Numerical Analysis Project 1 Ramson Munoz 6374761

Writing the Code:

Inputs:

Inputs for the code are dependent on the method used for the root finding problem. In this analysis, we look at how the Bisection method, Newton's method, the Secant method, and Muller's method tackle the root finding problem for two given cubic functions. All of the methods take tolerance and max iterations as inputs to use as exit conditions. Further, all of the methods take a pointer to the function we want to test as an input. This allows us to generalize the methods to different functions.

Since the methods share the above as inputs, below describes inputs that are unique to each method.

Bisection Method:

The bisection method takes in two bounds to define a closed interval on which the function is continuous. We supply our function in the code with a left and right bound on which it will search for the root of the function.

Newton's Method:

Newton's method takes in two unique inputs, a pointer to the derivative of the function, and an initial guess that is sufficiently close to the target value. This means that it can be helpful to couple Newton's with another method like Bisection to get a ballpark figure.

Secant Method:

The secant method avoids the need to define the derivative of the function because it uses the mean value theorem to define the lines used to find our root. Instead, it uses a second initial guess to jump start the algorithm.

Mullers Method:

Mullers works by mapping parabolic to sections of the original function. So, it takes in three points to jump-start the algorithm.

Outputs:

Each of the methods will output an approximation for where our original function evaluates to 0.

Choice of Parameters:

Bisection Method:

We start with the bisection method to get a ballpark value since it is known to always converge for continuous functions. The left bound was set to -10 and the right bound was set to 9 since the function is cubic and the highest degree term is positive. Tolerance was set to the given value of 1E-4. Max iterations was set to 1000 because to ensure a value was reached. This method on the interval [-3,3] revealing the presence of a second root.

For part B, the interval [-10,9] was examined. By graphing the function, it was revealed that the function only contained one root.

Newton's Method:

The bisection method revealed that P_0= 4.5 was a good guess for part a. The other parameters were unchanged. P_0= 10, 100, 1000, 10000 was selected for the initial guess in part B in order to test how far the initial guess could be before it broke.

Secant Method:

Similar to Newton's method, P_0= -1 and P_1=1 were used because they were near the approximate root. A similar process was repeated for part B.

Mullers Method:

For mullers method P_0= -1, P_1=0, P_2=1 were used as initial parameters. A similar process was repeated for part B.

Results and Analysis

Since each method produces the same answer for both parts, this analysis will discuss the speed of convergence of each method measured as minimizing the number of iterations taken for the approximate value to be within the bound of tolerance.

The following is the console output for the bisection method depicting the data for each iteration. The same printout was repeated for each method for both parts.

```
==============================================================================
                              Bisection Method
==============================================================================

Iteration p              Absolute Error     Relative Error
1        -0.5000000000    2.3750000000       inf
2        4.2500000000     5.0468750000       1.1176470588
3        6.6250000000     84.5722656250      0.3584905660
4        5.4375000000     23.8103027344      0.2183908046
5        4.8437500000     6.0215759277       0.1225806452
6        4.5468750000     0.2741889954       0.0652920962
7        4.6953125000     2.6734967232       0.0316139767
8        4.6210937500     1.1508311629       0.0160608622
9        4.5839843750     0.4262687191       0.0080954410
10       4.5654296875     0.0730459346       0.0040641711
11       4.5561523438     0.1013176168       0.0020362233
12       4.5607910156     0.0143226621       0.0010170762
13       4.5631103516     0.0293148935       0.0005082796
14       4.5619506836     0.0074844347       0.0002542044
15       4.5613708496     0.0034220334       0.0001271184
16       4.5616607666     0.0020304707       0.0000635551
17       4.5615158081     0.0006959638       0.0000317786
18       4.5615882874     0.0006672078       0.0000158890
```

This data allows us to see the values as the sequence converges to a point within the bounds. Although this information is useful to get an understanding of what each method looks like, more useful insight is found from the following figures.

Figure 1: Convergence comparison for part A

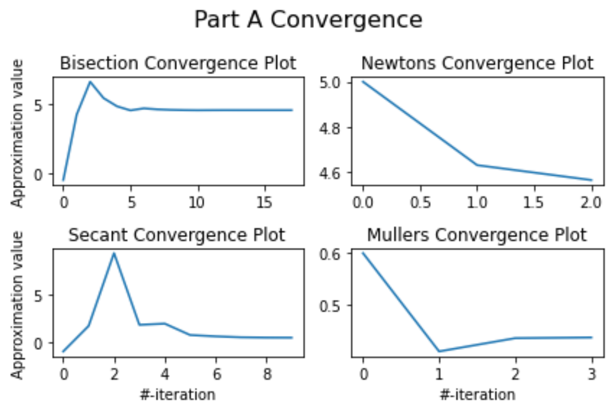
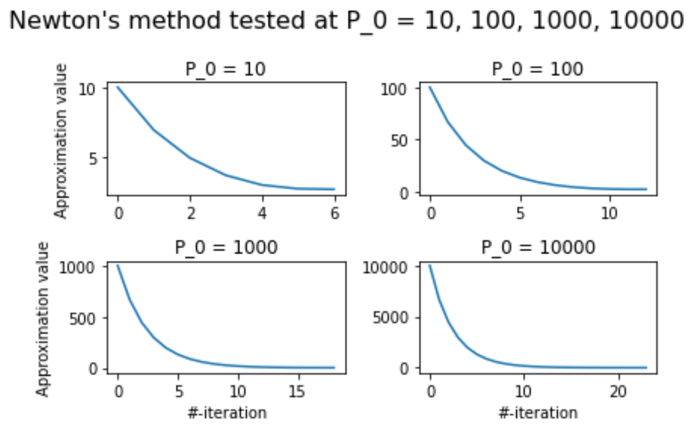


Figure 1 is a plot of the approximation at each iteration of the algorithm for each of the respective methods. This visualization grants easy comparison between each of the methods for identifying rates of convergence. The two subplots above for the Bisection and Newtons method depict the convergence around the root located at approximately 4.561. The subplots below marked Secant and Mullers convergence plot depict the second root for our function in part a found at approximately 0.438. The important thing to note here is the difference in the number of iterations taken for the value to stabilize and reach the tolerance. For the bisection method, it took more than 15 iterations whereas for Newton's method it only took 3. This makes sense since we expect Newton's method to converge very quickly for guesses close to the target value. And since the bisection method was used to get a good approximation first, we see the utility for Newtons. Similarly, for the secant method an Mullers method. Ultimately, the other methods converge more quickly, but it helps to already have good approximations of the values.

Figure 2: Effect of Further Initial Guess on Newton's Method for Part B



A similar process was used to answer part B and as a result, the data looked very similar to that of figure 1. The main difference is that there is only one root for the function in part b at approximately 2.690652076.

Since that analysis revealed similar results, a test of the claim of convergence for Newton's method was conducted. Here the initial guess P_0 was modified to see the effect on convergence. Surprisingly, all the test converged and the only thing that changed was the number of iterations. This is unlikely to hold generally, since Newton's method requires that the f'(p_i) is not equal to 0. It is surprising that such large initial guesses were still able to converge given this constraint. It is likely that this is due to the nature of the derivative of the cubic function which is itself quadratic and therefore only 0 at one point along the domain. If we were to test with a function that oscillates, the behavior is unlikely to repeat.

Method Definitions

```
def f1(x):
    return  (x**3) - (5*(x**2)) + (2*x)
def df1dx(x):
    return ((3*(x**2))) - (10*x) + 2
def f2(x):
    return (x**3) - (2*(x**2)) - 5
def df2dx(x):
    return ((3*(x**2))) - (4*x)

def bisectionMethod(function,left, right,tolerance, maxIter):
    i = 0
    function_left = function(left)
    iterationData = {}
    previous = None # initizalize previous
    # Print table header
    print(f"{'Iteration':<10} {'p':<20} {'Absolute Error':<20} {'Relative Error':<20}"
    while(i < maxIter)
        #calculates p
        current = left + float(right - left) / 2
        function_current = function(current)
        # Calculate absolute and relative errors
        absolute_error = abs(function_current - 0)
        if previous is not None:
            relative_error = abs((current - previous) / current) if current != 0 else 0
        else:
            relative_error = float('inf')  # No previous in the first iteration
        # Print iteration details in a nice table format
        print(f"{i +
1:<10} {current:<20.10f} {absolute_error:<20.10f} {relative_error:<20.10f}")
        iterationData[i] = [current,absolute_error,relative_error]
        if(function_current == 0 or float(right - left) / 2 < tolerance):
            return current,iterationData
        i += 1
        previous = current
        if(function_left * function_current > 0):
            left = current
        else:
            right = current
    print("Method failed after %d iterations, MAX ITERATIONS = %d\n" %(i,
maxIter))
    return iterationData
```

```
def newtons_method(function, derivative, initial_guess, tolerance, maxIter):
    current = initial_guess
    # Print table header
    print(f"{'Iteration':<10} {'p':<20} {'Function Value':<20} {'Absolute
Error':<20} {'Relative Error':<20}")
    for i in range(maxIter):
        function_value = function(current)
        derivative_value = derivative(current)
        # Check if the derivative is zero to avoid division by zero
      if derivative_value == 0:
            raise ValueError("The derivative is zero. No solution found.")
        # Update the current guess using Newton's formula
        next_guess = current - function_value / derivative_value
        # Calculate absolute and relative errors
        absolute_error = abs(function(next_guess))
        relative_error = abs((next_guess - current) / next_guess) if next_guess != 0
else float('inf')
        # Print iteration details in a nice table format
        print(f"{i +
1:<10} {next_guess:<20.10f} {function_value:<20.10f} {absolute_error:<20.10f} {re
lative_error:<20.10f}")
        # Check for convergence
        if absolute_error < tolerance:
            return next_guess
        current = next_guess  # Update current guess for the next iteration
```

```
def mullers_method(function, initial_guess_1, initial_guess_2, initial_guess_3,
tolerance, max_iter):
    x0, x1, x2 = initial_guess_1, initial_guess_2, initial_guess_3
    # Print table header
    print(f"{'Iteration':<10} {'p':<20} {'Function Value':<20} {'Absolute
Error':<20} {'Relative Error':<20}")
    for i in range(max_iter):
        f0, f1, f2 = function(x0), function(x1), function(x2)
        # Calculate the denominator
        denominator = (f1 - f0) * (f2 - f0) * (f2 - f1)
        # Check if denominator is zero
        if denominator == 0:
            raise ValueError("Denominator is zero. No solution found.")

        # Calculate the next guess using Müller's formula
        h1 = f1 - f0
        h2 = f2 - f0
        d = (h2 * h1 * (f1 - f0)) / denominator
        next_guess = x2 - (h2 * h1) / (h2 + h1 + (2 * d))

        # Calculate absolute and relative errors
        absolute_error = abs(function(next_guess))
        relative_error = abs((next_guess - x2) / next_guess) if next_guess != 0 else
float('inf')

        # Print iteration details in a nice table format
        print(f"{i +
1:<10} {next_guess:<20.10f} {function(next_guess):<20.10f} {absolute_error:<20.10f}
{relative_error:<20.10f}")

        # Check for convergence
        if absolute_error < tolerance:
            return next_guess

        # Update guesses for the next iteration
        x0, x1, x2 = x1, x2, next_guess

    # If the maximum number of iterations is reached, return the last guess
    return next_guess
```

```
def secant_method(function, initial_guess_1, initial_guess_2, tolerance, max_iter):
    current = initial_guess_1
    previous = initial_guess_2
    # Print table header
    print(f"{'Iteration':<10} {'p':<20} {'Function Value':<20} {'Absolute
Error':<20} {'Relative Error':<20}")
    for i in range(max_iter):
        function_current = function(current)
        function_previous = function(previous)
        # Check if the function values are too close to avoid division by zero
        if function_current - function_previous == 0:
            raise ValueError("Function values are too close. No solution found.")
        # Update the current guess using the Secant formula
        next_guess = current - function_current * (current - previous) /
(function_current - function_previous)
        # Calculate absolute and relative errors
        absolute_error = abs(function(next_guess))
        relative_error = abs((next_guess - current) / next_guess) if next_guess != 0
else float('inf')
        # Print iteration details in a nice table format
        print(f"{i +
1:<10} {next_guess:<20.10f} {function_current:<20.10f} {absolute_error:<20.10f} {
relative_error:<20.10f}")
        # Check for convergence
        if absolute_error < tolerance:
            return next_guess
        previous, current = current, next_guess  # Update guesses for the next iteration
```