ENUME
PROJECT B No. 21

Done by Filip Korzeniewski

1. Finding all zeros of function

Finding zeros of nonlinear function is proceeded by iterations method.
Firstly, it is required to find out how many zeros are in the whole interval of the function. The sufficient condition is that in local interval [a,b], f(a)*f(b) < 0. We know that there is at least one zero of the function in the interval. Wise way is to exceed the interval. When the interval is very small and f(a)*f(b) > 0 we can assume that there are no zeros in the interval. We exceed the interval with the step β until f(a)*f(b) < 0. When we have estimated the bracketing, we try to find its value. There are many iterations method capable of doing it. These methods start from some approximation of the local interval when the zero exists and narrow down the interval until it is smaller than some assumption δ. We can say that algorithm is convergent to α (α is the value of x, where f(x) = 0 in the local interval we were looking for then.

In general, index of convergence is the biggest number $p \geq 1$ like:
$\lim_{n\to\infty} \frac{|x_{n+1}-\alpha|}{|x_n-\alpha|^p} = k < \infty$, where k is coefficient of convergence

If p = 1, the method is linearly convergent then (in this case for convergent k < 1 is required). If p = 1 and k converges to 0 with increasing n, the method is superlinearly convergent. If p = 2, the method is quadratically convergent then. The bigger is index of convergence, the faster is the method.

There is given a function:
$f(x) = 1.4 * x * \cos(x) - \ln x$, where $x \in [2,11]$

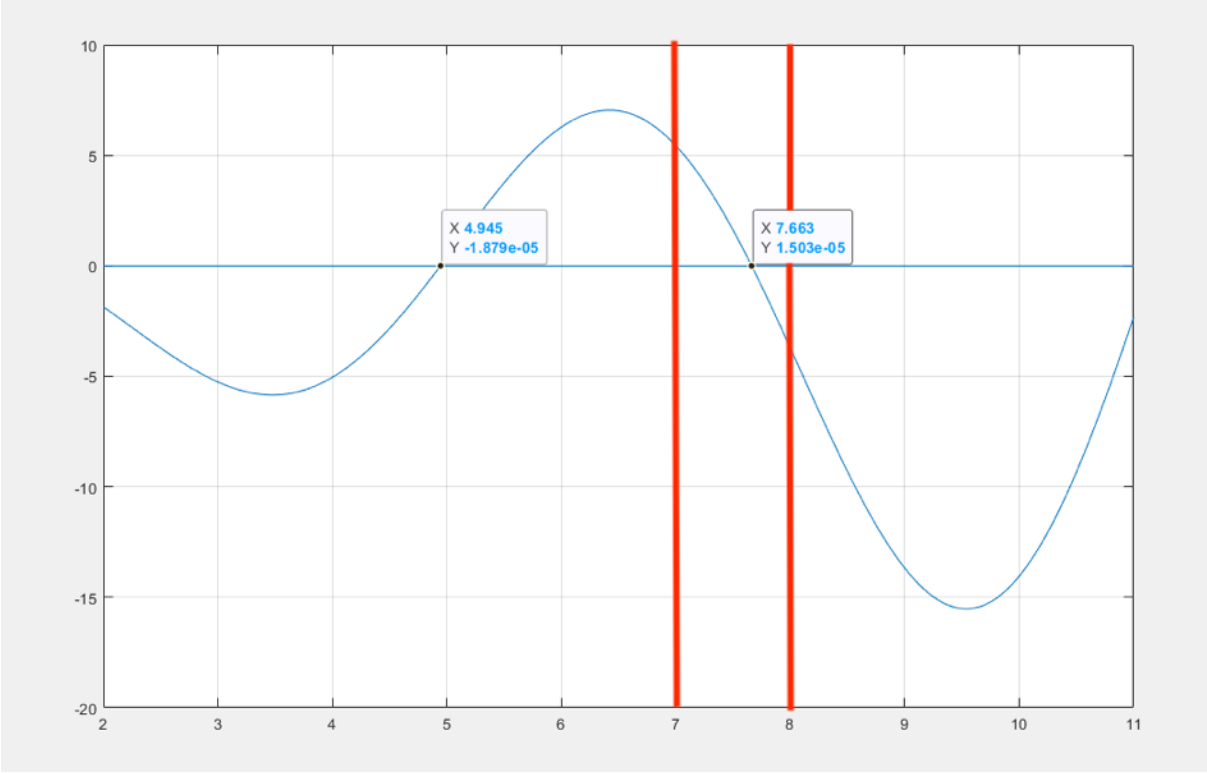

Figure 1: graph of the function with marked zeros and initial interval

It is said to find all zeros of the function using Two methods:

a) the bisection method;
b) the Newton's method.

Bisection method:

We start from interval [a, b] = [a_0, b_0] bracketing. In every iteration of the method:

1. Current interval containing zero of function [a_n, b_n] is divided into 2 halves with midpoint c_n:

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

2. The value f(c_n) is counted down. Multiplication is proceeded then – f(a_n)*f(c_n) and f(c_n)*f(b_n). One of these products is smaller than 0. The new interval is [a_{n+1}, b_{n+1}] equals the one that was smaller than zero ([a_n,c_n] or [c_n, b_n]).

Procedure is repeated as long as e.g. f(c_n) ≤ δ. It can be inaccurate if the function is "flat" in the area of root.

In our task first action is to find all subintervals of interval [2,11], where there is exactly one zero of given function. Then it is needed to conduct whole bisection method procedure.

Results:

	a	b
First root	4.94536345653521	4.94536415895449
Second root	7.66300451516603	7.66300581111883

Figure 2: intervals of function

f(first root)	f(second root)
8.36132133308354e-07	-5.69371626735205e-07

Figure 3: checking of values

We can see that value for either the first or second root is smaller than given delta. It suggests that the value got calculated properly. The interval where the root exists starts to differ at 10^{-6} value. It shows that accuracy is smaller at x axis. We can claim that for really "flat" surrounding (the more horizontal is tangent in area of root, the worse accuracy is) of root, the accuracy is not proper. In this case it is enough, but the difference is about 10. The bisection method is convergent linearly – $p = 1$ with index of convergence $k = 0.5$.

Newton's method:

Newton's method (tangent method) is about linear approximation of function. It results of cut off the expansion of Taylor series in current point x_n

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

Next point x_{n+1} results of an equation:

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

that leads to iteration formula:

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

The Newton's method is locally (in surrounding of root) convergent and is very quick there. The convergence is quadratic (index of convergence p = 2). The problem is when it is used to far from the root – it can be divergent there. It is more effective if the function is not "flat" and it should not be used if the function is almost horizontal (even in surrounding of root). The accuracy is strictly dependent on shape of function in the area of root.

Because of all restriction with using it locally, there is need to find intervals in the vicinity of root.

Results:

Local extremum is near to x = 3.5 (minimum) and x = 6.5 (maximum). When we try to pass the extremum with Newton's method the method is divergent then. It was confirmed using Matlab – it cannot be computed.
We need then give interval that is contained between extrema and contains the zero.

	a	b
First root	4.94536393708203	4.94536368581484
Second	7.66300510857952	7.66300510838418

Figure 2: intervals of results

These are intervals for precision 10e-7

f(first root)	f(second root)
8.61529353191060e-07	-1.01561692389396e-09

Figure 3:Checking of values

We can see that in the vicinity of second root (where the function is more vertical, which means that should be the better case for using Newton's function) the value is with precision 10e-9. It means that in previous iterations precision was smaller than 10e-7 and one iteration after the precision is 10e-9 which confirms that Newton's function works better with more vertical functions.

Conclusions and comparison:

Firstly, it can be said that Newton's function is not universal because does not work in every condition. In comparison to bisection method, which works in every condition it is its big disadvantage. When the same intervals of a,b are being compared when both function work, the Newton's method is much better. It can be concluded by examination the number of iterations. When precision was set to 10e-10, the Newton's method must have done 6 iterations for first root and 5 iterations for second root. The results were very good – first root was calculated with precision 10e-14 (using 6 iterations), second one with precision 10e-10 (using 5 iterations).
When the same intervals were applied to Bisection method, the method must have done for counting every root 32 iterations to achieve precision 10e-10.
It shows that Newton's method is much more efficient, but it cannot be used always.

Initial interval comparison:

For interval [7,8] there is:

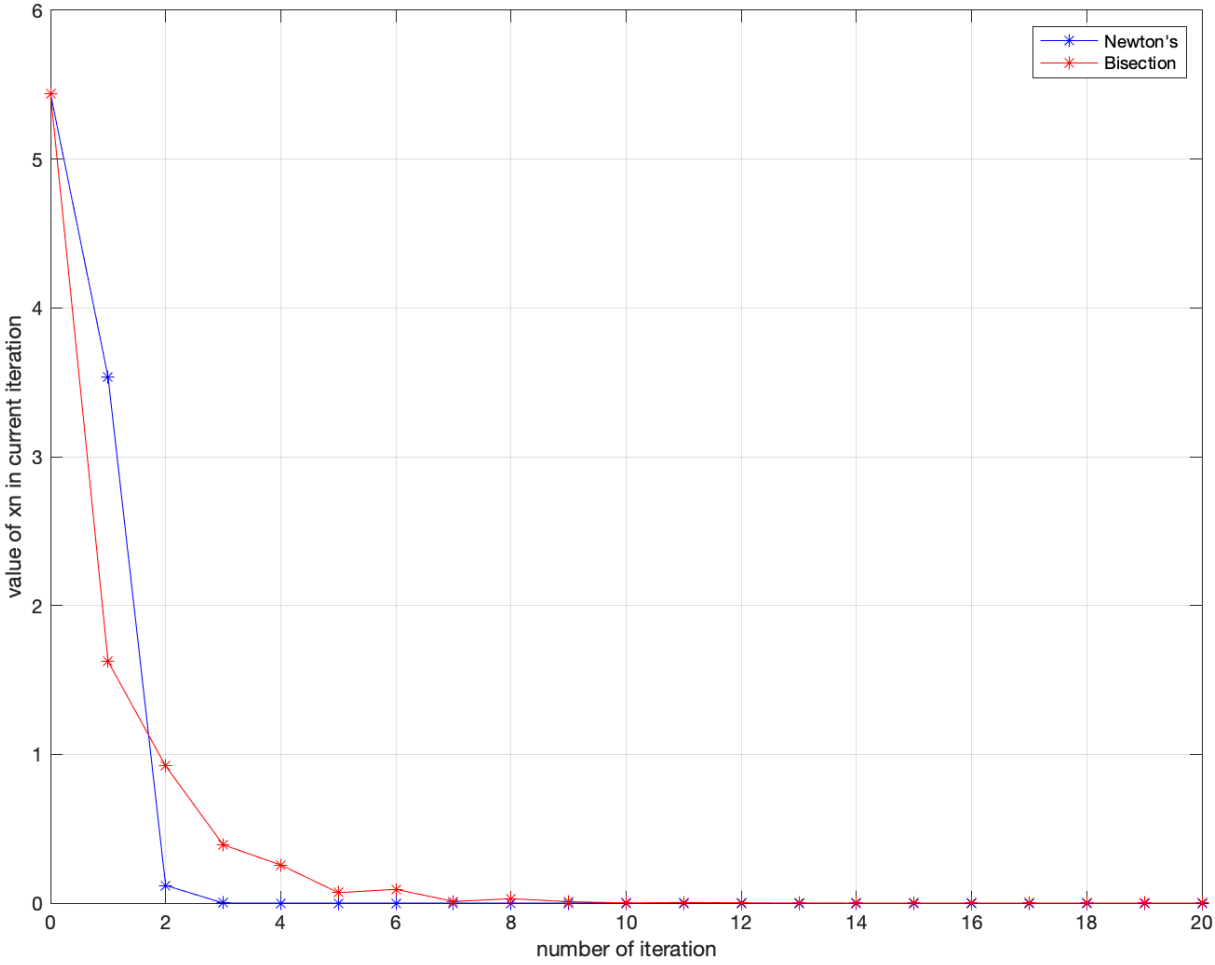

Figure 4: comparison of convergence Bisection - Newton's

	Bisection		Newton's	
iter.	f(x)	x	f(x)	x
1	1.62476781672551	7.50000000000000	-3.53670732495994	7.98488367953775
2	-0.921524067536361	7.75000000000000	-0.118851035705754	7.67440581397774
3	0.391641697944159	7.62500000000000	-0.000303504573050617	7.66303429717681
4	-0.256102355355106	7.68750000000000	-2.03122096920083e-09	7.66300510857952
5	0.0701297373855772	7.65625000000000	-3.55271367880050e-15	7.66300510838418
6	-0.0924148118357329	7.67187500000000	-3.55271367880050e-15	7.66300510838418
7	-0.0109973304258086	7.66406250000000	-3.55271367880050e-15	7.66300510838418
8	0.0296027952177056	7.66015625000000	-3.55271367880050e-15	7.66300510838418
9	0.00931184414420594	7.66210937500000	-3.55271367880050e-15	7.66300510838418
10	-0.000840469733023319	7.66308593750000	-3.55271367880050e-15	7.66300510838418
11	0.00423625612396483	7.66259765625000	-3.55271367880050e-15	7.66300510838418
12	0.00169803535427704	7.66284179687500	-3.55271367880050e-15	7.66300510838418
13	0.000428818341479342	7.66296386718750	-3.55271367880050e-15	7.66300510838418
14	-0.000205816814165427	7.66302490234375	-3.55271367880050e-15	7.66300510838418
15	0.000111502984196488	7.66299438476563	-3.55271367880050e-15	7.66300510838418
16	-4.71563598662961e-05	7.66300964355469	-3.55271367880050e-15	7.66300510838418
17	3.21734509465266e-05	7.66300201416016	-3.55271367880050e-15	7.66300510838418
18	-7.49141976497114e-06	7.66300582885742	-3.55271367880050e-15	7.66300510838418
19	1.23410242647282e-05	7.66300392150879	-3.55271367880050e-15	7.66300510838418
20	2.42480441858817e-06	7.66300487518311	-3.55271367880050e-15	7.66300510838418

Figure 5: comparison of values

This example confirms what was mentioned before – the Newton's is much faster. In this case Newton's converges quickly – in about 3rd iteration it has the value as Bisection in about 10th and its converges much more quicker with every iteration.

2. Finding all (real and complex) roots of the polynomial

There is given a polynomial:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

The polynomial has exactly *n* roots (zeros of polynomial). Also:

- Roots can be real and complex (pairs of conjugate numbers),
- Roots can be single or multiple.

For seeking real roots of polynomials, we can use previously shown methods (Bisection, Newton's and a couple of others). There are different, more complex methods, created especially for polynomials because of usage specific features of these functions (like multiple differentiability) and allowing to find also complex roots of polynomials.

The actual task:

There is given a polynomial:

$$f(x) = 2x^4 + 2x^3 - 5x^2 + 2x + 2$$

The task is to calculate its roots using Müller's method – MM1 and MM2 and compare the results between them. Furthermore, there is required comparison of results between MM2 and Newton's method for real roots (using the same initial points).

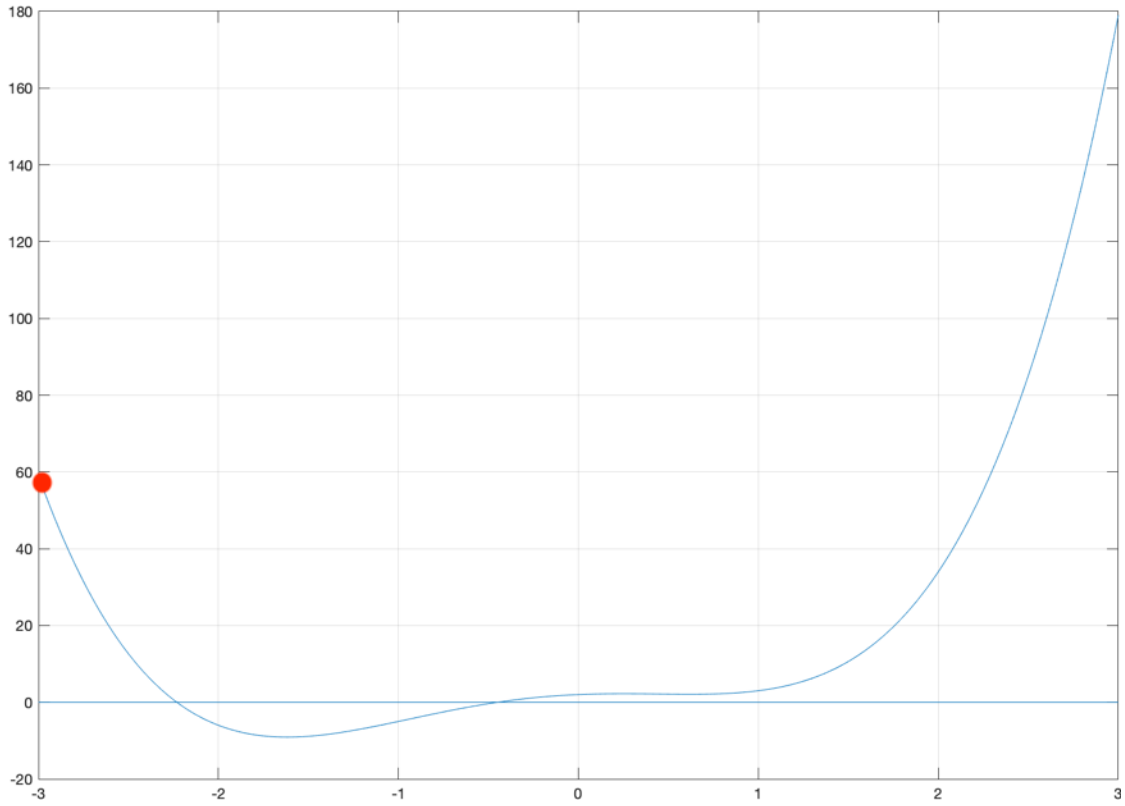

Figure 6: graph of the function [-3,3] with marked initial point

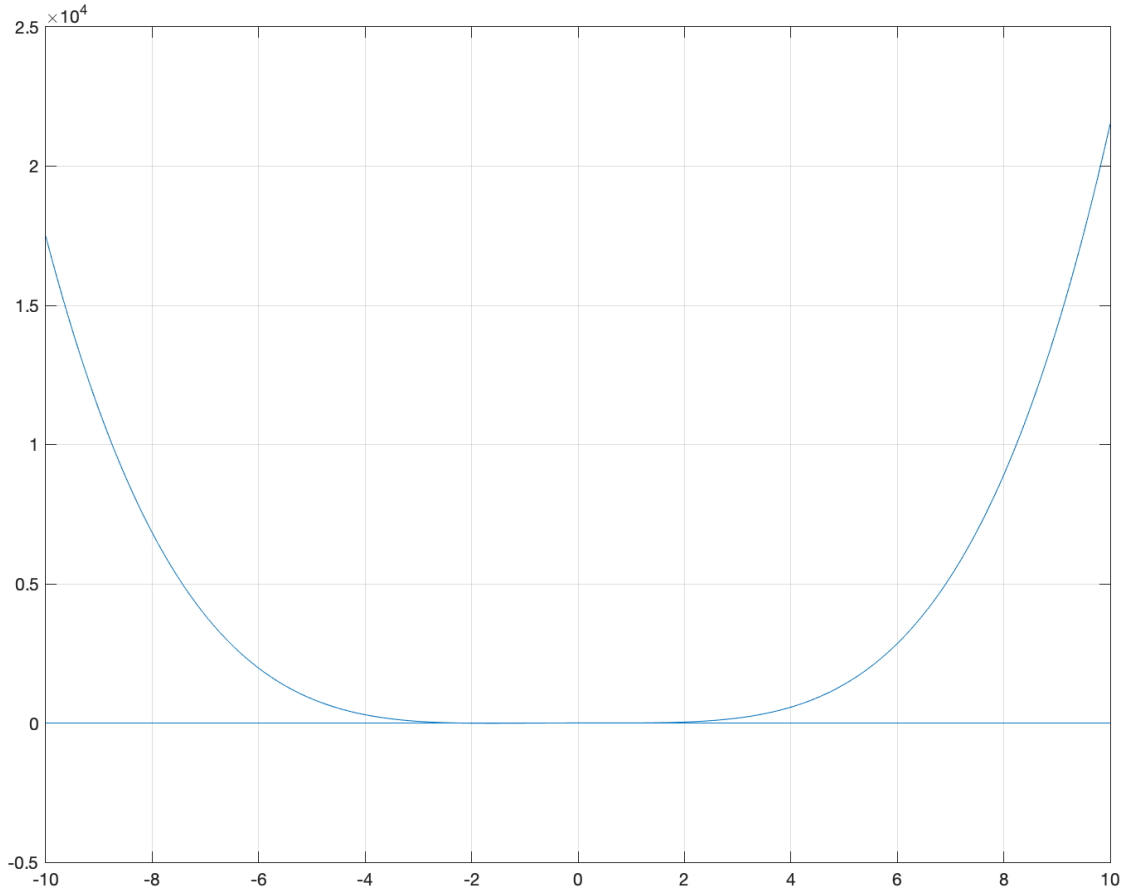

Figure 7: graph of the function [-10,10]

Müller's method:

Method uses approximation of polynomial in the vicinity of quadratic function's root. There are two submethods MM1 and MM2.

MM1:

There are three points x_0, x_1, x_2 and polynomial values in these points – $f(x_0)$, $f(x_1)$, $f(x_2)$. We will create quadratic function (the function will contain x_0, x_1, x_2). In next step we will find roots of this function and assume one of them as an approximation of root of polynomial. Let's take that x_2 is current approximation of polynomial's root. Let z be the iterational variable.

$$z = x - x_2$$

The differences:

$$z_0 = x_0 - x_2$$
$$z_1 = x_1 - x_2$$

The parabola:

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

Taking into consideration three given points, we have:

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

Hence, to evaluate a,b, we need to solve system of two linear equations:

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

Because we are interested in the root $y(z) = az^2 + bz + c$ with the minimal absolute value (that is placed as close as possible the x_2), to solve it numerical the best way is to use:

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

For another approximation we take root placed as close as possible to x_2 (the minimal absolute value):

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

where:

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

$z_{min} = z_-$, in other case.

Before going to next iteration, we reject the point placed the farthest from recently taken approximation point – x_3.

This algorithm will also be able to solve complex roots because it will work even if $b^2 - 4ac < 0$.

Results:

The polynomial has 4 roots because of the fact that maximum exponent equals 4. The roots were counted with precision 10e-10. It has been checked and this precision was achieved in no more than 23 iterations.

23	8	14	12
0.8397247358 - 0.5430123092i	0.8397247358 + 0.5430123091i	-0.4481741944	-2.2312752773

Figure 8: Roots counted with MM1 method and iterations

-1.140865180104e-11 + 4.60984583838e-11i	-6.2350125062948e-13 - 2.320366121466e-13i	1.5604850744921 4e-11	5.84954307214503 e-12

Figure 9: Checking the values

As we can see the precision that was achieved is very satisfying.

-2.23127527732552	0.839724735885168 - 0.5430123091998i	0.839724735885168 + 0.5430123091998i	-0.44817419444481

Figure 10: Results (roots) of the built-in function

-3.433919815165e-12 - 3.65585339778e-12i	1.554312234475e-14 + 6.8722805224e-14i	2.2397639298e-12	-1.6608936448e-13

Figure 11: Comparison of results - built in-MM1

We can see that using MM1 function the results are decent with even bigger precision that previously assumed.

MM2:

This version uses one point, not three as in previous case. For this point x_k there is calculated value of polynomial, its first derivative and second derivative. This version should be more efficient due to calculating value and two derivatives in *k* points (MM2) instead of calculating values of polynomial in *k+1* points (MM1).

By definition *y(z)*, for $z = x - x_k$, in point z = 0:

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

What leads into:

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

For approximation α we take the root of parabola with smaller absolute value:

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

Where z_{min} is taken among $\{z_+, z_-\}$ in the same way as it was in version MM1. This method works for complex roots. Moreover, is more efficient (locally) than secant method and not so much slower than Newton's method. What is more, looking at it, we can notice that this method should work not only for polynomials but also for different nonlinear functions.

Results:

The roots were counted with precision 10e-10. It has been checked and this precision was achieved in no more than 7 iterations.

6	3	7	11
-2.2312752773255	-0.44817419444481	0.839724735885168 + 0.5430123091998 78i	0.839724735885168- 0.5430123091998 78i

Figure 12: Roots counted with MM2 method and iterations

1.77635683940025e-15	0	6.66133814775094e-16 - 4.44089209850063e-16i	-8.88178419700125e-16 + 6.66133814775094e-16i

Figure 13: checking the values

As we can see the precision that we achieved is much better than we assumed (10e-10). In case of one root (-0.4417…) the precision that was achieved with one more iteration (second iteration was not enough to achieve 10e-10, but third caused that precision equaled eps).

-2.23127527732552	0.839724735885168 - 0.54301230919981 98i	0.839724735885168 + 0.54301230919981 98i	-0.44817419444481

Figure 14: Results (roots) of the in-built function

2.664535259100e-15	-1.66533453693e-16	3.330669073875e-16 + 1.08602461839 97i	3.330669073875e-16 - 1.08602461839 9i

Figure 15: comparison of results - built-in-MM2

Assuming precision at least 10e-10, we can see that one more iteration caused big improvement of results' precision. The results are in 10e-16 row of difference comparing to built-in function.

MM1 – MM2 comparison:

We can generally see that MM2 method is much better than MM1 method. The MM2 method is faster convergent and is more efficient. The convergence is about twice faster generally – for MM1 in the worst case were 18 iterations while for MM2 it was 10. Besides as it has been already mentioned, the MM2 method is more efficient to computing due to calculating value and two derivatives in k points (MM2) instead of calculating values of polynomial in $k+1$ points (MM1).

Below there is a graph comparing counting the real root – -2.23… **in the vicinity of -3** (initial point).

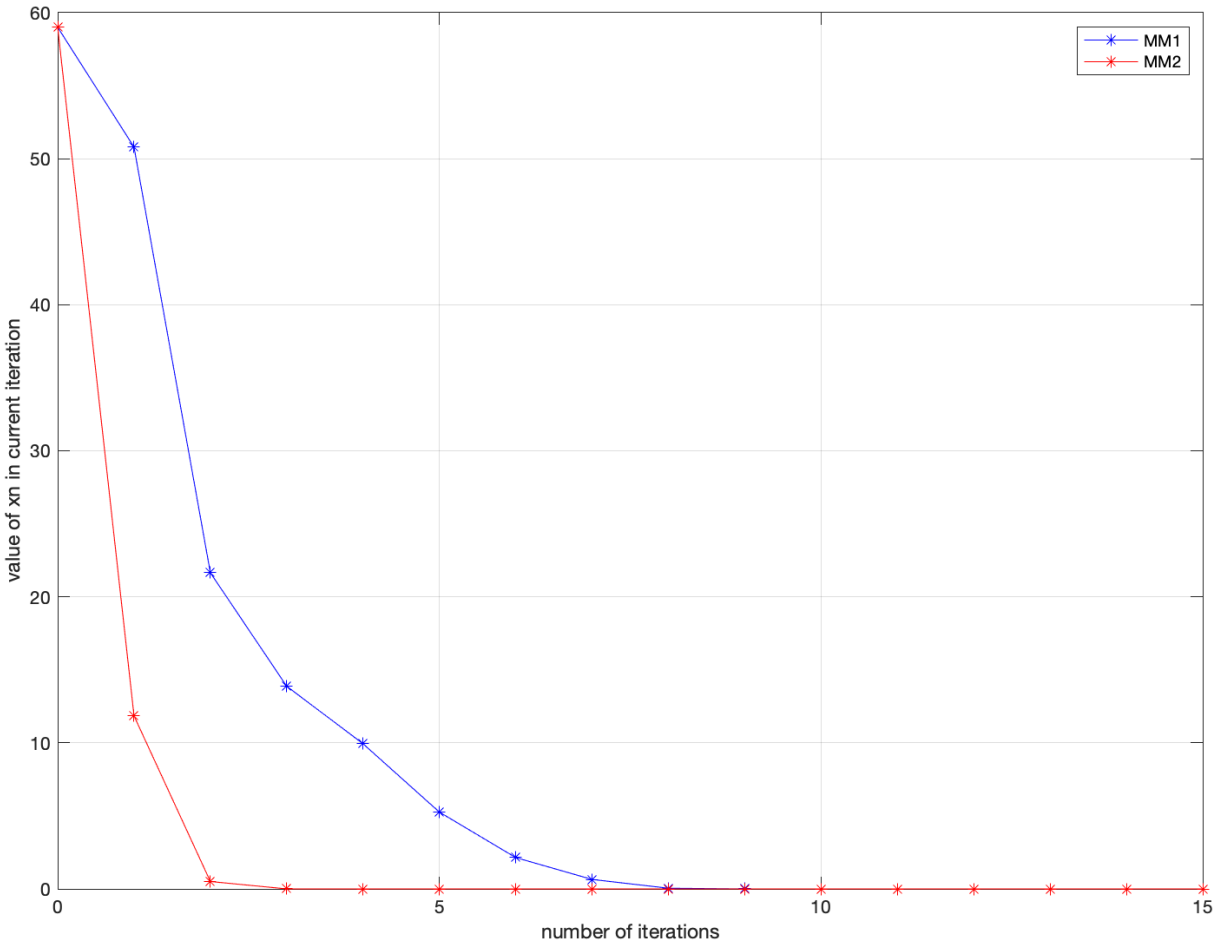

Figure 16: Convergence (bad case) for both methods

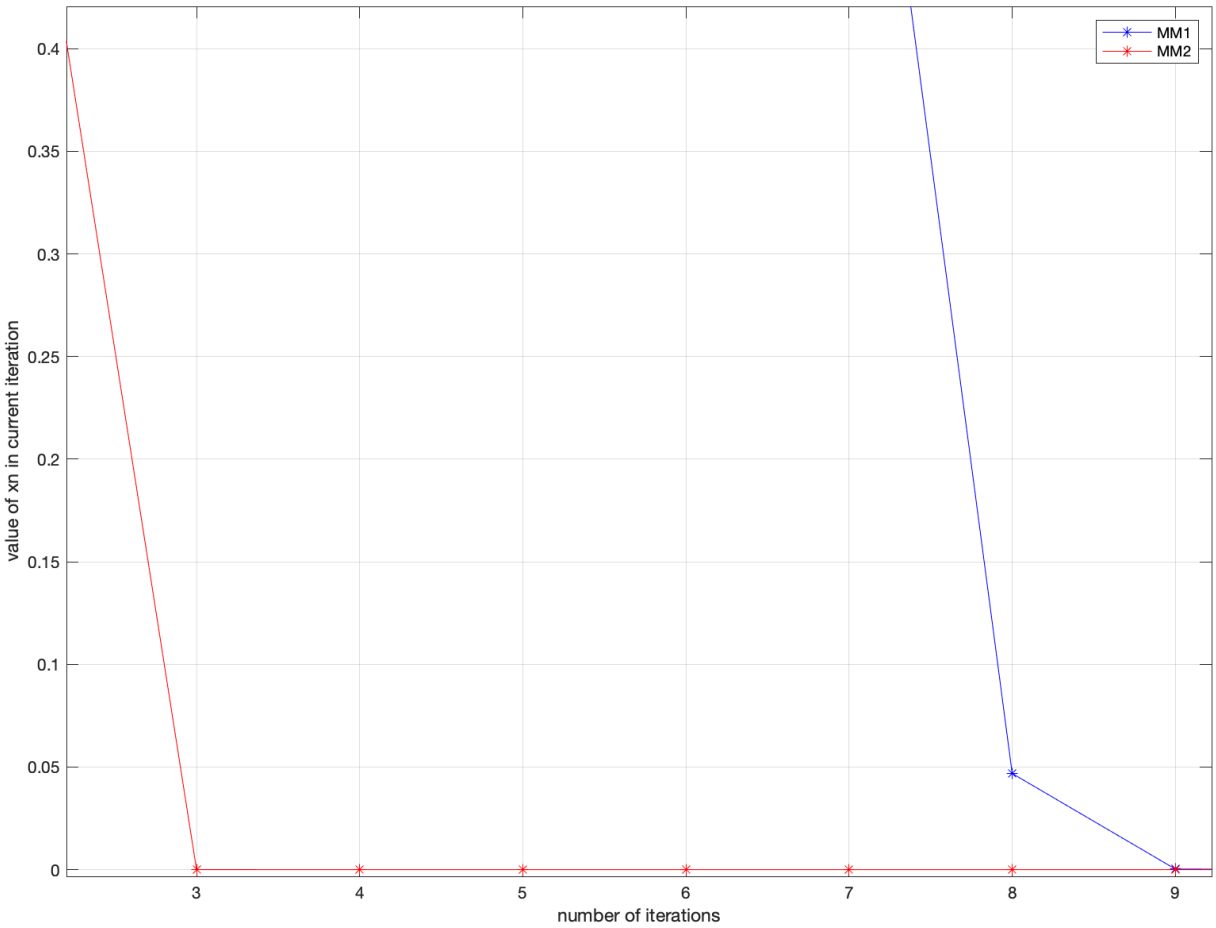

Figure 17: detailed graph of convergence

	MM1		MM2	
iterations	f(x)	x	f(x)	x
1	12.3112640062871	-2.77575757575758	-4.37885178577843	-2.23529411764706
2	-13.5411560712286	-2.22985235284064	-0.0401159032598120	-2.23037030483196
3	-13.8773839390692	-1.70734969983257	-9.37331113082252e-06	-2.23127500708309
4	-8.69659094805498	-1.25250103925355	1.15463194561016e-14	-2.23127527732552
5	-3.31888948100245	-0.856222594873069	1.77635683940025e-15	-2.23127527732552
6	-0.453357999590438	-0.551082097114805		
7	0.467482477757571	-0.381767211134845		
8	-0.00209528820546545	-0.448509058075748		
9	0.000231801610867333	-0.448140921816377		
10	4.45990193664869e-08	-0.448174188043111		
11	1.77635683940025e-15	-0.448174194444817		

Figure 18: Comparison of values

We can see that method MM2 is convergent in 3rd iteration, while MM1 is convergent at 9th iteration. It means that MM1 needs in this case 3 times more iterations to be convergent. It confirms the theory and previous conclusions.

MM2 – Newton's method comparison

Comparing Newton's and MM2 by definition we should obtain similar effect but the MM2 method should be a bit slower (index of convergence $p = 2$ for Newton's and $p = 1.84$ for MM2). In my case it occurred that locally in the interval that both functions started, the MM2 is faster convergent. It can be connected with the fact that (as mentioned previously), the Newton's method depends on shape of the function in the vicinity of root and is not so good when the function in the vicinity of root is more horizontal. This situation took place in this case. This could be the main reason of faster convergent MM2 than Newton's then.

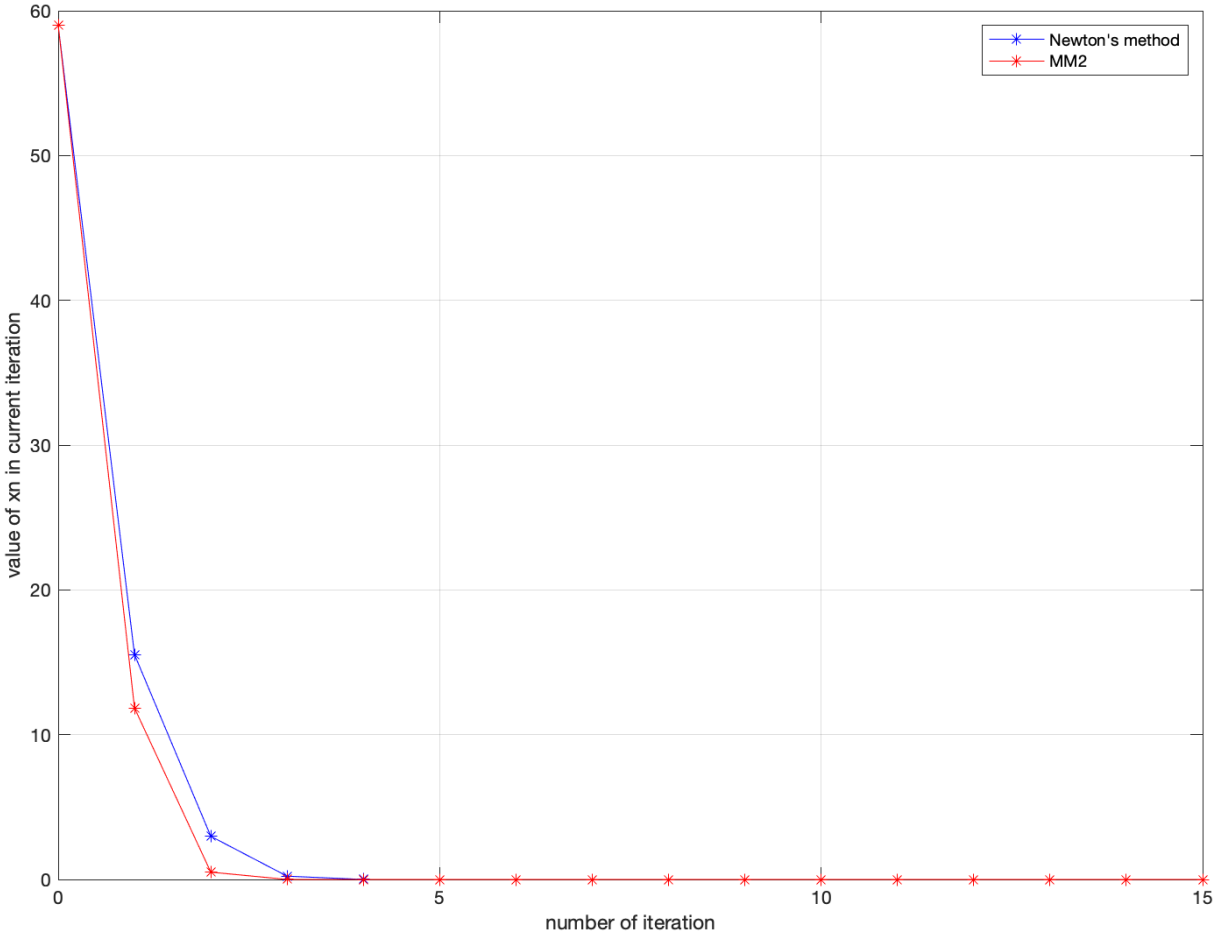

Figure 19: comparison of Newton's and MM2 convergence

	MM2		Newton's	
iterations	f(x)	x	f(x)	x
1	-4.37885178577843	-2.23529411764706	15.5362462168691	-2.54615384615385
2	-0.0401159032598120	-2.23037030483196	2.98034100194531	-2.30965516443530
3	-9.37331113082252e-06	-2.23127500708309	0.227236813327103	-2.23777629065505
4	1.15463194561016e-14	-2.23127527732552	0.00172972062316568	-2.23132514443876
5	1.77635683940025e-15	-2.23127527732552	1.02831423376415e-07	-2.23127528029028
6			1.77635683940025e-15	-2.23127527732552

Figure 20: Comparison of values

3. Finding all roots of polynomial using Laguerre's method

Laguerre's method is defined by formula:

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

Where *n* is degree of polynomial, and the sign in denominator take in the way to have the bigger absolute value. Laguerre's formula is the general formula of Muller and considers degree of polynomial. The method is a bit faster then. When taking into account polynomial with real roots, the Laguerre's method is convergent globally. Despite the lack of convergence analysis for case of complex roots, the method is considered to be good also for this case (but sometimes may be divergent). Generally, the method is considered to be one of the best methods for finding roots of polynomials.

The task:

The task is to find all roots of the of the polynomial from the previous task.

Results:

The Laguerre's method seems to not convergent for one root. Method found 3 roots instead of 4. It was checked precisely with interval [-100,100] with step 0.1. It is correct according to theory (the omitted root is the complex one). The precision was set to 10e-15

iterations	4	3	5
roots	-2.23127527732552	-0.448174194449912	0.839724735884386+ 0.543012309196404i

Figure 21: Roots calculated using Laguerre's method

1.77635683940025e-16	-3.54996032569943e-15	3.16446868708908e-15 + 1.16442411268736e-15i

Figure 22: Values of polynomial in the roots found by Laguerre's method

We can see that the values were calculated precisely.

Comparison MM2 – Laguerre's

We can see that for this case Laguerre's function is definitely slower than MM2 function. MM2 method is convergent to 0 with noticeable precision in 3 iteration, while Laguerre's in 8^{th}. It shows the fact that despite that Laguerre's method is considered as one of the best it has its disadvantages. Firstly, it sometimes can be not convergent like in this case – not found root. Besides sometimes it turned out to be much slower than some other method.

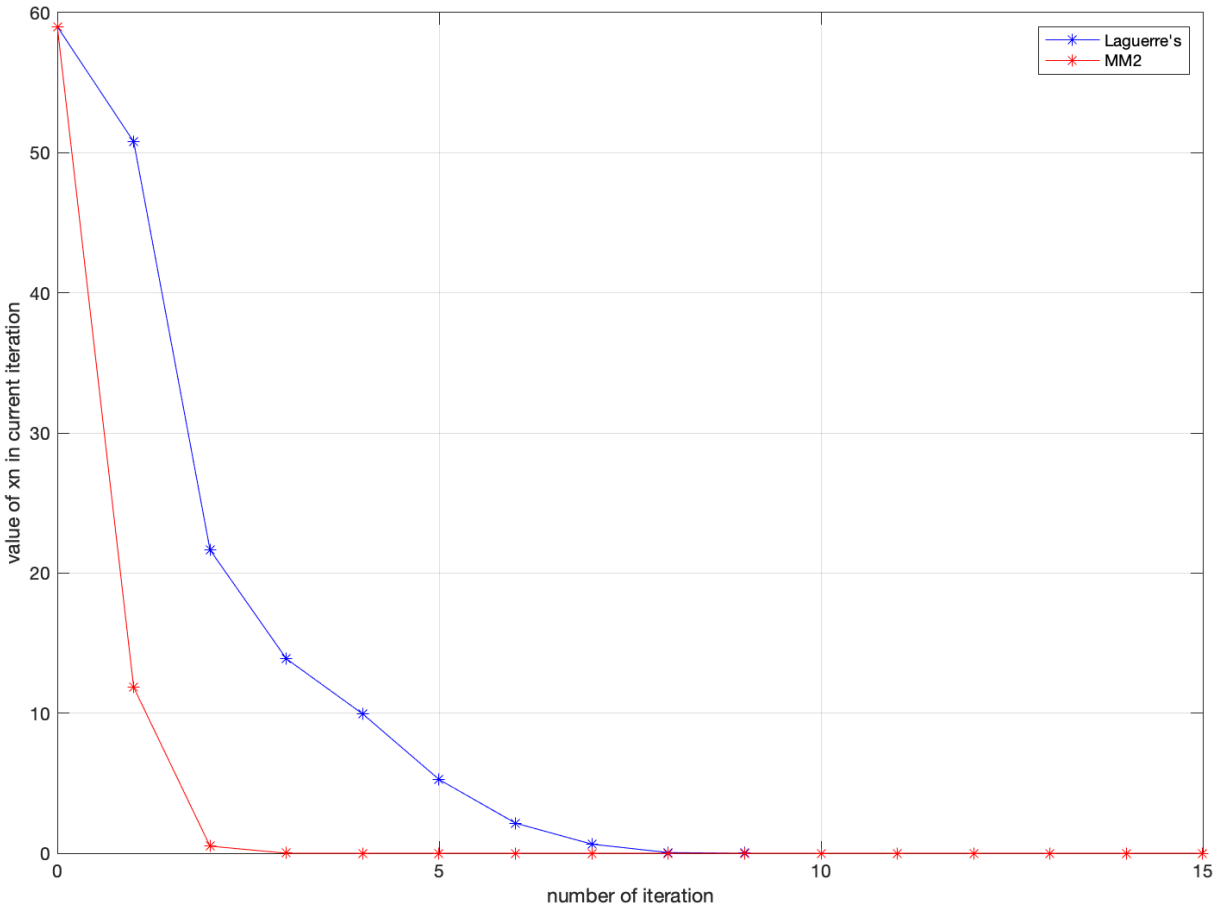

Figure 23: Comparison of Newton's - Laguerre's convergence

Laguerre's method usage for this function is definitely not efficient. It is much better to use Newton's one in this case.

	MM2		Laguerre's	
iterations	f(x)	x	f(x)	x
1	-4.37885178577843	-2.23529411764706	12.3112640062871	-2.77575757575758
2	-0.0401159032598120	-2.23037030483196	-13.5411560712286	-2.22985235284064
3	-9.37331113082252e-06	-2.23127500708309	-13.8773839390692	-1.70734969983257
4	1.15463194561016e-14	-2.23127527732552	-8.69659094805498	-1.25250103925355
5	1.77635683940025e-15	-2.23127527732552	-3.31888948100245	-0.856222594873069
6			-0.453357999590438	-0.551082097114805
7			0.467482477757571	-0.381767211134845
8			-0.00209528820546545	-0.448509058075748
9			0.000231801610867333	-0.448140921816377
10			4.45990193664869e-08	-0.448174188043111
11			1.77635683940025e-15	-0.448174194444817
12			0	-0.448174194444817

Figure 24:Comparison of values

4. Code

Task 1:

Bisection method:

```
function [y] = myfunc(a)
y = 1.4 * a * cos(a) - log(a);
end
```

Figure 25: formula for counting value in given argument of the given function

```
function [intervalMatrix] = localInterval(a,b,delta)
%interval matrix includes values of all local interval where f(x1)*f(x2)<0
%it includes the interval where there is all zeros
if a < b
    x1 = a;
    x2 = a + delta * 10e2;
elseif a > b
    x1 = b;
    x2 = b + delta * 10e2;
end

length = abs(x1 - x2);
beta = 1.01;
k = 0;
intervalMatrix = zeros(1,2);

while x2 < b
    if myfunc(x1) * myfunc(x2) < 0
        k = k + 1;
        intervalMatrix(k, 1) = x1;
        intervalMatrix(k, 2) = x2;
        x1 = x2;
        x2 = x2 + delta * 10e3;
    else
        x2 = x2 + beta*(length);
    end
end

if myfunc(b) == 0
    k = k + 1;
    intervalMatrix(k, 1) = b;
    intervalMatrix(k, 2) = b;
end

end
```

Figure 26: function for dividing general interval

This function divides general interval given by parameters a, b into a few new ones dependent on zeros. Every new interval contains one zero. The third parameter is δ. This function starts to exceed interval from length $\delta * 10^2$ and in every iteration the length of interval is multiplicated by 1.01. It returns the matrix [N×2], where N is the number of 0 in the function.

```
function [x] = bisection_funcLocal(a, b, delta, max_iter)
%function to find ONE zero of the function in local interval

if(a < b)
    x1 = a;
    x2 = b;
elseif a > b
    x2 = a;
    x1 = b;
elseif a == b
    if myfunc(a) == 0
        x = [a b];
    else
        disp("The interval equals 0")
        return;
    end
end


for i=1:max_iter

    c = (x1+x2)/2;

    if abs(myfunc(c)) <= delta
        x = [x1 x2];
        return;
    elseif myfunc(x1)*myfunc(c) < 0
        x2 = c;
    elseif myfunc(x2)*myfunc(c) < 0
        x1 = c;
    elseif myfunc(c) == 0
        x = [c c];
        return;
    end
end
x = [x1 x2];

end
```

Figure 27: bisection method function - one zero

This function allows to find interval of zero with δ accuracy. It is limited with maximum number of iterations given as a parameter. Function allows to find exactly one zero.

```
function [IntervalMatrix] = bisection_func(a, b, delta, max_iter)
%function that look for zeros in whole generally given interval

LocalIntervals = localInterval(a, b, delta);
IntervalMatrix = zeros(1,2);

for i=1:size(LocalIntervals)
    IntervalMatrix(i,:) = bisection_funcLocal(LocalIntervals(i,1), LocalIntervals(i,2), delta, max_iter);
end

end
```

Figure 28: general bisection function

This function allows to find all zeros of given function in interval given by parameters a, b, accuracy δ and maximum number of iterations.

```
% task1
%bisection method

y = bisection_func(2,11,10e-7,10e8);
check(1) = myfunc((y(1,1)+y(1,2))/2);
check(2) = myfunc((y(2,1)+y(2,2))/2);
```

Figure 29: Call of function

The function is called in interval [2,11] with $\delta = 10^{-7}$ and maximum number of iterations = 10^{8}

Newton's method

```
>> clear
>> syms x;
>> y = 1.4 * x * cos(x) - log(x);
>> py = diff(y)

py =

(7*cos(x))/5 - (7*x*sin(x))/5 - 1/x
```

Figure 30: Formula of function's derivative

```
function [y] = funcderiv(a)
%this function returns value of derivative in point

y = 1.4 * cos(a) - 1.4 * a * sin(a) - 1/a;
end
```

Figure 31: function counting derivative in point

```
function [x, iterations] = Newton_funcLocal(a,b,delta)
%function to find ONE zero of the function in local interval

iterations = 0;
if(a < b)
    x1 = a;
    x2 = b;
elseif a > b
    x2 = a;
    x1 = b;
elseif a == b
    if myfunc(a) == 0
        x = [a b];
        return;
    else
        disp("The interval equals 0")
        return;
    end
end

xn = x1;
xn1 = 0;

while xn1 < x2
    xn1 = xn - myfunc(xn)/funcderiv(xn);
    iterations = iterations + 1;

    if abs(xn1-xn) < delta
        x = [xn xn1];
        return;
    end
    xn = xn1;

end
end
```

Figure 32: Newton's function

Task2:

MM1:

```
function [y, iter] = MM1_final(x0, x1, x2, precision)
%function counting Roots

x = [abs(polyVal(x0)) abs(polyVal(x1)) abs(polyVal(x2))];
minim = min(x);
for i=1:2
    if x(i) == minim
        temp = x2;
        x2 = x(i);
        x(i) = temp;
    end
end

iter = 0;
while abs(polyVal(x2)) > precision
    [x0, x1, x2] = MM1_single(x0, x1, x2);
    iter = iter + 1;
end

y = x2;

end
```

Figure 33: Final function MM1

```
    function [y1,y2,y3] = MM1_single(x0, x1, x2)
    %function returning new 3 points

    [a, b, c] = coeff_count(x0, x1, x2);

    x3 = x3count(x2, a, b, c);

    x = [x0 x1 x2];
    dist = 0;
    for i=1:3
        newdist = abs(x3 - x(i));
        if newdist > dist
            dist = newdist;
            to_reject = i;
        end
    end

    x(to_reject) = x3;
    temp = x(3);
    x(3) = x(to_reject);
    x(to_reject) = temp;
    y1 = x(1);
    y2 = x(2);
    y3 = x(3);

    end
```

Figure 34: Function used for count every single approximation of MM1

```
function [a, b, c] = coeff_count(x0, x1, x2)
%counts coefficients of quadratic function created by points x1,x2,x3

c = polyVal(x2);

z0 = x0 - x2;
z1 = x1 - x2;

R1 = (polyVal(x0) - polyVal(x2)) / z0;
R2 = (polyVal(x1) - polyVal(x2)) / z1;

a = (R1 - R2) / (z0 - z1);
b = R1 - a*z0;

end
```

Figure 35: Function counting coefficients of quadratic function

```
function [z] = zmin(a, b, c)
%function counting zmin

delta = b^2 - 4*a*c;

if abs(b + sqrt(delta)) >= abs(b - sqrt(delta))
    z = (-2*c)/(b + sqrt(delta));
    return;
else
    z = (-2*c)/(b - sqrt(delta));
    return;
end

end
```

Figure 36: Function counting zmin

```
function [x] = x3count(x2, a, b, c)
%counts next approximation of zeros

x = x2 + zmin(a, b, c);

end
```

Figure 37: Function counting next approximation x3

MM2:

```
>> syms x;
>> y = 2*x^4 + 2*x^3 - 5*x^2 + 2*x + 2;
>> dy = diff(y);
>> ddy = diff(dy);
>> dy

dy =

8*x^3 + 6*x^2 - 10*x + 2

>> ddy

ddy =

24*x^2 + 12*x - 10
```

Figure 38: first and second derivative of function

```
function [z] = MM2_zmin(xk)
%function counting zmin minimal zmin

delta = (firstDerVal(xk))^2 - 2 * polyVal(xk) * secDerVal(xk);

if abs(firstDerVal(xk) + sqrt(delta)) >= abs(firstDerVal(xk) - sqrt(delta))
    z = (-2*polyVal(xk)) / (firstDerVal(xk) + sqrt(delta));
    return;
else
    z = (-2*polyVal(xk)) / (firstDerVal(xk) - sqrt(delta));
    return;
end


end
```

Figure 39: zmin of MM2 method

```
function [y] = MM2_single(xk)
%function counting next approximation of root

y = xk + MM2_zmin(xk);
end
```

Figure 40: function counting next approximation point of root

```
function [y, iter] = MM2_final(xk, precision)
%function counting root with given precision using MM2

iter = 0;
while abs(polyVal(xk)) > precision
    xk = MM2_single(xk);
    iter = iter + 1;
end

y = xk;

end
```

Figure 41: function counting local root MM2 - in the vicinity of given parameter

Task 3:

```
function [z] = LG_zmin(xk)
%counts z of the Laguerres function

delta = 3*(3*((firstDerVal(xk))^2) - 4*polyVal(xk)*secDerVal(xk));

if abs(firstDerVal(xk) + sqrt(delta)) >= abs(firstDerVal(xk) - sqrt(delta))
    z = (4*polyVal(xk)) / (firstDerVal(xk) + sqrt(delta));
    return;
else
    z = (4*polyVal(xk)) / (firstDerVal(xk) - sqrt(delta));
    return;
end

end
```

Figure 42: calculating zmin of Laguerre's

```
function [y] = LG_single(xk)
%function counting next approximation of root

y = xk - LG_zmin(xk);
end
```

Figure 43: Calculating one approximation using Laguerre's

```
function [y, iter] = LG_final(xk, precision)
%function counting root with given precision using Laguerre's method

iter = 0;
while abs(polyVal(xk)) >= precision
    xk = LG_single(xk);
    iter = iter + 1;
end

y = xk;

end
```

Figure 44: final function for calculating Laguerre's roots

```
%Lagguerre method

x = linspace(-100, 100, 100000);
nb = 0;
result = 0;
p = 0;
for i=1:size(x',1)
    [y, iter(i)] = LG_final(x(i),10e-10);
    for j=1:size(result' ,1)
        if abs(result(j) - y) > 10e-10
            nb = nb + 1;
        end
    end

    if nb == size(result', 1)
        p = p + 1;
        result(p) = y;
    end
    nb = 0;
end
```

Figure 45: function for calling Laguerre's function in interval [-100,100] with step every 0.1