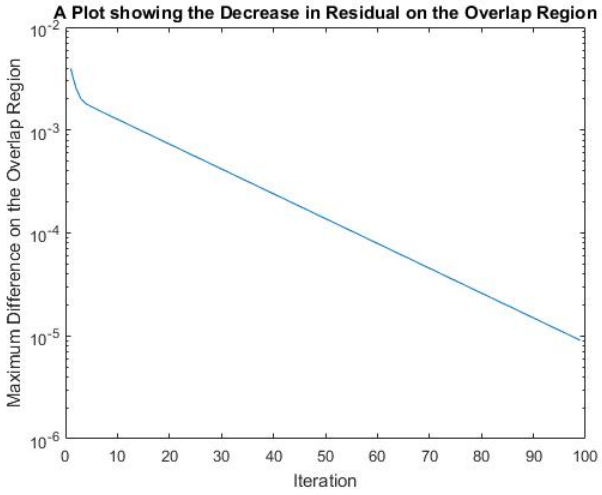


Figure 9: Test 2

4 Cavity Flow

In this final section, the previous sections are put together to solve the incompressible Navier-Stokes equations (4, 5) by operator splitting. The method begins with the advection part the result of which is used for the diffusion and finally the Poisson equation is used to correct the flow to take into account the pressure.

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\nabla\mathbf{u} = -\nabla p + \frac{1}{Re}\nabla^2\mathbf{u} \tag{4}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{5}$$

For this section most of the coding has been already completed in the previous sections, the script just needs to call the correct procedure for each part. However in order to solve for the pressure and to use the pressure to correct the flow the function 'Diff.m' is called and this must be modified for the L-shaped domain. The modifications are simple: updating to account for the new boundary shape and sweeping only inside the L-shape.

I used my previous error calculating scripts to gain an understanding the error we could expect from the different parts. Diffusion has a lower bound for the error caused by the alternating Schwarz, to obtain the most accurate answer possible and computationally efficient the tolerances for the other scripts should just as accurate but not more. Through testing the error was found to be of the order 1e-7 and so I set 1e-7 to be my desired tolerance for my functions. It is important to remember that the solution to the Poisson part is then used in the 'Diff.m', this effectively multiplies the possible error by a factor of 1/dx (i.e. 100) thus we should set the tolerance on the pressure to be 1e-9.

However the Poisson script is the most computational expensive and for this error tolerance would take a very long time to run. To evade this issue, I used the fact that we are looking for a steady state solution, so small perturbation from this solution should revert back to the steady solution. This means that for low time steps we can reduce the tolerance on the pressure to decrease the computation time. The best way to do this is to restrict the alternating Schwarz to less iterations as when the solution is close to the steady state the error from the alternating Schwarz will decrease as the initial guess and final solution are much closer together. This also specifies the maximum computation time used for that the Poisson script for each time step.

The reduction in computation time is significant, with a max number of alternating Schwarz cycles set to 300 (so that the tolerance was always met) the script took 1545.709 seconds to reach a time step of 150 which is one second, 1533.044 of which were spent in 'poissonSolve.m'. With a restriction to 5 alternating Schwarz cycles the time taken was 66.772, with

61.783 seconds in 'poissonSolve.m'. Further details as to the loss of accuracy can be found in the table below (3).

Max Number of Cycles	Time	Max Difference
50	593.689	-
5	66.772	6.4830e-06

Table 3: Results of restricted Alternating Schwarz test

From the brief error analysis, the error is approximately 1e-6. Now we can move on to studying the solution itself, starting with a plot of the velocity field (10) with Reynolds number 500.

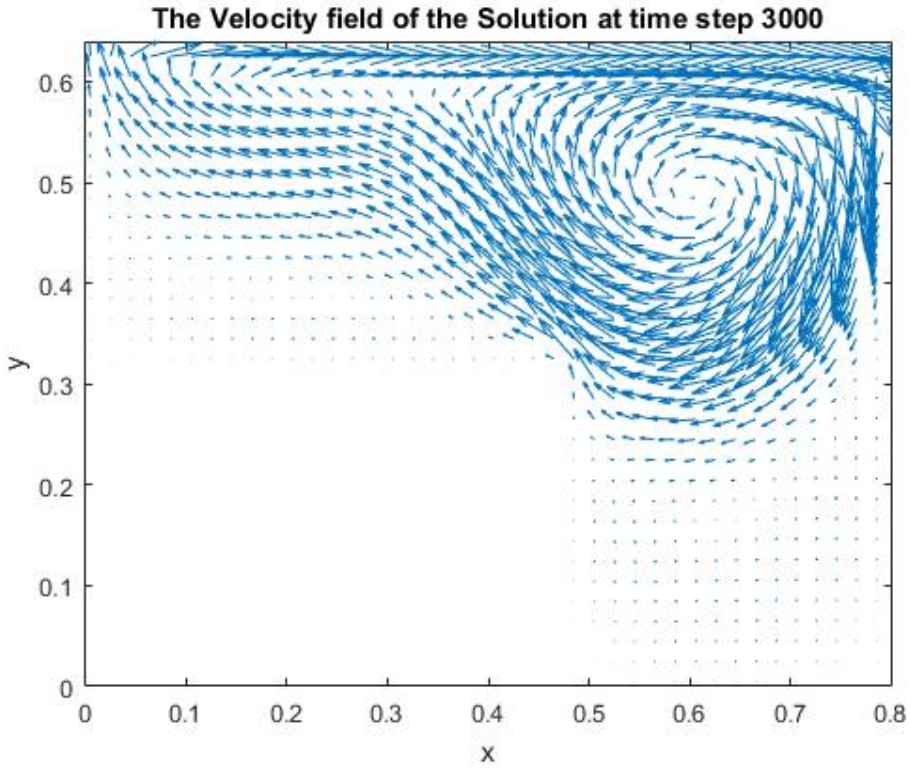


Figure 10: Velocity field

This plot clearly shows a main vortex centred at (0.605,0.49) spinning clockwise. We expect there to also be two smaller vortices at the ends of the L shape. The next two plots are an enlargement of those areas.

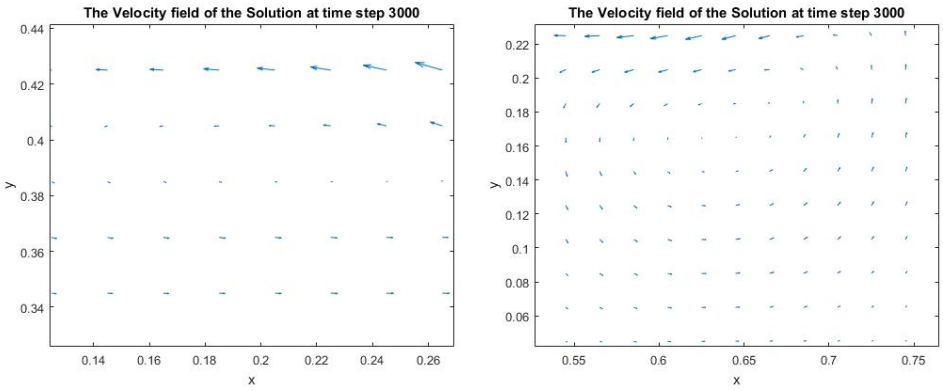


Figure 11: Enlarged Sections

Indeed the figure shows that two more vertices exist, one centred about (0.195,0.39) spinning anti-clockwise in an oval shape and the other centred at (0.64,0.17) spinning anti-clockwise too. Below are two plots of the vorticity of the solution. The one on the right has had it's colourmap clipped.

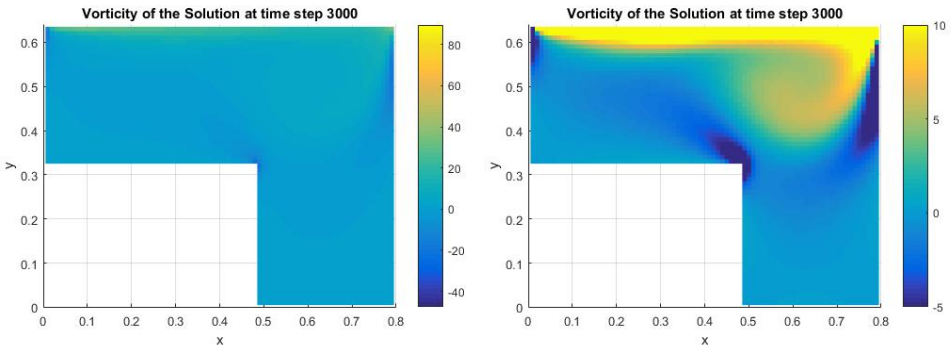


Figure 12: Vorticity

Finally for this Reynolds number there is a plot of the max change in solution between consecutive time steps (13). The plot shows an almost linear relationship with an average reduction in the max change of 1.0093 at each time step. The final values were 3.86862e-10 for u and 3.64871e-10 for v. This implies that the solution is close to a steady state of the numerical system.

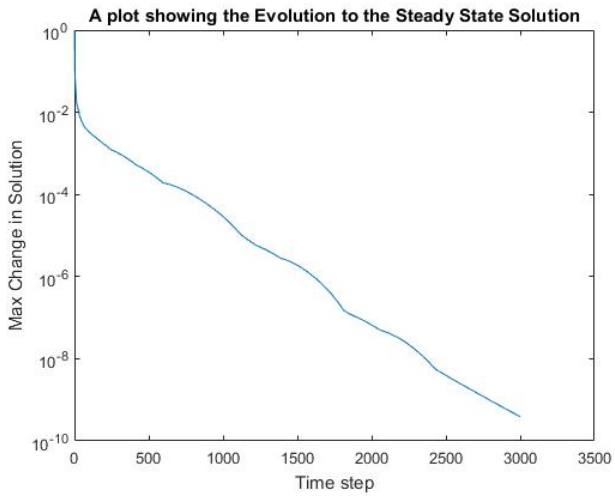
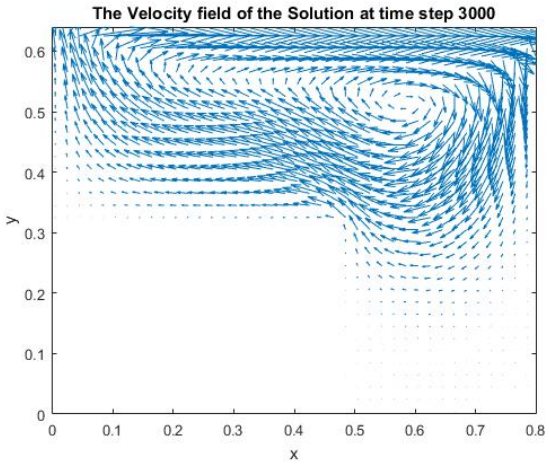


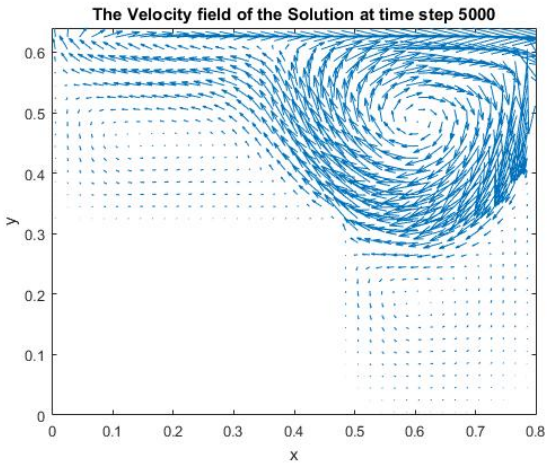
Figure 13: Change in Solution

Using the same code we can numerically solve many different fluid problem through adjusting the Reynolds number of the system. Following are the results of decreasing and increasing the Reynolds number about the 500 previously used.



If we reduce the Reynolds number to 100, this is equivalent to increasing the viscosity and as a result there is only one vortex.

Figure 14: Reynolds number = 100



If we increase the Reynolds number to 2000, this is equivalent to reducing the viscosity and as a result the vortices become more pronounced.

Figure 15: Reynolds number = 2000

In conclusion, the cravity flow code for a rectanglar domain has been successfully extended to an L shape domain. Only a small number of the possible problems on the L shape domain have been studied. The boundary and initial conditions could be changed to produce a variety of results and systems. The techniques use here also apply to adding more rectangular domains and so many different domains can be constructed, although I would recommend restructuring the code and the use of classes for easier domain decomposion.

Appendices

Matlab Code

```
function qnew = SemiLagrAdvectModified(u,v,q,qS,qn,qW,qE,n,m)

global dx dy
global dt Re


%...embedding
qq              = zeros(n+2,m+2);
qq(2:n+1,2:m+1) = q;

%...set the ghost values (four edges)
qq(1,2:m+1)   = 2*qW-qq(2,2:m+1);
qq(n+2,2:m+1) = 2*qE-qq(n+1,2:m+1);
qq(2:n+1,1)   = 2*qS-qq(2:n+1,2);
qq(2:n+1,m+2) = 2*qn-qq(2:n+1,m+1);

%...set the ghost values (four corners)
qq(1,1)       = -qq(2,2);
qq(n+2,1)     = -qq(n+1,2);
qq(n+2,m+2)   = -qq(n+1,m+1);
qq(1,m+2)     = -qq(2,m+1);

q1   = qq(2:n+1,2:m+1);

q2p  = qq(3:n+2,2:m+1);
q2m  = qq(1:n,2:m+1);

q3p  = qq(2:n+1,3:m+2);
q3m  = qq(2:n+1,1:m);

q4pp = qq(3:n+2,3:m+2);
q4mm = qq(1:n,1:m);
q4pm = qq(3:n+2,1:m);
q4mp = qq(1:n,3:m+2);

xi  = -u*dt/dx;
eta = -v*dt/dy;

Q2  = q2p.*(xi>0) + q2m.*(xi<0);
Q3  = q3p.*(eta>0) + q3m.*(eta<0);
Q4  = q4pp.*((xi>0) & (eta>0)) + q4mm.*((xi<0) & (eta<0)) + ...
    q4pm.*((xi>0) & (eta<0)) + q4mp.*((xi<0) & (eta>0));

qnew = (1-abs(xi)).*(1-abs(eta)).*q1 + ...
    abs(xi).*(1-abs(eta)).*Q2 + ...
    abs(eta).*(1-abs(xi)).*Q3 + ...
    abs(xi).*abs(eta).*Q4;
```

```
end


function qnew = SemiLagrAdvectMain(u,v,q,qS,qN,qW,qE)

global N M

u = reshape(u,N,M);
v = reshape(v,N,M);
q = reshape(q,N,M);

q1 = SemiLagrAdvectModified(u(1:48,33:64),v(1:48,33:64),...
    q(1:48,33:64),qS(1:48),qN(1:48),qW(33:64),...
    (q(48,33:64)+q(49,33:64))*0.5,48,32);
q2 = SemiLagrAdvectModified(u(49:80,1:64),v(49:80,1:64),...
    q(49:80,1:64),qS(49:80),qN(49:80),[qW(1:32),...
    (q(48,33:64)+q(49,33:64))*0.5],qE,32,64);

qnew = zeros(80,64);
qnew(1:48,33:64) = q1;
qnew(49:80,1:64) = q2;
qnew(1:48,1:32) = NaN;

end


%---------------------------------------
% Driver routine for Question one
%---------------------------------------

clear all
close all

global Rd Ad Pd
global Rp Ap Pp
global xLen yLen
global dx dy
global N M
global dt Re

%...Reynolds number
Re = 500;

%...domain size
xLen  = 0.8;
yLen  = 0.64;

N     = 80;
M     = 64;

x       = linspace(0,xLen,N+1);
y       = linspace(0,yLen,M+1);
```

```
xc      = (x(1:end-1)+x(2:end))/2;
yc      = (y(1:end-1)+y(2:end))/2;
[yy,xx] = meshgrid(yc,xc);
dx      = xLen/N;
dy      = yLen/M;
dt      = min(dx,dy)/1.5;

q   = zeros(N,M);
q(16:32,48:52) = 1;
ax    = ones(N,M)*1;
ay    = zeros(N,M)*0.1;

q(1:48,1:32) = NaN;
ax(1:48,1:32)  = NaN;
ay(1:48,1:32)  = NaN;

qS =  zeros(N,1);
qN =  zeros(N,1);
qW =  zeros(1,M);
qE =  zeros(1,M);


for i=1:50

    %...semi-Lagrangian advection
    q    = SemiLagrAdvectMain(ax,ay,q,qS,qN,qW,qE);


    if (mod(i,1)==0)
        fprintf('time step %i \n',i)
        %...graphics output
        figure(1);
        surf(xx,yy,q,'EdgeColor','None')
        axis image
        axis([0 xLen 0 yLen]);
        drawnow
        %pause(0.1)
    end
end


function unew = DiffusionSolveModified(u,Ubc)

global Rd Ad Pd
global dt D

u = u(:);
Ubc = Ubc(:);

%...setup right-hand side (for Crank-Nicolson)
rhs = u + dt/2*D*Ad{1}*u + Ubc;

rn  = norm(rhs - Ad{1}*u);
```

```
ic  = 0;
while ( (rn > 1e-10) && (ic < 10) )
    u  = MGVd(1,u,rhs);
    rn = norm(rhs - Ad{1}*u);
    ic = ic + 1;
end
%fprintf('iter %i (d)resnorm %g\n',ic,rn)

unew = reshape(u,64,32);

end


function qnew = DiffusionSolveMain(q,Qbc)

global dx dy
global dt D

q1 = q(1:64,33:64);
q2 = q(49:80,1:64);
error = 1;
rate = 2;
i = 0;

while error>1e-6 && rate>1.1 && i < 10

    B1 = zeros(64,32);
    B1(49:64,1) = (q2(1:16,32) + q2(1:16,33))*dt*D/dy/dy/2;
    B1(64,1:32) = B1(64,1:32) + (q2(16,33:64) + q2(17,33:64))*dt*D/dx/dx/2;
    q1 = DiffusionSolveModified(q(1:64,33:64),Qbc(1:64,33:64)+B1);

    B2 = zeros(32,64);
    B2(1,33:64) = (q1(48,1:32) + q1(49,1:32))*dt*D/dx/dx/2;
    q2 = DiffusionSolveModified(q(49:80,1:64)',(Qbc(49:80,1:64)+B2)')';

    temp = max(abs(q1(49:64,1)-q2(1:16,33)));
    temp = max(temp,max(abs(q1(64,:)-q2(16,33:64))));
    temp = max(temp,max(abs(q1(49,:)-q2(1,33:64))));
    rate = error/temp;
    error = temp;

    i = i + 1;
end

qnew = zeros(80,64);
qnew(1:64,33:64) = q1;
qnew(49:80,1:64) = q2;
qnew(1:48,1:32) = NaN;

end
```

```
%--------------------------------------
% Driver routine for Question two
%--------------------------------------

clear all
close all

global Rd Ad Pd
global xLen yLen
global dx dy
global N M
global D dt

%...diffusion coefficient
D = 1/500;

%...domain size
xLen  = 0.80;
yLen  = 0.64;

%...resolution, fine-mesh spacing
N     = 80;
M     = 64;

x       = linspace(0,xLen,N+1);
y       = linspace(0,yLen,M+1);
xc      = (x(1:end-1)+x(2:end))/2;
yc      = (y(1:end-1)+y(2:end))/2;
[yy,xx] = meshgrid(yc,xc);
dx      = xLen/N;
dy      = yLen/M;
dt      = min(dx,dy)/1.5;

%...setup the hierarchy of R,A,P
getRPd(64,32);
getAd(64,32);

%...setup potential field, boundary conditions
q    = zeros(N,M);
q(1:48,1:32)  = NaN;


uS =  zeros(N,1);
uN =  ones(N,1);
uW =  zeros(1,M);
uE =  zeros(1,M);

Qbc      = zeros(N,M);
Qbc(1,33:64) = uW(33:64)*dt*D/dx/dx;
Qbc(49,1:32) = uW(1:32)*dt*D/dx/dx;
Qbc(N,:) = uE*dt*D/dx/dx;
Qbc(1:48,33) = uS(1:48)*dt*D/dy/dy;
Qbc(49:80,1) = uS(49:80)*dt*D/dy/dy;
Qbc(:,M) = uN*dt*D/dy/dy;
```

```
for i=1:150

    %...diffusion
    q    = DiffusionSolveMain(q,Qbc);

    if (mod(i,10)==0)
        fprintf('time step %i \n',i)
        %...graphics output
        figure(1);
        surf(xx,yy,q,'EdgeColor','None')
        colorbar;
        axis image
        axis([0 xLen 0 yLen]);
        title('A plot of the Solution at Time = 1')
        xlabel('x')
        ylabel('y')
        drawnow
    end
end


function getAp(N,M)
%=====================================
% set up a hierarchy of Laplace
% operators (as cell arrays)
%=====================================

global Rp Ap Pp
global dx dy

%...number of levels and initialization
kk   = length(Pp);
Ap   = cell(kk+1,2);

%...Laplace operator
AAx      = spdiags(ones(N,1)*[1 -2 1]/dx/dx,-1:1,N,N);
AAx(1,1) = -1/dx/dx;
AAx(N,N) = -3/dx/dx;

AAy      = spdiags(ones(M,1)*[1 -2 1]/dy/dy,-1:1,M,M);
AAy(1,1) = -1/dy/dy;
AAy(M,M) = -1/dy/dy;

AAA  = kron(speye(M),AAx) +kron(AAy,speye(N));
AAA(3*N/4+1:N,3*N/4+1:N) = AAA(3*N/4+1:N,3*N/4+1:N) - 2/dy/dy*speye(N/4);
AAA(1,1) = -3/dx/dx;
%...lower-level operators (Galerkin condition)
Ap{1,1} = AAA;
for i=2:kk+1
    Ap{i,1} = Rp{i-1,1}*Ap{i-1,1}*Pp{i-1,1};
end
```

```
%...Laplace operator
AAx(N,N) = -1/dx/dx;

AAA  = kron(speye(M),AAx) + kron(AAy,speye(N));
AAA(N/2+1:N,N/2+1:N) = AAA(N/2+1:N,N/2+1:N) - 2/dy/dy*speye(N/2);
%...lower-level operators (Galerkin condition)
Ap{1,2} = AAA;
for i=2:kk+1
    Ap{i,2} = Rp{i-1,2}*Ap{i-1,2}*Pp{i-1,2};
end
end



function getRPp(N,M)
%=======================================
% set up a hierarchy of restriction
% and prolongation matrices (as cell
% arrays) (Neumann version)
%=======================================

global Rp Ap Pp
global ilevmin

%...number of levels and initialization
kk      = log(N)/log(2)-1;
ll      = log(M)/log(2)-1;
kl      = min(kk,ll);
ilevmin = kl;
Rp      = cell(kl,2);
Pp      = cell(kl,2);

%...set up prolongation
NN = N/2;
MM = M/2;
for i=1:kl

    %Neumann East, Dirichlet West
    PPx = sparse(2*NN,NN);
    for j=1:NN-1
        PPx(2*j:2*j+1,j:j+1) = [0.75 0.25; 0.25 0.75];
    end
    PPx(1,1)     = 1;
    PPx(2*NN,NN) = 0.5;

    %Neumann North and South
    PPy1 = sparse(2*MM,MM);
    for j=1:MM-1
        PPy1(2*j:2*j+1,j:j+1) = [0.75 0.25; 0.25 0.75];
    end
    PPy1(1,1)     = 1;
    PPy1(2*MM,MM) = 1;

    %Neumann North, Dirichlet South
```

```
    PPy2 = PPy1;
    PPy2(1,1)     = 0.5;

    %Splicing for Omega 1
    E = cat(2,eye(3*NN/4), zeros(3*NN/4,NN/4));
    F = kron(eye(MM),E);
    G = zeros(NN/4,NN);
    G(1:NN/4,3*NN/4+1:end) = eye(NN/4);
    H = kron(eye(MM),G);

    Pp{i,1} = kron(PPy1,PPx(:,1:3*NN/4))*F + kron(PPy2,PPx(:,3*NN/4+1:NN))*H;

    %Adding a single Dirichlet point
    Pp{i,1}(1,1) = 0.5;

    %Changing East boundary to Neumann
    PPx(2*NN,NN) = 1;

    %Splicing for Omega 2
    E = cat(2,eye(NN/2), zeros(NN/2,NN/2));
    F = kron(eye(MM),E);
    G = zeros(NN/2,NN);
    G(1:NN/2,NN/2+1:end) = eye(NN/2);
    H = kron(eye(MM),G);

    Pp{i,2} = kron(PPy1,PPx(:,1:NN/2))*F + kron(PPy2,PPx(:,NN/2+1:NN))*H;

    NN = NN/2;
    MM = MM/2;

end

%...set up restriction (transpose of prolongation)
for i=1:kl
    Rp{i,1} = transpose(Pp{i,1});
    Rp{i,2} = transpose(Pp{i,2});
end
end


function p = PoissonSolve(p0,rhs,i)

global Rp Ap Pp
global xLen yLen
global dx dy

%...V-cycles
p0 = p0(:);
rhs = rhs(:);

rn  = norm(rhs - Ap{1,i}*p0);
ic  = 0;
p   = p0(:);
```

```
%m = 0;
while ( (rn > 1e-8) && (ic < 10) )
    p  = MGVp(1,p,rhs,i);
    r  = rhs - Ap{1,i}*p;
    rn = norm(r);
    ic = ic + 1;
    %rhs = rhs - sum(r)*(dx/0.64)*(dy/0.32); <-----?
    %fprintf('iter %i (p)resnorm %g\n',ic,rn)
    %m = m + 1;
    %rr(m) = norm(r);
end

if (ic >= 10)
    %disp('max iter')
end


%   figure;
%   semilogy(rr,'-o'); grid on
%   xlabel('Iteration');
%   ylabel('log(Residual)');
%   title('The Logarithm of the Residual against number of V cycles');

p = reshape(p,64,32);

end


function pnew = PoissonSolveMain(p,rhs)

global dx dy

p1 = p(1:64,33:64);
p2 = p(49:80,1:64);
error = 1;

i = 0;
while (error>1e-6 && i < 100)
    B1 = zeros(64,32);
    B1(49:64,1) = -(p2(1:16,32) + p2(1:16,33))/dy/dy;
    B1(64,1:32) = B1(64,1:32) - (p2(16,33:64) + p2(17,33:64))/dx/dx;
    p1 = PoissonSolve(p(1:64,33:64),rhs(1:64,33:64)+B1,1);

    B2 = zeros(32,64);
    B2(1,33:64) = -(p1(48,1:32) + p1(49,1:32))/dx/dx;
    p2 = PoissonSolve(p(49:80,1:64)',(rhs(49:80,1:64)+B2)',2)';

    temp = max(abs(p1(49:64,1)-p2(1:16,33)));
    temp = max(temp,max(abs(p1(64,:)-p2(16,33:64))));
    temp = max(temp,max(abs(p1(49,:)-p2(1,33:64))));
    error = temp;
```

```
    i = i + 1;

end

fprintf(1,'iter: %i error: %g \n',i,error);

pnew = zeros(80,64);
pnew(1:64,33:64) = p1;
pnew(49:80,1:64) = p2;
pnew(1:48,1:32) = NaN;

end


function [qx,qy] = Diff2(q,N,M,typ,qS,qN,qW,qE)

global xLen yLen

dx = xLen/N;
dy = yLen/M;

qq = reshape(q,N,M);
qG = zeros(N+2,M+2);

qG(2:N+1,2:M+1)   =  qq;
if (typ=='D')
    qG(1,34:65)   = 2*qW(33:64)-q(1,33:64);
    qG(49,2:33)   = 2*qW(1:32)-q(49,1:32);
    qG(82,2:65)   = 2*qE-q(80,:);
    qG(2:49,33)   = 2*qS(1:48)-q(1:48,33);
    qG(50:81,1)   = 2*qS(49:80)-q(49:80,1);
    qG(2:81,66)   = 2*qN-q(:,64);
elseif (typ=='N')
    qG(1,34:65)    = q(1,33:64);
    qG(49,2:33)    = q(49,1:32);
    qG(82,2:65)    = q(80,:);
    qG(2:49,33)    = q(1:48,33);
    qG(50:81,1)    = q(49:80,1);
    qG(2:81,66)    = q(:,64);
end

qx = zeros(N,M);
qy = zeros(N,M);
qx(1:80,33:64) = (qG(3:82,34:65)-qG(1:80,34:65))/(2*dx);
qx(49:80,1:32) = (qG(51:82,2:33)-qG(49:80,2:33))/(2*dx);
qy(1:48,33:64) = (qG(2:49,32+3:M+2)-qG(2:49,32+1:M))/(2*dy);
qy(49:80,1:64) = (qG(50:81,3:M+2)-qG(50:81,1:M))/(2*dy);
qx(1:48,1:32)  = NaN;
qy(1:48,1:32)  = NaN;

end
```

```
%--------------------------------------
% Driver routine for Question four
%--------------------------------------

%clear all
close all

global Rd Ad Pd
global Rp Ap Pp
global xLen yLen
global dx dy
global N M
global dt Re

%...Reynolds number
Re = 2000;

%...domain size
xLen  = 0.80;
yLen  = 0.64;

%...resolution, fine-mesh spacing
%   and number of cycles
ncyc  = 10;
N     = 80;
M     = 64;

x       = linspace(0,xLen,N+1);
y       = linspace(0,yLen,M+1);
xc      = (x(1:end-1)+x(2:end))/2;
yc      = (y(1:end-1)+y(2:end))/2;
[yy,xx] = meshgrid(yc,xc);
dx      = xLen/N;
dy      = yLen/M;
dt      = min(dx,dy)/1.5;

%...setup the hierarchy of R,A,P
getRPd(64,32);
getRPp(64,32);
getAd(64,32);
getAp(64,32);

p0   = zeros(N,M); p0(1:48,1:32) = NaN;
u    = zeros(N,M); u(1:48,1:32)  = NaN;
v    = zeros(N,M); v(1:48,1:32)  = NaN;

uS =  zeros(N,1);
uN =  ones(N,1);
uW =  zeros(1,M);
uE =  zeros(1,M);

vS = zeros(N,1);
vN = zeros(N,1);
vW = zeros(1,M);
```

```
vE = zeros(1,M);

Ubc      = zeros(N,M);
Ubc(1,33:64) = uW(33:64)*dt/Re/dx/dx;
Ubc(49,1:32) = uW(1:32)*dt/Re/dx/dx;
Ubc(N,:) = uE*dt/Re/dx/dx;
Ubc(1:48,33) = uS(1:48)*dt/Re/dy/dy;
Ubc(49:80,1) = uS(49:80)*dt/Re/dy/dy;
Ubc(:,M) = uN*dt/Re/dy/dy;

Vbc      = zeros(N,M);
Vbc(1,33:64) = vW(33:64)*dt/Re/dx/dx;
Vbc(49,1:32) = vW(1:32)*dt/Re/dx/dx;
Vbc(N,:) = vE*dt/Re/dx/dx;
Vbc(1:48,33) = vS(1:48)*dt/Re/dy/dy;
Vbc(49:80,1) = vS(49:80)*dt/Re/dy/dy;
Vbc(:,M) = vN*dt/Re/dy/dy;


% figure(1);
% quiver(xx(1:2:end,1:2:end),yy(1:2:end,1:2:end),u(1:2:end,1:2:end)0,...
%  v(1:2:end,1:2:end),6)
% axis image
% axis([0 xLen 0 yLen]);
% drawnow

error = ones(1,5001);
rate = 2*ones(1,5000);

for i=1:5000
    %clc;
    %fprintf('time step %i \n',i)

    %...semi-Lagrangian advection
    uast    = SemiLagrAdvectMain(u,v,u,uS,uN,uW,uE);
    vast    = SemiLagrAdvectMain(u,v,v,vS,vN,vW,vE);

    %...diffusion
    unew    = DiffusionSolveMain(uast,Ubc);
    vnew    = DiffusionSolveMain(vast,Vbc);

    %...computing divergence
    %     [ux,uy] = Diff(unew,N,M,uS,uN,uW,uE);
    %     [vx,vy] = Diff(vnew,N,M,vS,vN,vW,vE);
    [ux,uy] = Diff2(unew,N,M,'D',uS,uN,uW,uE);
    [vx,vy] = Diff2(vnew,N,M,'D',vS,vN,vW,vE);
    Div     = ux + vy; Div(1,33) = 0;

    %...solving for pressure
    pnew    = PoissonSolveMain(p0,Div);
    p0      = pnew;

    %...correcting velocities
    %     [px,py] = Diff(pnew,N,M,vS,vN,vW,vE,typ);
```

```
    [px,py] = Diff2(pnew,N,M,'N',vS,vN,vW,vE);
    norm1 = max(max(abs(unew - px - u)));
    norm2 = max(max(abs(vnew - py - v)));
    u       = unew - px;
    v       = vnew - py;

    error(i+1) = max(norm1,norm2);
    rate(i) = error(i)/error(i+1);

    %fprintf(1,'MaxDiffU: %g MaxDiffV: %g \n',norm1,norm2);

    if (mod(i,100)==0)
        k = 2;
        fprintf('time step %i \n',i)
        fprintf(1,'MaxDiffU: %g MaxDiffV: %g \n',norm1,norm2);
        %...graphics output
        figure(1);
        [ux,uy] = Diff2(u,N,M,'D',uS,uN,uW,uE);
        [vx,vy] = Diff2(v,N,M,'D',vS,vN,vW,vE);
        vort    = uy-vx;
        surf(xx,yy,vort,'EdgeColor','None');
        axis image
        axis([0 xLen 0 yLen]);
        colorbar;
        drawnow

        figure(2);
        %surf(xx,yy,p0,'EdgeColor','None');
        quiver(xx(1:k:end,1:k:end),yy(1:k:end,1:k:end),u(1:k:end,1:k:end)...
            ,v(1:k:end,1:k:end),12/k)
        axis image
        axis([0 xLen 0 yLen]);
        drawnow
    end

end

figure(3);
semilogy(error);
xlabel('Time step')
ylabel('Max Change in Solution')
title('A plot showing the Evolution to the Steady State Solution')
```

References

[1] Prof. Peter Schmid provided a Matlab scripts containing the cavity flow code and its components, that was adapted and used in this project.