Author: Sławomir Batruch

Numerical Methods - Project 2

Task 1

The task description is as follows:

I Find all zeros of the function

$$f(x) = 1.2\sin(x)+2\ln(x+2)-5$$

in the interval [2, 12] using:
a) the false position method,
b) the Newton's method.

Introduction

For this task, I will need to do the following things:

1. Design of the function evaluation functions.
2. False position method design
3. Newton method design

Below are shown theoretical explanations regarding above-mentioned task parts.

False position method - theoretical background

This method is also known as *regula falsi method*. It's similar to bisection method

We begin by creating a secant starting from the point (f(a), a), ending at point (f(b), b). Figure shown below:

Fig. 6.1.

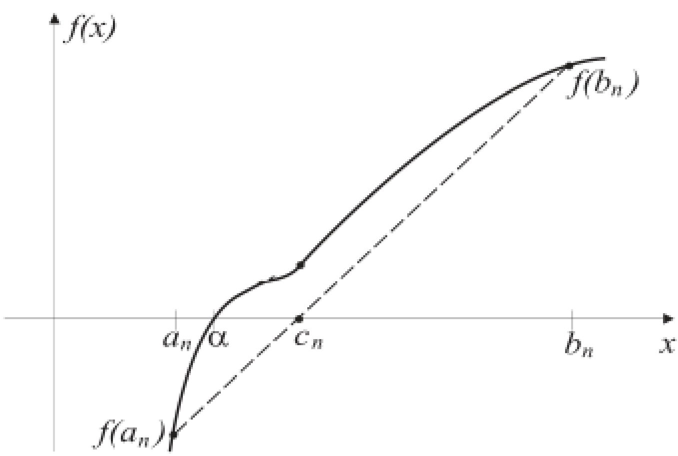


Figure 6.1. A construction of two subintervals by the secant line

Source: Numerical Methods, Piotr Tatjewski

cn is the point where the secant line intersects with the x-axis.

Then:

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

$$c_n = b_n - \frac{f(b_n)(b_n - a_n)}{f(b_n) - f(a_n)} = \frac{a_n f(b_n) - b_n f(a_n)}{f(b_n) - f(a_n)}. \qquad (6.4)$$

(f(c), c) will be one of our points for the secant line in the next iteration.
Then the 2nd point for the secant line is chosen as follows:

2. Products $f(a_n)f(c_n)$ and $f(c_n)f(b_n)$ are evaluated, then the next interval is chosen as the subinterval corresponding to the product with the negative value. The endpoints of this interval are denoted a_{n+1}, b_{n+1}.

Since we assume there exists a zero point for the function within the specified [a,b] interval, we simply check on which side the zero point is in relation to the c point.

Then the above-mentioned tests are repeated until we obtain a result satisfying our chosen accuracy. We repeat steps for a secant with endpoints [a,c] or [c,b], depending on the result of above-mentioned shown evaluation

Convergence

Usually the method has rather “ok” times of convergence, but there exists an edge case with really long convergence time. It happens when the chosen “a or b” endpoint for a secant line is unchanged every iteration. As we can see from the shown figure 6.1 from Numerical Methods book, this is an example where convergence will be slow, as the point a will be always chosen as one of the endpoints for the secant line.

Newton-Rhapson method - theoretical background

This method uses first order Taylor approximation to find a zero of a function.

The way the zero is calculated is described in 3 equations shown below. It uses a Taylor approximation, which is rearranged to get a general equation for next iteration approximation:

$$f(x) \approx f(x_n) + f'(x_n)(x - x_n). \qquad (6.7)$$

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

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

which leads to the iteration formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \qquad (6.8)$$

So we take some initial guess x_n and then have a recursive formula for an iterative way of finding the zero of some function.
Note that for this method we need a value of the first order derivative of our function.

Convergence

In accordance with the information from *Numerical Methods* book, I gathered the following information about convergence:

- The method is locally convergent, which means that if an initial point is too far from the root, a divergence may occur
- Convergence usually is fast, as the convergence rate is quadratic
- Method is particularly effective if the derivative of the function at the root is far from zero.
    - But if the derivative is close to zero, the method is sensitive to numerical errors when close to the root.

We should stop executing the method when the absolute value of the current and previous zero_point guess. In other words, we stop when
abs(x_prev - x_curr) < accuracy
Accuracy is some arbitrary accuracy chosen by the user, usually 10 to *some negative power*.

MATLAB - function value computation

Here I show two methods: one that evaluates the value of our task function at some x coordinate, second that evaluates the value of a 1st derivative of our task function at some coordinate x. f(x) value calculation:

```
function y = eval_f_x(x)
    if x < 2 || x > 12 % We shouldn't take values outside interval [2,12]
        error('ERR: Tried to evaluate function outside defined interval');
    end
    y = 1.2 * sin(x) + 2 * log(x+2) - 5; % As in the task description
end
```

f'(x) == d(f(x))/dx. First derivative value calculation:

```
function y = eval_f_dx(x)
    if x < 2 || x > 12 % We shouldn't take values outside interval [2,12]
        error('ERR: Tried to evaluate function outside defined interval');
    end
    y = (6*cos(x))/5 + 2/(x + 2); % Derivative calculated beforehand using:
    % syms x
    % diff(1.2 * sin(x) + 2 * log(x+2) - 5)
end
```

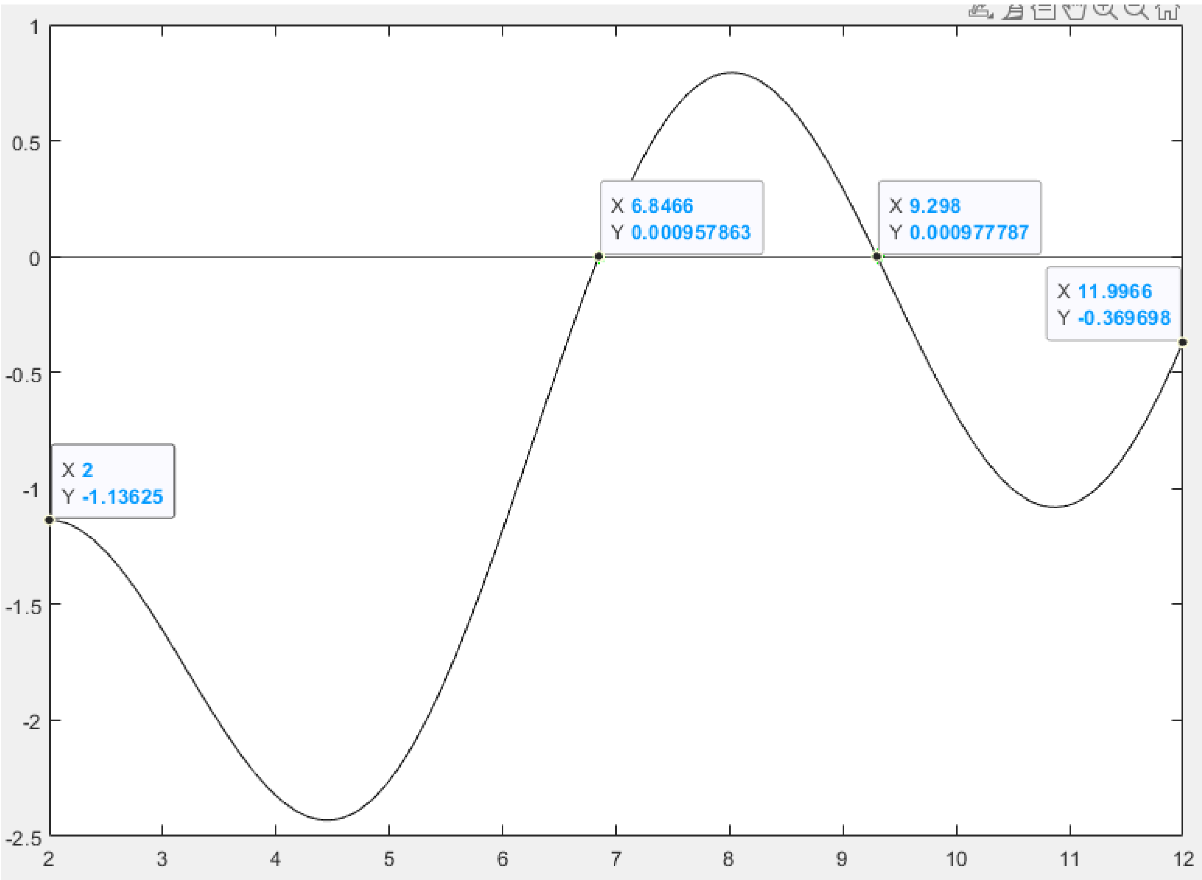
Derivative calculation was done as follows:

```
>> syms x
>> diff(1.2 * sin(x) + 2 * log(x+2) - 5)

ans =

(6*cos(x))/5 + 2/(x + 2)
```

These functions are non-complex, simple evaluations of a function value at a given point x (specified as the method argument)

MATLAB - Plotting the function

This is how the plot of the function looks like with marked zero points and the values at interval boundaries.



And this is the algorithm used to make this plot. Every step explained in comments:

```
function plot_f_x(step)
    boundaries = [2, 12];
    interval = abs(boundaries(1) - boundaries(2));

    % 'step' is inverted due to the fact that for e.g. step = 0.01
    % We  will need to iterate 100 times for every single whole digit
    % In other words, we will need to iterate 1/step times for every [2,3]
    % [3,4] etc.
    f_x_arr = zeros(1, interval * 1/step); % Preallocate the array.
    for i = 0 : size(f_x_arr, 2) % Iterate from 0 until the f_x_arr is filled
        f_x_arr(i+1) = eval_f_x(2 + i * step); % start from f(2)
        % Then next value will be f(2 + current_iteration * step)
    end
    x_range = 2:step:12; % Define the x_axis values. array of 2 to 12 with
    % difference between every element being 'step'
    mark = (abs(f_x_arr) < 1e-03); %  Define the condition for 0 point marking
    % We cannot use == 0, because our data is generated the way that 0
    % might not be present in the plot at any x considered by this algo
    plot(x_range, f_x_arr, 'k'); % Plot a dark line of f(x)
    % x axis is x_range, f_x_arr are corresponding values
    hold on; % Used for adding the zero points
    plot(x_range(mark), f_x_arr(mark), '*g'); % Add points that satisfy
    % criteria defined by 'mark'. *g == mark points with a green star

    % Added from console after the plot is created:
    % hold on
    % yline(0)
end
```

Function plot - appendix

Since at the boundaries, i.e. at a = 2 and b = 12, f(a) and f(b) are negative, this means f(a) * f(b) < 0 is not fulfilled. After initial testing the regula falsi method doesn't work, as it tries to evaluate the function value outside the interval.
To fix this, I decided to divide the given [2,12] interval into two subintervals:
[2,8] and [8,12]
x = 8 is a local maximum for this function.
That way I will be able to find two zero points that this function has in the [2,12] interval, but also this will allow the Regula Falsi method to work.

MATLAB - Newton's method

Since Newton's method is a bit more trivial than regula falsi method, I decided to show it first. This is the code used to realise Newton's method, with necessary comments:

```
% It is advised to choose the following intervals for considered function:
% [4.5, 8] ; [8, 10.5]
% Otherwise method will probably diverge
% Highest intervals should be [a, 8] ; [8, b]
function [iter_count, zero_point] = Newton(initial_guess, a, b)
    if (initial_guess < a || initial_guess > b) % The guess must be chosen in the interval
        error("Incorrect initial guess chosen");
    end
    if ( (a < 2 || a > 12) || (b < 2 || b > 12) ) % a and b need to be within [2,12] interval
        error("Chosen interval is not within [2,12] interval");
    end
    precision = 1e-05; % We stop when our zero point is within this tolerance
    zero_point = initial_guess;
    err = inf;
    iter_count = 0;
    % Display initial guess and the chosen interval:
    fprintf("Initial guess: %.5f \t ; chosen interval: [%d, %d]\n\n", initial_guess, a, b);
    % Set up for pretty formatting:
    fprintf("Iteration\t\t x value\t\t f(x)\t\t abs(f(zero_point))\n");
```

```
while err > precision
    % As in the formulas from the report
    previous_zero = zero_point; % Store the previous zero point
    zero_point = previous_zero - (eval_f_x(zero_point)/ eval_f_dx(zero_point));
    % xn+1     =     xn         -        f(x)          /         f'(x)

    err = abs(eval_f_x(zero_point)); % Distance from 0
    iter_count = iter_count + 1;

    % Print the values of the iteration. They are lined up with the
    % above-mentioned fprintf (pretty formatting)
    fprintf("\t%d \t\t %.10f \t %.10f \t %.10f\n", iter_count, zero_point, eval_f_x(zero_point), err);
end
format long; % Display the zero point in the 'long' format
```

Printing out algorithm values at given iteration is done within this function as it can be seen from the use of fprintf functions.

MATLAB - Newton's method output

Below I show different outputs from the Newton's method:

First - typical behavior. We check for the small intervals. First zero:

```
>> [iter, x0] = Newton(5.5, 4.5, 8)
Initial guess: 5.50000   ; chosen interval: [4.500000e+00, 8]

Iteration        x value         f(x)          abs(f(zero_point))
    1         7.1264349645    0.3185201388    0.3185201388
    2         6.8132979595   -0.0407199325    0.0407199325
    3         6.8455582720   -0.0003348658    0.0003348658
    4         6.8458280438   -0.0000000242    0.0000000242
```

2nd zero:

```
>> [iter, x0] = Newton(10, 8, 10.5)
Initial guess: 10.00000      ; chosen interval: [8, 1.050000e+01]

Iteration        x value         f(x)          abs(f(zero_point))
    1         9.1871025331    0.1120559594    0.1120559594
    2         9.3005782798   -0.0016354582    0.0016354582
    3         9.2989650431   -0.0000002147    0.0000002147
```

Now abnormal outputs when divergence occurs. Incorrect initial_guess. [8, 10.5]:

```
>> [iter, x0] = Newton(10.5, 8, 10.5)
Initial guess: 10.50000      ; chosen interval: [8.000, 10.500]

Iteration        x value         f(x)          abs(f(zero_point))
    1         8.0546291764    0.7919915567    0.7919915567
Error using Newton (line 26)
Local divergence detected @ iteration 2 ; x = 27.73078 outside [8.000, 10.500] interval
```

[5.5, 8]

```
>> [iter, x0] = Newton(8, 5.5, 8)
Initial guess: 8.00000   ; chosen interval: [5.500, 8.000]

Iteration        x value         f(x)          abs(f(zero_point))
Error using Newton (line 26)
Local divergence detected @ iteration 1 ; x = -23.19690 outside [5.500, 8.000] interval
```

Abnormal outputs: a situation **when interval constraints are turned off**:

This case is when a zero is found (method converges), but outside defined interval:

```
>> [iter, x0] = Newton(11, 2, 12)
Initial guess: 11.00000      ; chosen interval: [2, 12]

Iteration        x value         f(x)          abs(f(zero_point))
    1         17.7234842988   -0.1196549171    0.1196549171
    2         17.9172093761    0.0195436089    0.0195436089
    3         17.8932458723    0.0002768587    0.0002768587
    4         17.8928964724    0.0000000595    0.0000000595
```

Furthermore, note that the zeros of the function (computed in Wolfram Alpha):

Input interpretation

roots	$1.2\sin(x) + 2\log(x + 2) - 5 = 0$

Results

$x \approx 6.84582806330440\ldots$

$x \approx 9.29896483133585\ldots$

$x \approx 12.2964217705140\ldots$

$x \approx 16.4750412141317\ldots$

$x \approx 17.8928963971917\ldots$

Which shows that the found 0 is the furthest one. Despite zero at x = 12.3 and at x = 16.5 was closer to initial_guess, Newton's method did not manage to find them. Calling function `[iter, x0] = Newton(2, 2, 12)` resulted in:

```
207688       32525.4930410981    15.1177446410    15.1177446410
207689       32540.5978315210    15.7579625355    15.7579625355
207690       32527.4645294514    15.1163578899    15.1163578899
207691       32512.3478948489    15.7722461051    15.7722461051
207692       32525.4923075088    15.1184790375    15.1184790375
207693       32540.5905105070    15.7491791189    15.7491791189
207694       32527.4623629369    15.1141929737    15.1141929737
207695       32512.3261010742    15.7983963989    15.7983963989
Operation terminated by user during Newton (line 33)
```

As it can be seen the results are repeating. This is an example of complete divergence. Since no iteration limit was implemented in code, I terminated the code by CTRL + C shortcut.

Newton’s method conclusions

The method doesn’t seem to have good properties. It requires proper choice of the 0 point guesses, which might not always be possible. It is hard to make the method diverge if the wrong guess is picked.

The positive aspect is that the zero point might be found despite incorrect initial guesses, but it is that the point might be found outside the defined interval. As it can be seen, when interval constraints are removed, there is a chance a root outside the interval will be found, but also there is a possibility the method will completely diverge. These two cases are described in Newton's method output paragraph. Furthermore, if the guess is rather close, the number of iterations is relatively small.

MATLAB - Regula Falsi method

Here is the code used to realise the regula falsi (false position method). Note that despite eval_f_x has handling for incorrect x argument, we need handling for incorrect x argument also inside this function, as the interval is different than the global interval of [2, 12]:

```
% It is advised to choose the following intervals for considered function:
% [2, 8] ; [8, 12]
% It will also work for [2, 12], but it will only find one zero point
function [iter_count, c] = regula_falsi(a, b) % a,b - interval to operate on. c is our zero point
    if ( (a < 2 || a > 12) || (b < 2 || b > 12)) % a and b need to be within [2,12] interval
        error("Chosen interval is not within [2,12] interval");
    end
    if ( eval_f_x(a) * eval_f_x(b) >= 0) % Otherwise the method might diverge
        % We are unsure if there are any zero points in the interval if
        % above is true
        error("Function has no different signs at interval boundaries. f(a) * f(b) >= 0");
    end
    fprintf("Chosen interval: [%.3f, %.3f]\n\n", a, b);
    fprintf("Iteration\t\t x value\t\t f(x)\t\t\t abs(f(x))\t\t interval width abs(a-b)\n");
    iter_count = 0;
    err = inf;
    accuracy = 1e-05;
    while err > accuracy
        % we will use the formula from the book:
        % Calculate c point:
        c = (a * eval_f_x(b) - b * eval_f_x(a)) / (eval_f_x(b) - eval_f_x(a));
       %c = (a *     f(b)    - b *    f(a)    ) / (   f(b)     -    f(a)     )

        % Define new interval for next iteration. Per book formula:
        if (eval_f_x(a) * eval_f_x(c) < 0)
            % a = a. unchanged
            b = c;
        elseif (eval_f_x(c) * eval_f_x(b) < 0)
            a = c;
            % b = b. unchanged
        end

        err = abs(eval_f_x(c)); % Distance from 0
        iter_count = iter_count + 1;

        fprintf("\t%d \t\t %.10f \t %.10f \t\t %.10f \t\t %.10f\n", iter_count, c, eval_f_x(c), err, abs(a-b));
    end
    format long; % Display the zero point in proper format
end
```

MATLAB - Regula Falsi algorithm output

These are typical algorithm outputs for the Regula Falsi method:

```
>> [iter_count, x0] = regula_falsi(4.5, 8)
Chosen interval: [4.500, 8.000]

Iteration        x value        f(x)              abs(f(x))        interval width abs(a-b)
    1          7.1391852836   0.3314145782       0.3314145782       2.6391852836
    2          6.8223750154   -0.0292887859      0.0292887859       0.3168102683
    3          6.8480997217   0.0028176659       0.0028176659       0.0257247063
    4          6.8458421185   0.0000174440       0.0000174440       0.0234671031
    5          6.8458281501   0.0000001077       0.0000001077       0.0234531347
```

2nd zero point:

```
>> [iter_count, x0] = regula_falsi(8, 10.5)
Chosen interval: [8.000, 10.500]

Iteration        x value        f(x)              abs(f(x))        interval width abs(a-b)
    1          9.1026521141   0.1942685507       0.1942685507       1.3973478859
    2          9.3291627029   -0.0306760961      0.0306760961       0.2265105889
    3          9.2982730566   0.0007010793       0.0007010793       0.0308896464
    4          9.2989632427   0.0000016101       0.0000016101       0.0301994602
```

Now 2nd usage case - we check if maximum intervals are still ok for the algorithm:

```
>> [iter_count, x0] = regula_falsi(2, 8)
Chosen interval: [2.000, 8.000]

Iteration         x value          f(x)              abs(f(x))          interval width abs(a-b)
    1          5.5348614171    -1.7774128145      1.7774128145          2.4651385829
    2          7.2398761724     0.4278040466      0.4278040466          1.7050147553
    3          6.9091095142     0.0771644133      0.0771644133          1.3742480971
    4          6.8519304188     0.0075612822      0.0075612822          1.3170690017
    5          6.8463512172     0.0006492022      0.0006492022          1.3114898001
    6          6.8458723689     0.0000549876      0.0000549876          1.3110109518
    7          6.8458318116     0.0000046520      0.0000046520          1.3109703945
```

2nd zero point:

```
>> [iter_count, x0] = regula_falsi(8, 12)
Chosen interval: [8.000, 12.000]

Iteration         x value          f(x)              abs(f(x))          interval width abs(a-b)
    1          10.7367245955   -1.0710430306      1.0710430306          2.7367245955
    2          9.1637493944     0.1350330299      0.1350330299          1.5729752011
    3          9.3398606815    -0.0415736622      0.0415736622          0.1761112871
    4          9.2984036389     0.0005687467      0.0005687467          0.0414570426
    5          9.2989631360     0.0000017182      0.0000017182          0.0408975455
```

The only abnormal outputs would be for intervals outside of [2, 12], hence I am not going to show them. The algorithm wouldn't start due to invalid interval arguments. Also, for max interval range, the algorithm won't start due to the following:

```
>> [iter_count, x0] = regula_falsi(2, 12)
Error using regula_falsi (line 11)
Function has no different signs at interval boundaries. f(a) * f(b) >= 0
```

As it can be also seen from the plot

Newton's method conclusions

This method might take more iterations to converge, but no need for initial_guess is a positive feature. Convergence is better as it has better interval tolerance, but also it doesn't require an initial guess, which is helpful if we do not know or can not guess how the function looks like.

Task 1 conclusions

From my tests, for the considered function, regula falsi method worked better, as there is no need for initial_guess, and no intervals cause divergence. Newton's method needs proper initial_guess and interval choice, because otherwise the method will diverge or find a zero outside the interval. Finding zeros outside the interval is not a serious abnormality (it nonetheless is), but complete divergence is a big issue.

Task 2

II Find all (real and complex) roots of the polynomial

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

using the Müller's method implementing both the MM1 and MM2 versions. Compare the 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).

Introduction

This task consists of the following parts:

1. Implementation and use of MM1 version of the algorithm
2. Implementation and use of MM2 version
3. Using Newton's method for the polynomial
4. Function evaluation functions (with derivative evaluation)

MM1 method - theoretical background

The general principle of the MM1 method, based on the *Numerical Methods* book, is that we choose three points *x0, x1, x2* with corresponding values of *f(x0), f(x1), f(x2).* Then, a parabola is constructed that passes through these points. Zero points of this parabola are computed and one of them is selected as the solution for our polynomial.

Assuming x2 is an actual approximation of the polynomial root:

$$z = x - x_2$$

and use the differences

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

The interpolating parabola defined in the variable z is considered,

$$y(z) = az^2 + bz + c. \tag{6.32}$$

Then:

Considering the three given points, we have

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

Therefore, the following system of 2 equations must be solved to find a and b:

$$az_0^2 + bz_0 = f(x_0) - f(x_2), \tag{6.33}$$
$$az_1^2 + bz_1 = f(x_1) - f(x_2). \tag{6.34}$$

Calculating c is trivial, but as it is written in the above-mentioned book segment, we will need to calculate a and b to use later on. z0 and z1 are known and can be calculated using formulas given earlier.
Then, the following operations are done:

The roots of (6.32) are given by

$$z_+ = \frac{-2c}{b + \sqrt{b^2 - 4ac}}, \tag{6.35}$$

$$z_- = \frac{-2c}{b - \sqrt{b^2 - 4ac}}. \tag{6.36}$$

The root that has a smaller absolute value (i.e., nearer to the assumed current best root approximation x_2) is chosen for the next iteration,

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

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 x_3 is taken, together with those two from points selected from x_0, x_1, x_2 which are closer to x_3.

When z- and z+ are calculated, we check denominators from the formulas. Appropriate zmin is chosen, which is then added to our x2 (x2 being our initial actual approximation of a 0 point). Obtained product of addition - x3 is our new x2 in the next iteration. We also need to choose from x0, x1 and x2 two values that are the closest to the x3. Then these values are x0 and x1 for the next iteration, x3 is x2.
The process is repeated until desirable accuracy of the zero point is obtained, or we reach the iteration limit due to e.g. non-existence of a zero point.
Note that the system of equations for a and b was solved with the use of *Wolfram Alpha* software.

MM2 method - theoretical background

The MM2 method seems more trivial than the MM1 method in the sense of the complexity of the algorithm. We use derivative characteristics to compute *a,b,c*.
Note that y(z) and z definitions are the same as in MM1.

Using above-mentioned characteristics, we get the following formulas:

$$\begin{aligned} y(0) &= c = f(x_k), \\ y'(0) &= b = f'(x_k), \\ y''(0) &= 2a = f''(x_k), \end{aligned}$$

Which then leads to the same formula for the roots, but with a,b,c replaced with the corresponding function, derivative or 2nd order derivative value:

$$z_{+,-} = \frac{-2f(x_k)}{f'(x_k) \pm \sqrt{(f'(x_k))^2 - 2f(x_k)f''(x_k)}}. \tag{6.37}$$

zmin is chosen in the same way as in the MM1 method.

MATLAB - function value computation

Below I show matlab code for f(x) computation, but also for f'(x) and f''(x) which are needed for the MM2 algorithm:

f(x):

```
function y = eval_f_x(x)
    a0 = 1;
    a = [2, 5, 5, -2]; % Task description a numeration will be the same
    % Hence a is reversed. 1 is a(1) 2 is a(2), 5 is a(3) etc.
    y = a(4) * x^4 + a(3) * x^3 + a(2) * x^2 + a(1) + a0; % As in the task description
    % a0 is not in array, because in matlab arrays start at 1
end
```

f'(x):

```
function y = eval_f_dx(x)
    % a0 = 1;
    a = [2, 5, 5, -2]; % Task description a numeration will be the same
    % Hence a is reversed. 1 is a(1) 2 is a(2), 5 is a(3) etc.
    y = a(4) * 4*x^3 + a(3) * 3*x^2 + a(2) * 2*x + a(1); % Manually computed derivative (1st)
end
```

and finally f''(x):

```
function y = eval_f_d2x(x)
    % a0 = 1;
    a = [2, 5, 5, -2]; % Task description a numeration will be the same
    % Hence a is reversed. 1 is a(1) 2 is a(2), 5 is a(3) etc.
    y = a(4) * 12*x^2 + a(3) * 6*x + a(2) * 2; % Manually computed derivative (2st)
end
```

Computation is done in the same way as for task1, except for the way 'a' coefficients are passed into the function formula.

MATLAB - function zero points computation

Without using any of the task algorithms, this is the way to find the zero points of our function. I used *roots()* function to also find imaginary zero points of our function:

```
a =

    -2     5     5     2     1

>> roots(a)

ans =

   3.348971152643365 + 0.000000000000000i
  -0.669472854180488 + 0.000000000000000i
  -0.089749149231438 + 0.463633208095319i
  -0.089749149231438 - 0.463633208095319i
```

And as we can see, our polynomial has 2 real zeros, and 2 imaginary zeros. The plots are shown later, both for imaginary and real roots.

MATLAB - iteration information printout

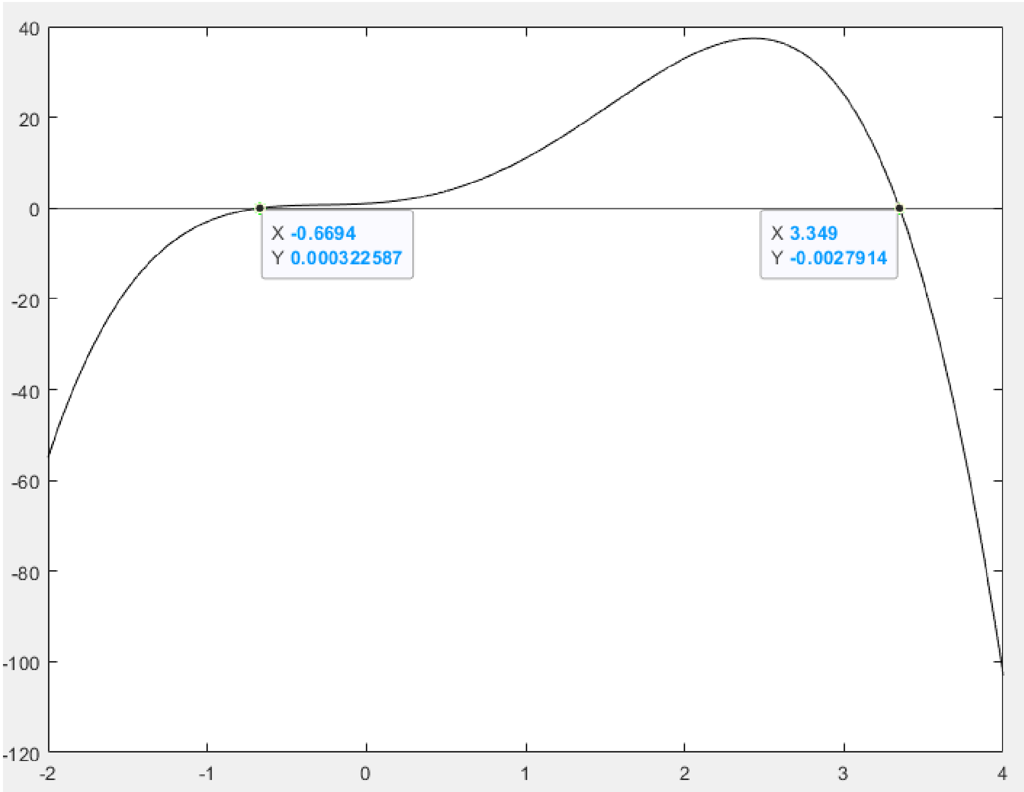
Below I show code that allows me to print information regarding current iteration info. Every column shows data regarding the iteration for MM1, MM2 or Laguerre algorithm. It is represented in a comma-separated fashion:

```
function pretty_print(iter, a_min, x) % Method realises pretty printout of every iteration for MM2, MM1 and laguerre
    if isnan(a_min) % Special call for 0th iteration is indicated by a_min = NaN
        fprintf("0 ;\t\t\t  -\t\t  ; %.5f + %.5fi ; %.5f + %.5fi\n", real(x), imag(x), real(eval_f_x(x)), imag(eval_f_x(x)));
    else
        fprintf("%d ; %.5f + %.5fi ; %.5f + %.5fi ; %.5f + %.5fi\n", ...
            iter, real(a_min), imag(a_min), real(x), imag(x), real(eval_f_x(x)), imag(eval_f_x(x)));
    end
end
% This method was written to reduce number of code lines in the algorithm
% That are not directly related to the algorithm itself (the calculations)
```

It uses fprintf to neatly show data.

MATLAB - Plotting the function

This is how the plot of the function looks like with marked zero points. Chosen interval was [-2,4]:

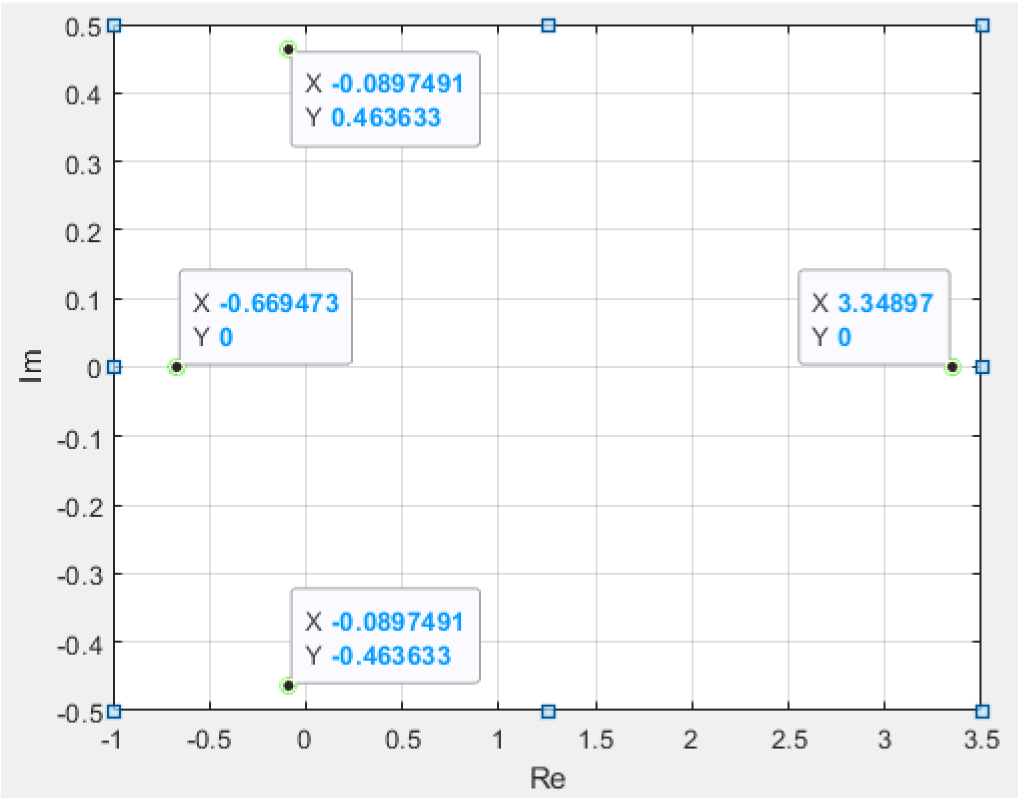


Note that I purposely skip pasting the code in this paragraph, as it is almost the same as the code used in task 1 for function plotting. Only values changed are of course the function used and also the interval related values.

But for this case, we also have complex roots. We can plot a re/im map showing the real roots with the complex roots. The following code will be used:

```
function plot_im_roots()
    a = roots( [-2 5 5 2 1] );
    plot(real(a), imag(a), 'go');
    grid on;
    xlabel('Re');
    ylabel('Im');
end
```

In general, it creates an array containing the roots of our equation.
Then we plot the real part of the results vs imaginary part, and mark the points with a green o (hence the argument ‘go’). Then the grid is turned on to better see where the points lay, and the x/y axis are marked as re/im. The following plot is obtained:



So from that we identify all our computed roots beforehand. if Y = 0, then the root is a real root. Otherwise, it is a complex root of ‘Re’ part equal to position on X axis, and ‘Im’ part equal to the position on Y axis (e.g. -0.0897491 - 0.463633i)

MATLAB - MM2 method implementation

Below I show screenshots of code of the MM2 algorithm implementation in MATLAB. I have chosen to show it before MM1, as the code turned out to be shorter and easier compared to MM1 method code.

```
function [iter_count, zero_point] = muller_2(zero_point)
    % zero_point is denoted as x0 in later iterations
    format long; % Suprisingly important instruction. Otherwise the precision will be bad
    tolerance = 1e-05;
    iter_count = 0;
    fprintf("Iteration ; z_min ; x0 ; f(x0)\n"); % Define contents of rows
    pretty_print(iter_count, NaN, zero_point); % Special NaN call for 0th iteration
    % If our initial_guess zero_point somehow is 100% correct, the
    % 'while' loop does not start. iter_count is returned to be 0
    % We stop iterating when our result is within our tolerance, or
    % divergence occurs (too big iteration count)
    while iter_count < 1000 && abs(eval_f_x(zero_point)) > tolerance
        % As in the formula from the book, define a,b,c:
        c = eval_f_x(zero_point);
        b = eval_f_dx(zero_point);
        a = eval_f_d2x(zero_point);

        % Define z- and x+
        z_neg = -2*c / (b - sqrt(b^2 - 2*a*c)); % z-
        z_pos = -2*c / (b + sqrt(b^2 - 2*a*c)); % z+

        % We want the z value with bigger absolute DENOMINATOR value
        if abs(z_neg) > abs(z_pos) % That means z_pos has smaller denominator
            z_min = z_pos;
        else % Otherwise, z_neg has smaller abs value of the denominator
            z_min = z_neg;
        end

        zero_point = zero_point + z_min; % Otherwise start the iteration with a new zero_point
        iter_count = iter_count + 1; % bump+ the iteration count

        % Iteration information printout
        pretty_print(iter_count, z_min, zero_point); % Prints current iteration info
        %helper = eval_f_x(zero_point);
        %fprintf("%d ; %.10f ; %.10f ; %.10f + %.10fi\n", iter_count, z_min, zero_point, real(helper), imag(helper));
    end
    if (abs(imag(zero_point)) < 1e-05) % If the imaginary part is negligible, we remove it from our zero point
        zero_point = real(zero_point);
    end
    disp('zero_point is '); disp(zero_point); % Display obtained zero_point
end
```

MATLAB - MM2 method output

Here are the obtained outputs from MM2 method:

For initial_guess not being too far away from the root:

```
>> [iter, x0] = muller_2(3)
Iteration ; z_min ; x0 ; f(x0)
0 ;          -        ; 3.00000 ; 25.00000
1 ; 0.35827 + 0.00000i ; 3.35827 + 0.00000i ; -0.90670 + 0.00000i
2 ; -0.00930 + 0.00000i ; 3.34897 + 0.00000i ; 0.00002 + 0.00000i
3 ; 0.00000 + 0.00000i ; 3.34897 + 0.00000i ; 0.00000 + 0.00000i
zero_point is
   3.348971152643363
```

Initial_guess being a bit farther away:

```
>> [iter, x0] = muller_2(10)
Iteration ; z_min ; x0 ; f(x0)
0 ;          -        ; 10.00000 ; -14479.00000
1 ; -3.06124 + 2.11761i ; 6.93876 + 2.11761i ; -648.69725 + -3499.40137i
2 ; -1.45429 + -2.10132i ; 5.48446 + 0.01630i ; -822.24833 + -13.22780i
3 ; -1.47755 + -0.90351i ; 4.00691 + -0.88722i ; -5.42505 + 186.72598i
4 ; -0.45830 + 0.83887i ; 3.54861 + -0.04834i ; -22.43939 + 6.33583i
5 ; -0.20120 + 0.04977i ; 3.34741 + 0.00143i ; 0.15115 + -0.13802i
6 ; 0.00156 + -0.00143i ; 3.34897 + -0.00000i ; 0.00000 + 0.00000i
zero_point is
   3.348971151344797
```

Now other roots. Initial_guesses are not too far, but not too close either:

```
>> [iter, x0] = muller_2(-3)
Iteration ; z_min ; x0 ; f(x0)
0 ;          -        ; -3.00000 ; -257.00000
1 ; 1.09122 + 0.73874i ; -1.90878 + 0.73874i ; -9.76140 + 60.67873i
2 ; 0.49788 + -0.73496i ; -1.41090 + 0.00377i ; -13.83638 + 0.15175i
3 ; 0.50090 + -0.30704i ; -0.91000 + -0.30327i ; -0.12629 + -3.09955i
4 ; 0.16001 + 0.28903i ; -0.74999 + -0.01423i ; -0.42701 + -0.08982i
5 ; 0.08172 + 0.01497i ; -0.66828 + 0.00074i ; 0.00529 + 0.00326i
6 ; -0.00120 + -0.00074i ; -0.66947 + -0.00000i ; 0.00000 + -0.00000i
zero_point is
  -0.669472853606977
```

Now - complex roots. Note the convergence range for these roots seems to be smaller than for non-complex:

```
>> [iter, x0] = muller_2(0 - i)
Iteration ; z_min ; x0 ; f(x0)
0 ;          -        ; 0.00000 + -1.00000i ; -6.00000 + 3.00000i
1 ; -0.28156 + 0.42447i ; -0.28156 + -0.57553i ; 0.54749 + 1.06482i
2 ; 0.18408 + 0.09871i ; -0.09748 + -0.47682i ; -0.03408 + 0.06623i
3 ; 0.00773 + 0.01319i ; -0.08974 + -0.46364i ; -0.00002 + -0.00001i
4 ; -0.00000 + 0.00000i ; -0.08975 + -0.46363i ; -0.00000 + 0.00000i
zero_point is
 -0.089749149231438 - 0.463633208095319i
```

And the 2nd zero point:

```
>> [iter, x0] = muller_2(0 + i)
Iteration ; z_min ; x0 ; f(x0)
0 ;           -        ; 0.00000 + 1.00000i ; -6.00000 + -3.00000i
1 ; -0.28156 + -0.42447i ; -0.28156 + 0.57553i ; 0.54749 + -1.06482i
2 ; 0.18408 + -0.09871i ; -0.09748 + 0.47682i ; -0.03408 + -0.06623i
3 ; 0.00773 + -0.01319i ; -0.08974 + 0.46364i ; -0.00002 + 0.00001i
4 ; -0.00000 + -0.00000i ; -0.08975 + 0.46363i ; -0.00000 + -0.00000i
zero_point is
 -0.089749149231438 + 0.463633208095319i
```

Example extra case. The algorithm still manages to converge, although in 24 iter:

```
>> [iter, x0] = muller_2(5000)
Iteration ; z_min ; x0 ; f(x0)
0 ;           -        ; 5000.00000 + 0.00000i ; -1249374874989999.00000 + 0.00000i
```

```
21 ; -1.56314 + 0.97649i ; 4.15145 + 0.96361i ; -12.99430 + -235.27167i
22 ; -0.53125 + -0.91650i ; 3.62020 + 0.04711i ; -32.30993 + -6.81794i
23 ; -0.27578 + -0.04991i ; 3.34442 + -0.00280i ; 0.43964 + 0.26892i
24 ; 0.00455 + 0.00280i ; 3.34897 + 0.00000i ; 0.00000 + -0.00000i
zero_point is
   3.348971149820774
```

I decided not to show every iteration.

MATLAB - MM1 method implementation

The following code is my implementation of the MM1 algorithm in MATLAB:

```
function [iter_count, x2] = muller_1(x0, x1, x2)
    format long;
    % x2 is our zero point.
    tolerance = 1e-05;
    iter_count = 0;
    fprintf("Iteration ; z_min ; x2 ; f(x2)\n"); % Define contents of rows
    pretty_print(iter_count, NaN, x2); % Special NaN call to get info for
    % 0th iteration before starting the loop

    % If our initial_guess x2 somehow is 100% correct, the
    % 'while' loop does not start. iter_count is returned to be 0
    % We stop iterating when our result is within our tolerance, or
    % divergence occurs (too big iteration count)
while iter_count < 1000 && abs(eval_f_x(x2)) > tolerance
    % x2 is assumed to be the initial guess
    % Define z0 and z1 per definition:
    z0 = x0 - x2;
    z1 = x1 - x2;

    % Computed a and b values below. c is trivial:
    %a = (-z0 * f(x1) + z0 * f(x2) + z1 * f(x0) - z1 * f(x2)) / (z0^2 * z1 - z0 * z1^2)
    a = (-z0 * eval_f_x(x1) + z0 * eval_f_x(x2) + z1 * eval_f_x(x0) - z1 * eval_f_x(x2)) / ...
        (z0 * z1 * (z0 - z1));
    %b = (z0^2 * (f(x1) - f(x2)) + z1^2 * (f(x2) - f(x0))) / z0*z1 * (z0-z1)
    b = (z0^2 * (eval_f_x(x1) - eval_f_x(x2)) + z1^2 * (eval_f_x(x2) - eval_f_x(x0))) / ...
        (z0 * z1 * (z0 - z1));
    c = eval_f_x(x2);

    % Define z- and x+
    z_neg = -2*c / (b - sqrt(b.^2 - 4*a*c)); % z-
    z_pos = -2*c / (b + sqrt(b.^2 - 4*a*c)); % z+

    if abs(z_neg) > abs(z_pos) % That means z_pos has smaller denominator
        z_min = z_pos;
    else % Otherwise, z_neg has smaller abs value of the denominator
        z_min = z_neg;
    end
    % We need to choose new x0 and x1. The ones being closest to x3
    x3 = x2 + z_min; % Start the iteration with a new zero_point
    if (abs(x0 - x3) > abs(x1 - x3) && abs(x0 - x3) > abs(x2 - x3)) % That means x0 is farthest away
        % So we do not assign it
        x0 = x1;
        x1 = x2;
    elseif (abs(x1 - x3) > abs(x0 - x3) && abs(x1 - x3) > abs(x2 - x3)) % x1 is farthest away
        %x0 = x0;
        x1 = x2;
    end
    iter_count = iter_count + 1; % bump+ the iteration count
    x2 = x3;
    pretty_print(iter_count, z_min, x2); % Prints current iteration info
    %fprintf("%d ; %.10f ; %.10f ; %.10f\n", iter_count, z_min, x2, eval_f_x(x2));
end
    if (abs(imag(x2)) < 1e-05) % If the imaginary part is negligible, we remove it from our zero point
        x2 = real(x2);
    end
    disp('zero_point is '); disp(x2); % Display obtained zero_point
end
```

MATLAB - MM1 method output

Here are the obtained outputs from the MM1 method. Please note that before showing some chosen results, I did some trial & error to check how different arguments influence the results (there are 3 input arguments for this method, one of them being the initial guess x2):

Finding root @ 3.3489:

```
>> [iter, x0] = muller_1(1,4,3)
Iteration ; z_min ; x2 ; f(x2)
0 ; N/A ; 3.00000 + 0.00000i ; 25.00000 + 0.00000i
1 ; 0.26355 + 0.00000i ; 3.26355 + 0.00000i ; 7.70022 + 0.00000i
2 ; 0.08120 + 0.00000i ; 3.34475 + 0.00000i ; 0.40750 + 0.00000i
3 ; 0.00425 + 0.00000i ; 3.34900 + 0.00000i ; -0.00265 + 0.00000i
4 ; -0.00003 + 0.00000i ; 3.34897 + 0.00000i ; 0.00000 + 0.00000i
zero_point is
   3.348971150432511
```

Finding other real root:

```
>> [iter, x0] = muller_1(-1,1, -0.2)
Iteration ; z_min ; x2 ; f(x2)
0 ; N/A ; -0.20000 + 0.00000i ; 0.75680 + 0.00000i
1 ; -0.12636 + 0.00000i ; -0.32636 + 0.00000i ; 0.68334 + 0.00000i
2 ; -0.24158 + 0.00000i ; -0.56793 + 0.00000i ; 0.35287 + 0.00000i
3 ; -0.15869 + 0.00000i ; -0.72662 + 0.00000i ; -0.28906 + 0.00000i
4 ; 0.06157 + 0.00000i ; -0.66505 + 0.00000i ; 0.01938 + 0.00000i
5 ; -0.00436 + 0.00000i ; -0.66941 + 0.00000i ; 0.00026 + 0.00000i
6 ; -0.00006 + 0.00000i ; -0.66947 + 0.00000i ; 0.00000 + 0.00000i
zero_point is
  -0.669472819158243
```

Now finding complex roots:

```
>> [iter, x0] = muller_1(-0.5i, 0.5i, 0)
Iteration ; z_min ; x2 ; f(x2)
0 ; N/A ; 0.00000 + 0.00000i ; 1.00000 + 0.00000i
1 ; -0.06818 + -0.42091i ; -0.06818 + -0.42091i ; 0.12771 + -0.17172i
2 ; -0.02137 + -0.04010i ; -0.08955 + -0.46101i ; 0.01022 + -0.00718i
3 ; -0.00020 + -0.00260i ; -0.08975 + -0.46362i ; 0.00008 + -0.00003i
4 ; 0.00000 + -0.00002i ; -0.08975 + -0.46363i ; -0.00000 + 0.00000i
zero_point is
 -0.089749151365638 - 0.463633210160498i
```

Note that for *x0 = -2i, x1 = 2i*:

```
>> [iter, x0] = muller_1(-2i, 2i, 0)
Iteration ; z_min ; x2 ; f(x2)
0 ; N/A ; 0.00000 + 0.00000i ; 1.00000 + 0.00000i
1 ; 0.05798 + 0.00000i ; 0.05798 + 0.00000i ; 1.13373 + 0.00000i
2 ; 0.07320 + 0.00000i ; 0.13119 + 0.00000i ; 1.35912 + 0.00000i
3 ; 0.10453 + -0.00000i ; 0.23572 + -0.00000i ; 1.80857 + -0.00000i
4 ; 0.21778 + -0.00000i ; 0.45350 + -0.00000i ; 3.31708 + -0.00000i
5 ; 0.15231 + -0.44731i ; 0.60581 + -0.44731i ; 2.87144 + -5.25752i
6 ; -0.39058 + -0.15145i ; 0.21522 + -0.59876i ; -1.30015 + -2.15077i
7 ; -0.22438 + 0.13378i ; -0.00916 + -0.46498i ; -0.16251 + -0.37796i
8 ; -0.06674 + -0.00613i ; -0.07589 + -0.47111i ; -0.06547 + -0.04075i
9 ; -0.01368 + 0.00696i ; -0.08957 + -0.46415i ; -0.00255 + 0.00055i
10 ; -0.00018 + 0.00052i ; -0.08975 + -0.46363i ; -0.00000 + 0.00000i
zero_point is
 -0.089749710110379 - 0.463633993079119i
```

so increasing the difference between x0 and x1, we increase iteration count.
e.g *x1 = -10i, x2 = 10i*

```
141 ; -0.07085 + -0.08405i ; -0.08606 + -0.42564i ; 0.14010 + -0.09580i
142 ; -0.00608 + -0.03674i ; -0.09214 + -0.46238i ; 0.01098 + 0.00662i
143 ; 0.00239 + -0.00127i ; -0.08975 + -0.46365i ; -0.00008 + 0.00005i
144 ; 0.00000 + 0.00002i ; -0.08975 + -0.46363i ; 0.00000 + -0.00000i
zero_point is
 -0.089749150583366 - 0.463633205485358i
```

We manage to find the root, but after 144 iterations. This is because x2 is not especially close to the actual root.

Now finding the other complex root, with smaller x0 and x1:

```
>> [iter, x0] = muller_1(-i, i, 0.5i)
Iteration ; z_min ; x2 ; f(x2)
0 ; N/A ; 0.00000 + 0.50000i ; -0.37500 + 0.37500i
1 ; -0.05609 + -0.00598i ; -0.05609 + 0.49402i ; -0.22221 + 0.07800i
2 ; -0.03710 + -0.02871i ; -0.09319 + 0.46531i ; 0.00172 + -0.01822i
3 ; 0.00343 + -0.00170i ; -0.08976 + 0.46361i ; 0.00011 + 0.00001i
4 ; 0.00001 + 0.00002i ; -0.08975 + 0.46363i ; -0.00000 + -0.00000i
zero_point is
 -0.089749150551568 + 0.463633213771112i
```

Note: when x2 is chosen to be -0.5i instead of 0.5i, the results are the same, with only difference that the imaginary parts have inverted signs. e.g.:

```
>> [iter, x0] = muller_1(-i, i, -0.5i)
Iteration ; z_min ; x2 ; f(x2)
0 ; N/A ; -0.00000 + -0.50000i ; -0.37500 + -0.37500i
1 ; -0.05609 + 0.00598i ; -0.05609 + -0.49402i ; -0.22221 + -0.07800i
2 ; -0.03710 + 0.02871i ; -0.09319 + -0.46531i ; 0.00172 + 0.01822i
3 ; 0.00343 + 0.00170i ; -0.08976 + -0.46361i ; 0.00011 + -0.00001i
4 ; 0.00001 + -0.00002i ; -0.08975 + -0.46363i ; -0.00000 + 0.00000i
zero_point is
 -0.089749150551568 - 0.463633213771112i
```

```
>> [iter, x0] = muller_1(-1000i, 1000i, -0.5i)
Iteration ; z_min ; x2 ; f(x2)
0 ; N/A ; -0.00000 + -0.50000i ; -0.37500 + -0.37500i
1 ; -0.00000 + -0.00000i ; -0.00000 + -0.50000i ; -0.37500 + -0.37500i
2 ; -0.00047 + -0.00020i ; -0.00047 + -0.50020i ; -0.37534 + -0.37181i
3 ; -0.08920 + 0.03536i ; -0.08967 + -0.46484i ; -0.00515 + 0.00265i
4 ; -0.00009 + 0.00120i ; -0.08976 + -0.46364i ; 0.00001 + 0.00007i
5 ; 0.00001 + 0.00001i ; -0.08975 + -0.46363i ; 0.00000 + -0.00000i
zero_point is
 -0.089749148514951 - 0.463633205653680i
```

So as it can be seen, increasing x0 and x1 does not have a big influence on the iteration count if x2 is rather close to the result. But if x2 is not close enough to the result, then increase in x0 and x1 values will cause more iterations to occur. Despite more iterations occuring, the method still can converge.

MM2 method vs Newton's method for real roots - comparison

Below I present a comparison between the MM2 method and Newton's method. Since Newton's method is unable to compute complex roots, hence I will show a comparison for real roots only. Note that Newton's method has been modified to better fit the .csv output format. [2,12] interval requirement has been removed from the function as well, since we are operating on a different function. Note that before-mentioned Muller_2 computation screenshots are not introduced again in this paragraph. Refer to earlier provided pictures. Interval for Newton's method is taken as x1 and x2 from the Muller_1 method. The interval in Newton's algorithm is irrelevant in our case, as long as it is correct (no divergence occurs) i.e. for different correct intervals, there is no difference in Newton's algorithm results.

Interval [1,4], initial guess = 3:

Raw outputs:

```
>> [iter, x0] = Newton(3, 1, 4)
Initial guess: 3.00000   ; chosen interval: [1.000, 4.000]

Iteration ; x value ; f(x) ; abs(f(zero_point))
0 ; 3.00000 ; 25.00000 ; 25.00000
1 ; 3.51020 ; -17.75679 ; 17.75679
2 ; 3.36710 ; -1.78047 ; 1.78047
3 ; 3.34924 ; -0.02558 ; 0.02558
4 ; 3.34897 ; -0.00001 ; 0.00001
Obtained zero point is
   3.348971209921166
```

```
>> [iter, x0] = muller_2(3)
Iteration ; z_min ; x0 ; f(x0)
0 ;          -        ; 3.00000 ; 25.00000
1 ; 0.35827 + 0.00000i ; 3.35827 + 0.00000i ; -0.90670 + 0.00000i
2 ; -0.00930 + 0.00000i ; 3.34897 + 0.00000i ; 0.00002 + 0.00000i
3 ; 0.00000 + 0.00000i ; 3.34897 + 0.00000i ; 0.00000 + 0.00000i
zero_point is
   3.348971152643363
```

Comparison tables:

MULLER_2			
Iteration	z_min	x0	f(x0)
0	N/A	3.00000 + 0.00000i	25.00000 + 0.00000i
1	0.35827 + 0.00000i	3.35827 + 0.00000i	-0.90670 + 0.00000i
2	-0.00930 + 0.00000i	3.34897 + 0.00000i	0.00002 + 0.00000i
3	0.00000 + 0.00000i	3.34897 + 0.00000i	0.00000 + 0.00000i
N/A	N/A	3.34897115264336	-3.28626015289046E-14

NEWTON			
Iteration	x value	f(x)	abs(f(zero_point))
0	3	25	25
1	3.5102	-17.75679	17.75679
2	3.3671	-1.78047	1.78047
3	3.34924	-0.02558	0.02558
4	3.34897	-0.00001	0.00001
N/A	N/A	3.34897120992116	-5.54232345262306E-06

Now for interval [-1, 1], initial guess = -0.5 (for 0, MULLER returned complex root):

Raw outputs:

```
>> [iter, x0] = Newton(-0.5, -1, 1)
Initial guess: -0.50000      ; chosen interval: [-1.000, 1.000]

Iteration ; x value ; f(x) ; abs(f(zero_point))
0 ; -0.50000 ; 0.50000 ; 0.50000
1 ; -0.78571 ; -0.67222 ; 0.67222
2 ; -0.69342 ; -0.11218 ; 0.11218
3 ; -0.67074 ; -0.00562 ; 0.00562
4 ; -0.66948 ; -0.00002 ; 0.00002
5 ; -0.66947 ; -0.00000 ; 0.00000
Obtained zero point is
  -0.669472854213756
```

```
>> [iter, x0] = muller_2(-0.5)
Iteration ; z_min ; x0 ; f(x0)
0 ; N/A ; -0.50000 + 0.00000i ; 0.50000 + 0.00000i
1 ; -0.18182 + 0.00000i ; -0.68182 + 0.00000i ; -0.05628 + 0.00000i
2 ; 0.01235 + 0.00000i ; -0.66947 + 0.00000i ; 0.00002 + 0.00000i
3 ; -0.00000 + 0.00000i ; -0.66947 + 0.00000i ; -0.00000 + 0.00000i
zero_point is
  -0.669472854180487
```

Comparison tables:

MULLER			
Iteration	z_min	x0	f(x0)
0	N/A	-0.50000 + 0.00000i	0.50000 + 0.00000i
1	-0.18182 + 0.00000i	-0.68182 + 0.00000i	-0.05628 + 0.00000i
2	0.01235 + 0.00000i	-0.66947 + 0.00000i	0.00002 + 0.00000i
3	-0.00000 + 0.00000i	-0.66947 + 0.00000i	-0.00000 + 0.00000i
N/A	N/A	-0.669472854180487	8.88178419700125E-16

NEWTON			
Iteration	x value	f(x)	abs(f(zero_point))
0	-0.5	0.5	0.5
1	-0.78571	-0.67222	0.67222
2	-0.69342	-0.11218	0.11218
3	-0.67074	-0.00562	0.00562
4	-0.66948	-0.00002	0.00002
5	-0.66947	0	0
N/A	-0.669472854213756	-1.47335477151955E-10	

Now for the last case: interval [-40, 40], initial guess = 30:

Raw outputs:

```
>> [iter, x0] = Newton(30,-40,40)
Initial guess: 30.00000      ; chosen interval: [-40.000, 40.000]
```

```
Iteration ; x value ; f(x) ; abs(f(zero_point))
0 ; 30.00000 ; -1480439.00000 ; 1480439.00000
1 ; 22.67827 ; -468081.64070 ; 468081.64070
2 ; 17.19496 ; -147904.09573 ; 147904.09573
3 ; 13.09368 ; -46677.83296 ; 46677.83296
4 ; 10.03367 ; -14695.58549 ; 14695.58549
5 ; 7.76184 ; -4603.33684 ; 4603.33684
6 ; 6.09255 ; -1426.12704 ; 1426.12704
7 ; 4.89360 ; -430.48299 ; 430.48299
8 ; 4.07731 ; -121.55321 ; 121.55321
9 ; 3.59134 ; -28.43084 ; 28.43084
10 ; 3.38707 ; -3.80291 ; 3.80291
11 ; 3.35011 ; -0.11069 ; 0.11069
12 ; 3.34897 ; -0.00010 ; 0.00010
13 ; 3.34897 ; -0.00000 ; 0.00000
```

```
Obtained zero point is
   3.348971152644301
```

```
>> [iter, x0] = muller_2(30)
Iteration ; z_min ; x0 ; f(x0)
0 ; N/A ; 30.00000 + 0.00000i ; -1480439.00000 + 0.00000i
1 ; -9.77274 + 6.89929i ; 20.22726 + 6.89929i ; -76836.82447 + -361527.10351i
2 ; -4.86950 + -6.89644i ; 15.35776 + 0.00286i ; -91938.15207 + -72.19213i
3 ; -4.87123 + -3.42060i ; 10.48653 + -3.41775i ; -4602.32618 + 22377.69383i
4 ; -2.39708 + 3.41162i ; 8.08945 + -0.00613i ; -5573.34353 + 19.42698i
5 ; -2.40418 + 1.63629i ; 5.68527 + 1.63017i ; -219.97762 + -1334.81905i
6 ; -1.09186 + -1.60754i ; 4.59342 + 0.02262i ; -290.00723 + -9.29618i
7 ; -1.13499 + -0.56104i ; 3.45842 + -0.53841i ; 13.38093 + 58.34003i
8 ; -0.09580 + 0.50916i ; 3.36262 + -0.02925i ; -1.26688 + 2.89378i
9 ; -0.01364 + 0.02925i ; 3.34898 + -0.00000i ; -0.00071 + 0.00020i
10 ; -0.00001 + 0.00000i ; 3.34897 + 0.00000i ; 0.00000 + -0.00000i
zero_point is
  3.348971152643362
```

Comparison tables:

MULLER_2			
Iteration	z_min	x0	f(x0)
0	N/A	30.00000 + 0.00000i	-1480439.00000 + 0.00000i
1	-9.77274 + 6.89929i	20.22726 + 6.89929i	-76836.82447 + -361527.10351i
2	-4.86950 + -6.89644i	15.35776 + 0.00286i	-91938.15207 + -72.19213i
3	-4.87123 + -3.42060i	10.48653 + -3.41775i	-4602.32618 + 22377.69383i
4	-2.39708 + 3.41162i	8.08945 + -0.00613i	-5573.34353 + 19.42698i
5	-2.40418 + 1.63629i	5.68527 + 1.63017i	-219.97762 + -1334.81905i
6	-1.09186 + -1.60754i	4.59342 + 0.02262i	-290.00723 + -9.29618i
7	-1.13499 + -0.56104i	3.45842 + -0.53841i	13.38093 + 58.34003i
8	-0.09580 + 0.50916i	3.36262 + -0.02925i	-1.26688 + 2.89378i
9	-0.01364 + 0.02925i	3.34898 + -0.00000i	-0.00071 + 0.00020i
10	-0.00001 + 0.00000i	3.34897 + 0.00000i	0.00000 + -0.00000i
N/A	N/A	3.34897115264336	7.90478793533111E-14

NEWTON			
Iteration	x value	f(x)	abs(f(zero_point))
0	30	-1480439	1480439
1	22.67827	-468081.6407	468081.6407
2	17.19496	-147904.0957	147904.0957
3	13.09368	-46677.83296	46677.83296
4	10.03367	-14695.58549	14695.58549
5	7.76184	-4603.33684	4603.33684
6	6.09255	-1426.12704	1426.12704
7	4.8936	-430.48299	430.48299
8	4.07731	-121.55321	121.55321
9	3.59134	-28.43084	28.43084
10	3.38707	-3.80291	3.80291
11	3.35011	-0.11069	0.11069
12	3.34897	-0.0001	0.0001
13	3.34897	0	0
N/A	3.3489711526443	-9.07478536760209E-11	

Task 2 - conclusions

In general, the MM2 method shows better traits than MM1. It only requires an initial_guess, and has better convergence time than the MM1 method. Of course MM1 can have good convergence time if the provided x0, x1 and x2 values are good, but overall, MM2 has better performance, even for an initial_guess being really far away from the actual 0 point.

Regarding the Newton vs MM2 comparison, it looks like the convergence time is quite similar, with Newton taking a few iterations longer to get the result.
The bigger the range of initial guess from the actual result, the bigger the difference. Nonetheless, the iteration count difference is not significant.

Task 3

III 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).

Introduction

This task will consist of the following

1. Design of the Laguerre's method
2. Comparison of the Laguerre's algorithm with MM2 for the same initial guesses for the polynomial given in task 2

All zero points will be considered, per task description, so we will have 4 cases.

Laguerre's method - theoretical background

Per *Numerical Methods* book, the following iterative formula is given for the k+1 th iteration of the Laguerre's method:

6.3.2. Laguerre's method

The method is defined by the following formula:

$$x_{k+1} = x_k - \frac{nf(x_k)}{f'(x_k) \pm \sqrt{(n-1)[(n-1)(f'(x_k))^2 - nf(x_k)f''(x_k)]}}, \quad (6.39)$$

where n denotes the order of the polynomial, and the sign in the denominator is chosen in a way assuring a larger absolute value of the denominator, as in the Müller's

Note that for k = 1 - x_1 is our initial guess.
Furthermore, the denominator has +-, because as it is mentioned in the text, we need to choose a bigger denominator value for our x_k+1 formula. This is a similarity to the MM1 and MM2 method from the previous task.
In general, the method is executed similarly to the MM1 and MM2 method, with the difference being only in the iterative x_k+1 formula.
Also, per *Numerical Methods* book:

"The Laguerre's formula (6.39) is slightly more complex, it takes also into account the order of the polynomial [...], therefore the Laguerre's method is better, in general"

We will test the above-mentioned in this task by comparing this method to MM2.
Also, per information from the *Numerical Methods* books, this algorithm is said to be the best method for polynomial root finding.

MATLAB - Laguerre's method

This is the code in MATLAB used to implement Laguerre's method. A lot of parts are reused and are the same as in the MM1 and MM2 algorithm's code. The difference still is in the formula used :

```
function [iter_count, xk] = laguerre(xk) % xk being initial guess, zero point
    format long;
    n = 4; % Order of our polynomial
    iter_count = 0;
    tolerance = 1e-05;
    fprintf("Iteration ; z_min ; xk ; f(xk)\n"); % Define contents of rows
    pretty_print(iter_count, NaN, xk); % Special NaN call to get info for
    % If our initial_guess xk somehow is 100% correct, the
    % 'while' loop does not start. iter_count is returned to be 0
    % We stop iterating when our result is within our tolerance, or
    % divergence occurs (too big iteration count)

while iter_count < 1000 && abs(eval_f_x(xk)) > tolerance
    % For this algorithm I decided to include denominators separatly
    % For better readability (these are way longer than for MM2)
    denominator_neg = eval_f_dx(xk) - ...
        sqrt( (n-1)*((n-1)*(eval_f_dx(xk)^2) - n*eval_f_x(xk)*eval_f_d2x(xk)) );
    denominator_pos = eval_f_dx(xk) + ...
        sqrt( (n-1)*((n-1)*(eval_f_dx(xk)^2) - n*eval_f_x(xk)*eval_f_d2x(xk)) );

    % Define two different x_k+1 values, as in the formula
    x_neg = (n * eval_f_x(xk)) / denominator_neg;
    x_pos = (n * eval_f_x(xk)) / denominator_pos;

    if abs(denominator_pos) > abs(denominator_neg) % We choose x_k+1 with bigger denominator
        x_min = x_pos;
    else % Otherwise, z_neg has smaller abs value of the denominator
        x_min = x_neg;
    end

    xk = xk - x_min; % Define new zero point for next iteration
    iter_count = iter_count + 1; % bump+ the iteration count

    pretty_print(iter_count, x_min, xk); % Prints current iteration info
    % fprintf("%d ; %.10f ; %.10f ; %.10f\n", iter_count, x_min, xk, eval_f_x(xk));
end
    end

    if (abs(imag(xk)) < 1e-05) % If the imaginary part is negligible, we remove it from our zero point
        xk = real(xk);
    end
    disp('zero_point is '); disp(xk); % Display obtained zero_point
end
```

MATLAB - raw Laguerre's method outputs

Here I present Laguerre's method outputs. I used the same initial points as I used in the previous task for the MM2 method testing: - 3; 10; -3; -i; i; 5000. Furthermore, in the method comparison paragraph, some extra cases might be considered. xk = 3:

```
>> [iter_count, zero_point] = laguerre(3)
Iteration ; z_min ; xk ; f(xk)
0 ;           -        ; 3.00000 + 0.00000i ; 25.00000 + 0.00000i
1 ; -0.34903 + 0.00000i ; 3.34903 + 0.00000i ; -0.00599 + 0.00000i
2 ; 0.00006 + 0.00000i ; 3.34897 + 0.00000i ; 0.00000 + 0.00000i
zero_point is
   3.348971152643363
```

For xk = 10:

```
>> [iter_count, zero_point] = laguerre(10)
Iteration ; z_min ; xk ; f(xk)
0 ;           -        ; 10.00000 + 0.00000i ; -14479.00000 + 0.00000i
1 ; 6.66082 + 0.00000i ; 3.33918 + 0.00000i ; 0.93999 + 0.00000i
2 ; -0.00979 + 0.00000i ; 3.34897 + 0.00000i ; -0.00000 + 0.00000i
zero_point is
   3.348971153621916
```

Xk = -3:

```
>> [iter_count, zero_point] = laguerre(-3)
Iteration ; z_min ; xk ; f(xk)
0 ; N/A ; -3.00000 + 0.00000i ; -257.00000 + 0.00000i
1 ; -2.12158 + 0.00000i ; -0.87842 + 0.00000i ; -1.47862 + 0.00000i
2 ; -0.21048 + 0.00000i ; -0.66795 + 0.00000i ; 0.00674 + 0.00000i
3 ; 0.00153 + 0.00000i ; -0.66947 + 0.00000i ; -0.00000 + 0.00000i
zero_point is
  -0.669472855949188
```

For complex, xk = -i:

```
>> [iter_count, zero_point] = laguerre(-i)
Iteration ; z_min ; xk ; f(xk)
0 ;           -        ; -0.00000 + -1.00000i ; -6.00000 + 3.00000i
1 ; 0.12691 + -0.51212i ; -0.12691 + -0.48788i ; 0.01166 + 0.21612i
2 ; -0.03720 + -0.02423i ; -0.08971 + -0.46364i ; -0.00014 + -0.00013i
3 ; 0.00004 + -0.00001i ; -0.08975 + -0.46363i ; -0.00000 + -0.00000i
zero_point is
 -0.089749149231412 - 0.463633208095312i
```

xk = i:

```
>> [iter_count, zero_point] = laguerre(i)
Iteration ; z_min ; xk ; f(xk)
0 ;           -        ; 0.00000 + 1.00000i ; -6.00000 + -3.00000i
1 ; 0.12691 + 0.51212i ; -0.12691 + 0.48788i ; 0.01166 + -0.21612i
2 ; -0.03720 + 0.02423i ; -0.08971 + 0.46364i ; -0.00014 + 0.00013i
3 ; 0.00004 + 0.00001i ; -0.08975 + 0.46363i ; -0.00000 + 0.00000i
zero_point is
 -0.089749149231412 + 0.463633208095312i
```

And special test case xk = 5000:

```
>> [iter_count, zero_point] = laguerre(5000)
Iteration ; z_min ; xk ; f(xk)
0 ;           -        ; 5000.00000 + 0.00000i ; -1249374874989999.00000 + 0.00000i
1 ; 4996.67947 + 0.00000i ; 3.32053 + 0.00000i ; 2.68850 + 0.00000i
2 ; -0.02844 + 0.00000i ; 3.34897 + 0.00000i ; -0.00000 + 0.00000i
zero_point is
   3.348971177043980
```

Laguerre vs MM2 method - result comparison

Note: pretty print function has been slightly modified to better comply with .csv style output. This simplified result exporting into Excel. It consists of removal of tabulators in fprintf, and changing ‘-’ into ‘N/A’ for the 0th iteration output.

Below are excel tables showing iteration statistics for Laguerre’s and MM2 method. The initial guess can be read from the 0th iteration x value. Note that there are some extra cases included that were not considered beforehand:

initial = 3:

LAGUERRE			
Iteration	z_min	xk	f(xk)
0	N/A	3.00000 + 0.00000i	25.00000 + 0.00000i
1	-0.34903 + 0.00000i	3.34903 + 0.00000i	-0.00599 + 0.00000i
2	0.00006 + 0.00000i	3.34897 + 0.00000i	0.00000 + 0.00000i
N/A	N/A	3.34897115264336	-3.28626015289046E-14

MULLER			
Iteration	z_min	x0	f(x0)
0	N/A	3.00000 + 0.00000i	25.00000 + 0.00000i
1	0.35827 + 0.00000i	3.35827 + 0.00000i	-0.90670 + 0.00000i
2	-0.00930 + 0.00000i	3.34897 + 0.00000i	0.00002 + 0.00000i
3	0.00000 + 0.00000i	3.34897 + 0.00000i	0.00000 + 0.00000i
N/A	N/A	3.34897115264336	-3.28626015289046E-14

initial = 10:

LAGUERRE			
Iteration	z_min	xk	f(xk)
0	N/A	10.00000 + 0.00000i	-14479.00000 + 0.00000i
1	6.66082 + 0.00000i	3.33918 + 0.00000i	0.93999 + 0.00000i
2	-0.00979 + 0.00000i	3.34897 + 0.00000i	-0.00000 + 0.00000i
N/A	N/A	3.34897115362191	-9.46869560536356E-08

MULLER_2			
Iteration	z_min	x0	f(x0)
0	N/A	10.00000 + 0.00000i	-14479.00000 + 0.00000i
1	-3.06124 + 2.11761i	6.93876 + 2.11761i	-648.69725 + -3499.40137i
2	-1.45429 + -2.10132i	5.48446 + 0.01630i	-822.24833 + -13.22780i
3	-1.47755 + -0.90351i	4.00691 + -0.88722i	-5.42505 + 186.72598i
4	-0.45830 + 0.83887i	3.54861 + -0.04834i	-22.43939 + 6.33583i
5	-0.20120 + 0.04977i	3.34741 + 0.00143i	0.15115 + -0.13802i
6	0.00156 + -0.00143i	3.34897 + -0.00000i	0.00000 + 0.00000i
N/A	N/A	3.34897115134479	1.25652046989444E-07

initial = -3:

LAGUERRE			
Iteration	z_min	xk	f(xk)
0	N/A	-3.00000 + 0.00000i	-257.00000 + 0.00000i
1	-2.12158 + 0.00000i	-0.87842 + 0.00000i	-1.47862 + 0.00000i
2	-0.21048 + 0.00000i	-0.66795 + 0.00000i	0.00674 + 0.00000i
3	0.00153 + 0.00000i	-0.66947 + 0.00000i	-0.00000 + 0.00000i
N/A	N/A	-0.669472855949188	-7.8328858954535E-09

MULLER_2			
Iteration	z_min	x0	f(x0)
0	N/A	-3.00000 + 0.00000i	-257.00000 + 0.00000i
1	1.09122 + 0.73874i	-1.90878 + 0.73874i	-9.76140 + 60.67873i
2	0.49788 + -0.73496i	-1.41090 + 0.00377i	-13.83638 + 0.15175i
3	0.50090 + -0.30704i	-0.91000 + -0.30327i	-0.12629 + -3.09955i
4	0.16001 + 0.28903i	-0.74999 + -0.01423i	-0.42701 + -0.08982i
5	0.08172 + 0.01497i	-0.66828 + 0.00074i	0.00529 + 0.00326i
6	-0.00120 + -0.00074i	-0.66947 + -0.00000i	0.00000 + -0.00000i
N/A	N/A	-0.669472853606977	2.53985210640905E-09

Complex roots, initial = -i:

LAGUERRE			
Iteratio	z_min	xk	f(xk)
0	N/A	-0.00000 + -1.00000i	-6.00000 + 3.00000i
1	0.12691 + -0.51212i	-0.12691 + -0.48788i	0.01166 + 0.21612i
2	-0.03720 + -0.02423i	-0.08971 + -0.46364i	-0.00014 + -0.00013i
3	0.00004 + -0.00001i	-0.08975 + -0.46363i	-0.00000 + -0.00000i
N/A	N/A	-0.089749149231412 - 0.463633208095312i	-3.419486915845482e-14 - 1.234568003383174e-13i

MULLER_2			
Iteratio	z_min	x0	f(x0)
0	N/A	-0.00000 + -1.00000i	-6.00000 + 3.00000i
1	-0.28156 + 0.42447i	-0.28156 + -0.57553i	0.54749 + 1.06482i
2	0.18408 + 0.09871i	-0.09748 + -0.47682i	-0.03408 + 0.06623i
3	0.00773 + 0.01319i	-0.08974 + -0.46364i	-0.00002 + -0.00001i
4	-0.00000 + 0.00000i	-0.08975 + -0.46363i	-0.00000 + 0.00000i
N/A	N/A	-0.089749149231438 - 0.463633208095319i	1.110223024625157e-15 + 1.110223024625157e-16i

Complex roots, initial = i:

LAGUERRE			
Iteration	z_min	xk	f(xk)
0	N/A	0.00000 + 1.00000i	-6.00000 + -3.00000i
1	0.12691 + 0.51212i	-0.12691 + 0.48788i	0.01166 + -0.21612i
2	-0.03720 + 0.02423i	-0.08971 + 0.46364i	-0.00014 + 0.00013i
3	0.00004 + 0.00001i	-0.08975 + 0.46363i	-0.00000 + 0.00000i
N/A	N/A	-0.089749149231412 + 0.463633208095312i	-3.419486915845482e-14 + 1.234568003383174e-13i

MULLER_2			
Iteration	z_min	x0	f(x0)
0	N/A	0.00000 + 1.00000i	-6.00000 + -3.00000i
1	-0.28156 + -0.42447i	-0.28156 + 0.57553i	0.54749 + -1.06482i
2	0.18408 + -0.09871i	-0.09748 + 0.47682i	-0.03408 + -0.06623i
3	0.00773 + -0.01319i	-0.08974 + 0.46364i	-0.00002 + 0.00001i
4	-0.00000 + -0.00000i	-0.08975 + 0.46363i	-0.00000 + -0.00000i
N/A	N/A	-0.089749149231438 + 0.463633208095319i	1.110223024625157e-15 - 1.110223024625157e-16i

Slightly abnormal case, initial = 100:

LAGUERRE			
Iteration	z_min	xk	f(xk)
0	N/A	100.00000 + 0.00000i	-194949799.00000 + 0.00000i
1	96.67725 + 0.00000i	3.32275 + 0.00000i	2.48316 + 0.00000i
2	-0.02622 + 0.00000i	3.34897 + 0.00000i	-0.00000 + 0.00000i
N/A	N/A	3.348971171723800	-1.84626413357591E-06

MULLER_2			
Iteration	z_min	x0	f(x0)
0	N/A	100.00000 + 0.00000i	-194949799.00000 + 0.00000i
1	-33.11953 + 23.41606i	66.88047 + 23.41606i	-10220053.65402 + -47648266.14008i
2	-16.55529 + -23.41537i	50.32518 + 0.00069i	-12178327.47695 + -679.89949i
3	-16.55553 + -11.70067i	33.76965 + -11.69998i	-636868.03892 + 2975792.32152i
4	-8.26865 + 11.69882i	25.50101 + -0.00116i	-759563.67011 + 142.11259i
5	-8.26925 + 5.83442i	17.23176 + 5.83326i	-39271.71481 + -185407.74356i
6	-4.11453 + -5.83029i	13.11723 + 0.00297i	-47038.02125 + -45.57726i
7	-4.11679 + -2.88109i	9.00044 + -2.87812i	-2299.54580 + 11428.16879i
8	-2.00985 + 2.86975i	6.99059 + -0.00838i	-2808.78880 + 16.14866i
9	-2.02055 + 1.34534i	4.97004 + 1.33697i	-90.77774 + -665.51236i
10	-0.86017 + -1.30541i	4.10987 + 0.03156i	-129.70400 + -8.16979i
11	-0.91454 + -0.23758i	3.19533 + -0.20602i	16.02122 + 15.04356i
12	0.15025 + 0.20734i	3.34558 + 0.00132i	0.32717 + -0.12671i
13	0.00339 + -0.00132i	3.34897 + -0.00000i	-0.00000 + 0.00000i
N/A	N/A	3.34897115743872	-4.64009052514313OE-07

Abnormal case, initial = 5000:

LAGUERRE

Iteration	z_min	xk	f(xk)
0	N/A	5000.00000 + 0.00000i	-1249374874989999.00000 + 0.00000i
1	4996.67947 + 0.00000i	3.32053 + 0.00000i	2.68850 + 0.00000i
2	-0.02844 + 0.00000i	3.34897 + 0.00000i	-0.00000 + 0.00000i
N/A	N/A	3.348971177043980	-2.36105615059045E-06

MULLER_2

Iteration	z_min	x0	f(x0)
0	N/A	5000.00000 + 0.00000i	-1249374874989999.00000 + 0.00000i
1	-1666.45823 + 1178.36385i	3333.54177 + 1178.36385i	-65553598544963.78906 + -305387157476000.37500i
2	-833.22903 + -1178.36384i	2500.31275 + 0.00001i	-78085915100767.34375 + -1587862.64611i
3	-833.22903 + -589.18179i	1667.08372 + -589.18178i	-4097095507952.27148 + 19086691974503.64062i
4	-416.61435 + 589.18176i	1250.46937 + -0.00002i	-4880366043854.45020 + 298372.50582i
5	-416.61435 + 294.59062i	833.85501 + 294.59060i	-256067318644.27655 + -1192916915579.41113i
6	-208.30684 + -294.59055i	625.54817 + 0.00004i	-305021963544.14764 + -81080.14726i
7	-208.30685 + -147.29475i	417.24132 + -147.29471i	-16003921869.69720 + 74556972717.12410i
8	-104.15275 + 147.29463i	313.08857 + -0.00008i	-19063643314.30795 + 20027.90818i
9	-104.15278 + 73.64625i	208.93579 + 73.64617i	-1000173076.41066 + -4659726883.28119i
10	-52.07503 + -73.64600i	156.86076 + 0.00017i	-1191419923.75676 + -5104.60364i
11	-52.07508 + -36.82085i	104.78568 + -36.82069i	-62492584.42735 + 291211786.16627i
12	-26.03480 + 36.82034i	78.75088 + -0.00034i	-74449080.99861 + 1312.14832i
13	-26.03491 + 18.40580i	52.71597 + 18.40545i	-3901108.03313 + -18195368.12798i
14	-13.01181 + -18.40472i	39.70416 + 0.00073i	-4649289.68411 + -347.71545i
15	-13.01211 + -9.19326i	26.69206 + -9.19253i	-242587.60745 + 1135826.17630i
16	-6.49411 + 9.19089i	20.19795 + -0.00164i	-289577.54829 + 97.59792i
17	-6.49505 + 4.57572i	13.70290 + 4.57409i	-14820.40263 + -70621.04545i
18	-3.22068 + -4.56996i	10.48223 + 0.00412i	-17815.78056 + -30.74190i
19	-3.22441 + -2.23844i	7.25781 + -2.23431i	-821.73075 + 4309.72463i
20	-1.54323 + 2.22143i	5.71458 + -0.01288i	-1024.03707 + 12.15857i
21	-1.56314 + 0.97649i	4.15145 + 0.96361i	-12.99430 + -235.27167i
22	-0.53125 + -0.91650i	3.62020 + 0.04711i	-32.30993 + -6.81794i
23	-0.27578 + -0.04991i	3.34442 + -0.00280i	0.43964 + 0.26892i
24	0.00455 + 0.00280i	3.34897 + 0.00000i	0.00000 + -0.00000i
N/A	N/A	3.34897114982077	2.73119711735603E-07

Abnormal case, initial = -5000:

LAGUERRE			
Iteration	z_min	xk	f(xk)
0	N/A	-5000.00000 + 0.00000i	-1250624875009999.00000 + 0.00000i
1	-4997.93150 + 0.00000i	-2.06850 + 0.00000i	-62.61043 + 0.00000i
2	-1.31647 + 0.00000i	-0.75204 + 0.00000i	-0.44259 + 0.00000i
3	-0.08275 + 0.00000i	-0.66929 + 0.00000i	0.00082 + 0.00000i
4	0.00018 + 0.00000i	-0.66947 + 0.00000i	-0.00000 + 0.00000i
N/A	N/A	-0.669472854183575	-1.36755051727277E-11

MULLER_2			
Iteration	z_min	x0	f(x0)
0	N/A	-5000.00000 + 0.00000i	-1250624875009999.00000 + 0.00000i
1	1666.87489 + 1178.65848i	-3333.12511 + 1178.65848i	-65619184987694.08594 + 305692697451446.68750i
2	833.43736 + -1178.65847i	-2499.68775 + 0.00001i	-78164040122024.60938 + 1583482.07053i
3	833.43736 + -589.32910i	-1666.25038 + -589.32909i	-4101194672873.83887 + -19105788230712.47656i
4	416.71852 + 589.32907i	-1249.53187 + -0.00002i	-4885248869508.66504 + -296257.80499i
5	416.71852 + 294.66427i	-832.81335 + 294.66425i	-256323523260.55286 + 1194110436105.93311i
6	208.35893 + -294.66421i	-624.45442 + 0.00004i	-305327146515.86542 + 79968.51411i
7	208.35894 + -147.33158i	-416.09548 + -147.33154i	-16019938197.60014 + -74631570174.88719i
8	104.17881 + 147.33146i	-311.91668 + -0.00008i	-19082720548.09814 + -19463.08184i
9	104.17882 + 73.66469i	-207.73785 + 73.66461i	-1001175902.61589 + 4664390478.37592i
10	52.08809 + -73.66445i	-155.64976 + 0.00016i	-1192613928.40148 + 4819.62778i
11	52.08813 + -36.83016i	-103.56163 + -36.83000i	-62556172.82833 + -291503898.28440i
12	26.04147 + 36.82970i	-77.52017 + -0.00031i	-74524552.24243 + -1168.98869i
13	26.04151 + 18.41080i	-51.47866 + 18.41049i	-3905540.51057 + 18213946.48054i
14	13.01570 + -18.40992i	-38.46296 + 0.00058i	-4654431.53227 + 275.86876i
15	13.01574 + -9.19716i	-25.44722 + -9.19659i	-243094.48813 + -1137148.70831i
16	6.49828 + 9.19556i	-18.94894 + -0.00103i	-290112.08204 + -61.41172i
17	6.49817 + 4.58332i	-12.45076 + 4.58229i	-14967.53236 + 70784.68975i
18	3.23180 + -4.58065i	-9.21896 + 0.00164i	-17956.41061 + 12.18655i
19	3.23107 + -2.26533i	-5.98790 + -2.26369i	-889.43063 + -4360.81348i
20	1.58709 + 2.26160i	-4.40081 + -0.00209i	-1087.28697 + -1.94354i
21	1.58484 + 1.09262i	-2.81596 + 1.09053i	-47.76627 + 260.29311i
22	0.75175 + -1.08842i	-2.06421 + 0.00211i	-62.11260 + 0.24444i
23	0.74939 + -0.49396i	-1.31482 + -0.49185i	-1.86495 + -14.36831i
24	0.31732 + 0.48566i	-0.99750 + -0.00619i	-2.96175 + -0.09211i
25	0.33617 + 0.14292i	-0.66133 + 0.13673i	0.22479 + 0.55631i
26	-0.00694 + -0.13152i	-0.66827 + 0.00521i	0.00558 + 0.02295i
27	-0.00120 + -0.00521i	-0.66947 + 0.00000i	0.00000 + 0.00000i
N/A	N/A	-0.669472628848291	9.97907708244483E-07

Extremely abnormal case, initial = -1e08:

LAGUERRE			
Iteration	z_min	xk	f(xk)
0	N/A	-100000000.00000 + 0.00000i	-200000004999999971888072627322880.00000 + 0.00000i
1	-100000000.62500 + -4.41261i	0.62500 + 4.41261i	-941.75622 + 53.64061i
2	1.56668 + 3.31518i	-0.94168 + 1.09743i	18.70716 + -2.77638i
3	-0.52519 + 0.94665i	-0.41648 + 0.15078i	0.68759 + 0.12448i
4	0.22166 + 0.17549i	-0.63814 + -0.02470i	0.13462 + -0.09387i
5	0.03131 + -0.02474i	-0.66945 + 0.00003i	0.00008 + 0.00014i
6	0.00002 + 0.00003i	-0.66947 + -0.00000i	0.00000 + -0.00000i
N/A	N/A	-0.669472854180463	1.063593657590900E-13

MULLER_2			
Iteration	z_min	x0	f(x0)
0	N/A	-100000000.00000 + 0.00000i	-200000004999999971888072627322880.00000 + 0.00000i
1	33333333.54167 + 23570226.18687i	-66666666.45833 + 23570226.18687i	-10493827422839478649233126457344.00000 + 48886395970859463837851133673472.00000i
2	16666666.77083 + -23570226.18687	-49999999.68750 + 0.00000i	-12500000312499991487605098151936.00000 + 7450580736622214.00000i
3	16666666.77083 + 11785113.09343i	-33333332.91667 + 11785113.09343i	-655864213927465867464698494976.00000 + 3055399748178714801015835590656.00000i
4	8333333.38542 + -11785113.09343i	-24999999.53125 + 0.00000i	-781250019531248905025365213184.00000 + 698491944058333.25000i
5	8333333.38542 + -5892556.54672i	-16666666.14583 + -5892556.54672i	-40991513370466344037659967488.00000 + -190962484261169217666652569600.00000i
6	4166666.69271 + 5892556.54672i	-12499999.45313 + 0.00000i	-48828126220702669535992348672.00000 + 0.00000i
7	4166666.69271 + 2946278.27336i	-8333332.76042 + 2946278.27336i	-2561969585654015660470042624.00000 + 11935155266322933167654174720.00000i
8	2083333.34635 + -2946278.27336i	-6249999.41406 + 0.00000i	-3051757888793824487022788608.00000 + 10913936625911.45898i
9	2083333.34635 + 1473139.13668i	-4166666.06771 + 1473139.13668i	-160123099103343199588974592.00000 + 745947204145150749946413056.00000i
10	1041666.67318 + -1473139.13668i	-3124999.39453 + 0.00000i	-190734868049591284292124672.00000 + 3183231515890.85400i
11	1041666.67318 + -736569.56834i	-2083332.72135 + -736569.56834i	-10007693693952447393824768.00000 + -46621700259063503735750656.00000i
12	520833.33659 + 736569.56834i	-1562499.38477 + -0.00000i	-11920929253093760141623296.00000 + -412114794468.01794i
13	520833.33659 + 368284.78417i	-1041666.04818 + 368284.78417i	-625480855870211996254208.00000 + 2913856266189393440538624.00000i
14	260416.66829 + -368284.78417i	-781249.37988 + 0.00000i	-745058078316935824539648.00000 + 128341784053.08585i

[...]

MULLER_2			
Iteration	z_min	x0	f(x0)
48	1.95157 + -2.77486i	-5.45544 + 0.00201i	-2444.45403 + 3.39534i
49	1.94984 + -1.35307i	-3.50560 + -1.35107i	-112.63690 + -588.54214i
50	0.93766 + 1.34897i	-2.56795 + -0.00209i	-142.80401 + -0.44134i
51	0.93467 + 0.62769i	-1.63327 + 0.62559i	-5.09213 + 33.44782i
52	0.41713 + -0.62189i	-1.21615 + 0.00371i	-7.40527 + 0.09791i
53	0.42478 + -0.24096i	-0.79136 + -0.23725i	0.08957 + -1.61502i
54	0.09262 + 0.22203i	-0.69875 + -0.01522i	-0.13620 + -0.07707i
55	0.02928 + 0.01531i	-0.66946 + 0.00009i	0.00004 + 0.00038i
56	-0.00001 + -0.00009i	-0.66947 + 0.00000i	0.00000 + 0.00000i
N/A	N/A	-0.669472854179979	2.249533892495490E-12

Note: For x = 3 case, MULLER is equivalent to MULLER_2.
At the bottom of every table shown, after the last iteration, I included more precise x0 and f(x0) values to show the obtained accuracy of the zero point.

Task 3 - conclusions and result analysis

As it can be seen from the Laguerre and MM2 method comparison, the MM2 method performs worse. Laguerre has better handling of abnormal input arguments, i.e. for initial guesses that are far from the actual zero point, Laguerre's method converges way faster (in a considerably smaller iteration count) than the MM2 algorithm.
The information from the book is correct - Laguerre method is better in general. It can converge really fast, even from initial guesses that are very far away from the zero point.
Of course MM2 algorithm is able to handle such situations as well, but it takes considerably more iterations for the MM2 method to converge in such a situation, as opposed to Laguerre's method that has a relatively small and constant iteration number (compared to the MM2 method) regardless of the initial guess value.

It should be noted though, that for a case where initial guess is relatively close to the actual zero point, the algorithms have very similar iteration count

End notes and sources

Algorithms might have been slightly changed after their code was posted as screenshots into the report. However, the above-mentioned changes mostly were related to the print-out methods, hence I decided not to update the code screenshots. Refer to original *.m files provided for 100% accurate code. The main principle of the algorithm remained unchanged.

Source and other programs list:

Piotr Tatjewski - Numerical Methods. Every theoretical illustration or text paragraph
Excel was used for algorithm data comparison
Wolfram Alpha was used for zero point computation for task 1 and for equation system solving for a and b for muller_1 method.