Math 6590 Project I: Data denoising in 1D and 2D with Tikhonov regularization: implementations with gradient descent, finite difference and finite element methods

Name: Li Dong
RIN: 661243168

Feb 11, 2016

Abstract

In this report, three classical methods, gradient descent, finite difference and finite element, are implemented on data denoising problems in a 1D signal and a 2D image, respectively. Three methods are compared based on the results and speed. Two classical linear solvers, Gauss-Seidel and conjugate gradient, are used to obtain the solution from the linear systems resulting from finite difference and finite element methods and the performance of the two solvers are compared. Also, the effect of the regularization parameter λ is investigated.

1 Problems

Use MATLAB(2015a, The Mathworks, Inc., Natick, MA, U.S.A,) to implement the following Tikhonov regularization [1] for 1D signal and 2D image denoising

$$\min_{\mathbf{u}} \mathcal{E}(u) = \int_\Omega |\nabla u|^2 dx + \frac{\lambda}{2}\int_\Omega (u-u_0)^2 dx \tag{1}$$

1. Implement the gradient descent method.
2. Solve the Euler-Lagrange equation $\frac{\partial \mathcal{E}}{\partial u} = 0$ to have the optimizer to the above problem using the finite difference method and the finite element method. Use Gauss-Seidel method and conjugate gradient method to solve the corresponding matrix problems.
3. Use the attached data (a 1D signal and a 2D image) to test your codes.

2 Methodology

In this section, a brief explanation about the three methods, gradient descent, finite difference and finite element, is presented.

2.1 Gradient descent method

Forward Euler method is used to implement the gradient descent method. We can design a sequence $\{u_n\}$, such that $\min_{\mathbf{u}} \mathcal{E}(u)$. Clearly, $\{u_n\}$ follows the descent direction of $\mathcal{E}(u)$ in the time evolution, which can be stated as

$$\begin{aligned} \frac{\partial u}{\partial t} &= -\frac{\partial \mathcal{E}}{\partial u}, && (2)\\ \Rightarrow u^{k+1} &= u^k - dt\frac{\partial \mathcal{E}(u^k)}{\partial u}, && (3)\end{aligned}$$

where $\frac{\partial \mathcal{E}}{\partial u} = -2\nabla u^k + \lambda(u^k - u_0)$. The initial condition u^0 is set to u_0. Homogeneous Neumann boundary condition is imposed in the Laplacian operator.

The max iteration number and the relative error tolerance ($|x_k - x_{k-1}|$) are set to 1000 and 1×10^{-4} and the convergence happens when either of the two conditions is met.

2.2 Finite difference method

The Euler-Lagrange equation given by $\frac{\partial \mathcal{E}}{\partial u} = 0$ can be written as

$$(-2\nabla + \lambda)u = \lambda u_0, \tag{4}$$

with the homogeneous Neumann boundary condition $\nabla u \cdot \vec{n} = 0$, which is, again, imposed in the Laplacian operator.

For the Gauss-Seidel and conjugate gradient linear solvers, the max iteration number and the relative error tolerance are set to 10 and 1×10^{-4} and the convergence happens when either of the two conditions is met.

2.3 Finite element method

In order to use finite element method, we need to multiply a test function on both sides of the Euler-Lagrange equation 4 and integrate by parts. Then we have a variational equation

$$2 < \nabla u, \nabla v >_\Omega + \lambda < u, v >_\Omega = \lambda < u_0, v >_\Omega, \tag{5}$$

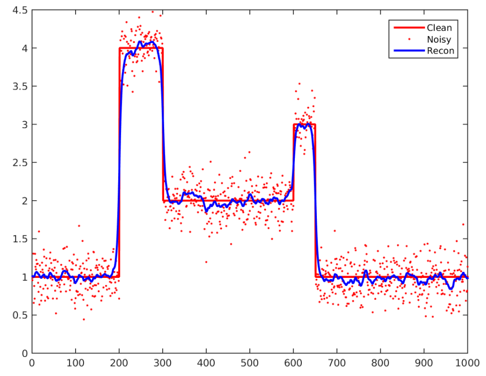
where $< f, g >_\Omega \equiv \int_\Omega fg d\Omega$. After approximating the variables in finite dimensions and expressing them with linear basis, we would have

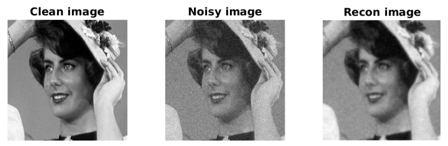
$$(2S + \lambda M)u = \lambda M u_0, \tag{6}$$

where the stiffness matrix $S^e_{ij} =< \nabla e_i, \nabla e_j >_{\Omega^e}$ and the mass matrix $M^e_{ij} =< e_i, e_j >_{\Omega^e}$ are defined in the elemental level and special care should be taken when assembling the global matrix. Details of the derivation and programming of this finite element method follows [2]. The noisy data on the boundaries are imposed as Dirichlet boundary conditions. Triangular mesh is generated by MATLAB's built-in function *delaunay*.

All three methods above are stated in a general way such that it can be applied both in 1D and 2D. Except that the gradient descent method uses a iterative way, both finite difference and finite element methods can reduce to a linear system $Ax = b$, such as Eq. 4 and 6, which are solved with Gauss-Seidel and conjugate gradient solvers iteratively.

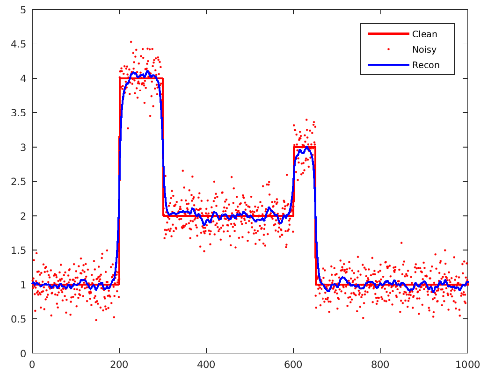
In order to have fair comparison, the convergence criteria for the finite element method is the same as that for finite difference method.

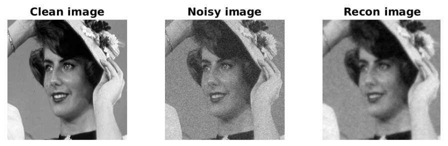




(a) (b)

Figure 1: Results for gradient method (a) 1D signal, $\lambda = 0.1$, (b) 2D image, $\lambda = 1$





(a) (b)

Figure 2: Results for finite difference method (a) 1D signal, $\lambda = 0.1$, (b) 2D image, $\lambda = 1$

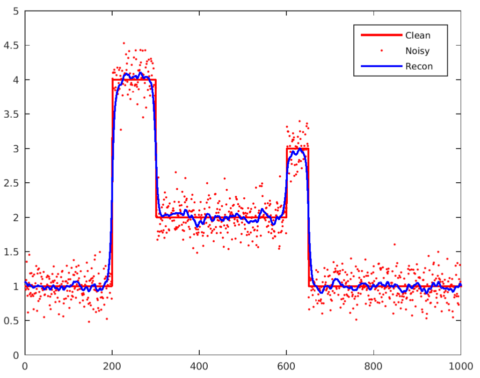
3 Results

All the results in this section are obtained from a Linux environment with Intel Core i7-3820 3.6GHz 4 cores with hyperthread technology and 32 GB ram. Yet the code is not parallelized to use multi-cores.

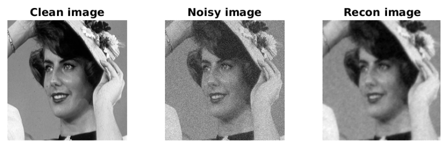
Figures 1, 2 and 3 appear quite similar and they are results from the above three methods with approximately proper λ. It can be seen that the noise are removed from the signal and image significantly.

Figures 4a and 4b display the computational time in second for both the 1D signal case and the 2D image case. For the method of gradient descent, it is simply a iteration process and no linear solver is needed, thus the time for constructing linear equation is simply the total time. The method of gradient descent uses 0.0319 s for 1D and 1.4524 s for 2D, the method of finite difference uses 0.3127 s and 0.0033 s for 1D with Gauss-Seidel and conjugate gradient solvers, respectively, and uses 566.2879 s and 0.0623 s for 2D with Gauss-Seidel and conjugate gradient solvers, respectively, and the method of finite element uses 0.3645 s and 0.0569 s for 1D with Gauss-Seidel and conjugate gradient solvers, respectively, and uses 664.5702 s and 145.0497 s for 2D with Gauss-Seidel and conjugate gradient solvers, respectively.

Also, Figures 5, 6 and 7 and Figures 8, 9 and 10 illustrate the effects of λ for the 1D signal and the 2D image with smaller and larger choices of λ than the λ used in Figures 1, 2 and 3.

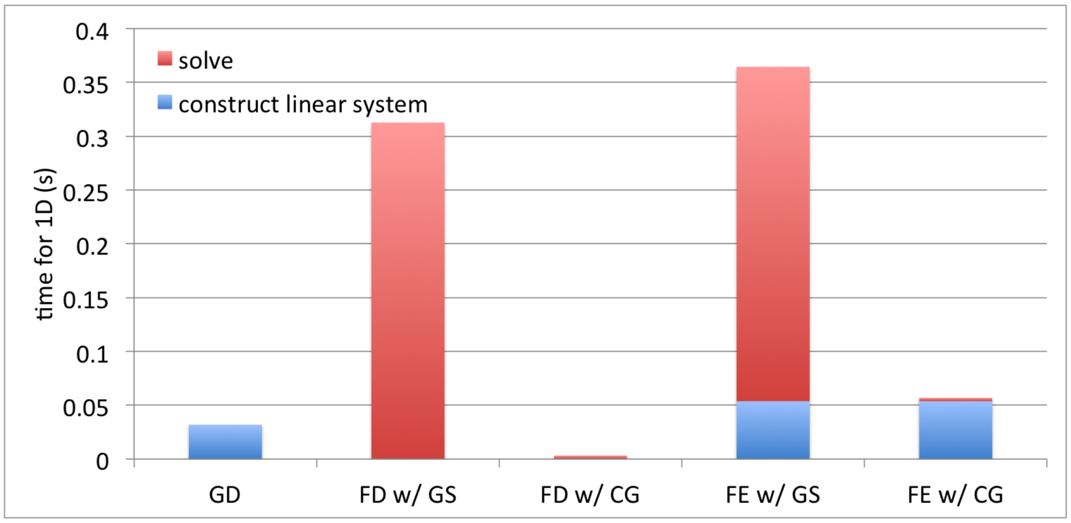


(a)

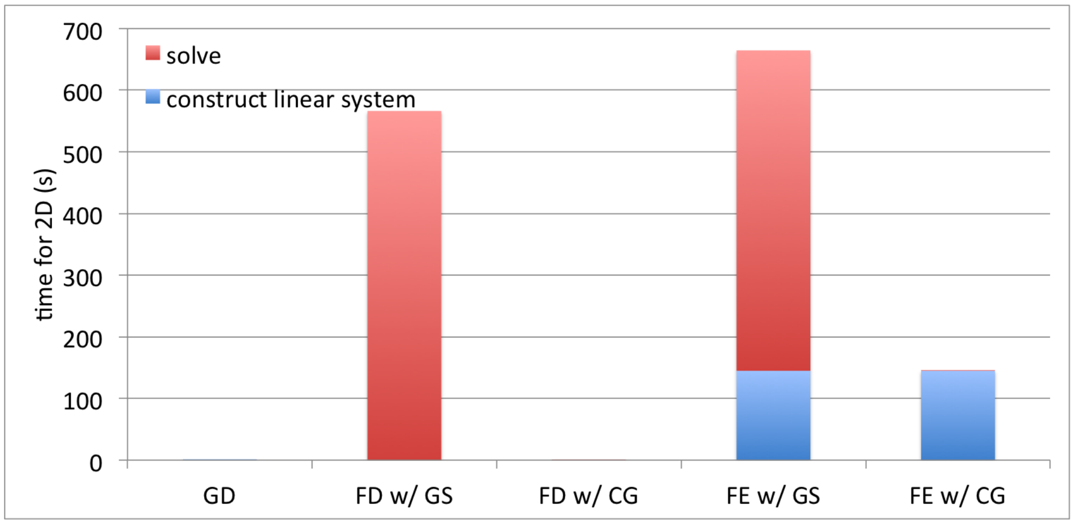


(b)

Figure 3: Results for finite element method (a) 1D signal, $\lambda = 0.1$, (b) 2D image, $\lambda = 1$

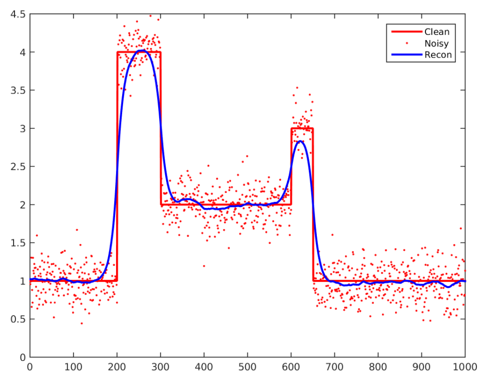


(a)

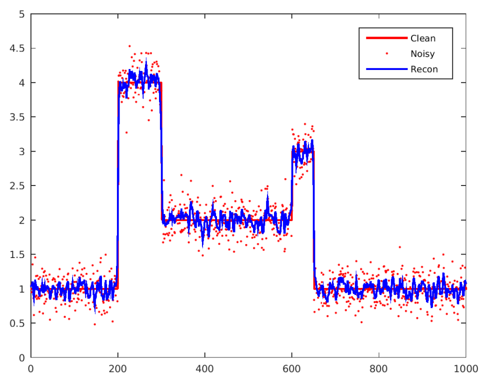


(b)

Figure 4: Computational time.

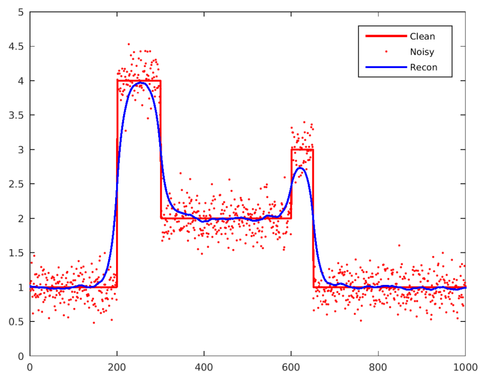


(a)

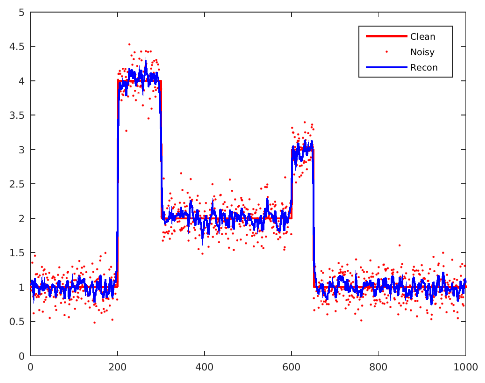


(b)

Figure 5: Results for gradient method of different λ for the 1D signal (a) $\lambda = 0.01$, (b) $\lambda = 1$.

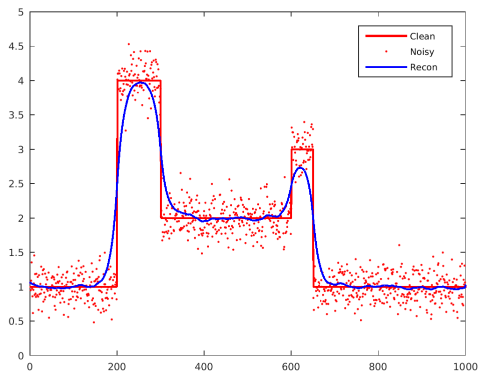


(a)

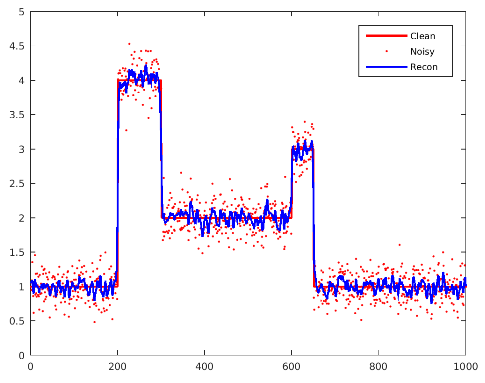


(b)

Figure 6: Results for finite difference method of different λ for the 1D signal (a) $\lambda = 0.01$, (b) $\lambda = 1$.

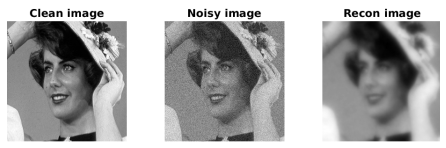


(a)

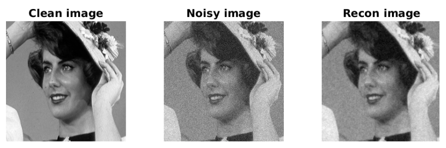


(b)

Figure 7: Results for finite element method of different λ for the 1D signal (a) $\lambda = 0.01$, (b) $\lambda = 1$.

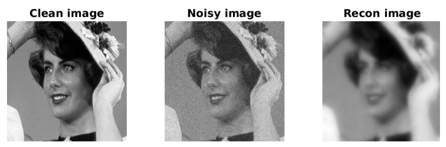


(a)

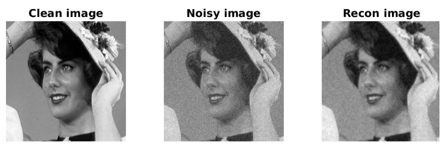


(b)

Figure 8: Results for gradient method of different λ for the 2D image (a) $\lambda = 0.1$, (b) $\lambda = 3$.

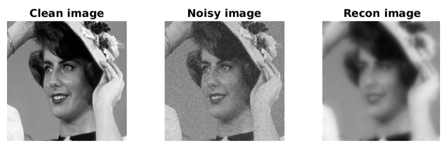


(a)

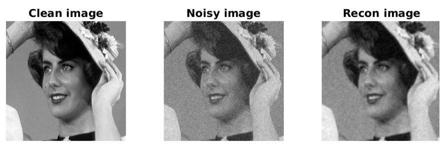


(b)

Figure 9: Results for finite difference method of different λ for the 2D image (a) $\lambda = 0.1$, (b) $\lambda = 3$.



(a)



(b)

Figure 10: Results for finite element method of different λ for the 2D image (a) $\lambda = 0.1$, (b) $\lambda = 3$.

4 Observation and Conclusions

1. By comparing the results from different methods, it is clear that different methods give almost the same solution provided the method can converge properly.
2. Gauss-Seidel solver most of the time needs more than 10 steps to converge and it requires a lot of time per iteration, which makes the 2D solving very slow, while conjugate gradient solver generally converges in 1 step.
3. Using same λ, the method of finite difference uses the least time for both 1D and 2D case. To use a quadrilateral mesh instead of the current triangular mesh for the 2D finite element method could reduce the time for constructing the linear system, but, most probably, it still needs more time than the method of finite difference.
4. The better convergence of finite element method does not seem to stand out in this case. Yet, the significant amount of time that finite element method uses in this study renders this method unattractive under this context.
5. λ controls the smoothness of the reconstruction. It can be guessed that the reconstruction will appear the same as the noisy data if λ is chosen to be infinitely large, provided this ill-posed problem can be solved properly, and everything will be smeared away if $\lambda = 0$.

References

[1] Andreĭ Nikolaevich Tikhonov, Vasiliĭ IAkovlevich Arsenin, and Fritz John. *Solutions of ill-posed problems*. Winston Washington, DC, 1977.

[2] Thomas JR Hughes. *The finite element method: linear static and dynamic finite element analysis*. Courier Corporation, 2012.

Appendix

MATLAB codes.

main.m calls three methods and save results.

```
clear all
close all
clc

[I1, I1_noise, I2, I2_noise] = LoadData;
iter=10; % for gauss-seidel and cg solver
tol=1e-4; % relative error tolerance for three methods

method=1; % 1: gradient descent; 2: finite difference; 3: finite element
lambda=[0.1; 1];
for i=1:size(lambda,2)
    lambda1=lambda(1,i);
    lambda2=lambda(2,i);

    % differential operator
    nx = length(I1);
    ex = ones(nx,1);
    Lap1 = spdiags([ex -2*ex ex], [-1 0 1], nx, nx);
    Lap1(1,1) = -2; Lap1(1,2) = 2; % impose homogeneous Neumann
    Lap1(end,end-1) = 2; Lap1(end,end) = -2;

    % laplacian operator
    [ny, nx] = size(I2);
    ex = ones(nx,1);
    Dxx = spdiags([ex -2*ex ex], [-1 0 1], nx, nx);
    Dxx(1,1) = -1; Dxx(1,2) = 1;
    Dxx(end,end-1) = 1; Dxx(end,end) = -1;
    ey = ones(ny,1);
    Dyy = spdiags([ey, -2*ey ey], [-1 0 1], ny, ny);
    Dyy(1,1) = -1; Dyy(1,2) = 1;
    Dyy(end,end-1) = 1; Dyy(end,end) = -1;
    Lap2 = kron(Dyy, speye(nx)) + kron(speye(ny), Dxx) ;

    switch method
        case 1
            %% gradient descent
            [I1_smooth_GD, I2_smooth_GD] = gradDescent(I1_noise, I2_noise, 1000, tol, lambda1

            figure;
            plot(I1,'r-','LineWidth',2);
            hold on;
            plot(I1_noise,'r.');
            plot(I1_smooth_GD,'b-','LineWidth',2);
            hlen = legend('Clean','Noisy','Recon');
            img_name=sprintf('1D_GD_%g.png',lambda1);
            saveas(gcf,img_name);
            % set(hlen,'FontSize',20);
            % set(gca,'FontSize',2);
            hold off

            figure;
            subplot(1,3,1);
            imshow(I2,[]);
            title('Clean image');%,'FontSize',20);
            subplot(1,3,2);
            imshow(I2_noise,[]);
            title('Noisy image');%,'FontSize',20);
            subplot(1,3,3);
            imshow(I2_smooth_GD,[]);
            title('Recon image');%,'FontSize',20);
            img_name=sprintf('2D_GD_%g.png',lambda2);
            saveas(gcf,img_name);

        case 2
            %% finite difference
            [I1_smooth_FD, I2_smooth_FD] = FD(I1_noise, I2_noise, iter, tol, lambda1,lambda2,

            figure;
            plot(I1,'r-','LineWidth',2);
            hold on;
            plot(I1_noise,'r.');
            plot(I1_smooth_FD,'b-','LineWidth',2);
            hlen = legend('Clean','Noisy','Recon');
            img_name=sprintf('1D_FD_%g.png',lambda1);
            saveas(gcf,img_name);
            % set(hlen,'FontSize',20);
            % set(gca,'FontSize',2);
            hold off

            figure;
            subplot(1,3,1);
            imshow(I2,[]);
            title('Clean image');%,'FontSize',20);
```

```
            subplot(1,3,2);
            imshow(I2_noise ,[]);
            title('Noisy image');%,'FontSize',20);
            subplot(1,3,3);
            imshow(I2_smooth_FD ,[]);
            title('Recon image');%,'FontSize',20);
            img_name=sprintf('2D_FD_%g.png',lambda2);
            saveas(gcf,img_name);

        case 3
            %% finite element
            [I1_smooth_FE, I2_smooth_FE] = FE(I1_noise, I2_noise(1:end,1:end), iter, tol, lam

            figure;
            plot(I1,'r-','LineWidth',2);
            hold on;
            plot(I1_noise,'r.');
            plot(I1_smooth_FE,'b-','LineWidth',2);
            hlen = legend('Clean','Noisy','Recon');
            img_name=sprintf('1D_FE_%g.png',lambda1);
            saveas(gcf,img_name);
            % set(hlen,'FontSize',20);
            % set(gca,'FontSize',2);
            hold off

            figure;
            subplot(1,3,1);
            imshow(I2,[]);
            title('Clean image');%,'FontSize',20);
            subplot(1,3,2);
            imshow(I2_noise ,[]);
            title('Noisy image');%,'FontSize',20);
            subplot(1,3,3);
            imshow(I2_smooth_FE ,[]);
            title('Recon image');%,'FontSize',20);
            img_name=sprintf('2D_FE_%g.png',lambda2);
            saveas(gcf,img_name);

    end

end
```

gradDescent.m implements the gradient descent method for both 1D and 2D cases.

```
function [I1_s, I2_s]=gradDescent(I1_n,I2_n, iter, tol, lambda1, lambda2, Lap1, Lap2)

dt = 0.1;
```

```
I1_k = I1_n;
error1 = 1e8;
tic;
for i=1:iter
    I1_k_tmp = I1_k - dt*(-2*Lap1*I1_k + lambda1*(I1_k-I1_n));
    error1 = norm(I1_k_tmp - I1_k);
    I1_k = I1_k_tmp;
    if error1 < tol
        str=sprintf('Reaching tol at iter %d!',i);
        display(str);
        break
    end
end
t_1D_GD=toc
I1_s = I1_k;

I2_k = I2_n;
I2_k_tmp = zeros(size(I2_n));
error2 = 1;
tic
for i=1:iter
    I2_k_tmp(:) = I2_k(:) - dt*(-2*Lap2*I2_k(:) + lambda2*(I2_k(:)-I2_n(:)));
    error2 = norm(I2_k_tmp - I2_k);
    I2_k = I2_k_tmp;
    if error2<tol
        break
    end
end
t_2D_GD=toc
I2_s = I2_k;

end
```

FD.m implements the finite difference method for both 1D and 2D cases.

```
function [I1_s, I2_s] = FD(I1_n,I2_n, iter, tol, lambda1, lambda2, Lap1, Lap2)

tic
A=-2*Lap1+lambda1*speye(size(Lap1));
b=lambda1*I1_n;
t_1D_FD=toc

tic
I1_s = Gauss_Seidel(A,b,iter, tol);
t_1D_FD_GS=toc

tic
I1_s = conjgrad(A,b,iter, tol);
```

```
t_1D_FD_CG=toc

% I1_s= ( -2*Lap1+lambda1*speye(size(Lap1)) ) \ ( lambda1*I1_n );

tic
I2_s = zeros(size(I2_n));
A=-2*Lap2+lambda2*speye(size(Lap2));
b=lambda2*I2_n(:);
t_2D_FD=toc

% tic
% I2_s(:) = Gauss_Seidel(A,b,iter, tol);
% t_2D_FD_GS=toc

tic
I2_s(:) = conjgrad(A,b,iter, tol);
t_2D_FD_CG=toc

% I2_s(:) = ( -2*Lap2+lambda2*speye(size(Lap2)) ) \ ( lambda2*I2_n(:) );


end
```

FE.m implements the finite element method for both 1D and 2D cases.

```
function [I1_s, I2_s] = FE(I1_n, I2_n, iter, tol, lambda1,lambda2)

tic
nx1=length(I1_n);

S_e = [1, -1; -1, 1];
M_e = [1/3, 1/6; 1/6, 1/3];

A = sparse(nx1,nx1);
M = sparse(nx1,nx1);
for i=1:nx1-2
    A_e = 2*S_e + lambda1*M_e;
    A(i:i+1,i:i+1) = A(i:i+1,i:i+1) + A_e;
    M(i:i+1,i:i+1) = M(i:i+1,i:i+1) + M_e;
end
f = M*lambda1*I1_n;
% apply left boundary condition
A(1,:) = 0;
A(1,1) = 1.0;
f(1) = I1_n(1);
f(2:end) = f(2:end) - A(2:end,1) * I1_n(1);
A(2:end,1) = 0;
% apply right boundary condition
```

```
A(end,:) = 0;
A(end,end) = 1.0;
f(end) = I1_n(end);
f(1:end-1) = f(1:end-1) - A(1:end-1,end) * I1_n(end);
A(1:end-1,end) = 0;
t_1D_FE=toc

tic
I1_s = Gauss_Seidel(A,f,iter, tol);
t_1D_FE_GS = toc

tic
I1_s = conjgrad(A,f,iter, tol);
t_1D_FE_CG = toc
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 2D
tic
downSampleRatio = 256/size(I2_n,1);
% generate uniform triangular mesh
a = 1;
b = 256/downSampleRatio;
h0 = (b-a)/(size(I2_n,1)-1);

[X, Y]=meshgrid(a:h0:b, a:h0:b);
t = delaunay(X, Y);
% figure;
% trimesh(t, X, Y, zeros(size(X)), 'EdgeColor', 'Black');
% view(2);
% axis equal;
p = zeros(length(X)*length(Y), 2);
for i = 1 : length(X)
    for j = 1 : length(X)
        p(j + length(X)*(i-1), 1) = X(j, i);
        p(j + length(X)*(i-1), 2) = Y(j, i);
    end
end

% boundary nodes
b=(p == a | p == b);
a=sum(b,2);
a=a(a>0);
Bnodes=zeros(size(a,1),1);        % Number of nodes on the boundary,
% initialization
j=1;
for i= 1:size(p,1)                % scanning through all the nodes
    if sum(b(i,:))>0              % that means the node is on the boudary
        Bnodes(j,1)=i;
        j=j+1;
    end
end


nodes=max(t(:));% total number of nodes

%{
    This is to index the nodes based on whether the node is a dof or dog.
    and what is the global dof and global dog
%}

Node_Flag=zeros(nodes,1); % this is to identify
                               % whether the node is dof or dog
Node_Index=zeros(nodes,1);% this gives the (global dog or dof)
                               % index of the node
nif=1;                         % nif: node index of dof
nig =1 ;                       % nig: node index of dog
for nn=1:nodes
    if sum(Bnodes==nn)==0      % this means the nodes is a dof
        Node_Flag(nn,1)=1;     % 1 means its a dof/or interior nodes
        Node_Index(nn,1)=nif;
        nif=nif+1;
    else
        Node_Flag(nn,1)=0;     % 0 means the non homogenous
                               % DBC/boundary nodes in present hw problem
        Node_Index(nn,1)=nig;
        nig=nig+1;
    end
end

%{
    This contains all the information about the node .
First column of NodeInfo:
    Global Node number
Second column of NodeInfo:
    0/1: Interior node/Boundary node respectively
    Note it has been assumed in the present HW problem that all the
    boundary nodes have non homogenous DBC
Third Column of NodeInfo:
   Global Indexing based on the node is a dof( interior node) or
   a dog (Boundary node)
%}
NodeInfo=[unique(t(:)),Node_Flag,Node_Index];

% Initialization
AI=max(NodeInfo(NodeInfo(:,2)==1,3));
```

```
% size of A matrix, max index of the interior node
A=sparse(AI,AI);%zeros(AI);
M=sparse(AI,AI);%zeros(AI);
BI=(max(NodeInfo(NodeInfo(:,2)==0,3)));% max index of the boudndary node
B=sparse(AI,BI);%zeros(AI,BI);
FI=zeros(AI,1);

for k=1:size(t,1)

    [K_El, load_El]=element(k, t, p, lambda2,I2_n);

    for ni=1:3
        if NodeInfo(t(k,ni),2)==1 % the node is the interior node, a dof
            for kni=1:3

            % the above node is interacting with the interior node, a dof
              if NodeInfo(t(k,kni),2)==1
                  A(NodeInfo(t(k,ni),3),NodeInfo(t(k,kni),3))=...
                  A(NodeInfo(t(k,ni),3),NodeInfo(t(k,kni),3))+K_El(ni,kni);
              else% the interior node is interact with the boundary node
                  B(NodeInfo(t(k,ni),3),NodeInfo(t(k,kni),3))=...
                  B(NodeInfo(t(k,ni),3),NodeInfo(t(k,kni),3))+K_El(ni,kni);
              end
            end
            FI(NodeInfo(t(k,ni),3),1)=FI(NodeInfo(t(k,ni),3),1)+...
                                        load_El(ni,1);
        end

    end
end
% Non homogenous Drichlet Boundary Condition
G=I2_n(:); G=G(NodeInfo(:,2)==0);
FI=FI-B*G;
t_2D_FE=toc

% I2_s=A \ FI;
% tic
% I2_s = Gauss_Seidel(A,FI,iter, tol);
% t_2D_FE_GS=toc

tic
I2_s = conjgrad(A,FI,iter, tol);
t_2D_FE_CG=toc

I2_s=reshape(I2_s,sqrt(length(I2_s)),sqrt(length(I2_s)));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
end
```

element.m computes stiffness and mass matrices for the finite element method.

```
function [El_stiff, El_load]=element(k, t, p, lambda2,I2_n)
a123=(p(t(k,1), :) + p(t(k,2), :) + p(t(k,3), :))/3;
K=[p(t(k,:)',:),[1;1;1]];
area=abs(det(K))/2;

El_mass_k=zeros(3);
for i=1:3
    for j=1:3

        El_mass_k(i,j)=shapefun(a123(1,1),a123(1,2),k,t(k,i), t, p)*...
            shapefun(a123(1,1),a123(1,2),k,t(k,j), t, p);
    end
end

El_mass_k=area*El_mass_k;
El_stiff_k=zeros(3);

Coor = zeros(2, 3);
Coor(1, :) = p(t(k, :), 1);
Coor(2, :) = p(t(k, :), 2);

grad_N = 1/det(K).*[Coor(2,2)-Coor(2,3), ...
                    Coor(2,3)-Coor(2,1), ...
                    Coor(2,1)-Coor(2,2);...
                    Coor(1,3)-Coor(1,2), ...
                    Coor(1,1)-Coor(1,3), ...
                    Coor(1,2)-Coor(1,1)];
% [y2-y3, y3-y1, y1-y2; x3-x2, x1-x3, x2-x1]

for ncx=1:3
    for ncy=1:3
      El_stiff_k(ncx,ncy)=sum(grad_N(:,ncx).*grad_N(:,ncy),1);
    end
end
El_stiff_k=El_stiff_k*area;

El_stiff=2*El_stiff_k+lambda2*El_mass_k;

El_load=[0;0;0];
for ni=1:3
    for  nj=1:3
        Coor= p(t(k,nj),:);
        El_load(ni)=El_load(ni)+...
            I2_n(Coor(2),Coor(1))*lambda2*El_mass_k(ni,nj);
    end
end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Input:      xn: x coordinate
%             yn: y coordinate
%             k: element number
%             n: node number
%             t: follows above
%             p: follows above
% Output:     shapefun_val = shape function evaluated at xn and yn
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [shapefun_val] = shapefun(xn, yn, k, n, t, p)

[~, fu]=find(t(k,:)==n);

K=[p(t(k,:)',:),[1;1;1]];

A=K;

A(fu,:)=[xn yn 1];

shapefun_val=det(A)/det(K);

end
```

Gauss_Seidel.m implements the classical Gauss Seidel solver.

```
function x=Gauss_Seidel(A,b,iter,tol)
    n = length(A);
    x = zeros(size(b));
    x_tmp = zeros(size(b));
    for i=1:iter
        for i = 1:n
            x_tmp(i) = (1/A(i, i))*(b(i) - A(i, 1:n)*x + A(i, i)*x(i));
        end
            error = norm(x_tmp-x);
            x = x_tmp;
            if error < tol
                str=sprintf('Reaching tol at iter %d!',i);
                display(str);
                break
            end
    end
end
```

conjgrad.m implements the classical conjugate gradient solver.

```
function x = conjgrad(A,b,iter, tol)

x = b;
r = b - A*x;
y = -r;
z = A*y;
s = y'*z;
t = (r'*y)/s;
x = x + t*y;

for i=1:iter
    for k = 1:numel(b);
       r = r - t*z;
       if( norm(r) < tol )
           str=sprintf('Reaching tol at iter %d!',i);
           display(str);
            return;
       end
       B = (r'*z)/s;
       y = -r + B*y;
       z = A*y;
       s = y'*z;
       t = (r'*y)/s;
       x = x + t*y;
    end
end

 end
```