ENUME - Project 2

Author: Michał Łezka
ID: 303873

1. **Find all zeros of the function f(x) = 1.2sin(x)+2ln(x+2)–5 in the interval [2, 12] using:**

a) the false position method,
b) the Newton's method.

Background

False position method

False position method is quite simple. We start from choosing two points 'a' and 'b'. The method is globally convergent, so we don't have to worry about choosing a proper range, but it needs 'a' and 'b' to have different signs. Then, we create a line between f(a) and f(b) and the point of intersection of said line with x axis is marked as 'c'.

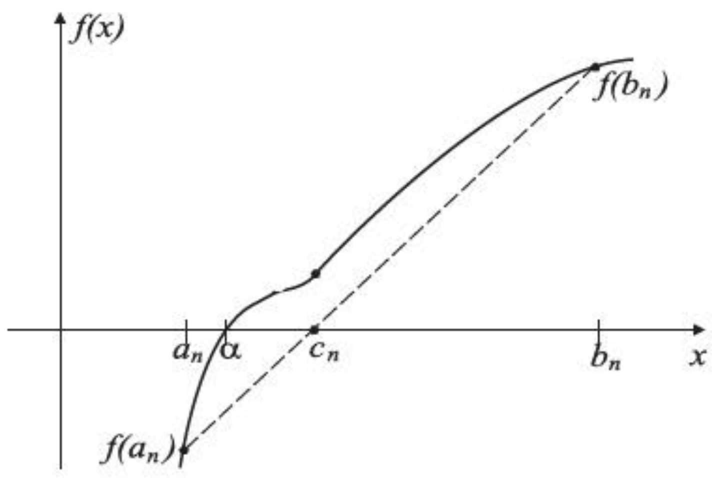


Figure 6.1. A construction of two subintervals by the secant line

It follows directly from the construction shown in Fig. 6.1 that:

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

thus

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

Knowing that, we can calculate ‘c’. Then we check whether f(a) and f(c) have different signs. If yes, we replace ‘b’ with ‘c’ and get a new interval. If not, we check whether f(c) and f(b) have different signs. If yes, we replace ‘a’ with ‘c’ and get a new interval. Now we repeat the process until c_n is close enough to 0 (when c_n < accuracy).

Convergence of this method is linear $p = 1$, however it can be very slow in specific conditions. If ‘a’ or ‘b’ doesn’t change, it means that the interval will not be shortened to zero. This causes slow, or sometimes even very slow convergence time.

Background

Newton’s method

Newton’s method, also called the tangent method, uses Taylor series at a current point x_n to find roots. This method is locally convergent, meaning that if our initial point is too far from the root, the divergence may occur, causing the method to fail. To find roots we follow this simple formula:

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

and continue to iterate until x_{n+1} is satisfactory small (x_{n+1} < accuracy)

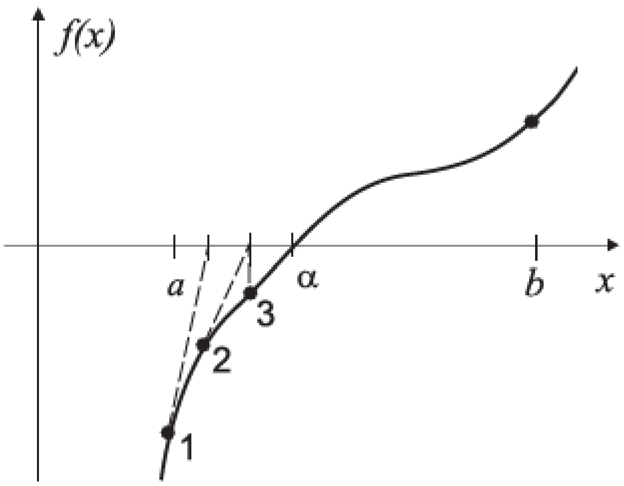


Figure 6.4. An illustration of the Newton’s method

The convergence of Newton’s method is very fast, as it is quadratic $p = 2$. This method is particularly effective when the function derivative at root is sufficiently far from zero (the slope of f(x) is steep). However, if derivative of f(x) is close to zero, then the method is very sensitive to numerical errors when close to the root.

MATLAB implementation

I will start with trivial functions that I needed for completing tasks:

Function calculating f(x)

```
% calculates f(x) as per task description
function y = calc_f(x)
    y = 1.2 * sin(x) + 2 * log(x+2) - 5;
end
```

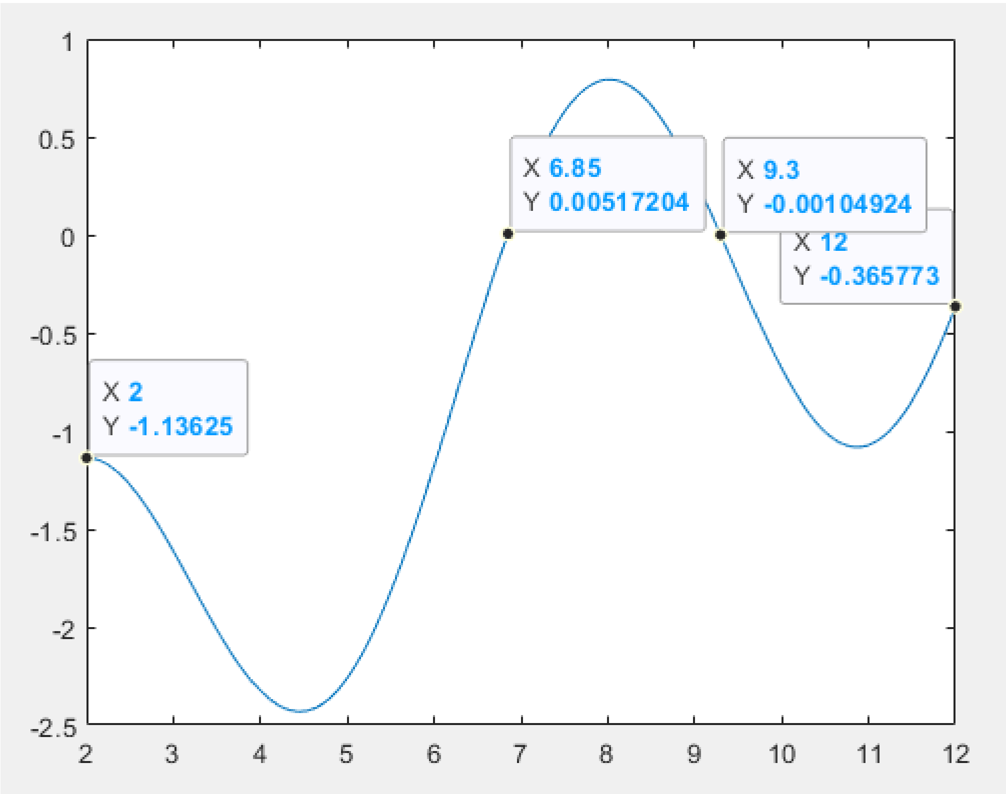
calculating the value x at derivative f'(x)

```
function y = calc_df(x)
    y = 2/(x+2) + 1.2 * cos(x);
end
```

and creation of the graph:

```
function plot_task1()
    x = 2:0.01:12;  % x = left border, step, right border
    plot(x, calc_f(x));
end
```

This is how graph looks with marked boundaries and roots:



False position method

Checks for initial conditions so a and b are within boundaries and checks if they have different sign. Then method is performed according to what was presented in the Background section, and code is commented.

```
function false_position(a, b)
    % initial conditions
    if (a < 2 || a > 12 || b < 2 || b > 12)
        error("a and b must be within [2,12] interval");
    end
    if ( calc_f(a) * calc_f(b) >= 0)
        error("a and b must have different signs!");
    end

    % initial printout:
    fprintf("Interval: [%f, %f]\n", a, b);
    fprintf("Iteration:\t\tc:\t\tvalue of c:\t\tinterval length:\n");
    err = inf;
    accuracy = 1e-6;
    iter_count = 0;
    % loop until error is small enough or iterates too many times
    while err > accuracy && iter_count < 2000
        a_val = calc_f(a);
        b_val = calc_f(b);
        % calculate c as per equation:
        c = (a * b_val - b * a_val) / (b_val - a_val);
        c_val = calc_f(c);
        % select new interval:
        if a_val * c_val < 0
            b = c;
        elseif c_val * b_val < 0
            a = c;
        end
        iter_count = iter_count + 1;
        err = abs(c_val); % calculate error
        % printing results:
        small_interval = abs(a-b);
        fprintf("\t%d\t\t%.8f\t%.8f\t\t%.8f\n", iter_count, c, c_val, small_interval);
    end
end
```

As we can see from the plot, both 2 and 12 have negative values so the result of false position is

```
>> false_position(2,12)
Error using false_position (line 7)
a and b must have different signs!
```

Because of that, we should split this graph into two intervals, e.g. [2, 8] and [8, 12].

Then, we get the following results:

```
>> false_position(2,8)
Interval: [2, 8]
Iteration:        c:        value of c:      interval length:
    1        5.53486142  -1.77741281      2.46513858
    2        7.23987617  0.42780405       1.70501476
    3        6.90910951  0.07716441       1.37424810
    4        6.85193042  0.00756128       1.31706900
    5        6.84635122  0.00064920       1.31148980
    6        6.84587237  0.00005499       1.31101095
    7        6.84583181  0.00000465       1.31097039
    8        6.84582838  0.00000039       1.31096696
```

```
>> false_position(8,12)
Interval: [8, 12]
Iteration:        c:        value of c:      interval length:
    1        10.73672460 -1.07104303      2.73672460
    2        9.16374939  0.13503303       1.57297520
    3        9.33986068  -0.04157366      0.17611129
    4        9.29840364  0.00056875       0.04145704
    5        9.29896314  0.00000172       0.04089755
    6        9.29896483  0.00000001       0.04089586
```

The results overlap with what we see on the graph.

Newton's method

Here it also checks for initial conditions, i.e., if initial point that we choose is between 'a' and 'b' and if 'a' and 'b' itself is between 2 and 12. Then method is performed according to what was presented in the Background section, and code is commented. The method diverges when x_{n+1} is out of bounds [a, b].

```
function Newton_method(init_point, a, b)
    % initial conditions
    if init_point < a || init_point > b
        error("Initial guess should be between a and b.");
    end
    if ( a < 2 || a > 12 || b < 2 || b > 12)
        error("a and b must be within [2,12] interval");
    end

    % initial printout
    fprintf("Initial guess: %f, interval: [%f, %f]\n", init_point, a, b);
    fprintf("Iteration:\t\tx:\t\tvalue of x:\n");
    accuracy = 1e-6;
    err = inf;
    iter_count = 0;
    curr_x = init_point;
    % loop until result is satisfactory or can't find root
    while err > accuracy && iter_count < 2000
        % calculate x_n+1 as per equation
        new_x = curr_x - calc_f(curr_x)/calc_df(curr_x);
        if new_x < a || new_x > b
            error("The method diverged!");
        end
        curr_x = new_x;
        err = abs(calc_f(new_x));
        iter_count = iter_count + 1;
        fprintf("\t%d\t\t%.8f\t%.8f\t\n", iter_count, new_x, calc_f(new_x));
    end

end
```

Results:

```
>> Newton_method(6, 5, 7)
Initial guess: 6.000000, interval: [5.000000, 7.000000]
Iteration:      x:       value of x:
    1       6.83897580  -0.00852001
    2       6.84581560  -0.00001546
    3       6.84582806  -0.00000000
>> Newton_method(9, 2, 12)
Initial guess: 9.000000, interval: [2.000000, 12.000000]
Iteration:      x:       value of x:
    1       9.31850859  -0.01983802
    2       9.29899314  -0.00002869
    3       9.29896483  -0.00000000
```

If the initial point is too far, the method can diverge:

```
>> Newton_method(11, 2, 12)
Initial guess: 11.000000, interval: [2.000000, 12.000000]
Iteration:      x:       value of x:
Error using Newton_method (line 21)
The method diverged!

>> Newton_method(10.5, 2, 12)
Initial guess: 10.500000, interval: [2.000000, 12.000000]
Iteration:      x:       value of x:
    1       8.05462918  0.79199156
Error using Newton_method (line 21)
The method diverged!
```

But it doesn't have to diverge if we're lucky, but it can find different root than we were hoping to find:

```
>> Newton_method(5, 2, 12)
Initial guess: 5.000000, interval: [2.000000, 12.000000]
Iteration:        x:       value of x:
    1        8.60782094  0.59806286
    2        9.55293951  -0.25949337
    3        9.29779436  0.00118617
    4        9.29896494  -0.00000011
```

The results of Newton's method also overlap with what we can see on the graph.

Conclusions

False position proved to be slower than Newton's method. It confirms the theory that says that false position isn't very fast, especially in certain situations where 'a' or 'b' doesn't change, but works globally, with every interval as long as f(a)*f(b)<0. Newton's method is much quicker however it requires from us choosing a good initial point as Newton's method is only locally convergent. If wrong initial point is chosen, function may diverge, which results in not finding any root, or it may find some random root, but often not the one we are hoping for.

3. Find all (real and complex) roots of the polynomial

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

using the Müller's method implementing both the MM1 and MM2 versions. Compare results. Find also real roots using the Newton's method and compare the results with the MM2 version of the Müller's method (using the same initial points).

Background

Müller's method – MM1

The idea behind Müller's method is to locally approximate the polynomial in a neighbourhood of a root by a quadratic function. For that, we choose 3 points that are close to a root, which we call x_0, x_1 and x_2. For those points we also need their polynomial values $f(x_0)$, $f(x_1)$ and $f(x_2)$. A quadratic function (a parabola) is constructed that passes through these points, then the roots of the parabola are found and one of the roots is selected for the next approximation of the solution.

Let's assume that x_2 is an actual approximation of the solution. Let's introduce new variables:

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

The following interpolating parabola is considered:

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

Considering we have 3 points, we get the following:

$$\begin{aligned} az_0^2 + bz_0 + c &= y(z_0) = f(x_0), \\ az_1^2 + bz_1 + c &= y(z_1) = f(x_1), \\ c &= y(0) = f(x_2). \end{aligned}$$

c is trivial to calculate, however for a and b we have to solve the following set of equations:

$$\begin{aligned} az_0^2 + bz_0 &= f(x_0) - f(x_2), \\ az_1^2 + bz_1 &= f(x_1) - f(x_2). \end{aligned}$$

$$\boldsymbol{bz_1 = \frac{f(x_1) - f(x_2) - az_1^2}{z_1}}$$

so

$$az_0^2 + \left(\frac{f(x_1) - f(x_2) - az_1^2}{z_1}\right) z_0 = f(x_0) - f(x_2)$$

$$az_0^2 - az_1 z_0 = \left(\frac{f(x_0)z_1 - f(x_2)z_1 - f(x_1)z_0 + f(x_2)z_0}{z_1}\right)$$

$$\boldsymbol{a = \frac{f(x_0)z_1 - f(x_2)z_1 - f(x_1)z_0 + f(x_2)z_0}{z_0 z_{1(z_0} - z_1)}}$$

Once we have that, we need to calculate x_3:

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

where

$$\begin{aligned} z_{min} &= z_+, \quad \text{if } |b + \sqrt{b^2 - 4ac}| \geq |b - \sqrt{b^2 - 4ac}|, \\ z_{min} &= z_-, \quad \text{in the opposite case.} \end{aligned}$$

For the next iteration the new point x3 is taken, together with those two from points selected from x0, x1, x2 which are closer to x3. We continue to calculate new x_3 until error is satisfyingly small.

Müller's method – MM2

This version of Müller's method uses values of a polynomial and its first and second order derivatives at one point only instead of creating a quadratic function. This method is often recommended as it is numerically slightly cheaper to calculate values of a polynomial and values of its first and second derivatives at one point only than calculating values at 3 different points.

Those are the equations for a, b, and c:

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

However, we don't need them to get root of the function and we can just use the following equation to acquire z_+ and z_-:

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

Then, just like in MM1 method, we choose the smaller one and add it to our initial guess x

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

We continue to calculate new x_k until error is satisfyingly small.

MATLAB implementation

I'll start from MM1 method and its trivial functions:

Calculates f(x)

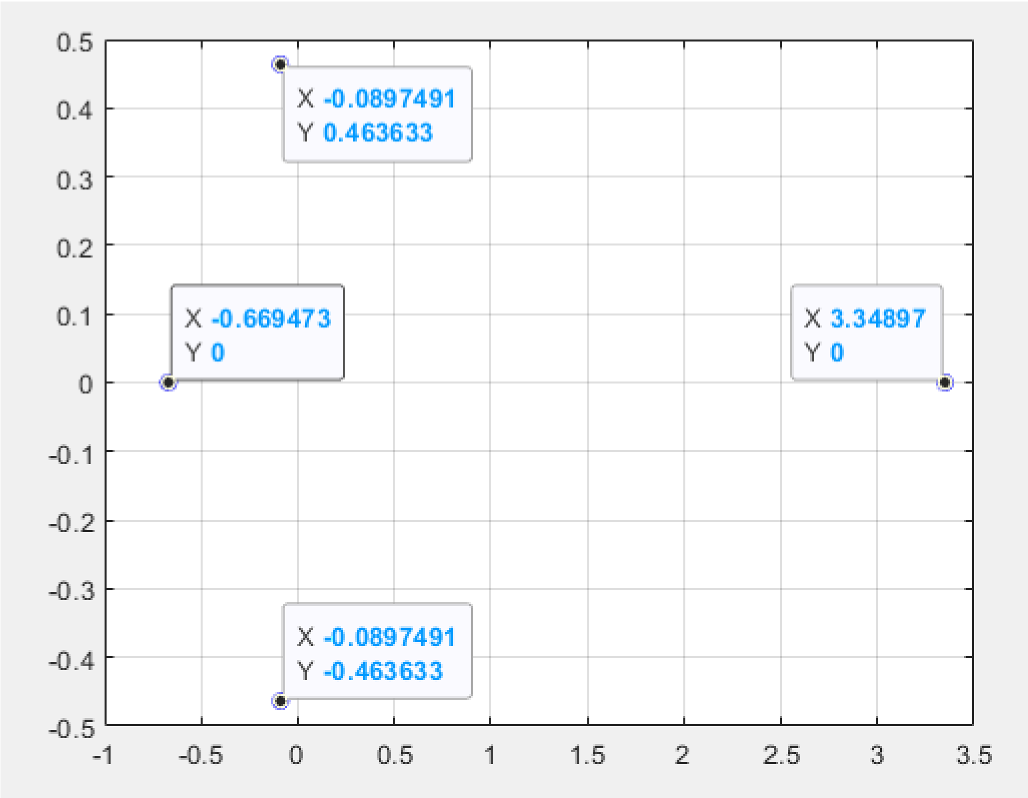
```
function y = calc_f(x)
    y = -2 * x.^4 + 5 * x.^3 + 5 * x.^2 + 2 * x + 1;
end
```

Creates the plot:

```
function plot_task2()
    r = roots([-2 5 5 2 1]);    % finds roots for -2*x.^4 + 5*x.^3 + 5*x.^2 + 2*x + 1
    plot(real(r),imag(r),'bo')  % plots with blue circles
    grid on
    xlabel('Real')
    ylabel('Imaginary')
end
```

Plot for this task is quite different than for the previous one. This is because we have both real and imaginary roots, so I've decided to create a plot in point 0 for real and imaginary numbers

This is how graph looks:



Now the MM1 method:

It starts with some basic assignments like error, accuracy, iteration count and formatted output:

```
function muller_MM1(x0, x1, x2)
    accuracy = 1e-6;
    err = inf;
    iter_count = 0;
    % printout of labels and iteration 0
    fprintf("Iteration\t\t\t\tx2\t\t\t\t\tf(x2)\n");
    fprintf("\t%d\t\t%.6f + %.6fi\t%.6f + %.6fi\n", iter_count, real(x2), imag(x2), real(calc_f(x2)), imag(calc_f(x2)));
```

After that we have the main loop that performs the whole algorithm until error is small enough or it couldn’t find a root. Algorithm follows steps described in previous part

```
while err > accuracy && iter_count < 2000   % loop until error is small enough or iterates too many times
    % assuming x2 is the actual approximation, we assign:
    z0 = x0 - x2;
    z1 = x1 - x2;
    % calculate a, b and c according to the equations
    a = (calc_f(x0)*z1 + calc_f(x2)*z0 - calc_f(x2)*z1 - calc_f(x1)*z0)/...
        (z0*z1*(z0-z1));
    b = (calc_f(x1) - calc_f(x2) - a*z1^2) / z1;
    c = calc_f(x2);
    % now we need to calculate z- and z+
    z_plus = -2*c/(b + sqrt(b.^2 - 4*a*c));
    z_minus = -2*c/(b - sqrt(b.^2 - 4*a*c));
    % choose the smaller one out of z- and z+ and add it to x2
    if abs(z_plus) < abs(z_minus)
        x3 = x2 + z_plus;
    else
        x3 = x2 + z_minus;
    end
    % reassigning values so that x3 and 2 closest numbers to x3 stay
    tmp0 = abs(x3 - x0);
    tmp1 = abs(x3 - x1);
    tmp2 = abs(x3 - x2);
    if tmp0 > tmp1 && tmp0 > tmp2
        x0 = x1;
        x1 = x2;
    elseif tmp1 > tmp0 && tmp1 > tmp2
        x1 = x2;
    end
    x2 = x3;
    err = abs(calc_f(x2));  % calculate error
    iter_count = iter_count + 1;
    % print results of iteration
    fprintf("\t%d\t\t%.6f + %.6fi\t%.6f + %.6fi\n", iter_count, real(x2), imag(x2), real(calc_f(x2)), imag(calc_f(x2)));
end
```

As for the results:

```
>> muller_MM1(2, 3, 4)
Iteration                x2                       f(x2)
     0        4.000000 + 0.000000i    -103.000000 + 0.000000i
     1        3.292273 + 0.000000i    5.235071 + 0.000000i
     2        3.346157 + 0.000000i    0.271680 + 0.000000i
     3        3.348983 + 0.000000i    -0.001173 + 0.000000i
     4        3.348971 + 0.000000i    0.000000 + 0.000000i

>> muller_MM1(2, 5, 8)
Iteration                x2                       f(x2)
     0        8.000000 + 0.000000i    -5295.000000 + 0.000000i
     1        4.328456 + 0.000000i    -193.226024 + 0.000000i
     2        3.653768 + 0.000000i    -37.498997 + 0.000000i
     3        3.249960 + 0.278857i    14.487877 + -22.322138i
     4        3.359337 + -0.020045i   -0.979411 + 1.972580i
     5        3.348668 + -0.000353i   0.029325 + 0.034113i
     6        3.348971 + 0.000001i    0.000009 + -0.000067i
     7        3.348971 + 0.000000i    -0.000000 + -0.000000i
```

As we can see, even if we input values that are not that close to the root, it can still find it (keep in mind that our initial guess, x_2, is equal to 8 here.

Other roots:

```
>> muller_MM1(-5, 2, -1)
Iteration                  x2                        f(x2)
     0        -1.000000 + 0.000000i   -3.000000 + 0.000000i
     1        -0.984541 + 0.000000i   -2.773319 + 0.000000i
     2        -0.792990 + 0.000000i   -0.725971 + 0.000000i
     3        -0.610431 + 0.066949i   0.265887 + 0.218287i
     4        -0.675930 + -0.005942i  -0.028657 + -0.027120i
     5        -0.669475 + -0.000226i  -0.000010 + -0.001000i
     6        -0.669472 + -0.000000i  0.000002 + -0.000000i
     7        -0.669473 + -0.000000i  0.000000 + -0.000000i
```

And imaginary roots:

```
>> muller_MM1(0, 1, 0.5i)
Iteration                  x2                        f(x2)
     0        0.000000 + 0.500000i    -0.375000 + 0.375000i
     1        -0.061864 + 0.446123i   0.013674 + 0.154819i
     2        -0.088739 + 0.462156i   0.003582 + 0.007740i
     3        -0.089743 + 0.463628i   0.000007 + 0.000037i
     4        -0.089749 + 0.463633i   0.000000 + 0.000000i
```

```
>> muller_MM1(0, 1, -0.5i)
Iteration                  x2                        f(x2)
     0        -0.000000 + -0.500000i  -0.375000 + -0.375000i
     1        -0.061864 + -0.446123i  0.013674 + -0.154819i
     2        -0.088739 + -0.462156i  0.003582 + -0.007740i
     3        -0.089743 + -0.463628i  0.000007 + -0.000037i
     4        -0.089749 + -0.463633i  0.000000 + -0.000000i
```

MM2 method:

trivial functions:

Calculating f(x):

```
function y = calc_f(x)
    y = -2 * x.^4 + 5 * x.^3 + 5 * x.^2 + 2 * x + 1;
end
```

Calculating f’(x):

```
function y = calc_df(x)
    y = -8*x^3 + 15*x^2 + 10*x + 2;
end
```

Calculating f’’(x):

```
function y = calc_ddf(x)
    y = -24*x^2 + 30*x + 10;
end
```

Looks pretty similar to M1, but doesn't calculate a, b, or c, has only one x – x_0

```
function muller_MM2(x0)     % x0 - initial guess
    accuracy = 1e-6;
    err = inf;
    iter_count = 0;
    % printout of labels and iteration 0
    fprintf("Iteration\t\t\tx0\t\t\t\tf(x0)\n");
    fprintf("\t%d\t\t%.6f + %.6fi\t%.6f + %.6fi\n", iter_count, real(x0), imag(x0), real(calc_f(x0)), imag(calc_f(x0)));
    while err > accuracy && iter_count < 2000   % loop until error is small enough or iterates too many times
        % there is no need to calculate a, b and c
        % so we jsut need to calculate z- and z+
        z_plus = -2*calc_f(x0)/(calc_df(x0) + sqrt(calc_df(x0).^2 - 2*calc_f(x0)*calc_ddf(x0)));
        z_minus = -2*calc_f(x0)/(calc_df(x0) - sqrt(calc_df(x0).^2 - 2*calc_f(x0)*calc_ddf(x0)));
        % choose the smaller one out of z- and z+ and add it to x0
        if abs(z_plus) < abs(z_minus)
            x0 = x0 + z_plus;
        else
            x0 = x0 + z_minus;
        end
        err = abs(calc_f(x0));  % calculate error
        iter_count = iter_count + 1;
        % print results of iteration
        fprintf("\t%d\t\t%.6f + %.6fi\t%.6f + %.6fi\n", iter_count, real(x0), imag(x0), real(calc_f(x0)), imag(calc_f(x0)));
    end
end
```

Results:

```
>> muller_MM2(7)
Iteration                x0                    f(x0)
    0        7.000000 + 0.000000i     -2827.000000 + 0.000000i
    1        4.973849 + 1.344968i     -87.695502 + -670.642779i
    2        4.113942 + 0.034996i     -130.728802 + -9.097838i
    3        3.200798 + -0.209244i    15.723191 + 15.432461i
    4        3.345521 + 0.001060i     0.333008 + -0.102033i
    5        3.348971 + -0.000000i    -0.000001 + 0.000001i
    6        3.348971 + 0.000000i     -0.000000 + -0.000000i
```

Even if we're far off the mark, we can still find the root rather quickly:

```
>> muller_MM2(50)
Iteration                x0                    f(x0)
    0        50.000000 + 0.000000i     -11862399.000000 + 0.000000i
    1        33.552778 + 11.623697i    -620154.056843 + -2898616.978118i
    2        25.338597 + 0.001516i     -739837.806805 + -182.353357i
    3        17.123727 + -5.794530i    -38253.978390 + 180587.097066i
    4        13.036413 + -0.002824i    -45810.294659 + 42.480113i
    5        8.946856 + 2.858666i      -2236.361380 + -11129.032359i
    6        6.951136 + 0.008555i      -2733.451126 + -16.174012i
    7        4.944512 + -1.326050i     -87.391057 + 647.302626i
    8        4.093224 + -0.031939i     -125.429795 + 8.124013i
    9        3.189662 + 0.186152i      15.885014 + -13.476449i
    10       3.346308 + -0.001704i     0.257371 + 0.164153i
    11       3.348971 + 0.000000i      0.000000 + -0.000001i
```

The rest of the results:

```
>> muller_MM2(-5)
Iteration              x0                      f(x0)
    0        -5.000000 + 0.000000i    -1759.000000 + 0.000000i
    1        -3.206757 + 1.240295i    -78.968439 + 422.734452i
    2        -2.349554 + 0.003069i    -101.898386 + 0.506674i
    3        -1.494953 + -0.567574i   -3.465270 + -23.746610i
    4        -1.119892 + -0.004081i   -5.136987 + -0.085096i
    5        -0.733347 + 0.198990i    0.161879 + 1.086157i
    6        -0.681064 + 0.012745i    -0.051000 + 0.059551i
    7        -0.669483 + -0.000007i   -0.000043 + -0.000032i
    8        -0.669473 + 0.000000i    -0.000000 + 0.000000i
```

Imaginary results:

```
>> muller_MM2(i)
Iteration              x0                      f(x0)
    0        0.000000 + 1.000000i     -6.000000 + -3.000000i
    1        -0.281555 + 0.575529i    0.547494 + -1.064816i
    2        -0.097475 + 0.476821i    -0.034081 + -0.066235i
    3        -0.089745 + 0.463636i    -0.000021 + 0.000013i
    4        -0.089749 + 0.463633i    -0.000000 + -0.000000i
```

```
>> muller_MM2(-i)
Iteration              x0                      f(x0)
    0        -0.000000 + -1.000000i   -6.000000 + 3.000000i
    1        -0.281555 + -0.575529i   0.547494 + 1.064816i
    2        -0.097475 + -0.476821i   -0.034081 + 0.066235i
    3        -0.089745 + -0.463636i   -0.000021 + -0.000013i
    4        -0.089749 + -0.463633i   -0.000000 + 0.000000i
```

MM1 and MM2 comparison

Comparison of far guesses:

```
>> muller_MM1(-10, 0, 10)
Iteration              x2                      f(x2)
    0        10.000000 + 0.000000i    -14479.000000 + 0.000000i
    1        2.576349 + 0.000000i     36.729321 + 0.000000i
    2        2.648692 + 0.000000i     35.849215 + 0.000000i
    3        4.012651 + 0.000000i     -105.930189 + 0.000000i
    4        3.277768 + 0.000000i     6.495302 + 0.000000i
    5        3.357446 + 0.000000i     -0.825769 + 0.000000i
    6        3.348882 + 0.000000i     0.008642 + 0.000000i
    7        3.348971 + 0.000000i     0.000001 + 0.000000i
    8        3.348971 + 0.000000i     0.000000 + 0.000000i
```

```
>> muller_MM2(10)
Iteration              x0                      f(x0)
    0        10.000000 + 0.000000i    -14479.000000 + 0.000000i
    1        6.938756 + 2.117614i     -648.697250 + -3499.401367i
    2        5.484462 + 0.016296i     -822.248326 + -13.227802i
    3        4.006911 + -0.887217i    -5.425049 + 186.725981i
    4        3.548608 + -0.048343i    -22.439390 + 6.335830i
    5        3.347409 + 0.001430i     0.151147 + -0.138019i
    6        3.348971 + -0.000000i    0.000000 + 0.000000i
```

```
>> muller_MM1(-10,-1,5i)
Iteration              x2                      f(x2)
    0        0.000000 + 5.000000i     -1374.000000 + -615.000000i
    1        -1.043348 + 5.449093i    -1062.891149 + -2066.947396i
    2        0.014306 + 3.409867i     -329.955983 + -186.380441i
    3        1.042849 + 3.017554i     -223.129532 + 151.199591i
    4        1.446928 + 2.255997i     -39.109623 + 128.828154i
    5        1.664399 + 1.500283i     23.128169 + 63.053201i
    6        1.890270 + 0.818583i     36.365968 + 22.305425i
    7        2.079432 + 0.066251i     34.411019 + 1.045014i
    8        2.671094 + -0.910328i    67.725226 + 2.880044i
    9        1.225071 + 0.113427i     15.572904 + 2.508525i
    10       0.722194 + 0.222466i     5.912766 + 3.130214i
    11       0.338448 + 0.263748i     1.802205 + 1.749462i
    12       0.011322 + 0.366452i     0.293191 + 0.533503i
    13       -0.070945 + 0.435612i    0.074030 + 0.136092i
    14       -0.089984 + 0.461211i    0.010452 + 0.004953i
    15       -0.089764 + 0.463630i    0.000050 + -0.000055i
    16       -0.089749 + 0.463633i    -0.000000 + -0.000000i
```

```
>> muller_MM2(5i)
Iteration              x0                      f(x0)
    0        0.000000 + 5.000000i     -1374.000000 + -615.000000i
    1        -1.103209 + 3.145111i    58.208564 + -367.330914i
    2        -1.435665 + 1.694392i    87.138076 + -8.634025i
    3        -1.338126 + 0.700400i    6.337185 + 18.869733i
    4        -1.061080 + 0.119083i    -3.655273 + 2.101042i
    5        -0.750976 + -0.110867i   -0.275975 + -0.687666i
    6        -0.675357 + 0.000893i    -0.026414 + 0.004066i
    7        -0.669472 + -0.000000i   0.000002 + -0.000001i
    8        -0.669473 + 0.000000i    0.000000 + 0.000000i
```

As we can see, when trying to find roots when we are guessing numbers that are far from polynomials roots, it takes notably less iterations for MM2 to find a root, compared to MM1 method.

Comparison of close guesses:

```
>> muller_MM1(2, 3, 4)
Iteration                x2                    f(x2)
    0        4.000000 + 0.000000i     -103.000000 + 0.000000i
    1        3.292273 + 0.000000i     5.235071 + 0.000000i
    2        3.346157 + 0.000000i     0.271680 + 0.000000i
    3        3.348983 + 0.000000i     -0.001173 + 0.000000i
    4        3.348971 + 0.000000i     0.000000 + 0.000000i
```

```
>> muller_MM2(4)
Iteration                x0                    f(x0)
    0        4.000000 + 0.000000i     -103.000000 + 0.000000i
    1        3.188976 + 0.000000i     13.538081 + 0.000000i
    2        3.349867 + 0.000000i     -0.086768 + 0.000000i
    3        3.348971 + 0.000000i     0.000000 + 0.000000i
```

```
>> muller_MM1(0, 0.5i, i)
Iteration                x2                    f(x2)
    0        0.000000 + 1.000000i     -6.000000 + -3.000000i
    1        -0.070956 + 0.479280i    -0.114288 + 0.043075i
    2        -0.088466 + 0.463972i    -0.004539 + 0.004441i
    3        -0.089750 + 0.463638i    -0.000016 + -0.000014i
    4        -0.089749 + 0.463633i    0.000000 + -0.000000i
```

```
>> muller_MM2(0.5i)
Iteration                x0                    f(x0)
    0        0.000000 + 0.500000i     -0.375000 + 0.375000i
    1        -0.089665 + 0.464843i    -0.005173 + -0.002640i
    2        -0.089749 + 0.463633i    0.000000 + 0.000000i
```

For close guesses, the MM2 algorithm also needs less iterations. From our observations we can see Müller's method MM2 performs better than MM1. Even though I am not able to accurately compare the speed of iterations for both methods, we know from theory that it is slightly faster to calculate values of 1st and 2nd order derivative in one point than values of a polynomial in 3 different points. That makes MM2 method clearly superior.

MM2 and Newton's method comparison

Note: Newton's method for this task is the same as for task 1, but with deleted if statement that didn't allow for choosing values outside of range [2, 12].

Comparison of far guesses:

(reminder: Newton_method(initial guess, left boundary, right boundary)

```
>> Newton_method(10, -5, 15)
Initial guess: 10.000000, interval: [-5.000000, 15.000000]
Iteration:      x:        value of x:
    1        7.73694905  -4535.07094709
    2        6.07442241  -1404.68395738
    3        4.88085488  -423.79407352
    4        4.06907149  -119.50128562
    5        3.58708171  -27.83974040
    6        3.38587830  -3.68040372
    7        3.35004498  -0.10399722
    8        3.34897210  -0.00009139
    9        3.34897115  -0.00000000
```

```
>> muller_MM2(10)
Iteration                x0                    f(x0)
    0        10.000000 + 0.000000i    -14479.000000 + 0.000000i
    1        6.938756 + 2.117614i     -648.697250 + -3499.401367i
    2        5.484462 + 0.016296i     -822.248326 + -13.227802i
    3        4.006911 + -0.887217i    -5.425049 + 186.725981i
    4        3.548608 + -0.048343i    -22.439390 + 6.335830i
    5        3.347409 + 0.001430i     0.151147 + -0.138019i
    6        3.348971 + -0.000000i    0.000000 + 0.000000i
```

```
>> Newton_method(50, -5, 51)
Initial guess: 50.000000, interval: [-5.000000, 51.000000]
Iteration:      x:        value of x:
    1        37.66899827 -3752433.80097108
    2        28.42521727 -1186770.60637651
    3        21.49852146 -375196.03014646
    4        16.31201615 -118533.43174499
    5        12.43410457 -37395.72371503
    6        9.54277515  -11765.05593809
    7        7.39925946  -3679.85937206
    8        5.82907936  -1136.17543320
    9        4.70917407  -340.11765691
    10       3.95955503  -93.90362117
    11       3.53244654  -20.56160150
    12       3.37197307  -2.26796566
    13       3.34939443  -0.04097106
    14       3.34897130  -0.00001421
    15       3.34897115  -0.00000000
```

```
>> muller_MM2(50)
Iteration                x0                    f(x0)
    0        50.000000 + 0.000000i    -11862399.000000 + 0.000000i
    1        33.552778 + 11.623697i   -620154.056843 + -2898616.978118i
    2        25.338597 + 0.001516i    -739837.806805 + -182.353357i
    3        17.123727 + -5.794530i   -38253.978390 + 180587.097066i
    4        13.036413 + -0.002824i   -45810.294659 + 42.480113i
    5        8.946856 + 2.858666i     -2236.361380 + -11129.032359i
    6        6.951136 + 0.008555i     -2733.451126 + -16.174012i
    7        4.944512 + -1.326050i    -87.391057 + 647.302626i
    8        4.093224 + -0.031939i    -125.429795 + 8.124013i
    9        3.189662 + 0.186152i     15.885014 + -13.476449i
    10       3.346308 + -0.001704i    0.257371 + 0.164153i
    11       3.348971 + 0.000000i     0.000000 + -0.000001i
```

Boundaries for Newton's method don't matter too much, they're here just to make sure that method did not diverge. From our tests we can see that MM2 method was faster than Newton's.

Comparison of close guesses:

```
>> Newton_method(3, 2, 4)
Initial guess: 3.000000, interval: [2.000000, 4.000000]
Iteration:      x:          value of x:
    1       3.51020408   -17.75679247
    2       3.36710076   -1.78047210
    3       3.34923547   -0.02558157
    4       3.34897121   -0.00000554
    5       3.34897115   -0.00000000
```

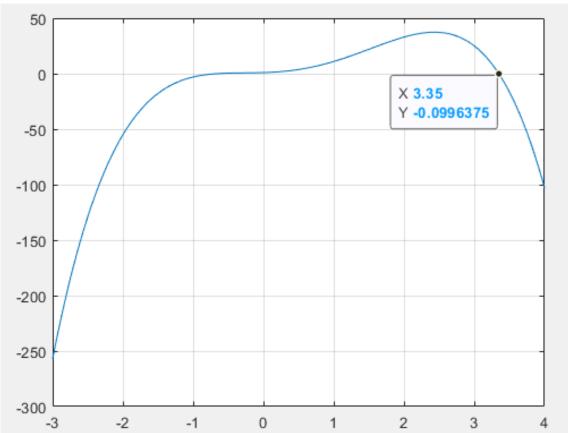
```
>> muller_MM2(3)
Iteration              x0                        f(x0)
    0       3.000000 + 0.000000i     25.000000 + 0.000000i
    1       3.358270 + 0.000000i     -0.906700 + 0.000000i
    2       3.348971 + 0.000000i     0.000018 + 0.000000i
    3       3.348971 + 0.000000i     0.000000 + 0.000000i
```

It is the same as for far guesses.

Conclusions

Out of all 3 methods, MM2 proved the fastest. Compared to MM1, it needed less iterations and even though I am not able to accurately compare the speed of iterations for both methods, we know from theory that it is slightly faster to calculate values of 1st and 2nd order derivative in one point than values of a polynomial in 3 different points. That makes MM2 method clearly superior from method MM1.

If we compare MM2 method to Newton's method, to my surprise, MM2 method proved slightly faster than Newton's method, even though it is stated in literature (*Numerical Methods* by *P. Tatjewski*) that MM2 method is "almost as fast as the Newton's method". It could be the fault of function not being steep, but that is also not the case as at root = 3.349, function is quite steep:



And another great benefit of MM2 is that it can calculate complex roots, which Newton's method cannot.

To sum it up, Müller's MM2 method is the fastest and the most versatile out of those 3.

<u>4.</u> 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).

Background

Laguerre's method

This method is very similar to Müller's MM2 method and is performed in the same way, with the exception for calculating z_- and z_+:

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

so, if we want to compare it to MM2 method, z_- and z_+ would be

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

and we choose the z that has smaller absolute value, just as previously.

Laguerre's method is slightly more complex, as it takes into consideration the order of a polynomial (n in the equation). In case of polynomials with real roots only, Laguerre's method is globally convergent. It can also usually find imaginary roots, but situations of divergence may happen. This method is regarded as one of the best methods for polynomial root finding.

MATLAB implementation

Code for Laguerre's method is very similar in construction to MM2, however equations for z_- and z_+ had to be changed, and now z_{+-} is subtracted instead of being added.

```
function Laguerre_method(x0)     % x0 - initial guess
    accuracy = 1e-6;
    err = inf;
    iter_count = 0;
    n = 4;  % order of polynomial
    % printout of labels and iteration 0
    fprintf("Iteration\t\t\t\tx0\t\t\t\t\tf(x0)\n");
    fprintf("\t%d\t\t%.6f + %.6fi\t%.6f + %.6fi\n", iter_count, real(x0), imag(x0), real(calc_f(x0)), imag(calc_f(x0)));
    while err > accuracy && iter_count < 2000   % loop until error is small enough or iterates too many times
        % just as for MM2, we only have to calculate z- and z+
        z_plus = n*calc_f(x0)/(calc_df(x0) + sqrt((n-1)*((n-1)*calc_df(x0).^2 - n*calc_f(x0)*calc_ddf(x0))));
        z_minus = n*calc_f(x0)/(calc_df(x0) - sqrt( (n-1) * ((n-1)*calc_df(x0).^2 - n*calc_f(x0)*calc_ddf(x0))));
        % choose the smaller one out of z- and z+ and SUBSTRACT it from x0
        if abs(z_plus) < abs(z_minus)
            x0 = x0 - z_plus;
        else
            x0 = x0 - z_minus;
        end
        err = abs(calc_f(x0));  % calculate error
        iter_count = iter_count + 1;
        % print results of iteration
        fprintf("\t%d\t\t%.6f + %.6fi\t%.6f + %.6fi\n", iter_count, real(x0), imag(x0), real(calc_f(x0)), imag(calc_f(x0)));
    end
end
```

Results:

```
>> Laguerre_method(100)
Iteration            x0                      f(x0)
    0       100.000000 + 0.000000i  -194949799.000000 + 0.000000i
    1       3.322749 + 0.000000i    2.483164 + 0.000000i
    2       3.348971 + 0.000000i    -0.000002 + 0.000000i
    3       3.348971 + 0.000000i    0.000000 + 0.000000i
>> Laguerre_method(-100)
Iteration            x0                      f(x0)
    0       -100.000000 + 0.000000i -204950199.000000 + 0.000000i
    1       -1.976191 + 0.000000i   -52.517391 + 0.000000i
    2       -0.739919 + 0.000000i   -0.367362 + 0.000000i
    3       -0.669351 + 0.000000i   0.000539 + 0.000000i
    4       -0.669473 + 0.000000i   -0.000000 + 0.000000i

>> Laguerre_method(-2i)
Iteration            x0                      f(x0)
    0       -0.000000 + -2.000000i  -51.000000 + 36.000000i
    1       -0.361327 + -0.673005i  1.149668 + 1.919048i
    2       -0.068590 + -0.452827i  -0.002125 + -0.112969i
    3       -0.089753 + -0.463630i  0.000024 + 0.000009i
    4       -0.089749 + -0.463633i  0.000000 + -0.000000i

Iteration            x0                      f(x0)
    0       0.000000 + 2.000000i    -51.000000 + -36.000000i
    1       -0.361327 + 0.673005i   1.149668 + -1.919048i
    2       -0.068590 + 0.452827i   -0.002125 + 0.112969i
    3       -0.089753 + 0.463630i   0.000024 + -0.000009i
    4       -0.089749 + 0.463633i   0.000000 + 0.000000i

>> Laguerre_method(218719234)
Iteration            x0                      f(x0)
    0       218719234.000000 + 0.000000i    -4576968206490399275309487519432704.000000 + 0.000000i
    1       0.625000 + 0.000000i    5.118652 + 0.000000i
    2       -0.228697 + 0.000000i   0.738840 + 0.000000i
    3       -0.354954 + -0.782955i  1.038745 + 3.216162i
    4       -0.079734 + -0.428257i  0.118756 + -0.114804i
    5       -0.089745 + -0.463653i  -0.000092 + 0.000034i
    6       -0.089749 + -0.463633i  0.000000 + 0.000000i
```

As we can see, Laguerre's method is very fast and finds the roots very quickly and works globally.

Laguerre and Müller's MM2 comparison

```
>> muller_MM2(10)
Iteration              x0                      f(x0)
    0        10.000000 + 0.000000i    -14479.000000 + 0.000000i
    1        6.938756 + 2.117614i     -648.697250 + -3499.401367i
    2        5.484462 + 0.016296i     -822.248326 + -13.227802i
    3        4.006911 + -0.887217i    -5.425049 + 186.725981i
    4        3.548608 + -0.048343i    -22.439390 + 6.335830i
    5        3.347409 + 0.001430i     0.151147 + -0.138019i
    6        3.348971 + -0.000000i    0.000000 + 0.000000i
```

```
>> Laguerre_method(10)
Iteration              x0                      f(x0)
    0        10.000000 + 0.000000i    -14479.000000 + 0.000000i
    1        3.339178 + 0.000000i     0.939990 + 0.000000i
    2        3.348971 + 0.000000i     -0.000000 + 0.000000i
```

```
>> muller_MM2(-200)
Iteration              x0                      f(x0)
    0        -200.000000 + 0.000000i -3239800399.000000 + 0.000000i
    1        -133.127670 + 47.284482i    -169955677.405875 + 791896215.411120i
    2        -99.693584 + 0.000304i  -202464598.936240 + 2451.207536i
    3        -66.259443 + -23.638615i    -10615438.207217 + -49485084.836694i
    4        -49.546335 + -0.000431i -12648437.768286 + -435.270240i
    5        -32.833188 + 11.812482i -661760.268713 + 3090777.348055i
    6        -24.484270 + 0.000860i  -789190.894349 + 108.489263i
    7        -16.135374 + -5.893313i -40971.318064 + -192686.817001i
    8        -11.975009 + -0.001403i -49019.678256 + -22.119022i
    9        -7.815066 + 2.923801i   -2477.642177 + 11932.701688i
    10       -5.759200 + 0.001981i   -3000.075844 + 3.902409i
    11       -3.704945 + -1.425553i  -139.638384 + -723.191606i
    12       -2.714195 + -0.002098i  -176.109970 + -0.514759i
    13       -1.726458 + 0.663663i   -6.478194 + 41.364470i
    14       -1.281516 + 0.003324i   -9.268479 + 0.101901i
    15       -0.830915 + -0.260837i  0.030731 + -2.046043i
    16       -0.713879 + -0.015602i  -0.215245 + -0.084461i
    17       -0.669343 + 0.000218i   0.000575 + 0.000963i
    18       -0.669473 + -0.000000i  0.000000 + -0.000000i
```

```
>> Laguerre_method(-200)
Iteration              x0                      f(x0)
    0        -200.000000 + 0.000000i -3239800399.000000 + 0.000000i
    1        -2.022147 + 0.000000i   -57.383601 + 0.000000i
    2        -0.745912 + 0.000000i   -0.404097 + 0.000000i
    3        -0.669322 + 0.000000i   0.000668 + 0.000000i
    4        -0.669473 + 0.000000i   -0.000000 + 0.000000i
```

```
>> muller_MM2(-600000)
Iteration              x0                      f(x0)
    0        -600000.000000 + 0.000000i  -259201079998200015749120.000000 + 0.000000i
    1        -399999.791668 + 141421.503550i -136000566662636419153920.000000 + 63357031580274427691008.000000i
    2        -299999.687502 + 0.000000i  -16200067499677609623552.000000 + 22838662037.176796i
    3        -199999.583337 + -70710.751774i -8500035415781925191680.000000 + -39598144736899099525120.000000i
    4        -149999.531255 + -0.000000i -10125042186773763194880.000000 + -4281070423.160099i

    ...

    38       -0.939115 + -0.007404i  -2.164293 + -0.092281i
    39       -0.628778 + 0.094275i   0.245052 + 0.333947i
    40       -0.667240 + 0.000804i   0.009842 + 0.003522i
    41       -0.669473 + -0.000000i  -0.000000 + -0.000000i
```

```
>> Laguerre_method(-600000)
Iteration              x0                      f(x0)
    0        -600000.000000 + 0.000000i  -259201079998200015749120.000000 + 0.000000i
    1        -2.070374 + 0.000000i   -62.828517 + 0.000000i
    2        -0.752284 + 0.000000i   -0.444176 + 0.000000i
    3        -0.669287 + 0.000000i   0.000822 + 0.000000i
    4        -0.669473 + 0.000000i   -0.000000 + 0.000000i
```

There is a vast difference in convergence time between Müller's MM2 method and Laguerre's method. Laguerre's method requires far fewer iterations than MM2 one. It is true for values both far and near to the root. It comes as no surprise that Laguerre's method is regarded as one of the best ones for polynomial root finding.

Code:

Task 1

calc_f(x)

```
% calculates f(x) as per task description
function y = calc_f(x)
    y = 1.2 * sin(x) + 2 * log(x+2) - 5;
end
```

calc_fd(x)

```
function y = calc_df(x)
    y = 2/(x+2) + 1.2 * cos(x);
end
```

plot_task1()

```
function plot_task1()
    x = 2:0.01:12;  % x = left border, step, right border
    plot(x, calc_f(x));
end
```

false_position(a, b)

```
function false_position(a, b)
    % initial conditions
    if (a < 2 || a > 12 || b < 2 || b > 12)
        error("a and b must be within [2,12] interval");
    end
    if ( calc_f(a) * calc_f(b) >= 0)
        error("a and b must have different signs!");
    end

    % initial printout:
    fprintf("Interval: [%f, %f]\n", a, b);
    fprintf("Iteration:\t\tc:\t\tvalue of c:\t\tinterval length:\n");
    err = inf;
    accuracy = 1e-6;
    iter_count = 0;
    % loop until error is small enough or iterates too many times
    while err > accuracy && iter_count < 2000
        a_val = calc_f(a);
        b_val = calc_f(b);
        % calculate c as per equation:
        c = (a * b_val - b * a_val) / (b_val - a_val);
        c_val = calc_f(c);
        % select new interval:
        if a_val * c_val < 0
            b = c;
        elseif c_val * b_val < 0
            a = c;
        end
        iter_count = iter_count + 1;
        err = abs(c_val); % calculate error
        % printing results:
        small_interval = abs(a-b);
        fprintf("\t%d\t\t%.8f\t%.8f\t\t%.8f\n", iter_count, c, c_val,
small_interval);
    end
end
```

Newton_method(init_point, a, b)

```
function Newton_method(init_point, a, b)
    % initial conditions
    if init_point < a || init_point > b
        error("Initial guess should be between a and b.");
    end
    if ( a < 2 || a > 12 || b < 2 || b > 12)
        error("a and b must be within [2,12] interval");
    end

    % initial printout
    fprintf("Initial guess: %f, interval: [%f, %f]\n", init_point, a, b);
    fprintf("Iteration:\t\tx:\t\tvalue of x:\n");
    accuracy = 1e-6;
    err = inf;
    iter_count = 0;
    curr_x = init_point;
    % loop until result is satisfactory or can't find root
    while err > accuracy && iter_count < 2000
        % calculate x_n+1 as per equation
        new_x = curr_x - calc_f(curr_x)/calc_df(curr_x);
        if new_x < a || new_x > b
            error("The method diverged!");
        end
        curr_x = new_x;
        err = abs(calc_f(new_x));
        iter_count = iter_count + 1;
        fprintf("\t%d\t\t%.8f\t%.8f\t\n", iter_count, new_x, calc_f(new_x));
    end

end
```

Task 2

calc_f(x)

```
function y = calc_f(x)
    y = -2 * x.^4 + 5 * x.^3 + 5 * x.^2 + 2 * x + 1;
end
```

calc_df(x)

```
function y = calc_df(x)
    y = -8*x^3 + 15*x^2 + 10*x + 2;
end
```

calc_ddf(x)

```
function y = calc_ddf(x)
    y = -24*x^2 + 30*x + 10;
end
```

plot_task2()

```
function plot_task2()
    r = roots([-2 5 5 2 1]);    % finds roots for -2*x.^4 + 5*x.^3 + 5*x.^2 + 2*x
+ 1
    plot(real(r),imag(r),'bo')  % plots with blue circles
    grid on
    xlabel('Real')
    ylabel('Imaginary')
end
```

plot_real()

```
function plot_real()
    x = -3:0.01:4;  % x = left border, step, right border
    plot(x, calc_f(x));
    grid on;
end
```

muller_MM1(x0, x1, x2)

```
function muller_MM1(x0, x1, x2)
    accuracy = 1e-6;
    err = inf;
    iter_count = 0;
    % printout of labels and iteration 0
    fprintf("Iteration\t\t\t\tx2\t\t\t\t\tf(x2)\n");
    fprintf("\t%d\t\t%.6f + %.6fi\t%.6f + %.6fi\n", iter_count, real(x2),
imag(x2), real(calc_f(x2)), imag(calc_f(x2)));
    while err > accuracy && iter_count < 2000   % loop until error is small enough
or iterates too many times
        % assuming x2 is the actual approximation, we assign:
        z0 = x0 - x2;
        z1 = x1 - x2;
        % calculate a, b and c according to the equations
        a = (calc_f(x0)*z1 + calc_f(x2)*z0 - calc_f(x2)*z1 - calc_f(x1)*z0)/...
            (z0*z1*(z0-z1));
        b = (calc_f(x1) - calc_f(x2) - a*z1^2) / z1;
        c = calc_f(x2);
        % now we need to calculate z- and z+
        z_plus = -2*c/(b + sqrt(b.^2 - 4*a*c));
        z_minus = -2*c/(b - sqrt(b.^2 - 4*a*c));
        % choose the smaller one out of z- and z+ and add it to x2
        if abs(z_plus) < abs(z_minus)
            x3 = x2 + z_plus;
```

```
        else
            x3 = x2 + z_minus;
        end
        % reassigning values so that x3 and 2 closest numbers to x3 stay
        tmp0 = abs(x3 - x0);
        tmp1 = abs(x3 - x1);
        tmp2 = abs(x3 - x2);
        if tmp0 > tmp1 && tmp0 > tmp2
            x0 = x1;
            x1 = x2;
        elseif tmp1 > tmp0 && tmp1 > tmp2
            x1 = x2;
        end
        x2 = x3;
        err = abs(calc_f(x2));  % calculate error
        iter_count = iter_count + 1;
        % print results of iteration
        fprintf("\t%d\t\t%.6f + %.6fi\t%.6f + %.6fi\n", iter_count, real(x2),
imag(x2), real(calc_f(x2)), imag(calc_f(x2)));
    end
end
```

muller_MM2(x0)

```
function muller_MM2(x0)     % x0 - initial guess
    accuracy = 1e-6;
    err = inf;
    iter_count = 0;
    % printout of labels and iteration 0
    fprintf("Iteration\t\t\t\tx0\t\t\t\t\tf(x0)\n");
    fprintf("\t%d\t\t%.6f + %.6fi\t%.6f + %.6fi\n", iter_count, real(x0),
imag(x0), real(calc_f(x0)), imag(calc_f(x0)));
    while err > accuracy && iter_count < 2000   % loop until error is small enough
or iterates too many times
        % there is no need to calculate a, b and c
        % so we just need to calculate z- and z+
        z_plus = -2*calc_f(x0)/(calc_df(x0) + sqrt(calc_df(x0).^2 -
2*calc_f(x0)*calc_ddf(x0)));
        z_minus = -2*calc_f(x0)/(calc_df(x0) - sqrt(calc_df(x0).^2 -
2*calc_f(x0)*calc_ddf(x0)));
        % choose the smaller one out of z- and z+ and add it to x0
        if abs(z_plus) < abs(z_minus)
            x0 = x0 + z_plus;
        else
            x0 = x0 + z_minus;
        end
        err = abs(calc_f(x0));  % calculate error
        iter_count = iter_count + 1;
        % print results of iteration
        fprintf("\t%d\t\t%.6f + %.6fi\t%.6f + %.6fi\n", iter_count, real(x0),
imag(x0), real(calc_f(x0)), imag(calc_f(x0)));
    end
end
```

Laguerre_method(x0)

```
function Laguerre_method(x0)     % x0 - initial guess
    accuracy = 1e-6;
    err = inf;
    iter_count = 0;
    n = 4;  % order of polynomial
```

```
    % printout of labels and iteration 0
    fprintf("Iteration\t\t\t\tx0\t\t\t\t\tf(x0)\n");
    fprintf("\t%d\t\t%.6f + %.6fi\t%.6f + %.6fi\n", iter_count, real(x0),
imag(x0), real(calc_f(x0)), imag(calc_f(x0)));
    while err > accuracy && iter_count < 2000   % loop until error is small enough
or iterates too many times
        % just as for MM2, we only have to calculate z- and z+
        z_plus = n*calc_f(x0)/(calc_df(x0) + sqrt((n-1)*((n-1)*calc_df(x0).^2 -
n*calc_f(x0)*calc_ddf(x0))));
        z_minus = n*calc_f(x0)/(calc_df(x0) - sqrt( (n-1) * ((n-1)*calc_df(x0).^2
- n*calc_f(x0)*calc_ddf(x0))));
        % choose the smaller one out of z- and z+ and SUBSTRACT it from x0
        if abs(z_plus) < abs(z_minus)
            x0 = x0 - z_plus;
        else
            x0 = x0 - z_minus;
        end
        err = abs(calc_f(x0));  % calculate error
        iter_count = iter_count + 1;
        % print results of iteration
        fprintf("\t%d\t\t%.6f + %.6fi\t%.6f + %.6fi\n", iter_count, real(x0),
imag(x0), real(calc_f(x0)), imag(calc_f(x0)));
    end
end
```

Newton_method(init_point, a ,b)

```
function Newton_method(init_point, a, b)
    % initial conditions
    if init_point < a || init_point > b
        error("Initial guess should be between a and b.");
    end

    % initial printout
    fprintf("Initial guess: %f, interval: [%f, %f]\n", init_point, a, b);
    fprintf("Iteration:\t\tx:\t\tvalue of x:\n");
    accuracy = 1e-6;
    err = inf;
    iter_count = 0;
    curr_x = init_point;
    % loop until result is satisfactory or can't find root
    while err > accuracy || iter_count > 2000
        % calculate x_n+1 as per equation
        new_x = curr_x - calc_f(curr_x)/calc_df(curr_x);
        if new_x < a || new_x > b
            error("The method diverged!");
        end
        curr_x = new_x;
        err = abs(calc_f(new_x));
        iter_count = iter_count + 1;
        fprintf("\t%d\t\t%.8f\t%.8f\t\n", iter_count, new_x, calc_f(new_x));
    end

end
```