Finite Elements for Nonlinear Problems

Computer Session B3

Many important phenomenons are modeled by nonlinear equations that make them hard to analyze and even harder to solve analytically. Fortunately, it is almost always possible to do simulations on nonlinear problems. Moreover, it turns out that the existing numerical algorithms for the solution of nonlinear problems are not very complicated to either derive or implement, although they generally require a lot of computational effort. In this computer session we shall apply finite elements to a nonlinear model problem and study two of the most common techniques for solving the resulting nonlinear system of equations, namely, fixed point, or Picard, iteration, and Newton's method.

Instructions

- ✔ To pass this session you must present your solutions to all problems with this mark to the instructor.
- ☆ This mark indicates that the problem is included in the course's problem demonstration sessions (gives bonus points).

Nonlinear Model Problem

Let us consider the nonlinear model problem

$$-\nabla \cdot (a(u)\nabla u) = f, \quad \text{in } \Omega \tag{1a}$$

$$u = 0, \quad \text{on } \partial\Omega \tag{1b}$$

where a is a given positive function depending on the unknown solution u. As usual f is a given source function, which, for simplicity, we assume does not depend on

u. This is not a severe restriction and after having worked through this computer session the case $f = f(u)$ should be easy to handle for you.

The variational formulation of (1) reads: find $u \in H_0^1$ such that

$$(a(u)\nabla u, \nabla v) = (f, v), \quad \forall v \in H_0^1 \tag{2}$$

As usual a finite element approximation to the variational equation (2) is obtained by replacing H_0^1 by the subspace $V_h \subset H_0^1$ of all continuous piecewise linears, which vanish at the boundary $\partial\Omega$, on a mesh of Ω. The resulting finite element method takes the form: find $u_h \in V_h$ such that

$$(a(u_h)\nabla u_h, \nabla v) = (f, v), \quad \forall v \in V_h \tag{3}$$

Let $\{\varphi_i\}_{i=1}^N$ denote the basis of hat functions for V_h. Then (3) is equivalent to

$$(a(u_h)\nabla u_h, \nabla \varphi_i) = (f, \varphi_i), \quad i = 1, \ldots, N \tag{4}$$

Writing u as the sum

$$u_h = \sum_{j=1}^{N} \xi_j \varphi_j \tag{5}$$

and substituting into (4) gives

$$\sum_{j=1}^{N} \xi_j (a(u_h)\nabla \varphi_j, \nabla \varphi_i) = (f, \varphi_i), \quad i = 1, \ldots, N \tag{6}$$

which is a nonlinear system of N equations for the unknowns ξ_j. In matrix form we write

$$A(\xi)\xi = b \tag{7}$$

where A is the $N \times N$ stiffness matrix with entries $A_{ij} = (a(u_h)\nabla \varphi_j, \nabla \varphi_i)$, and b is the $N \times 1$ load vector with entries $b_i = (f, \varphi_i)$.

Picard Iteration

The simplest technique for solving (7) is Picard iteration (fixed point iteration) defined by the following algorithm.

Algorithm 1 Picard Iteration

1: Choose a small number ϵ and set $\xi^0 = 0$.
2: **for** $k = 1, 2, 3, \ldots$ **do**
3: Solve the linear system

$$A(\xi^{k-1})\xi^k = b$$

4: **if** $\|\xi^k - \xi^{k-1}\| < \epsilon$ **then**
5: Stop.
6: **end if**
7: **end for**

The advantages of Picard iteration is that it gives a robust method, which is almost trivial to implement. The main drawback is that its rate of convergence is slow.

Newton's Method

The most successful techniques for solving nonlinear problems are based on Newton's method.

Perhaps the simplest way of deriving a Newton's method for a nonlinear problem is to linearize the continuous problem before applying the finite element discretization. Let us once again consider the model problem (1) and its variational formulation (2). To derive Newton's method we write u as the sum

$$u = u^0 + \delta \tag{8}$$

where u^0 is a current approximation to u and δ is a (small) correction. Substituting this into (2) gives

$$(a(u^0 + \delta)\nabla(u^0 + \delta), \nabla v) = (f, v) \tag{9}$$

We now linearize this equation around u^0 by first making a Taylor expansion of $a(u)$ around u^0, which yields

$$((a(u^0) + a'_u(u^0)\delta)\nabla(u^0 + \delta), \nabla v) = (f, v) \tag{10}$$

Then, we neglect the term $(a'_u(u^0)\delta\nabla\delta, \nabla v)$, which is quadratic with respect to δ and therefore hopefully small, to obtain

$$(a(u^0)\nabla\delta + a'_u(u^0)\delta\nabla u^0, \nabla v) = (f, v) - (a(u^0)\nabla u^0, \nabla v) \tag{11}$$

This is a linear equation for the correction δ. Replacing H_0^1 with the finite element space $V_h \subset H_0^1$ we end up with the following finite element method: find $\delta_h \in V_h$ such that

$$(a(u^0)\nabla\delta_h + a'_u(u_h^0)\delta_h\nabla u_h^0, \nabla v) = (f, v) - (a(u^0)\nabla u^0, \nabla v), \quad \forall v \in V_h \tag{12}$$

For simplicity, let us tacitly assume that $u^0 = u_h^0 \in V_h$.

Next we note that (12) is equivalent to

$$(a(u_h^0)\nabla\delta_h + a'_u(u_h^0)\delta_h\nabla u_h^0, \nabla\varphi_i) = (f, \varphi_i) - (a(u_h^0)\nabla u_h^0, \nabla\varphi_i), \quad i = 1, \ldots, N \tag{13}$$

Writing $\delta_h = \sum_{j=1}^N d_j\varphi_j$ for some unknown coefficients d_j, and inserting into (13) we get

$$\sum_{j=1}^N d_j(a(u_h^0)\nabla\varphi_j + a'_u(u_h^0)\varphi_j\nabla u_h^0, \nabla\varphi_i) = (f, \varphi_i) - (a(u_h^0)\nabla u_h^0, \nabla\varphi_i), \quad i = 1, \ldots, N \tag{14}$$

which is a linear system for the unknowns d_j. In matrix form we write this

$$Jd = r \tag{15}$$

where J is the $N \times N$ *Jacobian matrix* with entries

$$J_{ij} = (a(u_h^0)\nabla\varphi_j, \nabla\varphi_i) + (a'_u(u_h^0)\varphi_j\nabla u_h^0, \nabla\varphi_i) \tag{16}$$

and r is the $N \times 1$ *residual vector* with entries

$$r_i = (f, \varphi_i) - (a(u_h^0)\nabla u_h^0, \nabla\varphi_i) \tag{17}$$

Thus a better solution approximation to u than u_h^0 can be found by adding δ_h to u_h^0 and iterate. This leads to the following algorithm, which is Newton's method.

Algorithm 2 Newton's Method

1: Choose a small ϵ and set $u_h^0 = 0$.
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Assemble the Jacobian matrix J^k and the residual vector r^k, defined by

$$J_{ij}^k = (a(u_h^k)\nabla\varphi_j, \nabla\varphi_i) + (a_u'(u_h^k)\varphi_j\nabla u_h^k, \nabla\varphi_i)$$
$$r_i^k = (f, \varphi_i) - (a(u_h^0)\nabla u_h^0, \nabla\varphi_i)$$

4: Solve the linear system

$$J^k d^k = r^k$$

5: Set $u^{k+1} = u^k + \delta^k$
6: **if** $\|\delta^k\| < \epsilon$ **then**
7: Stop
8: **end if**
9: **end for**

In Algorithm 2 the iteration is terminated at stage k if the correction δ^k is small. Another alternative would be to stop when the residual r^k is small, since this would indicate that the equation is well satisfied by the computed solution u^k. Both these termination criteria are natural and it does not really matter which one is used.

In practice the assembly of J^k is simplified by using mass lumping. The second term on the right hand side of (16) is approximated by a diagonal matrix with the row sums on the diagonal. Formally, we have $\sum_{j=1}^N \varphi_j = 1$, which gives

$$(a_u'(u_h^k)\varphi_j\nabla u_h^k, \nabla\varphi_i) \approx \delta_{ij}(a_u'(u_h^k)\nabla u_h^k, \nabla\varphi_i) \tag{18}$$

Consequently, if A and b are the stiffness matrix and load vector defined by

$$A_{ij}^{(a)} = (a\nabla\varphi_j, \nabla\varphi_i) \tag{19}$$
$$b_i = (f, \varphi_i) \tag{20}$$

then

$$J^k = \mathrm{diag}(A^{(a_u')}u^k) + A^{(a)} \tag{21}$$

and

$$r^k = b - A^{(a)}u^k \tag{22}$$

Matlab Implementation

Below we present a Matlab code for assembling the Jacobian matrix (21) and the residual vector (22). The computation of the derivative a'_u is performed via numerical differentiation.

```
function [J,r] = jacres(p,e,t,u)
% triangle corner nodes
i=t(1,:); j=t(2,:); k=t(3,:);
% find triangle midpoints
x=(p(1,i)+p(1,j)+p(1,k))/3;
y=(p(2,i)+p(2,j)+p(2,k))/3;
% evaluate u, a, a', and f
uu=(u(i)+u(j)+u(k))/3;
aa=a(uu);
da=(a(uu+1.e-8)-a(uu))/1.e-8;
ff=f(x,y);
% assemble jacobian and residual
[Aa ,unused,b]=assema(p,t,aa',0,ff);
[Ada,unused]  =assema(p,t,da',0,0);
J=diag(Ada*u)+Aa;
r=b-Aa*u;
% enforce B.C.
for i=1:size(e,2)
  n=e(1,i);
  J(n,:)=0;
  J(n,n)=1;  r(n)=0;
end
```

Input to this routine is the usual point, edge, and triangle matrices `p`, `e`, and `t` describing the mesh, and a vector `u` containing the nodal values of the current approximation u^k. The routine calls the build-in subroutine `assema` for the actual assembly of the matrices $A^{(a)}$ and $A^{(a'_u)}$, and the load vector b. The coefficients a and f are assumed to be defined by two separate subroutines `a` and `f` defined elsewhere. Output is the assembled Jacobian matrix J^k and the residual vector r^k.

The boundary condition $u = 0$ is enforced by first assuming that u_h^0 vanish on the boundary $\partial\Omega$ and then construct the Jacobian J and the residual r so that the updates δ_h also vanish on $\partial\Omega$. Since $\delta_h \in V_h$ and since there are no hat functions

on $\partial\Omega$, this amounts to zeroing out the rows of J corresponding to nodes on the boundary. In doing so, one has to prevent J from becoming singular and to make sure that the nodal value of δ_h are indeed zero at all boundary nodes. This is accomplished by setting the diagonal entries of the zeroed rows to unity and the corresponding entries of r to zero. The node numbers of the boundary nodes retrieved form the first row of the edge matrix `e`.

The main routine takes the form:

```
function MyNewtonSolver(geom)
[p,e,t]=initmesh(geom,'hmax',0.1);
u=zeros(size(p,2),1);
for k=1:5
  [J,r]=jacres(p,e,t,u);
  d=J\r;
  u=u+d;
  sprintf('|d|=%f, |r|=%f', norm(d), norm(r))
end
pdesurf(p,t,u)
```

In each iteration we monitor the size of the update δ_h (or, rather its vector of nodal values d) and the residual to supervise the convergence.

Problem 1. ☆ Derive Newton's method for the nonlinear problems $-\Delta u = u+u^3$, $-\Delta u+\sin u = f$, and $-\nabla\cdot((1+u^2)\nabla u) = f$ with homogeneous boundary conditions $u = 0$.

Problem 2. ✔ Implement `MyNewtonSolver` outlined above and solve (1) on the unit square with $a = 0.1 + u^2$ and $f = 1$. Study the influence of the nonlinear term u^2 by also solving (1) with $a = 0.1$ and $f = 1$. Compare the shape of the computed solutions.

Problem 3. ☆ Show that if the Jacobian J is approximated by the stiffness matrix $A^{(a)}$ then Newton's method reduces to fixed point iteration. Also, show that if the problem is linear then Newton's method converges after a single iteration.

Problem 4. ✔ Verify numerically that the assumption $a > 0$ is necessary for existence of solutions by trying to solve (1) with $a = \epsilon + u^2$ with $\epsilon = 1$, 0.1, 0.075, 0.05, and 0.01. You should find that the method breaks down already at $\epsilon = 0.05$. This can be temporarily remedied by using a modified update formula of type $u^{k+1} = u^k + \alpha d^k$, where $0 < \alpha \leq 1$ is a (small) number, typically $\alpha = \epsilon$. This is the

damped Newton method. The introduction of α affects the rate of convergence and it thus takes more iterations to achieve a desired level of accuracy. However, even damping can not prevent the method from breaking down as ϵ becomes really small.

Problem 5. ☆ Derive Newton's method for $-\Delta u = 1+u^3$, and $-\Delta u = f(u)$ with $f(u)$ is a differentiable function depending on u. Assume $u = 0$ on the boundary.

Problem 6. ✔ Modify your code and solve the equation $-\Delta u = e^{-u}$ with $u = 0$ on the boundary using Newton's method. *Hint:* Use the build-in routine `assema` for the assembly of the occurring matrices and vectors. See the help pages for this routine by typing `help assema` at the Matlab prompt.