Numerical Methods, project B, Number 32

Krzysztof Rudnicki
Student number: 307585
Advisor: dr Adam Krzemieniowski

December 3, 2021

Contents

Chapter 1

Find all zeros of function

1.1 a) False position method

1.1.1 Problem

We have to find zeros of the function

$$f(x) = -2.1 + 0.3x - xe^{-x}$$

In the interval $[-5; 10]$ using false position method.

1.1.2 Theoretical Introduction

False position method also called *regula falsi* in fancier circles is similar to the bisection method, the difference is that the interval we use $[a_n, b_n]$ is divided into two subintervals. We have:

- α - The root
- a_n - 'left' interval
- b_n - 'right' interval
- $f(a_n)$ - Value at left interval
- $f(b_n)$ - Value at right interval

We get:

$$\frac{f(b_n) - f(a_n)}{b_n - a_n} = \frac{f(b_n) - 0}{b_n - c_n}$$

From which we get:

$$c_n = b_n - \frac{f(b_n)(b_n - a_n)}{f(b_n) - f(a_n)} = \frac{a_n f(b_n) - b_n f(a_n)}{f(b_n) - f(a_n)}$$

Then we choose next interval as in the bisection method so: we calculate products of function values at a_n and b_n and that subinterval is selected for the next iteration of *false position* method. This subinterval corresponds to the negative product value.

Properties of *false position method*

This method is always convergent, simillary to bisection method, since it will always choose and shorten the interval which contains the root. If the function is continous and differentiable the method is linearly convergent. That being said the convergence may become sluggish. It can happen if for example one of the endpoints of the intervals will remain the same and the iterating will not shorten the interval to 0. One of the examples of functions that lead to that are barrier functions used in constrained optimization methods.

Improvement to the method In order to improve the formula and avoid aforementioned situation we can take smaller value of the function for the value that does not change. For right end:

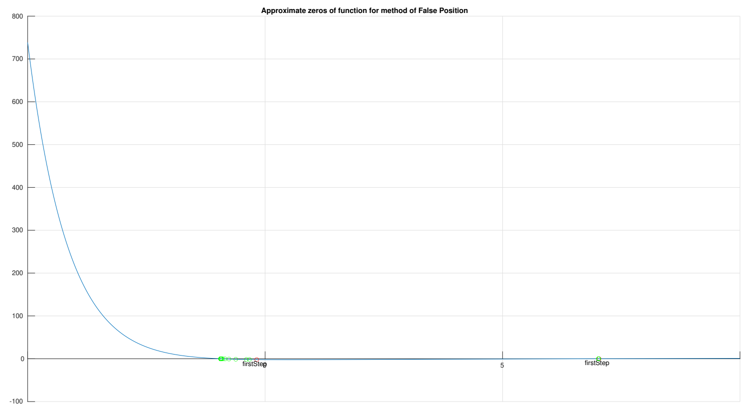
$$c_n = \frac{a_n \frac{f(b_n)}{2} - b_n f(a_n)}{\frac{f(b_n)}{2} - f(a_n)}$$

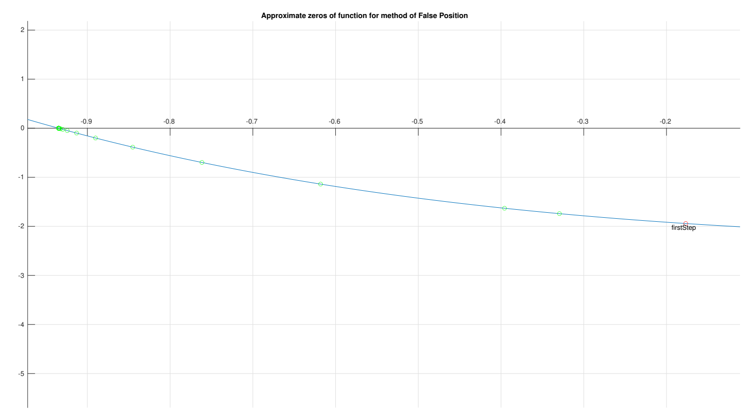
And for left end:

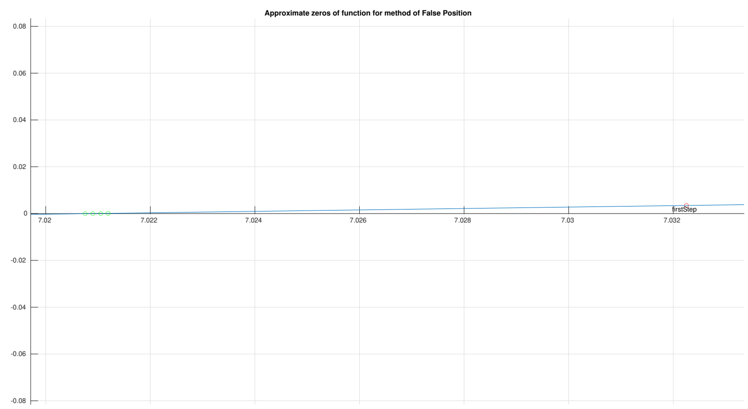
$$c_n = \frac{a_n f(b_n) - b_n \frac{f(a_n)}{2}}{f(b_n) - \frac{f(a_n)}{2}}$$

This is called *modified regula falsi* or *Illinois algorithm.* It is superlinearly convergent, globally convergent and length of intervals we get in each iterations converges to zero.

1.1.3 Results


Approximate zeros of function for method of False Position
800
700
600
500
400
300
200
100
0
-100
5
firstStep
firstStep


Approximate zeros of function for method of False Position
2
1
0
-1
-2
-3
-4
-5
-0.9
-0.8
-0.7
-0.6
-0.5
-0.4
-0.3
-0.2
firstStep



1.2 b) the Newton's method

1.2.1 Problem

We have to find zeros of the function

$$f(x) = -2.1 + 0.3x - xe^{-x}$$

In the interval $[-5; 10]$
using the Newton's method

1.2.2 Theoretical Introduction

The Newton's method also called *the tangent method* relies on first order part of its expansion into Taylor series for a given current approximation of root.

$$f(x) \approx f(x_n) + f^{'}(x_n)(x - x_n)$$

Then we obtain the next point x_{x+1} by finding root of linear function:

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

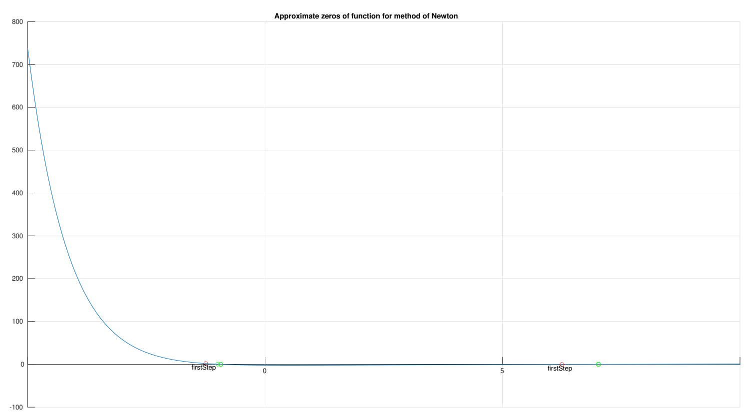
From this we get formula for x_{n+1}:

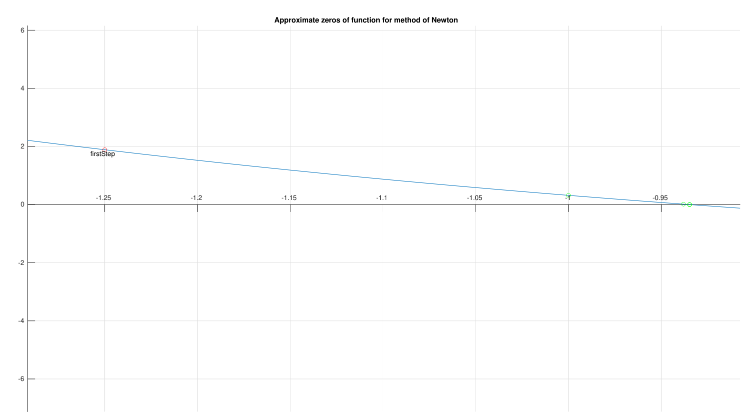
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

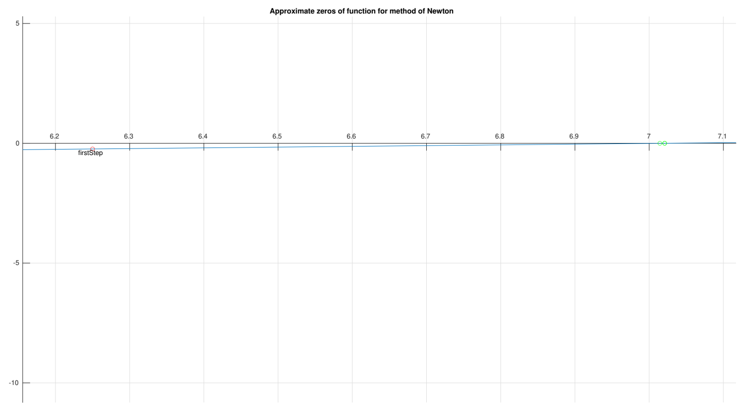
This method as opposed to *regula falsi* method is locally convergent, should we choose initial point too far from the root (area which is close enough to root is called set of attraction) then we can get a divergence. On the other side if the Newton's method will converge then it is quite rapid with convergence of order p = 2 - quadratic convergence.

Newton's method is also effective if the function derrivative is far from zero, so the slope of the function is steep, conversely if the derrivative is close to zero the method is not recommended.

1.2.3 Results







Approximate zeros of function for method of Newton
5
0
-5
-10
6.2
6.3
6.4
6.5
6.6
6.7
6.8
6.9
7
7.1
firstStep

Chapter 2

Find real and complex roots of the polynomial

2.1 Problem

We have to find all real and complex roots of the polynomial:

$$f(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

where:

$$[a_4\ a_3\ a_2\ a_1\ a_0] = [-2\ 12\ 4\ 1\ 3]$$

So our polynomial looks like this:

$$f(x) = -2x^4 + 12x^3 + 4x^2 + 1x + 3$$

Using the Müller's method. We have to implement both MM1 and MM2 versions. We also need to find real roots using the Newton's method and compare these results with what we got from MM2 version of the Müller's method.

2.2 Theoretical Introduction

Müller's method revoles around the idea of approximating the polynomial locally close to the root by a quadratic function. Based on three different points we can use quadratic interpolation and develop our method. This means that we can treat it as a generalization of secant method. That being said we can also realize it in an efficient way if we use just one point. We can use for this case values of polynomial, and its first and second derrivative at current point.

Accordingly there are two versions of Müller's method: **MM1** and **MM2**.

2.2.1 MM1

Given three points: $x_0; x_1; x_2$ and their polynomial values: $f(x_0), f(x_1), f(x_2)$ we construct a (quadratic) function passing through these points. Then we find roots of this parabola and we choose one of these rots for the approximation of the result.

For example: Assume that x_2 is the approximation of the root. Let's introduce variable z such that:

$$z = x - x_2$$

And differences:

$$z_0 = x_0 - x_2$$

$$z_1 = x_1 - x_2$$

We have quadratic function:

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

Using three points from above we get:

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

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

$$y(z_2) = c = f(x_2)$$

And then we get system of equation that we can solve to find a and b:

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

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

Roots are equal to:

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

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

We choose a root with smaller absolute value for next iteration:

$$z_{min} = \min |z_+, z_-|$$

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

Then we choose new point x_3 and two selected from x_0, x_1, x_2 which were closer to x_3.

This method should also work for $\Delta < 0$

2.2.2 MM2

This method being numerically more effective is usually recommended. We calculate values of a polynomial and its first and second derrivatives at one point.

from definition of quadratic function:

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

we can get:

$$z = x - x_k$$

If $z = 0$ then:

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

We can derive from that formula for roots:

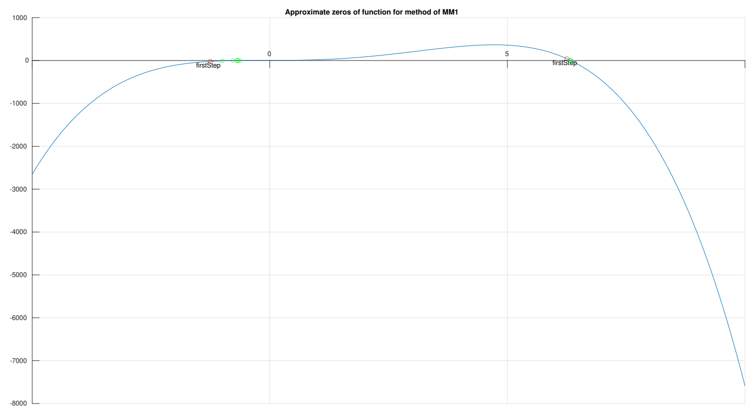
$$z_{\pm} = \frac{-2f(x_k)}{f'(x_k) \pm \sqrt{(f'(x_k))^2 - 2f(x_k)f''(x_k)}}$$

Then we choose root with smaller absolute value for next iteration:

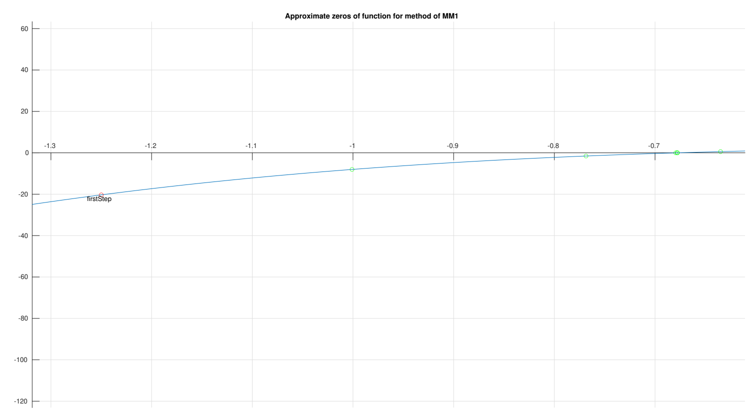
$$x_{k+1} = x_k + z_{min}$$

Again this method should be implemented in complex number arithmetic. This method is locally convergent with order of convergence equal to 1.84. It is locally more effective that secant method and it is almost as fast as Newton's method while being capable of finding complex roots. It can be used to find roots of polynomials or another nonlinear functions.

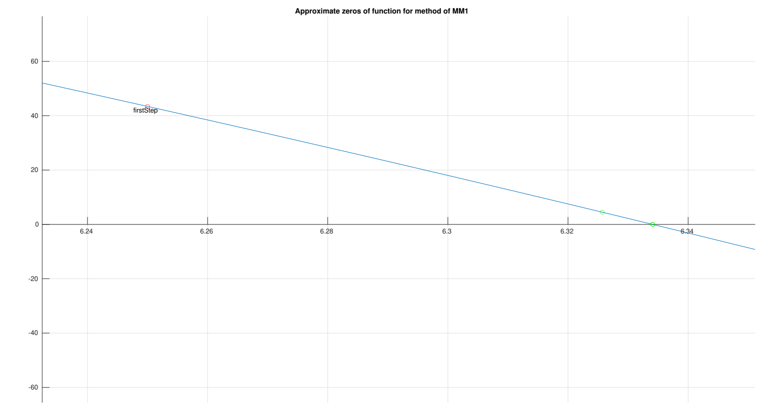
2.3 Results

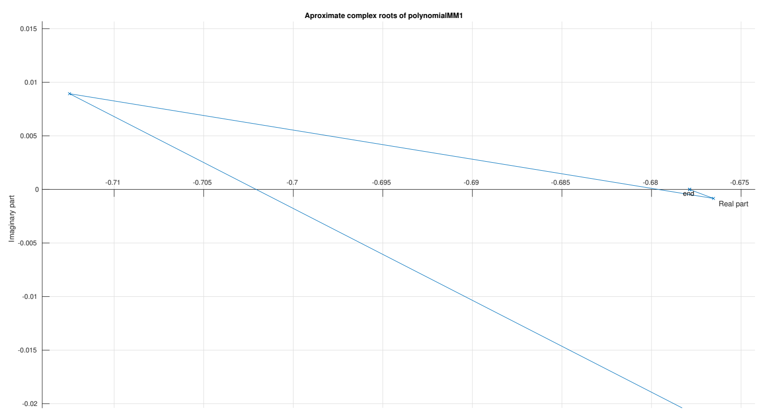


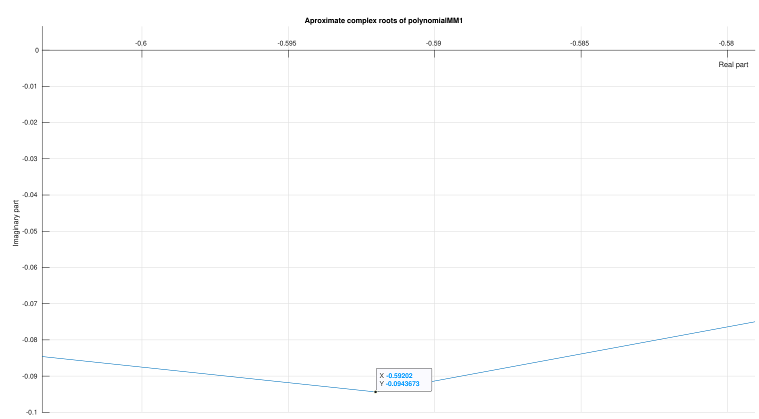
Approximate zeros of function for method of MM1
firstStep
firstStep

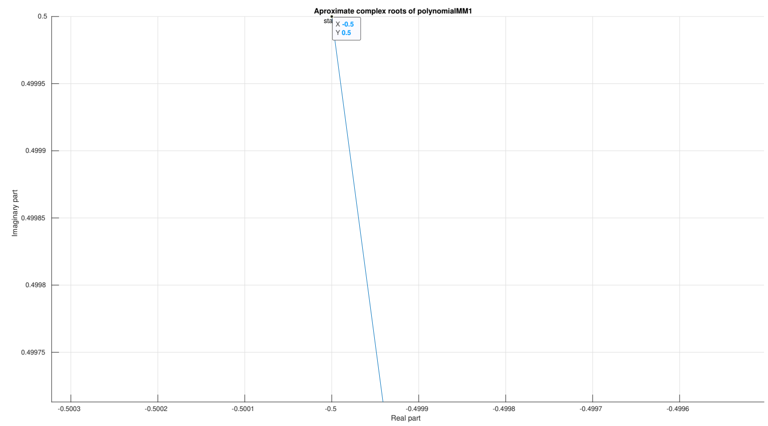


Approximate zeros of function for method of MM1
firstStep

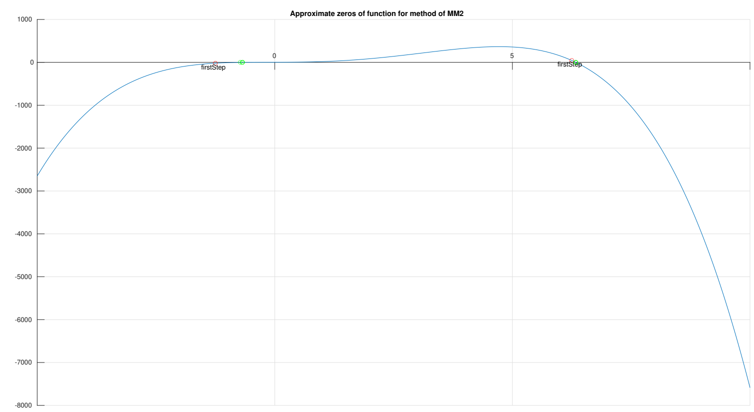

Approximate zeros of function for method of MM1
60
40
20
0
-20
-40
-60
6.24
6.26
6.28
6.3
6.32
6.34
firstStep


Aproximate complex roots of polynomialMM1
0.015
0.01
0.005
0
-0.005
-0.01
-0.015
-0.02
-0.71
-0.705
-0.7
-0.695
-0.69
-0.685
-0.68
-0.675
end
Real part
Imaginary part

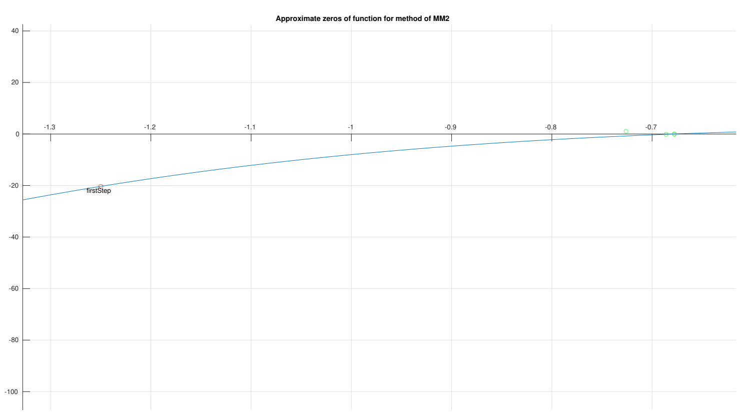

Aproximate complex roots of polynomialMM1
-0.6
-0.595
-0.59
-0.585
-0.58
Real part
0
-0.01
-0.02
-0.03
-0.04
-0.05
-0.06
-0.07
-0.08
-0.09
-0.1
Imaginary part
X -0.59202
Y -0.0943673



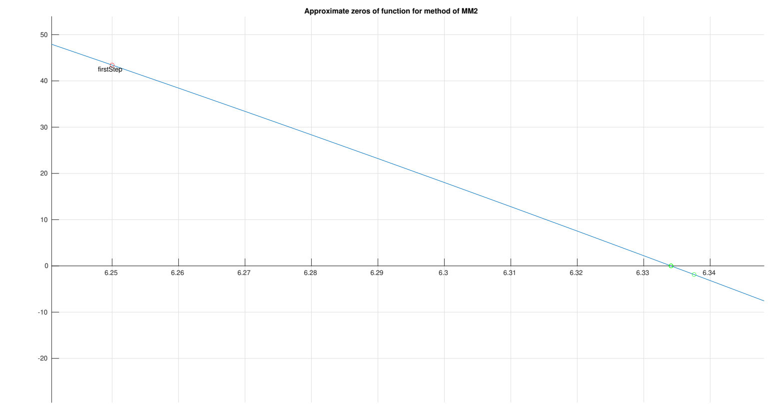
Aproximate complex roots of polynomialMM1
X -0.5
Y 0.5
Imaginary part
Real part

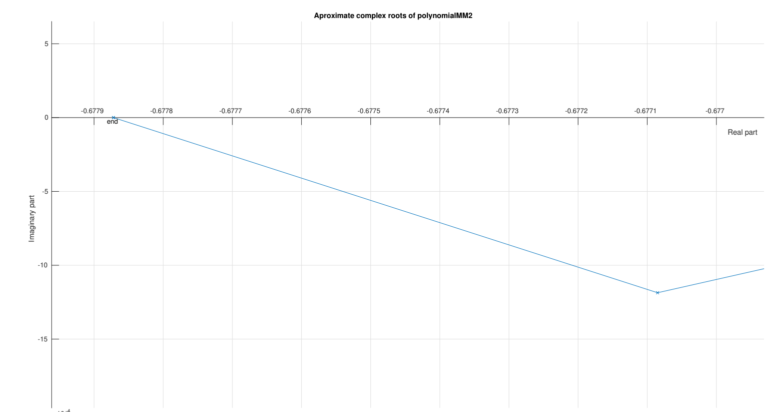


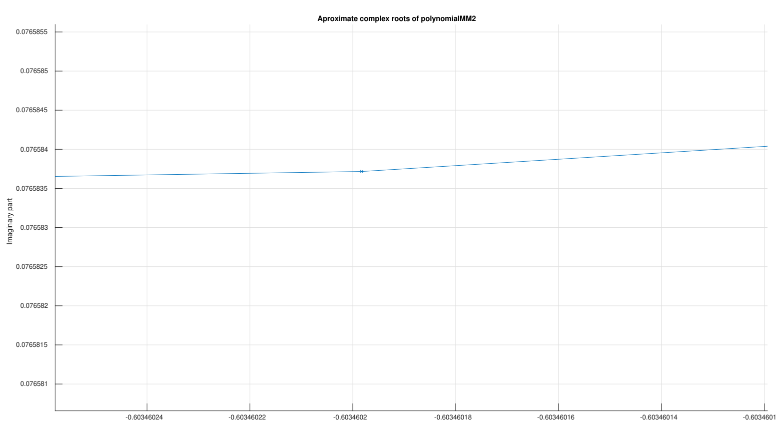
Approximate zeros of function for method of MM2
firstStep
firstStep

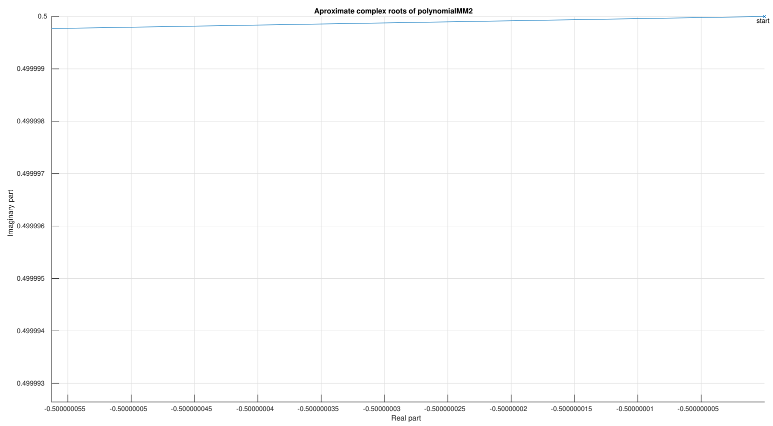


Approximate zeros of function for method of MM2
firstStep


Approximate zeros of function for method of MM2
firstStep


Aproximate complex roots of polynomialMM2
end
Real part
Imaginary part


Aproximate complex roots of polynomialMM2
Imaginary part
Real part



2.3.1 Comparison of results between MM1 and MM2

2.3.2 Comparison of results between Newton's method and MM2

Chapter 3

Find real and complex roots of the polynomial using Laguerre's method

3.1 Problem

We have to find all (real and complex) roots of the polynomial from previous exercise:

$$f(x) = -2x^4 + 12x^3 + 4x^2 + 1x + 3$$

Using the Laguerre's method. Then we should compare those results with the MM2 version of the Müller's method.

3.2 Theoretical Introduction

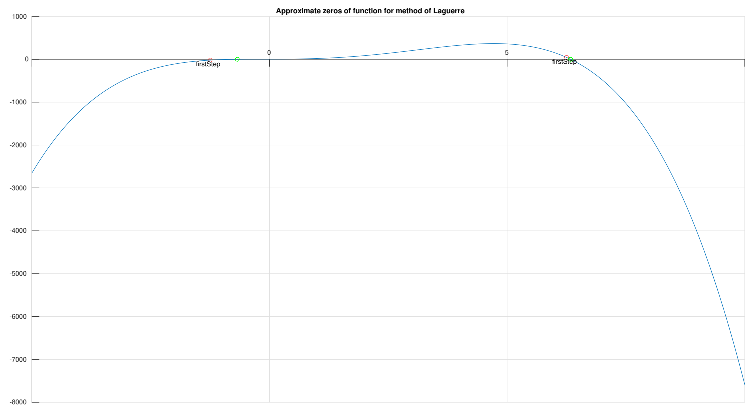
Laguerre's method is defined by a single formula:

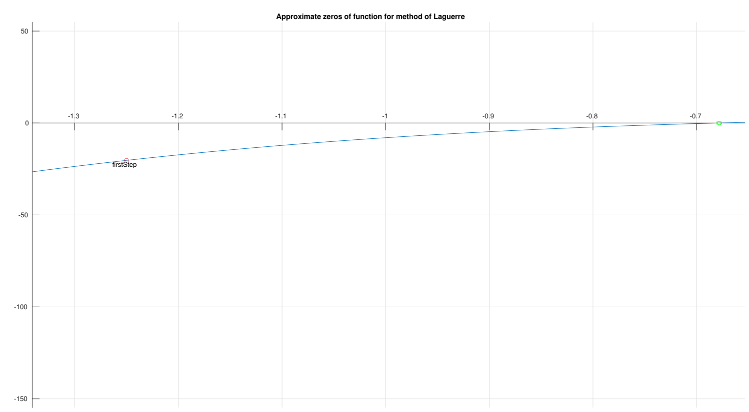
$$x_{k+1} = x_k - \frac{nf(x_k)}{f'(x_k) \pm \sqrt{(n-1)[(n-1)((f'(x_k))^2 - nf(x_k)f''(x_k))]}}$$

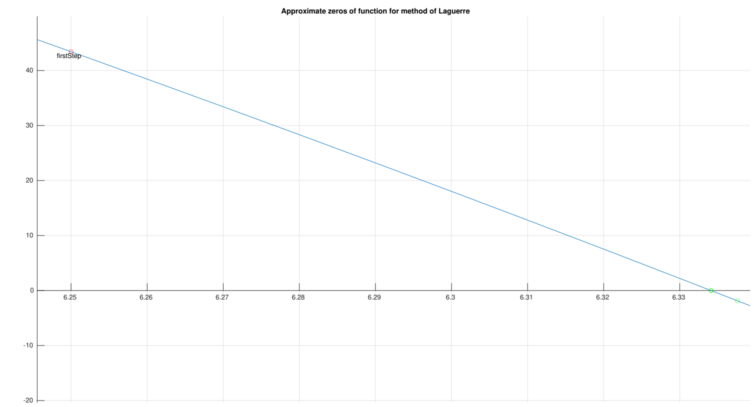
Where: n - order of the polynomial

This formula is similar to the one from MM2 but also takes order of the polynomial into consideration. In general this method is better. For polynomials with real roots it is globally convergent. It does not have formal analysis for complex roots but it usually shows good numerical properties, although divergence may happen.

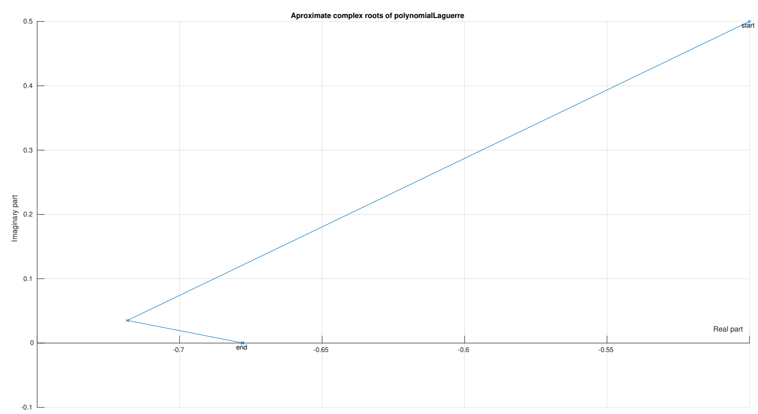
3.3 Results



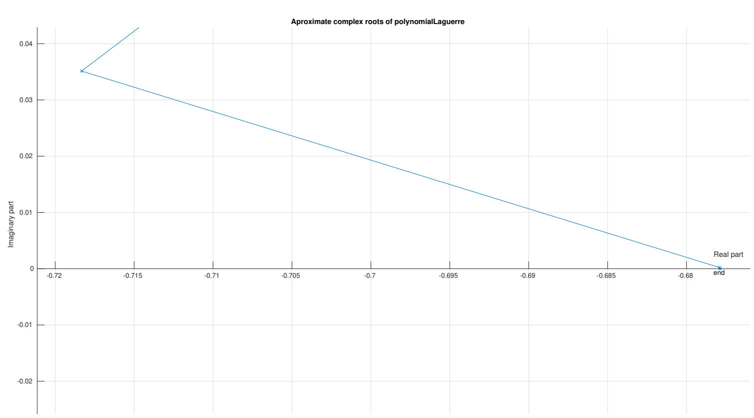




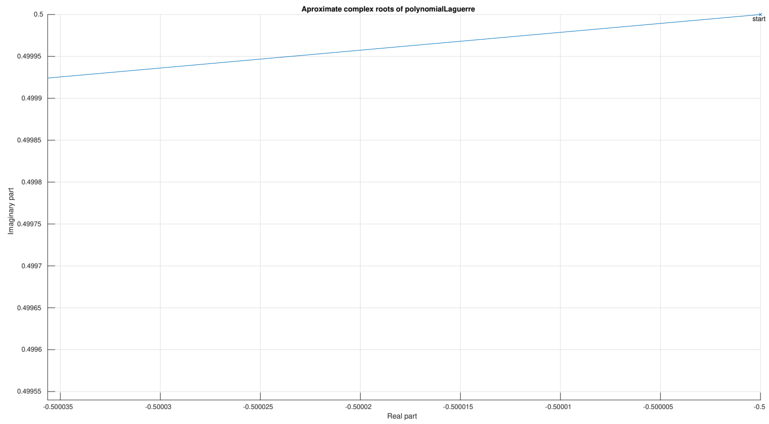
Approximate zeros of function for method of Laguerre
firstStep



Aproximate complex roots of polynomialLaguerre
Imaginary part
Real part
start
end



Aproximate complex roots of polynomialLaguerre
Imaginary part
Real part
end



3.3.1 Comparison of results between MM1 and MM2

Chapter 4

Code appendix

4.1 Task 1

4.1.1 task1Bisection.m

Top of task1Bisection.m

```
interval = [-5, 10];
rootBrackets = rootBracketing(@taskFunction, interval(1),
    interval(2));

printGraph(@taskFunction, 'False Position', @falsePosition,
    interval, rootBrackets, 'Approximate zeros of function for
    method of ');
```

taskFunction

```
function y = taskFunction(x)
    y = -2.1 + 0.3*x - x*exp(1)^(-x);
end
```

falsePosition

```
function [zero, iterations] = falsePosition(taskFunction, a, b,
    tolerance)

    [iterations, lastTwoA, lastTwoB, i] = initialize();
    [zero, iterations, a, b, lastTwoA, lastTwoB] =
       firstTwoIterations(a, b, taskFunction, iterations,
       lastTwoA, lastTwoB);
    [zero, iterations] = falsePositionLoop(taskFunction, zero,
       tolerance, lastTwoA, lastTwoB, i, a, b, iterations);
end
```

initialize

```
function [iterations, lastTwoA, lastTwoB, i] = initialize()
    iterations = double.empty(2, 0);
    lastTwoA = double.empty(2, 0);
    lastTwoB = double.empty(2, 0);
    i = 0;
end
```

taskFunction

```
function y = taskFunction(x)
    y = -2.1 + 0.3*x - x*exp(1)^(-x);
end
```

firstTwoIterations

```
function [zero, iterations, a, b, lastTwoA, lastTwoB] = firstTwoIterations(a, b, taskFunction, iterations, lastTwoA, lastTwoB)
    for j = 1 : 2
        zero = (a*taskFunction(b) - b * taskFunction(a)) / (taskFunction(b) - taskFunction(a));
        iterations(:, size(iterations, 2) + 1) = [zero, taskFunction(zero)];
        if sign(taskFunction(a)) ~= sign(taskFunction(zero))
            b = zero;
        else
            a = zero;
        end
        lastTwoA(j) = a;
        lastTwoB(j) = b;
    end
end
```

falsePositionLoop

```
function [zero, iterations] = falsePositionLoop(taskFunction, zero, tolerance, lastTwoA, lastTwoB, i, a, b, iterations)
    while abs(taskFunction(zero)) > tolerance
        [lastTwoA, i, a, lastTwoB, b, tolerance, zero, iterations] = insideLoop(lastTwoA, i, a, lastTwoB, b, tolerance, taskFunction, iterations);
    end
end
```

insideLoop

```
function [lastTwoA, i, a, lastTwoB, b, tolerance, zero,
    iterations] = insideLoop(lastTwoA, i, a, lastTwoB, b,
    tolerance, taskFunction, iterations)
    [lastTwoA, lastTwoB] = changeLastTwoAB(lastTwoA, lastTwoB,
        i, a, b);
    zero = calculateZero(lastTwoB, tolerance, a, b, lastTwoA,
        taskFunction);
    iterations(:, size(iterations, 2) + 1) = [zero,
        taskFunction(zero)];
    [a, b] = newSubInterval(taskFunction, a, b, zero);
end
```

changeLastTwoAB

```
function [lastTwoA, lastTwoB] = changeLastTwoAB(lastTwoA,
    lastTwoB, i, a, b)
    lastTwoA(mod(i, 2) + 1) = a;
    lastTwoB(mod(i, 2) + 1) = b;
end
```

calculateZero

```
function [zero] = calculateZero(lastTwoB, tolerance, a, b, lastTwoA, taskFunction)
    if(abs(lastTwoB(1) - lastTwoB(2)) < tolerance)
        zero = (a*(taskFunction(b) / 2) - b * taskFunction(a)) / (taskFunction(b) / 2 - taskFunction(a));
    elseif (abs(lastTwoA(1) - lastTwoA(2)) < tolerance)
        zero = (a*taskFunction(b) - b * (taskFunction(a) / 2)) / (taskFunction(b) - (taskFunction(a) / 2));
    else
        zero = (a*taskFunction(b) - b * taskFunction(a)) / (taskFunction(b) - taskFunction(a));
    end
end
```

newSubInterval

```
function [a, b] = newSubInterval(taskFunction, a, b, zero)
    if sign(taskFunction(a)) ~= sign(taskFunction(zero))
        b = zero;
    else
        a = zero;
    end
end
```

4.1.2 task1Newton.m

Top of task1Newton.m

```
interval = [-5, 10];
rootBrackets = rootBracketing(@taskFunction, interval(1), interval(2));

printGraph(@taskFunction, 'Newton', @newtonMethod, interval, rootBrackets, 'Approximate zeros of function for method of ');
```

taskFunction

```
function y = taskFunction(x)
    y = -2.1 + 0.3*x - x*exp(1)^(-x);
end
```

newtonMethod

```
function [zero, iterations] = newtonMethod(taskFunction, a, b, tolerance)
    [iterations, iteration, zero] = initialize(a, b);
    [zero, iterations] = newtonLoop(iterations, iteration, zero, a, b, tolerance, taskFunction);
end
```

initialize

```
function [iterations, step, zero] = initialize(a, b)
    iterations = double.empty(2, 0);
    step = sqrt(eps);
    zero = (a + b) / 2;
    iterations(:, size(iterations, 2) + 1) = [zero, taskFunction(zero)];
end
```

newtonLoop

```
function [zero, iterations] = newtonLoop(iterations, iteration, zero, a, b, tolerance, taskFunction)
    while abs(taskFunction(zero)) > tolerance
        [zero, iterations] = insideLoop(taskFunction, zero, iteration, iterations, a, b);
    end
end
```

insideLoop

```
function [zero, iterations] = insideLoop(taskFunction, zero, iteration, iterations, a, b)
    [zero, iterations] = calculateZeroIterations(taskFunction, zero, iteration, iterations);
    checkForDivergence(zero, a, b);
end
```

calculateZeroIterations

```
function [zero, iterations] = calculateZeroIterations(taskFunction, zero, iteration, iterations)
    derivative = (taskFunction(zero + iteration) - taskFunction(zero - iteration)) / (2 * iteration);
    zero = zero - taskFunction(zero) / derivative;
    iterations(:, size(iterations, 2) + 1) = [zero, taskFunction(zero)];
end
```

checkForDivergence

```
function checkForDivergence(zero, a, b)
    if zero < a || zero > b
        error('Divergent iteration');
    end
end
```

4.2 Task 2

4.2.1 task2MM1.m

Top of task2MM1

```
interval = [-5, 10];
rootBrackets = rootBracketing(@polynomial, interval(1),
    interval(2));

printGraph(@polynomial, 'MM1', @mm1, interval, rootBrackets, '
    Approximate zeros of function for method of ');

printComplexGraph(@polynomial, 'MM1', @mm1, [-1 + i, 0], '
    Aproximate complex roots of polynomial');
```

polynomial

```
function y = polynomial(x)
    y = -2 * x^4 + 12 * x^3 + 4* x^2 + 1 * x + 3;
end
```

mm1

```
function [approximation, iterations] = mm1(polynomial, a, b,
    tolerance)
    [approximation, approximationValue, iterations] =
        initialize(a, b, polynomial);
    [approximation, iterations] = mm1Loop(approximation,
        tolerance, approximationValue, iterations, polynomial);
end
```

initialize

```
function [approximation, approximationValue, iterations] =
   initialize(a, b, polynomial)
    approximation = [a, b, (a + b) / 2];
    approximationValue = arrayfun(polynomial, approximation);
    iterations = [approximation(3); polynomial(approximation(3)
       )];
end
```

mm1Loop

```
function [approximation, iterations] = mm1Loop(approximation,
   tolerance, approximationValue, iterations, polynomial)
    while abs(polynomial(approximation(3))) > tolerance
        [approximation, approximationValue, iterations] =
           insideLoop(approximation, approximationValue,
           polynomial, iterations);
    end
end
```

insideLoop

```
function [approximation, approximationValue, iterations] =
   insideLoop(approximation, approximationValue, polynomial,
   iterations)
    equationsSystem = createEquationSystem(approximation,
       approximationValue);
    [zPlus, zMinus] = rootsOfQuadraticFormula(equationsSystem,
       approximationValue);
    [approximation, approximationValue, iterations] =
       updateApproximations(zPlus, zMinus, approximation,
       iterations, polynomial);
end
```

createEquationSystem

```
function equationsSystem = createEquationSystem(approximation,
   approximationValue)
    [z0, z1, difference0, difference1] =
       initializeEquationSystem(approximation,
       approximationValue);
    equationsSystem = solveEquationSystem(z0, difference0, z1,
       difference1);
end
```

rootsOfQuadraticFormula

```
function [zPlus, zMinus] = rootsOfQuadraticFormula(
   equationsSystem, approximationValue)
    [a, b, c] = createApproximatedQuadraticFormula(
       equationsSystem, approximationValue);
    [zPlus, zMinus] = findRootsOfQuadraticFormula(a, b, c);
end
```

updateApproximations

```
function [approximation, approximationValue, iterations] =
   updateApproximations(zPlus, zMinus, approximation,
   iterations, polynomial)
    newApproximation = chooseNewRoot(zPlus, zMinus,
       approximation);
    iterations = addZeroToIterationVector(newApproximation,
       iterations, polynomial);
    worstApproximationIndex = getWorstApproximationIndex(
       approximation, newApproximation);
    [approximation, approximationValue] =
       deleteWorstApproximation(worstApproximationIndex,
       approximation, polynomial, newApproximation);
end
```

initializeEquationSystem

```
function [z0, z1, difference0, difference1] =
  initializeEquationSystem(approximation, approximationValue)
    z0 = approximation(1) − approximation(3);
    z1 = approximation(2) − approximation(3);
    difference0 = approximationValue(1) − approximationValue(3)
        ;
    difference1 = approximationValue(2) − approximationValue(3)
        ;
end
```

solveEquationSystem

```
function equationsSystem = solveEquationSystem(z0, difference0,
    z1, difference1)
    equationsSystem = [z0 ^ 2, z0, difference0; z1 ^ 2, z1,
        difference1];
    reductor = equationsSystem(2, 1) / equationsSystem(1, 1);
    equationsSystem(2, :) = equationsSystem(2, :) − reductor *
        equationsSystem(1, :);
    equationsSystem(2, 1) = 0;
    equationsSystem(2, :) = equationsSystem(2, :) ./
        equationsSystem(2, 2);
    equationsSystem(1, :) = equationsSystem(1, :) −
        equationsSystem(1, 2) * equationsSystem(2, :);
    equationsSystem(1, :) = equationsSystem(1, :) ./
        equationsSystem(1, 1);
end
```

createApproximatedQuadraticFormula

```
function [a, b, c] = createApproximatedQuadraticFormula(
   equationsSystem, approximationValue)
    a = equationsSystem(1, 3);
    b = equationsSystem(2, 3);
    c = approximationValue(3);
end
```

findRootsOfQuadraticFormula

```
function [zPlus, zMinus] = findRootsOfQuadraticFormula(a, b, c)
    zPlus = -2 * c / (b + sqrt(b ^ 2 - 4 * a * c));
    zMinus = -2 * c / (b - sqrt(b ^ 2 - 4 * a * c));
end
```

chooseNewRoot

```
function newApproximation = chooseNewRoot(zPlus, zMinus,
   approximation)
    if abs(zPlus) < abs(zMinus)
        newApproximation = approximation(3) + zPlus;
    else
        newApproximation = approximation(3) + zMinus;
    end
end
```

addZeroToIterationVector

```
function iterations = addZeroToIterationVector(newApproximation
   , iterations, polynomial)
    zero = newApproximation;
    iterations(:, size(iterations, 2) + 1) = [zero, polynomial(
       zero)];
end
```

getWorstApproximationIndex

```
function worstApproximationIndex = getWorstApproximationIndex(approximation, newApproximation)
    worstApproximationIndex = -1;
    worstApproximationDifference = 0;
    for i = 1:size(approximation, 2)
        diff = abs(approximation(i) - newApproximation);
        if diff > worstApproximationDifference
            worstApproximationIndex = i;
            worstApproximationDifference = diff;
        end
    end
end
```

deleteWorstApproximation

```
function [approximation, approximationValue] = deleteWorstApproximation(worstApproximationIndex, approximation, polynomial, newApproximation)
    approximation(worstApproximationIndex) = [];
    approximation(3) = newApproximation;
    approximationValue = arrayfun(polynomial, approximation);
end
```

4.2.2 task2MM2.m

top of task2MM2.m

```
interval = [-5, 10];
rootBrackets = rootBracketing(@polynomial, interval(1),
   interval(2));

printGraph(@polynomial, 'MM2', @mm2, interval, rootBrackets, '
   Approximate zeros of function for method of ');

printComplexGraph(@polynomial, 'MM2', @mm2, [-1 + i, 0], '
   Aproximate complex roots of polynomial');
```

polynomial

```
function y = polynomial(x)
    y = -2 * x^4 + 12 * x^3 + 4* x^2 + 1 * x + 3;
end
```

mm2

```
function [approximation, iterations] = mm2(polynomial, a, b,
   tolerance)
    [approximation, iterations] = initialize(a, b, polynomial);
    [approximation, iterations] = mm2Loop(approximation,
       iterations, polynomial, tolerance);
end
```

initialize

```
function [approximation, iterations] = initialize(a, b,
   polynomial)
    approximation = (a + b) / 2;
    iterations = [approximation; polynomial(approximation)];
end
```

mm2Loop

```
function [approximation, iterations] = mm2Loop(approximation, iterations, polynomial, tolerance)
    while abs(polynomial(approximation)) > tolerance
        [approximation, iterations] = insideLoop(approximation, polynomial, iterations);
    end
end
```

insideLoop

```
function [approximation, iterations] = insideLoop(approximation, polynomial, iterations)
        [a, b, c] = getABC(approximation, polynomial);
        [zPlus, zMinus] = findRoots(a, b, c);
        newApproximation = chooseNewApproximation(zPlus, zMinus, approximation);
        [approximation, iterations] = updateApproximations(newApproximation, iterations, polynomial);
end
```

getABC

```
function [a, b, c] = getABC(approximation, polynomial)
    c = polynomial(approximation);
    b = derivative(polynomial, approximation, 1);
    a = derivative(polynomial, approximation, 2) / 2;
end
```

findRoots

```
function [zPlus, zMinus] = findRoots(a, b, c)
    zPlus = -2 * c / (b + sqrt(b ^ 2 - 4 * a * c));
    zMinus = -2 * c / (b - sqrt(b ^ 2 - 4 * a * c));
end
```

chooseNewApproximation

```
function newApproximation = chooseNewApproximation(zPlus, zMinus, approximation)
    if abs(zPlus) < abs(zMinus)
        newApproximation = approximation + zPlus;
    else
        newApproximation = approximation + zMinus;
    end
end
```

updateApproximations

```
function [approximation, iterations] = updateApproximations(newApproximation, iterations, polynomial)
    approximation = newApproximation;
    iterations(:, size(iterations, 2) + 1) = [approximation, polynomial(approximation)];
end
```

derivative

```
function y = derivative(function_, x, degree)
    if degree == 0
        y = function_(x);
        return
    end

    step = sqrt(eps);
    y = (derivative(function_, x + step, degree - 1) - derivative(function_, x - step, degree - 1)) / (2 * step);
end
```

4.3 Task 3

Top of Task 3

```
interval = [-5, 10];
rootBrackets = rootBracketing(@polynomial, interval(1),
   interval(2));

printGraph(@polynomial, 'Laguerre', @laguerre, interval,
   rootBrackets, 'Approximate zeros of function for method of '
   );

printComplexGraph(@polynomial, 'Laguerre', @laguerre, [-1 + i,
   0], 'Aproximate complex roots of polynomial');
```

polynomial

```
function y = polynomial(x)
    y = -2 * x^4 + 12 * x^3 + 4* x^2 + 1 * x + 3;
end
```

laguerre

```
function [zero, iterations] = laguerre(polynomial, a, b,
   tolerance)
    [degree, zero, iterations] = initialize(a, b, polynomial);
    [zero, iterations] = laguerreLoop(polynomial, zero,
       tolerance, iterations, degree);

end
```

initialize

```
function [degree, zero, iterations] = initialize(a, b, polynomial)
    degree = 4;
    zero = (a + b) / 2;
    iterations = [zero; polynomial(zero)];
end
```

laguerreLoop

```
function [zero, iterations] = laguerreLoop(polynomial, zero, tolerance, iterations, degree)
    while abs(polynomial(zero)) > tolerance
        [iterations, zero] = insideLoop(polynomial, zero, degree, iterations);
    end
end
```

insideLoop

```
function [iterations, zero] = insideLoop(polynomial, zero, degree, iterations)
    [derrivative0, derrivative1, derrivative2] = calculateDerrivatives(polynomial, zero);
    [zPlus, zMinus] = calculateZ(degree, derrivative0, derrivative1, derrivative2);
    newZero = chooseNewZero(zPlus, zMinus, zero);
    [zero, iterations] = updateZeros(newZero, iterations, polynomial);
end
```

calculateDerrivatives

```
function [derrivative0, derrivative1, derrivative2] = calculateDerrivatives(polynomial, zero)
    derrivative0 = polynomial(zero);
    derrivative1 = derivative(polynomial, zero, 1);
    derrivative2 = derivative(polynomial, zero, 2);
end
```

calculateZ

```
function [zPlus, zMinus] = calculateZ(degree, derrivative0, derrivative1, derrivative2)
    expressionUnderSquareRoot = (degree - 1) * ((degree - 1) * derrivative1 ^ 2 - degree * derrivative0 * derrivative2);
    lagsqrt = sqrt(expressionUnderSquareRoot);

    zPlus = degree * derrivative0 / (derrivative1 + lagsqrt);
    zMinus = degree * derrivative0 / (derrivative1 - lagsqrt);
end
```

chooseNewZero

```
function newZero = chooseNewZero(zPlus, zMinus, zero)
    if abs(zPlus) < abs(zMinus)
        newZero = zero - zPlus;
    else
        newZero = zero - zMinus;
    end
end
```

updateZeros

```
function [zero, iterations] = updateZeros(newZero, iterations,
   polynomial)
    zero = newZero;
    iterations(:, size(iterations, 2) + 1) = [zero, polynomial(
       zero)];
end
```

4.3.1 rootBrackering.m

updateZeros

```
% find the root brackets of a function within the given range
function rootBrackets = rootBracketing(givenFunction,
   intervalLeft, intervalRight)
    [a, b, rootBrackets, resolution] = initializeValues(
       intervalLeft, intervalRight);
    rootBrackets = bracketingLoop(a, b, rootBrackets,
       intervalRight, resolution, givenFunction);
end
```

initializeValues

```matlab
function [a, b, rootBrackets, resolution]  = initializeValues(intervalLeft, intervalRight)
    % define search resolution
    resolution = (intervalRight - intervalLeft) / 6;
    % The higher the value of denominator the less iterations will it take
    % to reach the roots, however in order to have nice graph showing those
    % brackets I will choose relatively small denominator - I have choosen
    % the smallest natural number that still generates brackets on a graph

    % start search at the start of the range
    a = intervalLeft;
    b = intervalLeft + resolution;
    rootBrackets = double.empty(2, 0); % initialize empty vector of size 2
end
```

bracketingLoop

```
function rootBrackets = bracketingLoop(a, b, rootBrackets, intervalRight, resolution, givenFunction)
    while b ~= intervalRight % if the bracket can't be expanded end loop
        % if the function changes sign inside the interval that means that we passed through a root that means that a bracket has been found
        if sign(givenFunction(a)) ~= sign(givenFunction(b))
            % save bracket
            rootBrackets(:, size(rootBrackets, 2) + 1) = [a, b]; % Add the new bracket to existing ones
        end
        % check next bracket
        a = b;
        b = min(a + resolution, intervalRight);
        % Once a + resolution > intervalRight, then we will know that we
        % reached beyond the interval and we must stop
    end
end
```

4.3.2 printGraph.m

```matlab
% graph the real roots of a function
function printGraph(taskFunction, algorithmName, algorithm, interval, rootBrackets, plotTitle)
    figure()
    grid on; % Get y values lines
    hold on; % Retain current plot when adding new plots
    title([plotTitle, algorithmName]);
    set(gca, 'XAxisLocation', 'origin'); % Set properties of current axis
    x = interval(1):0.01:interval(2);
    % x is a vector of values between left and right interval with every value being higher by 0.01
    y = arrayfun(taskFunction, x); % We sketch the function from task for each x
    plot(x, y);

    % iterate over rootBrackets and add them to the plot
    for rootBracket = rootBrackets
        % find all zeros within the bracket using the given algorithm
        % Get all steps from the algorithm we use
        [~, steps] = algorithm(taskFunction, rootBracket(1), rootBracket(2), 1e-10);

        firstStepColor = [1 0 0]; % Red
        otherStepsColor =       [0 1 0]; % Green
        % plot first steps
        scatter(steps(1, 1), steps(2, 1), [], firstStepColor);
        text(steps(1, 1), steps(2, 1), 'firstStep', 'HorizontalAlignment', 'center', 'VerticalAlignment', 'top'); % It makes text appear neatly
        % plot other steps
        scatter(steps(1, 2:end), steps(2, 2:end), [], otherStepsColor);


```

```

    % print root table
    disp([plotTitle, ' (', algorithmName, ')']);
    columns = {'step', 'root', 'value at root'};
    disp(table((1:size(steps, 2))', steps(1, :)', steps(2, :)',
        'VariableNames', columns));
    end
end
```

4.3.3 printComplexGraph.m

```matlab
% graph the complex roots of a function
function printComplexGraph(printComplexGraph, algorithmName, algorithm, rootBrackets, plottitle)
    figure();
    grid on; % Get y values lines
    hold on; % Retain current plot when adding new plots
    title([plottitle, algorithmName]);
    xlabel(Real part);
    ylabel(Imaginary part);
    set(gca, 'XAxisLocation', 'origin'); % Set properties of current axis

    % find all zeros within the bracket using the given algorithm
    [~, steps] = algorithm(printComplexGraph, rootBrackets(1), rootBrackets(2), 1e-15);

    % plot first step
    text(real(steps(1, 1)), imag(steps(1, 1)), 'start', 'HorizontalAlignment', 'center', 'VerticalAlignment', 'top');

    % plot steps on graph
    plot(real(steps(1, :)), imag(steps(1, :)), '-x');

    % plot last step
    text(real(steps(1, end)), imag(steps(1, end)), 'end', 'HorizontalAlignment', 'center', 'VerticalAlignment', 'top');


    % print root table
    disp([plottitle, ' (', algorithmName, ')']);
    columns = {'step', 'root', 'abs value at root'};
    disp(table([1:size(steps, 2)]', steps(1, :)', abs(steps(2, :))', 'VariableNames', columns));
```

```
end
```

Bibliography

[1] Piotr Tatjewski (2014) *Numerical Methods*, Oficyna Wydawnicza Politechniki Warszawskiej