APMA-929 FINAL ASSIGNMENT: RECOVERING SURFACES FROM GRADIENT VALUES

COLIN MACDONALD

Abstract. We implement a finite element method (FEM) to recover a surface from its gradient values.

1. Introduction

Given $p(x, y)$ and $q(x, y)$ which approximate u_x and u_y respectively on a planar domain Ω, we wish to solve the minimization problem

$$\min_{u \in \mathcal{H}^1(\Omega)} \int_\Omega \left((u_x - p)^2 + (u_y - q)^2 \right) \mathrm{d}x\mathrm{d}y, \tag{1}$$

and thus recover the surface u. This problem has an equivalent *variational form*, namely, to find $u \in H^1(\Omega)$ such that

$$a(u, v) = L(v), \quad \text{for all } v \in \mathcal{H}^1(\Omega), \tag{2a}$$

where the *energy inner product* $a(\cdot, \cdot)$ and $L(v)$ are

$$a(u, v) = \int_\Omega u_x v_x + u_y v_y \mathrm{d}x\mathrm{d}y, \tag{2b}$$

$$L(v) = \int_\Omega p v_x + q v_y \mathrm{d}x\mathrm{d}y. \tag{2c}$$

2. Methodology

2.1. Galerkin form.

Consider a finite dimensional subspace $V_h \subset \mathcal{H}^1(\Omega)$ such that any $v \in V_h$ can be written as a linear combination of m basis functions $\phi^1, \phi^2, \ldots, \phi^m \in V_h$. The *Galerkin* approximation of (2) is thus to find $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \ldots, \alpha_m]$ such that

$$u_h = \sum_{i=1}^{m} \alpha_i \phi^i, \tag{3a}$$

satisfies

$$a(u_h, \phi^j) = L(\phi^j) \quad \text{for } j = 1, \ldots, m,$$

or, by linearity of the energy inner product,

$$\sum_{i=1}^{m} \alpha_i a(\phi^i, \phi^j) = L(\phi^j), \quad \text{for } j = 1, \ldots, m. \tag{3b}$$

Furthermore, (3) can be expressed as the matrix system

$$\mathsf{A}\boldsymbol{\alpha} = \boldsymbol{b}, \tag{4a}$$

where the entries of the *stiffness matrix* A and *load vector* $\boldsymbol{b}$ are

$$a_{ij} = a(\phi^i, \phi^j) = \int_\Omega \phi_x^i \phi_x^j + \phi_y^i \phi_y^j \mathrm{d}x\mathrm{d}y, \tag{4b}$$

$$b_j = L(\phi^j) = \int_\Omega p\phi_x^j + q\phi_y^j \mathrm{d}x\mathrm{d}y. \tag{4c}$$

For this project, we will use piecewise linear functions over a triangulation $T_h = \left\{\bigcup_{l=1}^{k} K_l\right\}$ of our domain Ω. That is,

$$V_h = \left\{v : v|_{K_l} \text{ is linear and } v \in C(\Omega)\right\}, \tag{5}$$

and our basis elements ϕ will consist of "hat" functions—piecewise linear functions which are unity at node i and zero at all other nodes (see Figure 1).

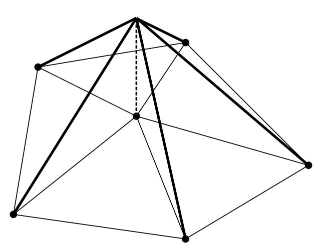

FIGURE 1. A "hat" function over a subset of a triangulation.

2.2. **Assembling the stiffness matrix.** To assemble A and $\boldsymbol{b}$ using (4b) and (4c), we iterate over T_h and compute the contributions of each triangle K_l to each entry of A and $\boldsymbol{b}$. Considering (4b) restricted to K_l, note that ϕ_x^i and ϕ_y^i are constant and thus the contribution from K_l to A is

$$\left(\phi_x^i \phi_x^j + \phi_y^i \phi_y^j\right) \int_{K_l} \mathrm{d}x\mathrm{d}y = \boxed{\operatorname{area}(K_l)\left(\phi_x^i \phi_x^j + \phi_y^i \phi_y^j\right)}. \tag{6}$$

The contribution to $\boldsymbol{b}$ is similarly thus

$$\phi_x^j \int_{K_l} p\mathrm{d}x\mathrm{d}y + \phi_y^j \int_{K_l} q\mathrm{d}x\mathrm{d}y. \tag{7}$$

However, we only know the values of p and q at the nodes so we approximate the integrals in (7) with quadrature rules to obtain

$$\boxed{\mathrm{area}(K_l)\left(\phi_x^j \frac{p_{l_1}+p_{l_2}+p_{l_3}}{3} + \phi_y^j \frac{q_{l_1}+q_{l_2}+q_{l_3}}{3}\right)}, \tag{8}$$

where p_{l_1}, p_{l_2}, and p_{l_3} are the values of p at the three corners l_1, l_2, and l_3 of K_l (and similarly for q).

Due to the support of the basis functions ϕ, it is not necessary to evaluate (6) and (8) for all possible i and j. In our case, we only need to evaluate (6) for all nine cases of $i, j \in \{l_1, l_2, l_3\}$ (which further reduces to six cases by symmetry) and (8) for the three cases $j = l_1, l_2, l_3$.

2.3. **Non-uniqueness of the solution.** As might be expected from (1) and (2), solutions of this problem are only unique up to an additive constant. Indeed, if the matrix is formed as described above, it will be singular[1]. To work around this problem, we choose node n_1 and remove the first row and first column from A and the first entry from $\boldsymbol{b}$ to form $\tilde{\mathsf{A}}$ and $\tilde{\boldsymbol{b}}$. The $m-1 \times m-1$ problem

$$\tilde{\mathsf{A}}\tilde{\boldsymbol{\alpha}} = \tilde{\boldsymbol{b}},$$

now has a unique solution for $\tilde{\boldsymbol{\alpha}}$. We then recover $\boldsymbol{\alpha} = [0, \tilde{\boldsymbol{\alpha}}]$.

Note that this process is *much* easier to implement than—and effectively the same as—using the subspace

$$\tilde{V_h} = \{v : v|_{K_l} \text{ is linear, } v \in C(\Omega), \text{ and } v(n_1) = 0\},$$

instead of V_h and discarding the ϕ^1 basis function.

2.4. **Implementation.** We assemble a Matlab implementation to construct A and $\boldsymbol{b}$ and solve the system for $\boldsymbol{\alpha}$. The code can be passed a mesh in the form of a list of points and triangles. If the triangles are omitted, the “delaunay” routine is used to construct a convex mesh using the given points. Solving the system (4) is accomplised with Matlab’s backslash operator (i.e., a QR routine).

3. Results

3.1. **Convergence test.** We perform a convergence test on a unit square domain, $\Omega = [0,1] \times [0,1]$ with a regular grid of equal-sized triangles and $u_{\text{ex}} = \frac{1}{2}\sin 2\pi x + y^2$. Figure 2 shows an example of this triangulation and a typical solution u_h.

[1]Or nearly singular due to roundoff error.

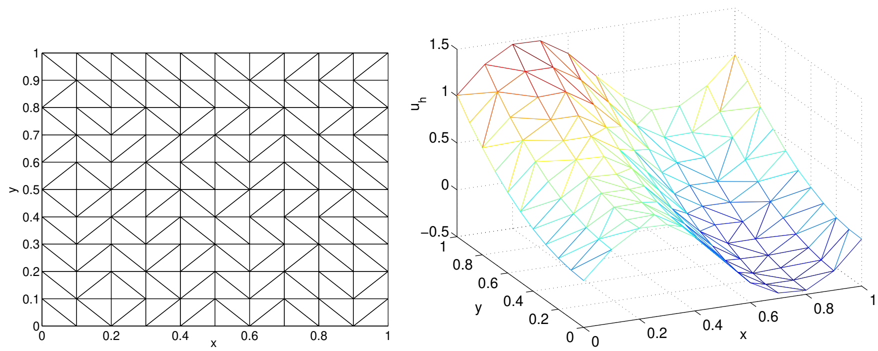

FIGURE 2. Triangulation and solution u_h for the convergence test problem.

h	$\|\boldsymbol{u}_{\text{ex}} - \boldsymbol{u}_h\|_\infty$	$\|\boldsymbol{u}_{\text{ex}} - \boldsymbol{u}_h\|_2$	$\|u_{\text{ex}} - u_h\|_{L_2}$
0.3536	0.1556	0.3928	0.1291
0.1768	0.0486	0.1880	0.0372
0.0884	0.0111	0.0811	0.0091
0.0442	0.0035	0.0400	0.0023
0.0221	0.0009	0.0196	
order	1.88	1.09	1.95

TABLE 1. Results of a convergence test. Note $\|\cdot\|_\infty$ and $\|\cdot\|_2$ refer to *vector* norms and that $\|\cdot\|_{L_2}$ is computed by quadrature.

3.2. **Canucks example.** In this example, we compute the solution using $p = 3x^2$ and $q = 3y^2$ corresponding to $u_{\text{ex}} = x^3 + y^3$ on the complicated, non-convex, beaten-by-Calgary domain[2] seen in Figure 3. For this mesh, the errors where $\|\boldsymbol{u}_{\text{ex}} - \boldsymbol{u}_h\|_\infty = 0.0178$ and $\|\boldsymbol{u}_{\text{ex}} - \boldsymbol{u}_h\|_2 = 0.1450$. When the mesh was refined (each triangle replaced with 4 new triangles), these errors became 0.0052 and 0.0753 respectively.

4. Conclusions

We have successfully implemented a program to recover surface values from gradient values given at discrete points on general convex domains. Additionally, non-convex domains can be handled if a mesh is provided. The program provides second-order results at least for smooth problems on regular domains.

[2]This geometry and mesh were created using Matlab's PDE toolbox.

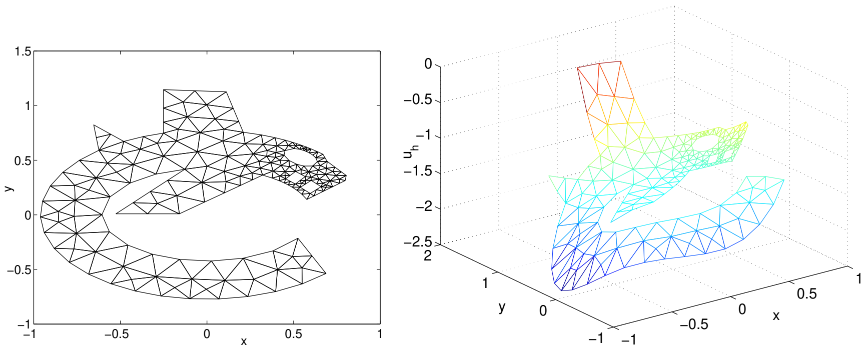

FIGURE 3. Triangulation and solution u_h for the Canucks test problem.

REFERENCES

[Joh87] Claes Johnson. *Numerical solution of partial differential equations by the finite element method*. Cambridge University Press, Cambridge, 1987.

[SL99] Murray R. Spiegel and John Liu. *Mathematical Handbook of Formulas and Tables*. Schaum's Outline series. McGraw-Hill, second edition, 1999.

APPENDIX A. SOURCE CODE

sfg.m — main routine.

```
function [alpha, tri] = sfg(x, y, p, q, tri, offset)
% SFG  Recover a surface from gradient values
%    [U] = SFG(X,Y,P,Q) returns the values of U at the points in the
%    vectors x, y such that U matches the component gradient values
%    given in the vectors p, q.  For uniqueness, U(X(1),Y(1)) will be
%    0.  A mesh will be generated inside the convex hull of x, y.
%
%    [U,TRI] = SFG(X,Y,P,Q) additionally returns the triangulation
%    as a three-column matrix where each row represents the nodes
%    of a triangle.
%
%    [U,TRI] = SFG(X,Y,P,Q,TRI) specify a 3-column matrix where each
%    row represents the nodes of a triangle.  This mesh does NOT have
%    to be convex.
%
%    [U,TRI] = SFG(X,Y,P,Q,TRI,OFFSET) specify a value for
%    U(X(1),Y(1)).  Set TRI to [] if you want SFG to perform a
%    triangulation.

  plots = 1;  % make some plots

  if (nargin < 4)
    error('not enough input arguments');
  end
  if (nargin < 5) | (isempty(tri))
    tri = delaunay(x,y);
  end
  if (nargin < 6)
    offset = 0;
  end
```

```
nnode = length(x);
[ntri, temp] = size(tri);

% build area and centroid data
areas = zeros(ntri,1);
centroids = zeros(ntri,2);
for n = 1:ntri
  a = [x(tri(n,1)) y(tri(n,1))];
  b = [x(tri(n,2)) y(tri(n,2))];
  c = [x(tri(n,3)) y(tri(n,3))];
  areas(n) = triarea(a,b,c);
  centroids(n,:) = (a+b+c)/3;
end

if (plots)
  figure(10);  clf;  subplot(2,2,1);
  plot(x,y,'ro');  hold on;
  trimesh(tri, x, y, 'color', [0,0,0]);
  xlabel('x');  ylabel('y');
end

% allocate a sparse matrix
A = spalloc(nnode,nnode,7*nnode);
b = zeros(nnode, 1);

% loop over all triangles
for k = 1:ntri
  % indicies of nodes
  ni1 = tri(k,1);
  ni2 = tri(k,2);
  ni3 = tri(k,3);
  % location of nodes
  n1 = [x(ni1) y(ni1)];
  n2 = [x(ni2) y(ni2)];
  n3 = [x(ni3) y(ni3)];

  % coefficients of unit linear fcns on triangle
  [phi1x, phi1y, temp] = lineartri(n1,n2,n3, 1, 0, 0);
  [phi2x, phi2y, temp] = lineartri(n1,n2,n3, 0, 1, 0);
  [phi3x, phi3y, temp] = lineartri(n1,n2,n3, 0, 0, 1);

  A(ni1,ni1) = A(ni1,ni1) + areas(k)*(phi1x*phi1x + phi1y*phi1y);
  A(ni1,ni2) = A(ni1,ni2) + areas(k)*(phi1x*phi2x + phi1y*phi2y);
  A(ni2,ni1) = A(ni1,ni2);
  A(ni1,ni3) = A(ni1,ni3) + areas(k)*(phi1x*phi3x + phi1y*phi3y);
  A(ni3,ni1) = A(ni1,ni3);
  A(ni2,ni2) = A(ni2,ni2) + areas(k)*(phi2x*phi2x + phi2y*phi2y);
  A(ni2,ni3) = A(ni2,ni3) + areas(k)*(phi2x*phi3x + phi2y*phi3y);
  A(ni3,ni2) = A(ni2,ni3);
  A(ni3,ni3) = A(ni3,ni3) + areas(k)*(phi3x*phi3x + phi3y*phi3y);

  pav = (p(ni1) + p(ni2) + p(ni3))/3;
  qav = (q(ni1) + q(ni2) + q(ni3))/3;
  b(ni1) = b(ni1) + areas(k)*(pav*phi1x + qav*phi1y);
  b(ni2) = b(ni2) + areas(k)*(pav*phi2x + qav*phi2y);
  b(ni3) = b(ni3) + areas(k)*(pav*phi3x + qav*phi3y);
end

if (plots)
  figure(10);  subplot(2,2,3);
  trimesh(tri, x, y, b);
  xlabel('x');  ylabel('y');  zlabel('b');
```

```
  end

  % crop the first node
  A = A(2:end,2:end);
  b = b(2:end);

  alpha= A\b;
  alpha = [0; alpha] + offset;

  if (plots)
    figure(10);  subplot(2,2,2);
    spy(A);
    title(['\kappa = ' num2str(condest(A))]);

    figure(10);  subplot(2,2,4);
    trimesh(tri, x, y, alpha);
    xlabel('x');  ylabel('y');  zlabel('u_h');
  end
```

lineartri.m — helper function, fits linear function over triangle.

```
function [cx,cy,cc] = lineartri(p,q,r,a,b,c)
%LINEAR_ON_TRI  return coefficients of a linear fcn on a triangle
%  the function has value a at p, b at q and c at r

  a1 = q(2) - r(2);
  b1 = -q(1) + r(1);
  c1 = -a1*r(1) - b1*r(2);
  d1 = 1./(a1*p(1) + b1*p(2) + c1);
  a1 = d1*a1;  b1 = d1*b1;  c1 = d1*c1;


  a2 = r(2) - p(2);
  b2 = -r(1) + p(1);
  c2 = -a2*r(1) - b2*r(2);
  d2 = 1./(a2*q(1) + b2*q(2) + c2);
  a2 = d2*a2;  b2 = d2*b2;  c2 = d2*c2;

  a3 = p(2) - q(2);
  b3 = -p(1) + q(1);
  c3 = -a3*p(1) - b3*p(2);
  d3 = 1./(a3*r(1) + b3*r(2) + c3);
  a3 = d3*a3;  b3 = d3*b3;  c3 = d3*c3;

  cx = a*a1 + b*a2 + c*a3;
  cy = a*b1 + b*b2 + c*b3;
  cc = a*c1 + b*c2 + c*c3;
```

triarea.m — helper function, returns triangle area.

```
function area = triarea(p,q,r)
%TRIAREA  Calculate the area of a triangle with nodes at p, q, r

% TODO: use det formula instead...

  a = norm(p-q);
  b = norm(q-r);
  c = norm(p-r);
  s = (a+b+c)/2;
  area = sqrt(s*(s-a)*(s-b)*(s-c));
```

convtest.m — convergence test.

```
%CONVTEST  test the convergence of sfg.m

N = 4;
dx = 1/N;
[xg,yg] = meshgrid(0:dx:1, 0:dx:1);
x = xg(:);   y = yg(:);

% Linear
%p = 2*ones(size(xg));
%q = 4*ones(size(yg));
%uex = 2*xg + 4*yg;
% Quadratic
%p = 2*xg;
%q = 2*yg;
%uex = xg.^2 + yg.^2;
p = pi*cos(2*pi*x);
q = 2*y;
uex = (1/2)*sin(2*pi*x) + y.^2;


[alpha, tri] = sfg(x, y, p, q, [], uex(1));

figure(1); clf;
trimesh(tri, xg, yg, alpha);
xlabel('x');  ylabel('y');  zlabel('u_h');

err = abs(alpha - uex(:));
figure(2); clf;
trimesh(tri, xg, yg, err);
xlabel('x');  ylabel('y');  zlabel('error');
merr = max(err)
n2err = norm(err)
h = dx*sqrt(2)

[ntri, temp] = size(tri)
sum1 = 0;
for k = 1:ntri
    disp(['processing triangle ' num2str(k) ' of ' num2str((N^2*2))]);
    % indicies of nodes
    ni1 = tri(k,1);
    ni2 = tri(k,2);
    ni3 = tri(k,3);
    % location of nodes
    n1 = [x(ni1) y(ni1)];
    n2 = [x(ni2) y(ni2)];
    n3 = [x(ni3) y(ni3)];

    [cx,cy,cc] = lineartri(n1,n2,n3,alpha(ni1),alpha(ni2),alpha(ni3));

    fstr = ['((1/2)*sin(2*pi*x) + y.^2 - ' num2str(cx,16) '*x - ' num2str(cy,16) '*y - ' num2str(cc,16)
    f = inline(fstr,'x','y');
    sum1 = sum1 + dblquadtri(f,n1,n2,n3,1e-12);
end

L2err = sqrt(sum1)

%merrs = [merrs merr]
%n2errs = [n2errs n2err]
%hs = [hs h]
```

canucks.m — Canucks example.

```
%CANUCKS  example for sfg.m
```

```
% this comes from the pdetool
% open the file canucks-pdetool-file.m with pdetool to edit
load canucks-mesh-coarse
tri = t(1:3,:)';
x = p(1,:)';
y = p(2,:)';

%uex = exp(-x.^2-y.^2) .* sin(9*x.^2+9*y.^2);
%p = -2.*x.*exp(-x.^2-y.^2).*sin(9*x.^2+9*y.^2)+18*exp(-x.^2-y.^2).*cos(9*x.^2+9*y.^2).*x;
%q = -2.*y.*exp(-x.^2-y.^2).*sin(9*x.^2+9*y.^2)+18*exp(-x.^2-y.^2).*cos(9*x.^2+9*y.^2).*y;
p = 3*x.^2;
q = 3*y.^2;
uex = x.^3 + y.^3;

[alpha, tri] = sfg(x, y, p, q, tri, uex(1));

figure(1); clf;
trimesh(tri, x, y, alpha);
xlabel('x');  ylabel('y');  zlabel('u_h');

err = abs(alpha - uex(:));
figure(2); clf;
trimesh(tri, x, y, err);
xlabel('x');  ylabel('y');  zlabel('error');
merr = max(err)
n2err = norm(err)
```

dblquadtri.m — quadrature over triangle, used by convtest.m.

```
function an = dblquadtri(f,p,q,r,tol,quadf)
% TODO: implement extra parametres to f (p1, p2, p3...)

  if (nargin < 5)
    tol = [];
  end
  if (nargin < 6)
    quadf = [];
  end

  debugplot = 0;

  t = (p+q+r)/3;
  t1 = t(1);
  t2 = t(2);

  ymin = min([p(2),q(2),r(2)]);
  ymax = max([p(2),q(2),r(2)]);
  xmin = min([p(1),q(1),r(1)]);
  xmax = max([p(1),q(1),r(1)]);

  a1 = p(2) - q(2);
  b1 = -p(1) + q(1);
  c1 = -a1*p(1) - b1*p(2);
  if(a1*t1 + b1*t2 + c1 > 0)
    ineq1 = 1;
  else
    ineq1 = -1;
  end

  a2 = q(2) - r(2);
  b2 = -q(1) + r(1);
  c2 = -a2*r(1) - b2*r(2);
```

```
if(a2*t1 + b2*t2 + c2 > 0)
  ineq2 = 1;
else
  ineq2 = -1;
end

a3 = r(2) - p(2);
b3 = -r(1) + p(1);
c3 = -a3*r(1) - b3*r(2);
if(a3*t1 + b3*t2 + c3 > 0)
  ineq3 = 1;
else
  ineq3 = -1;
end

if (debugplot)
  figure(99); clf; hold on;
  plot(p(1), p(2), 'rx');
  plot(q(1), q(2), 'rx');
  plot(r(1), r(2), 'rx');
  axis([xmin xmax ymin ymax]);

  xt = [xmin xmax];
  plot(xt,(-a1*xt-c1)/b1, 'b:');
  plot(xt,(-a2*xt-c2)/b2, 'b:');
  plot(xt,(-a3*xt-c3)/b3, 'b:');

  dx = (xmax-xmin)/100
  dy = (ymax-ymin)/100
  [xx,yy] = meshgrid(xmin:dx:xmax, ymin:dy:ymax);

  z = ineq1*(a1*xx + b1*yy + c1) >= 0 & ...
      ineq2*(a2*xx + b2*yy + c2) >= 0 & ...
      ineq3*(a3*xx + b3*yy + c3) >= 0;

  figure(98); clf;
  pcolor(xx,yy,z);
  xlabel('x');  ylabel('y');
end

g = inline('f(x,y).*(ineq1*(a1*x+b1*y+c1)>=0&ineq2*(a2*x+b2*y+c2)>=0&ineq3*(a3*x+b3*y+c3)>=0)',...
           'x','y','f','a1','b1','c1','a2','b2','c2','a3','b3','c3',...
           'ineq1','ineq2','ineq3');

an = dblquad(g, xmin, xmax, ymin, ymax, tol, quadf, ...
             f,a1,b1,c1,a2,b2,c2,a3,b3,c3,ineq1,ineq2,ineq3);
```