Introduction to Numerical Simulation (Fall 2003)
Problem Set #6 - Solutions

Problem 1

(a) We are asked to solve the discrete equations:

$$2\psi_i - \psi_{i+1} - \psi_{i-1} + \Delta x^2 (e^{\psi_i} - e^{-\psi_i}) = 0$$

for $i \in [2, ..., N-1]$,

$$2\psi_1 - \psi_2 - (-V) + \Delta x^2 (e^{\psi_1} - e^{-\psi_1}) = 0,$$

and

$$2\psi_N - \psi_{N-1} - V + \Delta x^2 (e^{\psi_N} - e^{-\psi_N}) = 0.$$

where $\Delta x = 1/(N+1)$ (not $1/(N-1)$ - the nodes at $x = 0$ and $x = 1$ are not included in the discretization, but rather enter through the boundary conditions). Define $\boldsymbol{M}$ to be the matrix:

$$\boldsymbol{M} = \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \cdots & 0 \\ & & \ddots & \ddots & \\ & & & \ddots & -1 \\ 0 & 0 & & -1 & 2 \end{bmatrix}$$

or

$$\begin{array}{cccc} & M_{1,1} = 2 & M_{2,1} = -1 & \\ M_{i,i-1} = -1 & M_{i,i} = 2 & M_{i,i+1} = -1 & i \neq 1, i \neq N \\ M_{N,N-1} = -1 & M_{N,N} = 2 & & \end{array}$$

so we can write the discrete equations as:

$$M\psi + \begin{bmatrix} V \\ 0 \\ 0 \\ \vdots \\ 0 \\ -V \end{bmatrix} + 2(\Delta x)^2 \sinh \psi = 0$$

The Jacobian is:

$$J = M + 2(\Delta x)^2 \begin{bmatrix} \cosh \psi_1 & 0 & 0 & \cdots & 0 \\ 0 & \cosh \psi_2 & 0 & \cdots & 0 \\ & & \ddots & \ddots & \\ & & & \ddots & 0 \\ 0 & 0 & & 0 & \cosh \psi_N \end{bmatrix}$$

Since $\cosh x > 1$ for any x and $\Delta x > 0$, $\boldsymbol{J}$ is strictly diagonally dominant for any set of $\psi = \psi_1 \ldots \psi_N$. We assume that by now it is obvious to you that a strictly diagonally dominant matrix is non-singular. Therefore, damped Newton methods should not get stuck around local minima; we suspect that they may be globally convergent.

(b) Some of our sample output from Matlab is below. It is clear from inspecting the norm of $\Delta\psi$ and the norm of the residual that near the solution we have quadratic convergence for $V = 1$ and $V = 20$. Note that $V = 20$ takes much longer to converge, however. When $V = 100$, on the first iteration Newton jumps to a point where the Jacobian is essentially singular. We were very lucky in this case, as the Matlab Newton was able to find its way back, although it did take many iterations. Often when this sort of problem occurs, Newton simply shoots the solution off into hyperspace, never to be seen again. (The code for this problem is the same as part (c), except with the damping removed.)

```
>> V = 1

V =

     1

>> q2
initial F = 1.41421
iter 1 norm dx = 5.52994 norm F = 0.000117637 alpha = 1
iter 2 norm dx = 0.0135486 norm F = 1.50786e-009 alpha = 1
iter 3 norm dx = 2.96714e-007 norm F = 5.09699e-016 alpha = 1
[V = 1] 3 iters, 4 residual evals, 3 solves
>> V = 20

V =

    20

>> q2
initial F = 28.2843
iter 1 norm dx = 110.599 norm F = 55168.2 alpha = 1
iter 2 norm dx = 8.62595 norm F = 20294.6 alpha = 1
iter 3 norm dx = 8.21118 norm F = 7464.97 alpha = 1
iter 4 norm dx = 7.83169 norm F = 2744.94 alpha = 1
iter 5 norm dx = 7.45207 norm F = 1008.26 alpha = 1
iter 6 norm dx = 7.06208 norm F = 369.097 alpha = 1
iter 7 norm dx = 6.65442 norm F = 133.737 alpha = 1
iter 8 norm dx = 6.217 norm F = 47.0612 alpha = 1
iter 9 norm dx = 5.72633 norm F = 15.4108 alpha = 1
iter 10 norm dx = 5.14775 norm F = 4.44848 alpha = 1
```

```
iter 11 norm dx = 4.3901 norm F = 1.0919 alpha = 1
iter 12 norm dx = 3.05901 norm F = 0.17631 alpha = 1
iter 13 norm dx = 1.09178 norm F = 0.00880268 alpha = 1
iter 14 norm dx = 0.0895884 norm F = 3.15307e-005 alpha = 1
iter 15 norm dx = 0.000462205 norm F = 5.32103e-010 alpha = 1
iter 16 norm dx = 1.07687e-008 norm F = 5.26293e-015 alpha = 1
[V = 20] 16 iters, 17 residual evals, 16 solves
>> V = 100

V =

   100

>> q2
initial F = 141.421
iter 1 norm dx = 552.994 norm F = 3.79415e+038 alpha = 1

Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 1.481015e-039.

iter 2 norm dx = 9.6352 norm F = 1.39579e+038 alpha = 1

Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 3.704588e-039.

iter 3 norm dx = 9.55277 norm F = 5.13483e+037 alpha = 1

Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 9.354063e-039.

[...many iterations...]

iter 17 norm dx = 8.64911 norm F = 4.26976e+031 alpha = 1

Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 5.790894e-033.

iter 18 norm dx = 8.58454 norm F = 1.57076e+031 alpha = 1
iter 19 norm dx = 8.51948 norm F = 5.77849e+030 alpha = 1
iter 20 norm dx = 8.45455 norm F = 2.12579e+030 alpha = 1

[...lots of iterations...]

iter 84 norm dx = 2.56064 norm F = 294.984 alpha = 1
iter 85 norm dx = 2.24425 norm F = 84.1995 alpha = 1
```

```
iter 86 norm dx = 1.74842 norm F = 15.5496 alpha = 1
iter 87 norm dx = 1.01034 norm F = 1.055 alpha = 1
iter 88 norm dx = 0.257081 norm F = 0.0223808 alpha = 1
iter 89 norm dx = 0.0118217 norm F = 3.25795e-005 alpha = 1
iter 90 norm dx = 2.06447e-005 norm F = 7.50139e-011 alpha = 1
[V = 100] 90 iters, 91 residual evals, 90 solves
>> diary off
```

(c) Our Matlab code is:

```
N = 100;
RelTol = 1.0e-6;
e = ones(N,1);
itermax = 200;
M = spdiags([-e 2*e -e], -1:1, N, N);
global M;
psi = zeros(N,1);
nres = 0;
nsolves = 0;
F = loadRHS(psi,N,V);
initialF = norm(F);
fprintf(1, 'initial F = %g\n', initialF);
nres = nres + 1;
iter = 0;
while(1)
  iter = iter + 1;
  nFold = norm(F);
  dx = solvesys(psi,N,F);
  nsolves = nsolves + 1;
  alpha = 1.0;
  F = loadRHS(psi + alpha * dx, N, V);
  nres = nres + 1;
  while(norm(F) >= nFold)
    alpha = 0.5 * alpha;
    F = loadRHS(psi + alpha * dx, N, V);
    nres = nres + 1;
  end
  %keyboard
  psi = psi + alpha * dx;
  fprintf(1, 'iter %d norm dx = %g norm F = %g alpha = %g\n', ...
    iter, norm(dx), norm(F), alpha);
  if (norm(dx) < RelTol * norm(psi) & norm(F) < RelTol * initialF)
    break;
  end;
  if (iter > itermax) break; end;
```

```
end;
fprintf(1, '[V = %g] %d iters, %d residual evals, %d solves \n', ...
  V, iter, nres, nsolves);
```

with the auxiliary functions:

```
function y = solvesys(psi,N,F)

global M;
h = 1 / (N+1);
J = M + h*h*2.0*spdiags([cosh(psi)], 0, N, N);
y = -J \ F;
```

and

```
function F = loadRHS(psi,N,V)

global M;
h = 1 / (N+1);
F = M * psi + h*h*2.0*sinh(psi);
F(1) = F(1) + V;
F(N) = F(N) - V;
```

In this code, we have moved the linear system solution completely outside the Newton code. For $V = 1, 20, 100$, this produces the results:

```
>> V = 1

V =

     1

>> q2
initial F = 1.41421
iter 1 norm dx = 5.52994 norm F = 0.000117637 alpha = 1
iter 2 norm dx = 0.0135486 norm F = 1.50786e-009 alpha = 1
iter 3 norm dx = 2.96714e-007 norm F = 5.09699e-016 alpha = 1
[V = 1] 3 iters, 4 residual evals, 3 solves
>> V = 20
```

```
V =

    20

>> q2
initial F = 28.2843
iter 1 norm dx = 110.599 norm F = 12.1519 alpha = 0.5
iter 2 norm dx = 7.73036 norm F = 3.95059 alpha = 0.5
iter 3 norm dx = 6.53317 norm F = 0.555773 alpha = 1
iter 4 norm dx = 4.53291 norm F = 0.0699472 alpha = 1
iter 5 norm dx = 1.62872 norm F = 0.00401589 alpha = 1
iter 6 norm dx = 0.134543 norm F = 1.8394e-005 alpha = 1
iter 7 norm dx = 0.000728218 norm F = 4.46258e-010 alpha = 1
iter 8 norm dx = 1.98401e-008 norm F = 5.73813e-015 alpha = 1
[V = 20] 8 iters, 11 residual evals, 8 solves
>> V = 100

V =

   100

>> q2
initial F = 141.421
iter 1 norm dx = 552.994 norm F = 101.067 alpha = 0.125
iter 2 norm dx = 8.89622 norm F = 22.08 alpha = 0.5
iter 3 norm dx = 7.44634 norm F = 6.30691 alpha = 1
iter 4 norm dx = 6.60395 norm F = 1.84926 alpha = 1
iter 5 norm dx = 5.5548 norm F = 0.451334 alpha = 1
iter 6 norm dx = 3.71235 norm F = 0.0691419 alpha = 1
iter 7 norm dx = 1.1737 norm F = 0.00305243 alpha = 1
iter 8 norm dx = 0.0756883 norm F = 8.14604e-006 alpha = 1
iter 9 norm dx = 0.000252867 norm F = 7.02203e-011 alpha = 1
iter 10 norm dx = 2.58659e-009 norm F = 8.05228e-014 alpha = 1
[V = 100] 10 iters, 15 residual evals, 10 solves
>> diary off
```

Note that not only did we eliminate the nasty problems associated with $V = 100$, we reduced the number of residual evaluations needed for $V = 20$ and $V = 100$. Actually, the damped Newton scheme worked so well that we got convergence in less than 200 residual evaluations for very large V (e.g., $V = 10^{15}$).

Problem 2

Key insights to writing efficient code to solve this problem are:

- If for Newton's method, the product of the Jacobian and some vector (any vector!) is needed, the product can be done using only evaluations of the Newton residual, F.
- Only *one* matrix-vector product is needed per GCR iteration.

If you wrote your GCR and Newton code efficiently, only minor modifications should have been needed for this problem set. Here is our Newton driver:

```
V = 10;
global V;
N = 100;
RelTol = 1.0e-15;
e = ones(N,1);
itermax = 20;
M = spdiags([-e 2*e -e], -1:1, N, N);
global M;
psi = zeros(N,1);
nres = 0;
nsolves = 0;
F = loadRHS_(psi,N);
initialF = norm(F);
fprintf(1, 'initial F = %g\n', initialF);
absb = [];
absdx = [];
nres = nres + 1;
for iter = 1:itermax,
  nFold = norm(F);
  %dx = solvesys(psi,N,F);
  dx = gcr(-F, psi, N, 0.1, 'nlp_matvec');
  %dx = gcr(-F, psi, N, 0.1, 'nlp_matvec_');
  %dx = gcr(-F, psi, N, min(0.1,nFold), 'nlp_matvec_');
  nsolves = nsolves + 1;
  alpha = 1.0;
  F = loadRHS_(psi + alpha * dx, N);
  nres = nres + 1;
  while(norm(F) >= nFold)
    alpha = 0.5 * alpha;
    F = loadRHS_(psi + alpha * dx, N);
    nres = nres + 1;
  end
  psi = psi + alpha * dx;
  absdx(iter) = norm(alpha * dx);
  absb(iter) = norm(F) / initialF;
  fprintf(1, 'iter %d norm dx = %g norm F = %g alpha = %g\n', ...
    iter, norm(dx), norm(F), alpha);
```

```
  if (norm(alpha * dx) < RelTol * norm(psi) & norm(F) < RelTol * initialF)
    break;
  end;
end;
fprintf(1, 'V = %g %d iters %d residual evals %d solves \n', V, iter, nres, nsolves);
absdx = absdx / norm(psi);
```

Three auxiliary functions are needed; the first is the GCR code, which has only been slightly modified:

```
function [xn,iter,norms] = gcr(b,xnom,maxiters,tol,matvec)
% generalized conjugate residual iteration
% terminate on norm(Ax-b) < tol * (norm(Ax0-b)
% or iters > maxiters
% xnom is the current approximate solution from Newton
% x0 = zeros in this code
norms = [];
iter = 1;
x = zeros(size(xnom));
r = b;
p(:,iter) = r;
whatnorm = 2;
Ar(:,iter) = feval(matvec,r,xnom,b);
Ap(:,iter) = Ar(:,iter);
norm_ap(iter) = Ap(:,iter)' * Ap(:,iter);
norms(iter,1) = norm(r,whatnorm);
gcreps = 1.0e-14;
while (1),
  inner = (Ap(:,iter)' * Ap(:,iter));
  norm_ap(iter) = inner;
  if inner < gcreps * gcreps,
    fprintf(1, 'GCR iter %d norm(Ap) = %g\n', iter, norm(Ap(:,iter)))
    xn = x;
    if (norms(1,1) > eps) norms = norms / norms(1,1); end;
    return;
  end;
  a = r' * Ap(:,iter) / inner;
  x = x + a*p(:,iter);
  r = r - a * Ap(:,iter);
  p(:,iter+1) = r;
  norms(iter+1,1) = norm(r,whatnorm);
  if (norms(iter+1,1)/norms(1,1)) < tol,
    break;
  end;
```

```
  if (iter >= maxiters),
    break;
  end;
  Ar(:,iter+1) = feval(matvec,r,xnom,b);
  Ap(:,iter+1) = Ar(:,iter+1);
  start = iter;
  for j = start:iter,
    beta = Ap(:,iter+1)' * Ap(:,j) / norm_ap(j);
    p(:,iter+1) = p(:,iter+1) - beta * p(:,j);
    Ap(:,iter+1) = Ap(:,iter+1) - beta * Ap(:,j);
  end;
  iter = iter + 1;
end;
xn = x;
norms(iter,1)/norms(1,1);
if (norms(1,1) > eps) norms = norms / norms(1,1); end;
```

The function to load the RHS is the same as before:

```
function F = loadRHS_(psi,N)

global M;
global V;
h = 1 / (N+1);
F = M * psi + h*h*2.0*sinh(psi);
F(1) = F(1) + V;
F(N) = F(N) - V;
```

The only real change in the code is in the computation of the matrix-vector products for GCR:

```
function y = nlp_matvec(r, xnom, b)

alpha = 1.0e-3;
y = loadRHS_(xnom+alpha*r, size(xnom,1)) + b;
y = y / alpha;
```

Note that the right-hand side for GCR is the current $F(x^k)$, and that we reuse this function evaluation in the computation of the matrix-vector product at each iteration of GCR. Thus, each iteration of GCR requires only one new function evaluation.

(a) Convergence results for "standard' Newton and "matrix-free" Newton are shown in Fig. 1. Two main differences are apparent. First, standard Newton converges quadratically, but the matrix-free Newton converges linearly. This is a reflection of the fact that we are only solving the Newton update equation approximately at each iteration. Note that for early Newton iterations, when Newton is fairly far from the solution, there is negligible difference between the two approaches-we don't have to solve the update equation accurately in this case. However, if we wish to observe quadratic convergence, we must solve the update equation with increasing accuracy as Newton progresses. Secondly, Newton stalls earlier when using the matrix-free approach. Standard Newton can push the norm of the residual down to machine precision, but the matrix-free Newton can only achieve a solution that is four or five orders of magnitude less accurate. This is a reflection of the approximations in our implementation of the matrix-free approach. First, we are coly computing matrix-vector products approximately. Secondly, (and this is probably less important), we are only solving the Newton update equation to a very low tolerance.

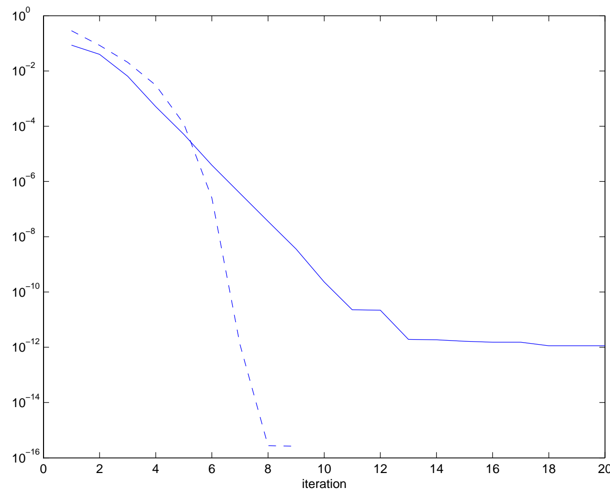

Figure 1: $\|F(x^k)\|/\|F(x^0)\|$ for nonlinear Poisson equation. Dashed line is standard Newton method; solid line is matrix-free/GCR -Newton with $\epsilon = 0.1$, $\alpha = 1.0e-3$.

(b) We made two modifications to the Newton code. The first modification adjusts α to compute a more precise matrix-vector product. Our strategy is to pick α such that the norm of the vector added to x^k is fixed relative to $\|x^k\|$. In other words, we fix $\|\alpha r\|/\|x^k\| = \beta$, so $\alpha = \beta\|x^k\|/\|r\|$, where β is some constant. A good rule of them is $\beta \simeq$ square root of machine precision ("eps" in Matlab). We also check for pathologically degenerate cases, such as very small $\|r\|$. The new code is:

```
function y = nlp_matvec_(r, xnom, b)
```

```
if norm(r,2) > eps,
  alpha = sqrt(eps) * norm(xnom,2) / norm(r,2);
  if alpha < sqrt (eps), alpha = sqrt(eps); end;
  y = loadRHS_(xnom + alpha*r, size(xnom,1)) + b;
  y = y / alpha;
else
  y = zeros(size(xnom));
end;
```

Using this new matrix-vector product, we produced the dash-dotted curve of Fig. 2. Note that we are able to achieve a much more accurate solution, but the convergence is still linear. However, the solution is not quite as accurate as can be achieved using direct methods to solve the linear Newton update equation.

To illustrate that we can still achieve quadratic convergence with the matrix-free method, we computed the dotted line of Fig. 2. The necessary modification was to adjust ϵ depending on how close Newton was to the solution. We chose $\epsilon = \min(0.1, \|F(x^k)\|)$.

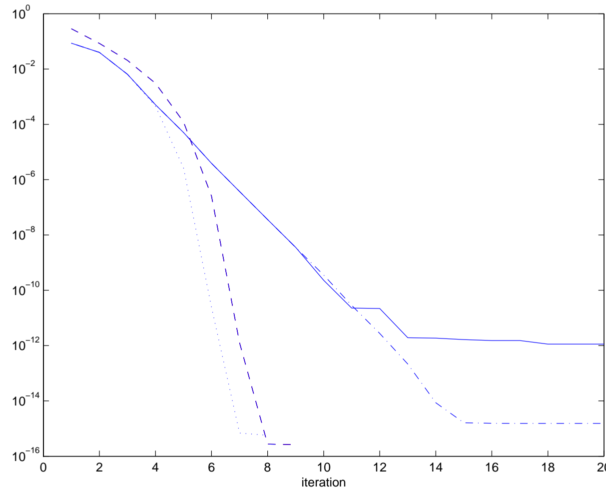

Figure 2: $\|F(x^k)\|/\|F(x^0)\|$ for nonlinear Poisson equation. Dashed line is standard Newton method; solid line is matrix-free/GCR -Newton with $\epsilon = 0.1$, $\alpha = 1.0e-3$. Dashed-dotted line is for $\epsilon = 0.1$, $\alpha \simeq 1.5 \times 10^{-8}\|x^k\|/\|p\|$ where p is the vector multiplied by the Jacobian. Dotted line is $\epsilon = \min(0.1, \|F(x^k)\|)$, $\alpha \simeq 1.5 \times 10^{-8}\|x^k\|/\|p\|$

(c) In this problem, we want to avoid excessive function evaluations incurred by solving the linear system more accurately than Newton needs, so we want low-accuracy (high ϵ) in GCR. At the same time, we don't want to make the solution so inexact that Newton takes

too many iterations to converge. Fig. 3 shows the number of function evaluations required for different (fixed) values of ϵ. It turns out that $\epsilon = 0.1$ is optimal.

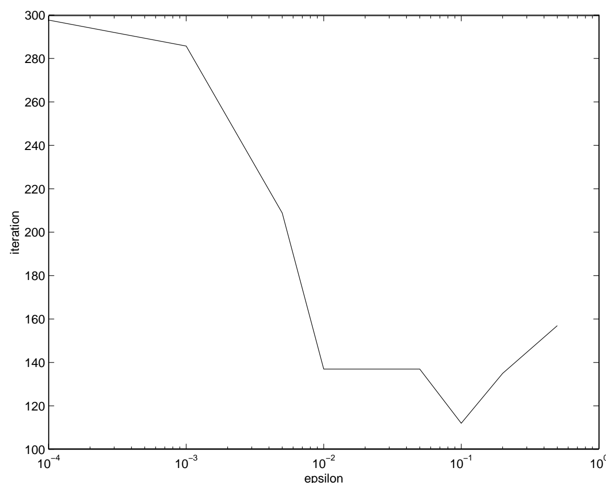

Figure 3: Number of function evaluations vs. ϵ required to solve non-linear Poisson equation to $\|F(x^k)\| < 10^{-3}$.

(d) For each column in the Jacobian we need one function evaluation. Thus, at each iteration, we need N function evaluations just to compute the Jacobian! After we compute that Jacobian, we must still solve the linear system in some manner (e.g., Guassian elimination). Unless GCR convergence is very slow, it much better to use GCR.

Problem 3

There are two "bugs" in the Newton code. The test files were designed to give hints as to what the bugs were. First, the cross-terms in the Jacobian are missing. When you first tried the Newton code on "test1.sys" and "test2.sys", you probably observed *linear* instead of quadratic convergence of Newton's method. This should have made you suspect that something was wrong with the Jacobian. The modified force.m file is:

```
function [fx,fy,dfxdx,dfxdy,dfydy,dfydx] = force(xn,yn,xo,yo,e)
% Determine the struct force

ln = sqrt(xn * xn + yn * yn);
lo = sqrt(xo * xo + yo * yo);

fx = e * xn * (1 - lo/ln);
```

```
dfxdx = e * (1 - (lo * (ln * ln - xn * xn)/(ln * ln * ln)));
dfxdy = e * (lo * xn * yn) / (ln * ln * ln);
fy = e * yn * (1 - lo/ln);
dfydy = e * (1 - (lo * (ln * ln - yn * yn)/(ln * ln * ln)));
dfydx = e * (lo * xn * yn) / (ln * ln * ln);
```

and the modified loadNewton.m file is:

```
function [Matrix,RHS] = loadNewton(Matrix, RHS, x)
% Evaluate the right-hand side and the jacobian for a newton's method.

global Joints Struts Loads;

% Zero the matrix and the rhs.
Matrix = 0 * Matrix;
RHS = 0 * RHS;

% Load the struts.
for i = 1:size(Struts,1)
 n1 = Struts(i,1);
 n2 = Struts(i,2);
 elasticity = Struts(i,3);

% Neither strut joint is fixed.
 if (n1 > 0) & (n2 > 0)
% Calculate the strut displacement.
   xorig = Joints(n1,1) - Joints(n2,1);
   yorig = Joints(n1,2) - Joints(n2,2);
   xnow = x(2 * n1 - 1) - x(2 * n2 - 1);
   ynow = x(2 * n1) - x(2 * n2);

% Calculate the indices into the matrix.
   x1 = 2 * n1 - 1;
   x2 = 2 * n2 - 1;
   y1 = 2 * n1;
   y2 = 2 * n2;

   [fx,fy,dfxdx,dfxdy,dfydy,dfydx] = force(xnow,ynow,xorig,yorig,elasticity);

% Load the RHS.
   RHS(x1) = RHS(x1) + fx;
   RHS(x2) = RHS(x2) - fx;
   RHS(y1) = RHS(y1) + fy;
   RHS(y2) = RHS(y2) - fy;
```

```
% Load the matrix.
% Diagonal Block.
   Matrix(x1, x1) = Matrix(x1,x1) + dfxdx;
   Matrix(x1, y1) = Matrix(x1,y1) + dfxdy;
   Matrix(y1, y1) = Matrix(y1,y1) + dfydy;
   Matrix(y1, x1) = Matrix(y1,x1) + dfydx;

   Matrix(x2, x2) = Matrix(x2,x2) + dfxdx;
   Matrix(x2, y2) = Matrix(x2,y2) + dfxdy;
   Matrix(y2, y2) = Matrix(y2,y2) + dfydy;
   Matrix(y2, x2) = Matrix(y2,x2) + dfydx;

% Off-Diagonal Blocks.
   Matrix(x1, x2) = Matrix(x1,x2) - dfxdx;
   Matrix(x1, y2) = Matrix(x1,y2) - dfxdy;
   Matrix(y1, y2) = Matrix(y1,y2) - dfydy;
   Matrix(y1, x2) = Matrix(y1,x2) - dfydx;

   Matrix(x2, x1) = Matrix(x2,x1) - dfxdx;
   Matrix(x2, y1) = Matrix(x2,y1) - dfxdy;
   Matrix(y2, y1) = Matrix(y2,y1) - dfydy;
   Matrix(y2, x1) = Matrix(y2,x1) - dfydx;

% The n2 joint is fixed.
 else
% Calculate the strut displacement.
   xorig = Joints(n1,1) - 0.0;
   yorig = Joints(n1,2) - 0.0;
   xnow = x(2 * n1 - 1) - 0.0;
   ynow = x(2 * n1) - 0.0;

% Calculate the indices into the matrix.
   x1 = 2 * n1 - 1;
   y1 = 2 * n1;

   [fx,fy,dfxdx,dfxdy,dfydy,dfydx] = force(xnow,ynow,xorig,yorig,elasticity);

% Load the RHS.
   RHS(x1) = RHS(x1) + fx;
   RHS(y1) = RHS(y1) + fy;

% Load the matrix.
   Matrix(x1, x1) = Matrix(x1,x1) + dfxdx;
   Matrix(x1, y1) = Matrix(x1,y1) + dfxdy;
```

```
   Matrix(y1, y1) = Matrix(y1,y1) + dfydy;
   Matrix(y1, x1) = Matrix(y1,x1) + dfydx;
 end
end

% Load the Loads.
for i = 1:size(Loads,1)
 n1 = Loads(i,1);
 x1 = 2 * n1 - 1;
 y1 = 2 * n1;
 RHS(x1) = RHS(x1) - Loads(i,2);
 RHS(y1) = RHS(y1) - Loads(i,3);
end
```

The second problem should have first been apparent from “test3.sys”. At the first Newton iteration, the Jacobian is

$$J = \begin{bmatrix} 10.2 & 0 \\ 0 & 0 \end{bmatrix}$$

which is obviously singular. However, as the RHS is

$$b = \begin{bmatrix} 10 \\ 0 \end{bmatrix}$$

there is a solution to the linear system. When you added the cross-terms to the Jacobian, you also (hopefully) found that the Jacobian at the first Newton iteration is always singular. The matlab code

```
dx = (Matrix \ (-RHS'))'
```

makes Matlab use some form of Gaussian elimination to solve the system. These algorithms fail on singular problems. You might have been lucky with some of the examples if there was some numerical error in your Jacobian, but the approach will generally fail. There are several possible fixes. We have used QR factorizations and GCR to solve singular problems - using either of there algorithms is satisfactory. Another solution is to use Matlab’s “pinv()” function. For sparse matrices, GCR is probably the way to go.

After implementing all these changes, you should have observed quadratic convergence on the first two examples. On the third example, “test3.sys”, you should have observed convergence in one iteration, since the foce-length relationship in that case reduces to a linear one, and Newton solves linear problems exactly in one iteration.