Numerical Methods, project B No. 53

Student: Piotr Skibiński

ID: 303840

Task 1

Find all zeros of the function $f(x) = 3.1 - 3x - e^{-x}$ in the interval [-5, 10].

About solving a nonlinear equation.

Iterative method for finding solutions of, f(x) = 0 (roots) where f is a nonlinear equation are going to be the subject of our tasks. Funding a roots is possible thanks to several algorithms in the task we are going to be discussing two of them the *bisection method* and *the Newton's method* but first of all we do need to understand how the concept works. Unfortunately in most cases it is not enough to just run an algorithm. First of all we need to proceed a *root bracketing* this phase trying to locate a root. Algorithmic implementation is mainly about checking the values on the two ends of an interval in the search of the following case: f(a) * f(b) < 0. If a function f(x) is continuous then at least one root is located in [a, b]. When we obtain our [a, b] we can find the root by using one of the iterative methods. It is good to mention that iterative in this case means that we are approximating our solution x_n and then trying to improve it $x_n \rightarrow_{n \rightarrow inf} \alpha$ where α is a root. The convergence ball of a iterative method is defined as a neighborhood of the solution of the root for given radius δ. We can express it as $\|x_0 - a\| \leq \delta$.

Convergence:

General order of convergence of iterative methods can be defined as the largest number p ≥ 1 such that $\lim_{n \rightarrow inf} \frac{|x_{n+1} - \alpha|}{|x_n - \alpha|^p} = k < inf$ where the k is a convergence factor. *The larger the order of convergence the better the convergence of the method we get.*

The bisection method

After proceeding the aforementioned root location we can our iterative method in this case it is going to be a **bisection method**. Let [a, b] = [a_0, b_0], be an initial interval which do contain the root a of our function. When it comes to bisection method for every n-th step of the iteration we do need to:

1. Divide the current interval $[a_n, b_n]$ into two equal subintervals.
2. Locate the middle of the interval $c_n = \frac{a_n + b_n}{2}$
3. Evaluate the value of the function $f(c_n)$
4. Evaluate $f(a_n)f(c_n)\ and\ f(c_n)f(b_n)$ choose the one with negative value.
5. Create a new subinterval from the chosen values and denote it as $[a_{n+1}, b_{n+!}]$

Following steps are repeated until $f(x_{n+1}) \leq \delta$, where δ is assumed solution accuracy. The following method is linearly convergent p = 1, and has the convergence factor k = $\frac{1}{2}$.

Disadvantages:

When it comes the algorithms there are always some pros and cons. This method may occur when the functions have a flat graph == its derivative has small absolute value around the neighborhood of the root. One of the solutions for this problem could be checking the length of the interval by simply evaluating $[b_0 - a_0]$.

The Newton's method.

The Newton method could also be called the tangent method. The idea here is that we operate on an approximation of the function f(x) by the first order part of its expansion into a Tylor series at a current point x_n where as in the previous example it is our approximation of the root. The method can be easily presented in a iteration formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

If the method is convergent then it usually very efficient with the p = 2.

Disadvantages:

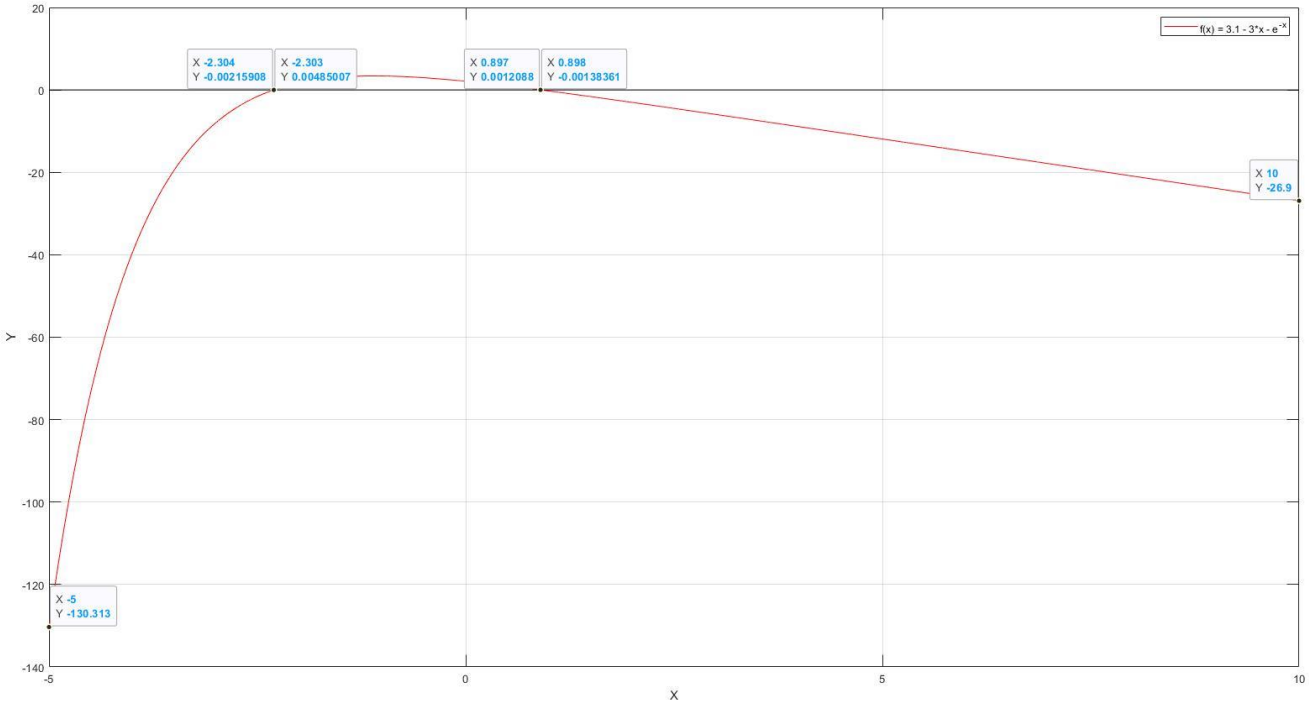
The method is locally convergent so if an initial point is too far from the root then there is a possibility of a divergent furthermore when the derivative f'(α) is close to zero it is not recommended to use it because in this case the algorithm is sensitive to numerical errors.

Results:

First of all I decided to draw the graph of a given function f(x).

```
>> x = TASK1(0.001);

>> plotTask1(x);
```



Little remark about my approach here:

In my code I am using my own method to find potential roots. First of all I generate the table of solutions with a given 'jump' simply doing **range = initial:obj.jump:final**; in my case initial value is = -5, final value = 10 and **jump is given in a constructor**. Method **findZerosA** then finds the cross of a x-axis (it is described in the code) and executes both algorithms. Bisection method with a **range(i-1)** and **range(i)** and Newtons with a **range(i-1)** + **range(i)/2** as on initial point where **I** is a iteration of a loop which have found a cross with the x-axis. In my opinion aforementioned solution will gave a fair comparison between those two methods.

Coming back to the results:

First of all lets generate the results, I will use jump = 1.
>> x = TASK1(1);
>> findZerosA(x);

This method generated the results:

For the root ID: 1, BisectionMethod initial interval: [-3.000000, -2.000000], and Newtons initial guess: -2.500000, we get:
Solution, Bisection: -2.30369210, number of interations: 21
Solution, Newtons: -2.30369211, number of interations: 4

For the root ID: 2, BisectionMethod initial interval: [0.000000, 1.000000], and Newtons initial guess: 0.500000, we get:
Solution, Bisection: 0.89746666, number of interations: 18
Solution, Newtons: 0.89746630, number of interations: 4

Where the build in accuracy is `1e-6;(variable x.accuracy)`

Conclusion:

Function $f(x) = 3.1 - 3x - e^{-x}$ in the interval [-5, 10] have two roots. Both methods generated correct results but Newtons method managed to obtain them significantly faster. All of this confirms the theory. As we know from the description part Newton's methods can be divergent but if it is convergent it is a quadratic convergence (p = 2) on the other hand Bisection method is only linearly convergent (p = 1).

Task2

Find all (real and complex) roots of the polynomial

$$f(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$
$$[a_4, a_3, a_2, a_1] = [-1, -7, 7, 3, 9]$$

Theory:
The above polynomial can be expressed in the following form:

$$f(x) = a_nx^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0$$

Polynomial in this form have exactly n roots. A important thing here is that a polynomial is a continues and differentiable nonlinear function. Thanks to that we can use methods from task1 to find there roots. So why would we need to use something different? The large disadvantage of the previous methods is that we are not able to find complex root with the use of them. In the aforementioned task we are asked to do so that's why we are going to use more advanced approach.

Muller's method:

The idea standing behind Muller's method is to approximate the polynomial locally in a neighborhood of a root with the use of quadratic function. The method can be developed using quadratic interpolation based on three different points (**Muller's method version MM1)**. An efficient approach using only one point is also available in this case we need to base on a value of the first and second derivative on the polynomial at the current point (**Muller's method version MM2)**. Both of them are going to be described below.

MM1

When it comes to MM1 method we need to consider three points $x_0,\ x_1,\ x_2$. And their corresponding values $f(x_0),\ f(x_1),\ f(x_2)$. Using those points we create a parabola that passes through those points, then the roots of the parabola are found and one of the them is being selected for an approximation of the solution.

This is how parabolas are created:

$$f(x_0) = a(x_0 - x_2)^2 - b(x_0 - x_2) + c$$
$$f(x_1) = a(x_1 - x_2)^2 - b(x_1 - x_2) + c$$
$$f(x_2) = a(x_2 - x_2)^2 - b(x_2 - x_2) + c \ \rightarrow\ c$$

After funding the above equations we obtain the following:

$$b = \frac{(x_0 - x_2)^2[f(x_1) - f(x_2)] - (x_1 - x_2)^2[f(x_0) - f(x_2)]}{(x_0 - x_2)(x_1 - x_2)(x_0 - x_1)}$$

$$a = \frac{(x_1 - x_2)\ [f(x_0) - f(x_1)] - (x_0 - x_2)\ [f(x_1) - f(x_2)]}{(x_0 - x_2)(x_1 - x_2)(x_0 - x_1)}$$

$$c = f(x_2)$$

Now we need to compute those roots using following formulas:

$$out1 = \frac{-2c}{b - \sqrt{b^2 - 4ac}}, \qquad out2 = \frac{-2c}{b + \sqrt{b^2 - 4ac}}$$

Finally we can get the x_3 (next approximation)

$$x_3 = x_2 + out_{min}$$

Where:

$$out_{min} = out1, \qquad if\ \left|b + \sqrt{b^2 - 4ac}\right| \geq \left|b - \sqrt{b^2 - 4ac}\right|$$
$$out_{min} = out2, \qquad \text{in the opposit case}$$

MM2

The another Muller method is using values of polynomial it's first and second derivative. On the numerical point of view, this method should be slightly more efficient mainly because of a fact that calculating polynomial at three different points is more expensive than calculating values of a polynomial it's first and second derivative.

An implementation of the method is relatively easy we simple need to calculate:

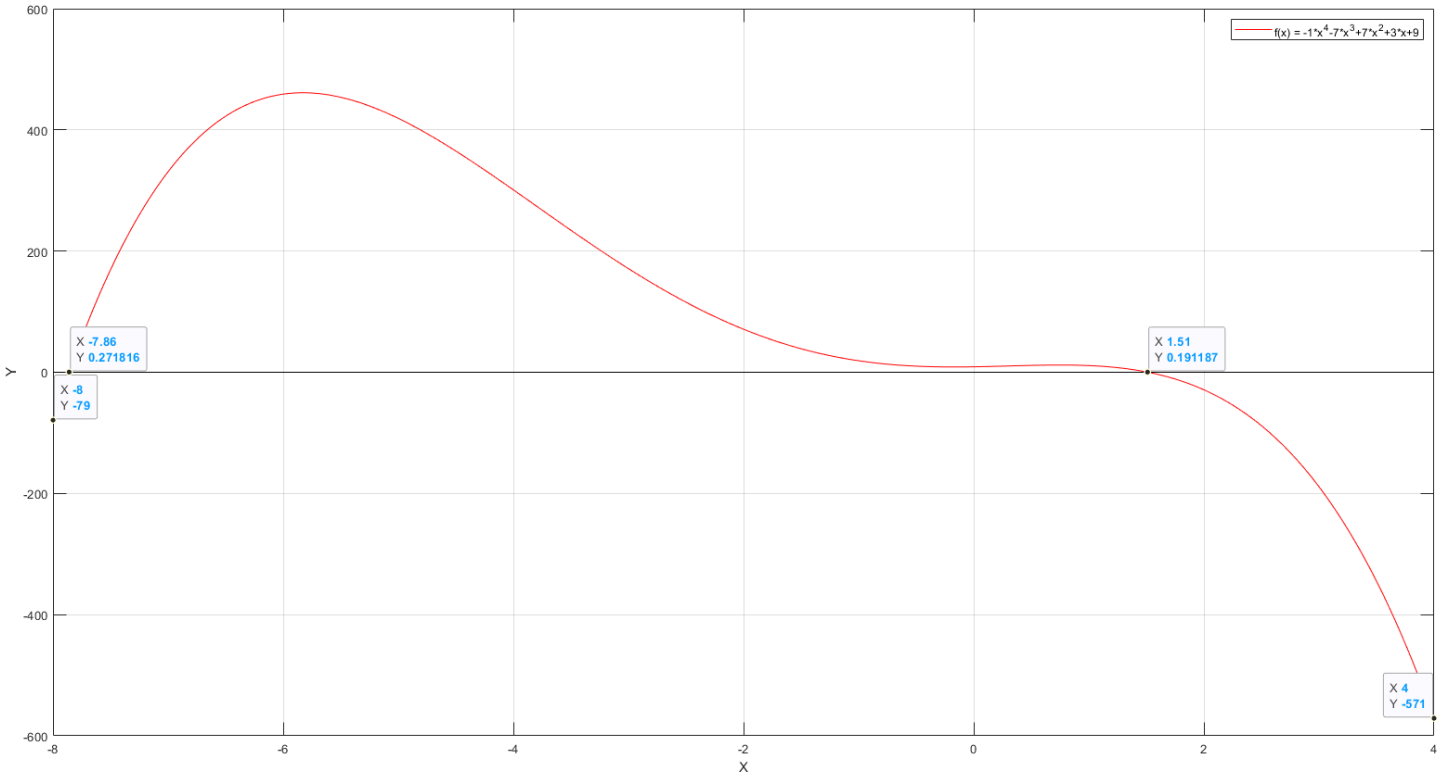
1. Evaluate a, b and c:
$$f(0) = c = f(x_k)$$
$$f'(0) = b = f'(x_k)$$
$$f"(0) = 2a = f"(x_k)$$
2. Calculate the roots:
$$out1 = \frac{-2c}{b - \sqrt{b^2 - 4ac}}, \qquad out2 = \frac{-2c}{b + \sqrt{b^2 - 4ac}}$$
3. And finally evaluate x_{k+1}
$$x_{k+1} = x_k + out_{min}$$

Results:

First of all I decided to draw the graph of a given function f(x).

```
>> x = TASK2;

>> plotTask2(x);
```



Above plot in in a range of <-8, 4>

Little remark 2:

Muller's method MM1 calculates root from the three initial ones, MM2 method does that with the use of only one argument, but there is still a small problem to proceed. In the task we are asked to calculate all roots of the polynomial that's why it is worth mentioning on how did I managed to do that. The method is simple. After the execution of MM1 and MM2 I am reconstructing the actual polynomial in the following way:

$$f_{new}(x) = \frac{f(x)}{(x - x_0)}$$

Where x_0 is a previously found root. I iterate through the loop as long as:

$$f(x_0) \; ! = f(x_1) \; ! = f(x_2)$$

In other words as long as the function is not constant.

Output:
After plotting the graph we can execute are methods. To obtain the results.
Initial points are
MM1: {-0.5, 0, 0.5}
MM2: {0}

>> findRootsTask2()

Method outputs the following:

Method MM1:
Iterations: 8, -0.327282-0.805359j
Method MM1:
Iterations: 7, -0.327282+0.805359j
Method MM1:
Iterations: 4, 1.515070
Method MM1:
Iterations: 4, -7.860505
Method MM2:
Iterations: 4, -0.327282+0.805360j
Method MM2:
Iterations: 4, -0.327282-0.805359j
Method MM2:
Iterations: 3, 1.515070
Method MM2:
Iterations: 3, -7.860505

After comparing the MM2 to Newtons method we get the following results:

>> compareRootsTask2()

Method MM2:
Iterations: 3, 1.515070
Newtons method:
Iterations: 5, 1.515070
Method MM2:
Iterations: 3, -7.860505
Newtons method:
Iterations: 3, -7.860505

Conclusions:

As we can see the MM2 algorithm is way more efficient than the MM1 it's order od convergence is 1.84 which pretty close to the Newton's method, furthermore, we are capable of finding all complex roots which is a great advantage. It's worth mentioning that Muller's method can not only be used for finding a roots of polynomials, but also for different nonlinear functions. The funny thing here is that in this case MM2 is faster than Newtons method besides the fact that Newtons method has higher order of convergence.

Task3

Find all (real and complex) roots of the polynomial f(x) from II using the Laguerre's method. Compare the results with the MM2 version of the Müller's method (using the same initial points).

The Laguerre's method:
The method can be defined in a following way:

$$x_{k+1} = x_k - \frac{nf(x_k)}{f'(x_k) \pm \sqrt{(n-1)[(n-1)\left(f'(x_k)\right)^2 - nf(x_k)f''(x_k)]}}$$

Where n is denoted as the order of the polynomial, and the sight should be chosen in the same way as in the Muller's case. It is easy to notice that in general MM2 and Laguerre's formulas are similar the second one has slightly more complex structure because the order of a polynomial is taken into account. The Laguerre's method is one of the best out there so how the numbers are looking?

Output:
After executing the
Initial points are
MM1: {-0.5, 0, 0.5}
Laguerre: {-0.5, 0, 0.5}

>> compareRootsTask3()

Method I managed to obtain following results.

Method Laguerre:
Iterations: 4, -0.327282-0.805359j
Method Laguerre:
Iterations: 4, -0.327282+0.805359j
Method Laguerre:
Iterations: 3, 1.515070
Method Laguerre:
Iterations: 3, -7.860505
Method MM2:
Iterations: 4, -0.327282+0.805360j
Method MM2:
Iterations: 4, -0.327282-0.805359j
Method MM2:
Iterations: 3, 1.515070
Method MM2:
Iterations: 3, -7.860505

Comments:

Unfortunately real world testing did not show any interesting results. Both methods obtain all 4 roots in the same number of operations, furthermore, after changing TASK2/3 accuracy from 1e-6 to 1e-10 I have generated following results:

Method Laguerre:
Iterations: 4, -0.327282-0.805359j
Method Laguerre:
Iterations: 4, -0.327282+0.805359j
Method Laguerre:
Iterations: 4, 1.515070
Method Laguerre:
Iterations: 3, -7.860505
Method MM2:
Iterations: 5, -0.327282+0.805359j
Method MM2:
Iterations: 4, -0.327282-0.805359j
Method MM2:
Iterations: 2, 1.515070
Method MM2:
Iterations: 2, -7.860505

While Laguerre method turned out to be slightly faster for complex roots it completely lost in when it came to real values.

THE END

Sources: lectures and 'Numerical Methods' by Piotr Tatjewski.