

Sunday, 9 December 2018

***Michał Tchórzewski***

Numerical Method

Project B No. 30

## Task 1

### 1.1. Aim

The aims of this task are finding zeros of given function:

$$f(x) = 0.7x * \cos(x) * \log(x + 1)$$

in the interval 2 lower bound and 11 upper bound.

With two methods:

1. The Bisection Method
2. The Newton's Method

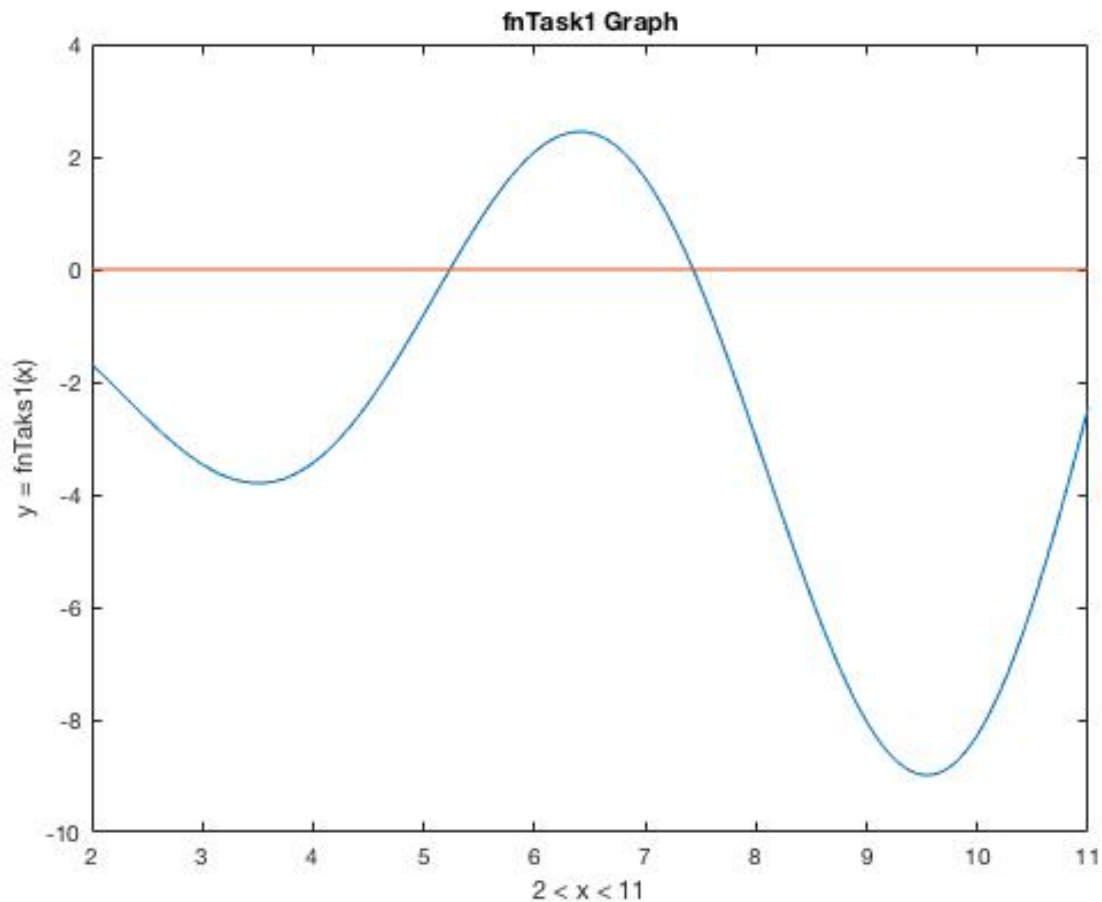
Implementation of algorithm in MatLab environment is included in report appendix, after coverage of third task.

### 1.2. Purpose

Both methods are meant to find function roots in given interval. With only difference in implementation complexity and effectiveness. The case A method is globally convergent (algorithm finds solution in every case) but very slow. The case B method is way faster than A, but only locally convergent. The combination of this two compromises both effectiveness and convergence capability. Therefore, it is the best option for finding function zeros.

### 1.3. Graph of $f(x)$

Graph below shows approximate plot of given function marked as blue solid line and  $y = 0$  orange line. Cross of this two indicates roots of  $f(x)$  in interval  $2 < x < 11$ . In the graph we can assume to initial intervals within which roots are located. Namely, first interval  $4.5 < x < 6$  and second  $7 < x < 8$ . In following solutions I will reuse intervals defined on graph to obtain exact root.



#### 1.4. The Bisection Method Description

The idea of Bisection Method is really simple. Basically divide range of number in exact two until middle point. The point created in exact half of previous interval, is equal to function root.

More formally

Let consider initial interval  $[a, b]$ , containing a root of the function. At every iteration define middle point, according to formal:

$$c_n = (a_n + b_n) / 2$$

Where,  $n$  denote iteration number. In next stem of algorithm define new intervals as follows  $[a_n, c_n]$  and  $[c_n, b_n]$ . Now, determine whether new interval contains root and choose it for iteration. Stop test should check value of function in middle point and compere it to

assumed tolerance and to avoid errors in case of flat function or incorrect initial range of x script should keep track of interval length.

Accuracy of the solution is obtained the number of iteration performed, it does not depend on accuracy of calculation of the function values.

$$\varepsilon_{n+1} = (1/2) \varepsilon_n$$

Where  $\varepsilon_n$  denotes interval length. Formula above shows that the bisection method is linearly convergent ( $p = 1$ ), with the convergence factor  $k = 1/2$ .

## 1.5. The Bisection Method Solution

For initial interval defined in section above I obtained exact root of  $f(x)$  using method called ***the\_bisection\_v2***. Which implementation can be found in appendix chapter.

$f(x) = 0$	Exact root	Accuracy	Initial interval	Iteration Number
$x_0$	5.235316312930081	1.746229827404022e-10	$4.5 < x < 6$	33
$x_1$	7.431717765372014	2.910383045673370e-11	$7 < x < 8$	35

## 1.5. The Newton's Method Description

The idea of the method is as follows: one starts with an initial guess which is reasonably close to the root, then the function is approximated by its tangent line and with use of formula next point closer to the root is defined methods end iteration when next point is equal to the equation root or difference between previous and current point value is less or equal to assumed tolerance.

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

## 1.6. The Newton's Method Solution

For the initial guess defined in section above I obtained exact root of  $f(x)$  using method called ***the\_newton\_v2***. Which implementation can be found in appendix chapter.

$f(x) = 0$	Exact root	Initial guess	Interval	Iteration Number
$x_0$	5.235316312914112	4.5	$4.5 < x < 6$	5
$x_1$	7.431717765372166	7	$7 < x < 8$	5

## 1.7. Conclusion

The bisection method require significant amount of time. The solution to any function needs to iterate numerous time before result can be presented. Although it is always convergent. The accuracy of the solution obtained by bisection method depends only on the number of iteration performed, it does not depend on the accuracy of calculations of the functions value. The method is imprecise when the function is very flat, its derivative has small absolute value in a neighbours of the root.

## Task 2

Due to fact task 2 and 3 are similar with only difference method used to obtain result. I will cover them in one chapter.

### 2.1. Aim

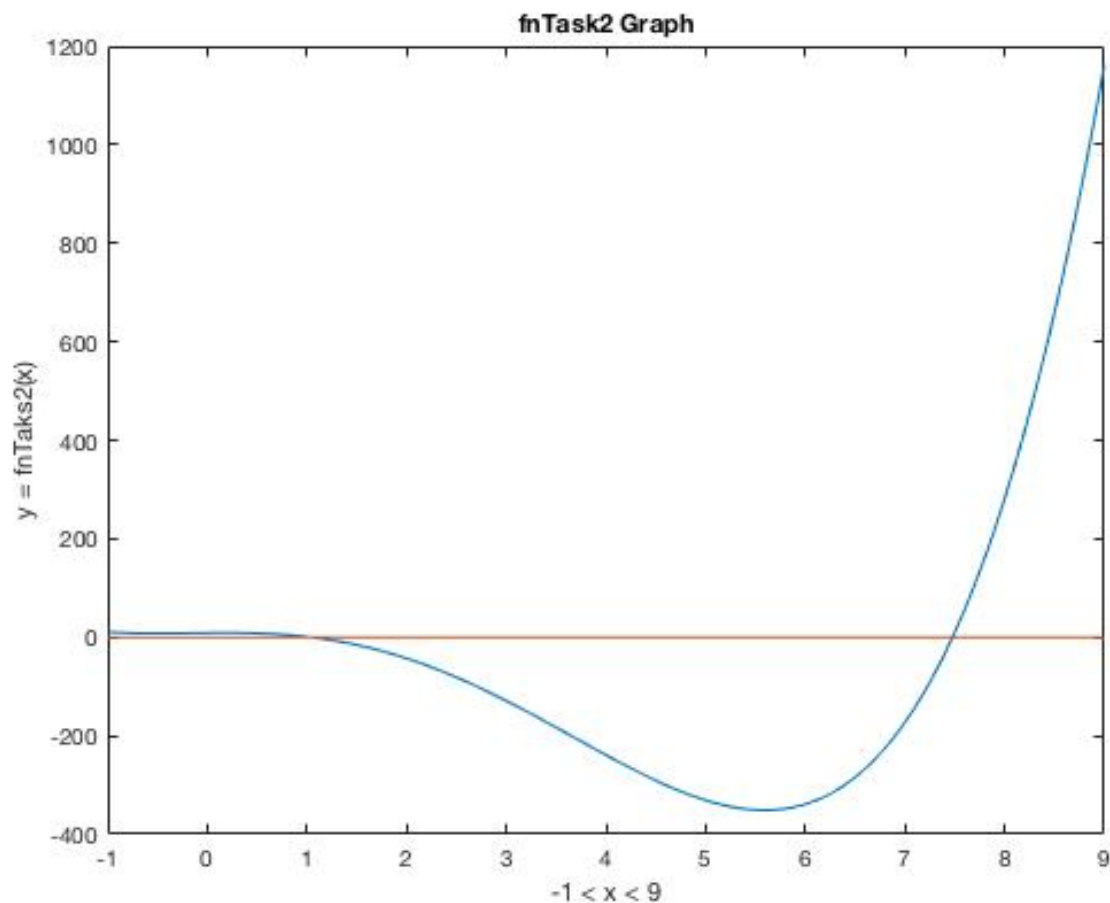
The aims of this task are finding real and complex zeros of given function:

$$f(x) = x^4 - 7x^3 - 4x^2 + 2x + 9$$

With methods:

1. The Muller Method in version MM1
2. The Muller Method in version MM2
3. The Newton Method from previous Task 1

Implementation of algorithm in MatLab environment is included in report appendix, after coverage of third task.

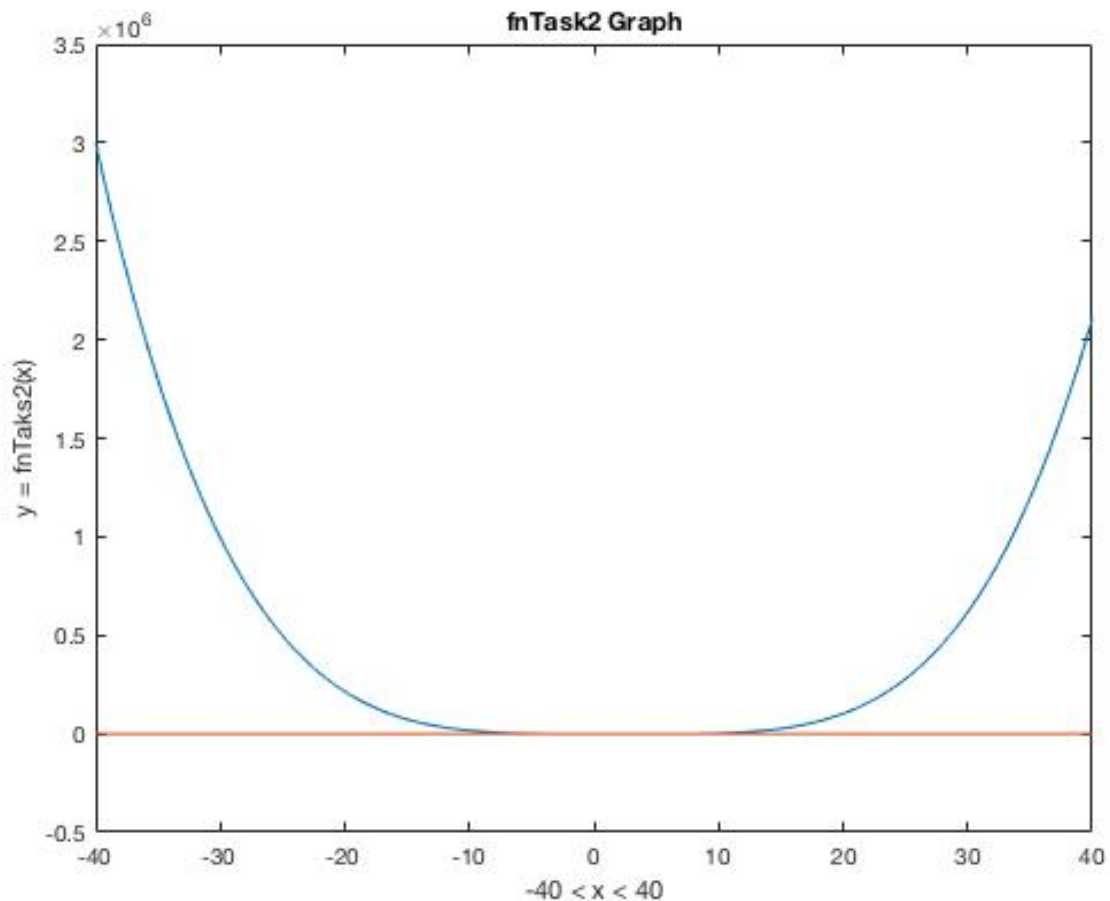


## 2.2. Purpose

Muller method is meant to find both complex and real roots of polynomial. Although, previous method can be applied, this is more general and accurate.

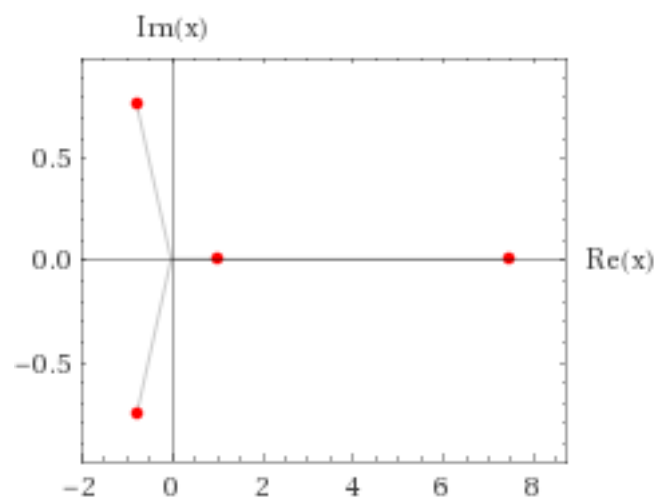
## 2.3. Graph of polynomial

Graph below shows approximate plot of given function marked as blue solid line



and  $y = 0$  orange line. Cross of this two indicates roots of  $f(x)$  in interval  $-40 < x < 40$ , we can see bigger picture how polynomial looks in bigger scale. In order to define interval, which contains roots we need better scope shown on second graph in range  $-1 < x < 9$ . We can assume to initial intervals within which roots are located. Namely, first interval  $0.5 < x < 2$  and second  $7 < x < 8$ . In following solutions I will reuse intervals defined on graph to obtain exact root.

Complex roots are located near  $x = -0.76$ , and are conjugated.



## 2.4. The Muller Method Description

### 2.4.1. MM1

Muller's method is a recursive method which generates an approximation of the root of  $f$  at each iteration. Starting with the three initial values  $x_0$ ,  $x_{-1}$  and  $x_{-2}$ , the first iteration calculates the first approximation  $x_1$ , the second iteration calculates the second approximation  $x_2$ , the third iteration calculates the third approximation  $x_3$ , etc. Hence the  $k^{\text{th}}$  iteration generates approximation  $x_k$ . Each iteration takes as input the last three generated approximations and the value of  $f$  at these approximations. Hence the  $k^{\text{th}}$  iteration takes as input the values  $x_{k-1}$ ,  $x_{k-2}$  and  $x_{k-3}$  and the function values  $f(x_{k-1})$ ,  $f(x_{k-2})$  and  $f(x_{k-3})$ . The approximation  $x_k$  is calculated as follows.

A parabola  $y_k(x)$  is constructed which goes through the three points  $(x_{k-1}, f(x_{k-1}))$ ,  $(x_{k-2}, f(x_{k-2}))$  and  $(x_{k-3}, f(x_{k-3}))$ . When written in the Newton form,  $y_k(x)$  is

$$y_k(x) = f(x_{k-1}) + (x - x_{k-1})f[x_{k-1}, x_{k-2}] + (x - x_{k-1})(x - x_{k-2})f[x_{k-1}, x_{k-2}, x_{k-3}],$$

where  $f[x_{k-1}, x_{k-2}]$  and  $f[x_{k-1}, x_{k-2}, x_{k-3}]$  denote divided differences. This can be rewritten as

$$y_k(x) = f(x_{k-1}) + (x - x_{k-1})f[x_{k-1}, x_{k-2}] + (x - x_{k-1})(x - x_{k-2})f[x_{k-1}, x_{k-2}, x_{k-3}],$$

where

$$w = f[x_{k-1}, x_{k-2}] + f[x_{k-1}, x_{k-3}] - f[x_{k-2}, x_{k-3}].$$

The next iterate  $x_k$  is now given as the solution closest to  $x_{k-1}$  of the quadratic equation  $y_k(x) = 0$ . This yields the recurrence relation

$$x_k = x_{k-1} - \frac{2f(x_{k-1})}{w \pm \sqrt{w^2 - 4f(x_{k-1})f[x_{k-1}, x_{k-2}, x_{k-3}]}}.$$



In this formula, the sign should be chosen such that the denominator is as large as possible in magnitude. We do not use the standard formula for solving quadratic equations because that may lead to loss of significance.

Note that  $x_k$  can be complex, even if the previous iterates were all real. This is in contrast with other root-finding algorithms like the secant method, Sidi's generalized secant method or Newton's method, whose iterates will remain real if one starts with real numbers. Having complex iterates can be an advantage (if one is looking for complex roots) or a disadvantage (if it is known that all roots are real), depending on the problem.

### 2.4.2. MM2

MM2 version is very similar to MM1 with difference that interpolating parabola coefficient are calculated from first and second order polynomial derivative. So it is more numerical demanding due to fact in same time machine has to calculate three different points.

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

$$y'(0) = b = f'(x_k)$$

$$y''(0) = a = f''(x_k)$$

In next step determine roots of interpolating parabola according to formula

$$z_{+,-} = (-2 * a(1)) / (a(2) + \sqrt{a(2)^2 - 2 * a(1) * a(3)})$$

For the next iteration choose  $z$  which has smaller absolute value ( $z_{\min}$ ) and new initial guess is as follows

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

The method is convergent locally so initial guess must be relatively close to the root of function.

## 2.5. The Muller Method Solution

### 2.5.1. MM1

$f(x) = 0$	Exact root	Interval $x_0 \ x_1 \ x_2$	Iteration Number
$x_0$	1.042010761009818	-1, 0, 1	5
$x_1$	7.477634623736573	7, 7.25, 7.5	3
$x_2$	-0.759822692373253 + 0.760087836981814i	-1, -0.5, -0.7	11
$x_3$	-0.759822692373253 - 0.760087836981814i	-1, -0.5, -0.7	11

### 2.5.2. MM2

$f(x) = 0$	Exact root	Initial Guess	Iteration Number
$x_0$	2.863225402935599	0.9	1
$x_1$	7.477634623736573	7	5
$x_2$	-0.743188915180179 + 0.806712298359589i	-1	3
$x_3$	-0.787889055015711 - 0.762013522690888i	-0.8	3

First root cannot be found in exact value, algorithm returns various value for different initial guesses close to exact root.

## 2.6. The Newton method Solution

$f(x) = 0$	Exact root	Initial guess	Interval	Iteration Number
$x_0$	Not Found	-1	$-1 < x < 0$	0
$x_1$	7.477634623736573	7	$7 < x < 8$	5

## 2.7. Conclusion

MM1 version is very handy due to fact that is convergent globally and initial guesses does only matter to iteration number after which solution is found. It's implementation is complicated and took me great amount of time. MM1, struggle to found both complex root, we can assume that second is conjugated of first.

MM2 version could not found root near  $x = 1$ , I suppose it's caused by the fact function is flat in that region so first and second order derivation used in algorithm will return similar value so interpolating parabola will be really flat so we can't exactly found precisely roots. Advantage of this version is quicker convergence in some point and more precise complex solution.

Newton Method was unable to find first polynomial root due to fact that function in this area is very flat with in case of method generates large numerical error or even lack to converge although roots is located within passed interval as initial parameter.

### Task 3

Third task uses polynomial presented in task 2 as well as aims.

#### 3.1. Laguerre Method description

In numerical analysis, Laguerre's method is a root-finding algorithm tailored to polynomials. In other words, Laguerre's method can be used to numerically solve the equation  $p(x) = 0$  for a given polynomial  $p(x)$ . One of the most useful properties of this method is that it is, from extensive empirical study, very close to being a "sure-fire" method, meaning that it is almost guaranteed to always converge to some root of the polynomial, no matter what initial guess is chosen.

The algorithm of the Laguerre method to find one root of a polynomial  $p(x)$  of degree  $n$  is:

1. Choose an initial guess  $x_0$
2. For  $k = 0, 1, 2, \dots$
3. If  $p(x)$  is very small, exit the loop
4. Calculate

$$G = \frac{p'(x_k)}{p(x_k)}$$

5. Calculate

$$H = G^2 - \frac{p''(x_k)}{p(x_k)}$$

6. Calculate  $a$ , where the sign is chosen to give the denominator with the larger absolute value, to avoid loss of significance as iteration proceeds.

$$a = \frac{n}{G \pm \sqrt{(n-1)(nH - G^2)}}$$

7. Set

$$x_{k+1} = x_k - a$$

8. Repeat until  $a$  is small enough or if the maximum number of iterations has been reached.

If a root has been found, the corresponding linear factor can be removed from  $p$ . This deflation step reduces the degree of the polynomial by one, so that eventually, approximations for all roots of  $p$  can be found. Note however that deflation can lead to approximate factors that differ significantly from the corresponding exact factors. This error is least if the roots are found in the order of increasing magnitude.

### 3.2. Laguerre Method solution

$f(x) = 0$	Exact root	Initial Guess	Iteration Number
$x_0$	1.055621848042888	0.9	1
$x_1$	7.477634623736573	7	5
$x_2$	-0.452883437700984 - 1.396620441130435i	-1	11
$x_3$	-0.766502617912967 - 0.773867634805335i	-0.8	4

### 3.3. Conclusions

Laguerre Method is more general and convergent globally. Solutions are obtained a bit slower but value is more precise. And can be found with any initial point.

## 4. Appendix

This section of report include code used to obtain result presented in text above. All MatLab code is shown inside box with black bold boarder. Proceeded with short description of functionality. I decided to implement all methods in recursive way. However, tasks require table of each successive iteration with arguments and returned value. At the end of paragraph, additional script is presented. Which consist of needed code changes in response to demand.

### 4.1. Bisection Method

Function input arguments are lower and upper bound of given interval, defined by user or external script, and current iteration count. Three variables are returned in case of success root, iteration number and accuracy of obtained solution.

```
function [root, iter_num, accuracy] = the_bisection_v2(lower_bound,
                                                    upper_bound, iter_num)

    middle_point = (lower_bound + upper_bound) / 2;
    fn_middle_point = fnTask_1(middle_point);

    if (abs(fn_middle_point) <= 1e-10
        && upper_bound - lower_bound < 0.1)

        %--- Test for solution accuracy ---%
        %--- and interval length ---%
        accuracy = 1/2 * (upper_bound - lower_bound);
        iter_num = iter_num + 1;
        root = middle_point;
    elseif (fn_middle_point * fnTask_1(upper_bound) < 0)
        %--- root is located between
            middle point and upper bound ---%
        iter_num = iter_num + 1;
        [root, iter_num, accuracy] = the_bisection_v2(middle_point,
                                                    upper_bound, iter_num);
    elseif (fnTask_1(lower_bound) * fn_middle_point < 0)
        %--- root is located between
            lower bound and middle point ---%
        iter_num = iter_num + 1;
        [root, iter_num, accuracy] = the_bisection_v2(lower_bound,
                                                    middle_point, iter_num);
    else
        error('Interval does not consist of function root.');
```

end

In case of failure error message is displayed and script is determined. Code above is exact translation of method to MatLab environment. Flow control is included inside boarder.

## 4.2. Newton's Method

Function input arguments are lower and upper bound of given interval, defined by user or external script, variable x obtained in previous iteration and current iteration count. Two variables are returned in case of success root, iteration number.

```
function [root, iter_num] = the_newton_v2(lower_bound, upper_bound, x,
                                         iter_num)

    iter_num = iter_num + 1;

    x_next = x - (fnTask_1(x) / dfnTask_1(x));

    if (x_next < lower_bound || x_next > upper_bound)
        error ('Method is divergent over given interval');
    elseif (abs(x-x_next) <= 1e-10)
        root = x_next;
    else
        [root, iter_num] = the_newton_v2(lower_bound, upper_bound,
                                         x_next, iter_num);
    end

end
```

In case of failure error message is displayed and script is determined. Code above is exact translation of method to MatLab environment. In first instruction, script increments iteration count. Later passes to book's defined formula for next variable x, called Newton's method. Inside if control flow, determine solution accuracy. Satisfying, difference between current and previous solution is less or equal to assumed tolerance, return root of function found in given range. Else call function itself.

### 4.3. Additional task 1 scripts

Given function in task 1, is represented as external function. It is much more effective and simpler, than symbolic function available in MatLab.

```
function y = fnTask_1(x)

    y = 0.7 * x * cos(x) - log(x + 1);

end
```

Case b, the Newton's method also require first order derivative of function above.

Drawback of assumed above function definition is every operation has to be defined in discrete file and manually. Functions return value of function/derivative in given point.

```
function y = dfnTask_2 (x)

    y = 4 * x^3 - 21 * x^2 - 8 * x + 2;

end
```

In order to doge any silly mistakes, derivation function was checked with wolfram site.

### 4.4. Müller's Method version MM1

Function input arguments are initial guesses of points, defined by user or external script and current iteration count. Two variables are returned in case of success root, iteration number.

```
function [root, iteration] = Muller_1_v2 (x_0, x_1, x_2, iteration)

    iteration = iteration + 1;

    %--- Incremental variable z ---%
    z_0 = x_0 - x_2;
    z_1 = x_1 - x_2;

    A = zeros(2, 2);
```



```

%--- Create&solve system of linear equations, Ax = b ---%
%--- Where x consist of a and b parabola coefficients ---%
A(1, 1) = z_0;
A(1, 2) = z_0^2;
A(2, 1) = z_1;
A(2, 2) = z_1^2;

b(1) = fnTask_2(x_0) - fnTask_2(x_2);
b(2) = fnTask_2(x_1) - fnTask_2(x_2);

%--- Quadratic Function Coefficients ---%
a(1) = fnTask_2(x_2); %--- c
a(2:3) = A\b; %--- b - second element,
               c - third element in vector a

%--- Roots of parabola interpolation ---%
z_root_1 = ( -2 * a(1) )
           / ( a(2) + sqrt(a(2)^2 - 4 * a(1) * a(3)) );

z_root_2 = ( -2 * a(1) )
           / ( a(2) - sqrt(a(2)^2 - 4 * a(1) * a(3)) );

if abs( z_root_1 ) <= abs( z_root_2 )
    z_min = z_root_1;
else
    z_min = z_root_2;
end

new_point = x_2 + z_min;

if abs( fnTask_2(new_point) ) < 1e-10
    root = new_point;
elseif new_point >= x_0 && new_point < x_1
    [root, iteration] = Muller_1_v2 (x_0, new_point,
                                   x_1, iteration);
elseif new_point >= x_1 && new_point < x_2
    [root, iteration] = Muller_1_v2 (x_1, new_point,
                                   x_2, iteration);
elseif new_point < x_1
    [root, iteration] = Muller_1_v2 (new_point, x_0,
                                   x_1, iteration);
else
    [root, iteration] = Muller_1_v2 (x_1, x_2, new_point,
                                   iteration);
end

end

```

I decided, the best possible way of finding coefficient of quadratic interpolation based on three points, is creation of matrix A and vector b.

So it is possible to solve it as set of linear equation and solution is simply obtained by expression  $Ax = b$ . The last part of code determines the smallest absolute value of obtained interpolated parabola's roots. After that, arranges order of next points passed as argument to function itself.

#### 4.5. Müller's Method version MM2

Function input arguments are initial guess point, defined by user or external script and current iteration count. Two variables are returned in case of success root, iteration number.

```
function [root, iteration] = Muller_2(initial_guess, iteration)

    iteration = iteration + 1;

    %--- Quadratic Function Coefficients ---%
    a(1) = fnTask_2(initial_guess); %--- c
    a(2) = dfnTask_2(initial_guess); %--- b,
                                     dfn - first order derivative
    a(3) = ( ddfnTask_2(initial_guess) ) / 2; %--- a,
                                     ddfn - second order derivative

    %--- Roots of parabola interpolation ---%
    z_root_1 = ( -2 * a(1) )
               / ( a(2) + sqrt( a(2)^2 - 2 * a(1) * a(3) ) );

    z_root_2 = ( -2 * a(1) )
               / ( a(2) + sqrt( a(2)^2 + 2 * a(1) * a(3) ) );

    if abs(z_root_1) < abs(z_root_2)
        z_min = z_root_1;
    else
        z_min = z_root_2;
    end

    if (fnTask_2(initial_guess + z_min) >= 1e-10)
        [root, iteration] = Muller_2(initial_guess + z_min,
                                     iteration);
    else
        root = initial_guess + z_min;
    end

end
```

## 4.6. Laguerre Method

Function input arguments are initial guess point, defined by user or external script and current iteration count. Two variables are returned in case of success root, iteration number.

```
function [root, iteration] = Laguerre(initial_guess, iteration)

    iteration = iteration + 1;

    %--- Quadratic Function Coefficients ---%
    a(1) = fnTask_2(initial_guess); %--- c
    a(2) = dfnTask_2(initial_guess); %--- b,
                                     dfn - first order derivative
    a(3) = ( ddfnTask_2(initial_guess) ) / 2; %--- a,
                                     ddfn - second order derivative

    denominator_1 = ( a(2) + sqrt( 3 * ( 3 * a(2)^2 - 2 * a(1)
                                     * a(3) ) ) );

    denominator_2 = ( a(2) - sqrt( 3 * ( 3 * a(2)^2 - 2 * a(1)
                                     * a(3) ) ) );

    if abs( denominator_1 ) > abs( denominator_2 )
        initial_guess = initial_guess - ( 4 * a(1) ) / ( a(2)
        + sqrt( 3 * ( 3 * a(2)^2 - 2 * a(1) * a(3) ) ) );
    else
        initial_guess = initial_guess - ( 4 * a(1) ) / ( a(2)
        - sqrt( 3 * ( 3 * a(2)^2 - 2 * a(1) * a(3) ) ) );
    end

    if ( fnTask_2(initial_guess) >= 1e-10 )
        [root, iteration] = Laguerre(initial_guess, iteration);
    else
        root = initial_guess;
    end
end
```

Denominator is introduced in purpose to determine greater one. Inside if statement solution is undergone to tolerance test. Failure indicates next iteration, method is called itself with new initial guess, passed as parameter.

## 4.7. Addition to MM2 and Laguerre's Method

Last two introduced methods required additional functions, to calculate second order derivative. As mentioned before I had to do this by hand and checked it with wolfram alpha site. Function returns value of derivative in given point.

```
function y = ddfnTask_2(x)
    y = 12 * x^2 - 42 * x - 8;
end
```

## 5.2 List of references

1. [https://en.wikipedia.org/wiki/Laguerre%27s\\_method](https://en.wikipedia.org/wiki/Laguerre%27s_method)
2. <https://www.wolframalpha.com/>
3. [https://en.wikipedia.org/wiki/Muller%27s\\_method](https://en.wikipedia.org/wiki/Muller%27s_method)
4. Piotr Tatjewski, Numerical Methods - Chapter 6. Nonlinear equations and roots of polynomials.