E7 Homework Assignment 10: Interpolation and Root Finding

The purpose of this assignment is to practice numerical methods techniques.

Note: A template will not be provided for this assignment. You still need to publish and submit your `.m` file. Neatly organize it using code cells and comments as was done in the previous templates. The autograder does not rely on the format of this file.

This completed assignment, named `HW10_Firstname_Lastname.m`, should be uploaded to bCourses along with the `.mat` file that is generated by the autograder. You may run the autograder as many times as you like to check your work as you complete the assignment; the `.mat` file will be regenerated each time, and your score will only be recorded based on the `.mat` file that you upload. You will also publish your `.m` files and upload a `.PDF` version of the result to bCourses.

Remember to upload the following to the bCourses website:

- `HW10_Firstname_Lastname.m`
- `HW10_Firstname_Lastname.pdf`
- `HW10_Score.mat`
- All function files that you are asked to create

Directions to upload files can be found at `http://guides.instructure.com/s/2204/m/4212/l/54353-how-do-i-upload-a-file-to-my-assignment-submission`

Part 1: Interpolation

1. A Taylor series expansion about 0 (also known as MacLaurin series) for the natural logarithm function can be written as follows:

$$\log(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \dots$$

This **only** works for $|x| < 1$ (it converges slowly for $x = 1$).

Write a function `myTaylor` with the function declaration line **(24 points)**

begin code

```
function [y, diff_norm] = myTaylor(x, N)
```

end code

The inputs are:

- `x`: a column vector of the x data points.
- `N`: a scalar that represents the number of terms to be included in the Taylor series expansion. For example, the equation shown above represents a Taylor series expansion with `N = 4` terms.

The outputs are:

- `y`: a column vector with the same size as `x` which is the Taylor series expansion of the input vector `x` with `N` terms.
- `diff_norm`: the scalar norm of the difference between the Taylor series expansion `y` and the function $\log(1+x)$ evaluated at `x`.

Your function is not valid for $x \leq -1$. If any of the `x` entries is invalid, your function should display a warning `'You have values that would result in imaginary numbers.'`. For every `x` value that is less than or equal to -1, the corresponding logarithm will be `NaN`. When calculating `diff_norm`, exclude the `NaN` entries.

If any of the `x` value is greater than or equal to 1, the logarithm should be calculated by making use of the mathematical identity

$$\begin{aligned}\log(1+x) &= -\log\left(\frac{1}{1+x}\right) \\ &= -\log\left(1+\left(\frac{1}{1+x}-1\right)\right) \\ &= -\log(1+s)\end{aligned}$$

where $s = \frac{1}{1+x} - 1 = -\frac{x}{1+x}$ and $|s| < 1$. It is important to notice that for this case, the value of `x(i)` will still need to be calculated approximately (i.e find a way to incorporate the approximating Taylor series for the identity.)

Include the following test case in your template file.

begin code

```
>> x = linspace(-2,2)';
>> N = 3;
[y, diff_norm] = myTaylor(x, N);
Warning: You have values that would result in imaginary numbers.
> In myTaylor (line 14)
>> diff_norm
diff_norm =
    3.3190
```

end code

Additionally, you may want to generate a graph to compare your approximation `y` and the true value just like in Figure 1.

Note: The figure will not be graded but it serves as a quick way to check the output of your function.

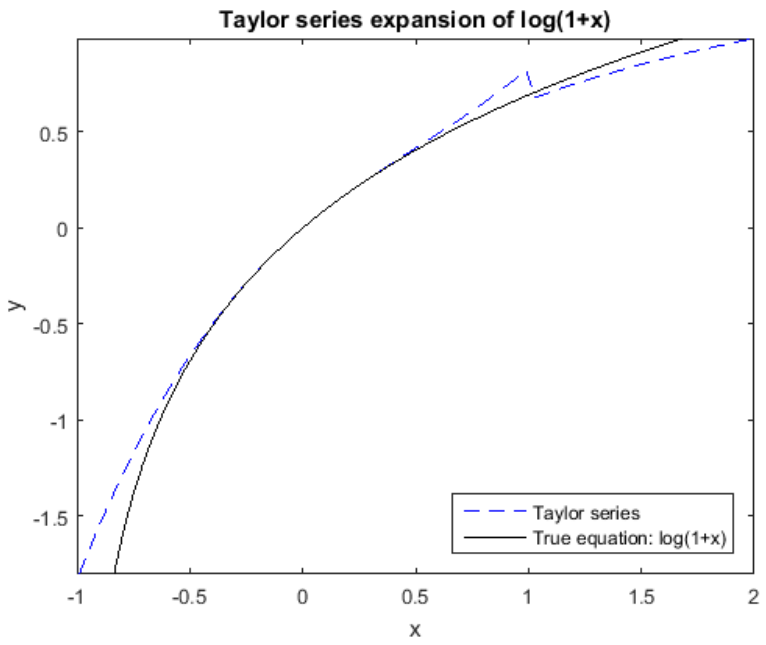

Figure 1: Result of Taylor series expansion

2. You are interested in understanding the mechanical properties of a new material under constant load. To do this, you place a load on the material and track the displacement (measured in mm) over a period of 60 minutes. In this example, your independent variable time is stored in the variable `t` and your dependent variable displacement is stored in the variable `d`. Now that you have a set of `t` and `d` data points, you want to interpolate a curve to fit the data. **(24 points)**

   Write a function `myInterpPlotter` with the function declaration line

   begin code
   ```
   function D = myInterpPlotter(t,d,T,option)
   ```
   end code

   The inputs are:

   - `t`: an n-by-1 column vector containing the time data points.
   - `d`: an n-by-1 column vector containing the displacement data points.
   - `T`: an m-by-1 column vector containing the time instances for which an interpolation is desired. In general, m $\geq$ n.
   - `option`: a string, containing either `'nearest'`, `'linear'`, `'cubic'`, `'regression'`, or `'cubicpoly'`.

   The output is

   - `D`: an m-by-1 column vector containing the interpolated displacement data.

   Use the MATLAB built-in function `interp1` to do the **interpolation** for the options `'nearest'`, `'linear'`, and `'cubic'`. Additionally, if the content of `option` is invalid, generate an error `'Option selection is incorrect.'` using the `error` function.

For the options 'regression' and 'cubicpoly', perform **regression** of the data values on a straight line and a cubic polynomial respectively. That means, you should solve for the coefficients of the line by representing the problem in the $Ax = b$ form, and evaluate T on the equation representing the line. Use backslash operator $\backslash$ to solve for the coefficients.

Include the following test cases in your template file.

begin code

```
>> t = [0 6 9 21 36 42 57 60]';
>> d = [ 1.0 1.2214 1.3499 1.4235 1.4567 1.5217 1.6063 1.6708]';

>> D1 = myInterpPlotter(t,d, linspace(0,60,100)', 'nearest');
>> D2 = myInterpPlotter(t,d, linspace(0,60,100)', 'linear');
>> D3 = myInterpPlotter(t,d, linspace(0,60,100)', 'cubic');
>> D3 = myInterpPlotter(t,d, linspace(0,60,100)', 'regression');
>> D3 = myInterpPlotter(t,d, linspace(0,60,100)', 'cubicpoly');
```

end code

Additionally, you may want to generate a graph of your data points (t versus d) as blue circles) as well as the interpolated line (T versus D) as a blue solid line to compare the result with Figures 3 to 7 in the **Appendix**.

Note: The figures will not be graded but they serve as a quick way to check the output of your function.

3. In this problem you will use interpolation to refine a image. From bcourses, you can download imdata.mat (class double), which is a 2-D array data storing a image that corresponds to a sample image. You can show this color image with the following code **(13 points)**

begin code

```
load imdata.mat
figure
imagesc(data)
```

end code

If you want to display the image in black, white, and gray, just add the following command.

begin code

```
colormap gray
```

end code

You need developing a function with the following head

begin code

```
function refinedimagedata = refineimage(imagedata)
```

end code

where input is the name of the image data file (string), and output is refined image data.

In the function, use built in function interp2 to create interpolated image data. Before using it, please learn this built-in function yourself and use the "spline" method to do the 2-D interpolation.

Test your function with the following code

begin code

```
load imdata.mat
figure(1)
imagesc(data)
colormap gray
axis image
axis off
refinedimagedata = refineimage('imdata.mat')
figure(2)
imagesc(refinedimagedata);
colormap gray
axis image
axis off
```

end code

Part 2: Root Finding

4. The spring shock system in automobiles is an example of a real mechanical system which involves the deflection of nonlinear springs that would demonstrate a damping effect over time (see behavior response in Figure 2 below). The resistance force of the spring F (N), with respect to time t (s), is given by the following equation

$$F(t) = k_1(\cos(\pi t))(e^{-\frac{t}{k_2\pi}})$$

where is a k_1 and k_2 are defined as spring constants (N/s) **(24 points)**.

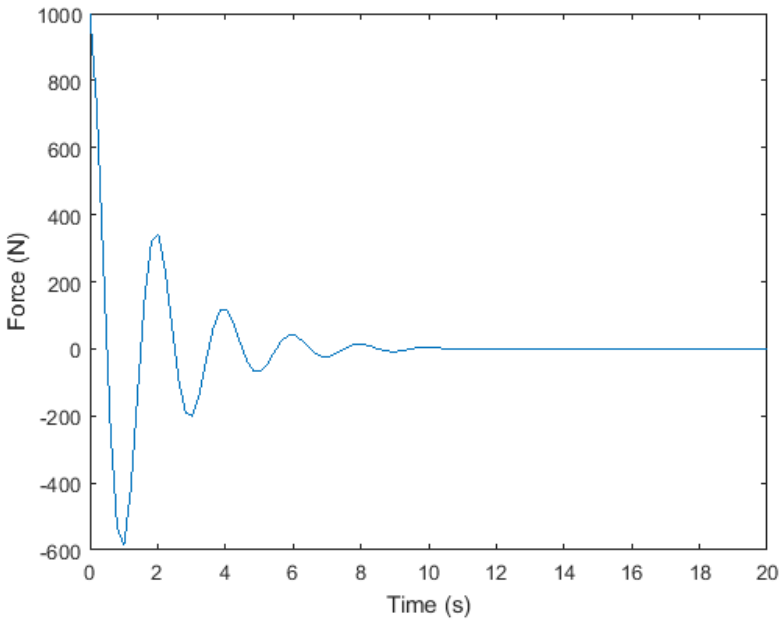

Figure 2: Damping effect with spring constants $k_1 = 1000$ N/s and $k_2 = 0.6$

We will use the above equation to describe the force on a car's shock system following a speed bump encountered at $t = 0$ seconds. We want to find out a time instance near $t = 5.5$ seconds when the resistance force F is equal to zero. In order to do that, we need to find the roots of $F(t)$.

You will find the roots using three different methods. In part (a), you will use the MATLAB built-in function `fzero`. In part (b), you will use the **Newton-Raphson method**. In part (c), you will use the **bisection method**.

Write a function `damping` with the function declaration line

begin code
```
function S = damping(k1,k2,initial)
```
end code

The inputs are:

- `k_1`, `k_2`: the spring constants defined above.
- `initial`: the time instance of interest, near which we want to find a root.

The output is:

- `S`: a 1-by-1 structure array. Its fieldnames and content will be described in detail in the following subproblems.

In your function, represent $F(t)$ as a function handle stored in variable `F`. Also, create a variable `tol = 1e-6` which represents the value of the tolerance required for the bisection method and the Newton-Raphson method.

(a) Use `fzero` to determine the root of `F` with `initial` as the guess location. Store the root to `S` in the fieldname `fzero_root`.

(b) Within `damping.m`, write a subfunction `myNewton` with function declaration line

begin code
```
function newton_root = myNewton(f, df, x0, tol)
```
end code

which will **recursively** calculate the root nearest to `x0` given the function handle `f`, the function handle of the derivative of `f` (which you need to calculate on your own) `df`, the initial guess `x0`, and the tolerance `tol`.

Store the root found using Newton-Raphson method to `S` in the fieldname `newton`.

(c) Within `damping.m`, write a subfunction `bisection` with function declaration line

begin code
```
function  bisect_root = bisection(f, a, b,tol)
```
end code

which will **recursively** calculate the root located between `a` and `b` given the function handle `f`, the endpoints of the range `a` and `b`, and the tolerance `tol`.

Use a range of ± 0.5 (make sure to check the signs of the function evaluated at the endpoints) and store the root found using bisection method to `S` in the fieldname `bisect_root`.

The expected output is shown below.

begin code

```
>> k1 = 1000; k2 = 0.6; initial = 5.5;
>> S = damping(k1,k2,initial)
S =
        fzero_root: 5.5000
        newton: 5.5000
        bisect_root: 5.5000

>> k1 = 1000; k2 = 0.6; initial = 4.97;
S = damping(k1,k2,initial)
S =
        fzero_root: 4.5000
        newton: 9.5000
        bisect_root: 4.5000
```

end code

5. In physical sciences and electrical engineering, dispersion relations describe the effect of dispersion in a medium on the properties of a wave traveling within that medium. A dispersion relation relates the wavelength or wavenumber of a wave to its frequency. For example, for waves propagating inside the ocean(called internal waves), the dispersion relation can be written as,

$$f(\omega, k) = \omega^2 - \frac{k}{a\sqrt{\frac{1}{\omega^2} - 1}} \tan(k\sqrt{\frac{1}{\omega^2} - 1}) = 0$$

where ω is the wave frequency, k is the wave number, a is a physical parameter specified by the ocean conditions.

The equation is transcendental and possesses infinite number of roots at a given frequency ω. We know that $k = n\pi\sqrt{\omega^2/(1-\omega^2)}$ where $n = 1, 2, 3, 4...\infty$ are approximate solutions to the equation. For instance, $k = \pi\sqrt{\omega^2/(1-\omega^2)}$ and $k = 2\pi\sqrt{\omega^2/(1-\omega^2)}$ are very close to the two solutions in the above equation. Physically n represents different branches of the wave solutions and is called mode number.**(15 points)**

(a) You will make use of `fzero` in Matlab to solve this equation at a given frequency($0 < \omega < 1$) and mode number.

begin code

```
function [k]=DSP(omega,n,a)
end
```

end code

(b) Plot the first 5 modes(i.e., $n = 1, 2, ...5$ respectively) in the same figure. Choose ω in $(0.60, 0.99)$ and $a = 0.1$. This figure will not be graded.

Appendix: Result Figures for Problem 2

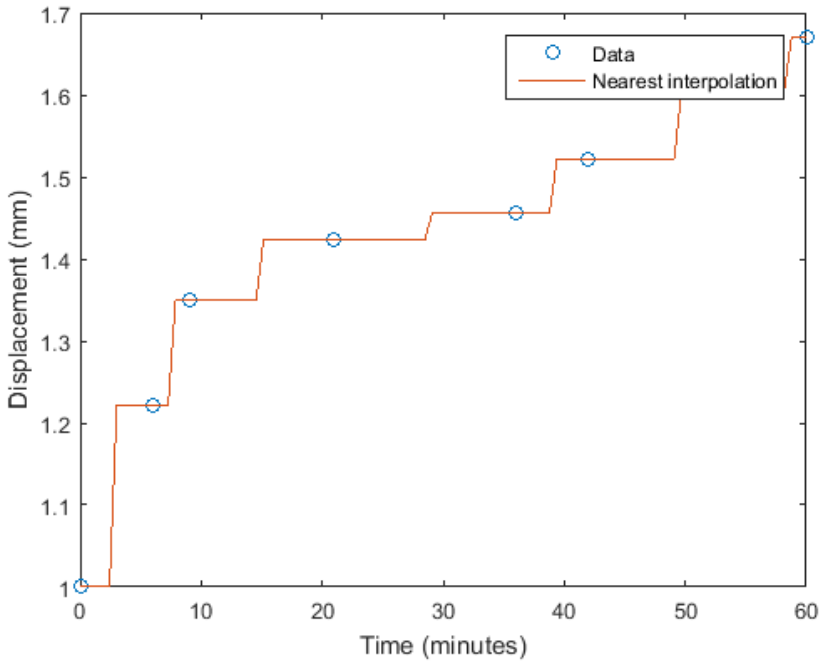

Figure 3: Result from `nearest` interpolation

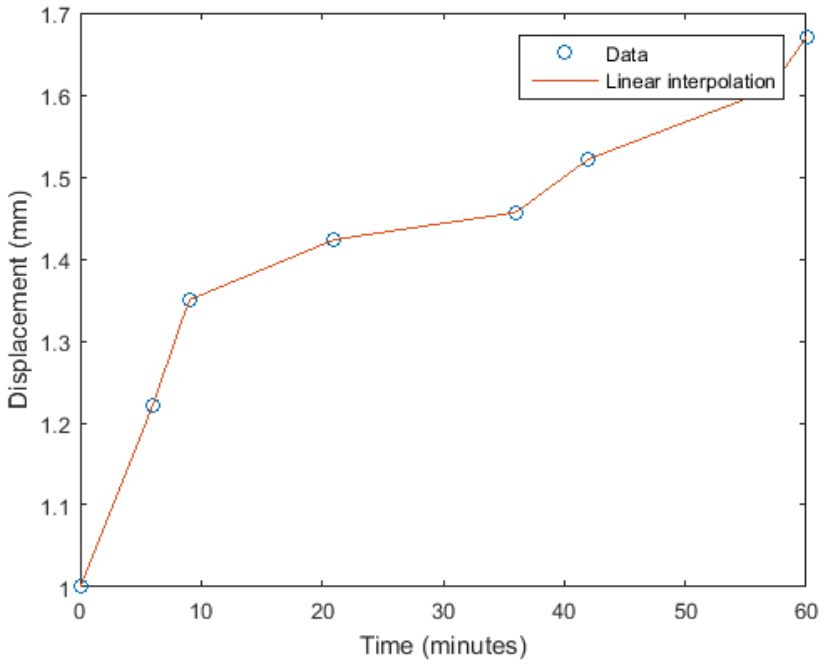

Figure 4: Result from `linear` interpolation

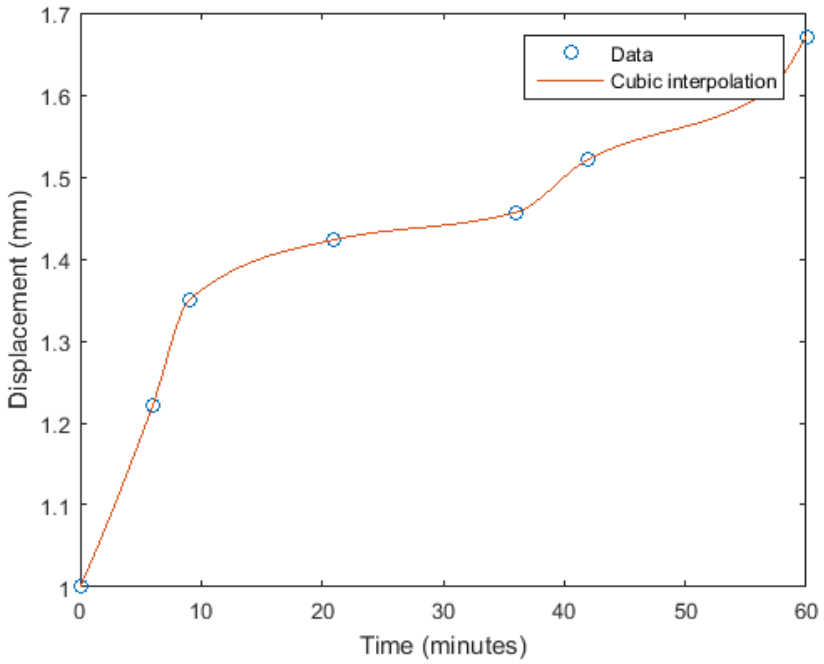

Figure 5: Result from `cubic` interpolation

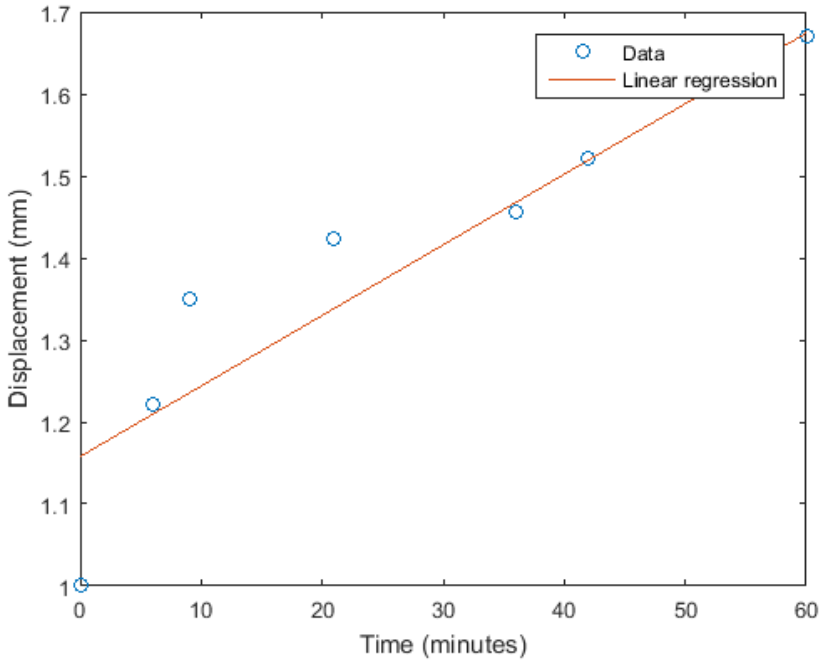

Figure 6: Result from `regression`

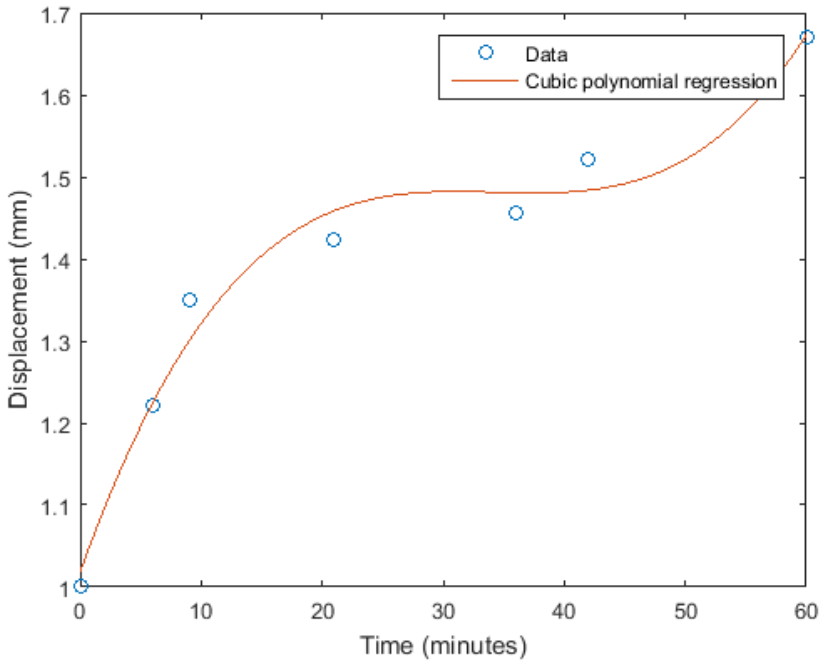

Figure 7: Result from `cubicpoly`