SMA5212/16.920J/2.097J - Numerical Methods for Partial Differential Equations

Massachusetts Institute of Technology — Singapore - MIT Alliance

SOLUTIONS

Problem Set 1 - Finite Differences and Iterative Methods

Handed out: February 10, 2003 — Due: March 10, 2003

Problem 1 - Flow in a channel

1) The four corners in the physical domain are located at $(x, y) = (0, 0), (b/2, 0), (b/2+a, h), (0, h)$. A mapping between the coordinates (ξ, η) on the unit square and (x, y) is

$$x = (b/2 + a\eta)\xi \tag{1}$$
$$y = h\eta. \tag{2}$$

The equation is transformed to

$$\frac{-1}{J^2}(a'\, u_{\xi\xi} - 2b'\, u_{\xi\eta} + c'\, u_{\eta\eta} + d'\, u_\eta + e'\, u_\xi) = f, \tag{3}$$

where

$$J = x_\xi y_\eta - x_\eta y_\xi = (b/2 + a\eta)h \tag{4}$$
$$a' = x_\eta^2 + y_\eta^2 = a^2\xi^2 + h^2 \tag{5}$$
$$b' = x_\xi x_\eta + y_\xi y_\eta = (b/2 + a\eta)a\xi \tag{6}$$
$$c' = x_\xi^2 + y_\xi^2 = (b/2 + a\eta)^2 \tag{7}$$
$$\alpha = a'x_{\xi\xi} - 2b'x_{\xi\eta} + c'x_{\eta\eta} = -2(b/2 + a\eta)a^2\xi \tag{8}$$
$$\beta = a'y_{\xi\xi} - 2b'y_{\xi\eta} + c'y_{\eta\eta} = 0 \tag{9}$$
$$d' = \frac{y_\xi\alpha - x_\xi\beta}{J} = 0 \tag{10}$$
$$e' = \frac{x_\eta\beta - y_\eta\alpha}{J} = 2a^2\xi. \tag{11}$$

The normal direction of the top boundary is $(n_x, n_y) = (0, 1)$, and the normal derivative is transformed to

$$\frac{\partial u}{\partial n} = \frac{1}{J}\left[(y_\eta n_x - x_\eta n_y)u_\xi + (-y_\xi n_x + x_\xi n_y)u_\eta\right] = \frac{1}{h}\left(-\frac{a}{b/2 + a}\xi u_\xi + u_\eta\right). \tag{12}$$

At the left boundary, $(n_x, n_y) = (-1, 0)$ and

$$\frac{\partial u}{\partial n} = -\frac{1}{b/2 + a\eta}u_\xi. \tag{13}$$

2) Forward Taylor series expansion:

$$v_{i+1} = v_i + \Delta x v'(x_i) + \frac{\Delta x^2}{2}v''(x_i) + \frac{\Delta x^3}{6}v^{(3)}(x_i) + \frac{\Delta x^4}{24}v^{(4)}(x_i) + \mathcal{O}(\Delta x^5). \tag{14}$$

Backward Taylor series expansion:

$$v_{i-1} = v_i - \Delta x v'(x_i) + \frac{\Delta x^2}{2} v''(x_i) - \frac{\Delta x^3}{6} v^{(3)}(x_i) + \frac{\Delta x^4}{24} v^{(4)}(x_i) + \mathcal{O}(\Delta x^5). \tag{15}$$

Subtract the backward T.S. from the forward T.S. to get an $\mathcal{O}(h^2)$ approximation of the first derivative:

$$\left.\frac{\partial v}{\partial x}\right|_i = \frac{v_{i+1} - v_{i-1}}{2\Delta x} + \mathcal{O}(\Delta x^2). \tag{16}$$

Add the backward T.S. to the forward T.S. to get an $\mathcal{O}(h^2)$ approximation of the second derivative:

$$\left.\frac{\partial^2 v}{\partial x^2}\right|_i = \frac{v_{i+1} - 2v_i + v_{i-1}}{\Delta x^2} + \mathcal{O}(\Delta x^2). \tag{17}$$

For the one-sided difference approximations at the boundaries, we need one more Taylor Series expansion:

$$v_{i+2} = v_i + 2\Delta x v'(x_i) + 2^2\frac{\Delta x^2}{2} v''(x_i) + 2^3\frac{\Delta x^3}{6} v^{(3)}(x_i) + 2^4\frac{\Delta x^4}{24} v^{(4)}(x_i) + \mathcal{O}(\Delta x^5). \tag{18}$$

Solving for the first derivative gives a forward difference expression:

$$\left.\frac{\partial v}{\partial x}\right|_i = \frac{-3v_i + 4v_{i+1} - v_{i+2}}{2\Delta x} + \mathcal{O}(\Delta x^2), \tag{19}$$

and similarly for the backward difference:

$$\left.\frac{\partial v}{\partial x}\right|_i = \frac{3v_i - 4v_{i-1} + v_{i-2}}{2\Delta x} + \mathcal{O}(\Delta x^2). \tag{20}$$

For the cross derivative we need the multidimensional Taylor Series expansion:

$$\begin{aligned} v_{i+1,j+1} = v_{i,j} &+ \left(\Delta x\frac{\partial}{\partial x} + \Delta y\frac{\partial}{\partial y}\right) v(x_{i,i}) + \frac{1}{2}\left(\Delta x\frac{\partial}{\partial x} + \Delta y\frac{\partial}{\partial y}\right)^2 v(x_{i,i}) + \\ &+ \frac{1}{6}\left(\Delta x\frac{\partial}{\partial x} + \Delta y\frac{\partial}{\partial y}\right)^3 v(x_{i,i}) + \frac{1}{24}\left(\Delta x\frac{\partial}{\partial x} + \Delta y\frac{\partial}{\partial y}\right)^4 v(x_{i,i}) + \mathcal{O}(\Delta x^5), \end{aligned} \tag{21}$$

and similarly for $v_{i+1,j-1}$, $v_{i-1,j+1}$, and $v_{i-1,j-1}$. Form the derivatives in the x- and the y-direction, respectively, to get the cross derivative approximation:

$$\left.\frac{\partial^2 v}{\partial x \partial y}\right|_i = \frac{v_{i+1,j+1} - v_{i+1,j-1} - v_{i-1,j+1} + v_{i-1,j-1}}{4\Delta x \Delta y} + \mathcal{O}(\Delta x^2). \tag{22}$$

The integral $\hat{Q}$ is evaluated as follows. First make a change of variable to transform the integral in x, y space to ξ, η space:

$$Q = \int\int u\, dxdy = \int\int u|J(\xi,\eta)|\, d\xi d\eta \tag{23}$$

This is evaluated numerically by averaging the values at neighboring grid points to get the cell averages, summing over all cells, and multiplying by the (constant) cell area.

3) The code `channelflow.m` can be found in Appendix A. It is written as a function that takes four input parameters: l, b, h, and N. The output is the flowrate $\hat{Q}$, the grid x, y, and the solution u. The calling syntax is:

```
[Q,x,y,u]=channelflow(l,b,h,N);
```

4) The solutions for $l = 3$, $h = 1$, and $b = 0.0, 0.5, 1.0$ are shown in Figure 1. The corresponding $\hat{Q}$ values are 0.0436 ($b = 0.0$), 0.0574 ($b = 0.5$), and 0.0285 ($b = 1.0$).

5) In Figure 2, the errors in $\hat{Q}$ and in the solution (L_∞ and L_2 norms) are shown, for $l = 3$, $h = 1$, and $b = 0.0, 0.5, 1.0$. All slopes are above 2 (except L_∞ for $b = 0.5$, because of a bad result for the coarse grid $N = 11$). This means that the convergence is second-order.

6) The Pareto Optimal Frontier is shown in Figure 3. This plot can be used to optimize the design. If, for example, some additional constraint is added (the flowrate $\hat{Q} \geq \hat{Q}_0$, etc), the linear interpolation between the points give an approximate value for the maximum possible I, and the corresponding design parameters.

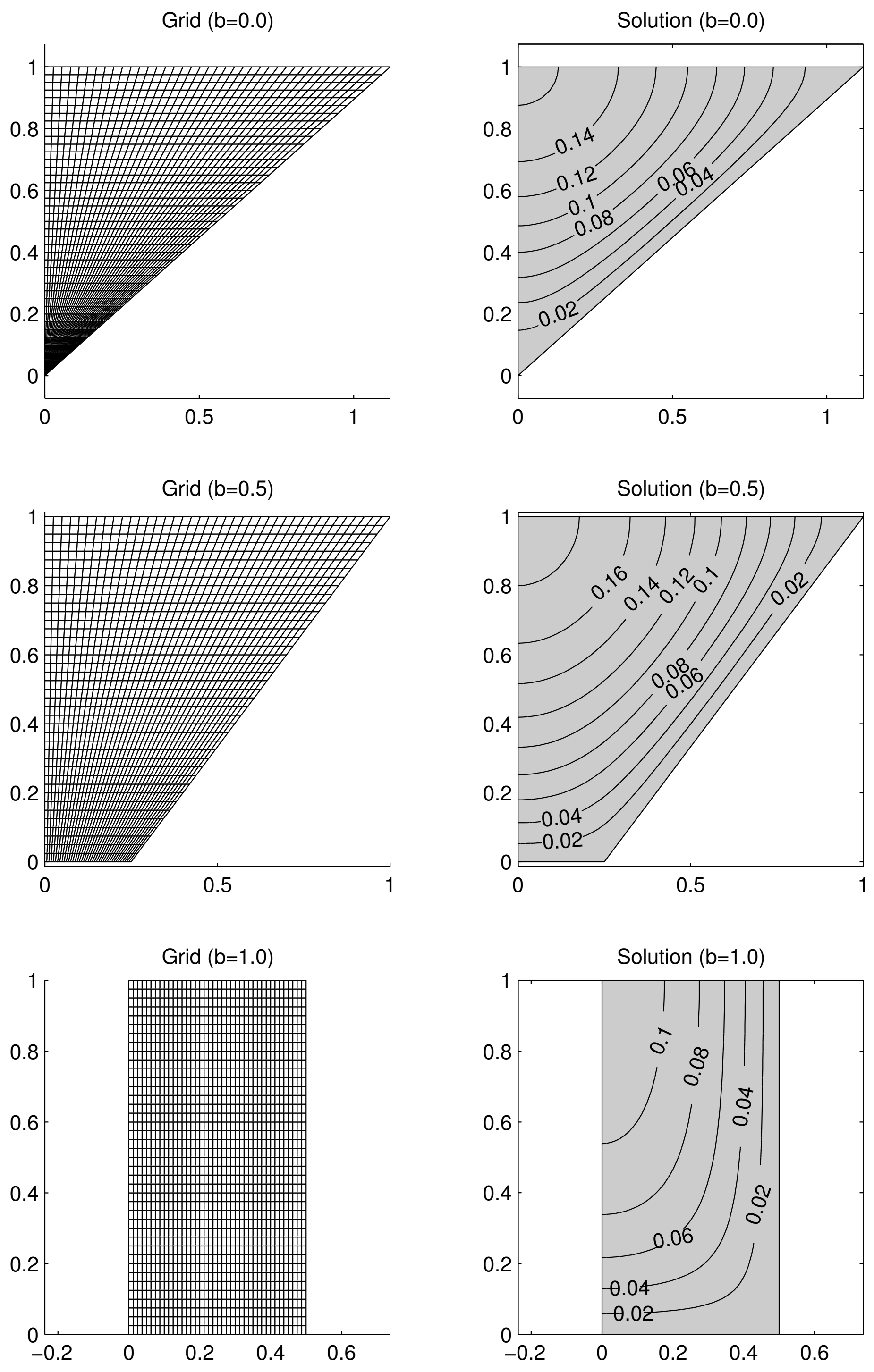
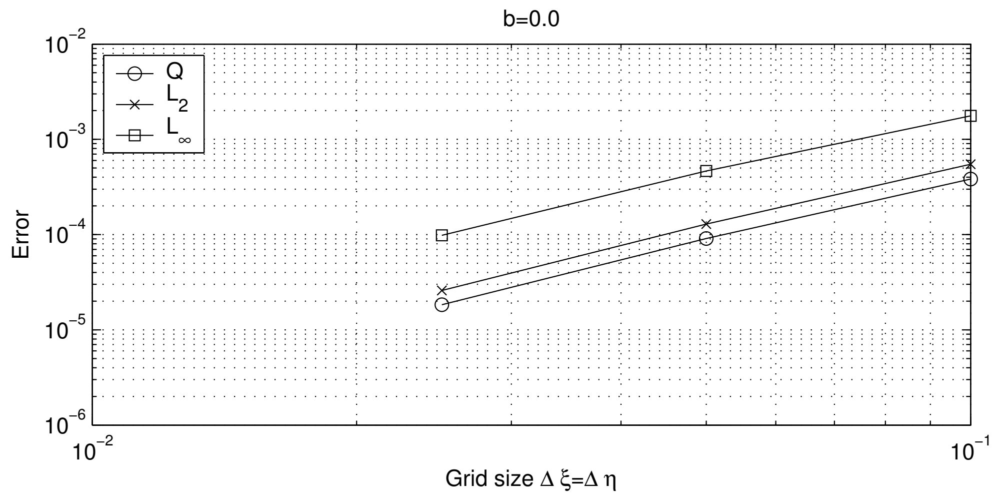
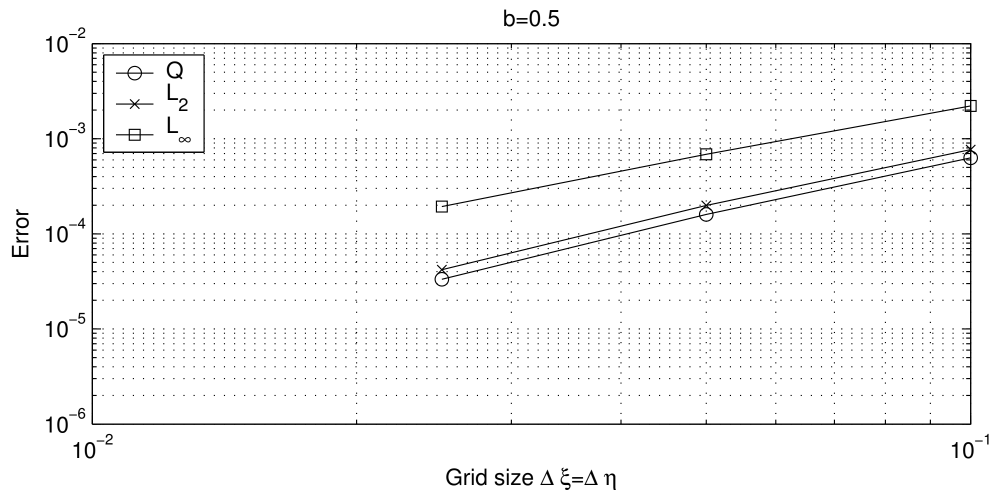


Figure 1: Grids and solutions for $l = 3$, $h = 1$, and varying b.





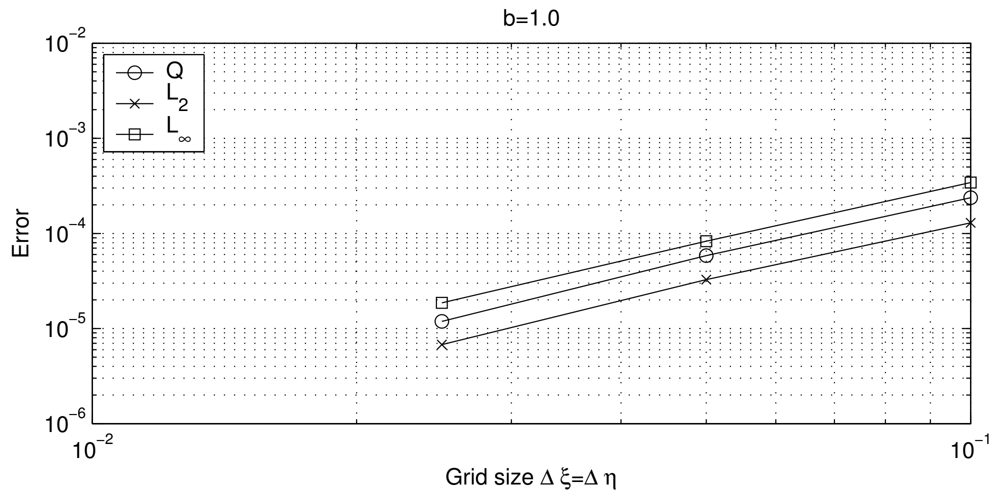


Figure 2: Plots of the errors in $\hat{Q}$ and in the solution (L_∞ and L_2 norms), for $l = 3$, $h = 1$, and varying b. All slopes are about 2 because of the second-order convergence.

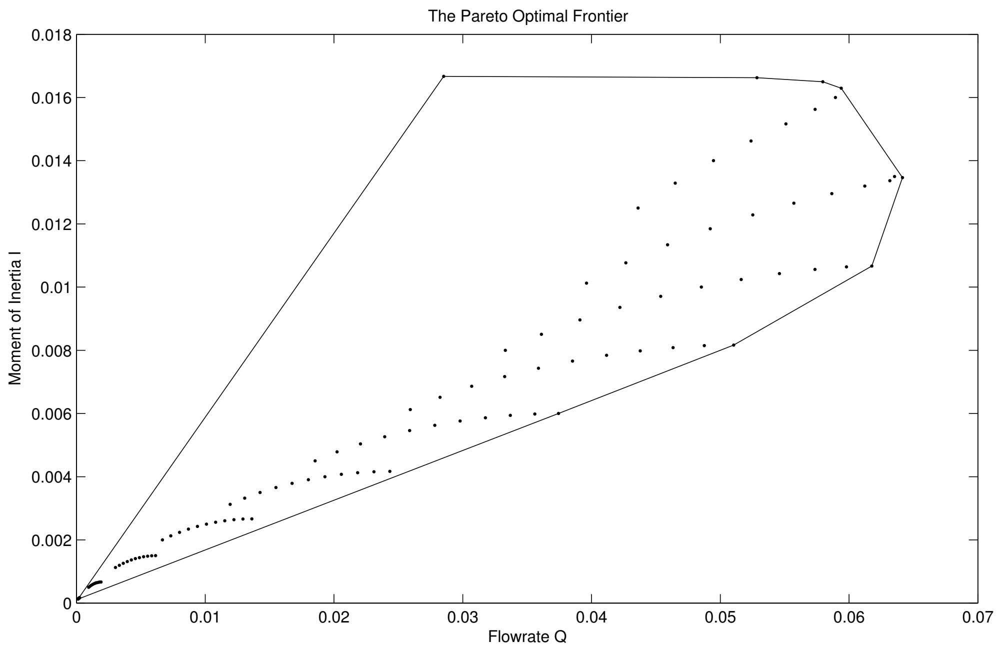


Figure 3: The Pareto Optimal Frontier.

Problem 2 - Iterative Methods: Jacobi, G-S, Multigrid

Problem Statement

Questions

1) Discretize ϕ on the domain according to $\phi_{i,j} \approx \phi(i\Delta x, j\Delta y)$, for $0 \leq i, j \leq N$ and $\Delta x = \Delta y = 1/N$. Form an initial guess $\phi^0_{i,j}$ and the right hand side $f_{i,j}$. With the Laplace operator discretized using second-order approximation, the Jacobi iteration is

$$\phi^{r+1}_{i,j} = \frac{1}{4}(\phi^r_{i+1,j} + \phi^r_{i-1,j} + \phi^r_{i,j+1} + \phi^r_{i,j-1} + h^2 f_{i,j}) \tag{24}$$

and the Gauss-Seidel iteration is, for the natural update ordering:

$$\phi^{r+1}_{i,j} = \frac{1}{4}(\phi^r_{i+1,j} + \phi^{r+1}_{i-1,j} + \phi^r_{i,j+1} + \phi^{r+1}_{i,j-1} + h^2 f_{i,j}) \tag{25}$$

Below is pseudo code for the Jacobi method,

```
for $i = 1$ to $N$
    for $j = 1$ to $N$
        $f_{i,j} = f(i\Delta x, j\Delta y)$
        $\phi^0_{i,j} = \phi^0(i\Delta x, j\Delta y)$.
    end
end
for $r = 0, 1, 2, \ldots$
    for $i = 1$ to $N$
        for $j = 1$ to $N$
            $\phi^{r+1}_{i,j} = \frac{1}{4}(\phi^r_{i+1,j} + \phi^{r+1}_{i-1,j} + \phi^r_{i,j+1} + \phi^{r+1}_{i,j-1} + h^2 f_{i,j})$
        end
    end
end
```

and below is pseudo code for the Gauss-Seidel method.

```
for $i = 1$ to $N$
    for $j = 1$ to $N$
        $f_{i,j} = f(i\Delta x, j\Delta y)$
        $\phi^0_{i,j} = \phi^0(i\Delta x, j\Delta y)$.
    end
end
for $r = 0, 1, 2, \ldots$
    for $i = 1$ to $N$
        for $j = 1$ to $N$
            $\phi^{r+1}_{i,j} = \frac{1}{4}(\phi^r_{i+1,j} + \phi^{r+1}_{i-1,j} + \phi^r_{i,j+1} + \phi^{r+1}_{i,j-1} + h^2 f_{i,j})$
        end
    end
end
```

In the code, a matrix formulation is used instead, since it generalizes nicely to the multigrid routines. Also, the G-S algorithm is hard to vectorize in MATLAB® with the update ordering shown above. The Laplace operator and the right hand side are discretized to form the system of linear equations $A\boldsymbol{\phi} = \mathbf{f}$. Define D, L, U as $A = D - L - U$, with D diagonal, L lower triangular, and U upper triangular. The iteration can then be written as (Jacobi)

$$D\boldsymbol{\phi}^{r+1} = (L + U)\boldsymbol{\phi}^r + \mathbf{f} \tag{26}$$

and (Gauss-Seidel)

$$(D - L)\boldsymbol{\phi}^{r+1} = U\boldsymbol{\phi}^r + \mathbf{f} \tag{27}$$

With a relaxation parameter ω, the Jacobi iteration changes to

$$\phi_{i,j}^{r+1} = \frac{\omega}{4}(\phi_{i+1,j}^r + \phi_{i-1,j}^r + \phi_{i,j+1}^r + \phi_{i,j-1}^r + h^2 f_{i,j}) + (1-\omega)\phi_{i,j}^r \tag{28}$$

and similarly for Gauss-Seidel. In matrix form,

$$\boldsymbol{\phi}^{r+1} = \omega D^{-1}\left((L + U)\boldsymbol{\phi}^r + \mathbf{f}\right) + (1-\omega)\boldsymbol{\phi}^r \tag{29}$$

2) a. In the implementation, the function `assemble` creates the matrix A and the right hand side $\mathbf{f}$ using a given grid size and a given block configuration. The linear system of equations can then by solved with the `jac` function or with the `gs` function. See Appendix B for all code.

In the solvers, the possibility to do convergence study is included by asking for a second output argument. To get the exact solution, MATLAB®'s direct sparse solver (\) is used. For these small problems, this is of course much faster than all the other solvers here.

b. In Figure 4, the convergence of the Jacobi method is shown, with $\omega = 1.0, 0.8, 0.5$, and of the G-S method with $\omega = 1.0, 1.3, 1.6$. For $\omega = 1.0$, the G-S method is twice as fast as Jacobi, as expected. The under-relaxed Jacobi is slower than the ordinary Jacobi, but that makes it a good smoother (see next section). The G-S method performs better with over relaxation, but when ω is close to 2, the method is becoming less stable. The value $\omega = 1.6$ gives good convergence without any oscillations, although a higher value gives a slightly faster convergence. With a red-black ordering Gauss-Seidel (not shown here), ω could be closer to 2 with good result.

c. To compare the solutions, we need to compute the flux. The function `fluxeval` does this using second-order one-sided difference approximations. With the lines

```
[A,f]=assemble(24,[1,7,14,16]);
u=gs(A,f,zeros(size(f)),1500,1.6);
flux=fluxeval(u)
```

we get exactly the same values as given in the example.

3) In the multigrid routine, injection is used as restriction operator. This simply means that some of the elements from the fine grid are extracted:

$$\boldsymbol{\phi}_{i,j}^{2h} = \boldsymbol{\phi}_{2i,2j}^{h}, \qquad 0 \le i, j \le N/2 \tag{30}$$

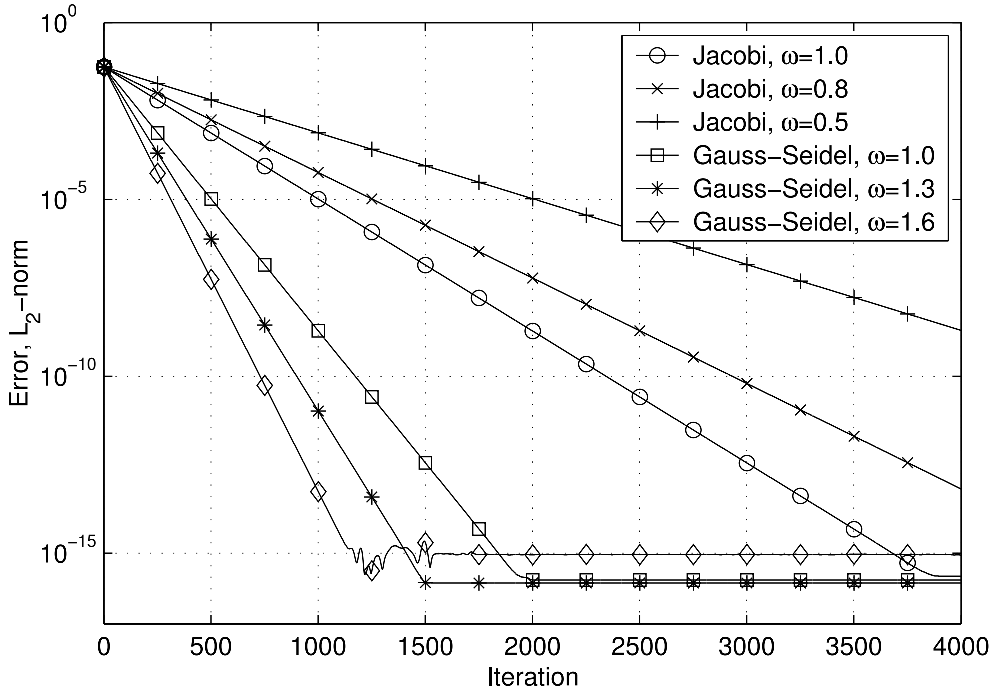


Figure 4: The convergence of the Jacobi method and the Gauss-Seidel method for different values of the relaxation parameter ω.

This is easy to do in MATLAB® using the colon operator, see the `restrict` function. For the prolongation, interpolation is used. This means that all the even elements on the fine grid are set to the values on the coarse grid, and the other values are interpolated:

$$\phi^h_{2i,2j} = \phi^{2h}_{i,j}, \qquad 0 \le i,j \le N/2 \tag{31}$$

$$\phi^h_{2i+1,2j} = \frac{1}{2}(\phi^{2h}_{i,j} + \phi^{2h}_{i+1,j}) \tag{32}$$

$$\phi^h_{2i,2j+1} = \frac{1}{2}(\phi^{2h}_{i,j} + \phi^{2h}_{i,j+1}) \tag{33}$$

$$\phi^h_{2i+1,2j+1} = \frac{1}{4}(\phi^{2h}_{i,j} + \phi^{2h}_{i,j+1} + \phi^{2h}_{i+1,j} + \phi^{2h}_{i+1,j+1}) \tag{34}$$

This is also relatively easy to do in MATLAB®, see the function `prolongate`.

The actual multigrid code for two grids is implemented in `mg`. Note that two different methods are available at the coarse grid, either solving exactly using \, or making ν_3 iterations.

a. In Figure 5, the convergence of the multigrid routine with Jacobi smoother and relaxation parameters 1/2 and 4/5 is compared with the convergence of pure Jacobi and Gauss-Seidel. The parameters $\nu_1 = \nu_2 = 2$, and two methods are used at the coarse level: the exact solver, and making $\nu_3 = 4$ Jacobi iterations.

   The multigrid solver is much better that the pure iterative methods, and $\omega = 0.8$ is better that $\omega = 0.5$. This is clear from the discussion in the lecture notes, since $\omega = 0.8$ gives a lower maximum multiplication factor for the high frequencies. We also see that the multigrid solvers with Jacobi at the coarse level converge slower than the solvers

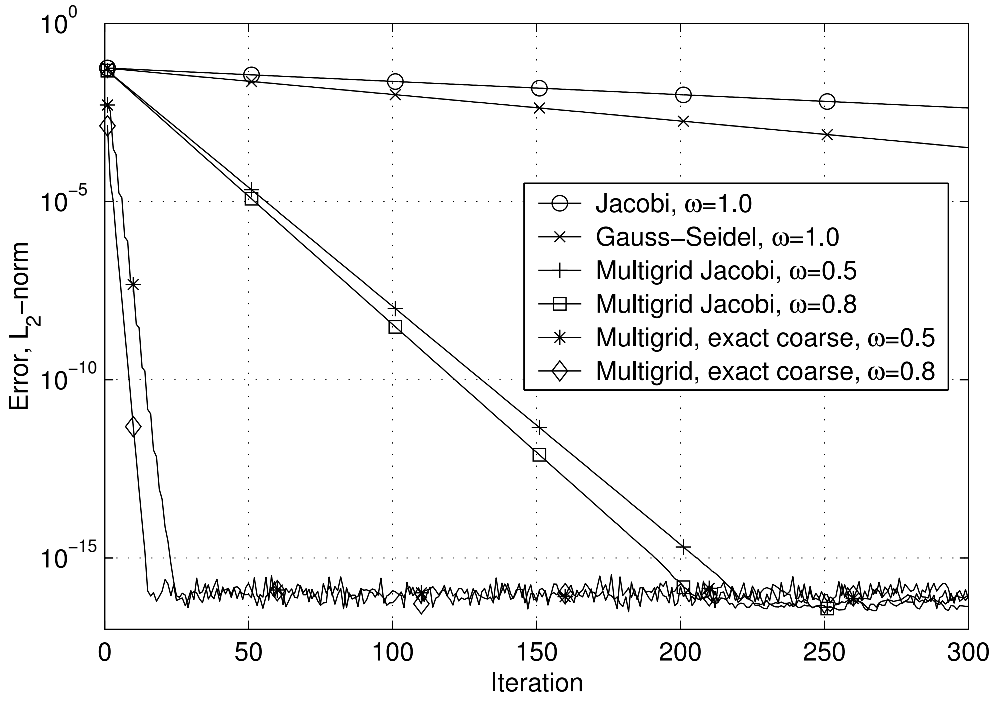


Figure 5: The convergence of the Jacobi method, the Gauss-Seidel method, and the multigrid solver.

with exact solution on the coarse level. This shows that it is a good idea to solve the coarser problem more accurately, since the time spent at the finest level is dominating (at least for real problems).

b. Figure 6 shows the convergence of multigrid with $\nu_1 = \nu_2 = 1$, $\nu_3 = 2$ and with $\nu_1 = \nu_2 = 2$, $\nu_3 = 4$. The second is much faster, showing again that the coarse level solution must be accurate, plus the fact that $\nu_1 = \nu_2 = 1$ probably does not decrease the high frequency error sufficiently.

4) There are $\binom{16}{4} = 1820$ different configurations of the four blocks, and the code below solves the equation for all of these and computes the fluxes, using the multigrid solver with $\omega = 0.8$. The maximum absolute difference from the given fluxes are computed and stored.

```
flux0=[
    0.0000 0.0000 0.0000 0.0000
    0.0075 0.0051 0.0076 0.0099
 % ...
    0.0099 0.0076 0.0051 0.0075
    0.0000 0.0000 0.0000 0.0000
    ];

configs=nchoosek(1:16,4);
nconfigs=size(configs,1);
err=zeros(1,nconfigs);
for ii=1:nconfigs
  u=mg(24,configs(ii,:),20,0.8,1);
```

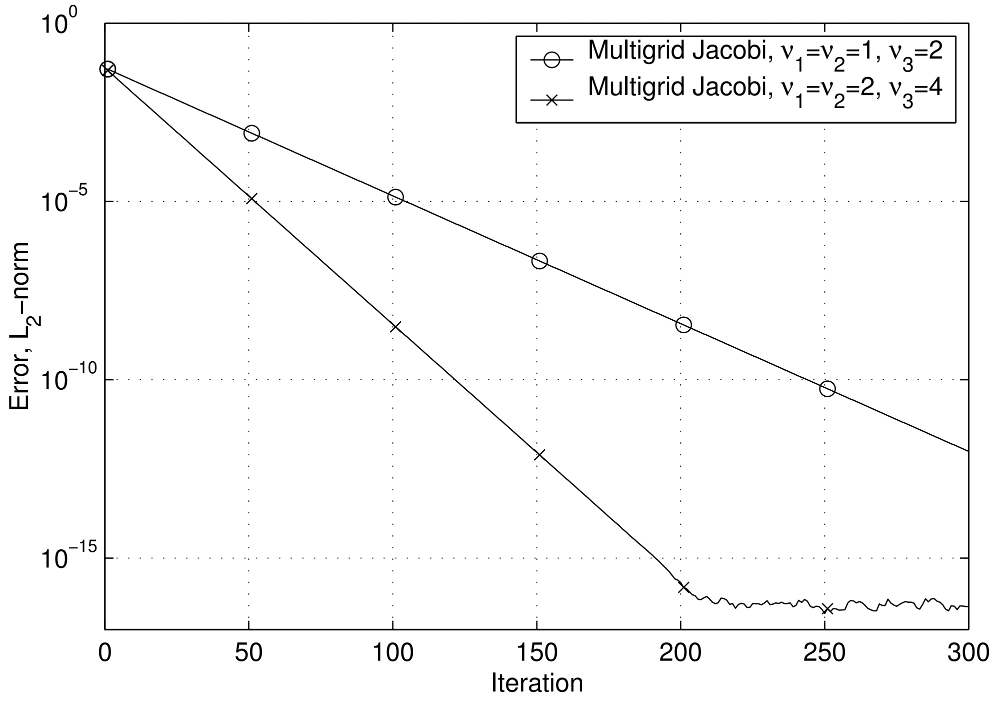


Figure 6: The convergence of the multigrid solver with different numbers of Jacobi iterations.

```
  flux=fluxeval(u);
  err(ii)=max(abs(flux(:)-flux0(:)));
end
```

In Figure 7, the errors for all configurations are shown. It is clear that one configuration has an error close to zero, and this one can be found by

```
[themin,pos]=min(err)
configs(pos,:)
```

which gives the correct configuration $(3, 5, 8, 15)$, which is illustrated in Figure 8. The error is $4.9 \cdot 10^{-5}$, that is, the rounding error in the values provided.

5) The generalized V-cycle multigrid routine is implemented in the function `mgv`. A recursive technique is used, very much in analogy with the pseudo code in the lecture notes. The subfunction `VGh` performs on V-cycle. At the coarsest grid, $\nu_1 + \nu_2$ Jacobi iterations are performed.

Figure 9 shows the convergence for some levels of refinement. To be able to use 5 levels, a grid of size $6 \cdot 2^4$ must be used, that is, 96×96. The convergence is faster the more levels we use, as expected. We know from before that the two-grid multigrid is much faster than Jacobi and Gauss-Seidel, so the higher levels of multigrid are of course much better than those.

In Figure 10, a study similar to the one in the lecture notes is shown. For the same *coarse* grid, different numbers of levels are used. This plot shows that you can have as fine mesh as you want without having to do more iterations, as long as you keep the same coarse level.

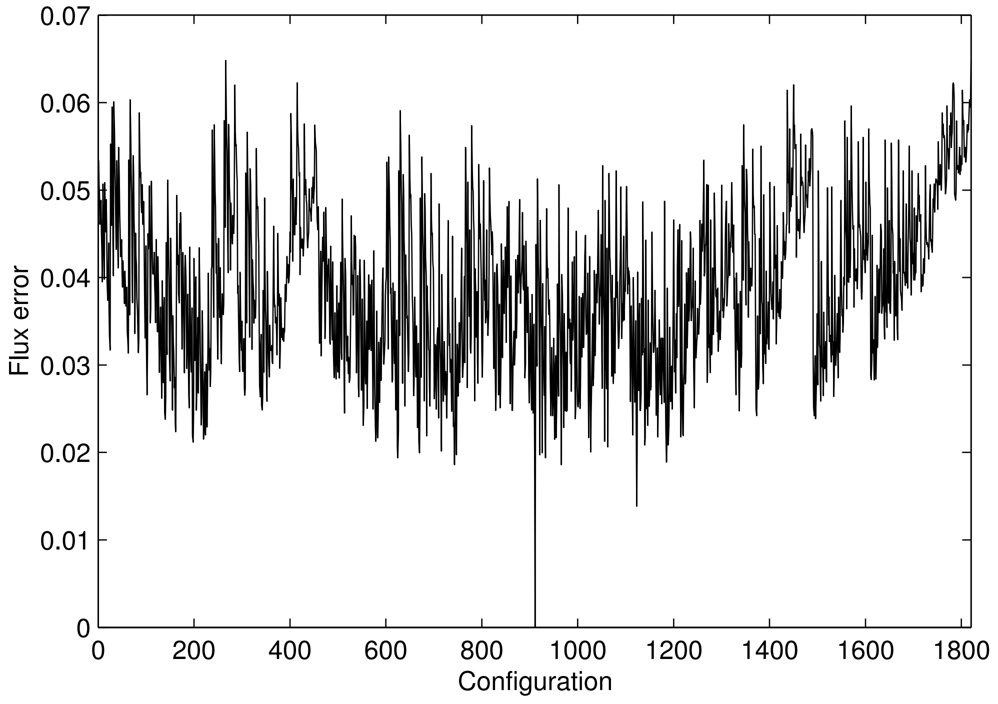


Figure 7: The error in the fluxes for all possible configurations of the source blocks. Configuration number 911, corresponding to source positions $(3, 5, 8, 15)$, gives essentially no error.

Another convergence study can be made where the problem is solved exactly on the coarsest mesh. This is often done in practice, since it is cheap to do and gives good convergence. We can see the following:

- The convergence is about the same for all grid sizes. This is consistent with the previous result, namely that the convergence is independent of the grid size, as long as the coarse grid is the same.
- For all problems, we get full machine precision in about 15 iterations. This is significantly better than before, when we just made four Jacobi iterations at the coarse mesh.

Of course, one does not have to implement a sparse Gaussian elimination routine, but instead just make a lot of Jacobi iterations at the innermost level. This should be cheap since the mesh is very coarse.

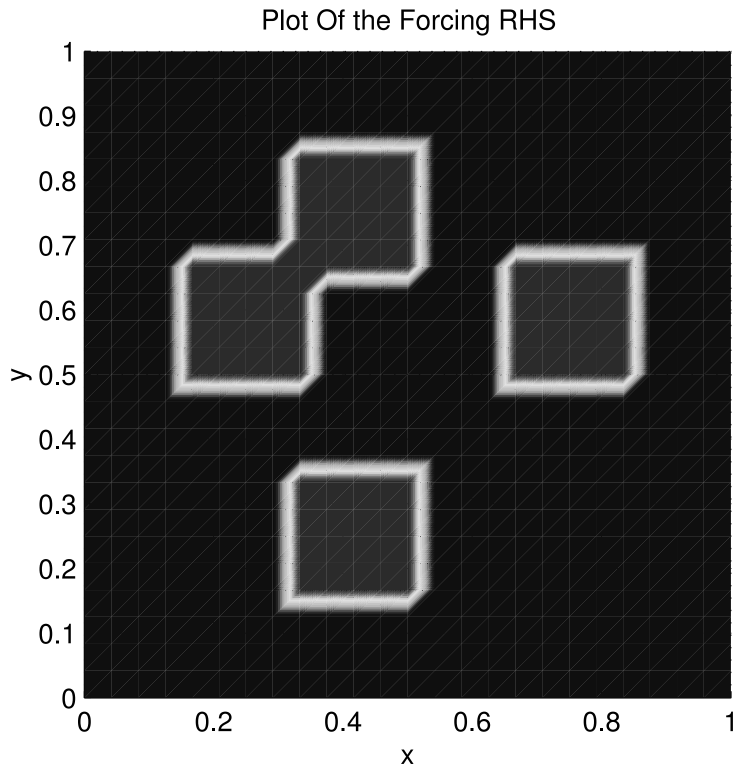


Figure 8: The configuration of the sources for the given fluxes, $(3, 5, 8, 15)$.

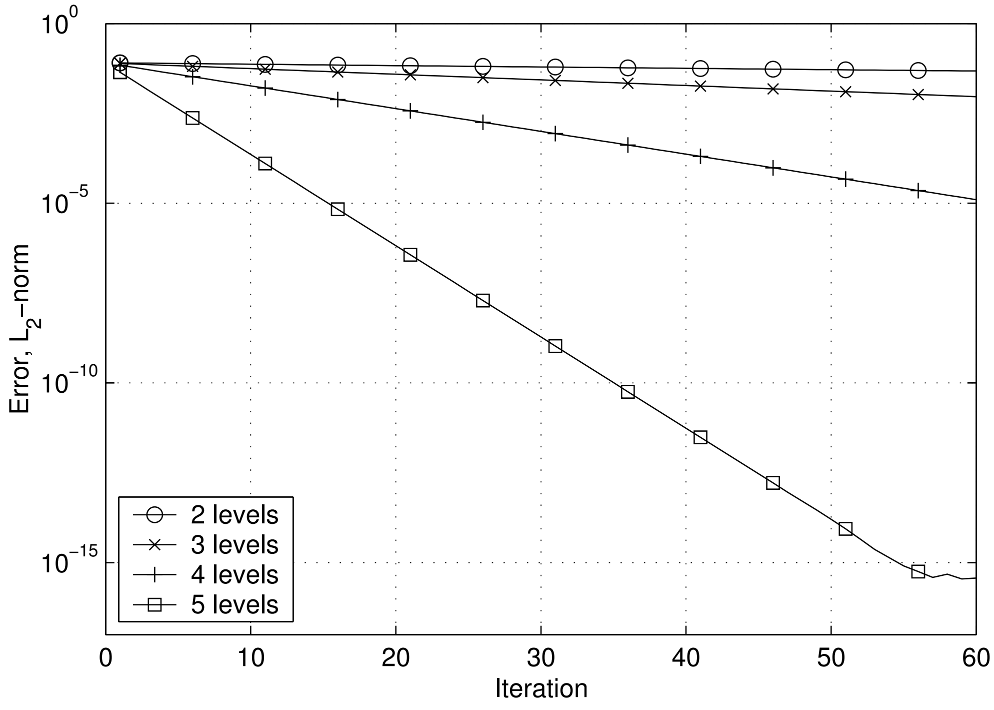


Figure 9: The convergence of the generalized V-cycle multigrid routine, for different numbers of refinements.

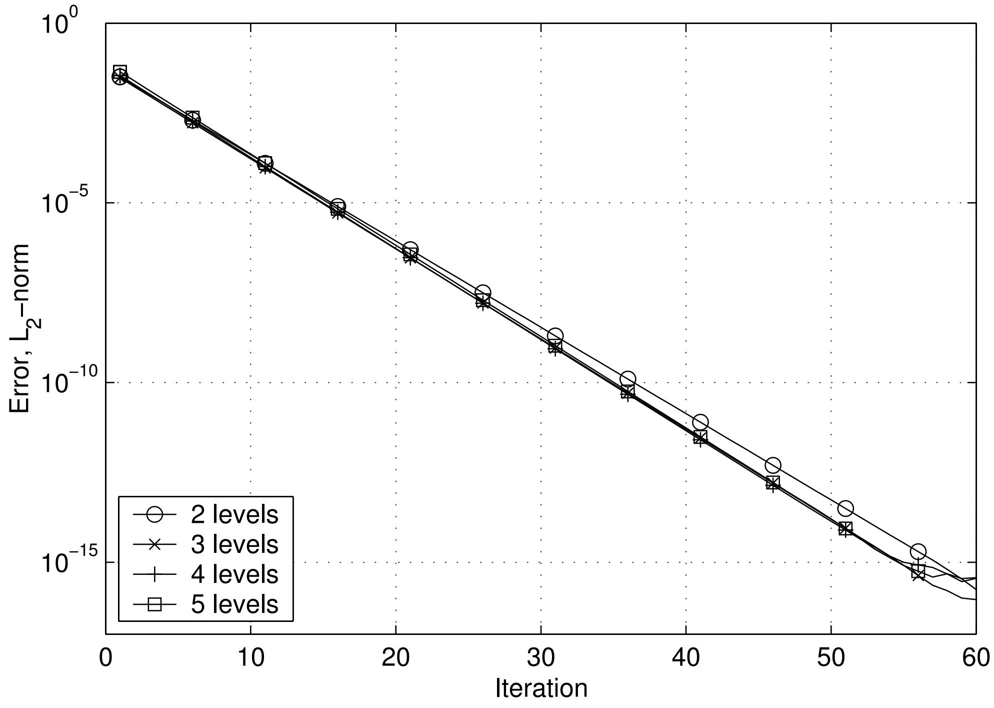


Figure 10: The convergence of the generalized V-cycle multigrid routine, for different numbers of refinements but with the *same coarse mesh*.

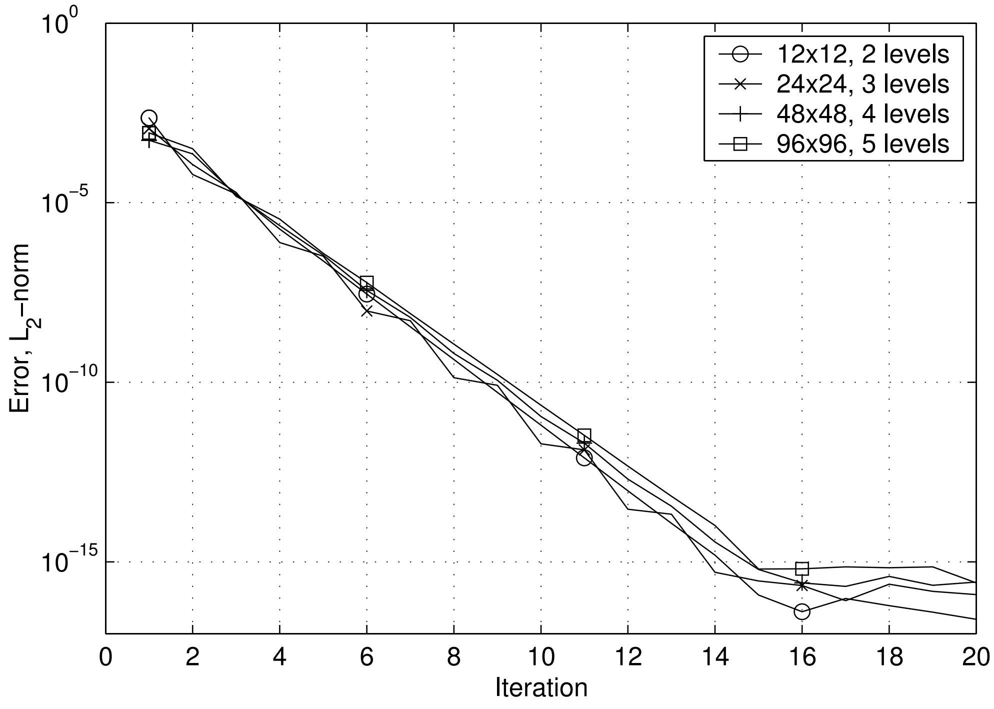


Figure 11: The convergence of the generalized V-cycle multigrid routine, for different numbers of refinements. On the coarsest mesh, the equation is solved exactly (using direct Gaussian elimination).

Appendix A: Code for Part #1

channelflow.m

```
function [Q,x,y,u]=channelflow(l,b,h,N)

% Parameters
a=sqrt(1/4*(l-b)^2-h^2);
dx=1/(N-1);

% Grid
[xi,eta]=ndgrid(0:dx:1,0:dx:1);
x=(b/2+a*eta).*xi;
y=h*eta;

% Coefficients in transformed equation
J=h*(b/2+a*eta);
a1=a^2*xi.^2+h^2;
b1=a*xi.*(b/2+a*eta);
c1=(b/2+a*eta).^2;
e1=2*a^2*xi;

% Initialize matrix and right hand side
A=sparse(N^2,N^2);
F=zeros(N^2,1);

% Mapping between (i,j) and position in solution vector
map=zeros(N,N);
map(:)=1:N^2;

% Molecules for du/dx, d2u/dx2, d2u/dy2, and d2u/dxdy
mol_xi=[0,-1,0;0,0,0;0,1,0]/2/dx;
mol_xixi=[0,1,0;0,-2,0;0,1,0]/dx^2;
mol_etaeta=mol_xixi';
mol_xieta=[1,0,-1;0,0,0;-1,0,1]/4/dx^2;

% Interior points
for i=2:N-1
  for j=2:N-1
    % Form stencil for full equation
    mol=-1/J(i,j)^2*(a1(i,j)*mol_xixi-2*b1(i,j)*mol_xieta+ ...
                     c1(i,j)*mol_etaeta+e1(i,j)*mol_xi);
    % Insert in A
    for i1=-1:1
      for j1=-1:1
        A( map(i,j), map(i+i1,j+j1) ) = mol(i1+2,j1+2);
      end
    end

    % RHS is 1
    F( map(i,j) ) = 1;
  end
end

% Bottom boundary (u=0)
for i=1:N
  A( map(i,1), map(i,1) ) = 1;
  F( map(i,1) ) = 0;
```

```
end

% Right boundary (u=0)
for j=1:N
  A( map(N,j), map(N,j) ) = 1;
  F( map(N,j) ) = 0;
end

% Top boundary (du/dn=0)
for i=2:N-1
  t1=-a/(a+b/2)*xi(i,N);
  t2=1;
  A( map(i,N), map(i+1,N) ) = t1/(2*dx);
  A( map(i,N), map(i-1,N) ) = -t1/(2*dx);
  A( map(i,N), map(i,N) ) = 3*t2/(2*dx);
  A( map(i,N), map(i,N-1) ) = -4*t2/(2*dx);
  A( map(i,N), map(i,N-2) ) = t2/(2*dx);
  F( map(i,N) ) = 0;
end

% Left boundary (du/dn=0)
for j=2:N
  A( map(1,j), map(1,j) ) = -3/(2*dx);
  A( map(1,j), map(2,j) ) = 4/(2*dx);
  A( map(1,j), map(3,j) ) = -1/(2*dx);
  F( map(1,j) ) = 0;
end

% Solve
U=A\F;
u=reshape(U,N,N);

% Compute Q
I=u.*abs(J);
mid=(I(1:N-1,1:N-1)+I(2:N,1:N-1)+I(1:N-1,2:N)+I(2:N,2:N))/4;
Q=sum(mid(:))*dx^2;
```

Appendix B: Code for Part #2

assemble.m

```
function [A,f]=assemble(gridsize,B)

h=1/gridsize;
n=gridsize-1;

f=zeros(n,n);
for ii=1:4
  row=floor((B(ii)-1)/4)+1;
  col=mod(B(ii)-1,4)+1;
  del=gridsize/6;
  f(del*col:del*(col+1),del*row:del*(row+1))=1;
end
f=f(:);

A1=spdiags([-ones(n,1) 2*ones(n,1) -ones(n,1)],-1:1,n,n)/h^2;
A=kron(A1,speye(n))+kron(speye(n),A1);
```

jac.m

```
function [u,err]=jac(A,f,u,niter,omega)

if nargout>=2
  uexact=A\f;
end

n=sqrt(length(f));
D=diag(diag(A));
LU=-(tril(A,-1)+triu(A,1));

err=zeros(1,niter);
for ii=1:niter
  u=omega*(D\(LU*u+f))+(1-omega)*u;
  if nargout>=2
    err(ii)=norm(u-uexact);
  end
end
```

gs.m

```
function [u,err]=gs(A,f,u,niter,omega)

if nargout>=2
  uexact=A\f;
end

n=sqrt(length(f));
DL=diag(diag(A))+tril(A,-1);
U=-triu(A,1);

err=zeros(1,niter);
for ii=1:niter
  u=omega*(DL\(U*u+f))+(1-omega)*u;
  if nargout>=2
    err(ii)=norm(u-uexact);
  end
end
```

fluxeval.m

```
function flux=fluxeval(u)

n=sqrt(size(u,1));
h=1/(n+1);
u0=zeros(n+2,n+2);
u0(2:n+1,2:n+1)=reshape(u,n,n);

bottom=(-3*u0(1,:)+4*u0(2,:)-u0(3,:))'/2/h;
top=(-3*u0(end,:)+4*u0(end-1,:)-u0(end-2,:))'/2/h;
left=(-3*u0(:,1)+4*u0(:,2)-u0(:,3))/2/h;
right=(-3*u0(:,end)+4*u0(:,end-1)-u0(:,end-2))/2/h;

flux=[left right bottom top];
```

restrict.m

```
function w2h=restrict(wh)

n=sqrt(size(wh,1));
wh=reshape(wh,n,n);
w2h=wh(2:2:end,2:2:end);
w2h=w2h(:);
```

prolongate.m

```
function wh=prolongate(w2h)

n=sqrt(size(w2h,1));
w2h0=zeros(n+2,n+2);
w2h0(2:n+1,2:n+1)=reshape(w2h,n,n);

n=2*n+3;
wh=zeros(n,n);
wh(1:2:n,1:2:n)=w2h0;
wh(2:2:n-1,1:2:n)=(wh(1:2:n-2,1:2:n)+wh(3:2:n,1:2:n))/2;
wh(1:2:n,2:2:n-1)=(wh(1:2:n,1:2:n-2)+wh(1:2:n,3:2:n))/2;
wh(2:2:n-1,2:2:n-1)=(wh(2:2:n-1,1:2:n-2)+wh(2:2:n-1,3:2:n))/2;
wh=wh(2:n-1,2:n-1);
wh=wh(:);
```

mg.m

```
function [u,err]=mg(gridsize,B,niter,omega,backslash)

nu1=2;
nu2=2;
nu3=4;

n=gridsize-1;
uh=zeros(n^2,1);
[Ah,fh]=assemble(gridsize,B);
A2h=assemble(gridsize/2,B);
if nargout>=2
  uexact=Ah\fh;
end
for ii=1:niter
  uh=jac(Ah,fh,uh,nu1,omega);
  rh=fh-Ah*uh;
  r2h=restrict(rh);
  if backslash
    e2h=A2h\r2h;
  else
    e2h=jac(A2h,r2h,zeros(size(r2h)),nu3,1.0);
  end
  eh=prolongate(e2h);
  uh=uh+eh;
  uh=jac(Ah,fh,uh,nu2,omega);
  if nargout>=2
    err(ii)=norm(uh-uexact);
  end
end
u=uh;
```

mgv.m

```
function [u,err]=mgv(gridsize,B,niter,omega,refinements)

nu1=2;
nu2=2;

finegrid=gridsize/2^refinements;

n=gridsize-1;
uh=zeros(n^2,1);
[Ah,fh]=assemble(gridsize,B);
if nargout>=2
  uexact=Ah\fh;
end
for ii=1:niter
  uh=VGh(uh,fh,B,finegrid,nu1,nu2,omega);
  if nargout>=2
    err(ii)=norm(uh-uexact);
  end
end
u=uh;

function uh=VGh(uh,fh,B,finegrid,nu1,nu2,omega)

n=sqrt(length(uh))+1;
Ah=assemble(n,B);
uh=jac(Ah,fh,uh,nu1,omega);
if n>finegrid
  r2h=restrict(fh-Ah*uh);
  e2h=VGh(zeros(size(r2h)),r2h,B,finegrid,nu1,nu2,omega);
  uh=uh+prolongate(e2h);
end
uh=jac(Ah,fh,uh,nu2,omega);
```