Poisson's equation by the FEM using a MATLAB mesh generator

The finite element method [1] applied to the Poisson problem

$$-\triangle u = f \quad \text{on } D, \qquad u = 0 \quad \text{on } \partial D, \tag{1}$$

on a domain $D \subset \mathbb{R}^2$ with a given triangulation (mesh) and with a chosen finite element space based upon this mesh produces linear equations

$$Av = b.$$

Figure 1 shows a particular triangulation for the unit disc $D_1 = \{(x, y) : x^2 + y^2 < 1\}$. There are five interior nodes corresponding to unknowns. In this note I will give enough details to set up and solve the 5×5 matrix problem which results when we choose piecewise-linear finite elements. More generally, I'll give a short MATLAB code which works with Persson and Strangs' one page mesh generator `distmesh2d.m` [2]. Thus I will approximately solve Poisson's equation on quite general domains in less than two pages of MATLAB.

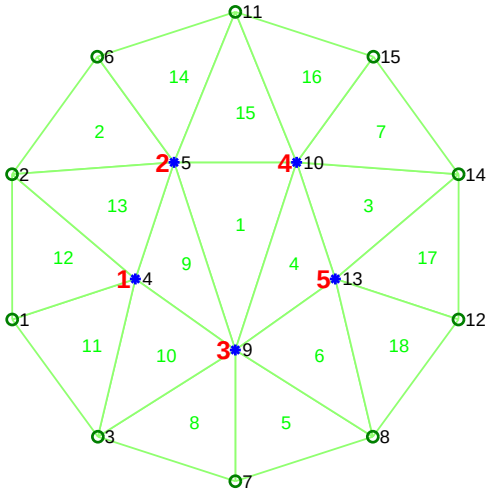

FIGURE 1. A simple finite element mesh on the unit disc with 18 triangles, 15 vertices and 5 interior vertices (i.e. locations of the unknowns). The interior vertices are numbered in bold.

[1]This version has one core change, namely, a completely different and far superior choice of reference triangle. (Thanks to David Maxwell for guidance!) The function `poissonv2.m` is changed appropriately. In addition, `poissonv2.m` does not itself include a call to `distmesh2d.m` [2]; see the examples.

Suppose there are N interior nodes $p_j = (x_j, y_j)$. At p_j the unknown u_j approximates $u(x_j, y_j)$. If φ_j is the hat function [1, page 29] corresponding to node p_j then $u(x, y) \approx u_h(x, y) = \sum_{j=1}^{N} u_j \varphi_j(x, y)$. We seek the column vector $v = (u_1, u_2, \ldots, u_N)^\top$ such that $Av = b$ where the entries of A, b are given by

$$a_{jk} = \int_D \nabla\varphi_j \cdot \nabla\varphi_k \qquad \text{and} \qquad b_j = \int_D f\varphi_j.$$

In fact, I construct the matrix A by going through the triangles in some order; such an order is given in figure 1. Write "$j \in T$" if node p_j is a corner of a triangle T. For each triangle T we can compute the contribution to a_{jk}, b_j because

$$a_{jk} = \sum_{\{T \text{ such that } j\in T \text{ and } k\in T\}} \int_T \nabla\varphi_j \cdot \nabla\varphi_k,$$

and, similarly,

$$b_j = \sum_{\{T \text{ such that } j\in T\}} \int_T f\varphi_j. \tag{2}$$

The contributions to A associated to a given T can be thought of as 3×3 matrix, the *element stiffness matrix* [1, equation (1.27)] for T.

To compute the element stiffness matrix it is useful, though not essential, to refer the whole problem to a standard reference triangle. In fact, if the original triangle T lies in the (x_1, x_2) plane then our reference triangle will be $R = \{(\xi_1, \xi_2) : \xi_1 + \xi_2 \le 1, \xi_1, \xi_2 \ge 0\}$ in a new (ξ_1, ξ_2) plane. Denote $x = (x_1, x_2)$ and $\xi = (\xi_1, \xi_2)$. Suppose $x^j = (x_1^j, x_2^j)$, $x^k = (x_1^k, x_2^k)$, $x^l = (x_1^l, x_2^l)$ are the corners of T. The affine map

$$\Phi(\xi) = \big((x_1^k - x_1^j)\xi_1 + (x_1^l - x_1^j)\xi_2 + x_1^j, (x_2^k - x_2^j)\xi_1 + (x_2^l - x_2^j)\xi_2 + x_2^j, \big)$$

sends R to T. Note that Φ sends

$$(0, 0) \to x^j, \quad (1, 0) \to x^k, \quad (0, 1) \to x^l.$$

We want to integrate over T by changing variables to an integral over R. For example,

$$\int_T f(x)\varphi_j(x)\, dx = \int_R f(\xi)\varphi_j(\xi)\, |J|\, d\xi$$

where $J = d\Phi$ is the differential of the change of variables, a 2×2 matrix, and $|J| = |\det(J)|$ is the Jacobian determinant:

$$|J| = \left|\det \begin{pmatrix} x_1^k - x_1^j & x_1^l - x_1^j \\ x_2^k - x_2^j & x_2^l - x_2^j \end{pmatrix}\right| = |(x_1^k - x_1^j)(x_2^l - x_2^j) - (x_1^l - x_1^j)(x_2^k - x_2^j)|.$$

In fact, to do the just-mentioned integral numerically I choose to also approximate f by a linear function on T, that is,

$$f \approx f(x^j)\varphi_j + f(x^k)\varphi_k + f(x^l)\varphi_l$$

on T so

$$b_j = \int_T f(x)\varphi_j(x)\,dx \approx |J| \begin{pmatrix} f(x^j) & f(x^k) & f(x^l) \end{pmatrix} \begin{pmatrix} \int_R \varphi_j^2\,d\xi \\ \int_R \varphi_k\varphi_j\,d\xi \\ \int_R \varphi_l\varphi_j\,d\xi \end{pmatrix}$$

Because $\varphi_j(\xi) = 1 - \xi_1 - \xi_2$, $\varphi_k(\xi) = \xi_1$, and $\varphi_k(\xi) = \xi_2$, we may complete this job by doing the following integrals:

$$\int_R \varphi_j^2\,d\xi = \int_R \varphi_k^2\,d\xi = \int_R \varphi_l^2\,d\xi = \frac{1}{12}, \qquad \int_R \varphi_j\varphi_k\,d\xi = \int_R \varphi_k\varphi_l\,d\xi = \int_R \varphi_l\varphi_j\,d\xi = \frac{1}{24}.$$

On the other hand we need to compute the contributions to the stiffness matrix. Using the summation convention,

$$\int_T \nabla\varphi_j \cdot \nabla\varphi_k\,dx = \int_T \frac{\partial\varphi_j}{\partial x_s}\frac{\partial\varphi_k}{\partial x_s}\,dx = \int_R \frac{\partial\varphi_j}{\partial\xi_p}\frac{\partial\xi_p}{\partial x_s}\frac{\partial\varphi_k}{\partial\xi_q}\frac{\partial\xi_q}{\partial x_s}\,|J|\,d\xi.$$

But

$$\frac{\partial\xi_p}{\partial x_s}\frac{\partial\xi_q}{\partial x_s} = \left[J^{-1}(J^{-1})^\top\right]_{pq} = \left[(J^\top J)^{-1}\right]_{pq},$$

and $\frac{\partial\varphi_j}{\partial\xi_p} = (-1,-1)$, $\frac{\partial\varphi_k}{\partial\xi_p} = (1,0)$, $\frac{\partial\varphi_l}{\partial\xi_p} = (0,1)$. Letting $Q = (J^\top J)^{-1}$ and noting that the area of R is $\frac{1}{2}$, we have, for the a_{jk} contribution,

$$\int_T \nabla\varphi_j \cdot \nabla\varphi_k\,dx = \frac{1}{2}\,|J|\,\frac{\partial\varphi_j}{\partial\xi_p}\,Q\left(\frac{\partial\varphi_k}{\partial\xi_p}\right)^\top.$$

That is, Q is the matrix of the quadratic form we need.

I have written such a program, namely `poissonv2.m` which appears on page 4. I now illustrate it by examples. First consider a problem with a known solution.

Example 1. *Suppose $D = D_1$ is the unit disc and suppose $f(x,y) = 4$. It is easy to check that $u(x,y) = 1 - x^2 - y^2$ is an exact solution to (1). Note that f is constant and thus the piecewise linear approximation involved in the load integrals is actually exact. To use* `distmesh2d.m` *and the* `poissonv2.m` *I first describe the disc by the signed distance function $d(x,y) = \sqrt{x^2+y^2} - 1$. I choose the mesh to be fine enough so that the typical triangle has linear dimension $h_0 = 0.5$ and get the mesh in figure 1. Then:*

```
>> f=inline('4','p'); fd=inline('sqrt(sum(p.^2,2))-1','p');
>> [p,t]=distmesh2d(fd,@huniform,0.5,[-1,-1;1,1],[]);
>> [uh,in]=poissonv2(f,fd,0.5,p,t);
```

The arrays `p` *and* `t` *are the coordinates of the points of the triangulation and the indices of corners of triangles, respectively. The output array* `in` *tells me which of the nodes are interior nodes. The array* `uh` *is the approximate solution at all nodes. I find that the approximate solution at the five interior points (see figure 1 for the order of the points) is*

```
>> uh(in>0)'
ans =
      0.76061       0.81055       0.84264       0.81055       0.76061
```

The maximum error is

```
>> u = 1-sum(p.^2,2);  err=max(abs(uh-u))
err =
     0.038957
```

and thus we have about a digit-and-a-half of accuracy at the nodes.

Exercise. *Run the following. What PDE problem is approximately solved by* `wh`*?*

```
>> f=inline('-pi^2*sin(pi*p(:,1)).*(1-p(:,2))','p');
>> fd=inline('drectangle(p,0,1,0,1)','p');
>> [p,t]=distmesh2d(fd,@huniform,0.1,[0,0;1,1],[0,0;0,1;1,0;1,1]);
>> [uh,in]=poissonv2(f,fd,0.1,p,t);
>> wh=uh+sin(pi*p(:,1)).*(1-p(:,2));
>> trimesh(t,p(:,1),p(:,2),wh), axis([-.2 1.2 -.2 1.2 0 1])
```

Evaluate the accuracy of the result by exactly solving the same problem a different way (i.e. a standard exact method).

Example 2. *Let's do a harder example than the previous, just to show off. Suppose D is a rectangular region with an off-center hole removed:*

$$D = \left\{(x,y) : -4 < x < 2, \quad -2 < y < 2, \quad \textit{and} \quad x^2 + y^2 > 1\right\}.$$

Suppose f is a function which is concentrated near $(x_0, y_0) = (-3, 1)$:

$$f(x,y) = e^{-4((x+3)^2+(x-1)^2)}$$

The commands necessary to approximately solve $-\triangle u = f$ with Dirichlet boundary conditions $u = 0$ on ∂D, and to display the answer, amount to four lines:

```
>> f=inline('exp(-4*((p(:,1)+3).^2+(p(:,2)-1).^2))','p');
>> fd=inline('ddiff(drectangle(p,-4,2,-2,2),dcircle(p,0,0,1))','p');
>> [p,t]=distmesh2d(fd,@huniform,0.2,[-4,-2;2,2],[-4,-2;-4,2;2,-2;2,2]);
>> [uh,in]=poissonv2(f,fd,0.2,p,t);  axis([-4.5 2.5 -2.5 2.5 0 .14])
```

The result is shown in figure 2. Because f is so concentrated around $(-3, 1)$, the result u_h is nearly an approximation of (a multiple of) the Green's function $G = G_{(x_0,y_0)}$ which solves $-\triangle G = \delta_{(x_0,y_0)}$ with Dirichlet boundary conditions.

Example 3. *Convergence is important. We redo example 1 with a sequence of meshes:*

$$h_0 = 0.5,\ 0.3,\ \ldots, 0.5\left(\frac{3}{5}\right)^5.$$

In figure 3 we see that the maximum error at the nodes goes to zero at rate $O(h_0^2)$. Roughly speaking, this is predicted by theorem 4.3 in [1]*. We also see that meshing by* `distmesh2d.m` *is consistently a lot more time-consuming than the execution of* `poissonv2.m`*.*

Code. The MATLAB function `poissonv2.m` looks like this:

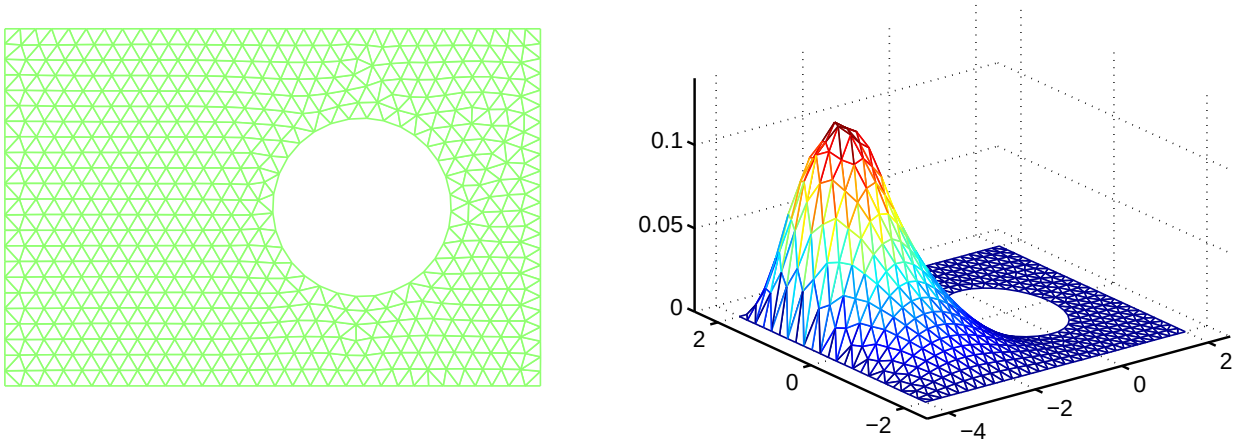

FIGURE 2. **(a)** Mesh for a harder example. **(b)** The approximate solution $u_h(x, y)$. Note f is *much* more concentrated near $(-3, 1)$ than is u_h.

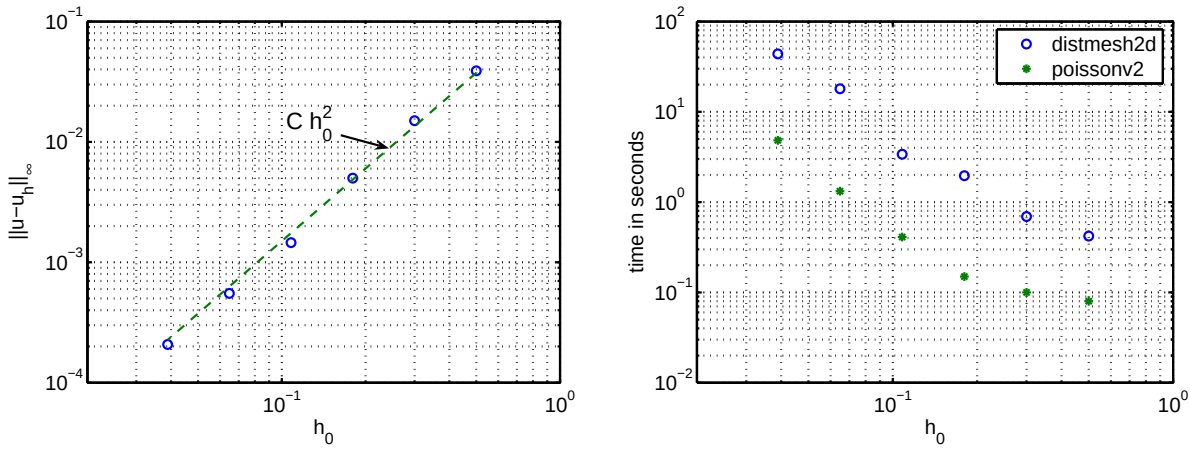

FIGURE 3. **(a)** Maximum error at nodes for the FEM solution of the Poisson equation on the unit disc with $f = 4$. Errors are $O(h_0^2)$. **(b)** Times.

```
function [uh,in]=poissonv2(f,fd,h0,p,t);
%POISSONV2  Solve Poisson's equation on a domain D by the FE method:
%...
%ELB 10/31/04

geps=.001*h0;  ind=(feval(fd,p) < -geps);  % find interior nodes
Np=size(p,1);  N=sum(ind);        % Np=# of nodes;  N=# of interior nodes
in=zeros(Np,1);  in(ind)=(1:N)';  % number the interior nodes
for j=1:Np, ff(j)=feval(f,p(j,:)); end   % eval f once for each node

% loop over triangles to set up stiffness matrix A and load vector b
```

```
A=sparse(N,N);  b=zeros(N,1);
for n=1:size(t,1)
    j=t(n,1); k=t(n,2); l=t(n,3); vj=in(j); vk=in(k); vl=in(l);
    J=[p(k,1)-p(j,1), p(l,1)-p(j,1); p(k,2)-p(j,2), p(l,2)-p(j,2)];
    ar=abs(det(J))/2;  C=ar/12;  Q=inv(J'*J);  fT=[ff(j) ff(k) ff(l)];
    if vj>0
        A(vj,vj)=A(vj,vj)+ar*sum(sum(Q));  b(vj)=b(vj)+C*fT*[2 1 1]'; end
    if vk>0
        A(vk,vk)=A(vk,vk)+ar*Q(1,1);  b(vk)=b(vk)+C*fT*[1 2 1]'; end
    if vl>0
        A(vl,vl)=A(vl,vl)+ar*Q(2,2);  b(vl)=b(vl)+C*fT*[1 1 2]'; end
    if vj*vk>0
        A(vj,vk)=A(vj,vk)-ar*sum(Q(:,1));  A(vk,vj)=A(vj,vk); end
    if vj*vl>0
        A(vj,vl)=A(vj,vl)-ar*sum(Q(:,2));  A(vl,vj)=A(vj,vl); end
    if vk*vl>0
        A(vk,vl)=A(vk,vl)+ar*Q(1,2);  A(vl,vk)=A(vk,vl); end
end

uh=zeros(Np,1);  uh(ind)=A\b;                % solve for FE solution
trimesh(t,p(:,1),p(:,2),uh), axis tight      % display
```

Finally, a note about numerical linear algebra. The system $Ax = b$ which we solve is symmetric, positive-definite, and very sparse [1]. The standard advice for solving such systems is to use the method of *conjugate gradients* [3]. Furthermore, *preconditioning* by *incomplete Cholesky decomposition* is appropriate and recommended. We might, therefore, suppose that the following, or something similar, should be faster than "`A\b`":

```
R=cholinc(A,'0'); uh=pcg(A,b,1e-8,max(2*sqrt(N),20),R',R);
```

From an extremely small amount of experimentation, I note that this more sophisticated method *seems not to be faster*, even for 10^4 nodes. This could be a consequence of my use of MATLAB's `cholinc` and `pcg` commands, but I think that it is actually because MATLAB's "\" is very well optimized, and because, for the cases I tried, it turns out that the mesh-ordering coming from `distmesh2D.m` produces a strongly band-limited matrix. Thus the "`A\b`" method is faster, not to mention easier to type. On the other hand, if you choose someday to implement the finite element method using a compiled language like C++ or Fortran then you would be wise to consider a preconditioned conjugate gradient method.

References

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

[2] P.-O. Persson and G. Strang, *A simple mesh generator in MATLAB*, SIAM Review, 46 (2004), pp. 329–345.

[3] L. N. Trefethen, *Numerical Linear Algebra*, SIAM, Philadelphia, 1997.