Numerical Methods

Project B No. 61

Jakub Tomkiewicz
300183

Problem 1

Description of the problem

Finding all zeros of the function:

$$f(x) = 0.3 * \sin(x) - \ln(x + 1)$$

in the interval [2,12] using:

a) the Secant method
b) the Newton's method

Theoretical grounds

Solving a nonlinear equation

To find a root of nonlinear function, first thing is to set an interval wherein root is located, such phase is called root bracketing. The simplest way to perform this operation is to create a graphical diagram and plot function, then intervals can be defined.

Having found intervals containing roots, interactive method can be initialised.

Order of convergence of an iterative method is defined by the largest number $p \geq 1$ such that:

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

where: α - root k - convergence factor

The bigger is the order of convergence, the better is the convergence method. If $p = 1$, method is convergent linearly.=, when $p = 2$, the convergence is quadratic.

Secant method

In this method, a secant line joining two last obtained points is created. If this two points we mark as x_{n-1} and x_n, then a new point can be determined by formula:

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

In the secant method order of convergence is $p \approx 1.618$

Secant method is locally convergent, a lack of convergence may happen, if the initial interval of the root is not sufficiently small.

Newton's method

This method, called also the tangent method, works 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:

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

Next point, x_{n+1}, proceeds as a root of the obtained function:

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

Such iteration formula can be achieved:

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

Newton's method, as Secant method, is locally convergent. The convergence is quadratic, with the order $p = 2$, which means that it is usually very fast.

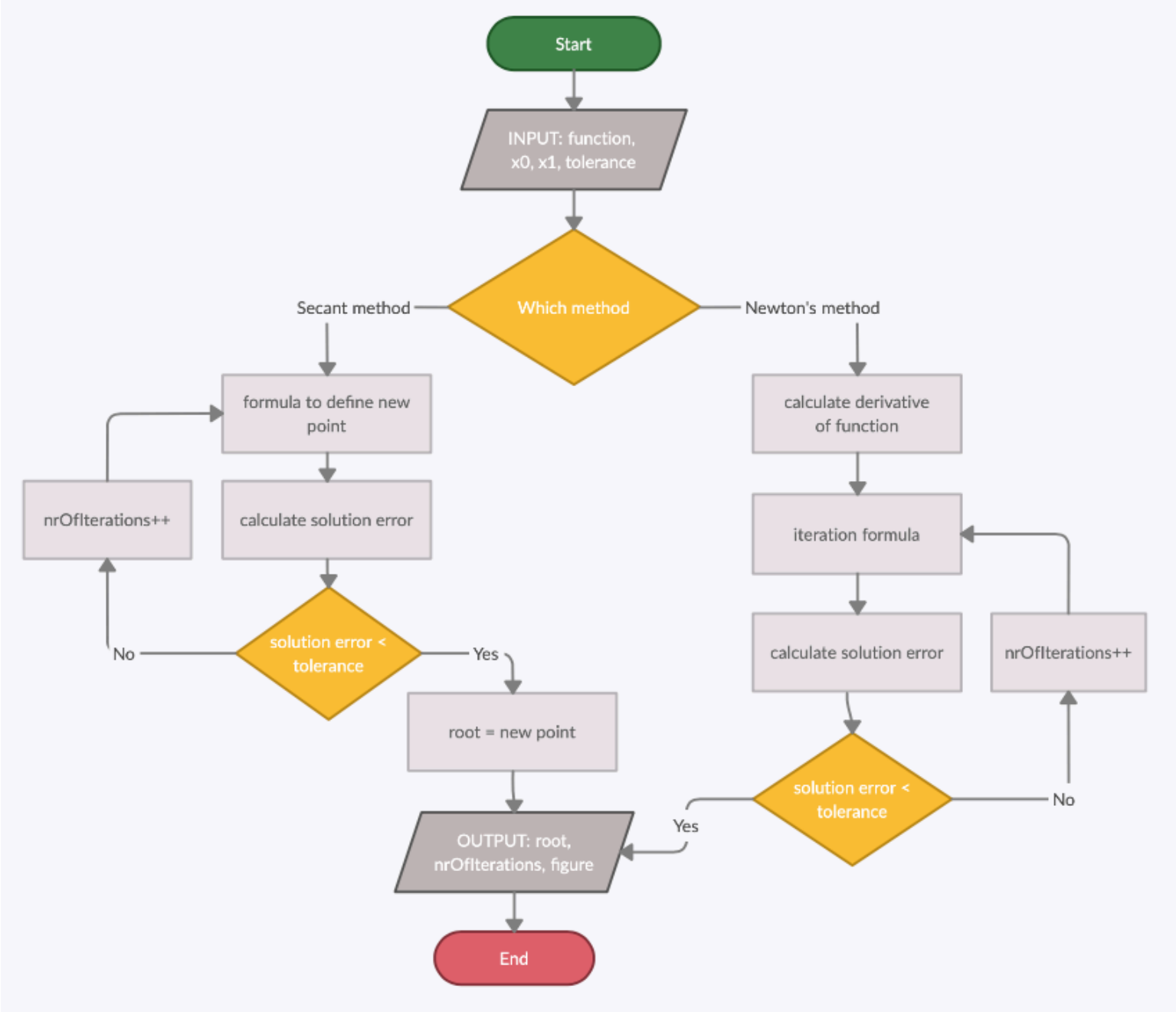
Method is especially effective when function derivative at the root is sufficiently far from zero. It is not advised to use when derivative is close to zero, then it is very sensible to numerical errors when close to root.

Problem analysis

In the beginning we need to know how many roots are in our function, so it needs to be plotted on the diagram. Knowing intervals wherein roots are located we are able to initiate at first Secant method and then Newton's method. Both methods will be implemented as functions that will take start and end of the interval as arguments. Having roots, comparison of needed iterations to obtain them will be made.

Both methods will be performed using **tolerance** set to 1e-9 and **max number of iterations** equal to 50.

Algorithm applied



Solution

Legend:

red star - root

cyan square - first subinterval

magenta square - second subinterval

blue square - third subinterval

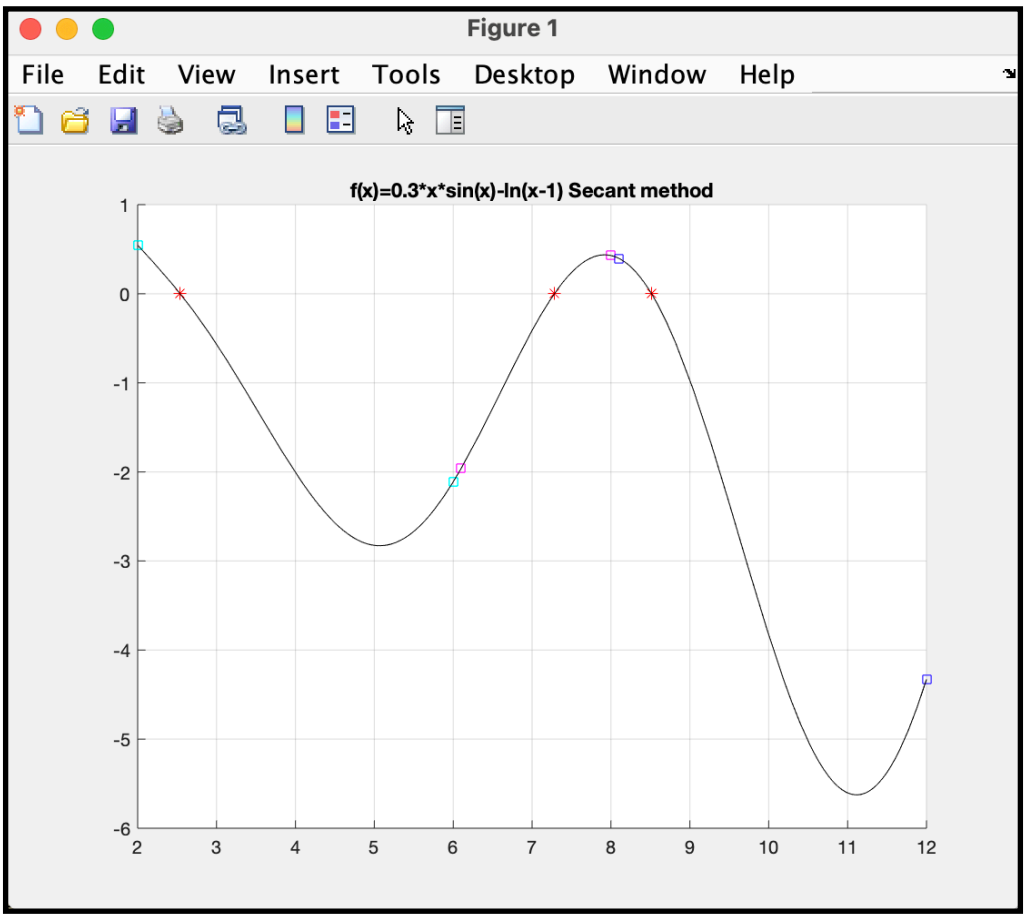


Figure 1
File Edit View Insert Tools Desktop Window Help
f(x)=0.3*x*sin(x)-ln(x-1) Secant method
1
0
-1
-2
-3
-4
-5
-6
2 3 4 5 6 7 8 9 10 11 12

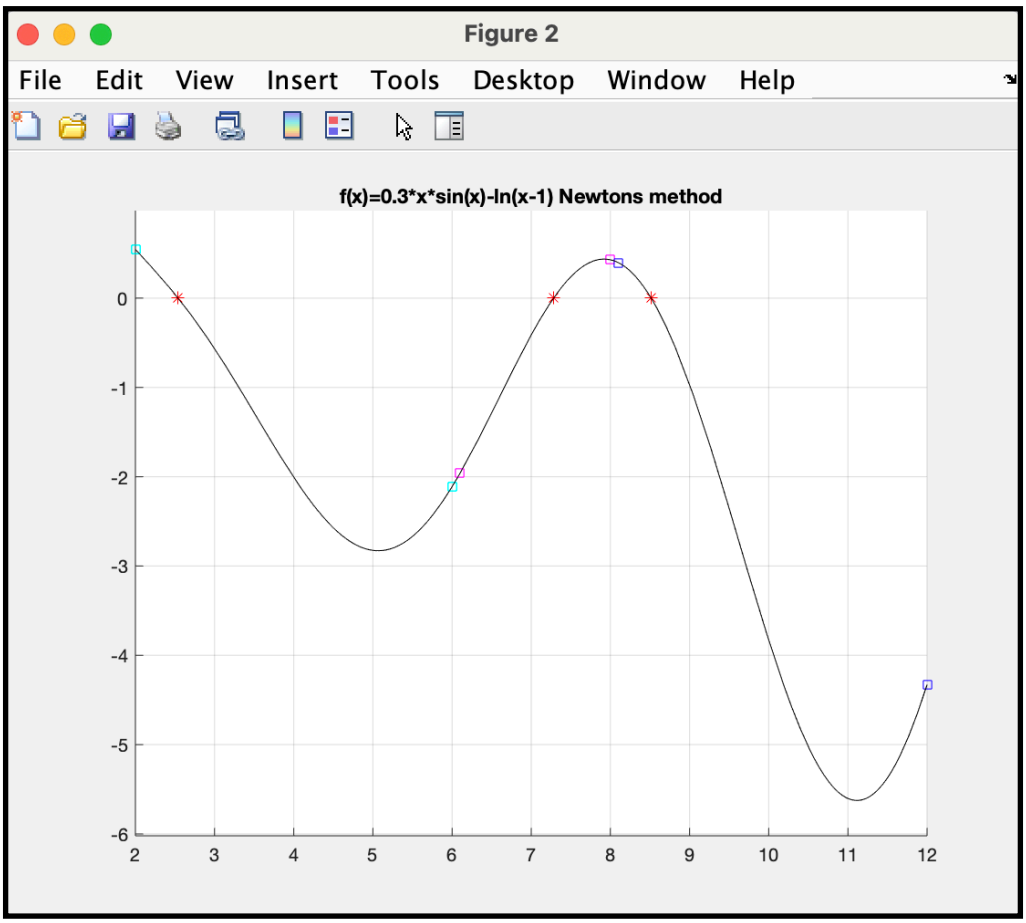


Figure 2
File Edit View Insert Tools Desktop Window Help
f(x)=0.3*x*sin(x)-ln(x-1) Newtons method
0
-1
-2
-3
-4
-5
-6
2 3 4 5 6 7 8 9 10 11 12

Secant method results with successive iteration points

First root:

Iteration	Argument	Function value
1	2.8210	-0.3328
2	2.2267	0.3251
3	2.5204	0.0211
4	2.5408	-0.0014
5	2.5395	6.0303e-06
6	2.5395	1.7188e-9
7	**2.5395**	**-1.6653e-15**

Second root:

Iteration	Argument	Function value
1	7.6595	0.3585
2	5.9176	-2.2275
3	7.4180	0.1581
4	7.3186	0.0448
5	7.2792	-0.0044
6	7.2827	9.4130e-5
7	7.2826	1.8654e-7
8	**7.2826**	**-8.0429e-12**

Third root:

Iteration	Argument	Function value
1	8.4274	0.1187
2	8.5227	-0.0118
3	8.5141	8.9325e-4
4	8.5147	5.7541e-6
5	8.5147	-2.8425e-9
6	**8.5147**	**9.7700e-15**

Newton's method results with successive iteration points

First root:

Iteration	Argument	Function value
1	2.5077	0.0350
2	2.5398	-2.5412e-4
3	2.5395	-1.3267e-9
4	**2.5395**	**-2.2204e-16**

Second root:

Iteration	Argument	Function value
1	7.2624	-0.0261
2	7.2824	-2.9298e-4
3	7.2826	-3.9277e-8
4	**7.2826**	**-6.6613e-16**

Third root:

Iteration	Argument	Function value
1	8.5971	-0.1287
2	8.5194	-0.0070
3	8.5147	-2.6528e-5
4	**8.5147**	**-3.8874e-10**

Comparison Secant vs Newton's

	Secant method	Newton's method
First root	2.5395	2.5395
Needed iterations	7	4
Second root	7.2826	7.2826
Needed iterations	8	4
Third root	8.5147	8.5147
Needed iterations	6	4

Conclusions

To accurately compare both Secant and Newton's methods subinterval points needed to be exactly the same.

As said in the Theoretical Background, Newton's method proven to be faster, as it needed less operations to obtain the same correct outcome with the same certainty in a given interval. Effectiveness of Newton's method was approximately 1.75 better then Secant method.

What is worth noting, is the fact that all function derivatives at the roots were far from zero, so Newton's method was able to show it's potential. If derivatives were closer to zero, method might have some problems.

On the other hand, Secant method indeed was slower, but to obtain the result no derivative of the functions needed to be calculated.

Problem 2

Description of the problem

Finding all (real and complex) roots of polynomials

$$f(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 \qquad [a_4\ a_3\ a_2\ a_1\ a_0] = [2\ 3\ -6\ 4\ 7]$$

using the Muller's method implementing both MM1 and MM2 versions. Results needs to be compared. Using Newton's method real roots needs to be found and compared with the MM2 version of the Muller's method (using same initial points).

Theoretical grounds

Muller's method

Idea of this method is to approximate the polynomial locally in the neighbourhood of a root by a quadratic function. It is developed using quadratic interpolation based on three different points, which allows to treats it as a generalisation of secant method, where linear interpolation is based on two points.

The order of convergence of Muller's method is ≈ 1.84, which places it between secant method and Newton's method.

Two versions of Muller's method are discussed, named respectively MM1 and MM2.

MM1

Three points x_0, x_1, x_2 are considered with corresponding polynomial values $f(x_0), f(x_1), f(x_2)$. A quadratic function, parabola, passing through these points is constructed, roots of parabola are found and one root is selected for next approximation.

x_2 is assumed as actual approximation of the polynomial root and new variable is created:

$$z = x - x_2$$

$$z_0 = x_0 - x_2$$

$$z_1 = x_1 - x_2$$

Interpolating parabola defined in variable z is examined:

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

For three points we have:

$$az_0^2 + bz_0 + c = y(z_0) = f(x_0)$$

$$az_1^2 + bz_1 + c = y(z_1) = f(x_1)$$

$$c = y(0) = f(x_2)$$

The roots are given by:

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

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

The root with the smaller absolute value is chosen for the next iteration.

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

where:

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

$z_{min} = z_-$ in opposite case

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

MM2

This method is slightly more effective numerically, as it is numerically more expensive to measure. It calculates values of polynomial of its first and second derivatives at one point only.

It follows directly parabola:

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

And we have:

$$z = x - x_k$$

That at the point $z = 0$:

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

To find the roots formula is used:

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

Root with smaller value is chosen:

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

where z_{min} is selected from $\{z_+, z_-\}$ in identical way as in MM1.

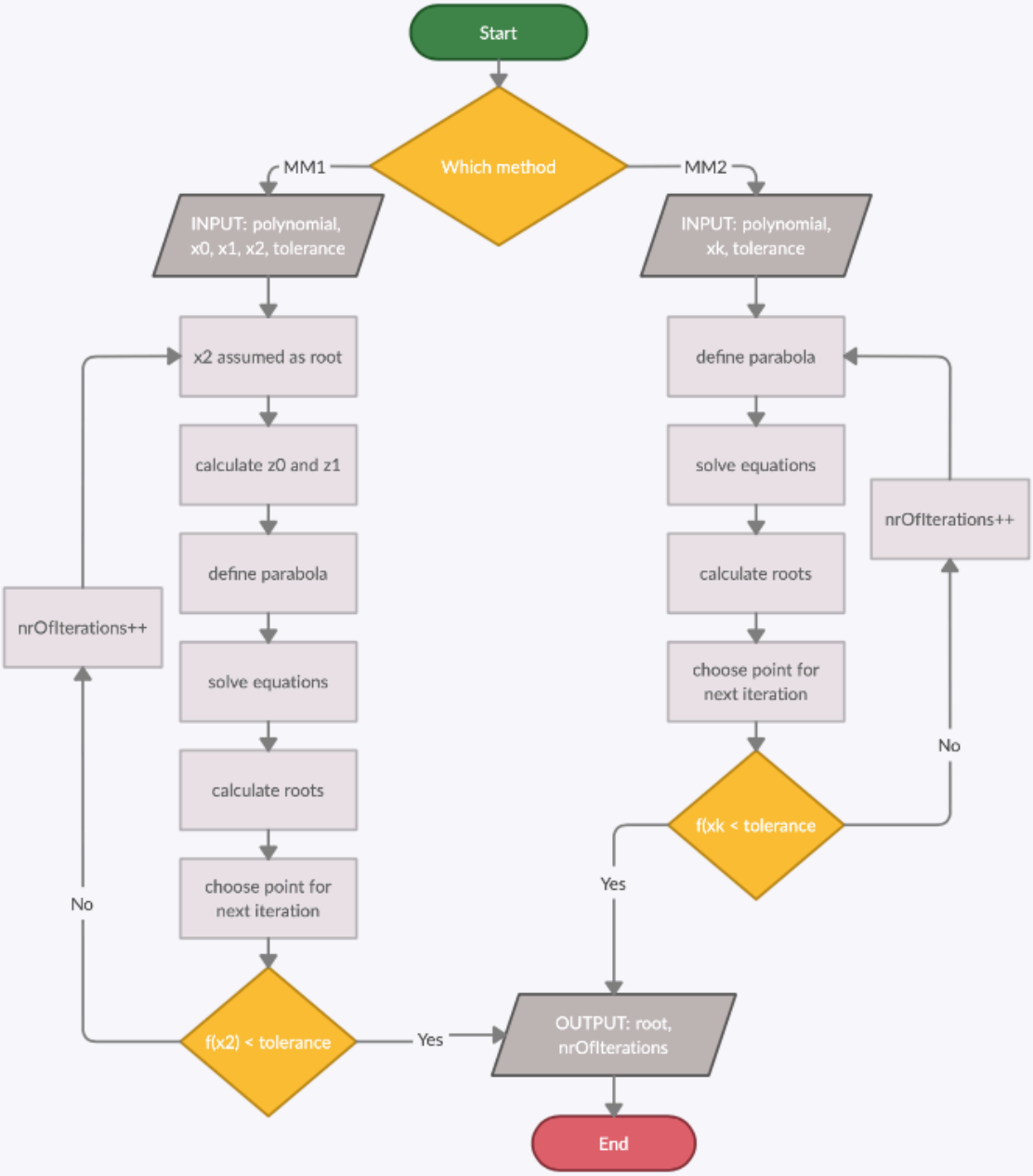
Problem analysis

As in Problem 1 the first thing is to find and set intervals wherein roots are located, but this time we need to remember that we are looking for both real and imaginary numbers. In this example both methods will be implemented as functions too, this time however polynomial will be send as argument.

What is new in Problem 2 is that we are not only comparing MM1 and MM2 methods, but Newton's method from earlier task will be set side by side additionally.

Methods will be performed using the same **tolerance** set to 1e-9 and **max number of iterations** equal to 50.

Algorithm applied



Solution

Legend:

red star - root

blue square - points x_0, x_1, x_2 in MM1 and x_k in MM2

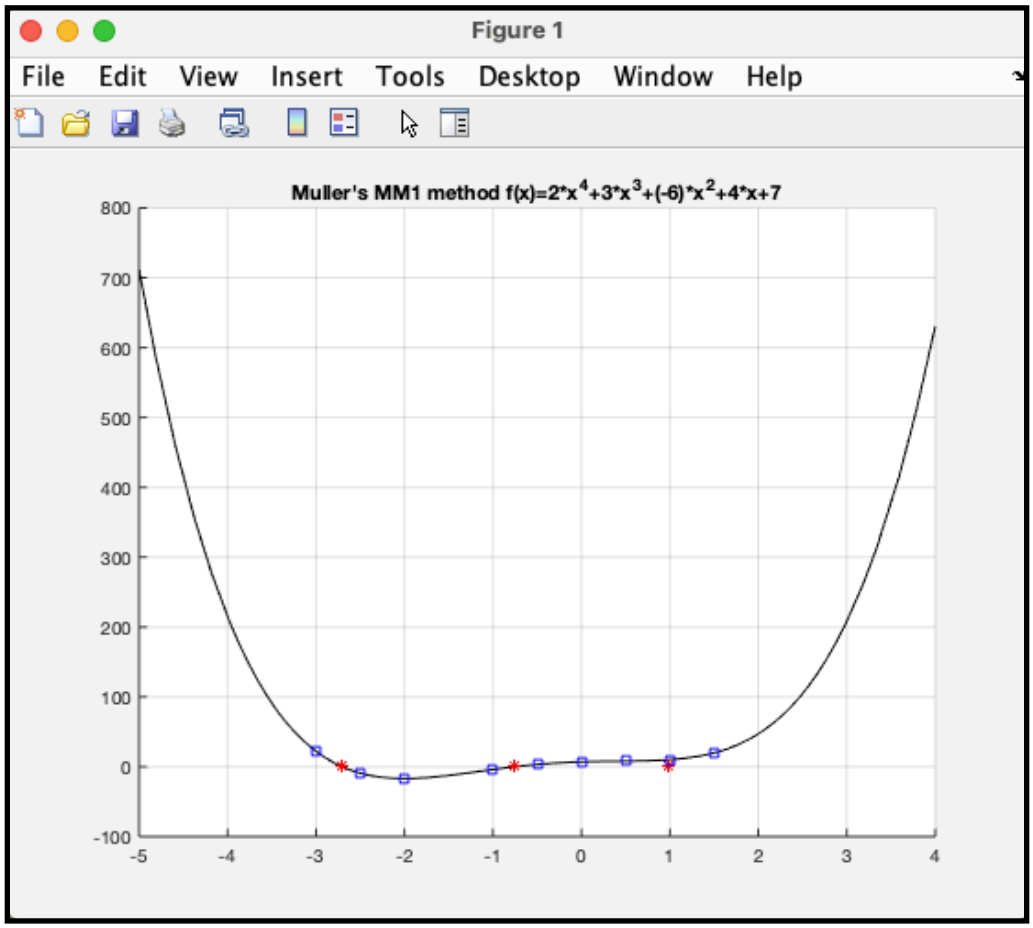


Figure 1
File Edit View Insert Tools Desktop Window Help
Muller's MM1 method f(x)=2*x^4+3*x^3+(-6)*x^2+4*x+7
800
700
600
500
400
300
200
100
0
-100
-5 -4 -3 -2 -1 0 1 2 3 4

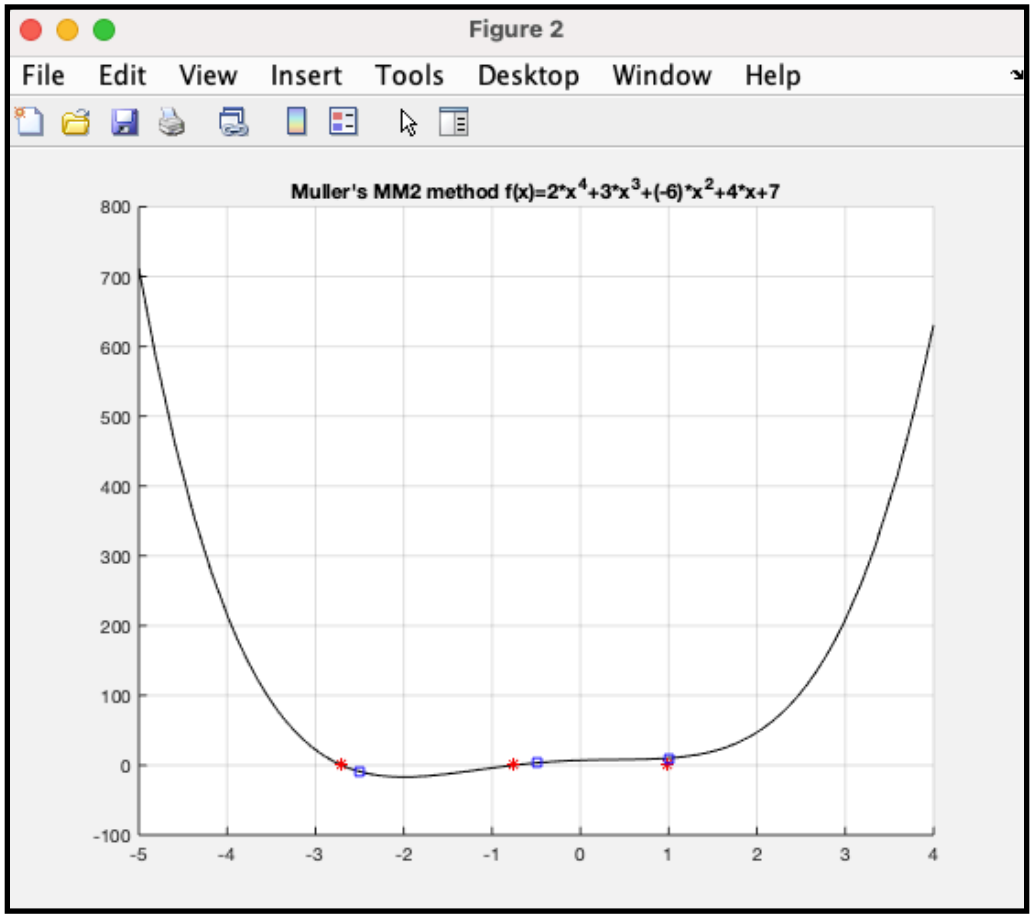


Figure 2
File Edit View Insert Tools Desktop Window Help
Muller's MM2 method f(x)=2*x^4+3*x^3+(-6)*x^2+4*x+7
800
700
600
500
400
300
200
100
0
-100
-5 -4 -3 -2 -1 0 1 2 3 4

MM1 results with successive iteration points

First root:

Iteration	Argument	Function value
1	-2.6925	-0.7127
2	-2.7051	-0.0144
3	-2.7054	1.2340e-05
4	-2.7054	-1.3522e-11
5	**-2.7054**	**-5.3291e-15**

Second root:

Iteration	Argument	Function value
1	-0.7543	-0.0711
2	-0.7494	9.2029e-04
3	-0.7495	-1.8949e-07
4	-0.7495	1.2434e-14
5	**-0.7495**	**0**

Third root:

Iteration	Argument	Function value
1	0.7058 - 0.7716i	5.3252 + 1.7908i
2	0.9989 - 0.9787i	-2.5079 + 1.5290i
3	0.9774 - 0.8687i	0.1825 - 0.1515i
4	0.9774 - 0.8778i	0.0025 - 0.0023i
7	**0.9774 - 0.8780i**	**-3.5527e-15 - 4.4409e-16i**

Fourth root:

Iteration	Argument	Function value
1	0.7058 + 0.7716i	5.3252 - 1.7908i
2	0.9989 + 0.9787i	-2.5079 - 1.5290i
3	0.9774 + 0.8687i	0.1825 + 0.1515i
4	0.9774 + 0.8778i	0.0025 + 0.0023i
7	**0.9774 + 0.8780i**	**-3.5527e-15 + 4.4409e-16i**

MM2 results with successive iteration points

First root:

Iteration	Argument	Function value
1	-2.7082	0.1572
2	-2.7054	-4.0731e-07
3	-2.7054	-5.3291e-15
4	**-2.7054**	**-5.3291e-15**

Second root:

Iteration	Argument	Function value
1	-0.7479	0.0228
2	-0.7495	1.1196e-08
3	-0.7495	0
4	**-0.7495**	**-8.8818e-16**

Third root:

Iteration	Argument	Function value
1	0.8624 + 0.8586i	2.2257 - 1.5639i
2	0.9777 + 0.8772i	0.0092 + 0.0172i
3	0.9774 + 0.8780i	5.5710e-09 + 7.3296e-10i
4	0.9774 + 0.8780i	1.7764e-15 + 1.7764e-15i
5	**0.9774 + 0.8780i**	**1.7764e-15 + 1.7764e-15i**

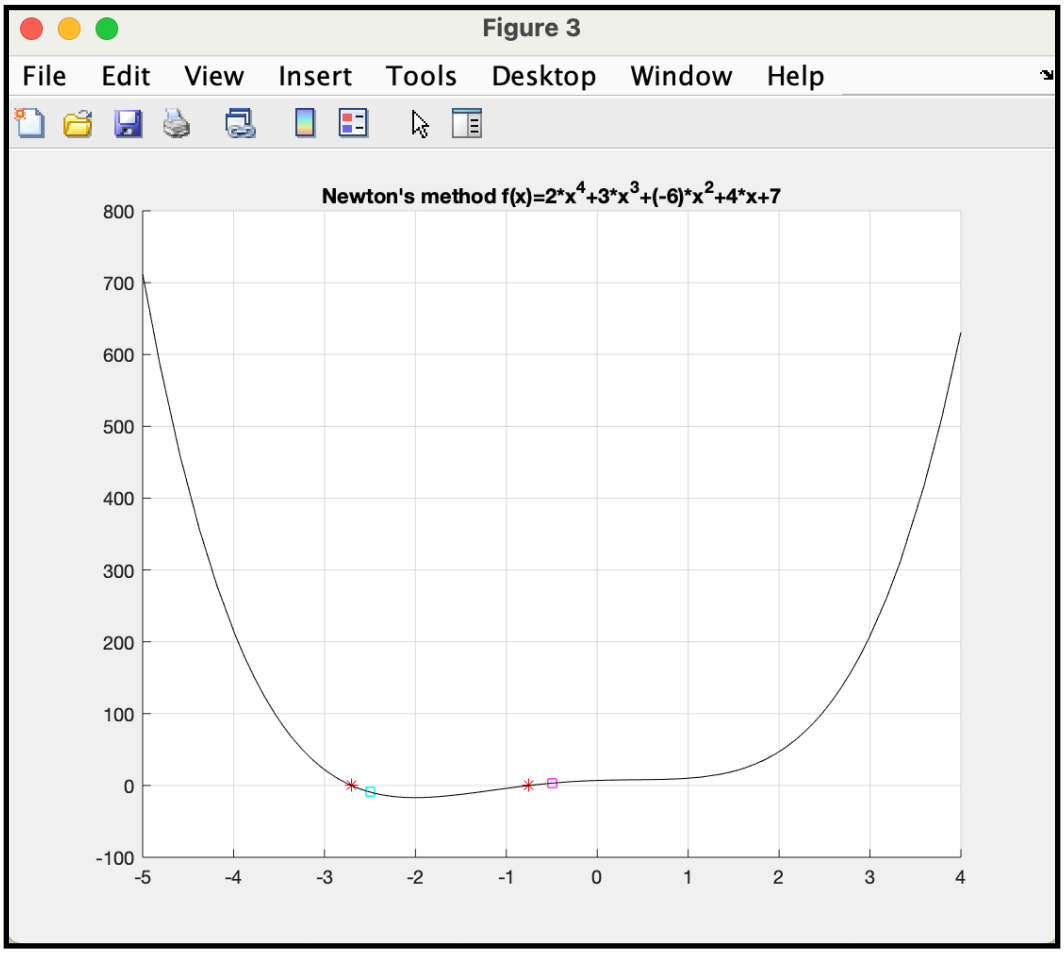
Fourth root:

Iteration	Argument	Function value
1	0.8624 - 0.8586i	2.2257 + 1.5639i
2	0.9777 - 0.8772i	0.0092 - 0.0172i
3	0.9774 - 0.8780i	5.5710e-09 - 7.3296e-10i
4	0.9774 - 0.8780i	1.7764e-15 - 1.7764e-15i
5	**0.9774 - 0.8780i**	**1.7764e-15 - 1.7764e-15i**

Comparison MM1 vs MM2

	MM1	MM2
First root	-2.7054	-2.7054
Needed iterations	5	4
Second root	-0.7495	-0.7495
Needed iterations	5	4
Third root	0.9774 - 0.8780i	0.9774 + 0.8780i
Needed iterations	7	5
Fourth root	0.9774 + 0.8780i	0.9774 - 0.8780i
Needed iterations	7	5

Newton's method (only real roots)



First root:

Iteration	Argument	Function value
1	-2.7662	3.6255
2	-2.7089	0.1965
3	-2.7054	6.9726e-04
4	**-2.7054**	**8.8880e-09**

Second root:

Iteration	Argument	Function value
1	-0.7889	-0.5879
2	-0.7501	-0.0087
3	-0.7495	-2.0961e-06
4	**-0.7495**	**-1.2168e-13**

Comparison MM2 vs Newton's method

	MM2	Newton's
First root	-2.7054	-2.7054
Needed iterations	5	4
Second root	-0.7495	-0.7495
Needed iterations	5	4

Conclusions

This time to accurately compare MM1, MM2 and Newton's methods initial point was exactly the same for MM2 and Newton's method, additionally it was one of points used for MM1. Newton's method was set to use only one initial point, this way comparing it to MM2 was honourable.

Comparison of MM1 and MM2 method does not show significant differences, MM2 was always faster, but only by one or two iterations. For sure MM2 is more comfortable to use, as it needs one initial point, not three as in MM1.

Newton's method set side by side with MM2 was faster, but as in previous comparison only slight differences were made. However, Newton's method is not able to find complex numbers, only 2 of 4 roots were found using it.

Problem 3

Description of the problem

Finding all (real and complex) roots of the polynomial $f(x)$ from Problem 2 using the Laguerre's method. Results needs to be compared with MM2 version of the Muller's method (using same initial points).

Theoretical grounds

In Laguerre's method we use following formula:

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

where n is the order of polynomial

Sign in the denominator is chosen as in the Muller's method, assuring larger absolute value.

The Laguerre's method is regarded as one of the best methods for polynomial root finding. It is complex, as it takes order of the polynomial, but it's accuracy and speed is indisputable.

In case of real roots Laguerre's method is globally convergent, it converges starting from any real initial point. For a complex root case numerical practise has shown good numerical properties.

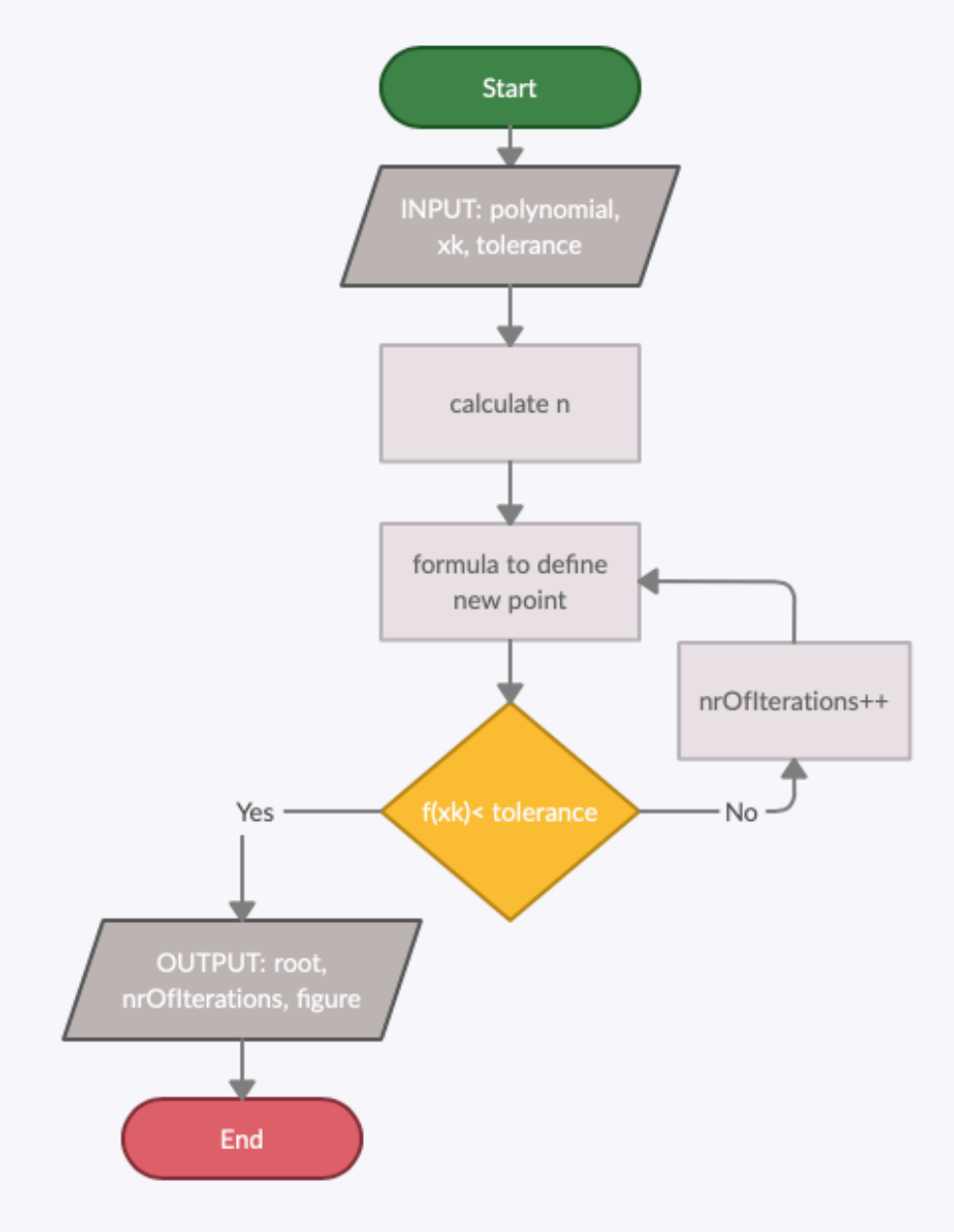
Problem analysis

This problem is the easiest, as we have already results of MM2 method and initial points, which for the reliability of the comparison, are exactly the same for both methods.

Only one function needs to be created, for Laguerre's method. It takes already existing polynomial and as mentioned earlier initial points. Moreover, part of Laguerre's method is the same as Muller's method, which means that part of already existing MATLAB code can be copied.

Methods will be performed using the same **tolerance** set to 1e-9 and **max number of iterations** equal to 50.

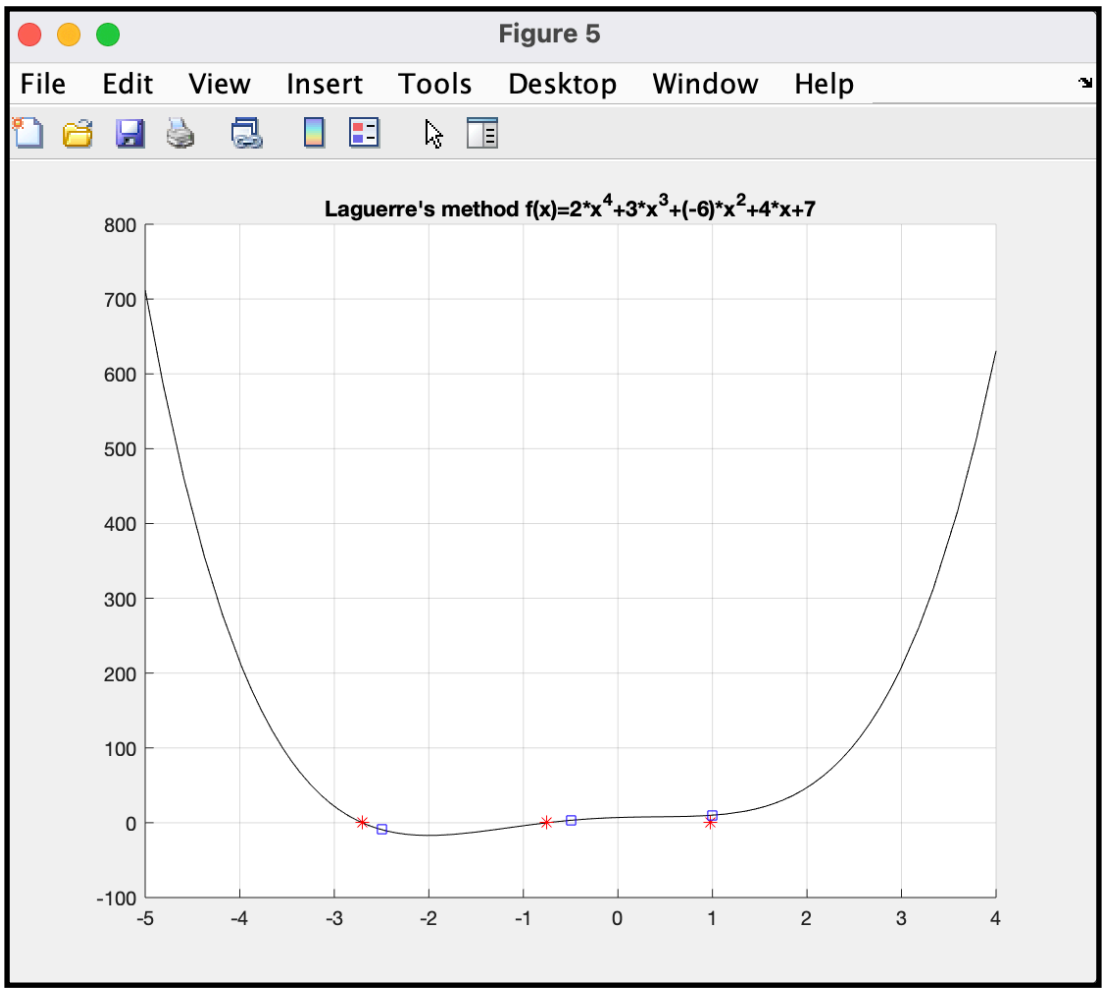
Algorithm applied



Solution

Legend:

red star - root
blue square - initial points



First root:

Iteration	Argument	Function value
1	-2.7052	-0.0115

Second root:

Iteration	Argument	Function value
1	-0.7463	0.0458
2	-0.7495	1.1606e-07
3	-0.7495	-8.8818e-16
4	-0.7495	0

Third root:

Iteration	Argument	Function value
1	0.9390 + 0.8449i	1.2219 - 0.1798i
2	0.9774 + 0.8780i	-2.9540e-04 + 5.6223e-05i
3	0.9774 + 0.8780i	2.6645e-15 - 1.7764e-15i

Fourth root:

Iteration	Argument	Function value
1	0.9390 - 0.8449i	1.2219 + 0.1798i
2	0.9774 - 0.8780i	-2.9540e-04 - 5.6223e-05i
3	0.9774 - 0.8780i	2.6645e-15 + 1.7764e-15i

Comparison MM2 vs Laguerre's method

	MM2	Laguerre's
First root	-2.7054	-2.7052
Needed iterations	4	1
Second root	-0.7495	-0.7495
Needed iterations	4	4
Third root	0.9774 + 0.8780i	0.9774 + 0.8780i
Needed iterations	5	3
Fourth root	0.9774 - 0.8780i	0.9774 - 0.8780i
Needed iterations	5	3

Conclusions

As it was quoted in Theoretical backgrounds comparison of MM2 and Laguerre's method shows clearly, Laguerre's effectiveness is indisputable when looking for real roots. It managed to improve significantly already very good results and strengthen it's position as good for complex roots.

In defense of MM2 method it can be said that it is more certain when it comes to finding complex roots, Laguerre's was faster, but MM2 is the question-less method.

MATLAB Code

```
% close all windows and figures, clear command window
close all;
clear all;
clc;

%%%%%%%%%% ENUME %%%%%%%%%%
%%%%% PROJECT B NO.61 %%%%%
% JAKUB TOMKIEWICZ 300183 %

%%%%%%%% PROBLEM 1 %%%%%%%%
f = @(x)0.3*x*sin(x)-log(x-1); % my function

% interval [2 12] divided into 3 parts
[iter_Sacant1, root_Sacant1] = SecantMethod(f,50,1e-9,2,6,1); % [2 6]
[iter_Sacant2, root_Sacant2] = SecantMethod(f,50,1e-9,6.1,8,2); % [6 8]
[iter_Sacant3, root_Sacant3] = SecantMethod(f,50,1e-9,8.1,12,3); % [8 12]

hold on;
fplot(f,[2 12],'k'); % plot function in interval
title("f(x)=0.3*x*sin(x)-ln(x-1) Secant method");
hold off;

% print nr of iterations and root
iter_Sacant1
root_Sacant1

iter_Sacant2
root_Sacant2

iter_Sacant3
root_Sacant3

figure(2);
% interval [2 12] divided into 3 parts
[iter_Newton1, root_Newton1] = NewtonsMethod(f,50,1e-9,2,6,1,1); % [2 6]
[iter_Newton2, root_Newton2] = NewtonsMethod(f,50,1e-9,6.1,8,2,1); % [6 8]
[iter_Newton3, root_Newton3] = NewtonsMethod(f,50,1e-9,8.1,12,3,1); % [8 12]

hold on;
fplot(f,[2 12],'k'); % plot function in interval
title("f(x)=0.3*x*sin(x)-ln(x-1) Newtons method");
hold off;

% print nr of iterations and root
iter_Newton1
root_Newton1

iter_Newton2
root_Newton2

iter_Newton3
root_Newton3
```

```
%%%%%%%% PROBLEM 2 %%%%%%%%
f = @(x)2*x^4 + 3*x^3 + (-6)*x^2 + 4*x + 7;
p = [2 3 -6 4 7];

% MM1 method
figure(1);
hold on;
grid on;
fplot(f,[-5 4],'k');

% first root
x0 = -2;
x1 = -2.5;
x2 = -3;
[nrIterationsMM1_1, rootMM1_1] = MM1method(p,x0,x1,x2,50,1e-9);
plot(rootMM1_1,f(rootMM1_1), 'r*');
plot(x0,f(x0),'bs');
plot(x1,f(x1),'bs');
plot(x2,f(x2),'bs');

% second root
x0 = 0;
x1 = -0.5;
x2 = -1;
[nrIterationsMM1_2, rootMM1_2] = MM1method(p,x0,x1,x2,50,1e-9);
plot(rootMM1_2,f(rootMM1_2), 'r*');
plot(x0,f(x0),'bs');
plot(x1,f(x1),'bs');
plot(x2,f(x2),'bs');

% third root
x0 = 0.5;
x1 = 1-0.25i;
x2 = 1.5;
[nrIterationsMM1_3, rootMM1_3] = MM1method(p,x0,x1,x2,50,1e-9);
plot(rootMM1_3,f(rootMM1_3), 'r*');
plot(x0,f(x0),'bs');
plot(x1,f(x1),'bs');
plot(x2,f(x2),'bs');

% fourth root
x0 = 0.5;
x1 = 1+0.25i;
x2 = 1.5;
[nrIterationsMM1_4, rootMM1_4] = MM1method(p,x0,x1,x2,50,1e-9);
plot(rootMM1_4,f(rootMM1_4), 'r*');
plot(x0,f(x0),'bs');
plot(x1,f(x1),'bs');
plot(x2,f(x2),'bs');

title("Muller's MM1 method f(x)=2*x^4+3*x^3+(-6)*x^2+4*x+7");
hold off;
```

```
% print nr of iterations and root
nrIterationsMM1_1
rootMM1_1

nrIterationsMM1_2
rootMM1_2

nrIterationsMM1_3
rootMM1_3

nrIterationsMM1_4
rootMM1_4

% MM2 method
figure(3);
hold on;
grid on;
fplot(f,[-5 4],'k');

% first root
xk = -2.5;
[nrIterationsMM2_1, rootMM2_1] = MM2method(p,xk,50,1e-9);
plot(rootMM2_1,f(rootMM2_1), 'r*');
plot(xk,f(xk),'bs');

% second root
xk = -0.5;
[nrIterationsMM2_2, rootMM2_2] = MM2method(p,xk,50,1e-9);
plot(rootMM2_2,f(rootMM2_2), 'r*');
plot(xk,f(xk),'bs');

% third root
xk = 1+0.25i;
[nrIterationsMM2_3, rootMM2_3] = MM2method(p,xk,50,1e-9);
plot(rootMM2_3,f(rootMM2_3), 'r*');
plot(xk,f(xk),'bs');

% fourth root
xk = 1-0.25i;
[nrIterationsMM2_4, rootMM2_4] = MM2method(p,xk,50,1e-9);
plot(rootMM2_4,f(rootMM2_4), 'r*');
plot(xk,f(xk),'bs');

% print nr of iterations and root
nrIterationsMM2_1
rootMM2_1

nrIterationsMM2_2
rootMM2_2

nrIterationsMM2_3
rootMM2_3

nrIterationsMM2_4
rootMM2_4
```

```
title("Muller's MM2 method f(x)=2*x^4+3*x^3+(-6)*x^2+4*x+7");
hold off;

% Newton's method
figure(4);
[iter_Newton1, root_Newton1] = NewtonsMethod(f,50,1e-9,-2.5,-2.5,1,2);
[iter_Newton2, root_Newton2] = NewtonsMethod(f,50,1e-9,-0.5,-0.5,2,2);

hold on;
fplot(f,[-5 4],'k');
title("Newton's method f(x)=2*x^4+3*x^3+(-6)*x^2+4*x+7");
hold off;


%%%%%%%% PROBLEM 3 %%%%%%%%
figure(5);
hold on;
grid on;
fplot(f,[-5 4],'k');

% first root
xk = -2.5;
[iter_Laguerre_1, root_Laguerre_1] = LaguerresMethod(p,xk,50,1e-9);
plot(root_Laguerre_1,f(root_Laguerre_1), 'r*');
plot(xk,f(xk),'bs');

% second root
xk = -0.5;
[iter_Laguerre_2, root_Laguerre_2] = LaguerresMethod(p,xk,50,1e-9);
plot(root_Laguerre_2,f(root_Laguerre_2), 'r*');
plot(xk,f(xk),'bs');

% third root
xk = 1+0.25i;
[iter_Laguerre_3, root_Laguerre_3] = LaguerresMethod(p,xk,50,1e-9);
plot(root_Laguerre_3,f(root_Laguerre_3), 'r*');
plot(xk,f(xk),'bs');

% fourth root
xk = 1-0.25i;
[iter_Laguerre_4, root_Laguerre_4] = LaguerresMethod(p,xk,50,1e-9);
plot(root_Laguerre_4,f(root_Laguerre_4), 'r*');
plot(xk,f(xk),'bs');

% print nr of iterations and root
root_Laguerre_1
iter_Laguerre_1

root_Laguerre_2
iter_Laguerre_2

root_Laguerre_3
iter_Laguerre_3

root_Laguerre_4
iter_Laguerre_4
```

```
title("Laguerre's method f(x)=2*x^4+3*x^3+(-6)*x^2+4*x+7");
hold off;


%%%%%%%% FUNCTIONS %%%%%%%%

% Problem 1 Secant Method
function [nrIterations, root] = SecantMethod(f,imax,assumedTolerance,x0,x1,part)

    nrIterations = 1;

    hold on;
    grid on;

    % for every part of interval use different colors to define intervals
    if(part == 1)
       plot(x0, f(x0), 'cs');
       plot(x1, f(x1), 'cs');
    end

    if(part == 2)
       plot(x0, f(x0), 'ms');
       plot(x1, f(x1), 'ms');
    end

    if(part == 3)
       plot(x0, f(x0), 'bs');
       plot(x1, f(x1), 'bs');
    end

    for i=1:imax % max nr of iterations

        % x2 is xn+1, x1 is xn, x0 is xn-1
        x2=(x0*f(x1)-x1*f(x0))/(f(x1)-f(x0)); % formula to define new point

        x0 = x1;
        x1 = x2;

        solutionError = abs(x1 - x0); % calculate error

        root = x2;

        if solutionError < assumedTolerance % if error < tolerance STOP
            break;
        end

        nrIterations = nrIterations + 1;
    end

    plot(root, f(root), 'r*'); % plot root on graph
    hold off;
end
```

```
% Problem 1 Newtons Method
function [nrIterations, root] =
NewtonsMethod(f,imax,assumedTolerance,x0,x1,part,problem)

    nrIterations = 1;

    hold on;
    grid on;

    % for every part of interval use different colours
    if(part == 1)
       plot(x0, f(x0), 'cs');
       plot(x1, f(x1), 'cs');
    end

    if(part == 2)
       plot(x0, f(x0), 'ms');
       plot(x1, f(x1), 'ms');
    end

    if(part == 3)
       plot(x0, f(x0), 'bs');
       plot(x1, f(x1), 'bs');
    end

    root = (x0 + x1)/2; % initial point in middle of interval

    if problem == 1
    df = @(x)(3*sin(x))/10 - 1/(x - 1) + (3*x*cos(x))/10; % derivative of
function
    end

    if problem == 2
        df = @(x)8*x^3 + 9*x^2 - 12*x + 4; % derivative of function
    end

    for i=1:imax

        root2 = root; % remember old value
        root = root - (f(root)/df(root)); % iteration formula

        solutionError = abs(root - root2);

        if solutionError < assumedTolerance % if error < tolerance STOP
            break;
        end

        nrIterations = nrIterations + 1;
    end

    plot(root, f(root), 'r*');
end
```

```
% Problem 2 MM1 Method
function [nrIterations, root] = MM1method(p,x0,x1,x2,imax,assumedTolerance)

    nrIterations = 0;
    c = 1; % something bigger then tolerance to begin with

    while nrIterations < imax && abs(c) > assumedTolerance % c = f(x2)
        z0 = x0 - x2;
        z1 = x1 - x2;

        % polynomial evaluation
        c = polyval(p,x2);
        a = (polyval(p,x1)*z0 - c*z0 - polyval(p,x0)*z1 + c*z1)/(z1^2*z0 -
z0^2*z1);
        b = (polyval(p,x0) - c - a*z0^2)/z0;

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

        if abs(b + sqrt(delta)) >= abs(b - sqrt(delta)) % chosing root with
smaller value
            zmin = (-2*c)/(b + sqrt(delta));
        else
            zmin = (-2*c)/(b - sqrt(delta));
        end

        root = x2 + zmin;

        % looking for two points close to root
        lengthx0 = abs(x0 - root);
        lengthx1 = abs(x1 - root);
        lengthx2 = abs(x2 - root);

        % we assume that x2 is root of polynomial
        if lengthx2 >= lengthx1 && lengthx2 >= lengthx0
            x2 = root;
        end

        if lengthx1 >= lengthx2 && lengthx1 >= lengthx0
            x1 = x2;
            x2 = root;
        end

        if lengthx0 >= lengthx2 && lengthx0 >= lengthx1
            x0 = x2;
            x2 = root;
        end

        nrIterations = nrIterations+1;
    end
end
```

```
% Problem 2 MM2 Method
function [nrIterations, root] = MM2method(p,root,imax,assumedTolerance)

    nrIterations = 0;

    % derivatives
    dp = polyder(p);
    d2p = polyder(dp);

    c = 1; % something bigger then tolerance to begin with

    while nrIterations < imax && abs(c) > assumedTolerance % c = f(xk)

        c = polyval(p,root);
        b = polyval(dp,root);
        a = polyval(d2p,root)/2;
        delta = b^2-4*a*c;

        if abs(b + sqrt(delta)) >= abs(b - sqrt(delta)) % chosing root with
smaller value
            zmin = (-2*c)/(b + sqrt(delta));
        else
            zmin = (-2*c)/(b - sqrt(delta));
        end

        root = zmin + root;

        nrIterations = nrIterations+1;
    end
end

% Problem 3 Laguerres Method
function [nrIterations, root] = LaguerresMethod(p,root,imax,assumedTolerance)

    nrIterations = 0;

    % derivatives
    dp = polyder(p);
    d2p = polyder(dp);
    n = length(p);
    n = n-1;

    c = 1; % something bigger then tolerance to begin with

    while nrIterations < imax && c > assumedTolerance % c = f(xk)

        c = polyval(p,root);

        denominator1 = polyval(dp,root) +
sqrt((n-1)*((n-1)*(polyval(dp,root)^2)-n*polyval(p,root)*polyval(d2p,root)));
        denominator2 = polyval(dp,root) -
sqrt((n-1)*((n-1)*(polyval(dp,root)^2)-n*polyval(p,root)*polyval(d2p,root)));

        % choosing larger value of denominator
        if abs(denominator1) > abs(denominator2)
            denominator = denominator1;
```

```
        else
            denominator = denominator2;
        end

        % xk+1 = xk - fraction
        root = root - (n*polyval(p,root))/denominator;

        nrIterations = nrIterations+1;
    end
end
```

Sources of knowledge

„Numerical Methods” Piotr Tatjewski
MathWorks Help Center