Lab Assignment #7
Topics: Curve fitting, interpolation, numerical root finding
Due 4/3/2020 at 11:59pm on bCourses

The assignment will be graded by an autograder which will check the outputs of your functions. For your functions to be scored properly, it is important that the names of your functions **exactly match** the names specified in the problem statements, and input and output variables to each function are in the **correct order** (i.e. use the given function header exactly). Instructions for submitting your assignment are included at the end of this document.
If any problem specifies a certain method and/or algorithm to be used, you must implement that method and/or algorithm or you will receive a zero (0) on that problem.
On this assignment and future assignments, you are required to check that the input follows the format we provide in the table. For example, if we tell you that x is a positive integer, your code must evaluate if the x provided is a positive integer. If it is not valid, your program should terminate and you should display (disp) a message to the user why it has terminated.

1 Interpolation methods

1.1 The Lagrange polynomial

`function [y_dot,p] = myLagrange(x,y,x_dot)`		
Input	Type	Description
`x`	`1xM double`	A row vector with the x data that you should fit your Lagrange polynomial to
`y`	`1xM double`	A row vector with the y data that you should fit your Lagrange polynomial to
`x_dot`	`1xN double`	A row vector of target points that you should evaluate your Lagrange polynomial at
Output		
`y_dot`	`1xN double`	The value of your Lagrange polynomial at the given points
`p`	`1xM double`	The polynomial coefficients of the Lagrange polynomial (from highest to lowest order)

Details

The Lagrange polynomial for a given data set is defined as the polynomial of least degree which passes through all the points of the data set. For a set of M data points, the Lagrange polynomial is guaranteed to be of degree $(M-1)$ or lower, and can therefore be written in the following form:

$$L(x) = a_{M-1}x^{M-1} + a_{M-2}x^{M-2} + ... + a_1x + a_0$$

where x is the independent variable and a_i, $i = 0, 1, ..., M-1$ are the coefficients of the polynomial.

In this question, you will write a function that calculates the coefficients of the Lagrange polynomial for a given data set by setting up a system of equations. You will then use that Lagrange polynomial to generate interpolated data for a given set of target points `x_dot`.

Restriction: You should not use Matlab's `polyfit` or `polyval` functions.

Tips

- In this problem, you can assume that $m > 1$, all the elements of x and y won't be NaN, Inf, or -Inf, and that the elements of x are unique.
- You can solve this question by setting up a linear system for the polynomial coefficients and solving it.
- Note that the list of polynomial coefficients is listed from highest to lowest order, i.e. `p(1)` $= a_{M-1}$, while `p(end)` $= a_0$.

1.2 Comparing interpolation methods

`function [fig, y_dot] = compareInterpMethods(x,y,x_dot,interp_methods)`		
Input	Type	Description
`x`	`1xM double`	A row vector with the x data
`y`	`1xM double`	A row vector with the y data
`x_dot`	`1xN double`	A row vector of x points that you should evaluate your interpolation methods for
`interp_methods`	`1xQ cell`	A cell array of `char` arrays, listing which methods to compare
Output		
`fig`	`matlab.ui.Figure`	A figure handle to the resulting figure
`y_dot`	`QxN double`	An array of y points that represent the interpolated value corresponding to different `interp_methods`

Details

For this problem, you will write a function that compares a set of interpolation methods specified by `interp_methods`. Your function will then plot the interpolated values along with the original data for comparison, as shown in the figure below:

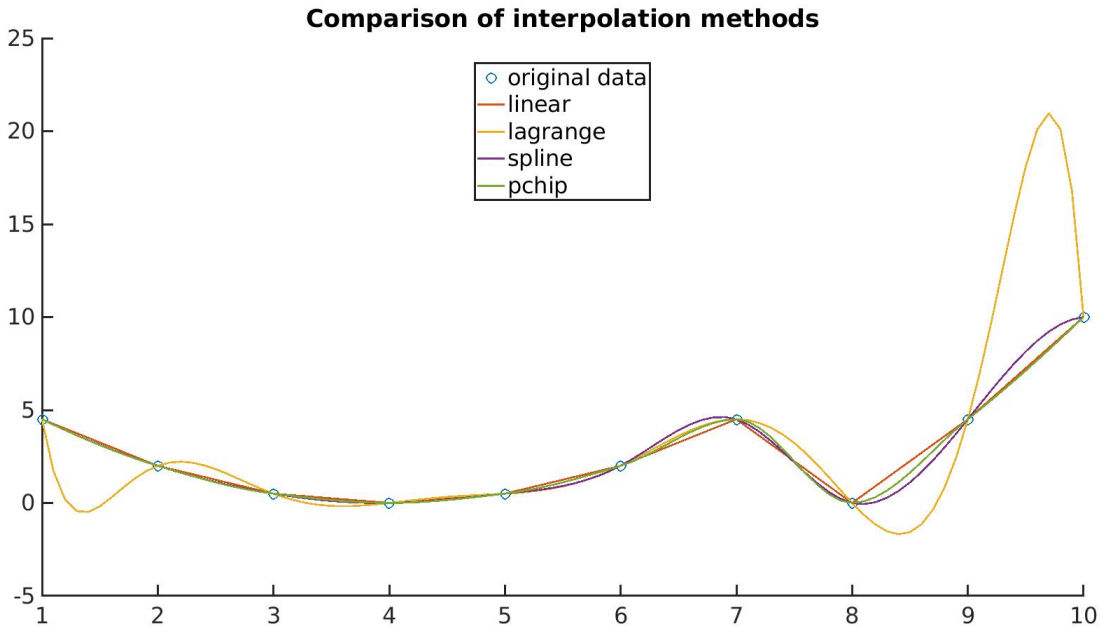


Figure 1: Example from tester

`interp_methods` will be a cell array containing a set of the following `char` arrays:

- `'linear'`. In this case, you should perform linear interpolation using Matlab's built in `interp1` function.
- `'spline'`. In this case, you should perform interpolation using Matlab's built in `spline` function.
- `'pchip'`. In this case, you should perform linear interpolation using Matlab's built in `pchip` function.
- `'lagrange'`. In this case, you should perform interpolation using your `myLagrange` function.

Again, use `fig=figure;` **before** you do any plotting in order to create the figure handle. All plotting commands will then be applied to the most recent figure by default.

Tips

- You can assume that $m > 1$, that all the elements of x and y won't be NaN, Inf, or -Inf, and that the elements of x are unique.
- Your figure should contain a legend that labels each line plotted with its corresponding interpolation method and labels `'original data'` for the original data.
- You may assume that `interp_methods` will not have any character arrays in it other than those listed above and will not have any duplicates.
- `interp_methods` may be empty. If this is the case, you should only plot the original data.

2 Root Finding: Methods

2.1 Newton's Method

`function [rt, n_iter] = newtonsMethod(f, df, x0, tol, max_iter)`		
Input	Type	Description
`f`	`function_handle`	A function handle representing the function $f(x)$
`df`	`function_handle`	A function handle representing the derivative of $f(x)$
`x0`	`1x1 double`	The initial root guess
`tol`	`1x1 double`	A positive value, specifying the tolerance within which you should find the root
`max_iter`	`1x1 double`	The maximum number of iterations of Newton's method that can be applied
Output		
`rt`	`1x1 double`	The value of the root as estimated by Newton's method
`n_iter`	`1x1 double`	The actual number of iterations performed

Details

Here you will program the Newton-Raphson root finding method with details specified as follows. Your code will continue performing iterations of this root finding method until it has found the root within the given tolerance. The function should return when it has found a root for which `f(rt)` is within `tol` of 0 (inclusive), or after `max_iter` iterations have been performed.

- If the initial guess for the root is an actual root within the specified tolerance, `n_iter` should be 0.
- If the function has not found a root after performing `max_iter` iterations, you should return `rt = NaN` and `n_iter = max_iter`.
- If `df` is ever 0 at the time you apply Newton's method during iteration, you should return `rt = NaN` and `n_iter = Inf`.
- **MATLAB built-in functions `fzero` and `roots` should NOT be used.**

2.2 Bisection Method

`function [rt, n_iter] = bisectionMethod(f, interval, tol, max_iter)`		
Input	Type	Description
`f`	`function_handle`	A function handle representing the function $f(x)$
`interval`	`1x2 double`	The initial interval of x-values to start with
`tol`	`1x1 double`	A positive value, specifying the tolerance within which you should find the root
`max_iter`	`1x1 double`	The maximum number of iterations of bisection method that can be applied
Output		
`rt`	`1x1 double`	The value of the root as estimated by bisection method
`n_iter`	`1x1 double`	The actual number of iterations performed

Details

Here you will program the bisection root finding method with details specified as follows. The function should return when it has found a root for which `f(rt)` is within `tol` of 0 (inclusive) or after `max_iter` iterations have been performed.

- If the midpoint of the initial interval is an actual root within the specified tolerance, `n_iter` should be 0.
- If the function has not found a root after performing `max_iter` iterations, you should return `rt = NaN` and `n_iter = max_iter`.
- If the value of f at the endpoints of your initial interval are the same sign, you should return `rt = NaN` and `n_iter = Inf`.
- **MATLAB built-in functions `fzero` and `roots` should NOT be used.**

2.3 MATLAB's built-in function

`function [true_rts, my_rts, n_iters] = myFzero(p, x0, interval, max_iter)`		
Input	Type	Description
`p`	`1xN double`	The coefficients of a polynomial
`x0`	`1x1 double`	The initial root guess
`interval`	`1x2 double`	An interval that should contain the root
`max_iter`	`1x1 double`	The maximum number of iterations of Newton's method or bisection method that should be applied
Output		
`true_rts`	`1x(N-1) double`	The roots of the polynomial, as found by Matlab's `roots` function
`my_rts`	`1x4 double`	The root found by Matlab's `fzero` function (given `x0`), `fzero` (given `interval`), your `newtonsMethod` function, and your `bisectionMethod` function, in that order.
`n_iters`	`1x4 double`	The number of iterations performed by Matlab's `fzero` function (given `x0`), `fzero` (given `interval`), your `newtonsMethod` function, and your `bisectionMethod` function, in that order.

Details

For this problem, you will write a function that compares the performance of your Newton's method and bisection method function with Matlab's built-in root finding functions `fzero` and `roots`. You function should do the following:

1. Find all of the roots of the polynomial using Matlab's `roots` function.
2. Find a root of the polynomial and the number of iterations by Matlab's `fzero` function, when given `x0` is a starting value.
3. Find a root of the polynomial and the number of iterations by Matlab's `fzero` function, when given `interval` as a starting interval.
4. Find a root of the polynomial and the number of iterations by your `newtonsMethod` function, when given `x0` as a starting value.
5. Find a root of the polynomial and the number of iterations by your `bisectionMethod` function, when given `interval` is a starting interval.

You should run the functions you have written with number of iterations up to the given `max_iter` and set `tol` equal to that of the tolerance used by `fzero`. To obtain the number of iterations used by `fzero`, you should make use of the 4th output of the `fzero` function, which is a `struct` containing iteration information within its fields. `fzero` uses two different root finding methods within a single run and the number of iterations that are performed for each

those methods are stored in the fields `intervaliterations` and `iterations`. Your answer should be the sum of the two. You can check more information about fzeros here.

Tips

- The root function calculates the roots of a single-variable polynomial represented by a vector of coefficients.
- The input `p` represents the coefficients of the polynomial from highest to lowest order.
- You may assume that the sign of the polynomial is different for the two values given in `interval`.
- You can access fields within a `struct` by using the `.` operator. For example, if you store the 4th output of fzero in a variable called `iter_info`, you can access the field `intervaliterations` by typing `iter_info.intervaliterations`.

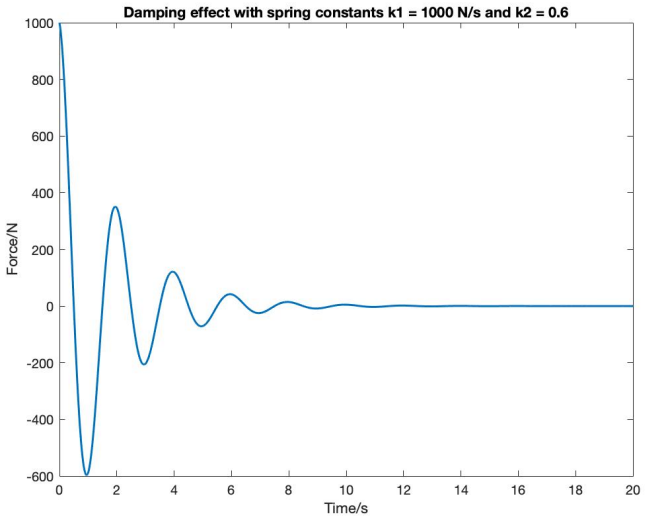
3 Applying root finding to an engineering problem

`function [t] = damping(k1,k2,x0,interval,tol, max_iter)`		
Input	Type	Description
`k1`	`1x1 double`	spring constant k_1
`k2`	`1x1 double`	spring constant k_2
`x0`	`1x1 double`	The initial guess to start with
`interval`	`1x2 double`	The initial interval (inclusive) of x-values to start with
`tol`	`1x1 double`	A positive value, specifying the tolerance which you should find the root within
`max_iter`	`1x1 double`	The maximum number of iterations of Newton's method or bisection method that can be applied
Output		
`t`	`1x2 double`	The root from Newton's Method and Bisection Method

Details

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 below figure). The resistance force of the spring $F(N)$, with respect to time t (s), is given by the following equation

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



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. Given an approximation, we want to find the exact time when $F = 0$. For example, we want to find out a time near $t = 5.5$ seconds when the resistance force F is equal to zero. For the Newton method, we can take $t = 5.5$ as the initial guess to start with. For the bisection method, we take the time interval $[5.5-0.5, 5.5+0.5]$ to start with.

4 Newton's method in higher dimensions

`function [rts, fig] = newtons2D(f, dfdx, dfdy, X, Y, pts, n_iter)`		
Input	Type	Description
`f`	`function_handle`	A function handle representing the function $f(x, y)$
`dfdx`	`function_handle`	A function handle to $\frac{\partial f}{\partial x}(x, y)$
`dfdy`	`function_handle`	A function handle to $\frac{\partial f}{\partial y}(x, y)$
`X`	`MxN double`	A grid of x values on which to plot your function
`Y`	`MxN double`	A grid of y values on which to plot your function
`pts`	`Qx2 double`	A curve of (x, y) points to initialize Newton's method with
`n_iter`	`1x1 double`	The number of iterations of Newton's method to apply
Output		
`rts`	`Qx2 double`	The value of the roots for each point in `pts` as estimated by Newton's method
`fig`	`1x1 matlab.ui.Figure`	A figure handle to the 3D plot of the problem

Details

It is possible to generalize Newton's method to be applied to functions of more than one variable. For this problem, you will write a Matlab function that applies Newton's method to a function of 2 inputs and creates a 3D plot to visualize the results. Note that for a function of 2 variables x and y, the roots are where the surface intersects with the plane z = 0.

Newton's method for a function of more than one variable can be written as follows:

$$\vec{x}_{n+1} = \vec{x}_n - \frac{f(\vec{x}_n)\vec{\nabla} f(\vec{x}_n)}{||\vec{\nabla} f(\vec{x}_n)||^2}$$

Where $\vec{x}_n$ is the n^{th} iteration of Newton's method for a vector of inputs $\vec{x}$. Note that $\vec{x_n}$ represents a 2D input vector $[x_n, y_n]$. $\vec{\nabla}$ is the gradient operator, and $||\quad||$ is the vector magnitude operator.

For a 2D function $f(x, y)$, your function should apply Newton's method to a curve of initial points and then plot the surface $z = f(x, y)$ in 3D space. Also, plot the plane $z = 0$ as well as the curve of roots found from Newton's method, as shown below.

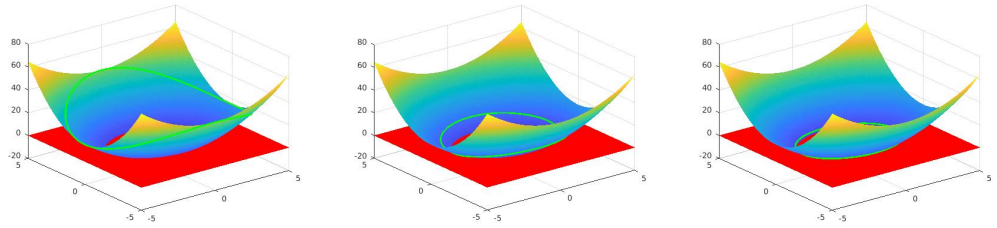


Figure 2: Test case 1-3 : 0-2 iterations of Newton's method

Tips

- Again, use `fig=figure;` **before** you do any plotting in order to create the figure handle. All plotting commands will then be applied to the most recent figure by default.
- You should use the MATLAB built-in function surf to plot the surface defined by $f(x, y)$ as well as the plane $z = 0$. You should use MATLAB built-in function plot3 to plot the roots.
- The gradient operator in two dimensions is defined as $\vec{\nabla} = \vec{i}\frac{\partial f}{\partial x} + \vec{j}\frac{\partial f}{\partial x}$.
- The magnitude of a vector $\vec{v} = (v_x, v_y)$ in two dimensions is defined as $||\vec{v}|| = \sqrt{v_x^2 + v_y^2}$
- You may assume that the given function handles `f`, `dfdx`, and `dfdy` can take arrays or matrices of x and y values as inputs and correctly produce outputs for them on a element-by-element basis.

Submission Guidelines

When you are ready to test one or more of your function(s), download the `Lab07Tester.m` script from bCourses and run it in the same directory as your function(s). This will run a series of test cases for your function(s) and display the outputs. You should check these results with the example outputs displayed. Note that the series of tests that the script will run is not exhaustive and that it is up to you to make sure that your function fully satisfies the requirements of the problem. If you want to run all the test cases for one function, select that part of the tester script and push the `Run Section` button. If you wish to only check a single test case, you may do so by highlighting the appropriate lines in Matlab and then pressing F9 (on Windows and linux) and `shift + fn + F7` (on a Mac) to run the selected text in the Command Window.

The script will also pack your function into a zip file for final submission. Your function file names and headers should match the examples given in the problem statement. If they do not match, the tester script will not be able to find them and will issue a warning. If you do not correct this problem, your missing function(s) will not be able to be graded and **you will receive 0 points**, so be sure to check carefully! Make sure to upload the `Lab07.zip` file to bcourses before the deadline. You may resubmit your zip file by re-uploading to bcourses before the deadline, but please be aware that only the most recent zip submission is considered for grading. That said, it is critical that you include your most recent version of **all** of your functions in each new zip submission.

Your final zip file for this assignment should contain the following:

- myLagrange.m
- compareInterpMethods.m
- newtonsMethod.m
- bisectionMethod.m
- myFzero.m
- damping.m
- newtons2D.m