Numerical methods

Project B

Aayush Gupta

ID - 309601

Task 1:

Question

Find all zeros of the function

$$f(x) = 1.1x*\sin(x) - \ln(x + 5) \quad (1)$$

in the interval [0, 10] using:

a) the secant method,

b) the Newton's method.

Theoretical background

This project is about root finding. Root-Finding problem is the problem of finding a root of the equation f(x) = 0, where f(x) is a function of a single variable. Not so many functions have an analytical expression for the root, that is why we need special algorithms for general root finding problem.

To find a root of the function, first we need to have an interval where in a root is located, this should call a root bracketing.

In order to do the process of root bracketing algorithmically, then checking values of the function of both ends of the interval is a core of the procedure-if a function f(x) Is continuous on a closed interval [a,b] and $f(a)\ f(b) < 0$, then at least one root of f(x) is located within [a,b].

If an interval [a=x_1,b=x_2] ,is not containing a root ($f(x_1)\ f(x_2) > 0$),then a reasonable way to proceed is to expandthe interval. Having found an interval containing a root we can initiate an iterative method. Iterative means that we startan initial approximation x_0 of the root α and successively improve this approximation,

$$xn- > \alpha, \text{ where } n- > \infty \quad (2)$$

There is two important aspects of an iterative method: **convergence** and **stopping criterion**.

Convergence

Generally, order of convergence of an iterative method is defined as the largest number p ≥ 1 such that

$$\lim_{n\to\infty} \frac{|x_{n+1} - a|}{|x_n - a|} = k > \infty \quad (3)$$

where the coefficient k is the convergence factor. The larger the order of convergence, the better the convergence of the algorithm.

Stopping criteria:

$$|f(c_k)| \le \epsilon \quad (4)$$

(The functional value is less than or equal to the tolerance).

Or

$$\frac{|c_{k-1} - c_k|}{|c_k|} \le \epsilon \quad (5)$$

(The relative change is less than or equal to the tolerance)

Or

The number of iterations k is greater than or equal to a predetermined number.

Secant method:

The secant method is defined by the recurrence relation

$$x_n = \frac{x_{n-2}f(x_{n-1}) - x_{n-1}f(x_{n-2})}{f(x_{n-1}) - f(x_{n-2})} \tag{6}$$

As can be seen from the recurrence relation, the secant method requires two initial values, x0 and x1, which should ideally be chosen to lie close to the root. Starting with initial values x_0 and x_1, we construct a line through the points (x_0, f(x_0)) and (x_1, f(x_1)) then we use our recursive formula for approximate the real solution. The algorithm is stopped when the accuracy is achieved.

Newton's method

The Newton's method, also called the tangent method, operates by approximation of the function f(x) by the first part of its expansion into a Taylor series at a current point x_n,

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

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

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

which leads to the iteration formula

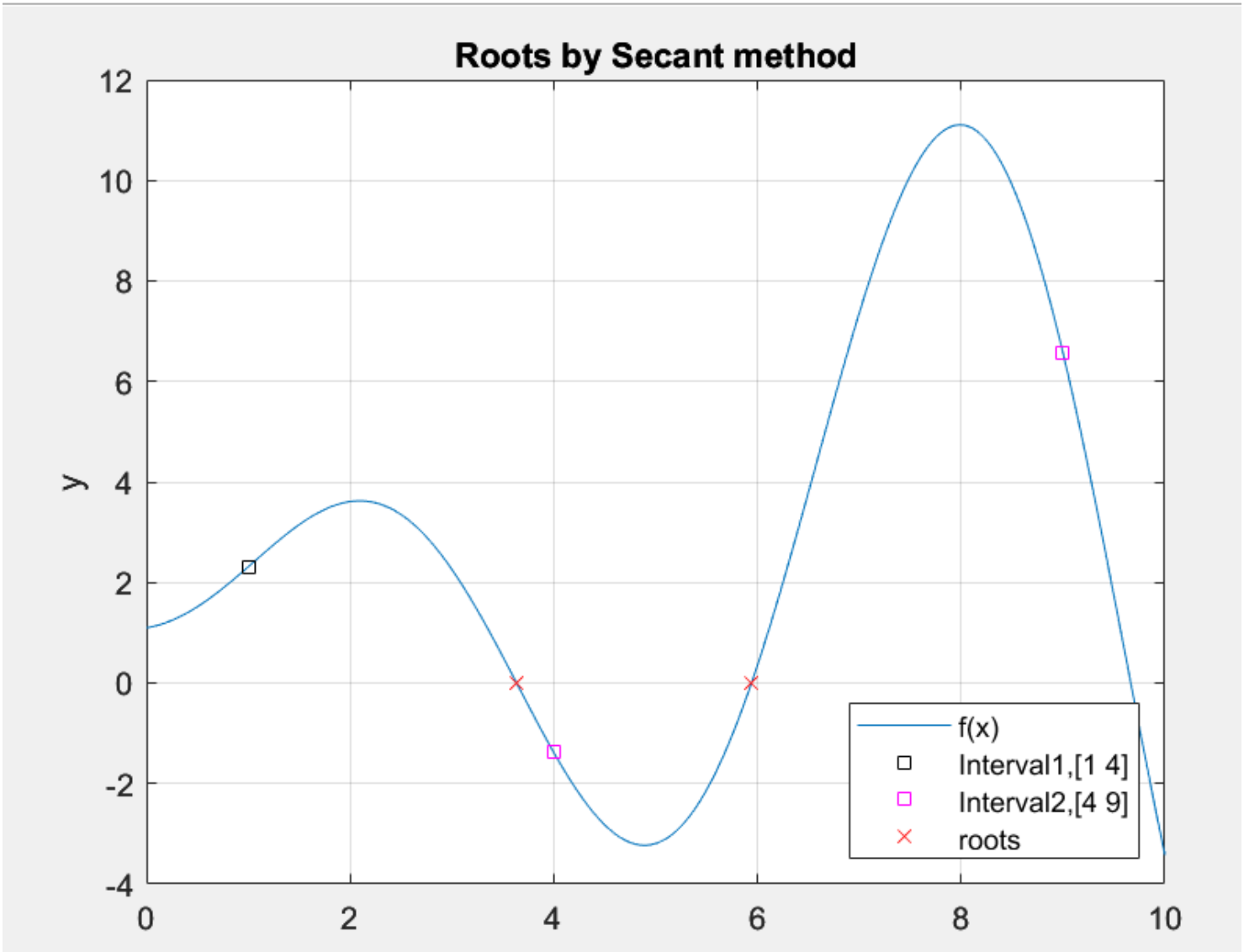
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{7}$$

Convergence of Newton's method

The Newton's method is locally convergent-if an initial point is too far from the root, then a divergence might occur. However, if the Newton's is convergent, then it is usually very fast, as the convergence is quadratic. In case $f'(\alpha)$ is close to zero, Newton's method is not recommended, as then it is very sensitive to numerical errors.

My solution:

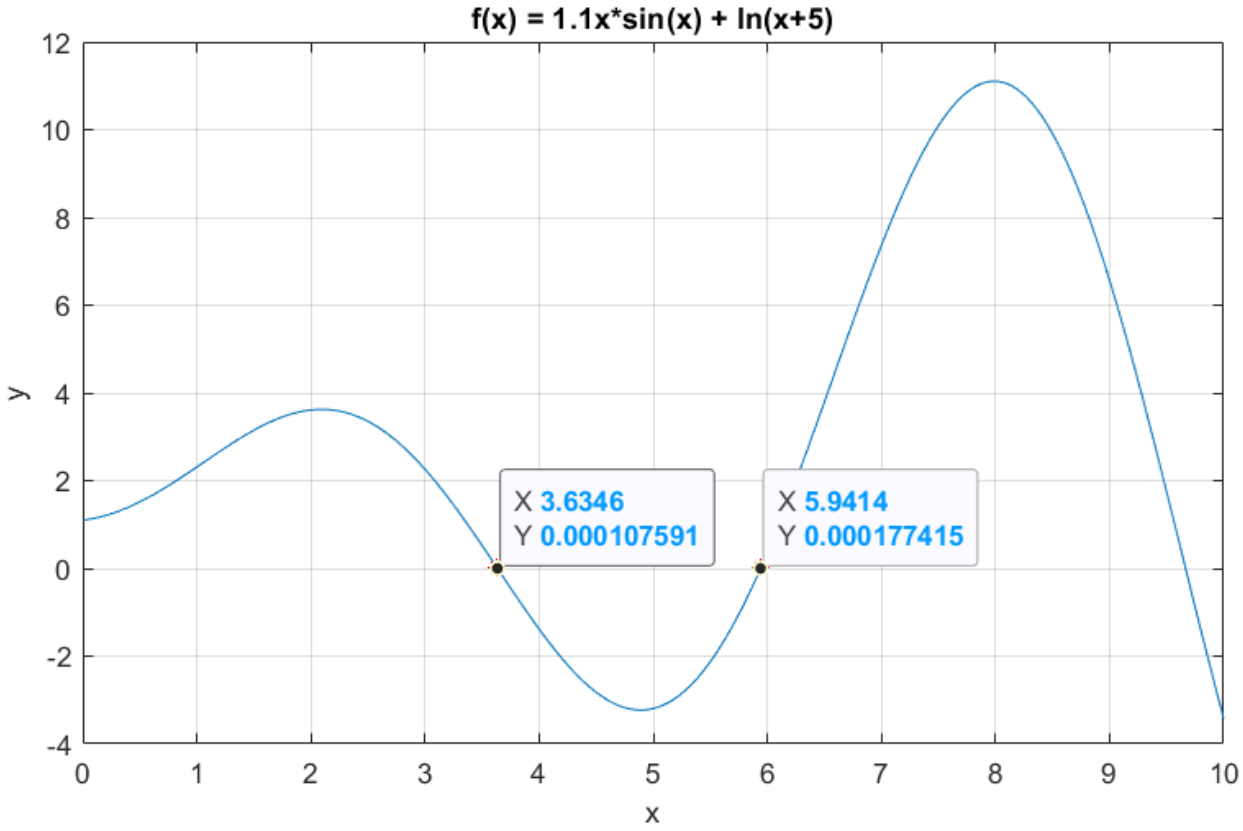
The graph of function in the integral of [0,10]



The graph shows that there is two roots: one root is around 3.6 and the second one is at around 5.9. Therefor we can divide the interval into two smaller intervals: [1, 4] and [4, 9].

$f(1) * f(4) < 0$

$f(4) * f(9) < 0$



The graph shows that there is two roots: one root is around 3.6 and the second one is at around 5.9.

method	Starting points	Number of iteration	result
Secant	(1,4)	9	3.5346
Secant	(4,9)	9	5.9414
Newton's	3.66	7	3.5346
Newton's	5.99	7	5.9414

Conclusion:

Thus the Newton's method is faster than the Secant method which is superliner, while the Newton's method is quadratic, in both cases we got accurate results although Secant method was two times slower than the other method.

Code:

```
clc
clear all
```

```
% this is our interval
x = 0:0.01:10
for k = 1:1001
f(k) = fun(x(k));
end
plot(x,f);
grid on;
results = Secant(1, 4)
results = Secant(4, 9)
results = Newton(3.66)
results = Newton(5.99)
hold on;
plot(3.6346, fun(3.6346), 'r*')
plot(5.9414, fun(5.9414), 'r*')
grid on;xlabel('x');ylabel('y')
title('f(x) = 1.1x*sin(x) + ln(x+5)')
function results = Newton(x1)
for i = 1:100
x = x1 - fun(x1) / dfun(x1);
results(i) = x;
err = abs(x - x1);
x1 = x;
if err < 10e-12
fprintf("We can find the root of this polynomial by: %d steps!", i);
break
end
end
end
function results = Secant(x0, x1)
for i = 1:100
x2 = (x0 * fun(x1) - x1 * fun(x0)) / (fun(x1) - fun(x0));
results(i) = x2;
x0 = x1;
x1 = x2;
err = abs(x1 - x0);
if err < 10e-12
fprintf("We can find the root of this polynomial by: %d steps!", i);
break
end
end
end
function [y] = fun(x)
y =  1.1*x*sin(x)+log(x+3);
end
function [y] = dfun(x)
 y = 1.1*sin(x)+1.1*x*cos(x)+(2/(x+3));
end
```

Task 2:

Find all (real and complex) roots of the polynomial

$f(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$, $[a_4, a_3, a_2, a_1, a_0] = [1 \quad 0.5 \quad 5 \quad -2 \quad -3]$

using the Muller's method implementing both the MM1 and MM2 versions. Compare the results.

Find also real roots using the Newton's method and compare the results with the MM2 version of the Muller's method (using the same initial points).

Theoretical background:

The idea of the Muller's method is to approximate the polynomial locally in a neighborhood of a root by a quadratic function. The method can be developed using a quadratic interpolation based on three different points.

There are two versions for Muller's method: MM1, MM2

MM1

Considering three points x_0, x_1, x_2 alongside their polynomial values $f(x_0)$, $f(x_1)$, and $f(x_2)$. A quadratic function is constructed that passes through these points, and then the roots of the parabola are found and one of the roots is selected for the next approximation of the solution. Assuming that x_2 is an actual approximation of the solution, let's introduce a new, incremental variable

$$z = x - x_2$$

and using the differences

$$z_0 = x_0 - x_2$$

$$z_1 = x_1 - x_2$$

The interpolating parabola defined in the variable is considered,

$$y(z) = az_2 + bz + c$$

Considering the three given points we get the following system of 2 equations which must be solved in order to find a and b

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

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

So the roots are given by

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

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

The root that has a smaller absolute value is chosen for the next iteration,

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

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

MM2:

This version of Muller's method 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. And we get formula for the roots:

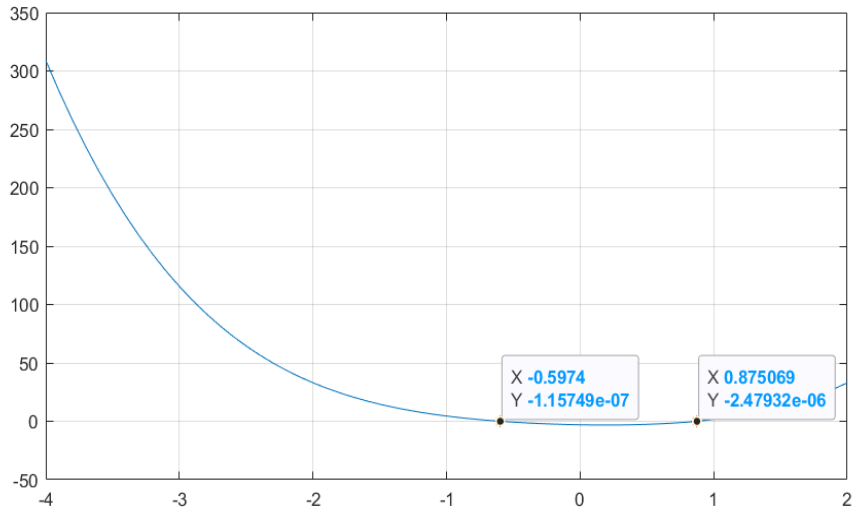
$$z\pm = \frac{-2f(x_k)}{f'(x_k \pm \sqrt{2f(x_k)f''(x_k)})}$$

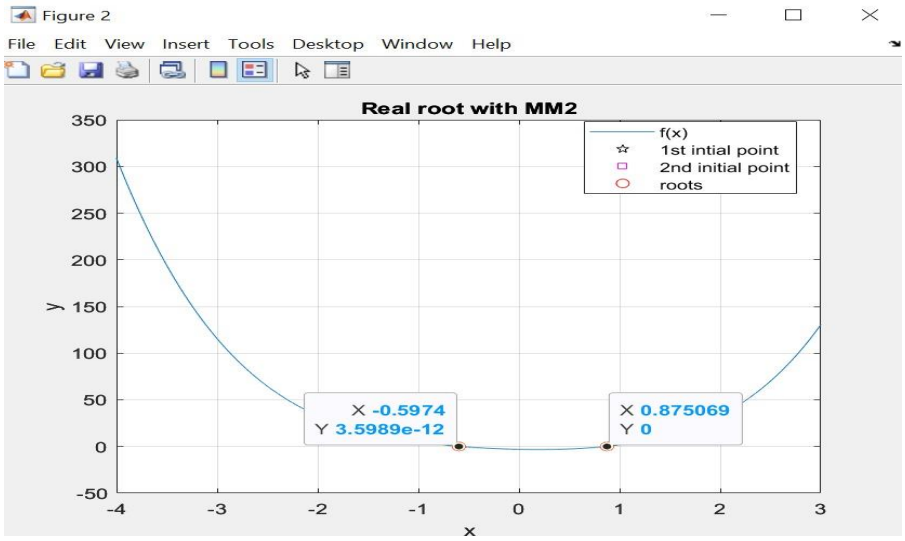
In MM1 and MM2 chosen points have to be near root because Müller's method converges locally.

MM2 realization is a little bit more effective than the MM1 realization. Müller's method lets finding complex roots.

After finding one root, the best option is to simplify the polynomial.
Polynomial deflation is a process of dividing the starting polynomial by (x-x0) where x0 is a root that was found earlier.
$f(x) = Q(x) * (x - x_0)$
$Q(x) = q_n * x^{n-1} + \cdots + q_2 * x + q_1$
where f(x) is an n^{th} degree polynomial
and Q(x) is a result of polynomial deflation

My solution:





The results are –

We can find the root of this polynomial by: 18 steps!
result =

Columns 1 through 7

1.1102 + 0.0000i 1.0383 + 0.0000i 0.8881 - 0.0869i 0.8892 - 0.0744i 0.8887 - 0.0241i 0.8898 - 0.0232i 0.8799 - 0.0167i

Columns 8 through 14

0.8800 - 0.0167i 0.8773 - 0.0176i 0.8773 - 0.0176i 0.8775 - 0.0169i 0.8775 - 0.0169i 0.8775 - 0.0168i 0.8775 - 0.0168i

Columns 15 through 18

```
  0.8774 - 0.0168i   0.8774 - 0.0168i   0.8774 - 0.0168i   0.8774 - 0.0168i

We can find the root of this polynomial by: 21 steps!
result =

  Columns 1 through 7

  0.1429 + 0.5151i   0.2273 + 0.3207i   0.1416 + 0.3191i   0.1748 + 0.3300i   0.1840 + 0.3190i   0.1847 + 0.3265i   0.1833 + 0.3255i

  Columns 8 through 14

  0.1848 + 0.3251i   0.1847 + 0.3252i   0.1846 + 0.3248i   0.1846 + 0.3248i   0.1847 + 0.3247i   0.1847 + 0.3247i   0.1847 + 0.3247i

  Columns 15 through 21

  0.1847 + 0.3247i   0.1847 + 0.3247i   0.1847 + 0.3247i   0.1847 + 0.3247i   0.1847 + 0.3247i   0.1847 + 0.3247i   0.1847 + 0.3247i

We can find the root of this polynomial by: 5 steps!
result =

  -0.9729   -0.6430   -0.5982   -0.5974   -0.5974

We can find the root of this polynomial by: 4 steps!
result =

   0.9198    0.8761    0.8751    0.8751

We can find the root of this polynomial by: 4 steps!
result =

  0.8810 - 0.0069i   0.8751 - 0.0000i   0.8751 + 0.0000i   0.8751 - 0.0000i

We can find the root of this polynomial by: 4 steps!
result =

  0.8810 + 0.0069i   0.8751 + 0.0000i   0.8751 - 0.0000i   0.8751 + 0.0000
>>
```

Conclusion:

For all methods we were able to get correct results but with different speeds; The MM2 was the fastest and then then after it comes the newton method and the slowest was MM1. When it comes to finding complex numbers, Newton's and MM2 have same number of iterations.

Code:

```
clc
clear all
v = [1 0.5 5 -2 -3];
x = -4:0.1:2;
for i = 1:61
f(i) = p(x(i));
end
plot(x, f);
grid on;
hold on;
plot(-0.5974, p(-0.5974),'r*');
plot(0.875069 ,p(0.875069),'r*');
% MM1
result = MM1(0, 1, 1.5)
result = MM1(-2, -1, 0)
% MM2
result = MM2(v, -1.8)
result = MM2(v, 1.2)
result = MM2(v, 1 - 0.06i)
result = MM2(v, 1 + 0.06i)
function result = MM1(x0, x1, x2)
p0 = p(x0);
p1 = p(x1);
for i = 1:50
p2 = p(x2);
z1 = x1 - x0;
z2 = x2 - x1;
f1 = (p1 - p0) / z1;
f2 = (p2 - p1) / z2;
a = (f2 - f1) / (x2 - x0);
b = f2 + z2 * a;
d = sqrt(b * b + 4 * p2 * a);
if abs(b - d) < abs(b + d)
e = b + d;
else
e = b - d;
end
z = -2 * p2 / e;
x0 = x1;
x1 = x2;
x2 = x2 + z;
result(i) = x2;
if abs(z) < 10e-10
fprintf("We can find the root of this polynomial by: %d steps!", i);
return;
end
po = p1;
p1 = p2;
end
end
function result = MM2(v, x)
c = p(x);
polder = polyder(v);
polder2 = polyder(polder);
for i = 1:50
a = 0.5 * polyval(polder2, x);
```

```
b = polyval(polder, x);
delta = sqrt(b*b - 2*a*c);
x1 = -2 * c / (b + delta);
x2 = -2 * c / (b - delta);
if abs(x1) < abs(x2)
x = x + x1;
else
x = x + x2;
end
result(i) = x;
c = p(x);
if abs(c) < 10e-10
fprintf("We can find the root of this polynomial by: %d steps!", i);
return;
end
end
end
function y = p(x)
y = 1*x^4 + 0.5*x^3 + 5*x^2 -2*x -3;
end
```

Task 3:

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

Theoretical background:

Laguerre's method is defined by the following formula:

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

Where n denotes the order of the polynomial, and the sign in the denominator is chosen in a way assuring a large absolute value of the denominator.

Laguerre's method is globally convergent so it converges to real roots for any real starting point x0. This method has to be implemented on complex numbers because even if the roots are real, negative number may occur under the square root (1).

Laguerre's method vs Muller's method

As we can see this formula and Muller's formula are similar. The Laguerre's formula is slightly more complex, it takes also into account the order of the polynomial, 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.

My solution:

Method	Starting point	Number of iterations	Results
Laguerre's	-1.8	4	-0.5974
Laguerre's	1.2	3	0.8751
Laguerre's	0.1-0.06i	3	-0.5974 - 0.0000i
Laguerre's	0.1+0.06i	3	-0.5974 + 0.0000i
MM2	-1.8	5	-0.5974
MM2	1.2	4	0.8751
MM2	0.1-0.06i	5	-0.5974 - 0.0000i
MM2	0.1+0.6i	5	-0.5974 + 0.0000i

Conclusion:

As we can see, we were able to prove that using same initial points, Laguerre's method is seemingly faster than MM2 method.
And when using initial points which are not close to the real root, Laguerre's method still manages to find the root quite fast, while MM2 takes a bit more iterations to find the root, so Laguerre's method is more efficient.

Code:

```
clc
clear all
v = [1 0.5 5 -2 -3];
fprintf("The Laguerre's methods results: ");
result = Laguerre(v, 10)
result = Laguerre(v, -1.8)
result = Laguerre(v, 1.2)
result = Laguerre(v, 0.1 - 0.06i)
result = Laguerre(v, 0.1 + 0.06i)
fprintf("The Muller's secod methods results: ");
result = MM2(v, 10)
result = MM2(v, -1.8)
result = MM2(v, 1.2)
```

```
result = MM2(v, 0.1 - 0.06i)
result = MM2(v, 0.1 + 0.06i)
function result = Laguerre(v, x)
polder = polyder(v);
polder2 = polyder(polder);
10
n = length(v) - 1;
for i = 0:50
px = polyval(v, x);
if abs(px) < 10e-12
fprintf("We can find the root of this polynomial by: %d steps!", i);
return;
end
g = polyval(polder, x) / px;
h = g * g - polyval(polder2, x) / px;
c = sqrt((n - 1) * (n * h - g * g));
if abs(g - c) > abs(g + c)
x = x - (n / (g - c));
else
x = x - (n / (g + c));
end
result = x;
end
end
function result = MM2(v, x)
c = p(x);
polder = polyder(v);
polder2 = polyder(polder);
for i = 1:50
a = 0.5 * polyval(polder2, x);
b = polyval(polder, x);
delta = sqrt(b*b - 2*a*c);
x1 = -2 * c / (b + delta);
x2 = -2 * c / (b - delta);
if abs(x1) < abs(x2)
x = x + x1;
else
x = x + x2;
end
result(i) = x;
c = p(x);
if abs(c) < 10e-10
fprintf("We can find the root of this polynomial by: %d steps!", i);
return;
end
end
end
function y = p(x)
y = 1*x^4 + 0.5*x^3 + 5*x^2 -2*x -3;
end
```