MATH 228b: HW6

April Novak

April 25, 2017

1

The level set method propagates an implicit curve through a domain by solving a time-dependent convection equation, where the velocity in this equation is directly related to the physical processes that would move such a surface. Tracking the location of a surface is essential for numerical applications with moving boundaries, since the surface location must be known in order to apply boundary conditions correctly. For an implicit surface given by $\phi(\vec{x})$, where $\phi = 0$ on the surface, $\phi < 0$ within the surface, and $\phi > 0$ outside, the convection equation that propagates the surface is:

$$\frac{\partial \phi}{\partial t} + \vec{V} \cdot \nabla \phi = 0 \tag{1}$$

where $\vec{V}$ is the speed of the surface. In physical applications, it is often the case that the velocity of the surface is in the normal direction $\vec{n}$ of the curve, dictated by some speed function F:

$$\vec{V} = F\vec{n} \tag{2}$$

Inserting this speed into the convection equation gives the differential equation that propagates the implicit surface:

$$\frac{\partial \phi}{\partial t} + F|\nabla \phi| = 0 \tag{3}$$

For cases where F is always positive, this initial value problem can be restated as a boundary value problem in terms of an arrival function T, where $T(\vec{x})$ represents the time for the interface to reach location $\vec{x}$ from its initial location Γ. This equation is:

$$|\nabla T|F = 0 \tag{4}$$

where $T = 0$ on Γ, since it takes zero time to arrive at the initial condition. Because T represents the time to reach the arrival point, $T(x_{arrival}, y_{arrival}) = t$. The level set with $T = t$ represents the furthest point that can be reached in time t from the initial condition. Individual points move in the normal direction to these level sets, which allows this formulation to determine the optimal travel paths to the arrival location given an initial position.

Instead of solving the above equation, we will use a time-dependent method that assumes that the solution will converge to the steady-state result in a sufficient number of iterations. Hence, we will solve:

$$\frac{\partial T}{\partial t} + |\nabla T|F = 1 \tag{5}$$

for initial condition $T(x_0, y_0) = 0$, where x_0 and y_0 are on the initial curve Γ. Integrating the above equation in time until steady-state is reached will give the solution to the Eikonal equation. A first-order upwinded scheme is used in space, and a simple forward Euler method in time. The initial condition for all cases is set to 0.0, and after every iteration, the sole boundary condition of $\phi(0.2, 0.2) = 0$ is strictly implied. A layer of ghost cells of a large value of ϕ are inserted on the periphery in order to simplify the application of the one-sided upwind differencing. For this time-stepping solution, a time step of 0.0001 seconds is chosen, and appears sufficient.

Figures 1 through 4 show contours of the solution to ϕ after reaching an iteration tolerance (defined based on successive differences in the infinity norm of ϕ from one iteration to the next) of $1e-10$ for the various cases given. The optimal path from the starting point and the given arrival point is calculated using a simple stepping scheme. Starting at the arrival point, a local gradient is computed using Matlab's `interp2` function applied to the results of Matlab's `gradient` function. The starting point is propagated for a small time step of `0.001` in the direction of this normal. After this short time, the gradient of this new location is re-computed, the point moved again, and the process continued until the point is within a distance of h from the starting point. The optimal paths are shown in the figures below as black lines, with the direction of travel beginning at $(0.2, 0.2)$ and ending at the given arrival point. As can be seen, the optimal path follows the local normal of ϕ, and reaches the starting point since this method is essentially a type of steepest-descent method.

Case 4 places a square of width 0.2 centered on $(0.2, 0.2)$ and at $(0.6, 0.6)$, where the speed inside the square is 0.01 and outside is 1.0. Because the speed is so low through these two regions, the optimal path is to move around both squares, instead of taking the shortest-distance path through the centers of the squares.

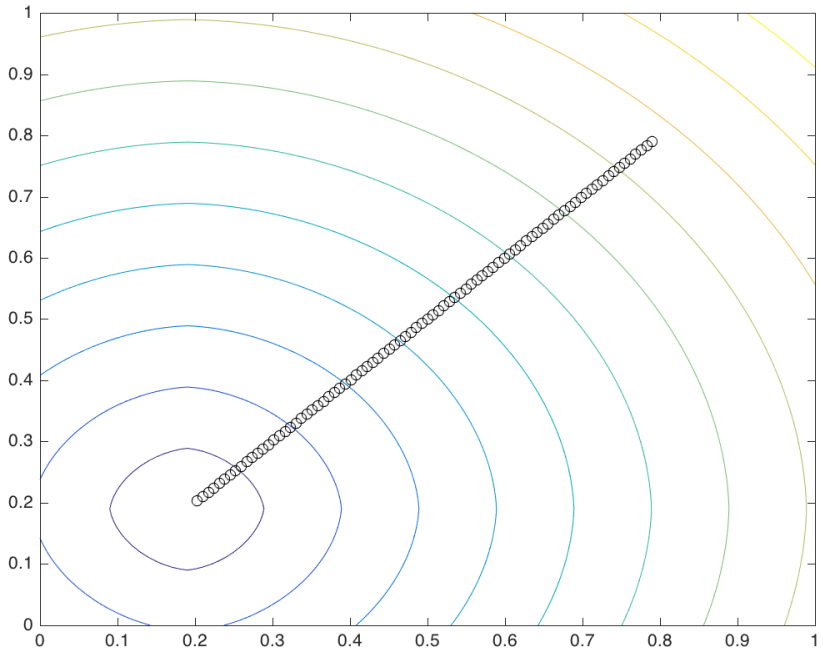

Figure 1. Contours of $\phi(x, y)$ and the optimal travel path for case 1.

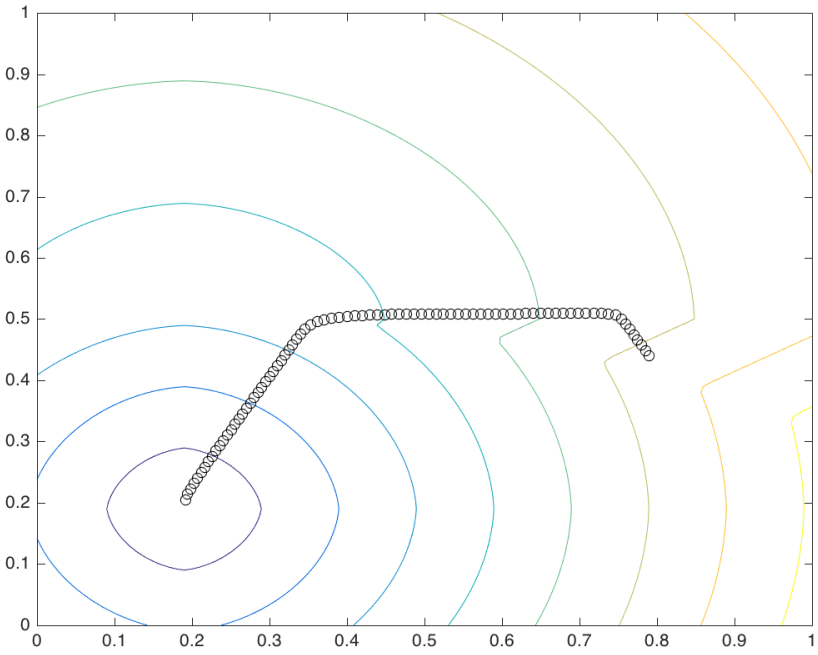

Figure 2. Contours of $\phi(x, y)$ and the optimal travel path for case 2.

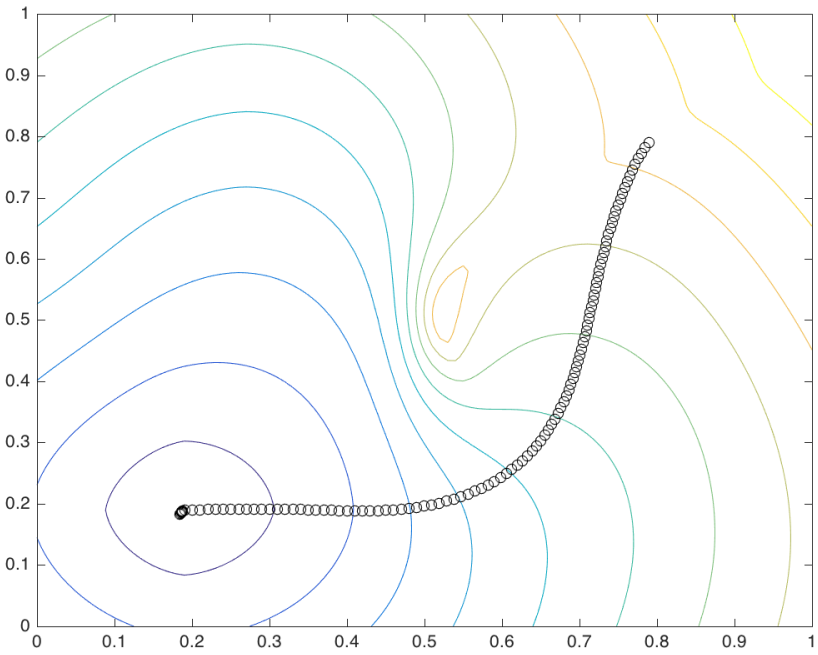

Figure 3. Contours of $\phi(x, y)$ and the optimal travel path for case 3.

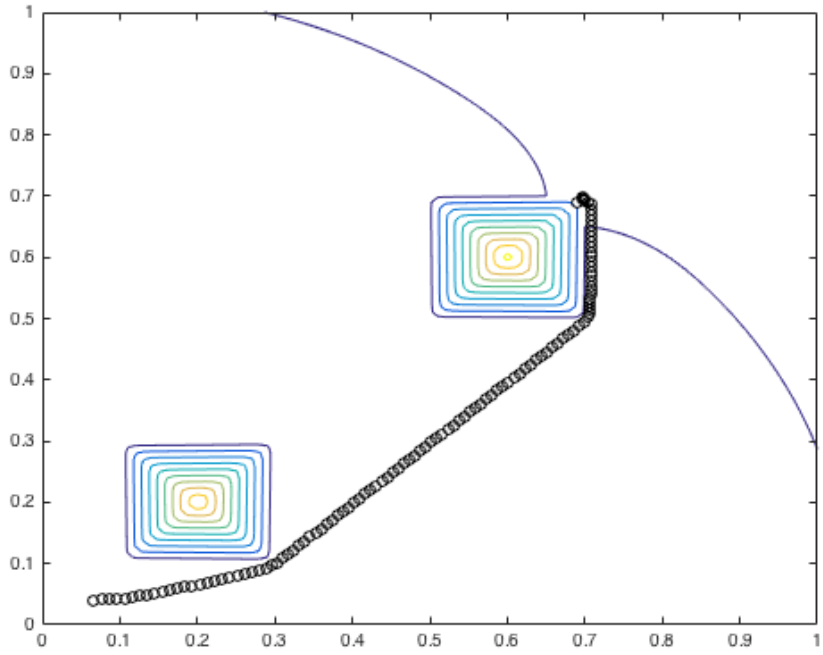

Figure 4. Contours of $\phi(x, y)$ and the optimal travel path for case 4.

2

(a)

The first part of this question develops a Gauss-Seidel solver given a matrix system $Au = b$ for an initial guess u_0. This method is very similar to the Jacobi method, where instead of solving the time-independent problem of interest, we insert a time derivative term and iterate until the solution converges to a steady-state result (with the assumption that this steady-state result is therefore also equivalent to the solution to the problem with no time derivative at all). Gauss-Seidel differs from the Jacobi method in that the most recent iterate is used to perform the update. The code developed for this section is included in the Appendix.

(b)

This question solves the Poisson equation using the multigrid method. The first step is a modification to Professor Persson's `pmesh` function (which I assume is okay because we were told for assignment 3 that we could use his function moving forward to avoid points being lost across multiple assignments) such that it outputs the `data` structured array data structure containing `data.p`, `data.t`, and `data.e` members. This is performed by manipulating the contents of the original `pmesh` function, so make sure to use the `pmesh` function that I submit with this assignment! This edited function is included in the Appendix.

In order to simplify the multigrid method, which relies upon repeated applications of interpolation (expanding a coarse solution to a finer mesh) and reduction (removing elements from a solution to coarsen), these interpolation and reduction matrices are pre-computed. The matrices A and vectors b are then saved for the finest refinement level by running `fempoi` only a single time, and then using the expansion and reduction matrices to project the finest matrix onto the coarser meshes.

(c) The last part of this question solves the Poisson equation using the multigrid method, as shown in the `mgsolve` function. A feature of the multigrid method is that the number of iterations required is independent of the mesh refinement. Convergence rates are shown in Fig. 5, where it can indeed be seen that the number of iterations required to reach an infinity norm error of $1e-10$ is roughly constant except for the coarsest mesh (and I'm not sure why that one converges faster). However, this is somewhat misleading, since the work per iteration does *not* stay constant with mesh spacing.

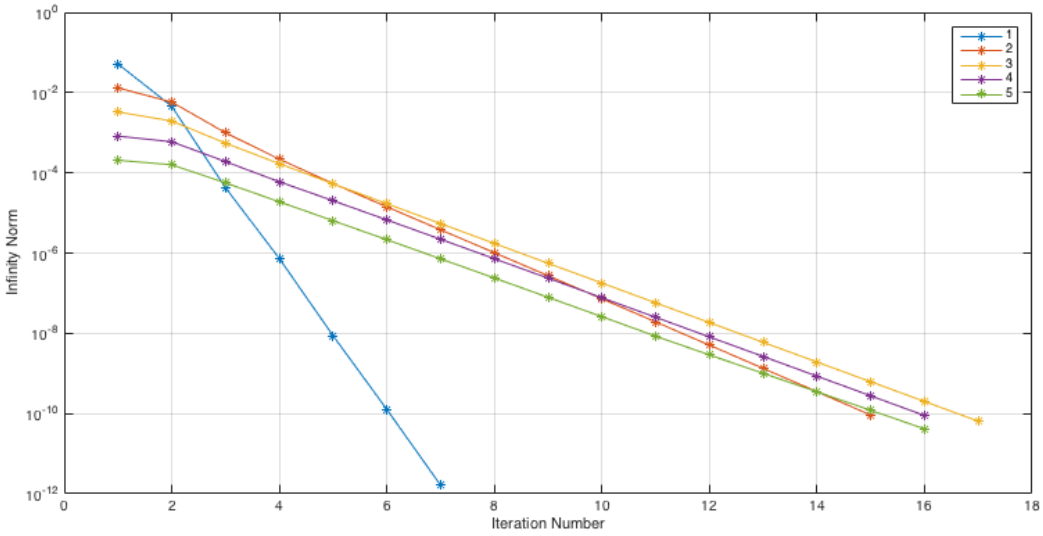

Figure 5. Infinity norm as a function of iteration number for various refinement levels

3 Appendix

3.1 Question 1

3.1.1 eikonal.m

```
% solve the Eikonal equation on unit square
clear all
X = 1;
Y = 1;

dx = 0.01;  dy = 0.01;       % mesh spacing = 1/100
dt = 1e-3;                   % time step
x  = 0:dx:X; y = 0:dy:Y;     % Cartesian grid
[XX, YY] = meshgrid(x, y);
numx = length(x);
numy = length(y);

opt   = 4;                   % flag to select which case to run
initx = 0.2;                 % x-coordinate of starting point
inity = 0.2;                 % y-coordinate of starting point
norm = 1.0; ifig = 0;        % arbitrary initial values
tol  = 1e-10;                % iteration tolerance

phi = 0.0 .* ones(numx, numy);
F   = ones(numx, numy);
nx  = zeros(numx, numy);
ny  = zeros(numx, numy);

switch opt
    case 1
        arrivx = 0.8;
        arrivy = 0.8;
    case 2
```

```
        for i = 1:numy
            if y(i) < 0.5
                F(i, :) = 0.5;
            end
        end
        arrivx = 0.8;
        arrivy = 0.45;
    case 3
        for i = 1:numx
            for j = 1:numy
                F(i, j) = 1 - 0.9 * cos(4 .* pi .* x(j)) ...
                    .* exp(-10 .* ((x(j) - 0.5) .^ 2 + (y(i) - 0.5).^2));
            end
        end
        arrivx = 0.8;
        arrivy = 0.8;
    case 4
        initx = 0.05;
        inity = 0.05;
        arrivx = 0.8;
        arrivy = 0.8;
        low = [0.1, 0.3];
        for i = 1:numx
            for j = 1:numy
                if (low(1) < x(i)) && (x(i) < low(2))
                    if (low(1) < y(j)) && (y(j) < low(2))
                        F(i, j) = 0.01;
                    end
                end
                if (low(1) + 0.4 < x(i)) && (x(i) < low(2) + 0.4)
                    if (low(1) + 0.4 < y(j)) && (y(j) < low(2) + 0.4)
                        F(i, j) = 0.01;
                    end
                end
            end
        end
    otherwise
        disp('Problem selection undefined.')
end

while norm > tol
    phi_prev = phi;

    for i = 1:length(x) % loop over x-coordinates
        for j = 1:length(y) % loop over y-coordinates
            plus = grad_plus(phi, i, j, dx, dy, numx, numy);
            phi(i, j) = phi(i, j) - dt * (max(F(i, j), 0) * plus) + dt;
        end
    end

    % apply the only boundary condition
    phi(initx/dx, inity/dy) = 0.0;

    % find the infinity norm of the difference
    norm = max(max(abs(phi_prev - phi)));
end

contour(XX, YY, phi)
hold on
```

```
% compute gradients, then find the normal vectors
[fx, fy] = gradient(phi, dx, dy);
nx       = fx ./ sqrt(fx .^ 2 + fy .^ 2);
ny       = fy ./ sqrt(fx .^ 2 + fy .^ 2);

dn   = 0.01; % time step for tracing out the shortest path
xc   = [x(arrivx/dx)];
yc   = [y(arrivy/dy)];
dist = sqrt((xc - initx).^2 + (yc - inity).^2);

% find the optimal path by moving in direction opposite the gradient
i = 1;
while ((dist > 2*dx) && (i < 2000))
    nx_loc = interp2(XX, YY, nx, xc(end), yc(end));
    ny_loc = interp2(XX, YY, ny, xc(end), yc(end));

    i = i + 1;
    if i > 2000
        break;
    end

    xc = [xc, xc(end) - nx_loc * dn];
    yc = [yc, yc(end) - ny_loc * dn];

    dist = sqrt((xc(end) - initx).^2 + (yc(end) - inity).^2);
end

hold on
scatter(xc, yc, 'ko')

saveas(gcf, sprintf('case%i.png',opt))
```

3.1.2 grad_plus.m

```
function plus = grad_plus(phi, i, j, dx, dy, numx, numy)
    big = 1000;

    if i == 1
        a = max(phi(i, j)/dx - big, 0) ^ 2;
    else
        a = max((phi(i, j) - phi(i - 1, j))/dx, 0) ^ 2;
    end

    if i == numx
        b = min(big - phi(i, j)/dx, 0) ^ 2;
    else
        b = min((phi(i + 1, j) - phi(i, j))/dx, 0) ^ 2;
    end

    if j == 1
        c = max((phi(i, j) - big) / dy, 0) ^ 2;
    else
        c = max((phi(i, j) - phi(i, j - 1)) / dy, 0) ^ 2;
    end

    if j == numy
        d = min(big -phi(i, j) / dy, 0) ^ 2;
```

```
    else
        d = min((phi(i, j + 1) - phi(i, j)) / dy, 0) ^ 2;
    end

    plus = sqrt(a + b + c + d);
end
```

3.2 Question 2, part a

3.2.1 gauss_seidel.m

```
function [u, res] = gauss_seidel(A, b, u, niter)
% A - matrix in A * u = b
% b - vector in A * u = b
% u - initial Gauss-Seidel solution guess
% niter - number of Gauss-Seidel iterations

% Gauss-Seidel solution to Au = b
res = zeros(1, niter + 1);
res(1) = max(b - A * u);

for it = 1:niter
    u = u + inv(tril(A)) * (b - A * u);
    res(it + 1) = max(b - A * u);
end
```

3.3 Question 2, part b

3.3.1 pmesh.m

```
function [p,t,e, data] = pmesh(pv, hmax, nref)
% PMESH  Delaunay refinement mesh generator.
% UC Berkeley Math 228B, Per-Olof Persson <persson@berkeley.edu>
% then, modified by April Novak

data = struct('p', {}, 't', {}, 'e', {}, 'T', {}, 'R', {}, ...
              'A', {}, 'b', {}, 'u', {}, 'err', {}, 'r', {});

p = [];
for i = 1:size(pv,1)-1
  pp = pv(i:i+1,:);
  L = sqrt(sum(diff(pp,[],1).^2, 2));
  if L>hmax
    n = ceil(L / hmax);
    pp = interp1([0,1], pp, (0:n) / n);
  end
  p = [p; pp(1:end-1,:)];
end

while 1
  t = delaunayn(p);
  t = removeoutsidetris(p, t, pv);

  area = triarea(p,t);
  [maxarea, ix] = max(area);
  if maxarea < hmax^2 / 2, break; end
```

```
  pc = circumcenter(p(t(ix,:),:));
  p(end+1,:) = pc;
end

e = boundary_nodes(t);

% save the first field
data(1).p = p;
data(1).t = t;
data(1).e = e;

for iref = 1:nref
    [pmid, ia] = edgemidpoints(p,t);
    p = [p; pmid];
    t = delaunayn(p);
    t = removeoutsidetris(p, t, pv);
    e = boundary_nodes(t);

    % save all subsequent fields
    data(iref + 1).p = p;
    data(iref + 1).t = t;
    data(iref + 1).e = e;

    % compute the interpolation matrix to upgrade refinement iref
    old_pts = length(data(iref).p);
    new_pts = length(data(iref + 1).p);
    data(iref).T  = zeros(new_pts, old_pts);

    data(iref).T(1:old_pts, 1:old_pts) = eye(old_pts);
    num_tri = length(data(iref).t(:, 1)); % number of triangles from previous

    for i = 1:(new_pts - old_pts) % for each row in T that is _new_
        row = ia(i); % row from _entire_ pmid

        if row <= num_tri % on side 1-2
            subrow = row;
            data(iref).T(i + old_pts, data(iref).t(subrow, 1)) = 0.5;
            data(iref).T(i + old_pts, data(iref).t(subrow, 2)) = 0.5;
        elseif ((num_tri < row) && (row <= 2*num_tri)) % on side 2-3
            subrow = row - num_tri;
            data(iref).T(i + old_pts, data(iref).t(subrow, 2)) = 0.5;
            data(iref).T(i + old_pts, data(iref).t(subrow, 3)) = 0.5;
        else % on side 3-1
            subrow = row - 2*num_tri;
            data(iref).T(i + old_pts, data(iref).t(subrow, 3)) = 0.5;
            data(iref).T(i + old_pts, data(iref).t(subrow, 1)) = 0.5;
        end
    end

    % compute the reduction matrix as the transpose of T, with rows
    % scaled to unity
    data(iref).R = transpose(data(iref).T);
    for i = 1:length(data(iref).R(:, 1)) % loop over the rows
        data(iref).R(i, :) = data(iref).R(i, :) ./ sum(data(iref).R(i, :));
    end
end

function [pmid, ia] = edgemidpoints(p, t)
```

```
pmid = [(p(t(:,1),:) + p(t(:,2),:)) / 2;
        (p(t(:,2),:) + p(t(:,3),:)) / 2;
        (p(t(:,3),:) + p(t(:,1),:)) / 2];

[pmid, ia, ic] = unique(pmid, 'rows');


function a = triarea(p, t)

d12 = p(t(:,2),:) - p(t(:,1),:);
d13 = p(t(:,3),:) - p(t(:,1),:);
a = abs(d12(:,1).*d13(:,2) - d12(:,2).*d13(:,1)) / 2;

function t = removeoutsidetris(p, t, pv)

pmid = (p(t(:,1),:) + p(t(:,2),:) + p(t(:,3),:)) / 3;
isinside = inpolygon(pmid(:,1), pmid(:,2), pv(:,1), pv(:,2));
t = t(isinside,:);

function pc = circumcenter(p)

dp1 = p(2,:) - p(1,:);
dp2 = p(3,:) - p(1,:);

mid1 = (p(1,:) + p(2,:)) / 2;
mid2 = (p(1,:) + p(3,:)) / 2;

s = [-dp1(2),dp2(2); dp1(1),-dp2(1)] \ [-mid1 + mid2]';
pc = mid1 + s(1) * [-dp1(2),dp1(1)];
```

3.3.2 mginit.m

```
function data = mginit(pv, hmax, nref)

[p, t, e, data] = pmesh(pv, hmax, nref);

% determine the A and b matrices at the finest mesh by calling fempoi
[data(nref + 1).u, data(nref + 1).A, data(nref + 1).b] = ...
    fempoi(data(nref + 1).p, data(nref + 1).t, data(nref + 1).e);

% reduce the finest-mesh A and b
for i = fliplr(1:nref)
    data(i).A = data(i).R * data(i + 1).A * data(i).T;
    data(i).b = data(i).R * data(i + 1).b;
end

end
```

3.4 Question 2, part c

3.4.1 mgsolve.m

```
function [sol, res] = mgsolve(data, vdown, vup, tol, nref)
% data  - structured array containing problem information
% vdown - pre-smoothing G-S iterations
```

```
% vup   - post-smoothing G-S iterations
% tol   - res convergence tolerance

res            = [];
loop_iter      = 1;
data(nref + 1).u = 0 .* data(nref + 1).b;
data(nref + 1).r = data(nref + 1).b - data(nref + 1).A * data(nref + 1).u;
res(loop_iter) = max(abs(data(nref + 1).r));

while res(loop_iter) > tol
    i = nref;

    % perform G-S on the original equation (A * u = b)
    [data(i + 1).u] = gauss_seidel(data(i + 1).A, data(i + 1).b, data(i + 1).u, vdown)
        ↪ ;

    % compute a residual (r = b - A * ~u~)
    data(i + 1).r = data(i + 1).b - data(i + 1).A * data(i + 1).u;

    % restrict to a coarser mesh (r_h -> r_2h)
    data(i).r = data(i).R * data(i + 1).r;

    for i = fliplr(1:(nref - 1))
        % smoothing the error equation (initial guess is zero)
        % solves (A * e_2h = r_2h)
        [data(i + 1).err] = gauss_seidel(data(i + 1).A, data(i + 1).r, 0 .* data(i +
            ↪ 1).r, vdown);

        % compute a residual (r_2h = r_2h - A * ~e_2h~)
        data(i + 1).r = data(i + 1).r - data(i + 1).A * data(i + 1).err;

        % restrict the residual
        data(i).r = data(i).R * data(i + 1).r;
    end

    % analytically solve on the coarsest mesh
    data(1).err = data(1).A \ data(1).r;

    % interpolate upwards
    for i = 1:(nref - 1)
        % project back up
        data(i + 1).err = data(i + 1).err + data(i).T * data(i).err;

        % perform G-S iterations to improve projected-up error
        [data(i + 1).err] = gauss_seidel(data(i + 1).A, data(i + 1).r, data(i + 1).err
            ↪ , vup);
    end

    % at the last point, the equation is u_h = u_h + e_h;
    data(nref + 1).u = data(nref + 1).u + data(nref).T * data(nref).err;

    % smooth u_h
    [data(nref + 1).u] = gauss_seidel(data(nref + 1).A, data(nref + 1).b, data(nref +
        ↪ 1).u, vup);

    % compute the residual for this newest iterate
    loop_iter = loop_iter + 1;
    data(nref + 1).r = data(nref + 1).b - data(nref + 1).A * data(nref + 1).u;
    res(loop_iter) = max(abs(data(nref + 1).r));
```

```
end

sol = data(nref + 1).u;
end
```