PHYS 410 - Computational Physics - HW 1 Writeup

Introduction

In this homework I implement numerical root finding algorithms. The first question uses bisection and Newton's method to find all roots of a function in a 1D domain. The second question uses Newton's method to find the solution to a multi-dimensional system of equations near an initial guess.

Review of theory

Convergence

We use relative error as our convergence criterion. Convergence is thus attained when, for some ε,

$$\delta x^{(n)}/x^{(n+1)} < \varepsilon. \tag{1}$$

While this is a better descriptor of convergence than absolute error, we should be careful with any roots close or equal to zero, where the algorithm may never converge. Thus it is necessary to limit the number of iterations performed. I choose 50 iterations for Newton's method as suggested in class, since it has quadratic convergence; I choose 2500 (equal to 50 squared) iterations for bisection because it has linear convergence.

Bisection

Bisection is a root finding algorithm with linear convergence. Given an initial interval $(x_{\text{min}}, x_{\text{max}})$ where $f(x_{\text{min}})f(x_{\text{max}}) < 0$, we can find at least one root. We define $x_{\text{mid}} = (x_{\text{min}} + x_{\text{max}})/2$. For each iteration, there are three cases:

1. $f(x_{\text{mid}}) = 0$. Then we have found a root at x_{mid}.
2. $f(x_{\text{mid}})f(x_{\text{max}}) < 0$. Then we update x_{min} to x_{mid}.
3. $f(x_{\text{mid}})f(x_{\text{max}}) < 0$. Then we update x_{max} to x_{mid}.

We can obtain the error using

$$\delta x^{(n)} = x_{\text{max}} - x_{\text{min}}. \tag{2}$$

Newton's Method

Newton's method is a root finding algorithm with quadratic convergence. Given a good initial guess (one that results in convergence).

In 1-D, we can iterate using the recursive relation

$$x^{(n+1)} = x^{(n)} - \delta x^{(n)}, \tag{3}$$

$$\delta x^{(n)} = f(x^{(n)})/f'(x^{(n)}). \tag{4}$$

In d-D, we can iterate using the recursive relation

$$\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} - \delta\boldsymbol{x}^{(n)}, \tag{5}$$

$$\boldsymbol{J}[\boldsymbol{x}^{(n)}]\delta\boldsymbol{x}^{(n)} = \boldsymbol{f}(\boldsymbol{x}^{(n)}), \tag{6}$$

where $\boldsymbol{J}[\boldsymbol{x}^{(n)}]$ is the Jacobian matrix of $\boldsymbol{f}(\boldsymbol{x})$, defined as

$$J_{\text{ij}} = \partial f_{\text{i}}/\partial \text{x}_{\text{j}}. \tag{7}$$

Numerical Approach

Problem 1

For this problem I first use bisection to find a preliminary root, iterating until a root with a relative error of *tol1* is obtained (**Eq. 1,2**). It's noteworthy to mention that I only proceed if $f(x_{\text{min}})f(x_{\text{max}}) < 0$. Then I use Newton's method using the preliminary root as the initial guess, iterating (**Eq. 3,4**) until a root with a relative error of *tol2* is obtained (**Eq. 1,4**) or the iteration limit is hit. This overall root finding algorithm is called the hybrid algorithm.

To find all roots of a function in some domains, I search for unique roots in small domain subdivisions. First I discretize the domain, with a subdivision length of $\Delta x = 0.01$. I determine this from graphing the function and finding that any root is at least Δx away from any domain boundaries or any other roots. Therefore, each subdivision contains exactly one root or none. In addition, the example function is monotonic in the neighbourhood of the roots, and thus $f(x_{\min})f(x_{\max}) < 0$ if a root exists. Then I apply the hybrid algorithm to find roots in all subdivisions. In the case the same root is found in two subdivisions, I store all roots in a vector and run the *unique()* function on it.

Problem 2

For this problem I first write the system of equations as a d-dimensional function $\boldsymbol{f}(\boldsymbol{x})$, where for a root $\boldsymbol{x}_0$, $\boldsymbol{f}(\boldsymbol{x}_0) = \boldsymbol{0}$. Then I compute the Jacobian using **Eq. 7**. I store both of these as static methods in a *utils* class (**Appendix A**). I then iterate (**Eq. 5**), solving for $\delta\boldsymbol{x}^{(n)}$ using

$$\delta\boldsymbol{x}^{(n)} = \boldsymbol{J}[\boldsymbol{x}^{(n)}] \setminus \boldsymbol{f}(\boldsymbol{x}^{(n)}), \tag{8}$$

where \ is the left division operation in MATLAB. The iteration is terminated until a root with a relative error of *tol* is obtained (**Eq. 1,8**) or the iteration limit is hit.

Results

Problem 1

Running the command: `findall(@f,@dfdx,0,2,0.001,10^-12,0.01)`, we obtain the roots of the example function. We can verify the roots using `f(ans)`.

```
>> findall(@f,@dfdx,0,2,0.001,10^-12,0.01)

ans =

    0.0123    0.1090    0.2929    0.5460    0.8436    1.1564    1.4540    1.7071    1.8910    1.9877

>> f(ans)

ans =

   1.0e-09 *

         0         0         0    0.0003    0.0046   -0.0086    0.0013    0.0038    0.0188    0.1096
```

Problem 2

Running the command: `newtond(@utils.f, @utils.jac, [-1.00;0.75;1.50], 10^-12)`, we obtain the solution to the example system. We can verify the solution using `utils.f(ans)`

```
>> newtond(@utils.f, @utils.jac, [-1.00;0.75;1.50], 10^-12)

ans =

   -0.5777
    0.4474
    1.0844

>> utils.f(ans)

ans =

   1.0e-11 *

    0.1379
   -0.0286
   -0.0374
```

Discussion/Conclusions

I implemented algorithms to find roots of functions in 1-D and n-D. These algorithms could be used when an explicit solution is difficult or computationally inefficient to find. As predicted by theory, Newton’s method converged much faster than bisection.

No generative AI was used.

Appendix A - utils.m

```
% defines the equations in the system and the Jacobian of the system
classdef utils
   methods(Static)
       % Function which implements the nonlinear system of equations.
       % Function is of the form f(x) where x is a length-d vector, and
       % which returns length-d column vector.
       function fx = f(x)
           y = x(2);
           z = x(3);
           x = x(1);
           fx = zeros(3,1);
           fx(1) = x^2+y^4+z^6-2;
           fx(2) = cos(x*y*z^2)-x-y-z;
           fx(3) = (x+y-z)^2-y^2-z^3;
       end
       % jac: Function which is of the form jac(x) where x is a length-d vector, and
       % which returns the d x d matrix of Jacobian matrix elements for the
       % nonlinear system defined by f.
       function jac = jac(x)
           y = x(2);
           z = x(3);
           x = x(1);
           jac = zeros(3);
           jac(1,:) = [2*x, 4*y^3, 6*z^5];
           jac(2,1) = -y*z^2*sin(x*y*z^2)-1;
           jac(2,2) = -x*z^2*sin(x*y*z^2)-1;
           jac(2,3) = -2*x*y*z*sin(x*y*z^2)-1;
           jac(3,:) = [2*(x+y-z), 2*(x-z), -2*x-2*y+(2-3*z)*z];
       end
   end
end
```