Math 226B Project 2

Theodore Kwan
tmkwan@uci.edu

March 11, 2016

Contents

1 Project Description

This project analyzes different iterative methods used in approximating the solution to the Poisson equation via the finite difference and finite element methods.

2 Finite Difference Method

Consider the Poisson equation on the unit square $\Omega = (0, 1) \times (0, 1)$:

$$\begin{cases} -\Delta u = 1, & (x, y) \in \Omega \\ u(x, y) = 0, & (x, y) \in \partial\Omega \end{cases} \tag{1}$$

The exact solution to (1) is given by:

$$\begin{aligned} u(x, t) = {} & \frac{x(1-x)}{2} \\ & + \frac{8}{\pi^3} \sum_{k=1}^{\infty} \left[\frac{\sinh[(2k-1)\pi(1-y)] + \sinh[(2k-1)\pi y]}{\sinh[(2k-1)\pi]} \right] \frac{\sin[(2k-1)\pi x]}{(2k-1)^3} \end{aligned}$$

2.1 Smoothers

This section shows the different smoothing algorithms, and tests them against the mesh size.

2.1.1 Weighted Jacobi

First, the weighted Jacobi method was implemented for the Poisson equation on a uniform mesh. The Matlab implementation is given by:

Listing 1: Weighted Jacobi

```
%% Weighted Jacobi
%
%   Written by Ted Kwan for Math 226B
%
%   This function implements the weighted
%   Jacobi method.
function [u] = wJac(u,f,omega)
    [n,m]=size(u);i=2:m-1;j=2:n-1; uJ=u;
    %%% Normal iteration
    uJ(i,j) =(f(i,j)+u(i-1,j)+u(i+1,j) ...
            +u(i,j-1)+u(i,j+1))/4;
    %%% Weighted correction
    u=omega*u+(1-omega)*uJ;
end
```

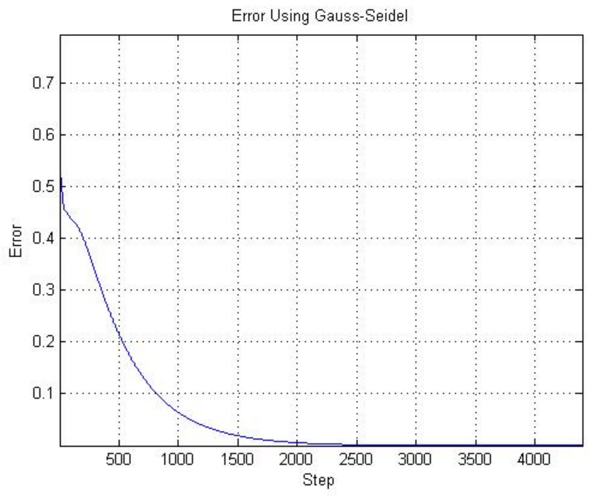
2.1.2 Gauss-Seidel

Next, the Gauss-Seidel method was implemented for the Poisson equation on a uniform mesh. The Matlab implementation is given by:

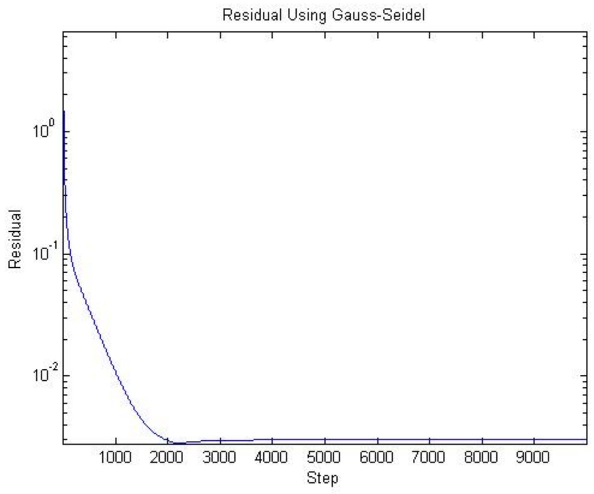
Listing 2: Gauss-Seidel

```
%% Gauss-Sidel
%
%   Written by Ted Kwan for Math 226B
%
%   This function implements the Gauss-Seidel
%   method using the code in the FDMcode
%   document from the class notes by
%   Professor Chen.
function [u] = GaussSid(u,f)
    [n,m] = size(u);
    for j = 2:n-1
        for i = 2:m-1
            u(i,j)=(f(i,j)+u(i-1,j)+u(i+1,j)...
            +u(i,j-1)+u(i,j+1))/4;
        end
    end
end
```

To measure the error, we can plot the difference between the approximation from Gauss-Seidel against the solution obtained by mldivide. The plot for this is:



The residual was not used as the stopping criterion, since it does not go below the discretization error. This could be due to the fact that the exact solution has a maximum value of 0.07. To see this, the residual was plotted against the number of iterations:



2.1.3 Comparing iterations

Letting:

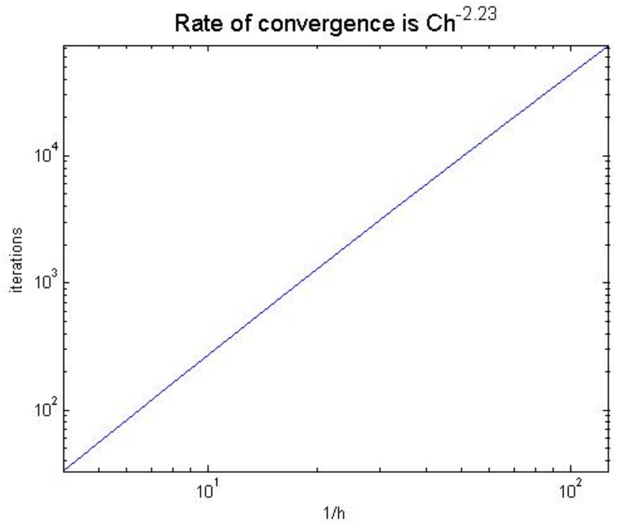
$$h = \frac{1}{2^n}, \quad n \in \{2, \ldots, 7\} \tag{2}$$

We can plot the amount of iterations needed to drive the relative error below the discretization error h^2. There was one issue with the error tracking used. When the residual:

$$\|f - Au\|$$

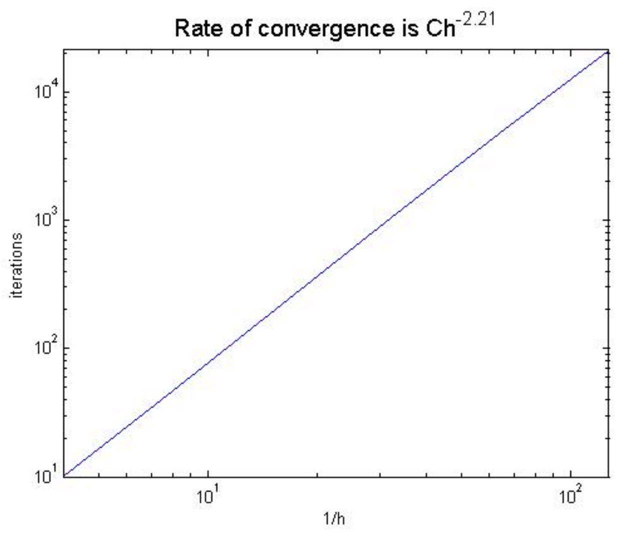
is used as the stopping criterion, the multi-grid methods do not terminate, and instead, they stagnate. This could be due to the fact that the exact solution has an extremely small maximum value. Regardless, the methods still converge to the exact solution within the error tolerance measured by the difference between the exact solution and the finite difference approximation. For this reason, the stopping criterion of comparing the approximated solution to the solution obtained by mldivide was used to determine the number of iterations needed.

For the Weighted Jacobi method, the number of iterations needed is plotted below:



We can see that the iterations increase following a power law with the size of the matrix.

For the Gauss-Seidel method:

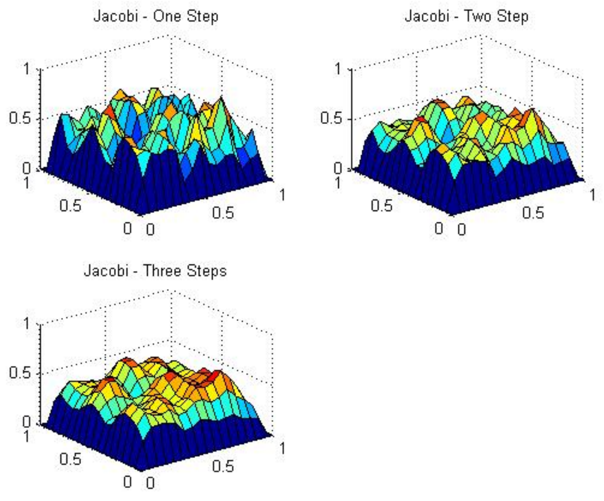


We can see that the iterations also increase following a power law with the size of the matrix.

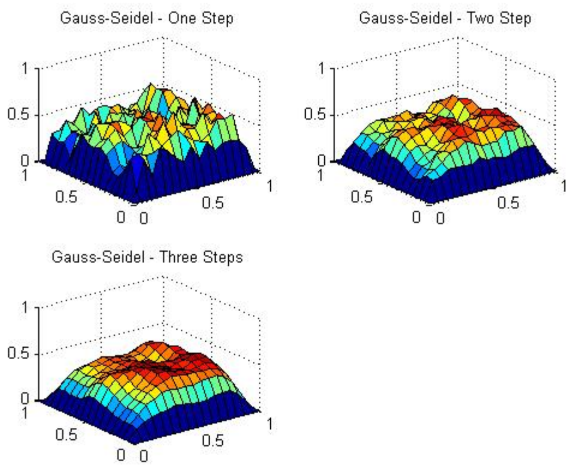
2.1.4 Error Smoothing

To see how the errors can be smoothed by the classical iterative methods, we can plot the error on the grid for the first 3 steps with a random initial guess.

For the weighted Jacobi method, the error on the grid for the first 3 steps is:



For the Gauss-Seidel method, the error on the grid for the first 3 steps is:



2.1.5 Red-Black Gauss-Seidel

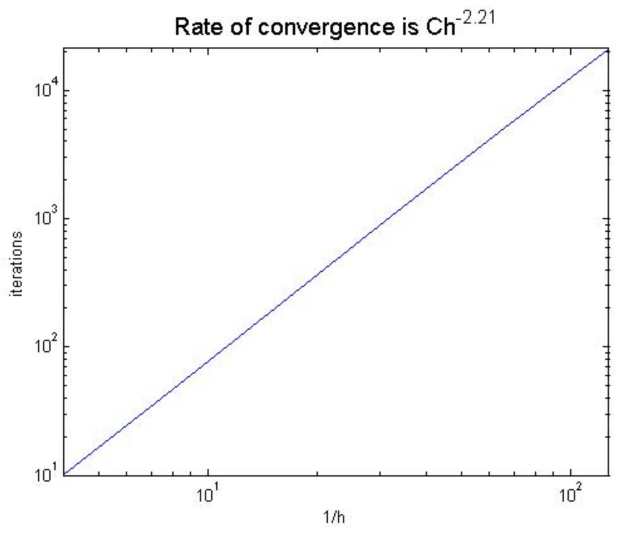
The red-black Gauss Seidel method can also be implemented. In Matlab, this is:

Listing 3: Gauss-Seidel Symmetric

```
%% Gauss-Sidel Red Black
%
%   Obtained from FDMcode and used for
%   Project 2
function [u] = GaussSidRB(u,f)
[n,m] = size(u);
% case 1 (red points): mod(i+j,2) == 0
i = 2:2:m-1; j = 2:2:n-1;
u(i,j) = (f(i,j) + u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1))/4;
i = 3:2:m-1; j = 3:2:n-1;
u(i,j) = (f(i,j) + u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1))/4;
% case 2 (black points): mod(i+j,2) == 1
i = 2:2:m-1; j = 3:2:n-1;
u(i,j) = (f(i,j) + u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1))/4;
i = 3:2:m-1; j = 2:2:n-1;
u(i,j) = (f(i,j) + u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1))/4;

end
```

The amount of iterations needed to drive the relative error below the discretization error is found to be:



2.2 Two-Grid Method

This section implements the Two-Grid method, then tests it against the same values of h that were used in 2.1.3.

2.2.1 Prolongation and Restriction

The index maps have been encoded into the prolongation and restriction maps, so that they do not need an extra parameter passed to them.

The matrix-free Prolongation map is given by the Matlab script given on the next page.

Listing 4: Prolongation

```
%% Prolongation
%
%   Written by Ted Kwan for Math 226B
%
%   This function implements the bilinear
%   prolongation map for the multi—grid
%   method.
function [uf] = Prol(uc,Nc)
    %%% Initial Setup
    %
    Nf=2*Nc-1; jc=[2:Nc-1];
    jf=[2.*jc-1]; uf=zeros(Nf,Nf);
    %%% Center
    %
    uf(jf,jf)=uc(jc,jc);
    %%% Sides
    %
    uf(jf-1,jf)=uf(jf-1,jf)+(0.5*uc(jc,jc));
    uf(jf,jf-1)=uf(jf,jf-1)+(0.5*uc(jc,jc));
    uf(jf+1,jf)=uf(jf+1,jf)+(0.5*uc(jc,jc));
    uf(jf,jf+1)=uf(jf,jf+1)+(0.5*uc(jc,jc));
    %%% Corners
    %
    uf(jf+1,jf+1)=(uf(jf+1,jf+1)+(0.25*uc(jc,jc)));
    uf(jf+1,jf-1)=(uf(jf+1,jf-1)+(0.25*uc(jc,jc)));
    uf(jf-1,jf+1)=(uf(jf-1,jf+1)+(0.25*uc(jc,jc)));
    uf(jf-1,jf-1)=(uf(jf-1,jf-1)+(0.25*uc(jc,jc)));
end
```

The matrix-free Restriction map is given by the Matlab script:

Listing 5: Restriction

```
%% Restriction
%
%   Written by Ted Kwan for Math 226B
%
%   This function implements the bilinear
%   restriction map for the multi—grid
%   method.
function [uc] = Res(uf,Nc)
    Nf=2*Nc-1; jc=[2:Nc-1];
    jf=[2.*jc-1]; uc=zeros(Nc,Nc);
    uc(jc,jc)=(4*uf(jf,jf)+2*(uf(jf-1,jf)+uf(jf,jf-1)+...
            uf(jf+1,jf)+uf(jf,jf+1))+uf(jf-1,jf+1)+...
            uf(jf+1,jf-1)+uf(jf-1,jf-1)+uf(jf+1,jf+1))./4;
end
```

2.2.2 Solution with Two-Grid

Using the methods above, the solution to (1) can be approximated by the finite difference method using the two-grid method to solve the resulting system.

2.2.3 Two-Grid Method

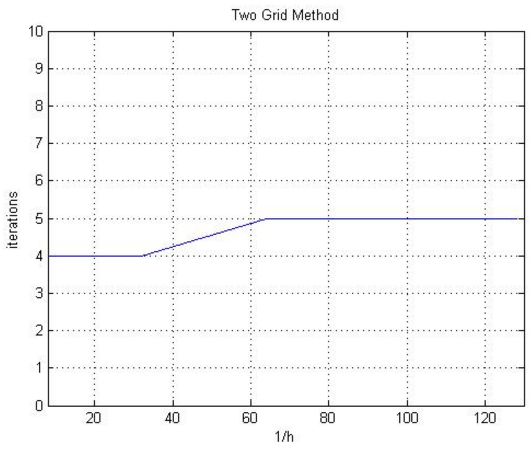
The two-grid method can be implemented in Matlab for the Poisson equation on a uniform grid. The implementation is given by:

Listing 6: Two-Grid Method

```
%% Two-Grid Method
%
%   Written by Ted Kwan for Math 226B
%
%   This function implements the two-grid
%   method for the Poisson equation on the
%   unit square with a uniform grid.
function [u] = TwoGridP(u,f,h,m,itr)
    %%% Setup Index maps
    %
    Nc=(1/(2*h))+1;
    %%% Pre-Smoothing
    %
    for i=1:m
        u=GaussSid(u,f);
    end
    %%% Solve at Coarse-Grid
    %
    Au=AuP(u); rc=Res(f-Au,Nc);
    ec=Psolve(2*h,rc);
    %%% Post Smoothing
    %
    u=u+Prol(ec,Nc);
    for i=1:m
        u=GaussSidB(u,f);
    end
end
```

2.2.4 Compare Iterations

The iterations required for the two-grid are lower than those needed for any of the classical iterative methods. For the different values of h used previously, the number of iterations required for convergence is plotted below:



2.3 Multi-Grid Method

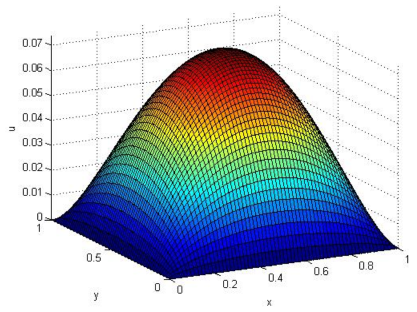
This section implements the Multi-Grid method, then tests it against the same values of h that were used in 2.1.3.

2.3.1 Prolongation and Restriction

The Prolongation and Restriction maps used for the two-grid method were the same as those used for the multi-grid method. The functions were simply re-used.

2.3.2 Solution with Multi-Grid

Using the method above, the solution to (1) can be approximated by the finite difference method using the multi-grid method to solve the resulting system. A plot of the solution is given below:



2.3.3 Multi-Grid Method

The multi-grid method can be implemented in Matlab for the Poisson equation on a uniform grid. The recursive implementation is given by:

Listing 7: Multi-Grid Method

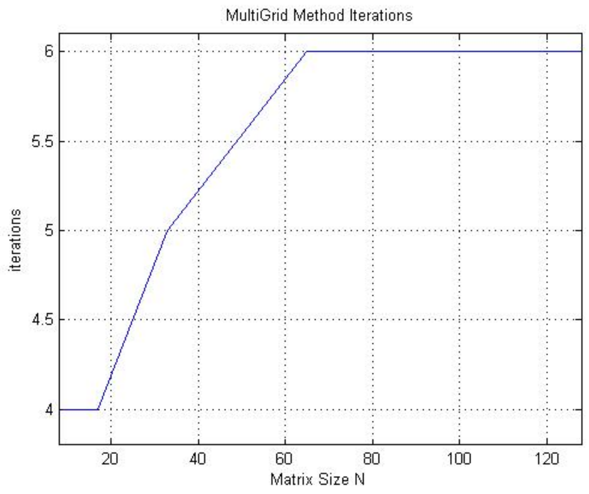
```
%% Recursive Multigrid V—Cycle
%
%   Written by Ted Kwan for Math 226B
%
%   This function implements the recursive
%   multi—grid method for the Poisson
%   equation on the unit square with a
%   uniform grid.
function [u,hc] = MultiGridP(u,r,h,m,itr)
    %%% Coarse Grid
    %   Solve directly.
    if(itr==1)
        hc=h;
        u=Psolve(h,r);
        return ;
    end
    %%% Setup index map
    %
    Nc=(1/(2*h))+1;
    %%% Pre—Smoothing
    %
    for i=1:m
        u=GaussSid(u,r);
    end
    %%% Restriction
    %
    rf=(r-AuP(u)); rc=Res(rf,Nc);
    %%% Coarse Grid Correction
    %
    zc=zeros(Nc,Nc);
    [ec,hc]=MultiGridP(zc,rc,2*h,m,itr-1);
    %%% Post Smoothing
    %
    u=u+Prol(ec,Nc);
    for i=1:m
        u=GaussSidB(u,r);
    end
end
```

2.3.4 Compare Iterations

The iterations required for the multi-grid are by far the lowest. The difference is so considerable, that the multi-grid method needed to be plotted on a linear scale instead of a log scale. Starting with a random initial guess and using the parameters:

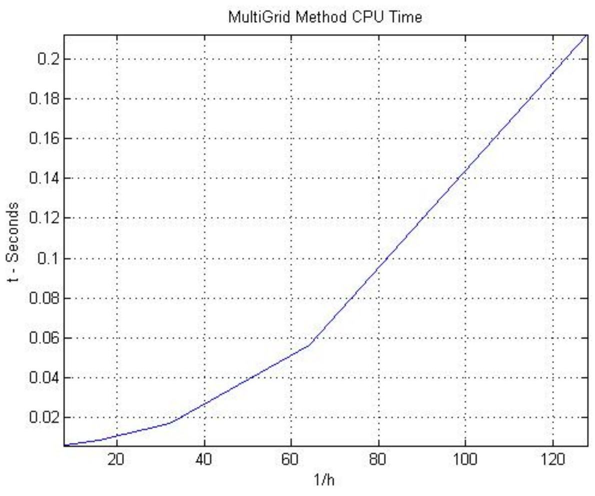
J	h on coarsest grid	Smoothing Steps	Tolerance
2	0.25	3	0.0011372
3	0.25	3	0.0002869
4	0.25	3	7.1889e-05
5	0.25	3	1.7983e-05
6	0.25	3	4.4963e-06

the number of iterations required for the values of h used previously is plotted below:



2.3.5 Check CPU Time

Varying h between $\frac{1}{4}$ and $\frac{1}{128}$, we see that the CPU time required to run the multi-grid method on (1) is:



So the multi-grid method gets slower as takes more steps on a finer grid, but it is still relatively quick (under 0.2 seconds).

3 Multi-Grid Finite Element Method

Consider the Poisson equation on the unit circle $\Omega = \{(x, y) \ : \ x^2 + y^2 < 1\}$:

$$\begin{cases} -\Delta u = 1, & (x, y) \in \Omega \\ u(x, y) = 0, & (x, y) \in \partial\Omega \end{cases} \tag{3}$$

From project 1, we know that the exact solution to (3) is given by:

$$u(x, y) = \frac{1}{4}\left[1 - x^2 - y^2\right] \tag{4}$$

3.1 Finite Element Method Setup

This section shows the modified finite element method function to handle assembling the stiffness matrix, the prolongation and restriction matrices as well as the smoothers.

3.1.1 Prolongation and Restriction Matrices

To construct the prolongation and restriction matrices, the code from the notes was modified to create the correct prolongation matrix. From this, we create the restriction matrix by taking the transpose. The matlab code for this is:

Listing 8: Prolongation Matrix

```matlab
%% Create Prolongation Matrix
%
%   Written by Ted Kwan for Math 226B
%
%   This function implements constructs the
%   prolongation matrix using a Hierarchical basis.
%   Written using notes from Professor Chen.
function [Pro] = ProHB(nCoarseNode,nTotal,HB)
    nFineNode=nTotal-nCoarseNode; %Number of fine nodes.
    coarseNode=[1:nCoarseNode]'; coarseNodeFineIdx=coarseNode;
    ii=[coarseNodeFineIdx; double(HB(:,1)); double(HB(:,1))];
    jj=[coarseNode; double(HB(:,2)); double(HB(:,3))];
    ss=[ones(nCoarseNode,1);0.5*ones(nFineNode, ...
        1);0.5*ones(nFineNode,1)];
    Pro=sparse(ii,jj,ss,nTotal,nCoarseNode);
end
```

3.1.2 Finite Element Method

This section contains the matlab code for creating the stiffness matrices at each level by the triple product, stores the smoothers at each level and sets up the system and solves it by the Multi-grid method. The Matlab code used is:

Listing 9: Finite Element Multi-grid Method

```
%% Finite Element Method - Multigrid method
%
%   Written by Ted Kwan for Math 226B
%
%   Returns u(x,t) to approximate the solution
%   to the Laplace equation -\Delta u=f
%
%   This function runs the finite element method
%   using the multi-grid method to solve the linear
%   system. The function can also run the preconditioned
%   conjugate gradient method by adding pcg as a 4th
%   argument.
function [u,node,elem,k,res]= FiniteElemMG(node,elem,f,J,varargin)
    %% Initial Setup
    %
    HB=cell(J,1); N=zeros(J+1,1);
    freeNodes=cell(J,1); BdNodes=cell(J,1); As=cell(J,1);
    %% Refine Grid and Save
    %
    for i=1:J
        N(i)=length(node(:,1));
        [~,~,isBdNode]=findboundary(elem);
        freeNodes{i}=find(~isBdNode);
        BdNodes{i}=find(isBdNode);
        [node,elem,bdFlag,HB{i}] = uniformrefine(node,elem);
    end
    N(J+1)=length(node(:,1));
    [bdNode,~,isBdNode]=findboundary(elem);
    freeNode=find(~isBdNode);
    [Asfull,area]=AssembleStiffnessFine(node,elem);
    As{J}=Asfull(freeNode,freeNode); clear Asfull;
    Pro=cell(J,1); Res=cell(J,1); Smoothers=cell(J+1,2);
    %% Prol, Res, Stiff And Smoother Matrices
    %
    for i=J:-1:2
        Protemp=ProHB(N(i),N(i+1),HB{i});
        if(i==J)
            Pro{i}=Protemp(freeNode,freeNodes{i});
        else
            Pro{i}=Protemp(freeNodes{i+1},freeNodes{i});
        end
        Res{i}=Pro{i}';
        As{i-1}=Res{i}*As{i}*Pro{i};
        Smoothers{i,1}=tril(As{i});
        Smoothers{i,2}=triu(As{i});
    end
    %% Calculate RHS
    %
    b=FiniteElemRHS(node,elem,f,area,N(J+1));
    %% Boundary Condition.
    %
    %%% Dirichlet Boundary Conditions.
    %
    g_D=varargin{1}; u = zeros(N(J+1),1); r = b(freeNode);
    u(bdNode) = g_D(node(bdNode,1),node(bdNode,2));
    %r(bdNode) = g_D(node(bdNode,1),node(bdNode,2));
    itrmax=varargin{2}; tol=varargin{3};
    %% Solve Au=b
    %
    if(length(varargin)>3 && strcmpi(varargin{4},'pcg'))
        %%% Preconditioned CG
        %
        [u(freeNode),k,res]=PCGvCycleAlg(r,u(freeNode),J,As,...
                                Res,Pro,Smoothers,tol,itrmax,1);
    else
        %%% Multi-Grid Method
        %
        res=zeros(itrmax,1); tol=tol*norm(r);
        err=2*tol; k=1;
        while(err>tol && k<itrmax)
        e=FEMvCycleAlg(r,J,As,Res,Pro,Smoothers,1);
        u(freeNode)=u(freeNode)+e;
        r=(r-As{J}*e); err=norm(r);
        res(k)=err; k=k+1;
        end
        res=res(find(res));
    end
end
```

3.2 Multi-Grid Method

This section shows the V-cycle method, and the results of using this method on the PDE (3).

3.2.1 V-Cycle

The non-recursive V-cycle was used in the finite element method. The code for this is:

Listing 10: Multi-Grid Method

```
%% Algebraic Multi—Grid V—Cycle
%
%   Written by Ted Kwan for Math 226B
%
%   This function implements a J—level v—cycle to be used alone
%   or as the preconditioner for the conjugate gradient method.
%   This function is used with the finite element method in
%   order to solve the system Au=b where A is the stiffness
%   matrix
function [e] = FEMvCycleAlg(r,J,As,Res,Pro,R,mu)
    ri=cell(J,1); ei=cell(J,1); ri{J}=r;
    for i=J:-1:2
        %%% Pre Smoothing
        %
        ei{i}=R{i,1}\(ri{i});
         for j=1:(mu-1) % Fixup because first step
                        % is outside of loop
             ei{i}=ei{i}+R{i,1}\(ri{i}-As{i}*ei{i});
         end
        %%% Update and Restrict
        %
        ri{i-1}=Res{i}*(ri{i}-As{i}*ei{i});
    end
    %%% Solve on Coarse Grid
    %
    ei{1}=As{1}\ri{1};
    for i=2:J
        %%% Prolongate and Correct
        %
        ei{i}=ei{i}+Pro{i}*ei{i-1};
        %%% Post Smoothing
        %
        for j=1:mu
            ei{i}=ei{i}+R{i,2}\(ri{i}-As{i}*ei{i});
        end
    end
    e=ei{J};
end
```

3.2.2 Preconditioned Conjugate Gradient

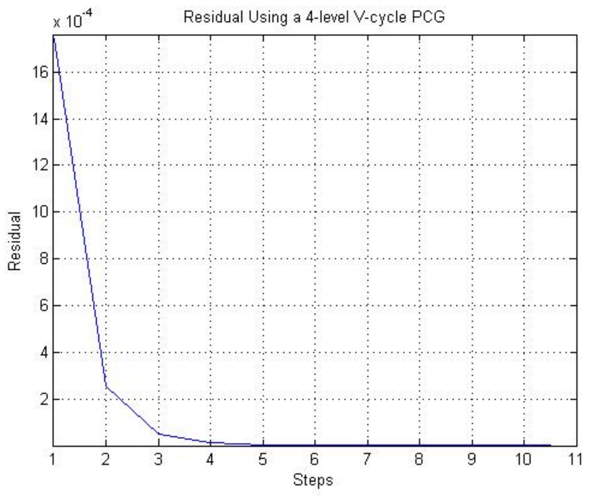
The multi-grid method was used as a preconditioner for the conjugate gradient method. The matlab code used to do this is:

Listing 11: Preconditioned Conjugate Gradient

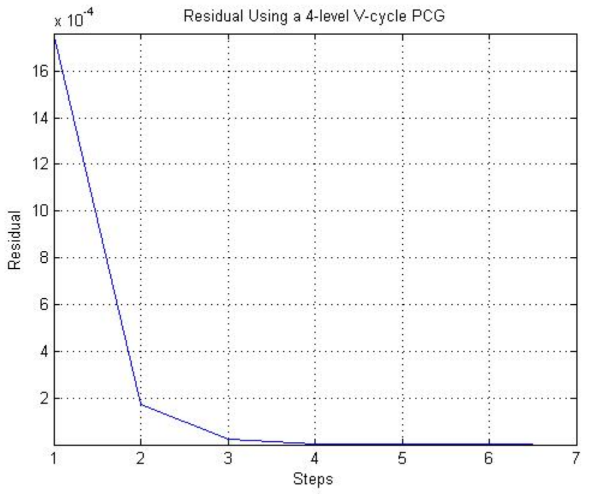
```
%% Finite Element Preconditioned Conjugate Gradient
%
%   Written by Ted Kwan for Math 226B
%
%   This function implements the preconditioned
%   conjugate gradient method using a J-level
%   v-cycle as the preconditioner. This method is
%   used with the finite element method in order to
%   solve the system Au=b where A is the stiffness matrix
function [u,k,res] = PCGvCycleAlg(b,u,J,As,Res,Pro,R,tol,maxitr,mu)
    %%% Initial Setup
    %
    nmb=norm(b);tol=tol*nmb; k=1; err=2*tol;
    r=b-As{J}*u; res=zeros(maxitr,1);
    %% Main Loop
    %
    while(err>tol && k<maxitr)
        Br=FEMvCycleAlg(r,J,As,Res,Pro,R,mu);
        rho=r'*Br;
        if(k==1)
            p=Br;
        else
            beta=rho/rho_old; p=Br+beta.*p;
        end
        Ap=As{J}*p; alpha=rho/(p'*Ap);
        u=u+alpha*p; r=r-alpha*Ap; rho_old=rho;
        k=k+1; err=norm(r);
        res(k)=err;
    end
    res=res(find(res));
end
```

3.2.3 Residual Decrease

When we set $J = 4$ for the multi-grid, the residual decreases exponentially:



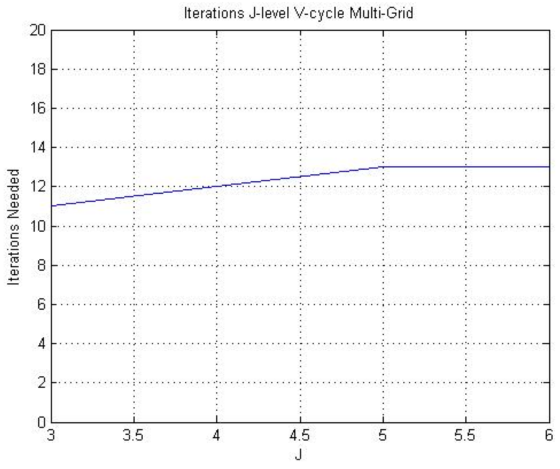
When we set $J = 4$ for the multi-grid preconditioned conjugate gradient method, the residual decreases at the rate shown in the graph:



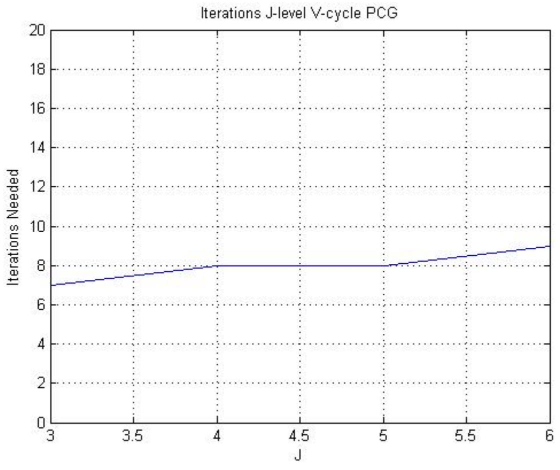
By the above graphs, the preconditioned conjugate gradient method is faster at diminishing the error.

3.2.4 Iterations Used

When we vary J from 3 to 6, the amount of iterations needed stays relatively uniform. The tolerance solved for this graph is 1×10^{-6}, with one smoothing step.



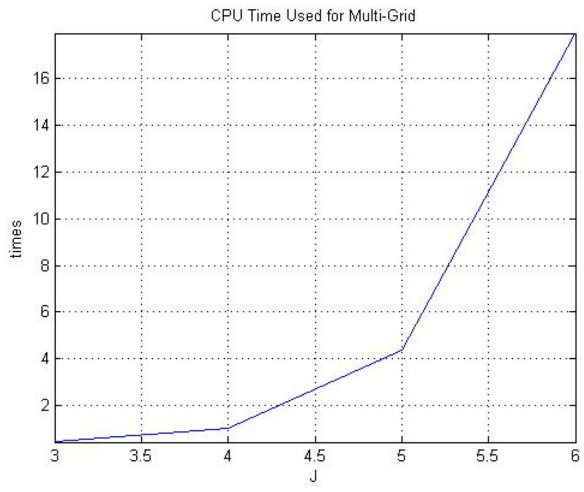
Using the multigrid preconditioned conjugate gradient method, we see that the number of iterations is lower. It takes about $\frac{1}{2}$ the iterations as the multi-grid method. This is shown in the image below:



3.2.5 CPU Time

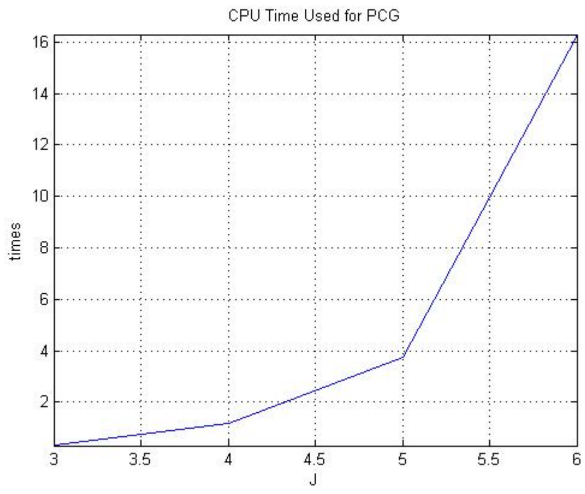
The time required to solve the system via the multi-grid method to a relative tolerance for of 1×10^{-6} plotted for both the V-cycle and the preconditioned conjugate gradient method.

For the V-cycle multigrid method, the time to solve each system is:



As we can see, the time increases to be somewhat slow (over 16 seconds). This is still relatively fast, and most likely faster than any of the classical iterative methods.

For the preconditioned conjugate gradient method, the time is slightly less than that of the multi-grid method. Solving in the finest grid requires only 16 seconds.



4 Algebraic Multi-Grid method

Consider the Poisson equation on a domain Ω which approximates the shape of Lake Superior:

$$\begin{cases} -\Delta u = 1, & (x,y) \in \Omega \\ u(x,y) = 0, & (x,y) \in \partial\Omega \end{cases} \tag{5}$$

Alternatively, consider the algeraic system:

$$A\mathbf{u} = \mathbf{b} \tag{6}$$

Where A is the stiffness matrix for Ω and $\mathbf{b}$ is a random vector.

4.1 Algebraic Multi-Grid Method Setup

This section shows the modified finite element method function which assembles the stiffness matrix, the prolongation and restriction matrices as well as stores the smoothers. The Matlab function for this is:

Listing 12: Finite Element Multi-grid Method

```
%% Finite Element Method - Multigrid method
%
%   Written by Ted Kwan for Math 226B
%
%   This method solves Ax=b using an algebraic
%   preconditioned conjugate gradient method with
%   a J-level v-cycle as the preconditioning matrix.
%   The finite element method can also be solved by
%   using additional inputs.
%
%%% Inputs
%
% * node - Nx2 Matrix of node coordinates.
% * elem - NTx3 Matrix of elements.
% * J - Number of steps for v-cycle.
% * itrmax - max number of iterations.
% * tol - tolerance for solving system.
%
function [u,node,elem,stop]= AlgebraicPCG(node,elem,J,varargin)
    %% Initial Setup
    %
    [~,~,isBdNode]=findboundary(elem);
    N=length(node(:,1)); bdNode=find(isBdNode);
    freeNode= find(~isBdNode);
    Pro=cell(J,1); Res=cell(J,1); A=cell(J,1);
    itrmax=varargin{1}; Smoothers=cell(J,2);
    %% Finest Stiffness Matrix
    %
    [Afull,~,area]=assemblematrix(node,elem);
    A{J}=Afull(freeNode,freeNode);
    %% Prolongation, Restriction, Smoother and Stiffness Matrices
    %
    for i=J:-1:2
        [isC,As] = coarsenAMGc(A{i});
        [Pro{i},Res{i}] = interpolationAMGs(As,isC);
        A{i-1}=Res{i}*A{i}*Pro{i};
        Smoothers{i,1}=tril(A{i});
        Smoothers{i,2}=triu(A{i});
    end
    u = zeros(N,1);
    %% Calculate RHS and Solve
    %
    if(length(varargin)>3)
        %% Finite Element Method
        %
        f=varargin{4};
        r=FiniteElemRHS(node,elem,f,area,N);
        tol=(varargin{2});
        %%% Dirichlet Boundary Condition.
        %
        g_D=varargin{3};
        u(bdNode) = g_D(node(bdNode,1),node(bdNode,2));
        b=r(freeNode);
        %%% Solve
        %
        [u(freeNode),stop,res]=PCGvCycleAlg(b,u(freeNode),J,A,...
                                Res,Pro,Smoothers,tol,itrmax,5);
    else
        %% Algebraic Multi-Grid
        %
        r=varargin{3}; b = r(freeNode);
        tol=(varargin{2});
        [u,stop,res]=PCGvCycleAlg(b,u(freeNode),J,A, ...
                                Res,Pro,Smoothers,tol,itrmax,5);
    end
end
```

4.2 Multi-Grid Method

This section shows the V-cycle method, and the results of using this method on the PDE (5), as well as the algebraic equation (6).

4.2.1 V-Cycle

A non-recursive V-cycle was used in the algebraic multi-grid. The code for this is the same as the code used in part 2.

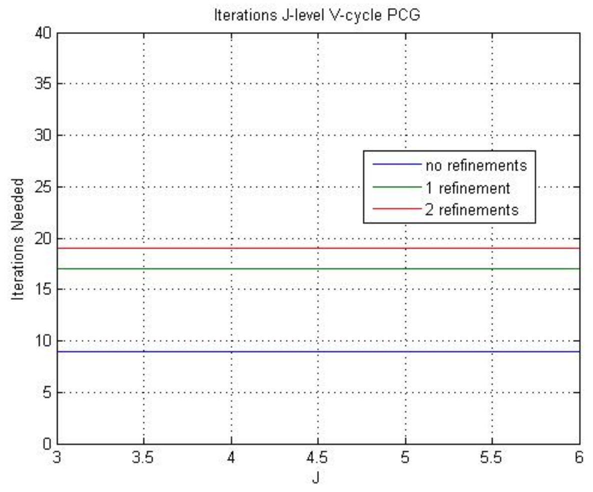
4.2.2 Preconditioned Conjugate Gradient

The multi-grid method was used as a preconditioner for the conjugate gradient method. The code to do this is the same as that used in part 2.

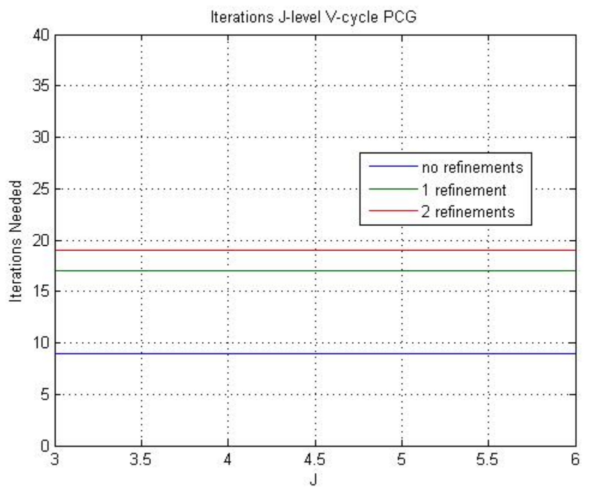
4.2.3 Iterations Used

For different finte meshes, when we vary J from 3 to 6, the amount of iterations needed increases slightly.

For the algebraic multi-grid preconditioned conjugate gradient method, the amount of iterations needed to solve below a relative tolerance of 1×10^{-6} is:



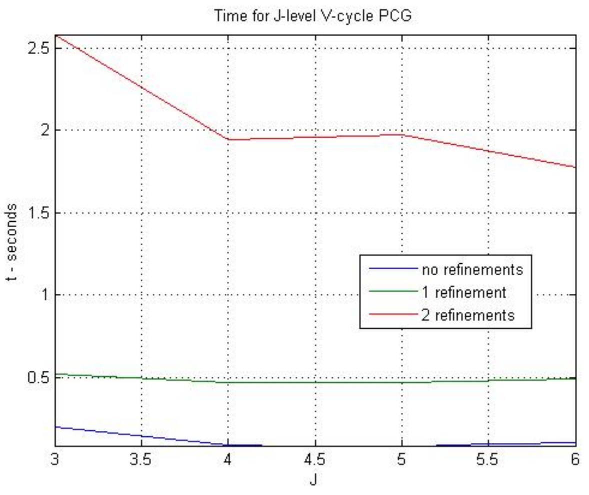
For the finite element multi-grid preconditioned conjugate gradient method, the amount of iterations needed to solve below a relative tolerance of 1×10^{-6} is:



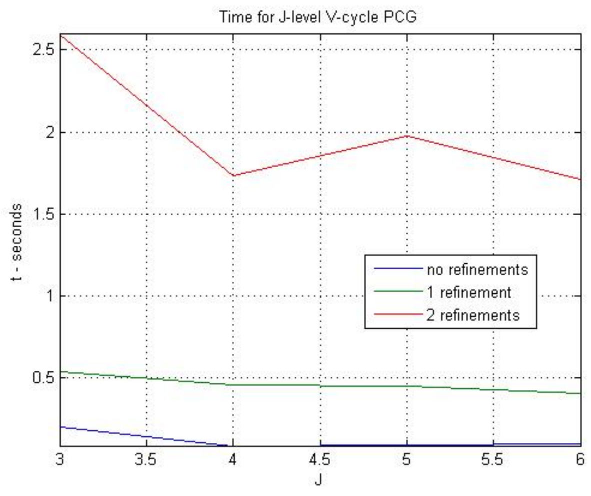
4.2.4 CPU Time

For the preconditioned conjugate gradient method, the time required decreases as we add more levels to the V-cycle. This could be due to the fact that the matrix which needs to be solved via mldivide decreases in size as J increases.

For the algebraic multi-grid preconditioned conjugate gradient method, the time needed is:

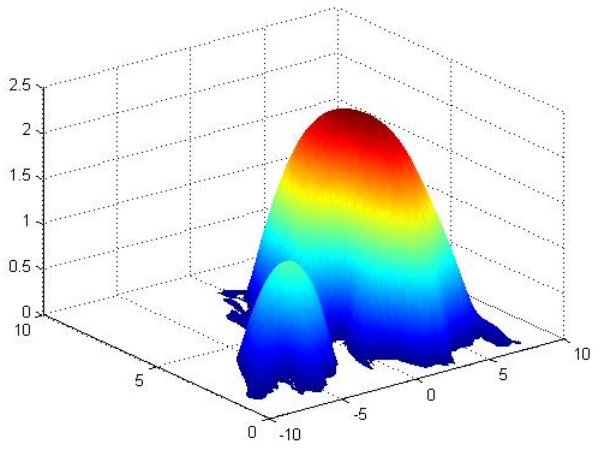


For the finite element multi-grid preconditioned conjugate gradient method, the time needed is:



4.3 Solution Plot

The solution to the PDE (5) is plotted on Lake Superior below:



5 Supporting Code

5.1 Finite Difference Method

For the Finite difference method, a few subroutines were needed in addition to the ones notes above.

5.1.1 Finite Difference Matlab Function

First, the finite difference method needed a Matlab function in order to run. The code for this is:

Listing 13: Finite Difference Method

```matlab
%% Poisson Equation Finite Difference
%
%   Written by Ted Kwan For Math 226A
%   Project 2
%   This solves the Poisson equation with Dirichlet
%   boundary conditions.
%
%   The matrix solving method can be chosen by inputting
%   any of the following:
%   * 'Gauss' - Gauss-Seidel.
%   * 'Gaussrb' - Gauss-Seidel Symmetric.
%   * 'Jacobi' - Weighted Jacobi Method.
%   * 'TwoGrid' - Two Grid Method.
%   * 'MultiGrid' - Multi-Grid method (must specify steps)
function [u,x,y,err,stop] = PoissonFinDifMG(h,f,g,type,itrmax ...
                                                    ,steps)
    %% Initial Setup
    %
    [x,y]=ndgrid(0:h:1,0:h:1); nx=length(x(1,:)); nxint=nx-2;
    ny=length(y(:,1)); nyint=ny-2;
    dx=h; dy=h; N=(ny)*(nx); u=zeros(N,1);
    isbd = true(ny,nx); isbd(2:end-1,2:end-1) = false;
    isfree=~isbd; bdidx = find(isbd(:));
    intidx=find(isfree(:)); u(bdidx)=g(x(bdidx),y(bdidx));
    %% Setup Residual
    %
    u=reshape(u,ny,nx); fu=(h.^2)*f(x,y);
    %% Initial guess for Gauss-Sidel
    %
    errgr=zeros(size(u)); err=zeros(itrmax,1);
    u(intidx)=rand(size(u(intidx)));
    icur=itrmax; uapx=Psolve(h,fu);
    tol=(h.^2)*max(abs(uapx(:)));
    errcur=2*tol; stop=1; omega=0.43;
    er=zeros(size(u));
    %% Solve System
    %
    %   Check iterative method used and solves
    %   the linear system.
    if(strcmpi(type,'gauss'))
        %%% Gauss-Seidel
        %
        while (errcur > tol && icur >stop)
            u = GaussSid(u,fu);
            errcur = max(abs(uapx(:)-u(:)));
            err(stop)=errcur; stop=stop+1;
        end
        err=err(find(err));
    elseif(strcmpi(type,'gaussrb'))
        %%% Gauss—Seidel Symmetric
        %
        while (errcur > tol && icur >stop)
            u = GaussSidRB(u,fu);
            errcur = max(abs(uapx(:)-u(:)));
            err(stop)=errcur; stop=stop+1;
        end
        err=err(find(err));
    elseif(strcmpi(type,'jacobi'))
        %%% Weighted Jacobi
        %
        while (errcur > tol && icur >stop)
            u=wJac(u,fu,omega);
            errcur = max(abs(uapx(:)-u(:)));
            err(stop)=errcur; stop=stop+1;
        end
        err=err(find(err));
    elseif(strcmpi(type,'twogrid'))
        %%% Two—Grid
        %
        while (errcur > tol && icur >stop);
            u=TwoGridP(u,fu,h,3,35);
            errcur = max(abs(uapx(:)-u(:)));
            err(stop)=errcur; stop=stop+1;
        end
        err=err(find(err));
    elseif(strcmpi(type,'multigrid'))
        %%% Multigrid
        %
        while (errcur > tol && icur >stop)
            u=MultiGridP(u,fu,h,3,steps);
            errcur = max(abs(uapx(:)-u(:)));
            err(stop)=errcur; stop=stop+1;
        end
        err=err(find(err));
    else
        %%% Implements mldivide.
        %
        u=Psolve(h,fu); u=reshape(u,ny,nx);
        err=0;
    end
end
```

5.1.2 Matrix free product

The following method was created in order to calculate the product $A\mathbf{u}$ without forming the matrix.

Listing 14: Matrix Free Product

```
%% Matrix Free Calculation of Au
%
%   Written by Ted Kwan for Math 226B
%
%   This function is the matrix free version
%   of A*u. Created using code from FDMcode
%   document by Professor Chen.
function [Au] = AuP(u)
    [n,m]=size(u); i=2:n-1; j=2:m-1;
    Au=zeros(n,m);
    Au(i,j) = 4*u(i,j) - u(i-1,j) - u(i+1,j) ...
            - u(i,j-1) - u(i,j+1);
end
```

5.1.3 Tensor Product Matrix Solution

The following method was created in order to solve the matrix equation $A\mathbf{u} = \mathbf{f}$ by creating the matrix, then using mldivide to directly solve the system.

Listing 15: Solve Finite Difference Matrix Equation

```
%% Create Matrix and solve
%
%   Written by Ted Kwan for Math 226B
%
%   This method solves the matrix equation for
%   the finite difference method.
function [u] = Psolve(h,fu)
    %%% Setup System
    %
    b=fu(:); nx=(1/h)+1; ny=(1/h)+1;
    N=(ny)*(nx); u=zeros(N,1);
    isbd = true(ny,nx);
    isbd(2:end-1,2:end-1) = false;
    isfree=~isbd; intidx=find(isfree(:));
    %%% Create Matrix
    %
    ex = ones(nx,1); ey = ones(ny,1);
    Tx = spdiags([-ex 2*ex -ex], -1:1, nx, nx);
    Ty = spdiags([-ey 2*ey -ey], -1:1, ny, ny);
    A = kron(speye(nx),Ty) + kron(Tx,speye(ny));
    %%% Solve
    %
    u(intidx)=A(intidx,intidx)\b(intidx); u=reshape(u,ny,nx);
end
```

5.2 Finite Element Method

For the Finite element method, a few subroutines were needed in addition to the ones notes above.

5.2.1 Assemble Stiffness Matrix

The following method assembles the stiffness matrix for the finite element method.

Listing 16: Stiffness Matrix

```
%% Assemble Stiffness Matrix.
%
%   Quick method to generate sparse matrix
%   A, the stiffness matrix.
%
function [A,area] = AssembleStiffnessFine(node,elem)
    N=size(node,1); NT=size(elem,1);
    ii=zeros(9*NT,1); jj=zeros(9*NT,1); sA=zeros(9*NT,1);
    ve(:,:,3)=node(elem(:,2),:)-node(elem(:,1),:);
    ve(:,:,1)=node(elem(:,3),:)-node(elem(:,2),:);
    ve(:,:,2)=node(elem(:,1),:)-node(elem(:,3),:);
    area=0.5*abs(-ve(:,1,3).*ve(:,2,2)+ve(:,2,3).*ve(:,1,2));
    index=0;
    for i=1:3
       for j=1:3
           ii(index+1:index+NT)=elem(:,i);
           jj(index+1:index+NT)=elem(:,j);
           sA(index+1:index+NT)=dot(ve(:,:,i),ve(:,:,j),2)./(4*area);
           index=index+NT;
       end
    end
    A=sparse(ii,jj,sA,N,N);
end
```

5.2.2 Calculate Right Hand Side

The following method calculates the right hand side for the finite element method. It was put into a file for code readability.

Listing 17: Calculate Right Hand Side

```
%% Finite Element Method RHS
%
%   Written by Ted Kwan for Math 226B Project 2
%   This function calculates the right-hand side of the FEM.
function [b] = FiniteElemRHS(node,elem,f,area,N)
    %% Calculate RHS
    mid1 = (node(elem(:,2),:)+node(elem(:,3),:))/2;
    mid2 = (node(elem(:,3),:)+node(elem(:,1),:))/2;
    mid3 = (node(elem(:,1),:)+node(elem(:,2),:))/2;
    bt1 = area.*(f(mid2(:,1),mid2(:,2))+f(mid3(:,1),mid3(:,2)))/6;
    bt2 = area.*(f(mid3(:,1),mid3(:,2))+f(mid1(:,1),mid1(:,2)))/6;
    bt3 = area.*(f(mid1(:,1),mid1(:,2))+f(mid2(:,1),mid2(:,2)))/6;
    b = accumarray(elem(:),[bt1;bt2;bt3],[N,1]);
end
```

6 Processor Information

The processor used for the time calculations was an Intel Core i7 720 QM with the following specifications:

- Clock Rate of 1.6 GHz (without overclocking).
- Front Side Bus of 2500 MHz.
- L1 Cache Size of 256 KB.
- L2 Cache Size of 1024 KB.
- L3 Cache Size of 6144 KB.
- 4 cores with 2 threads per core.