Parker Lange

App Sci

Project 1

10/02/2019

Problem 1)

1. Task: Write a function to return the intervals of a one iteration golden section search.

**function** **[**lowerX**,**upperX**]** **=** GetGoldenRulePoints**(**lowerBound**,**upperBound**)**

%

% [lowerX, upperX] = GetGoldenRulePoints(lowerBound, upperBound)

%

% This is a function used to get the starting x values for the golden rule

% search. It will return these two x values.

% Input:

% lowerBound: This is the lower bracket of our search

% upperBound: This is the upper bracket of our search

% Outuput:

% lowerX: This is the lower x value for the start of our search

% upperX: This is the upper x value for the search.

lowerX **=** upperBound **-** **((**1.618 **-** 1**)** **\*** **(**upperBound **-** lowerBound**));** % Calculates our lower x value

upperX **=** lowerBound **+** **((**1.618 **-** 1**)** **\*** **(**upperBound **-** lowerBound**));** % Calculates our upper x value

**end** % end function

1. Task: Use the function from 1A, and calculate x1, and x2, given the interval [5, 7]. Then, throw out the left most area and repeat. Next, do the same for the right most, and finally throw out the last left most area.

Below is the code for this, as well as a graph showing the X values after each run of this function.

% Problem 1

% base function call with bounds of 5 and 7

**[**lowerX0**,** upperX0**]** **=** GetGoldenRulePoints**(**5**,** 7**);**

% second function call, throwing out the left region

**[**lowerX1**,** upperX1**]** **=** GetGoldenRulePoints**(**lowerX0**,** 7**);**

% third function call, throwing out right region

**[**lowerX2**,** upperX2**]** **=** GetGoldenRulePoints**(**lowerX0**,** upperX1**);**

% the final bounds after throwing out the final left region are lowerX2, and upperX1

|  |  |  |  |
| --- | --- | --- | --- |
| x | First Function Call | Second Function Call | Third Function Call |
| x2 (lower x) | 5.7640 | 6.2362 | 6.0558 |
| x1 (upper x) | 6.2360 | 6.5278 | 6.2361 |

This is the table for the x values after each function call

The final bounds after throwing out the last left region are lower x: 6.0558, upper x:6.5278

Problem 2)

1. Task: Find the derivative of the function:
2. Task: Write a Newton Raphson root solver function.

**function** **[**rootLocation**,**numIterations**]** **=** NewtonRaphson**(**initialGuess**,**func**,** ...

funcDerivative**,** error**,** maxIterations**)**

%

% function [rootLoc,numIterations] = NewtonRaphson(initialGuess,func, ...

% funcDerivative, aoe, maxIterations)

%

% This is a function using the newton raphson method to calculate a root

% This calculation is done based on the func, and funcDerivative

% Inputs:

% initalGuess: This is our initial guess for the root of the function

% func: This is the function we are evaluating at

% funcDerivative: This is the derivative of the function used

% error: This is our desired margin of error. Once we reach this we are done

% maxIterations: This is the maximum number of times we want to loop

% through this function, searching for a root

% Outputs:

% rootLocation: This is our location of the root of the func

% numIterations: This is the number of iterations taken to find root

% If the user doesnt inclue 5 items in constructor

% or if our max iterations is empty

% then set max iterations to 100

**if** **(**nargin **<** 5 **||** isempty**(**maxIterations**))**

maxIterations **=** 100**;**

**end** % end if

% If the user doesnt include 4 items in constructor

% or if the error is empty, then set our error

**if** **(**nargin **<** 4 **||** isempty**(**error**))**

error **=** 1e-6**;**

**end** % end if

numIterations **=** 0**;** % sets our initial iterations

y0 **=** func**(**initialGuess**);** % gets our y naught

y0Prime **=** funcDerivative**(**initialGuess**);** % gets y prime naught

relativeChange **=** abs**(**error**)** **\*** 100**;** % Forces the function to run once

% This is a while loop for finding our intercept

% while our relative change is greater than our error,

% and while the iterations is less than our max

% we continue calculations for finding the intercept

**while** **((**relativeChange **>** error**)** **&&** **(**numIterations **<** maxIterations**))**

numIterations **=** numIterations **+** 1**;** % increment iterations

newPointX **=** initialGuess **-** **(**y0 **/** y0Prime**);** % find our new x value

% this calcualtes our new relative change

relativeChange **=** abs**((**newPointX **-** initialGuess**)** **/** newPointX**);**

initialGuess **=** newPointX**;** % assign our 'guess' to our new point

y0 **=** func**(**initialGuess**);** % calculate y naught

y0Prime **=** funcDerivative**(**initialGuess**);** % calculate y naught prime

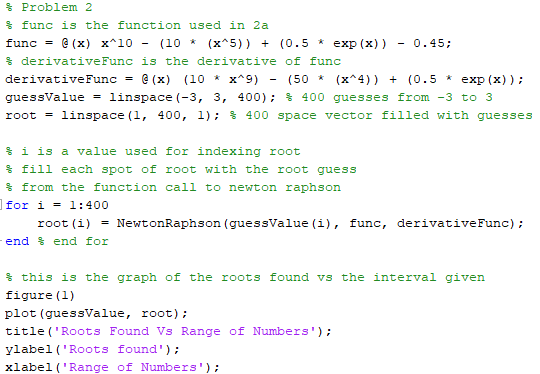
**end** % end while

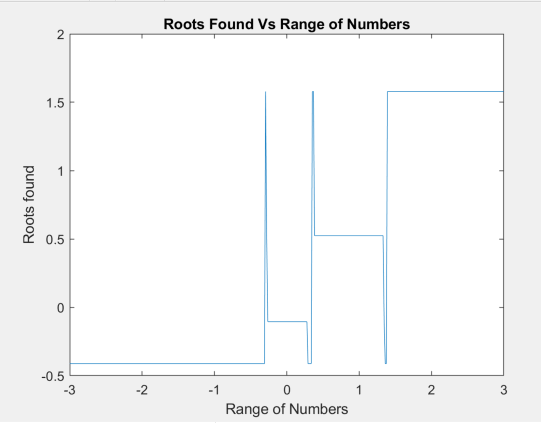
% once we are done looping, we assign

%the root location to our initial guess

rootLocation **=** initialGuess**;**

1. Task: Create a vector with several hundred points ranging from -3 to 3, then use these points as our guesses and call the Newton Raphson method. Then, store the result of the function call in a vector of same size as the guesses.



The four values that show up in the root vector are: -0.4120, -0.1057, 0.5243, and 1.5785.

This is the graph showing the roots at each location from -3 to 3.

1. Task: Does N-R always find the closest root? Find at least one spot where it doesn’t, and explain what happens in words.

No, Newton Raphson does not always find the closest root. In my vector of 400 guesses evenly spaced from -3 to 3, at a guess value of 0.02932, it tells me the root is -0.4120. This happens when we are near a maximum or minimum value of the function. In the case above, it skips to the -0.4120 value because it is a point near a maximum, so the slope is not very steep. So, when it does the necessary calculations to find the next X value, because y’ is closer to 1, it goes back farther and it skips over the closest root it had previously been attaching to.

1. Task: Use the bisection function to compare the iterations taken to find the root against the Newton Raphson method.

% Bisection function call. Bracket of [1,3] and an error of 1e-10.

**[**bisecIterations**,** biSecVal**]** **=** Bisection**(**func**,** 1**,** 2**,** 1e-10**);**

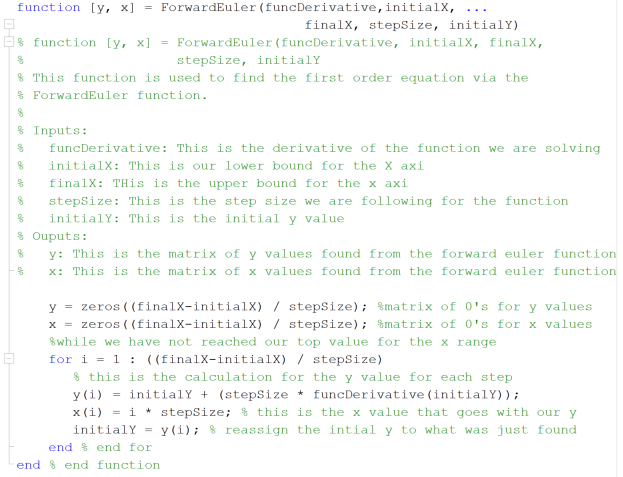
% N-R call, we pass in the function, its derivative, the error, and guess

% we set the guess value to 3, to make the comparison fair.

**[**rootValue**,** iterationsValue**]** **=** NewtonRaphson**(**2**,** func**,** derivativeFunc**,** 1e-10**);**

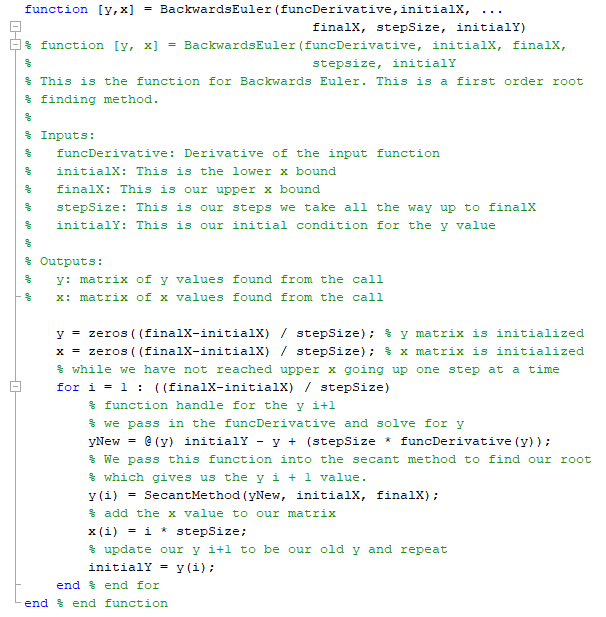
For the comparison of the two functions, I decided to use a bracketing size of [1, 2] for Bisection. This bracketing size was set up because there is a root roughly in the middle of that bracket. I figured this would somewhat help give the Bisection method an “advantage”. To help give the Bisection function as much of an advantage as possible, for the Newton Raphson function call, I set the guess as the edge of the bracket used for Bisection. When ran, both functions found the correct root. However, it took the Bisection method **32 iterations**, and took the Newton Raphson method **8 iterations**. As shown, even with all of the additional “advantages”, the Newton Raphson root finding method is still 4x faster at obtaining the root. This was a big difference in time taken.

Problem 3)

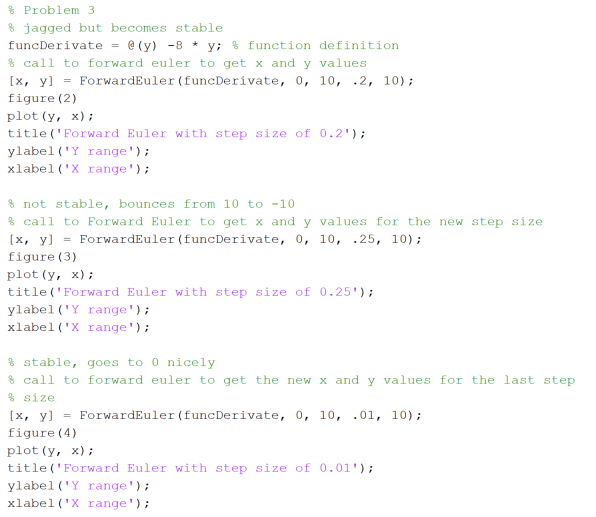
1. Task: Write a general-purpose Forward Euler Function

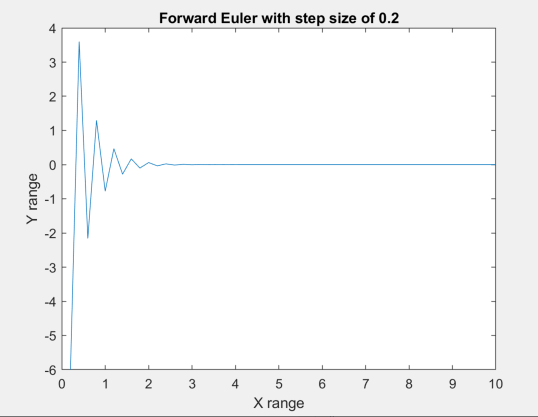
This is my function for the Forward Euler Function. Comments show what it does.

1. Task: Create a Backwards Euler function.

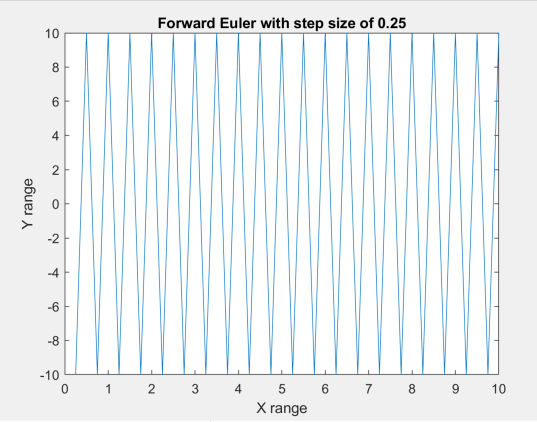
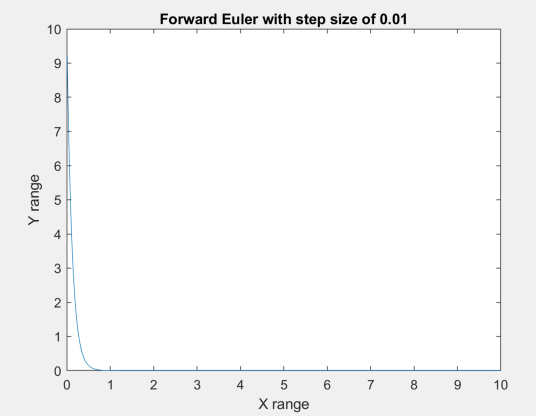


This is the Backwards Euler function. The comments describe the code above.

1. Task: Use the function given, and plot the result of Forward Euler with different step sizes.

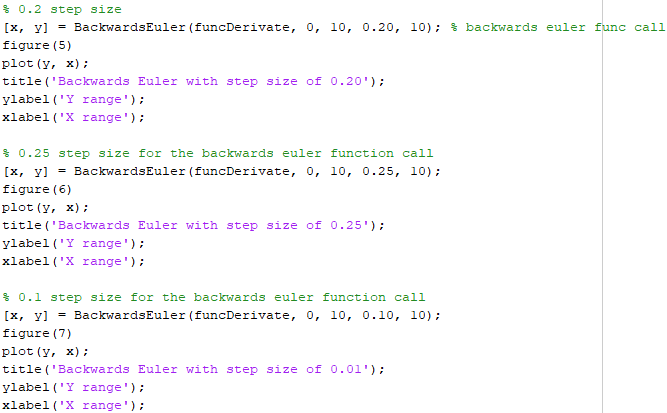
This is the code used to plot the 3 different Forward Euler functions. As shown, the steps sizes are 0.2, 0.25, and 0.1.

This is the graph Y vs X values, with a step size of 0.2

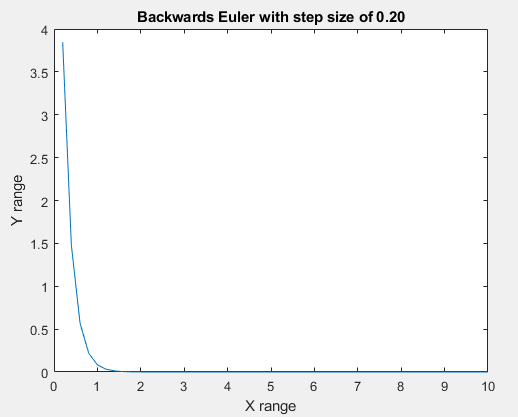
This is the Y vs X with a step size of 0.25

This is the Y vs X graph with a step size of 0.1

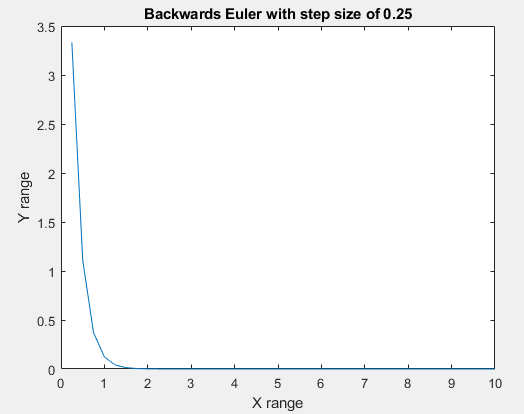
As shown in the 3 graphs above, the smaller the step size, the more stable the graph is. At 0.1 the graph is completely stable, and it is also stable at 0.2, However, at 0.25, the graph bounces continuously and never goes to 0.

1. Task: Use the Backwards Euler method to plot the same step sizes, and show the difference in the graphs.

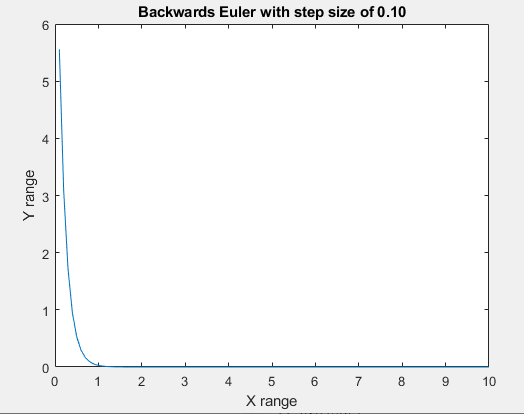
This is the code used to call and plot the Backwards Euler method for each step size.



This is the Y vs X graph for the step size of 0.2



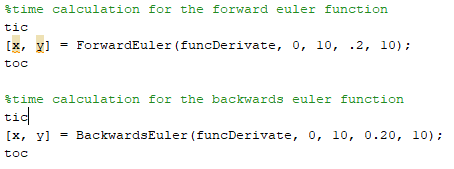
This is the graph of the step size of 0.25



This is the Y vs X graph with a step size of 0.1

As we can see, the Backwards Euler method is a lot more stable and does not have any bounces on the graph at the same step sizes as the Forward Euler. For 0.25 on Forward Euler, the graph went 10/-10 constantly, and the same step size on the Backwards Euler does not have this issue at all.

1. Task: Compare the two Euler methods with the tic and toc function



This is the code used to compare the time it takes for both of the Euler methods to finish.



This is the result of the code written above.

As we can see from the Tic and Toc functions used above, the Backwards Euler function takes two times as long as the Forward Euler to do the calculations. This shows what was talked about in class: The Forward Euler method is faster, but cannot use larger step sizes. The Backwards Euler function can handle any step size, but there is a cost to this, and that cost is the time taken for execution.