ESC 407H
Computational Methods in Engineering Science
Homework 3
Solving Nonlinear Equations and Rootfinding

Due Date: Thursday, October 8 at 11:59 p.m.

Use `format long` for all calculations done in MATLAB.

Problem 1

Create the following four MATLAB functions that find roots of $f(x) = 0$, which are described below.

`bisect` that implements the bisection method.

`false_pos` that implements the method of false position,

`secant` that implements the secant method,

`newton` that implements Newton's method.

For `bisect`, `false_pos`, and `secant` the inputs and outputs should be as follows.

inputs (i) the function $f(x)$ as a function handle; (ii) a row vector containing a and b, which are two initial values of x that bracket the root (they do not need to bracket the root for `secant`), with $a < b$; (iii) the desired relative error; and (iv) the maximum number of iterations allowed.

outputs (i) the approximation for the root, (ii) the function value at the root, (iii) the approximate relative error, and (iv) the number of iterations used.

For `newton`, the inputs and outputs should be as follows.

inputs (i) the function $f(x)$ as a function handle, (ii) $f'(x)$ as a function handle, (iii) the initial guess for the root, (iv) the desired relative error, (v) the maximum number of iterations allowed, and (vi) a flag that tells the function whether or not the maximum number of iterations is checked in the function.

outputs (i) the approximation for the root, (ii) the function value at the root, (iii) the approximate relative error, and (iv) the number of iterations used.

You should test your functions as you complete them by finding the roots of a simple function like $f(x) = x^2 - 100$ for which you know all the answers.

Next, plot the function $f(x) = -1 + 7e^{-x} \sin x$ and then write a script that calls each of the four functions you have created to determine its largest positive root. Use a relative error tolerance of 10^{-10} for each.* For each function call, you should report

*I have noticed that many of you input this tolerance as `10^-10` or `10^(-10)`. The first one makes me nervous, but it works. Note that most programming languages allow you to use the shorthand `1e-10` for the same thing. It is easier to read, shorter, and you don't have to worry about parentheses.

to the Command Window (i) the approximate root, (ii) the function value at the root, (iii) the final approximate relative error, and (iv) the number of iterations needed. Comment on the convergence rates that you see. Are they as expected? Finally, the script should use MATLAB's `fzero` command to find the "true" relative error for the approximation found by each of your functions.

Notes You functions should check for valid and missing inputs. For example `bisect` and `false_pos` should both check that a and b bracket the root. All functions should check whether or not the desired relative error and the maximum number of iterations have been input. If not, defaults should be set.

Problem 2

When laying water mains and some other underground utilities, utility companies must be concerned with the possibility of freezing. Soil and weather conditions are highly variable, but let's simplify our model and assume the soil is homogeneous and the outdoor temperature is constant. If you make these assumptions, then the temperature $T(x,t)$ (°C) at time t (s), a distance x (m) below the surface, is given by

$$\frac{T(x,t) - T_a}{T_i - T_a} = \operatorname{erf}\left(\frac{x}{2\sqrt{\alpha t}}\right)$$

where T_a is the constant temperature of the air, T_i is the initial soil temperature,

$$\alpha = \frac{k}{\rho c_p}$$

is the thermal diffusivity (m^2/s) of the soil, k is its thermal conductivity ($\mathrm{W}/(\mathrm{m}\cdot\mathrm{K})$), ρ is its density (kg/m^3), c_p is its specific heat capacity ($\mathrm{J}/(\mathrm{kg}\cdot\mathrm{K})$), and erf is the *error function*, which is defined as

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-\xi^2}\, d\xi.$$

Your goal is to determine the depth at which a water main should be buried so that it will not freeze until after the soil has been exposed for 60 days to a constant air temperature of $T_a = -15\,^\circ\mathrm{C}$. Assume that the initial soil temperature is $T_i = 20\,^\circ\mathrm{C}$ and that $\alpha = 0.138\times10^{-6}\ \mathrm{m}^2/\mathrm{s}$. Write a script and use your `bisect` and `newton` functions to find the depth to an accuracy of 1 mm. For each, report the depth, the approximate relative error, and the number of iterations required.

Problem 3

The function $f(x) = \ln(x^2+1) - e^{0.4x}\cos\pi x$ has an infinite number of roots.

(a) Use your `newton` function to determine to within 10^{-8} the only negative root.

(b) Use your `newton` function to determine to within 10^{-8} the four smallest positive roots.

(c) Determine a resonable initial approximation to find the nth smallest positive root of $f(x)$. It will help to plot $f(x)$.

(d) Use your result from (c) to determine the 25th positive root of f to within 10^{-8}.

You should write a MATLAB script for parts (a), (b), and (d). Your script should report the approximation of the root, the relative error of the approximation, and the number of iterations required to converge to the root.

Problem 4

Newton's method converges quadratically as long as the starting guess is close enough to the root. Outside the region of quadratic convergence, Newton's method can be very sensitive to the choice of starting guess. This problem will investigate this behavior of Newton's method, which is characteristic of *chaotic* systems.

Consider the complex valued function

$$f(z) = z^3 - 2z + 2,$$

where $z \in \mathbb{C}$.* This equation has three solutions in the complex plane that I will call z_i, $i = 1, 2, 3$. Now carry out the following steps.

1. By trial and error, use your `newton` function to find the three roots of f. Verify the roots you find using MATLAB's `roots` function.

2. Write a script that investigates to which of the three roots that each point in the complex plane converges. The script should investigate a rectangle in the complex plane with the lower left corner at $z = -2 - 1.5i$ and the upper right corner at $z = 2.5 + 1.5i$. The points you investigate should be no farther apart than 0.0025 in both the real and imaginary directions.† The closer together they are, the more awesome your final result will be!

    - Your script should define the function handles for $f(z)$ and $f'(z)$.
    - Your script should use `roots` to find the three roots of $f(z)$ and store their values.
    - Make sure you preallocate large arrays.
    - If your `newton` functions throws an error if it reaches the maxiumum number of iterations, then set the flag so that your function does not do that.
    - Next, your script should go through every point in the rectangle defined above and determine to which root it converges and assign an integer number to that point. If it converges to root 1, then assign it the number 1, if root 2, then the number 2, and if root 3, then a 3. If it does not converge

*As $\mathbb{R}$ represents the set of all real-valued numbers, $\mathbb{C}$ represents the set of all complex-values numbers.

†On my laptop, it took 51 seconds for the run to complete using that spacing. With a spacing of 0.0002, it took 02:06 (hh:mm).

to any of the three roots in 20 iterations, then move on to the next point and assign that point the number 0. Store these values in an array and then plot the array using MATLAB's `imagesc` command. Make sure that you tell `imagesc` the range in the complex plane it should be using for its output. Make a nice `colormap` if you want or use the default one. Also plot the three roots on the plot as easily seen dots.

When you are done, you should get an image that looks something like that seen in Fig. 1, though this image may have more points than you are willing to wait for.*

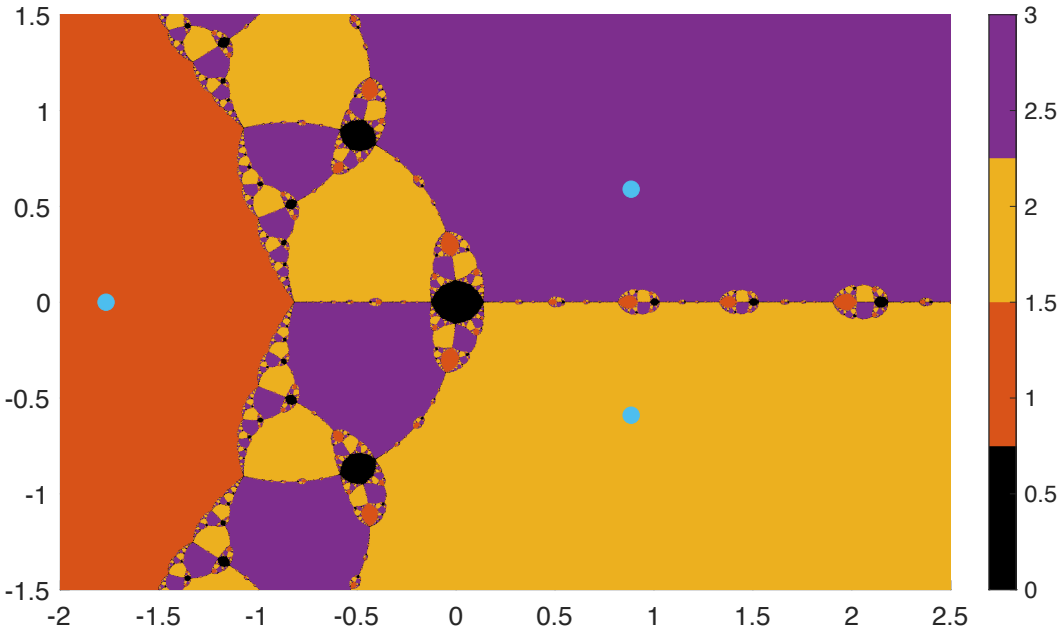

Figure 1. The *basins of attraction* of Newton's method for the polynomial $f(z) = z^3 - 2z + 3$ using 1,350,075,001 points. The orange/red points converge to $z = -1.7693$, the purple points converge to $z = 0.8846 + 0.5897i$, and the yellow points converge to $z = 0.8846 - 0.5897i$. The points colored black do not converge to any of the roots at 20 iterations.

🦄 **Extra Credit** (10%): Instead of coloring points according to just which root they converge, color them by *how fast* they converge. Doing this using three of MATLAB's color maps (`hot`, `cool`, and `summer`), I get the image shown in Fig. 2. The smaller the number within each color map, the faster it converges.

Problem 5

The LORAN (Long Range Navigation) system calculates the position of a ship at sea using signals from fixed transmitters using *hyperbolic navigation*. From the time differences of the incoming signals, a pair of stations can locate the position of a ship as a point on a hyperbolic curve that lies between the two stations (the stations are at the foci of the hyperbolic curve). By adding another pair of stations, a second hyperbolic

*It took my laptop 07:51 (hh:mm) to generate that image.

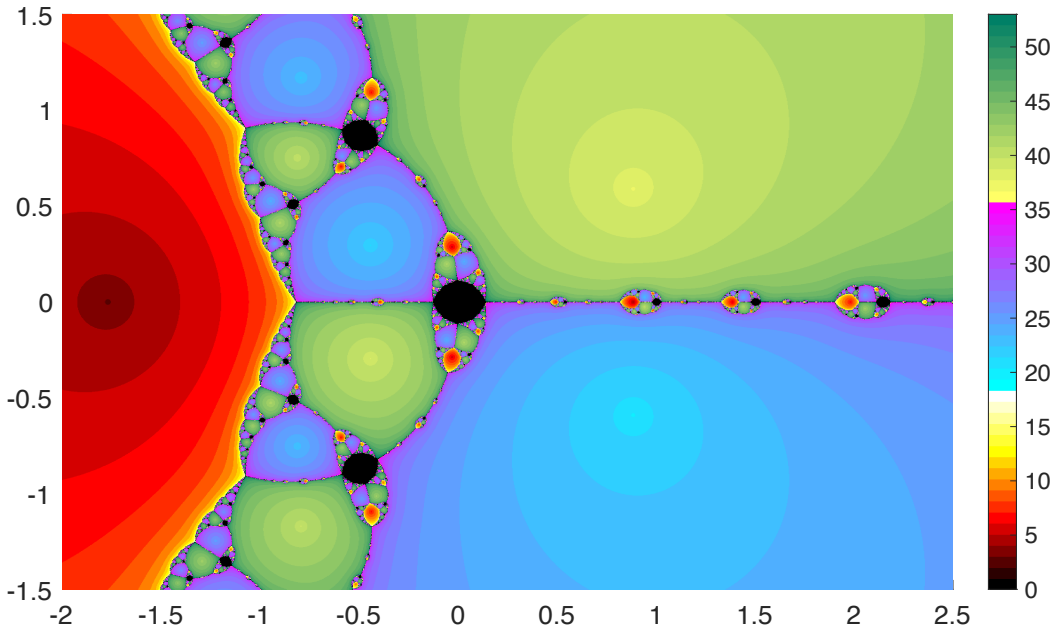

Figure 2. The *basins of attraction* of Newton's method for the polynomial $f(z) = z^3 - 2z + 3$ using 1,350,075,001 points with the colors chosen based on both the root it converges to and how many iterations it takes to converge. Note that no point converged in fewer than 3 iterations. In addition, for roots 2 and 3, you need to subtract 17 and 34, respectively to get the number of iterations. I haven't had time to come with a better way to do this yet.

curve is generated and the location of the ship will be on one of the intersections of the two hyperbolic curves. Since there will, in general, be multiple points of intersection, another navigation method is used to eliminate all the points but one. The Global Position System (GPS) we are all familiar with works similarly, but with much more accuracy than LORAN had.

An example of the type of equations that are generated are

$$\begin{aligned} \frac{x^2}{186^2} - \frac{y^2}{300^2 - 186^2} &= 1, \\ -\frac{(x-300)^2}{500^2 - 279^2} + \frac{(y-500)^2}{279^2} &= 1. \end{aligned}$$

These can be solved by hand by finding the roots of a fourth degree polynomial, but navigation systems must solve them hundreds of times per second, so it must be automated. I want you to find the four roots of these equations using Newton's method.

Recall from class that for a system of equations $\vec{f}(\vec{x}) = \vec{0}$, the increment in $\vec{x}$, which we called $\vec{h}$, is found by solving the following linear system of equations at each iteration

$$[J](\vec{x}_i)\vec{h}_i = -\vec{f}(\vec{x}_i),$$

where $[J](\vec{x}_i)$ the Jacobian matrix of $\vec{f}$ evaluated at $\vec{x}_i$. We then we update our estimate

of the root using Newton's method using

$$\vec{x}_{i+1} = \vec{x}_i + \vec{h}_i.$$

Note that we have not covered the solution of linear systems of algebraic equations so, for now, I want you to use MATLAB's built-in solver, which, given a linear system $[A]\vec{x} = \vec{b}$, the solution in MATLAB is simply `x = A\b`.

With this in mind, write a MATLAB function called `newton_sys` that finds a root of a nonlinear system of equations using Newton's method and use it to find the four roots of the system of equations given above. The function should take as inputs

(i) $\vec{f}$ as a vector function handle,

(ii) $[J]$ as a square matrix function handle,

(iii) the initial guess $\vec{x}_0$ for the root as a column vector,

(iv) the desired error for the approximation, and

(v) the maximum number of iterations.

The function should output

(i) the approximate root $\vec{x}_r$,

(ii) the relative error when the function was exited, and

(iii) the number of iteration executed by the function.

Write a MATLAB script that calls the function four times to find the four roots. This may take some trial and error, but that is the nature of rootfinding.

Since you are not approximating a scalar value, the error will need to be computed a little differently here. One thing we can do is to check the relative error for each element of $\vec{x}$ at each iteration and when the relative error for all of them is below a desired value, then stop. Another thing you could do is to stop when the Euclidean distance between two successive estimates of $\vec{x}_r$ is below a desired value. The second one is far easier and more efficient, so please use it.

As you have seen, MATLAB can handle passing vectors of functions with ease. For example, the nonlinear system

$$x^3 + y = 1,$$
$$-x + y^3 = -1,$$

can be input in MATLAB as the following anonymous function

```
f = @(x) [x(1)^3 + x(2) - 1; x(2)^3 - x(1) + 1];
J = @(x) [3*x(1)^2, 1; -1, 3*x(2)^2];
```

where I have also included the Jacobian of $\vec{f}$. You can test your function on these equations knowing that the only root is $(x, y) = (1, 0)$. We used an anonymous function like the first one when solving ordinary differential equations in Homework 1.