CE3-08 – Process Model Solution and Optimization

Problem Sheet for Topic #2

---

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.

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

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

---

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 $\varepsilon \in [0,1]$.

(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.

(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.

(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.

(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.

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

---

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 $\Delta\mathbf{x}$!

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

(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.

(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`.

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

(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.

(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).