Project B No. 16
Final report

Author : Wiktor Łazarski

Index number : 281875

Field of study : Computer Science

Faculty : Electronics and Information Technology, Warsaw University of Technology

Table of content

1.1 Bisection method - Introduction

The first task is to write a program which will find all zeros of the function :

$$f(x) = 2.1 - 2x - e^{-x}$$

in the interval [-5, 10] using :

a) bisection method
b) Newton's method

Both methods are an examples of **iterative methods** of finding the roots of polynomials. Before applying any of that method one must find an interval where zero of that function is located. This phase is called *root bracketing***.** The easiest way to find it is to draw approximate plot of the function and pick them simply by looking where the function crosses with $y = 0$ line. Such picked interval will satisfied the necessary conditions for properly working iterative algorithm, which are :

- `[a, b]` is a closed interval
- Function `f(x)` is continuous in the interval `[a, b]`
- $f(a) \cdot f(b) < 0$

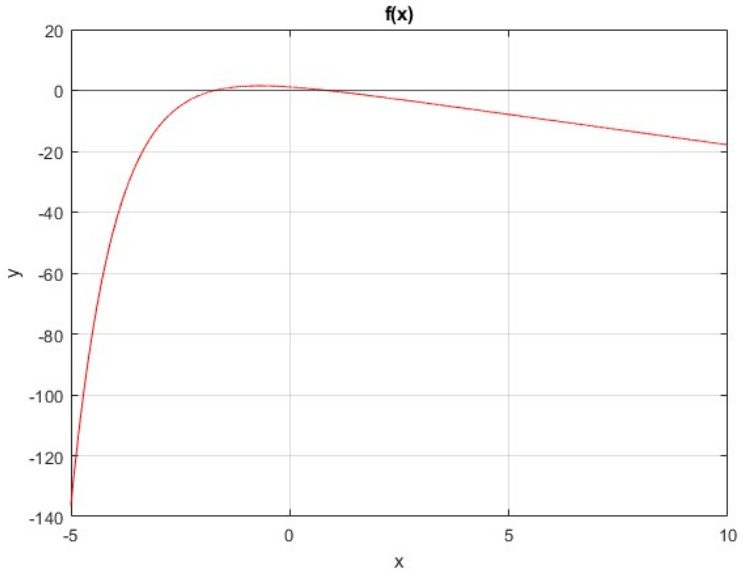
To compare iterative methods we will look at their *order of convergence*. It is defined as the largest number $p \geq 1$ such that :

$$\lim_{n\to\infty} \frac{|x_{n+1} - \alpha|}{|x_n - \alpha|^p} = k < \infty$$

where the coefficient *k* is the *convergence factor*. The largest the order of convergence is the better the convergence of the method is. The quality of iterative method will be valid in terms of order of convergence (*p*).

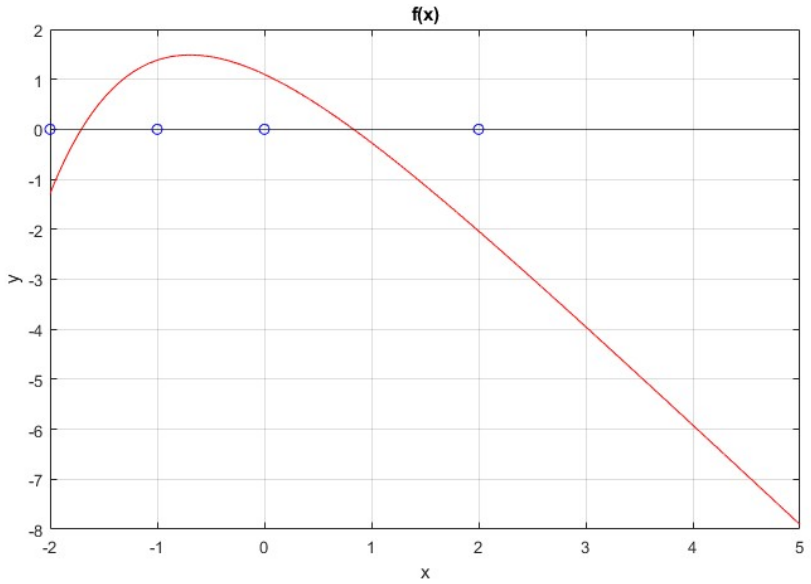
Before implementing any of researching method we must firstly pick intervals in which we will be looking for roots.

Approximate plot of the function *f(x)*



1.1 Bisection method - Introduction

We can observe that the interval, we are given, contains two roots of function *f(x)* but from the plot we draw it is hard to choose scope of intervals where zeros should be looked for. For that matter, I will redraw the function *f(x)* but narrowing down the scope of it to `[-2, 5]` hoping that it will allow me to see crossing (with `y=0` straight line) points.



With this being done it is now trivial to pick those two intervals where we can expect zeros to be. Firstly, I will consider interval `[-2, -1]` and then interval `[0, 2]`.

Now, we can procced with our researches of **bisection method**. Using bisection method we start from initial interval `[a, b] = [a₀, b₀]` isolating the root. In algorithm every iteration we :

1) Divide current interval into two equal subintervals, with the division point c_n located in the middle of the interval

$$c_n = \frac{a_n + b_n}{2},$$

and we compute the value of $f(c_n)$.

2) Next step is to compute $f(a_n) \cdot f(c_n)$ and $f(b_n) \cdot f(c_n)$ and the interval for next iteration is assembled. Algorithm should swap c_n either with a_n or b_n according to which multiplication result in negative number.

The method is repeated as long as, for example $|f(c_n)| \leq \delta$, where δ is assumed accuracy of task. The test may be inaccurate for functions which derivatives is relatively small in area of root. For that matter, we also check the length of interval $b_n - a_n$, demanding that it will be sufficiently small.

Accuracy of the obtain solution from bisection method does depend only from number of performed iterations and does not depend on accuracy of calculating *f(x)* made on the edges

1.1 Bisection method - Introduction

of consecutive isolating intervals. Let $\varepsilon_n = b_n - a_n$ indicate the length of interval in *n-ty* iteration, $\varepsilon_0 = [a, b], n = 0,1,2, \ldots$. Therefore

$$\varepsilon_{n+1} = \frac{1}{2}\,\varepsilon_n$$

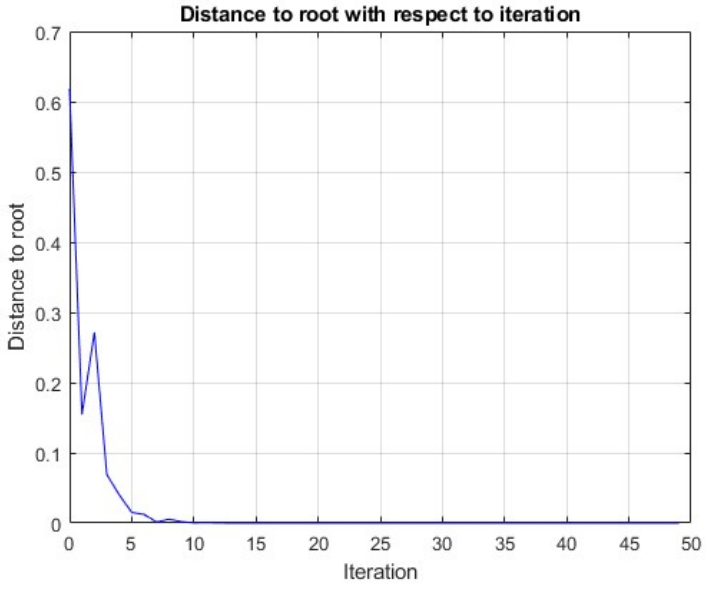
because of that bisection method is convergent linearly ($p = 1$) and its convergence factor $k = 0.5$.

1.2 Bisection method - Results

Before assembling the researching script I set the accuracy of the result to 10^{-20} and the length of interval (in case of "flat" functions) cannot be longer than 10^{-15}. With such set ups I received 50 iterations **for the first interval `[-2, -1]`.** The first ten of them presents as follow :

Iteration	x	f(x)
1	-1.5000	0.6184
2	-1.7500	-0.1547
3	-1.6250	0.2716
4	-1.6875	0.0691
5	-1.7188	-0.0401
6	-1.7032	0.0152
7	-1.7109	-0.0123
8	-1.7071	0.0015
9	-1.7089	-0.0054
10	-1.7081	-0.0012

Because of the difficulty to analyse the table I plot the function presenting iteration with respect to the distance of current c_n to root.



1.2 Bisection method - Results

Conclusions : From the plot we can easily read that after 20 iteration of bisection method we are almost at zero of the function *f(x)*. The final answer will be picked for 50th iteration anyway because it is our the most precise solution.

The first root of the function *f(x)* in interval [-2, -1]
x = -1.70745552317304

But we must remember that it is our approximate zero the real value equals :

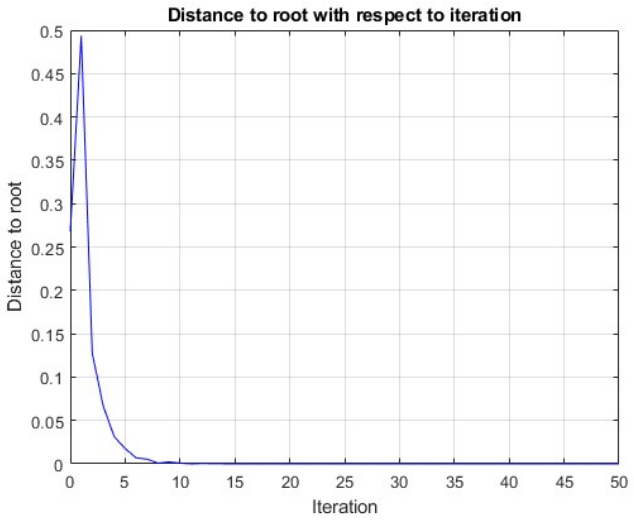
$$f(-1.70745552317304) = -1.77635683940025 \cdot 10^{-15}$$

The procedure was repeated **for the interval `[0, 2].`** The following results were obtained :

Number of iteration equals 51 and the first ten of them presents as follow :

Iteration	x	f(x)
1	1	-0.2679
2	0.5000	0.4935
3	0.7500	0.1277
4	0.8750	-0.0669
5	0.8125	0.0313
6	0.8438	-0.0176
7	0.8282	0.0069
8	0.8359	-0.0054
9	0.8321	0.0008
10	0.8339	-0.0023

and the plot :



1.2 Bisection method - Results

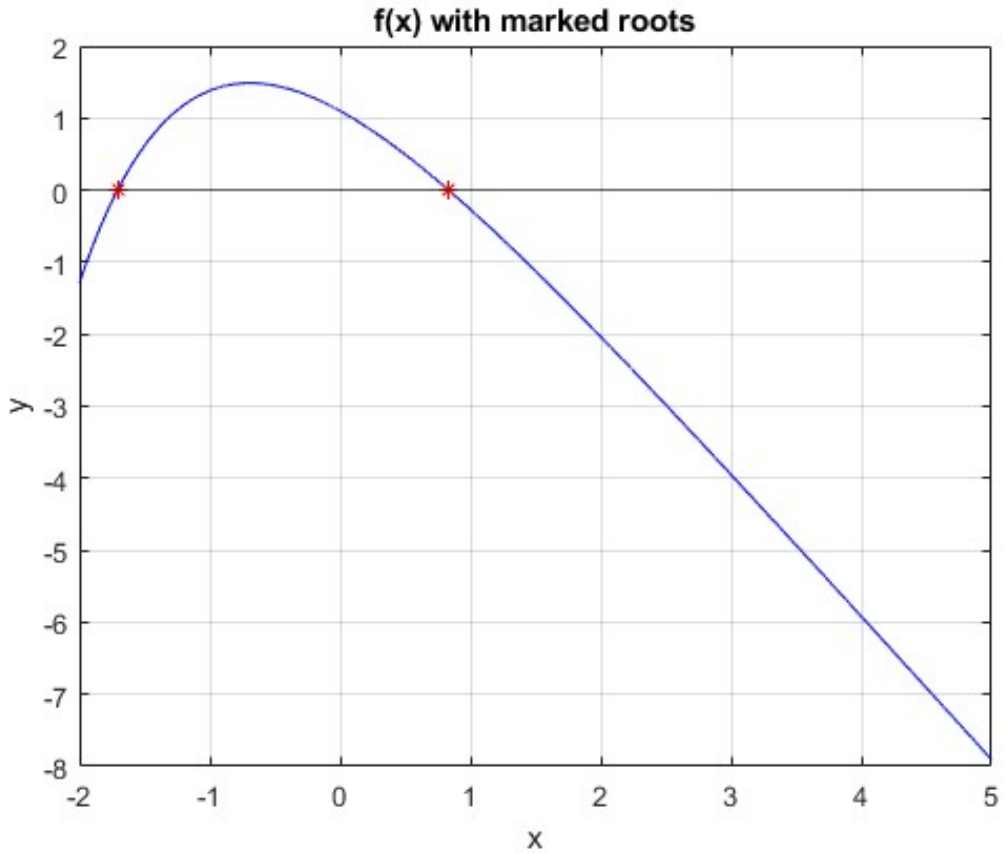
The second root of the function *f(x)* in interval [0, 2]
x = 0.832525223157208

But the real value actually equals :

$$f(0.832525223157208) = -9.99200722162641 \cdot 10^{-16}$$

Final results present as follow :

First root	x = -1.70745552317304
Second root	x = 0.832525223157208



Remark : Points were marked with a real value they give for function *f(x)*. We can see that marked points really indicate the roots of a given function *f(x)* what confirms the correctness of our implementation.

The implementation of bisection method can be found in *Appendices*.

1.3 Newton's method - Introduction

Having researches on bisection method we can move forward to subpoint b of Task 1 and do some researches on **Newton's method**. Newton's method, also called *the tangent method*, operates by approximation of the function *f(x)* by the first order part of its expansion into a Taylor series at a current point x_n (current approximation of a root).

$$f(x) \approx f(x_n) + f'(x_n)(x - x_n)$$

The next point, x_{n+1}, results as a root of the obtained linear function :

$$f(x_n) + f'(x_n)(x_{n+1} - x_n) = 0$$

which leads to the iteration formula

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

The Newton's method is *locally convergent* if an initial point is too far from the root(outside its area of attraction) than a divergence may occur. However, if the Newton's method is convergent, then it is usually very fast, as the convergence is quadratic ($p = 2$).

The Newton's method is particularly effective when the function derivative at the root is sufficiently far from zero (the slope of the curve of *f(x)* is steep). On the other hand, it is not recommended when the derivative $f'(\propto)$ is close to zero, as then it is very sensitive to numerical errors when close to the root.

1.4 Newton's method - Results

Before assembling the researching script I set the accuracy of the result to 10^{-16} because during implementing and acquiring results it occurred that it was impossible to get results with better accuracy. The last thing that I had to do before running Newton's method was to pick initial points. The simplest way to do it is to pick xs from the intervals we setted up previously. For that matter, I first run the method if $x_n = -2$ and then with $x_n = 0$.

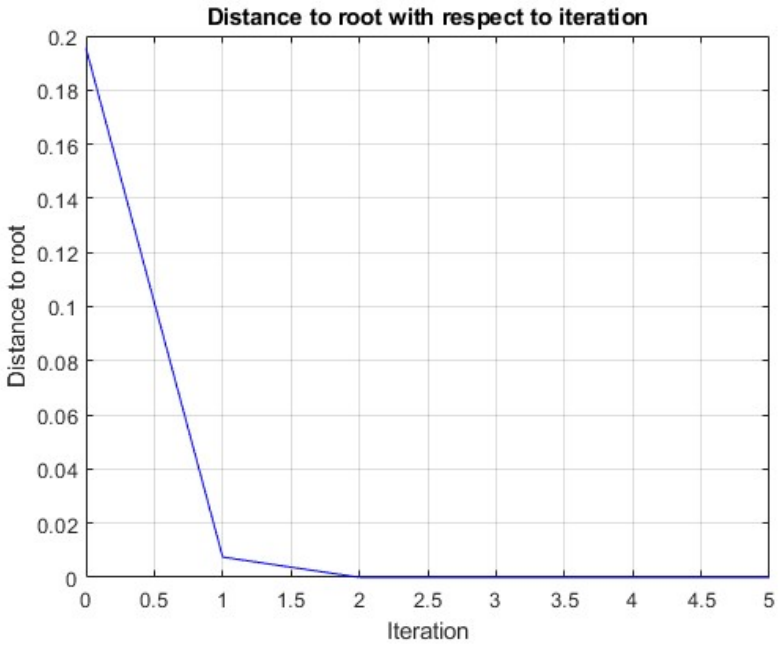
For x_n = -2, number of iteration of algorithms received is 6 and the results of each iteration presents as follow :

Iteration	x	y
1	-1.7609	-0.1955
2	-1.7096	-0.0076
3	-1.7075	-1.2491e-05
4	-1.7075	-3.4822e-11
5	-1.7075	8.8818e-16
6	-1.7074	0

Remark : It may not be visible, because of rounding error, but after next iteration xs are getting smaller and smaller.

1.4 Newton's method - Results

Using the same approach as when we researched bisection method we can draw the plot showing the distance to root with respect to iteration number.



Conclusion : It can be seen that after 3rd iteration we are almost at zero. However, the most precise (with our assembled accuracy) will be the result of x obtained in the 6th iteration. Hence, the first approximate root of our function, obtain using Newton's method is

The first root of the function *f(x)*, starting point -2
x = -1.70745552317304

But the real value actually equals :

$$f(-1.70745552317304) = 0$$

Remark : In that case approximate root is equal to the real value but it may not be always an issue.

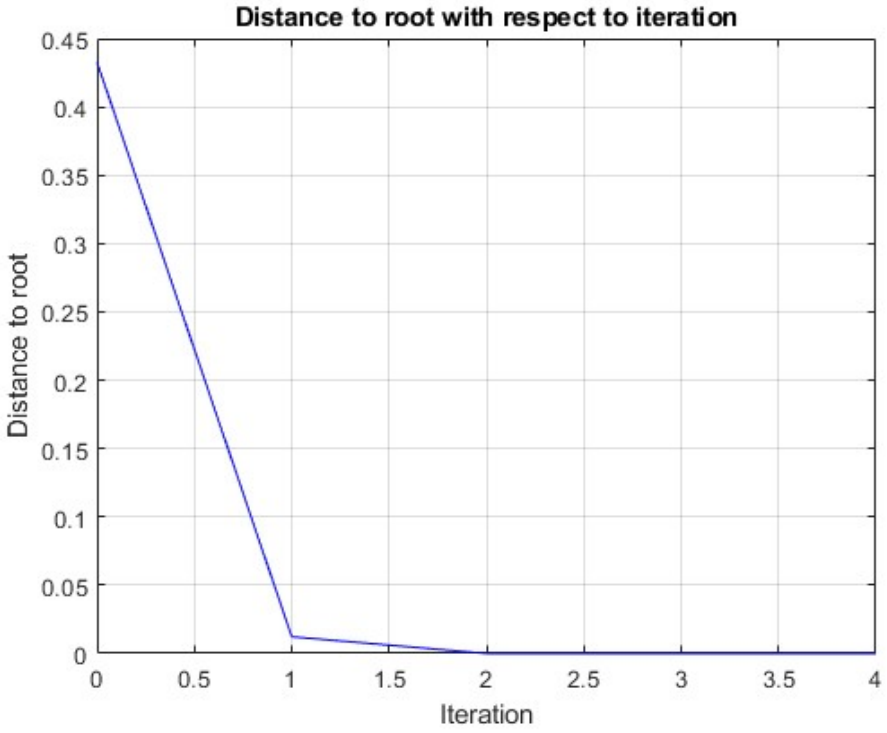
The procedure was repeated **for starting point $x_n = 0$.** The following results were obtained.

Number of iterations equals 5 and they presents as follow :

Iteration	x	y
1	1.1000	-0.4329
2	0.8404	-0.0123
3	0.8326	-1.3216e-05
4	0.8326	-1.5507e-11
5	0.8326	5.5512e-17

1.4 Newton's method - Results

and the plot



The second root of the function *f(x)*, starting point 0
x = 0.832525223157207

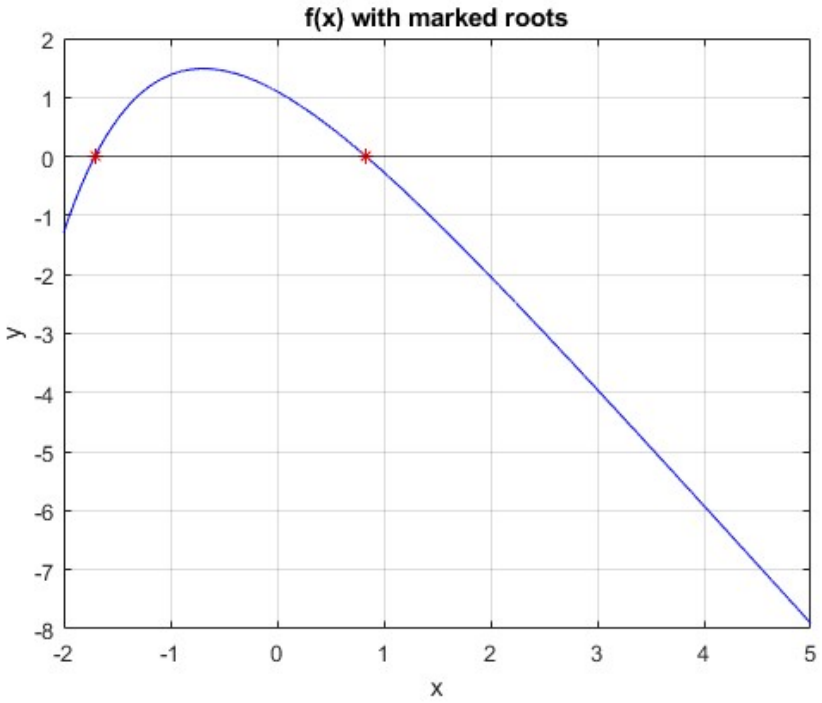
But the real value actually equals :

$$f(0.832525223157207) = 5.55111512312578 \cdot 10^{-1}$$

Final results present as follow :

First root	x = -1.70745552317304
Second root	x = 0.832525223157207

1.4 Newton's method - Results



Remark : Points were marked with a real value they give for function *f(x)*. We can see that marked points really indicate the roots of a given function *f(x)* what confirms the correctness of our implementation.

The implementation of Newton's method can be found in *Appendices*.

1.5 Bisection and Newton's method - Comparison

All acquired results aggregated in the table

Property	bisection method	Newton's method
Number of iterations	50	6
First approximate root	-1.70745552317304	-1.70745552317304
Real value in approximate first root	$-1.77635683940025 \cdot 10^{-15}$	0
Second approximate root	0.832525223157208	0.832525223157207
Real value in approximate second root	$-9.99200722162641 \cdot 10^{-16}$	$5.55111512312578 \cdot 10^{-17}$

1.5 Bisection and Newton's method - Comparison

Summary : First of all, it is worth to mention that number of iterations using Newton's method is significantly smaller. It confirms us about different *order of convergence*. For bisection method it is linear whereas for Newton's method it is quadratic. To check whether it is true we can simply compute the logarithm with a base 2 of number of iterations using bisection method and round it up. It will give us the number of iteration using Newton's method.

$$ln_2(n_b) = ln_2(50) = 5.6439 \approx 6$$

,where n_b is a number of iterations in bisection method. As expected Newton's method is *p* times faster than bisection method.

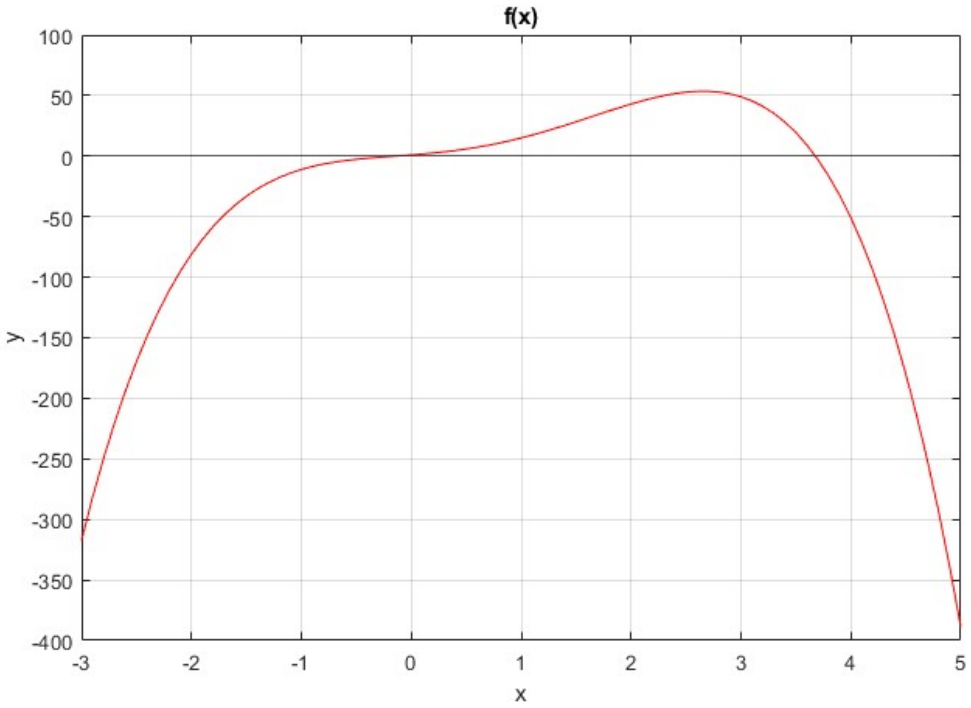
Precision of the results is random. We cannot indicate more precise method because it depends from the function we are researching. In our case, Newton's method returned better approximation for the first root but worse for the second one. Looking at the received value of the first root we can conclude that they are exactly the same. How is it possible that they real value of *f(x)* are different? It is due to the result presenting in the Matlab environment if somehow we could increase that precision we would see that they differ in next positions which are currently invisible.

2.1 Introduction

The second task is to write a program which will find all zeros (real and complex) of the polynomial :

$$f(x) = -2x^4 + 6x^3 + 3x^2 + 7x + 1$$

using Müller's method implementing MM1 and MM2 versions.



Remark : All roots will be looked for in the interval shown in the plot, namely [-3, 5].

General idea of Müller's method is focused on approximation of polynomial in area of roots of quadratic function. The method can be developed using a quadratic interpolation based on three different points. Therefore, it can be treated as a generalization of the secant method, where a linear interpolation based on two points was applied. However, an efficient realization basing on an information from one point only is also available, i.e., basing on values of the polynomial and its first and second derivative at a current point.

MM1. Let's consider three points x_0, x_1, x_2, together with corresponding polynomial values $f(x_0)$, $f(x_1)$ and $f(x_2)$. A quadratic function is constructed that passes through these points, then the roots of the parabola are found and one of the roots is selected for the next approximation of the solution.

Without loss of generality, assume that x_2 is an actual approximation of the solution (the root of the polynomial). Introduce a new, incremental variable

$$z = x - x_2$$

and use the difference

$$z_0 = x_0 - x_2$$

$$z_1 = x_1 - x_2$$

2.1 Introduction

The interpolating parabola defined in the variable *z* is considered,

$$y(z) = az^2 + bz + c$$

Considering the three given points, we have

$$az_0{}^2 + bz_0 + c = y(z_0) = f(x_0)$$
$$az_1{}^2 + bz_1 + c = y(z_1) = f(x_1)$$
$$c = y(0) = f(x_2)$$

Therefore, the following system of 2 equations must be solved to find *a* and *b* :

$$az_0{}^2 + bz_0 = f(x_0) - f(x_2)$$
$$az_1{}^2 + bz_1 = f(x_1) - f(x_2)$$

Because we are interested in root of parabola with the smallest module (the one closest to x_2), hence for finding a root it is highly recommended to use formulas :

$$z_+ = \frac{-2c}{b + \sqrt{b^2 - 4ac}}$$
$$z_- = \frac{-2c}{b - \sqrt{b^2 - 4ac}}$$

For next iteration of approximation we pick root which is the closest to x_2 :

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

,where :

$z_{min} = z_+$, gdy $|b + \sqrt{b^2 - 4ac}| \geq |b - \sqrt{b^2 - 4ac}|$

$z_{min} = z_-$, otherwise

For the next iteration the new point x_3 is taken, together with those two from point selected from x_0, x_1, x_2 which are closer to x_3.

It should be noted that the algorithm should work properly also in cases when $\sqrt{b^2 - 4ac} < 0$ occurs, as this is a standard situation, leading to iterations towards a complex root. Therefore, algorithms of Müller's method should be implemented in a complex number arithmetic. The formula given above can be directly used also in this arithmetic.

MM2. Another version of the Müller's method, *using values of a polynomial and of its first and second order derivatives at one point only* is often recommended. It is slightly more effective numerically, because it is numerically slightly more expensive to calculate values of a polynomial at three different points than to calculate values of a polynomial and of its first and second derivatives at one point only.

2.1 Introduction

It follows directly from the definition of the parabola used in MM1 version, for $z \stackrel{\text{def}}{=} x - x_k$, that at the point $z = 0$:

$$y(0) = c = f(x_k)$$

$$y'(0) = b = f'(x_k)$$

$$y''(0) = 2a = f''(x_k)$$

which leads to the formula for the roots :

$$z_{+,-} = \frac{-2f(x_k)}{f'(x_k) \pm \sqrt{(f'(x_k))^2 - 2\,f(x_k)\,f''(x_k)}}$$

To approximate the solution α, a root with smaller absolute value is chosen :

$$x_{k+1} = x_k + z_{min}$$

where z_{min} is selected from $\{z_+, z_-\}$ in identical way as it was done in the MM1 version.

The MM2 version of the Müller's method should be implemented in a complex number arithmetic, for the same reason as it was for the MM1 version.

2.2 MM1 – Results

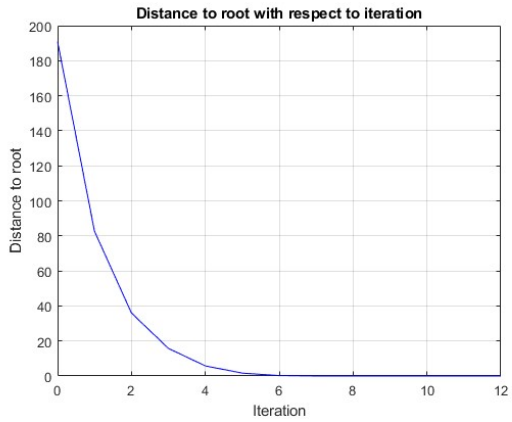
Remark : Method was called for every integer x in the given interval $x = -3, \dots, 5$ and the points were picked in the following way :

$$x_0 = x - 2, \qquad x_1 = x - 1, \qquad x_2 = x$$

Every table presents only first 7 iterations however the plot on the right hand site presents (distance to the root) for more of them.

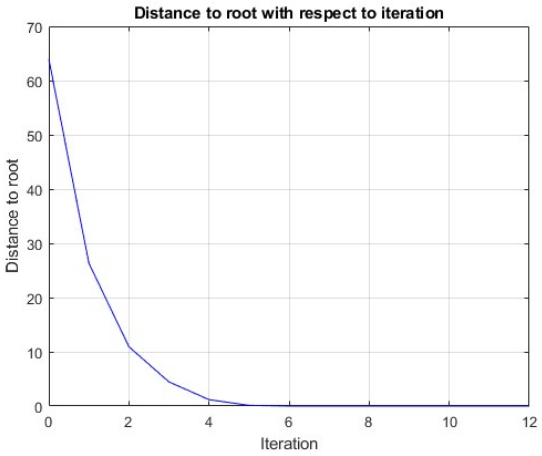
For x = -3

x	y
-2.43 + 0.94i	-57.64 + 181.99i
-1.76 + 1.10i	23.59 + 79.21i
-1.20 + 1.18i	28.90 + 21.39i
-0.75 + 1.21i	15.65 - 1.83i
-0.43 + 1.13i	3.83 - 4.32i
-0.25 + 1.01i	-0.38 - 1.59i
-0.24 + 0.91i	-0.21 + 0.11i



For x = -2

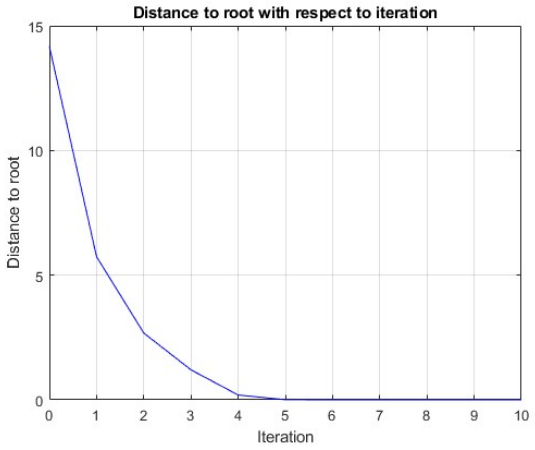
x	y
-1.76 + 0.66i	-25.26 + 58.76i
-1.23 + 0.81i	4.92 + 25.88i
-0.82 + 0.89i	7.60 + 7.85i
-0.52 + 0.94i	4.38 + 0.79i
-0.33 + 0.94i	1.11 - 0.37i
-0.26 + 0.92i	0.07 - 0.10i
-0.26 + 0.91i	-0.00 - 0.01i



2.2 MM1 – Results

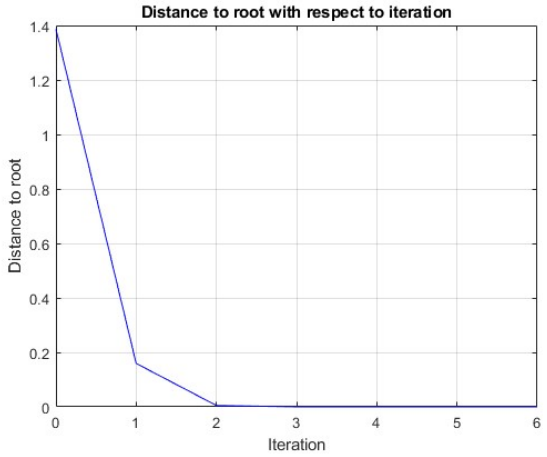
For x = -1

x	y
-1.07 + 0.35i	-9.48 + 10.53i
-0.68 + 0.52i	-0.80 + 5.68i
-0.39 + 0.65i	0.46 + 2.63i
-0.25 + 0.81i	0.02 + 1.20i
-0.25 + 0.91i	-0.17 + 0.08i
-0.26 + 0.91i	0.00 - 0.01i
-0.26 + 0.91i	-4.40e-06 + 3.68e-06i



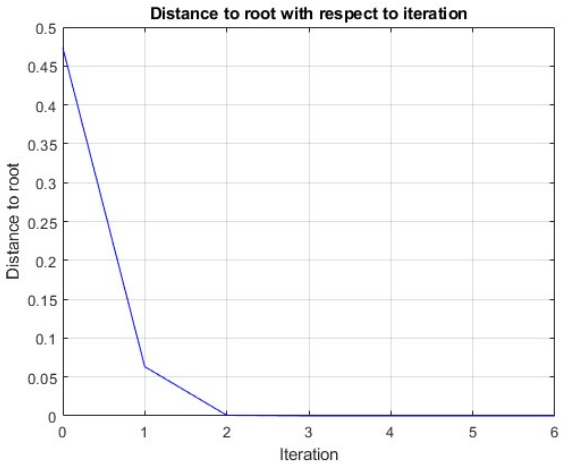
For x = 0 :

x	y
0.05	1.38
-0.12	0.15
-0.15	-0.01
-0.14	1.90e-05
-0.14	3.85e-10
-0.14	0
-0.14	0



For x = 1 :

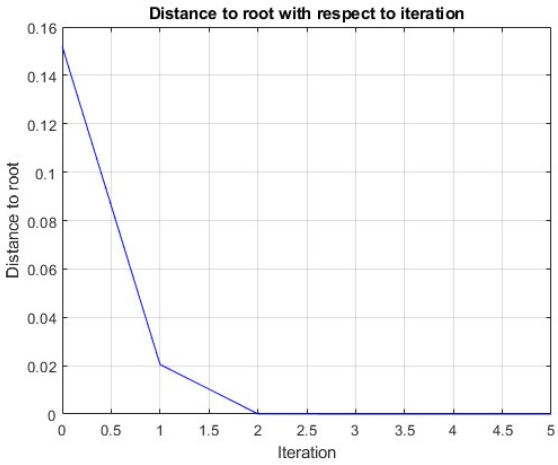
x	y
-0.07	0.47
-0.13	0.06
-0.15	-0.01
-0.14	5.34e-07
-0.14	6.18e-13
-0.14	0
-0.14	0



2.2 MM1 – Results

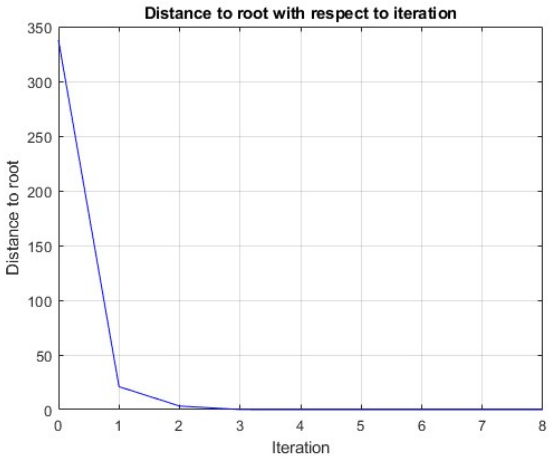
For x = 2 :

x	y
-0.17	-0.15
-0.14	0.02
-0.14	7.54e-05
-0.14	6.10e-09
-0.14	0
-0.14	0



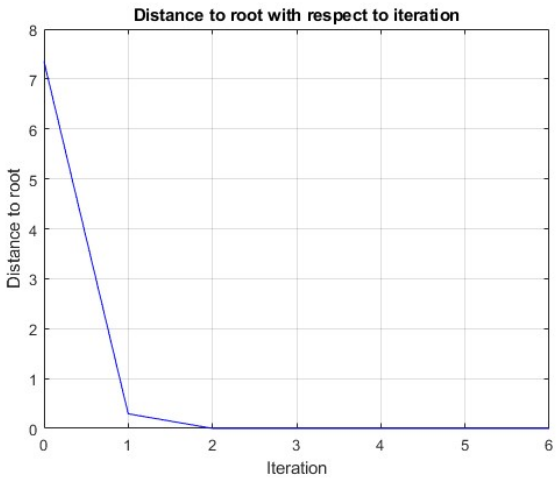
For x = 3 :

x	y
4.89	-337.61
3.48	20.98
3.65	3.26
3.66	-0.07
3.67	7.35e-05
3.67	2.26e-10
3.6	-5.32e-14



For x = 4 :

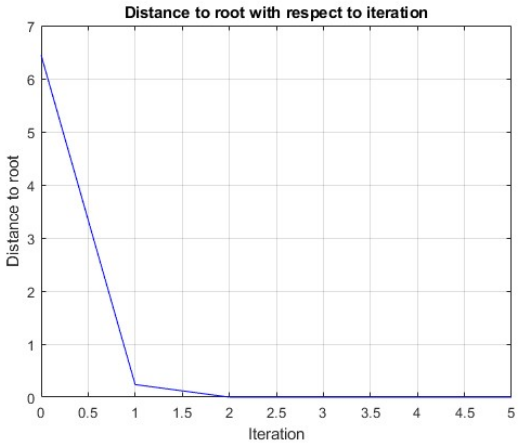
x	y
3.61	7.35
3.67	0.29
3.67	0.01
3.67	-2.97e-08
3.67	3.19e-14
3.67	-1.42e-14
3.67	-1.42e-14



For x = 5 :

2.2 MM1 – Results

x	y
3.72	-6.44
3.67	-0.23
3.67	0.0007
3.67	-1.29e-08
3.67	-1.42e-14
3.67	-1.42e-14



Conclusions : Looking at every plot we can see that method converge no matter which point, from the interval, we will choose. The reason in our changing the initial point is because we want to compute every root of our polynomial. Therefore the following results are obtained : (due to the basic principle of algebra if we find one complex root than its conjugation is also a root of that polynomial) :

Roots found using MM1 version of the Müller's method				
Obtain Root	`-0.26 - 0.91i`	`-0.26 + 0.91`	`-0.14`	`3.67`
Real value of *f(x)*	`-0.00 - 0.01i`	`-0.00 - 0.01i`	`0`	`-1.42e-14`

Test : Obtain values are consistent with build-in Matlab's function ***roots***, which gives the same results as my function what confirms the correctness of my algorithms.

The implementation of MM1 method can be found in *Appendices*.

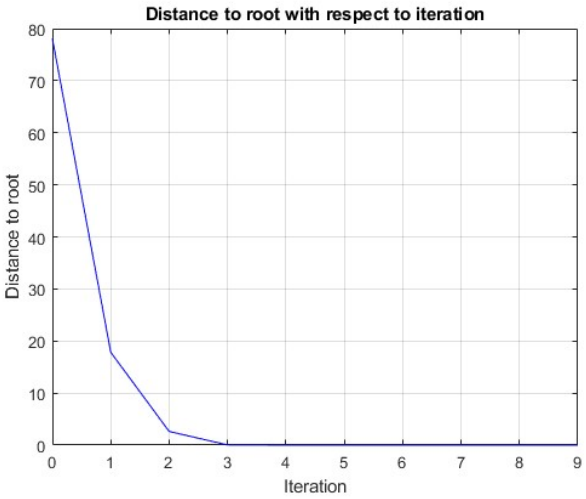
2.3 MM2 – Results

Remark : Method was called for every integer x in the given interval $x = -3, \dots, 5$.

Below we can see the tables and plots for every first 10 iterations :

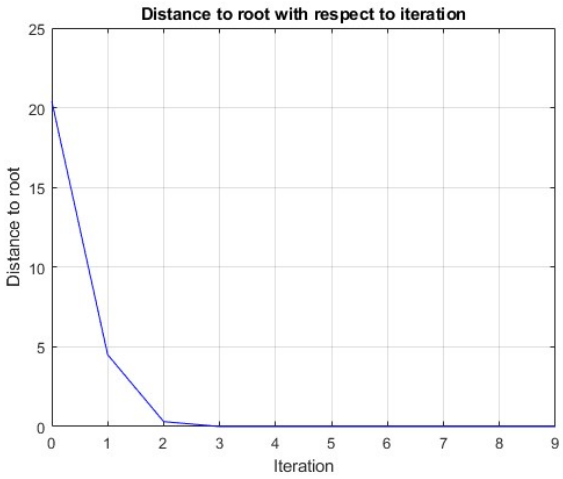
For x = -3 :

x	y
-1.84 - 0.81i	-16.47 - 76.33i
-0.92 - 1.07i	15.84 - 8.16i
-0.36 - 1.03i	1.74+ 1.96i
-0.26 - 0.91i	-0.044 - 0.01i
-0.26 - 0.91i	-2.73e-07 - 2.48e-07i
-0.26 - 0.91i	8.88e-16 - 1.77e-15i
-0.26 - 0.91i	8.88e-16 + 0.00i
-0.26 - 0.91i	0.00 + 8.88e-16i
-0.26 - 0.91i	8.88e-16 - 8.88e-16i
-0.26 - 0.91i	-6.66e-16 + 0.00i



For x = -2 :

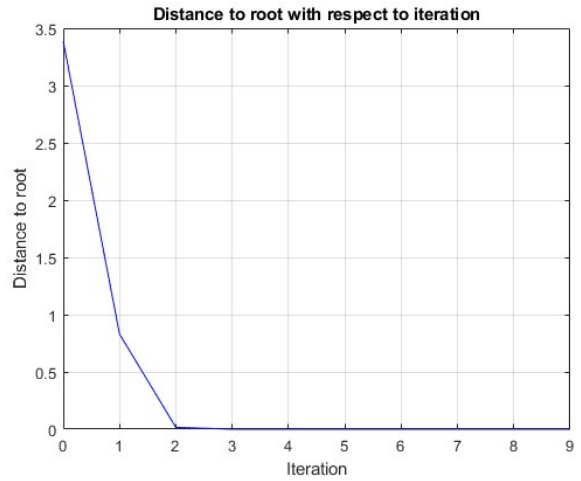
x	y
-1.19 - 0.58i	-5.21 - 19.73i
-0.56 - 0.82i	3.31 - 3.04i
-0.28 - 0.90i	0.29 - 0.08i
-0.26 - 0.91i	0.00 - 3.57e-05i
-0.26 - 0.91i	8.88e-15 - 4.44e-15i
-0.26 - 0.91i	0.00 + 8.88e-16i
-0.26 - 0.91i	8.88e-16 - 8.88e-16i
-0.26 - 0.91i	-6.66e-16 + 0.00i
-0.26 - 0.91i	0.00 + 8.88e-16i
-0.26 - 0.91i	8.88e-16 - 8.88e-16i



2.3 MM2 – Results

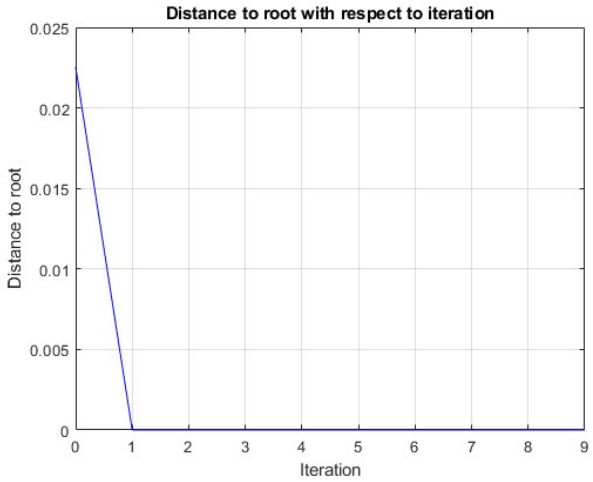
For x = -1 :

x	y
-0.50 - 0.39i	-1.25 - 3.14i
-0.27 - 0.02i	-0.81 - 0.15i
-0.14 + 0.01i	0.01 + 0.01i
-0.14 - 1.71e-08i	1.53e-08 - 1.12e-07i
-0.14 - 6.61e-24i	0. 00- 4.32e-23i
-0.14 + 0.00i	0.00 + 0. 00i
-0.14 + 0.00i	0.00 + 0. 00i
-0.14 + 0.00i	0.00 + 0. 00i
-0.14 + 0.00i	0.00 + 0. 00i
-0.14 + 0.00i	0.00 + 0. 00i



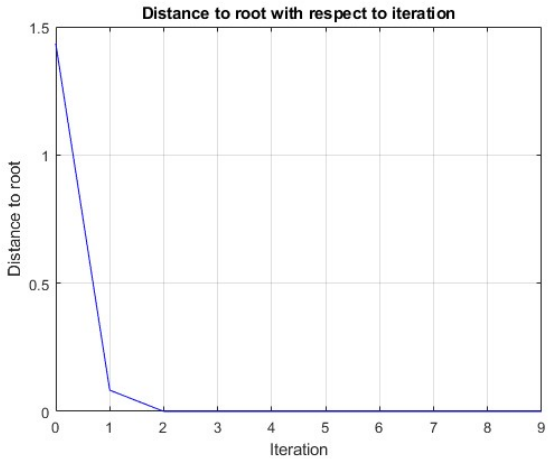
For x = 0 :

x	y
-0.15	-0.02
-0.14	2.96e-07
-0.14	0
-0.14	0
-0.14	0
-0.14	0
-0.14	0
-0.14	0
-0.14	0
-0.14	0



For x = 1 :

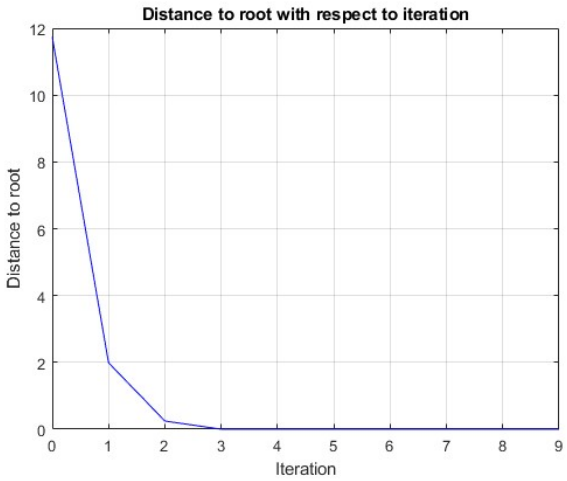
x	y
-0.27+ 0.18i	-0.75 + 1.21i
-0.16 + 0.00i	-0.08 + 0.001i
-0.14- 1.64e-07i	1.47e-05 - 1.07e-06i
-0.14 + 2.77e-18i	0.00 + 1.81e-17i
-0.14 - 3.85e-34i	0.00 - 2.51e-33i
-0.14 + 0.00i	0.00 + 0.00
-0.14 + 0.00i	0.00 + 0.00
-0.14 + 0.00i	0.00 + 0.00
-0.14 + 0.00i	0.00 + 0.00
-0.14 + 0.00i	0.00 + 0.00



2.3 MM2 – Results

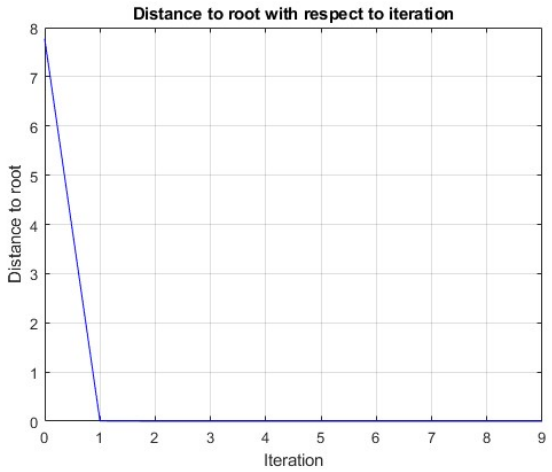
For x = 2 :

x	y
0.84 + 0.00i	11.73 + 0.00i
-0.19 + 0.34i	-0.22 + 1.97i
-0.15 + 0.03i	-0.03 + 0.24i
-0.14 + 5.41e-05i	-0.00 + 0.00i
-0.14 + 9.47e-14i	-1.27e-12 + 6.18e-13i
-0.14 + 0.00i	0.00 + 0.00i
-0.14 + 0.00i	0.00 + 0.00i
-0.14 + 0.00i	0.00 + 0.00i
-0.14 + 0.00i	0.00 + 0.00i
-0.14 + 0.00i	0.00 + 0.00i



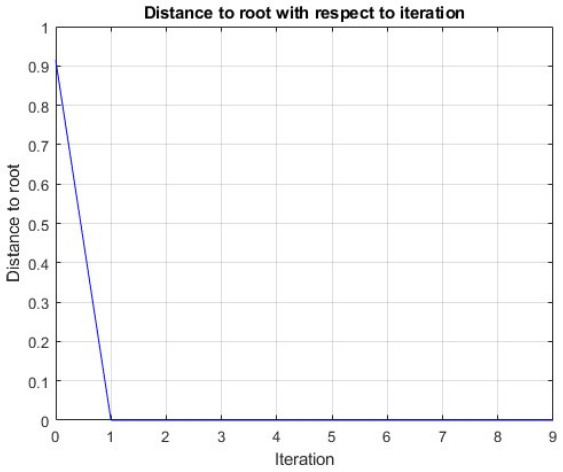
For x = 3 :

x	y
3.73	-7.77
3.67	0.004
3.67	-1.48e-12
3.67	-1.42e-14
3.67	-1. 42e -14
3.67	-1. 42e -14
3.67	-1. 42e -14
3.67	-1. 42e -14
3.67	-1. 42e -14
3.67	-1. 42e -14



For x = 4 :

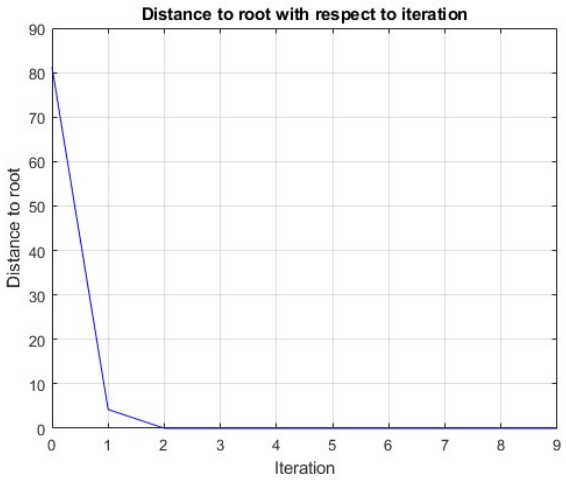
x	y
3.66	0.91
3.67	-9.23e-06
3.67	-1.42e-14
3.67	-1. 42e -14
3.67	-1. 42e -14
3.67	-1. 42e -14
3.67	-1. 42e -14
3.67	-1. 42e -14
3.67	-1. 42e -14
3.67	-1. 42e -14



2.3 MM2 – Results

For x = 5 :

x	y
3.76 - 0.58i	22.62 + 78.04i
3.68 - 0.03i	-1.06 + 4.09i
3.67 - 4.79e-06i	-0.00 + 0.00i
3.67 + 5.17e-17i	-7.10-14 - 6.48e-15i
3.67 - 1.23e-32i	3.19-14 + 1.54e-30i
3.67 + 0. 00i	-1.42-14 + 0.0i
3.67 + 0. 00i	-1.42-14 + 0.00i
3.67 + 0. 00i	-1.42-14 + 0.00i
3.67 + 0. 00i	-1.42-14 + 0.00i
3.67 + 0.00i	-1.42-14 + 0.00i



Conclusions : Looking at every plot we can see deduce that MM2 version is as good as MM1 one. The roots which we found are the same as previously what confirms the correctness of my algorithms. It also confirms that MM1 and MM2 versions have the same *order of convergence* (p = 1.84). More about comparison in section *Müller's method and Newton's method – Comparison*.

Roots found using MM2 version of the Müller's method				
Obtain Root	`-0.26 - 0.91i`	`-0.26 + 0.91`	`-0.14`	`3.67`
Real value of *f(x)*	`-6.66e-16 + 0.00i`	`-6.66e-16 + 0.00i`	`0`	`-1.42e-14`

Remark : Due to the basic principle of algebra if we find one complex root than its conjugation is also a root of that polynomial.

Test : Obtain values are consistent with build-in Matlab's function ***roots***, which gives the same results as my function what confirms the correctness of my algorithms.

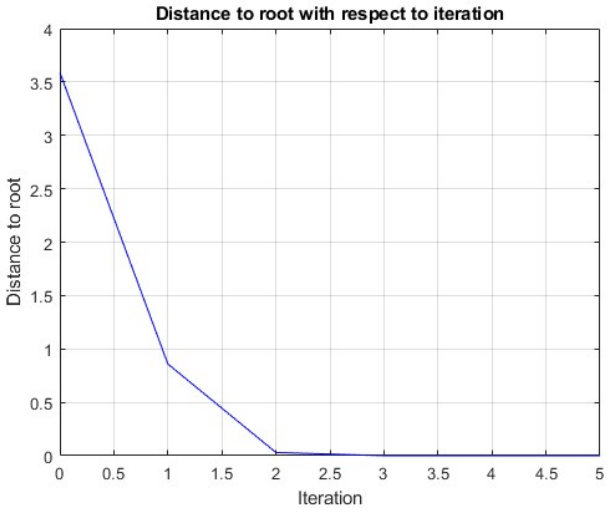
The implementation of MM2 method can be found in *Appendices*.

2.3 Newton's method – Results

Initial points that were picked are **-1** and **4**. They were picked directly form the plot that we draw in *Introduction* section. The tolerance was set to 10^{-20} and 10^{-13}.

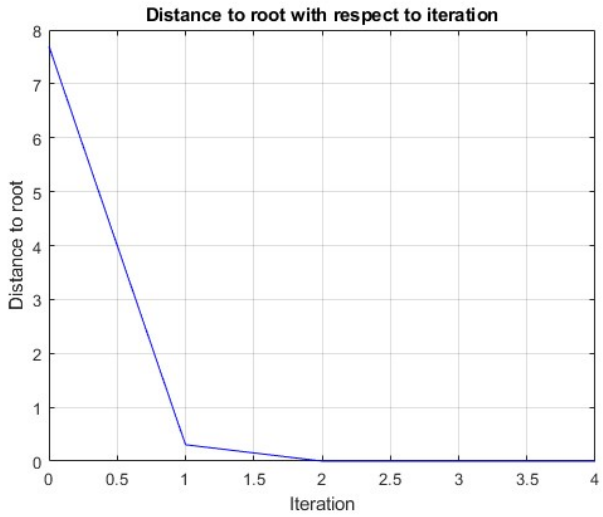
For x_n = -1 :

x	y
-0.59	-3.58
-0.27	-0.85
-0.15	-0.03
-0.14	-5.20-07
-0.14	4.44-16
-0.14	0



For x_n = 4 :

x	y
3.73	-7.69
3.67	-0.30
3.67	-0.01
3.7	-1.78e-09
3.67	-5.32e-14

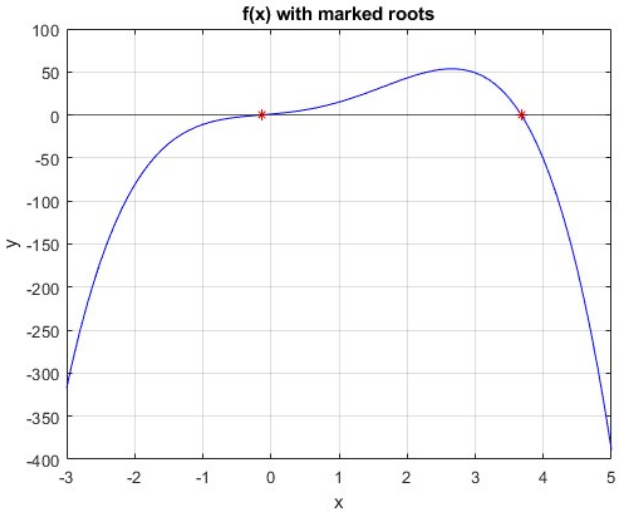


Roots found using Newton's method		
Obtain Root	-0.14	3.67
Real value of *f(x)*	0	-5.32e-14

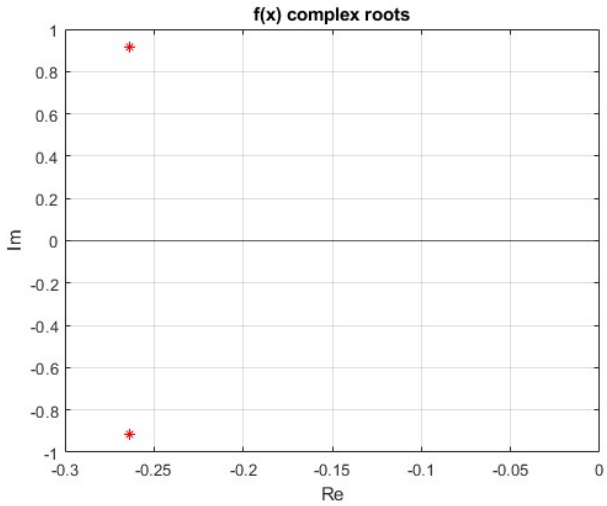
2.4 Müller’s methods and Newton’s method – Comparison

In general, Newton's methods, MM1, MM2, showed similar convergence rates, because the number of iteration is similar, we can state that it confirms the theory that the order of convergence are similar for Newton's method p = 2, for the Muller method p = 1.84.
As expected, the Müller method finds both real and imaginary roots, whereas Newton's method only finds real roots. For the proposed intervals, Muller's methods found one of the complex root, the other is a conjugated number. It is worth mentioning that due to the necessity of solving the system of equations in MM1 (finding coefficients *a* and *b*) for higher iterations, the matrix of this system loses its good conditioning, so that the algorithm ceases to give correct results.
Newton’s method for some initial point stopped to converge. These points are too far from the attractions of the root, or the derivative of the function gives a small value in that points. Due to the fact that in some points values of the derivative are small, Müller's methods are faster in such cases

Real roots marked on the plot :



Complex roots marked on the complex plane :



3.1 Introduction

The third task is to write a program which will find all zeros (real and complex) of the polynomial :

$$f(x) = -2x^4 + 6x^3 + 3x^2 + 7x + 1$$

using Laguerre's method.

(It is polynomial from task 2 – go to *Task 2 : Müller's methods - Introduction* to see the plot.)

The method is defined by the following formula :

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

where *n* denotes the order of the polynomial, and the sign in the denominator is chosen in a way assuring a larger absolute value of denominator, as in the Müller's method.

It is easy to observe that the formula is similar to the one used in MM2 method. The Laguerre's formula is slightly more complex, it takes also into account the order of the polynomial (it can be treated as an augmentation of formulas used in MM2) therefore the Laguerre's method is better, in general. In the case of polynomials with real roots only, the Laguerre's method is convergent starting from any real initial point, thus it is globally convergent. Despite a lack of formal analysis for a complex roots case, the numerical practise has shown good numerical properties also in this case (although situations of divergence may happen). The Laguerre's method is regarded as one of the best method for polynomial root finding.

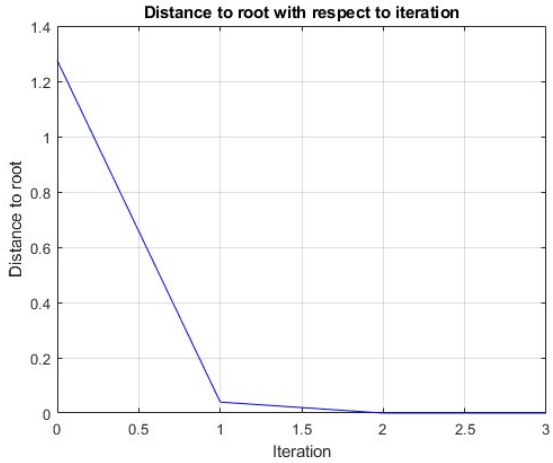
3.2 Results

Remark : Method was called for every integer x in the given interval $x = -3, \dots, 5$.

Below we can see the tables and plots for every first few iterations :

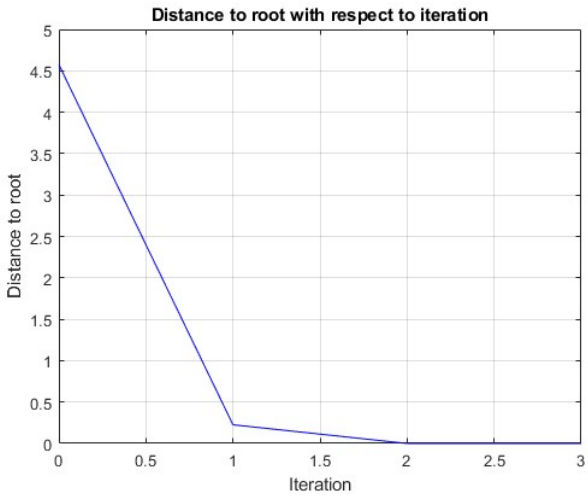
For x = -3 :

x	y
0.038	1.27
-0.15	-0.039
-0.1494	1.638e-06
-0.1494	0



For x = -2

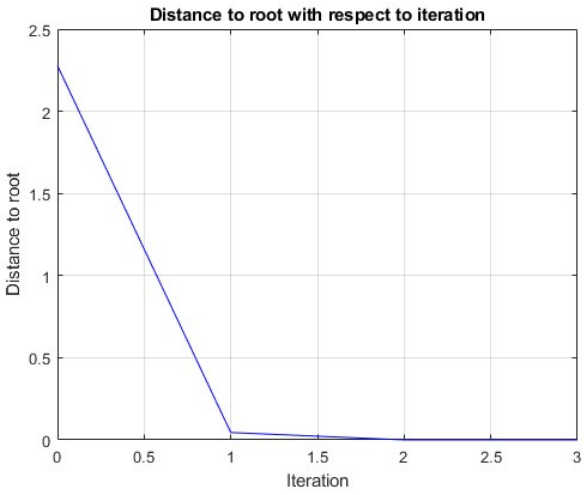
x	y
0.10 - 0.88i	-3.12 - 3.33i
-0.27 - 0.90i	0.19 - 0.11i
-0.26 - 0.91i	4.90e-06 + 1.51e-05i
-0.26 - 0.91i	8.88e-16 + 0.00i



3.2 Results

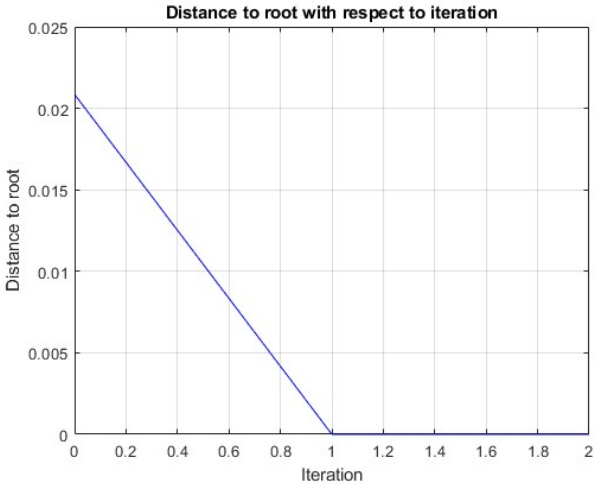
For x = -1

x	y
-0.08 - 0.8i	-1.34 - 1.83i
-0.26 - 0.91i	0.01 + 0.04i
-0.26 - 0.91i	1.14e-07 + 2.66e-08i
-0.26 - 0.91i	8.88e-16 - 8.88e-16i



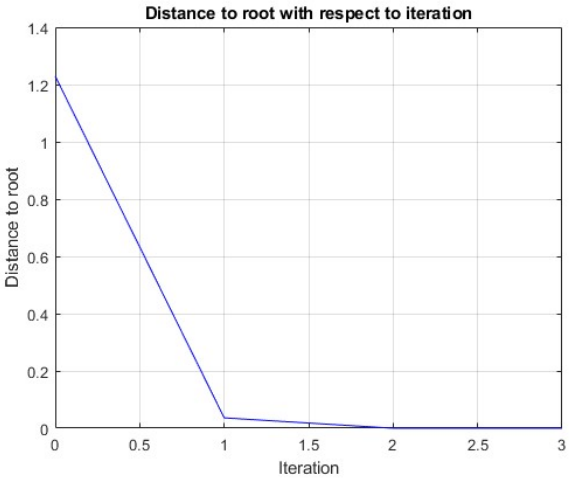
For x = 0 :

x	y
-0.15	-0.02
-0.14	2.34e-07
-0.14	0



For x = 1 :

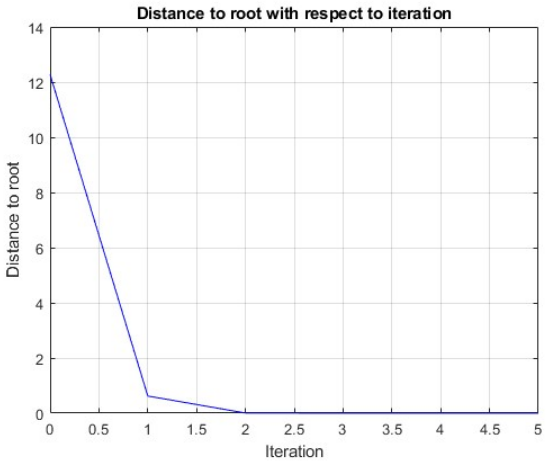
x	y
0.03	1.22
-0.15	-0.03
-0.14	1.22e-06
-0.14	0



3.2 Results

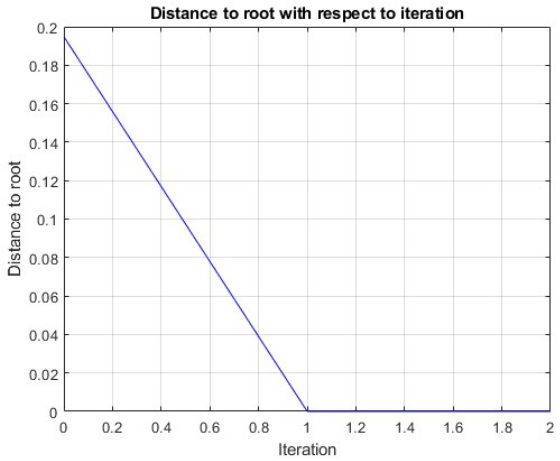
For x = 2 :

x	y
0.87	12.26
-0.05	0.62
-0.15	-0.005
-0.14	4.45e-09
-0.14	2.22e-16
-0.14	0



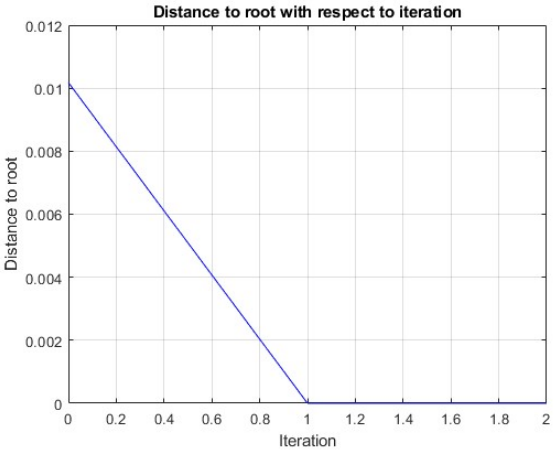
For x =3 :

x	y
3.67	-0.19
3.67	1.40 e-09
3.67	-1.42e-14



For x = 4 :

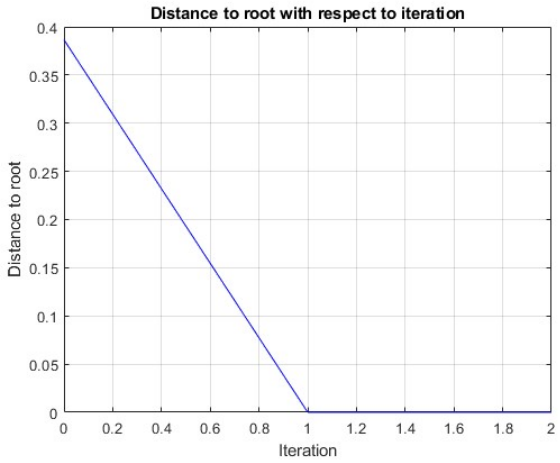
x	y
3.67	0.01
3.67	-2.41e-13
3.67	-1.42e-14



3.2 Results

For x = 5 :

x	y
3.67	0.38
3.67	-1.11e-08
3.67	-1.42e-14



Roots found using Laguerre's method				
Obtain Root	`-0.26 - 0.91i`	`-0.26 + 0.91`	`-0.14`	`3.67`
Real value of *f(x)*	`8.88e-16 - 8.88e-16i`	`8.88e-16 - 8.88e-16i`	`0`	`-1.42e-14`

Remark : Due to the basic principle of algebra if we find one complex root than its conjugation is also a root of that polynomial.

Test : Obtain values are consistent with build-in Matlab's function ***roots***, which gives the same results as my function what confirms the correctness of my algorithms.

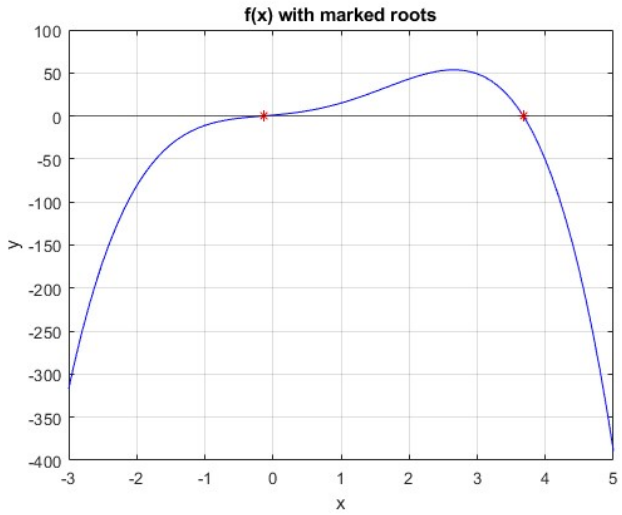
The implementation of Laguerre's method can be found in *Appendices*.

3.3 Laguerre's and MM2 method – Comparison

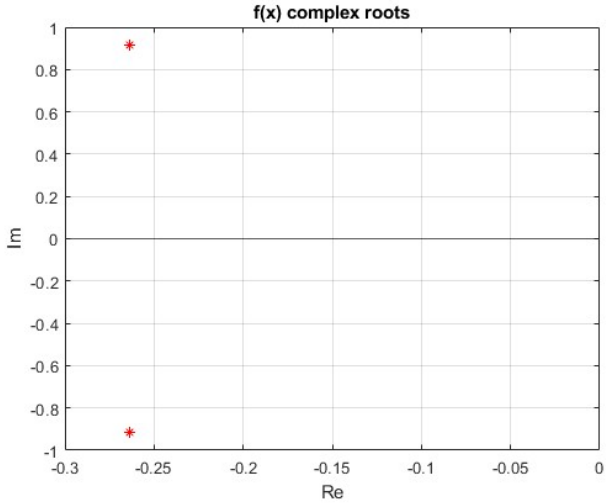
The Laguerre's method and MM2 method are very similar. The first one is an improvement of the second one. Both method computed the same roots but Laguerre's method did that around two times faster than MM2 method. It confirms statement we made in *Introduction* that it is faster than MM2. Also no matter which point I will pick I will always converge to one of roots what also is a big advantage of Laguerre's method over MM2 method.

Without surprise we obtain the same plots with roots marked as previously using Müller's method.

Real roots marked on the plot :



Complex roots marked on the complex plane :



4.1 Bisection method

```
function [x, y] = bisection(a, b, f, accuracy)
%bisection function finds the all roots of a given function
within the range
%a and b

x = [];
y = [];

if(f(a) * f(b) > 0)
    error('There is no root of function f between a and b');
end

c = a;
cnt = 1;
while abs(f(c)) > accuracy
   if abs(b - a) < 10^-15
       return;
   end

   c = (a + b) / 2;
   if f(a) * f(c) < 0
       b = c;
   else
       a = c;
   end

   x(cnt) = c;
   y(cnt) = f(c);
   cnt = cnt + 1;
end

end
```

4.2 Newton's method

```
function [x, y] = newton(xn, f, diffF, accuracy)
% newton is a function that computes root of a given function
% xn - starting point
% f - function f(x)
% diffF - first derivative of function f
% accuracy - wanted accuracy of the solution

cnt = 1;

while abs(f(xn)) > accuracy
        xn = xn - f(xn)/ diffF(xn);

        x(cnt) = xn;
        y(cnt) = f(xn);
        cnt = cnt + 1;
end


end
```

4.3 MM1 method

```
function [x,y] = MM1(x0, x1, x2, f, iter)
% x0,x1, x2 - starting points
% f - polynomial
% iter - number of iteration

for i = 1:iter
        %creating incremental variable
        z0 = x0 - x2;
        z1 = x1 - x2;

        %computing a and b of parabola
        c = f(x2);
        A = [z0^2 , z0; z1^2, z1];
        B = [f(x0) - c; f(x1) - c];
        sol = A\B;
        a = sol(1,1);
        b = sol(2,1);

        %picks the smallest
        if(abs(b + sqrt(b^2 - 4*a*c)) >= abs(b - sqrt(b^2 -
4*a*c)))
            zmin = (-2*c)/(b + sqrt(b^2 - 4*a*c));
        else
            zmin = (-2*c)/(b - sqrt(b^2 - 4*a*c));
        end

        %next approximation of root
        x3 = x2 + zmin;
        x(i) = x3;
        y(i) = f(x3);

        %remove the most far away point
        if(abs(x3 - x1) < abs(x3 - x0))
            temp = x1;
            x1 = x0;
            x0 = temp;
        end
        if(abs(x3 - x2) < abs(x3 - x1))
            temp = x2;
            x2 = x1;
            x1 = temp;
        end
        %Przygotowujemy się do kolejnej iteracji
        x2 = x3;

end

end
```

4.4 MM2 method

```
function [x, y] = MM2(xn, f, fir_der, sec_der, iter)

for i = 1:iter
        square_root = sqrt((fir_der(xn))^2 - 2 * f(xn) *
sec_der(xn));

        if(abs(fir_der(xn) + square_root) > abs(fir_der(xn) -
square_root))
            z = (-2*f(xn))/ (fir_der(xn) + square_root);
        else
            z = (-2*f(xn))/ (fir_der(xn) - square_root);
        end

        %compute next approximation
        xn = xn + z;
        x(i) = xn;
        y(i) = f(xn);
end

end
```

4.5 Laguerre's method

```
function [x, y] = laguerre(xn, f, n, fir_der, sec_der, iter)

    for i = 1:iter
        square_root = sqrt((n - 1)*((n - 1) * fir_der(xn)^2 -
n * f(xn) * sec_der(xn)));

        if(abs(fir_der(xn) + square_root) > abs(fir_der(xn) -
square_root))
            x(i) = xn - (n * f(xn)) / (fir_der(xn) +
square_root);
        else
            x(i) = xn - (n * f(xn)) / (fir_der(xn) -
square_root);
        end

        xn = x(i);
        y(i) = f(xn);
    end

end
```