CE3-08 – Process Model Solution and Optimization

Problem Sheet for Topic #2, with Answers

---

1. The bisection method is a slow, yet simple and robust, method for solving scalar nonlinear equations. In this question, you are to develop and test an implementation of this method.

   Consider the following **incomplete** MATLAB™ function, named (`bisection.m`), which is inspired from the bisection method algorithm on Slide #9 of Lecture 2. (**This incomplete m-file is given on Blackboard**.)

bisection.m

```
function xm = bisection( f, xl, xu, eps, maxit )

% STEP 0: INITIALIZATION
fl = f(xl);
fu = f(xu);
if(                          % <- TO BE COMPLETED
                             % <- TO BE COMPLETED
end

% DISPLAY
fprintf( '%4s  %12s  %12s  %12s  %12s  %12s  %12s  %12s \n',...
        'it','xl','xm','xu','f(xl)','f(xm)','f(xu)','err');

% MAIN LOOP
for it = 0:                  % <- TO BE COMPLETED

    if( it>0 )
        xm0 = xm;
    end

    % STEP 1: CALCULATE MID-POINT
    xm =                     % <- TO BE COMPLETED
    fm =                     % <- TO BE COMPLETED

    % DISPLAY
    if( it==0 )
        fprintf( '%4d  %12.4e  %12.4e  %12.4e  %12.4e  %12.4e  %12.4e \n',...
        it+1, xl, xm, xu, fl, fm, fu );
    else
        fprintf( '%4d  %12.4e  %12.4e  %12.4e  %12.4e  %12.4e  %12.4e  %12.4e \n',...
        it+1, xl, xm, xu, fl, fm, fu, abs((xm-xm0)/xm) );
    end

    % STEP 2a: ROOT TO THE LEFT OF xm
    if(                      % <- TO BE COMPLETED
        xu =                 % <- TO BE COMPLETED
        fu =                 % <- TO BE COMPLETED

    % STEP 2b: ROOT TO THE RIGHT OF xm
    else
        xl =                 % <- TO BE COMPLETED
        fl =                 % <- TO BE COMPLETED
    end
```

```
    % STEP 3: STOPPING
    if( it>0 &&              % <- TO BE COMPLETED
        break;
    end

end
```

The input arguments of this function are as follows:

- f , the nonlinear function to be solved, $f(x^*) = 0$
- xl, a lower bound on the actual root, $x_\ell^{(0)} < x^*$
- xu, an upper bound on the actual root, $x_\mathrm{u}^{(0)} > x^*$
- eps, the user tolerance for terminating the iterations, $\left|f(x_\mathrm{m}^{(k)})\right| < \epsilon_\mathrm{tol}$
- maxit, the maximum allowed number of iterations

On successful completion, xm should contain an estimate of the root $x_\mathrm{m} \approx x^*$ within the specified tolerance ϵ_tol.

(a) Complete the foregoing m-file implementing to bisection method.

Answers. A possible Matlab implementation that uses the proposed m-file skeleton is as follows:

bisection.m

```
function xm = bisection( f, xl, xu, eps, maxit )

% STEP 0: INITIALIZATION
fl = f(xl);
fu = f(xu);
if( fl*fu >= 0 )
    error('f(xl)*f(xu)=%e not less than zero',fl*fu);
end

% DISPLAY
fprintf( '%4s  %12s  %12s  %12s  %12s  %12s  %12s  %12s \n',...
         'it','xl','xm','xu','f(xl)','f(xm)','f(xu)','err');

% MAIN LOOP
for it = 0: 1: maxit

    if( it>0 )
        xm0 = xm;
    end

    % STEP 1: CALCULATE MID-POINT
    xm  = 0.5*(xl+xu);
    fm  = f(xm);

    % DISPLAY
    if( it==0 )
        fprintf( '%4d  %12.4e  %12.4e  %12.4e  %12.4e  %12.4e  %12.4e \n',...
        it, xl, xm, xu, fl, fm, fu );
    else
```

```
        fprintf( '%4d  %12.4e  %12.4e  %12.4e  %12.4e  %12.4e  %12.4e  %12.4e \n',...
        it, xl, xm, xu, fl, fm, fu, abs((xm-xm0)/xm) );
    end

    % STEP 2a: ROOT TO THE LEFT OF xm
    if( fl*fm<0 )
        xu = xm;
        fu = fm;

    % STEP 2b: ROOT TO THE RIGHT OF xm
    else
        xl = xm;
        fl = fm;
    end

    % STEP 3: STOPPING
    if( it>0 && abs(fm)<eps )
        break;
    end

end
```

(b) Debug and test your m-file by considering the same problem as on slide #10 of Lecture 2:

$$\boxed{\text{Find } x \text{ such that } \exp(x) = 2 - x, \text{ for } x \in [0,1]}$$

Answers. Calling the above m-file in Matlab's command window gives the following output:

Command Window

```
>> format long
>> f = @(x) exp(x)-2+x;
>> xroot = bisection_answer( f, 0., 1., 1e-6, 100 )
 it          xl          xm          xu       f(xl)       f(xm)       f(xu)         err
  0  0.0000e+00  5.0000e-01  1.0000e+00 -1.0000e+00  1.4872e-01  1.7183e+00
  1  0.0000e+00  2.5000e-01  5.0000e-01 -1.0000e+00 -4.6597e-01  1.4872e-01  1.0000e+00
  2  2.5000e-01  3.7500e-01  5.0000e-01 -4.6597e-01 -1.7001e-01  1.4872e-01  3.3333e-01
  3  3.7500e-01  4.3750e-01  5.0000e-01 -1.7001e-01 -1.3670e-02  1.4872e-01  1.4286e-01
  4  4.3750e-01  4.6875e-01  5.0000e-01 -1.3670e-02  6.6745e-02  1.4872e-01  6.6667e-02
  5  4.3750e-01  4.5312e-01  4.6875e-01 -1.3670e-02  2.6346e-02  6.6745e-02  3.4483e-02
  6  4.3750e-01  4.4531e-01  4.5312e-01 -1.3670e-02  6.2904e-03  2.6346e-02  1.7544e-02
  7  4.3750e-01  4.4141e-01  4.4531e-01 -1.3670e-02 -3.7015e-03  6.2904e-03  8.8496e-03
  8  4.4141e-01  4.4336e-01  4.4531e-01 -3.7015e-03  1.2915e-03  6.2904e-03  4.4053e-03
  9  4.4141e-01  4.4238e-01  4.4336e-01 -3.7015e-03 -1.2057e-03  1.2915e-03  2.2075e-03
 10  4.4238e-01  4.4287e-01  4.4336e-01 -1.2057e-03  4.2686e-05  1.2915e-03  1.1025e-03
 11  4.4238e-01  4.4263e-01  4.4287e-01 -1.2057e-03 -5.8158e-04  4.2686e-05  5.5157e-04
 12  4.4263e-01  4.4275e-01  4.4287e-01 -5.8158e-04 -2.6946e-04  4.2686e-05  2.7571e-04
 13  4.4275e-01  4.4281e-01  4.4287e-01 -2.6946e-04 -1.1339e-04  4.2686e-05  1.3784e-04
 14  4.4281e-01  4.4284e-01  4.4287e-01 -1.1339e-04 -3.5352e-05  4.2686e-05  6.8913e-05
 15  4.4284e-01  4.4286e-01  4.4287e-01 -3.5352e-05  3.6668e-06  4.2686e-05  3.4455e-05
 16  4.4284e-01  4.4285e-01  4.4286e-01 -3.5352e-05 -1.5843e-05  3.6668e-06  1.7228e-05
 17  4.4285e-01  4.4285e-01  4.4286e-01 -1.5843e-05 -6.0879e-06  3.6668e-06  8.6139e-06
 18  4.4285e-01  4.4285e-01  4.4286e-01 -6.0879e-06 -1.2105e-06  3.6668e-06  4.3069e-06
 19  4.4285e-01  4.4285e-01  4.4286e-01 -1.2105e-06  1.2282e-06  3.6668e-06  2.1535e-06
 20  4.4285e-01  4.4285e-01  4.4285e-01 -1.2105e-06  8.8147e-09  1.2282e-06  1.0767e-06

xroot =

   0.442854404449463
```

Notice that the specified tolerance of 10^{-6} on the function value is satisfied after 20 iterations – The approximate root is $x^* \approx 0.4428544$ and corresponding function value, $f(x^*) \approx 8.81 \cdot 10^{-9}$.

---

2. The Ergun equation

$$\frac{\Delta P\,\rho}{G_0^2}\frac{D_p}{L}\frac{\epsilon^3}{1-\epsilon} = 150\frac{1-\epsilon}{\frac{D_p\,G_0}{\mu}} + 1.75$$

is used to describe the flow of a fluid through a packed bed. In this equation, ΔP stands for the pressure drop, ρ for the density of the fluid, G_o for the mass velocity (i.e., the mass flow rate divided by cross-sectional area), D_p for the diameter of the particles within the bed, μ for the fluid viscosity, L for the length of the bed, and ϵ for the void fraction of the bed.

In this problem, you are to find the void fraction of the bed, given the dimensionless quantities $\frac{D_p\,G_o}{\mu} = 1000$ and $\frac{\Delta P\,\rho\,D_p}{G_o^2\,L} = 20$.

(a) In Matlab, plot the function

$$f(\varepsilon) = \frac{\Delta P\,\rho}{G_o^2}\frac{D_p}{L}\varepsilon^3 - 150\frac{(1-\varepsilon)^2}{\frac{D_p\,G_o}{\mu}} - 1.75(1-\varepsilon),$$

for the void fraction $\varepsilon \in [0, 1]$.

Answers. Sample Matlab code to create the plot, as well as the plot itself (as Fig. 1) are provided below:

Command Window

```
>> e = 0:0.01:1;  % void fraction values to plot
>> f = 20*e.^3 - 150*(1-e).^2/1000 - 1.75*(1-e); % function value for void fractions
>> plot(e,f); % creating the plot
>> xlabel('Void Fraction')
>> ylabel('Function Value')
```

(b) By inspection, choose an interval $[\varepsilon_\ell, \varepsilon_\mathrm{u}]$, of width $\varepsilon_\mathrm{u} - \varepsilon_\ell = 0.2$, bracketing the actual root ε^* to the Ergun equation.

Answers. One possible interval is $[\varepsilon_\ell, \varepsilon_\mathrm{u}] = [0.25, 0.45]$. Checking to ensure the root is bracketed: $f(0.25) \times f(0.45) = -1.084375 \times 0.814625 = -0.88336$. Since this product is less than zero, the root is bracketed.

(c) Apply the **bisection method** to get a coarse estimate of the root ε^*. Start from the interval $[\varepsilon_\ell, \varepsilon_\mathrm{u}]$ chosen previously and perform 4 iterations only. Report **all** your intermediate calculations as well as the final root estimate—denoted by $\varepsilon^{(0)}$ subsequently.

Answers. We begin with the interval specified previously: $[0.25, 0.45]$. The value of the function at the midpoint of the interval ($\varepsilon_m = 0.35$), is then computed: $f(0.35) =$

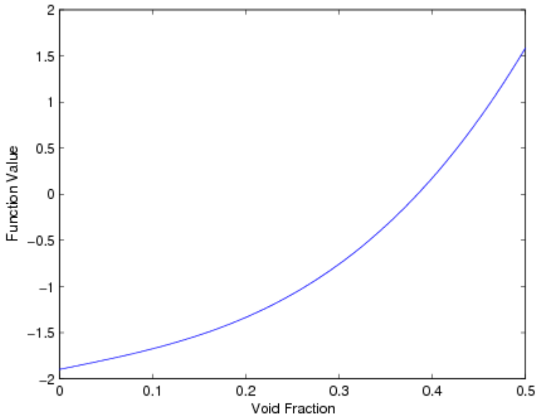

Figure 1: Plot showing the function value for different void fractions in Question 2.

-0.343375. Now the following comparison is made to determine which end of the original interval should be discarded:

$$f(\varepsilon_\ell)f(\varepsilon_m) = (-1.084375) \times (-0.343375) = 0.372 > 0$$

Since this product is above zero, the root must lie in the upper subinterval, therefore ε_ℓ is discarded and we replace it with ε_m. Note that if instead $f(\varepsilon_\ell)f(\varepsilon_m)$ was less than zero, we would have concluded that the root was in the lower subinterval and ε_u would have been discarded and replaced with ε_m.

The procedure is then repeated for the next three iterations. The results are summarized in the following table:

Iteration	ε_ℓ	$f(\varepsilon_\ell)$	ε_u	$f(\varepsilon_\mathrm{u})$	ε_m	$\mathrm{f}(\varepsilon_m)$	$E_a\%$
1	0.25	-1.084	0.45	0.815	0.35	-0.343	
2	0.35	-0.343	0.45	0.815	0.40	0.176	12.50
3	0.35	-0.343	0.40	0.176	0.375	-0.0977	6.667
4	0.375	-0.0977	0.40	0.176	0.3875	0.0356	3.226

(d) Refine the estimate of the root ε^* by applying the **Newton-Raphson method**. Use the value $\varepsilon^{(0)}$ obtained previously as the initial guess and perform 3 iterations. Report **all** your intermediate calculations as well as the final root estimate. Compare this estimate with the root on your graph.

Answers. We begin with $\varepsilon^{(0)} = 0.3875$ which was obtained from the bisection method in Question 2(c). For the Newton Raphson method, the first derivative of the function $f(\varepsilon)$ is required. This derivative was computed: $f'(\varepsilon) = 60\varepsilon^2 - 0.30\varepsilon + 2.05$. Three iterations of the Newton-Raphson method are shown below:

1st Iteration:

$$\varepsilon^{(1)} = \varepsilon^{(0)} - \frac{f(\varepsilon^{(0)})}{f'(\varepsilon^{(0)})} = 0.3875 - \frac{0.03556}{10.943} = 0.3842502$$

2nd Iteration:

$$\varepsilon^{(2)} = \varepsilon^{(1)} - \frac{f(\varepsilon^{(1)})}{f'(\varepsilon^{(1)})} = 0.3842502 - \frac{0.00024281}{10.794} = 0.3842281$$

3rd Iteration:

$$\varepsilon^{(3)} = \varepsilon^{(2)} - \frac{f(\varepsilon^{(2)})}{f'(\varepsilon^{(2)})} = 0.3842281 - \frac{4.282 \cdot 10^{-6}}{10.793} = 0.3842277$$

The final solution of 0.3842277 appears to be very close to the root in Fig. 1.

(e) Repeat Question 2(d) by applying the **secant method**, in lieu of the Newton-Raphson method. In particular, use the value $\varepsilon^{(0)}$ obtained previously as the initial guess, along with the extra initial point $\varepsilon^{(-1)} = \varepsilon^{(0)} + 0.01$, and perform 3 iterations.

Answers. We begin with $\varepsilon^{(0)} = 0.3875$ which was obtained from the bisection method in Question 2(c). The question states that $\varepsilon^{(-1)} = 0.3875 + 0.01 = 0.3975$. Three iterations of the Secant method are shown below:

1st Iteration:

$$\varepsilon^{(1)} = \varepsilon^{(0)} - \frac{f(\varepsilon^{(0)})(\varepsilon^{(0)} - \varepsilon^{(-1)})}{f(\varepsilon^{(0)}) - f(\varepsilon^{(-1)})} = 0.3875 - \frac{0.03556(0.3875 - 0.3975)}{0.03556 - 0.1473} = 0.38432$$

2nd Iteration:

$$\varepsilon^{(2)} = \varepsilon^{(1)} - \frac{f(\varepsilon^{(1)})(\varepsilon^{(1)} - \varepsilon^{(0)})}{f(\varepsilon^{(1)}) - f(\varepsilon^{(0)})} = 0.38432 - \frac{0.00097465(0.38432 - 0.3875)}{0.00097465 - 0.03556} = 0.38423$$

3rd Iteration:

$$\varepsilon^{(3)} = \varepsilon^{(2)} - \frac{f(\varepsilon^{(2)})(\varepsilon^{(2)} - \varepsilon^{(1)})}{f(\varepsilon^{(2)}) - f(\varepsilon^{(1)})} = 0.38423 - \frac{0.0000067387(0.38423 - 0.38432)}{0.0000067387 - 0.00097465} = 0.3842283$$

It appears as if the secant method is converging to the same root that the Newton-Raphson method was converging to.

(f) Check your results by comparing with Matlab's `fzero` function.

Answers. A possible way of invoking **`fzero`** is given below. Here, we start the search with the interval $[0.25, 0.45]$ which was obtained in Question 2(b). The results are in agreement with those found in Questions 2(d) and 2(e).

Command Window

```
>> format long
>> f = @(e) 20*e.^3 - 150*(1-e).^2/1000 - 1.75*(1-e);
>> options = optimset( 'display', 'iter' );
>> xroot = fzero( f, [0.25 0.45], options )

 Func-count    x          f(x)             Procedure
    2           0.45      0.814625        initial
    3       0.364205     -0.207077        interpolation
    4       0.381594    -0.0282697        interpolation
    5       0.384251   0.000247643        interpolation
    6       0.384228  -1.38875e-06        interpolation
    7       0.384228  -6.76195e-11        interpolation
    8       0.384228   4.44089e-16        interpolation
    9       0.384228   4.44089e-16        interpolation

Zero found in the interval [0.25, 0.45]
```

```
xroot =

   0.384227703256626
```

---

3. Newton's method is the basis for many advanced numerical methods for solution of nonlinear equations and optimization problems. The following Matlab function, named (`newton.m`), provides a basic implementation of this method (see Slide #38 of Lecture 2):

newton.m

```
function [x, f, it] = newton( g, dg, xini, eps, maxit )

it = 0;
x = xini;
f = g(x);
df = dg(x);

fprintf('iter    ||dx||     ||f(x)||\n');
fprintf('%4d           - %11.4e\n', it, norm(f) );

while( norm(f) > eps && it <= maxit )
    x0 = x;
    dx = - df \ f;
    x = x0 + dx;
    f = g(x);
    df = dg(x);
    it = it+1;

    fprintf('%4d %11.4e %11.4e\n', it, norm(dx), norm(f) );
end
end
```

The input arguments to this function are as follows:

- g , the vector-valued function calculating the residual of the nonlinear equation
- dg , the matrix-valued function calculating the Jacobian of the residual function
- `xini`, the initial guess
- eps, the user tolerance for terminating the iterations
- `maxit`, the maximum allowed number of iterations

On successful completion, the function returns:

- `x`, an (approximate) solution to the system of nonlinear equations
- `f`, the residual value at the approximate solution
- `it`, the iteration count

Consider the following system of nonlinear equation:

$$\begin{cases} 2(x_2)^2\cos(x_1) + x_1 = 1 \\ x_2 - 2\exp(x_1) = 2 \end{cases}$$

(a) Perform 3 iterations **by hand** of Newton's method, starting from the initial point $x_1^{(0)} = x_2^{(0)} = 0$.
Hint: Use Matlab's left division operator in your calculations of the Newton step $\Delta\mathbf{x}$!

Answers. The system of nonlinear equations to solve is the following:

$$\begin{aligned} 0 = f_1(\mathbf{x}) &= 2(x_2)^2\cos(x_1) + x_1 - 1 \\ 0 = f_2(\mathbf{x}) &= x_2 - 2\exp(x_1) - 2 \end{aligned}$$

Before we begin to show the results for the individual iterations, we need to compute the Jacobian matrix for that system:

$$\mathbf{J}(\mathbf{x}) = \begin{pmatrix} -2(x_2)^2\sin(x_1) + 1 & 4x_2\cos(x_1) \\ -2\exp(x_1) & 1 \end{pmatrix}$$

Each iteration proceeds with the following two steps:

$$\begin{aligned} \mathbf{J}(\mathbf{x}^{(k)}) \cdot \mathbf{\Delta x} &= -\mathbf{f}(\mathbf{x}^{(k)}) \\ \mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} + \mathbf{\Delta x} \end{aligned}$$

<u>Starting Point</u>: $\mathbf{x}^{(0)} = (0 \;\; 0)^{\mathsf{T}}$

<u>Iteration 1</u>:

$$\mathbf{f}^{(0)} = \begin{pmatrix} -1 \\ -4 \end{pmatrix}, \qquad \mathbf{J}^{(0)} = \begin{pmatrix} 1 & 0 \\ -2 & 1 \end{pmatrix}$$

Next, we solve the following system of equations for Δx:

$$\begin{pmatrix} 1 & 0 \\ -2 & 1 \end{pmatrix} \begin{pmatrix} \Delta x_1 \\ \Delta x_2 \end{pmatrix} = -\begin{pmatrix} -1 \\ -4 \end{pmatrix}$$

This solution can be done by hand (since it is a small system) using any of the methods for solving linear systems of equations detailed in Lecture 1. For simplicity, we use the left division operator in Matlab here – The solution is:

$$\begin{pmatrix} \Delta x_1 \\ \Delta x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 6 \end{pmatrix}$$

The next iterate $\mathbf{x}^{(1)}$ is obtained as:

$$\begin{pmatrix} x_1^{(1)} \\ x_2^{(1)} \end{pmatrix} = \begin{pmatrix} x_1^{(0)} \\ x_2^{(0)} \end{pmatrix} + \begin{pmatrix} \Delta x_1 \\ \Delta x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 6 \end{pmatrix} = \begin{pmatrix} 1 \\ 6 \end{pmatrix}$$

We then compute the new function values for the purpose of checking whether to stop the iterations or not:

$$\mathbf{f}(\mathbf{x}^{(1)}) \approx \begin{pmatrix} 38.9018 \\ -1.4366 \end{pmatrix} \quad \rightarrow \quad \|\mathbf{f}(\mathbf{x}^{(1)})\|_2 \approx 38.9283$$

Since both the error term and the function values are still very high we should continue on to iteration 2. Note in particular that the new iterate $\mathbf{x}^{(1)}$ is worse than the initial point $\mathbf{x}^{(0)}$ here.

Results of the first iteration and of the next two iterations are shown in the following table:

iter	Δx_1	Δx_2	$x_1^{(k)}$	$x_2^{(k)}$	$f_1(\mathbf{x}^{(k)})$	$f_2(\mathbf{x}^{(k)})$	$\Vert\mathbf{f}(\mathbf{x}^{(k)})\Vert_2$
1	1	6	1	6	38.9018	-1.4366	38.9283
2	-5.272	-27.228	-4.272	-21.228	-389.069	-23.255	389.763
3	0.556	23.271	-3.717	2.043	-11.723	-0.0052	11.723

The second iteration produces worse results, too. It is only from the third iteration onwards that the Newton's method starts producing improving estimates.

(b) Check your results with the m-file `newton.m` above.

Answers. The following results are obtained with the m-file `newton.m`. Note in particular that the results of iterations 1-3 are identical to those in Question 3(a) above. Convergence to a root of the nonlinear equation system is obtained after 7 iterations – Observe the very fast (quadratic) convergence after iteration 4.

Command Window

```
>> format long
>> f = @(x) [ 2*x(2)^2*cos(x(1))+x(1)-1.; x(2)-2*exp(x(1))-2. ];
>> J = @(x) [ [ -2*x(2)^2*sin(x(1))+1 4*x(2)*cos(x(1)) ]; [ -2*exp(x(1)) 1 ] ];
>> [xroot, froot, iter] = newton( f, J, [0;0], 1e-8, 10 )
iter    ||dx||      ||f(x)||
   0           -  4.1231e+00
   1  6.0828e+00  3.8928e+01
   2  2.7733e+01  3.8976e+02
   3  2.3278e+01  1.1723e+01
   4  3.0360e+00  1.2968e+00
   5  1.7452e-01  4.8047e-03
   6  1.2906e-03  6.2358e-06
   7  1.7429e-06  1.1521e-11

xroot =

  -6.606365043880198
   2.002703473467496


froot =

   1.0e-10 *

  -0.115214504603500
  -0.000039968028887


iter =

     7
```

(c) Compare the solution given by `newton.m` with the results of Matlab's intrinsic function `fsolve`. What is happening? Repeat the comparison for different initial points.

Answers. A possible way of invoking `fsolve` is shown below. We show the results from the same initial point as above first:

Command Window

```
>> format long
>> f = @(x) [ 2*x(2)^2*cos(x(1))+x(1)-1.; x(2)-2*exp(x(1))-2. ];
>> options = optimoptions( @fsolve, 'display', 'iter', 'tolfun', 1e-8 );
>> xroot = fsolve( f, [0;0], options )

                                         Norm of      First-order   Trust-region
 Iteration  Func-count     f(x)          step         optimality    radius
     0          3              17                            7               1
     1          6         7.89366              1           4.33               1
     2          9         3.30734              1           6.31               1
     3         10         3.30734              1           6.31               1
     4         13         1.53307           0.25           1.47            0.25
     5         16        0.277508          0.625          0.759           0.625
     6         19       0.0881395       0.471301           3.92            1.56
     7         22     4.85978e-06        0.02444         0.0293            1.56
     8         25     5.11118e-15    0.000170233       9.43e-07            1.56
     9         28     4.93038e-32     1.0039e-08       2.95e-15            1.56

Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the selected value of the function tolerance, and
the problem appears regular as measured by the gradient.

xroot =

  -1.379754662862844
   2.503280564387298
```

Observe that `fsolve` terminates successfully, yet finds another root than with Newton's method. Starting the search at $\mathbf{x}^{(0)} = (-7\ \ 2)^\mathsf{T}$, closer to the root found in Question 3(b), gives:

Command Window

```
>> xroot = fsolve( f, [-7;2], options )

                                         Norm of      First-order   Trust-region
 Iteration  Func-count     f(x)          step         optimality    radius
     0          3         3.87611                         12.3               1
     1          6       0.0994265        0.31241           2.32               1
     2          9      0.00044499      0.0753744           0.16               1
     3         12     1.65122e-08     0.00582437       0.000976               1
     4         15     2.40657e-17    3.59141e-05       3.73e-08               1
     5         18               0    1.37123e-09              0               1

Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the selected value of the function tolerance, and
the problem appears regular as measured by the gradient.

xroot =

  -6.606365043876977
```

```
    2.002703473467509
```

Nonetheless, the starting point $\mathbf{x}^{(0)} = (-6 \;\; 2)^{\mathsf{T}}$ produces yet another root as:

Command Window
```
 >> xroot = fsolve( f, [-6;2], options )

                                          Norm of      First-order   Trust-region
  Iteration  Func-count     f(x)          step         optimality    radius
      0          3         0.464279                        5.23               1
      1          4         0.464279       0.600968         5.23               1
      2          7       0.00587312       0.150242        0.267            0.15
      3          8       0.00587312       0.249821        0.267           0.376
      4         11        0.0025774      0.0624552        0.108          0.0625
      5         12        0.0025774       0.146675        0.108           0.156
      6         15       0.00130741      0.0366689       0.0391          0.0367
      7         18       0.00115198      0.0916722        0.227          0.0917
      8         21      2.47544e-09     0.00610682     0.000332          0.0917
      9         24      4.57275e-19    1.42694e-05     4.51e-09          0.0917

 Equation solved.

 fsolve completed because the vector of function values is near zero
 as measured by the selected value of the function tolerance, and
 the problem appears regular as measured by the gradient.

 xroot =

   -5.693416869033022
    2.006736130096785
```

Many additional roots of this nonlinear system can be found by choosing other initializations.

(d) Modify the m-file `newton.m` to implement a simple linesearch (see Slides # 40 and #47 of Lecture 2). Call the new m-file `newtonvar.m`.

Answers. A possible modification of the m-file `newton.m` above in order to implement the linesearch strategy is as follows:

newtonvar.m
```
function [x, f, it] = newton_stepsize( g, dg, xini, eps, maxit )

it = 0;
x = xini;
f = g(x);
df = dg(x);

fprintf('iter    alpha     ||f(x)||\n');
fprintf('%4d           - %11.4e\n', it, norm(f) );

while( norm(f) > eps && it <= maxit )
    x0 = x;
    f0 = f;
```

```
    dx = -df \ f;

    alpha = 1;
    x = x0 + dx;
    f = g(x);
    while( norm(f) >= norm(f0) )
      alpha = 0.5 * alpha;
      x = x0 + alpha * dx;
      f = g(x);
    end

    df = dg(x);
    it = it+1;
    fprintf('%4d %11.4e %11.4e\n', it, alpha, norm(f) );
end
end
```

(e) Test your new m-file `newtonvar.m` for the above example. Compare its behavior and performance with that of `newton.m` and `fsolve`.

Answers. The results produced by the m-file `newtonvar.m` from the default initial point $\mathbf{x}^{(0)} = (0\ \ 0)^{\mathsf{T}}$ are shown below. Note that the root is now identical to the one found with `fsolve` in Question 3(c).

Command Window

```
>> format long
>> f = @(x) [ 2*x(2)^2*cos(x(1))+x(1)-1.; x(2)-2*exp(x(1))-2. ];
>> J = @(x) [ [ -2*x(2)^2*sin(x(1))+1 4*x(2)*cos(x(1)) ]; [ -2*exp(x(1)) 1 ] ];
>> [xroot, froot, iter] = newtonvar( f, J, [0;0], 1e-6, 10 )
iter    alpha      ||f(x)||
   0           -  4.1231e+00
   1  1.2500e-01  3.5246e+00
   2  1.0000e+00  2.1978e+00
   3  1.0000e+00  7.2935e-01
   4  1.0000e+00  9.3844e-03
   5  1.0000e+00  2.6008e-06
   6  1.0000e+00  3.4268e-13

xroot =

  -1.379754662862819
   2.503280564387304


froot =

   1.0e-12 *

   0.342614825399323
  -0.006661338147751


iter =

     6
```

(f) Modify the m-file `newtonvar.m` to implement the Levenberg-Marquardt method (see Slides # 42-43 of Lecture 2). Call the new m-file `levenberg.m`. For simplicity, use a fixed relaxation parameter λ, which should be added to the list of input arguments of the function.

Answers. A possible modification of the m-file `newtonvar.m` above in order to implement the Levenberg-Marquardt method is as follows:

newtonvar.m

```
1  function [x, f, it] = levenberg( g, dg, xini, eps, maxit, lambda )
2  
3  it = 0;
4  x = xini;
5  f = g(x);
6  df = dg(x);
7  
8  fprintf('iter    alpha     ||f(x)||\n');
9  fprintf('%4d           - %11.4e\n', it, norm(f) );
10 
11 while( norm(f) > eps && it <= maxit )
12     x0 = x;
13     f0 = f;
14     dx = - ( df'*df + lambda*eye(length(x)) ) \ ( df'*f );
15 
16     alpha = 1;
17     x = x0 + dx;
18     f = g(x);
19     while( norm(f) >= norm(f0) )
20       alpha = 0.5 * alpha;
21       x = x0 + alpha * dx;
22       f = g(x);
23     end
24 
25     df = dg(x);
26     it = it+1;
27     fprintf('%4d %11.4e %11.4e\n', it, alpha, norm(f) );
28 end
29 end
```

The two changes compared with `newtonvar.m` are:

line 1 The extra input argument `lambda` is used to pass the value of the relaxation parameter λ

line 14 The Newton search direction is replaced by the Levemberg-Marquardt direction (without scaling):

$$\left[\mathbf{J}(\mathbf{x}^{(k)})^{\mathrm{T}}\mathbf{J}(\mathbf{x}^{(k)}) + \lambda\mathbf{I}\right]\Delta\mathbf{x} = -\mathbf{J}(\mathbf{x}^{(k)})^{\mathrm{T}}\mathbf{f}(\mathbf{x}^{(k)})$$

(g) Test your new m-file `levenberg.m` for the above example. Compare its behavior and performance with that of `newtonvar.m` (after selecting the Levenberg-Marquardt algorithm).

Answers. The results produced by the m-file `levenberg.m` from the default initial point $\mathbf{x}^{(0)} = (0\ \ 0)^{\mathsf{T}}$ are shown below for $\lambda = 1$ and $\lambda = 0.01$.

Command Window

```
>> format long
>> f = @(x) [ 2*x(2)^2*cos(x(1))+x(1)-1.; x(2)-2*exp(x(1))-2. ];
>> J = @(x) [ [ -2*x(2)^2*sin(x(1))+1 4*x(2)*cos(x(1)) ]; [ -2*exp(x(1)) 1 ] ];
>> [xroot, froot, iter] = levenberg( f, J, [0;0], 1e-6, 100, 1e0 )
iter    alpha     ||f(x)||
   0           -  4.1231e+00
   1  1.0000e+00  1.7776e+00
   2  5.0000e-01  1.1983e+00
   3  1.0000e+00  9.1253e-01
   4  1.0000e+00  2.6550e-01
   5  1.0000e+00  1.1867e-01
   6  1.0000e+00  5.4054e-02
   7  1.0000e+00  2.5356e-02
   8  1.0000e+00  1.1914e-02
   9  1.0000e+00  5.6027e-03
  10  1.0000e+00  2.6356e-03
  11  1.0000e+00  1.2400e-03
  12  1.0000e+00  5.8344e-04
  13  1.0000e+00  2.7452e-04
  14  1.0000e+00  1.2917e-04
  15  1.0000e+00  6.0781e-05
  16  1.0000e+00  2.8600e-05
  17  1.0000e+00  1.3457e-05
  18  1.0000e+00  6.3322e-06
  19  1.0000e+00  2.9795e-06
  20  1.0000e+00  1.4020e-06
  21  1.0000e+00  6.5969e-07

xroot =

  -1.379754576182506
   2.503279948558605


froot =

   1.0e-06 *

  -0.017614845360114
  -0.659453223805784


iter =

    21
```

Command Window

```
>> format long
>> f = @(x) [ 2*x(2)^2*cos(x(1))+x(1)-1.; x(2)-2*exp(x(1))-2. ];
>> J = @(x) [ [ -2*x(2)^2*sin(x(1))+1 4*x(2)*cos(x(1)) ]; [ -2*exp(x(1)) 1 ] ];
>> [xroot, froot, iter] = levenberg( f, J, [0;0], 1e-6, 100, 1e-2 )
iter    alpha     ||f(x)||
   0           -  4.1231e+00
   1  2.5000e-01  3.2649e+00
   2  1.0000e+00  2.4016e+00
   3  5.0000e-01  8.6200e-01
```

```
    4  1.0000e+00  5.4302e-01
    5  1.0000e+00  2.5285e-02
    6  1.0000e+00  5.2633e-04
    7  1.0000e+00  4.1961e-05
    8  1.0000e+00  3.4247e-06
    9  1.0000e+00  2.7952e-07

xroot =

  -1.379754626135649
   2.503280303450367


froot =

   1.0e-06 *

  -0.007473145435810
  -0.279421014104386


iter =

     9
```

Notice that the roots are the same as the one found in Question 3(d). However, it takes more iterations for the Levenberg-Marquardt method to converge as the relaxation parameter λ increases. This behavior is expected as increasing λ brings robustness into the search direction, but sacrifices efficiency. When $\lambda = 0$, it can also be checked that `levenberg.m` produces the same results as `newtonvar.m`.

Finally, the results of `fsolve` with the Levenberg-Marquardt algorithm are as follows:

Command Window

```
>> format long
>> f = @(x) [ 2*x(2)^2*cos(x(1))+x(1)-1.; x(2)-2*exp(x(1))-2. ];
>> options = optimoptions( @fsolve, 'display', 'iter', 'tolfun', 1e-8, ...
                           'algorithm', 'levenberg-marquardt' );
>> xroot = fsolve( f, [0;0], options )

                                         First-Order                    Norm of
 Iteration  Func-count    Residual       optimality      Lambda           step
     0           3             17               7          0.01
     1           8        3.15998            3.28             1         1.45774
     2          12        2.12295            1.53            10        0.209109
     3          15         1.5816            1.32             1        0.159233
     4          18        1.08901            7.39           0.1        0.616965
     5          21       0.028024             2.1          0.01        0.501826
     6          24    3.96804e-05          0.0833         0.001       0.0405091
     7          27    5.72778e-12        3.12e-05        0.0001     0.000742744
     8          30    2.12475e-21         4.8e-11         1e-05     5.18491e-07

Equation solved, fsolve stalled.

fsolve stopped because the relative size of the current step is less than the
default value of the step size tolerance and the vector of function values
is near zero as measured by the selected value of the function tolerance.
```

```

xroot =

  -1.379754662856771
   2.503280564344271

```

Observer, in particular, the variation in the relaxation parameter λ from iteration to iteration in Matlab's implementation of the Levenberg-Marquardt algorithm.