Numerical Methods

ProjectAssignment B: Nonlinear algebraic equations

May 11, 2023

Krzysztof Watras

Tutor's name: Jakub Wagner

Computer Science

Warsaw University of Technology,
Faculty of Electronics and Technology

I declare that this piece of work, which is the basis for recognition of achieving learning outcomes in the Numerical Methods course, was completed on my own.
Krzysztof Watras

Contents

Introduction

In this report I analyze the difference between methods of finding the solutions to nonlinear algebraic equations. By solving the equation I mean finding such x such that $f(x) = 0$. I will compare methods given by comparing them to **fzero** function as a reference method built-in to MATLAB language. Function that I use in the report is the following:

$$f(x) = 2.5 \cdot \cos^3(-\frac{x}{7} - 1.5) - 0.01 \cdot \left(\frac{x}{3}\right)^3 + 2, \text{ where } x \in [0, 10] \quad (1)$$

Function is visualized on Figure 1

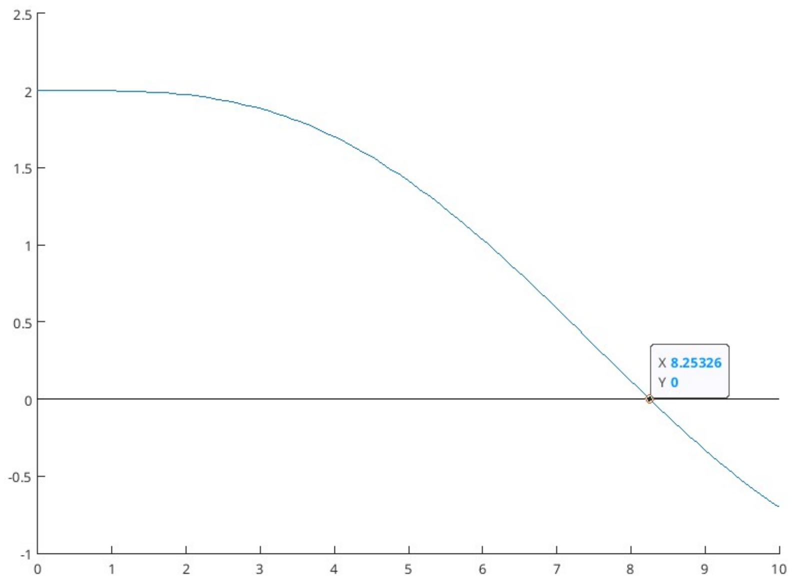

Figure 1: Function f(x) and the intersection with the y=0

As one can clearly see, the intersection is found at the point $x = 8.25326$. This is verified by **fzero** function that returns 8.2533 and after use of *format long* to increase number of digits displayed we get 8.253263117902842, which is consistent with the observation from the graph.

All of my algorithms will use common interface to all:
As input arguments I'll use:

- **f** is a *function handle* of function analyzed
- **x** is a 2 element vector containing the start and end of the interval
- **xeps** is a maximal error that we allow this function to return

And the return will be a vector of consecutive solutions. This will allow for easier testing as well as comparison between solutions.

Only exception from this standard will be Newton's method where **x** represents the starting point for the algorithm. This is because newton's method needs only one point to work.

When comparing the algorithms, the ρ property will be stated. It is an exponent of convergence and is a good measure that informs how quickly the algorithm converges to the true correct value.

Bisection method

Bisection method is an iterative algorithm used for finding roots of an equation. It works by continuously splitting the initial interval into halves, until the solution is found. Here, the exponent of convergence $\rho = 1$.

My solution in code:

```
function result = bisection(f, x, xeps)
    a=x(1); b=x(2);
    i = 1;
    while abs(0.5*(a-b)) > xeps
        result(i) = 0.5 * (a+b);
        if (f(a) * f(result(i)) < 0)
            b = result(i);
        else
            a = result(i);
        end
        i = i + 1;
    end
end
```

This can be summarized by following steps:

1. Initialize variables
2. Is maximal error smaller than the specified one? End if true
3. Save result of current approximation to return vector
4. Adjust interval accordingly
5. Goto step 2

We find max error by calculating $|\frac{a-b}{2}|$. If this error is below the intended threshold we stop the algorithm. This is because in the worst case our solution is only half of the interval length away from true root of the equation.

We verify that this with a graph of total error. As one can clearly see, from Figure 2, the error indeed goes down and algorithm stops around the expected 10^{-12} value of error. Note: Algorithm does not stop exactly as the "true error" hits 10^{-12}. This is because algorithm does not know this "true error". Instead, it knows what is the max error at each iteration. Hence, it continues to run for a couple more iterations.

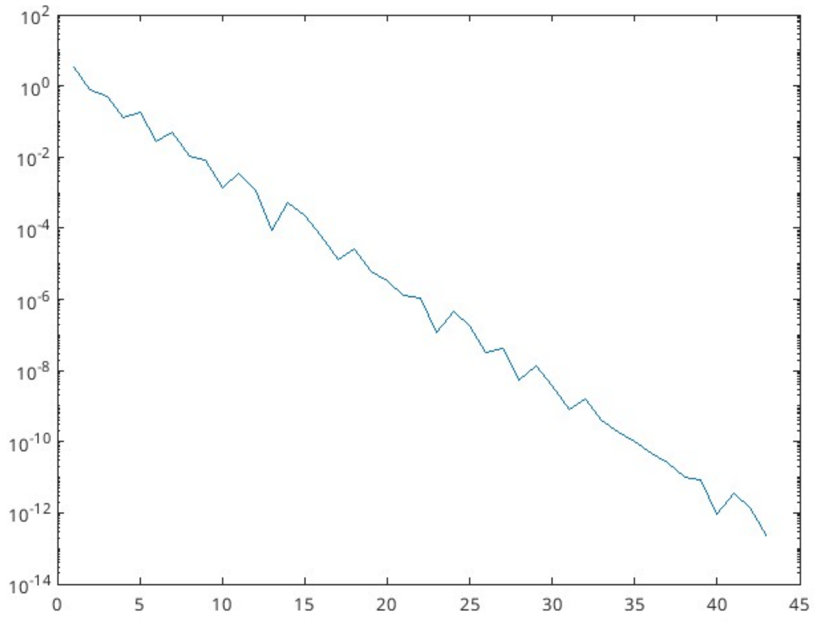

Figure 2: Absolute error in each iteration of bisection method

Secant Method

The secant method is an iterative root-finding algorithm that uses successive solutions of secant lines to approximate a root of a function f. It has a $\rho = 1.618$ or $\frac{\sqrt{5}+1}{2}$. Following equation describes the algorithm.

$$x_{i+1} = x_i - \frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})}$$

It is similar to the Newton-Raphson method, but instead of using the derivative of the function, it uses a numerical approximation of the derivative.

My solution in code:

```
function result = secant(f, x, xeps)
    result = x;
    i = 2;
    while abs(result(i)-result(i-1)) > xeps
        xi = result(i);
        xi1 = result(i-1);
```

```
            fx = f(xi);
            fx1 = f(xi1);
            result(i+1) = xi - (xi-xi1)/(fx-fx1)*fx;

            i = i + 1;
        end
    end
```

This can be summarized by following steps:

1. Initialize variables
2. Is maximal error smaller than the specified one? End if true
3. Calculate the secant of point we are at
4. Set variables for next iteration
5. Goto step 2

Newton's Method

Newton's method is perhaps the most famous method of solving nonlinear equations. It is very efficient method having a ρ factor of 2. It was also famously used in Fast Inverse Of Square Root algorithm by William Kahan [2]. Following equation describes the algorithm.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

My solution in code:

```
function result = newton(f, x, xeps)
    fdx = @(x) 1/eps* imag(f(x + 1i*eps));

    result(1) = x;
    i = 1;
    while 1
        xi = result(i);
        fx = f(xi);
        fxdelta = fdx(xi);
        result(i+1) = xi - fx/fxdelta;
        i = i + 1;
        if abs(result(i)-result(i-1)) < xeps, break; end
    end
end
```

This can be summarized by following steps:

1. Initialize variables
2. Is maximal error smaller than the specified one? End if true
3. Calculate the secant of point we are at
4. Set variables for next iteration
5. Goto step 2

Note: In my implementation I use following property to calculate the derivative of function.

$$f'(x) = \frac{Imag(f(x + i\cdot))}{}$$

This was introduced to us during the consultations for this rapport. A good paper on this property is "Using Complex Variables to Estimate Derivatives of Real Functions" by Lyness and Moler, which was published in SIAM Journal on Numerical Analysis in 1967 [3]. What is fascinating about this property is that it works for calculating the first derivative of function, but does not calculate the second derivative of the same function.

Muller's Method, second version

The method is iterative, meaning it uses an initial guess and then improves upon it with each iteration. The algorithm requires three initial guesses, $x_0, x_1,$ and x_2. Here we use $x_{min}, x_{avg},$ and x_{max} respectively. Muller Method has ρ value equal to 1.84 so lower that the Newton's method, but higher than secant method.

Following equation describes the algorithm.

$$M = \begin{bmatrix} a_i \\ b_i \end{bmatrix} = \begin{bmatrix} -(x_i - x_{i-1})^2 & (x_i - x_{i-1}) \\ -(x_i - x_{i-2})^2 & (x_i - x_{i-2}) \end{bmatrix}^{-1} \cdot \begin{bmatrix} f(x_i) - f(x_{i-1}) \\ f(x_i) - f(x_{i-2}) \end{bmatrix}, \text{ and } c_i = f(x_i)$$

And then, the iterative formula is equal to

$$x_i = x_i - \frac{2c_i}{b_i + sgn(b_i)\sqrt{b_i^2 - 4a_i c_i}}$$

My solution in code:

```
function result = muler2(f, x, xeps)
    result(1) = x(1); result(2) = mean(x); result(3) = x(2);
    i = 3;
    while abs(result(i)-result(i-1)) > xeps
        matrix = [
            -(result(i)-result(i-1))^2 (result(i)-result(i-1));
            -(result(i)-result(i-2))^2 (result(i)-result(i-2));
            ];
        vec = [
            f(result(i))-f(result(i-1))
            f(result(i))-f(result(i-2))
            ];
        foo = matrix\vec;
        ai = foo(1); bi = foo(2);
        ci = f(result(i));
        result(i+1) = result(i) - 2*ci/(bi + sign(bi)*sqrt(bi^2 + 4*ai*ci )
        i = i + 1;
    end
end
```

Comparison of different methods

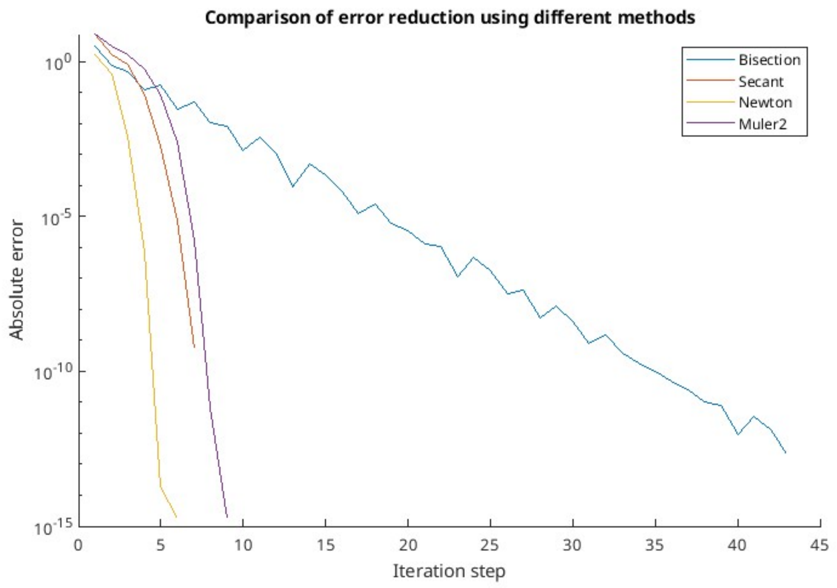

Figure 3: Absolute error in each iteration of different methods

Figure 3 shows the consecutive result error for each algorithm. As one can see, the worst method in terms of number of iterations is the bisection method. The best one was Newton's method. Secant and Muller methods are comparable, though secant method performed better. This observation is consistent with the ρ properties of those algorithms. Newton's method has highest value of ρ and converges the fastest to true value. Bisection method has lowest value of ρ and converges the slowest to true value. Secant method has ρ comparable to Muller's method and has similar convergence. However, it did perform better while calculating this particular function, despite lower ρ value. This is a good reminder that while general rules are good guidelines, they are not definite and to be sure one needs to test their solutions to be sure.

Crucially, the scalability of the algorithms is consistent with the results we got in a single error size test. When testing the performance of algorithms across errors $\in [10^{-15}, 10^{-1}]$ we can see which algorithms perform and scale the best. This can be seen on Figure 4.

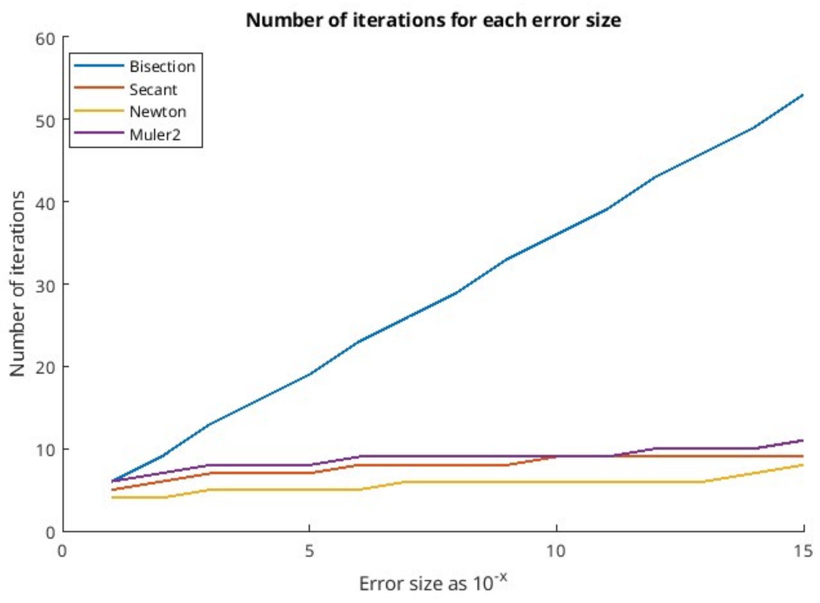

Figure 4: Number of iterations needed to obtained result for each algorithm and error $\in [10^{-15}, 10^{-1}]$

Comparison of theoretical number of iterations and true iteration number in a bisection method

For a bisection method an expected value of iterations is expressed by following formula.

$$I = log_2(\frac{|b-a|}{\delta})$$

which in our function is equal to

$$I = log_2(\frac{10}{\Delta}), \text{ where } \Delta \in [10^{-15}, 10^{-1}]$$

As one can see on Figure 5, the true number of iterations closely follows the theoretical value that one should expect.

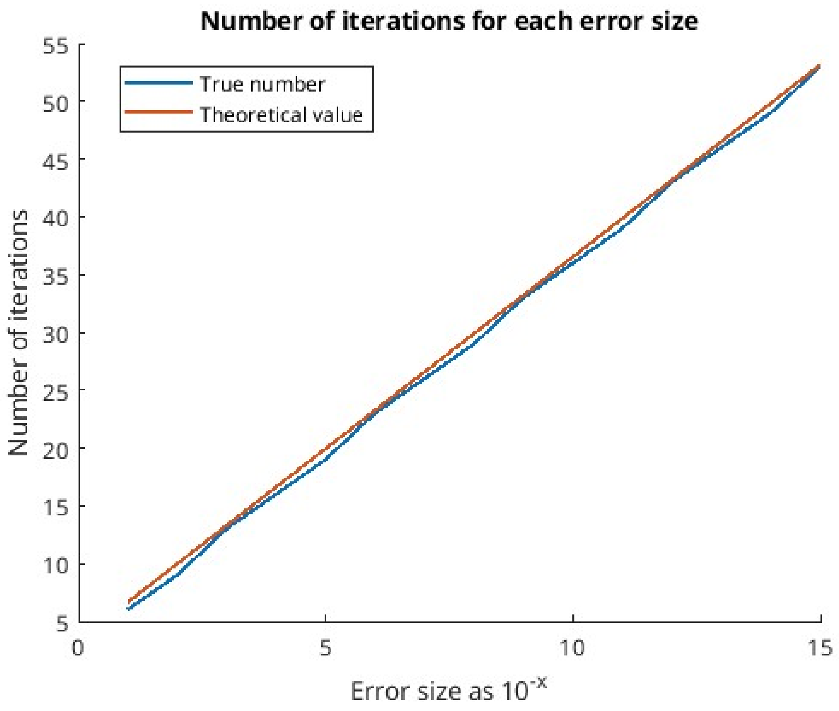

Figure 5: Comparison of true number of iterations and theoretical number needed

Comparison of convergence of Newton's method based on starting point

Without code and simply knowing the function shape and how Newton's method works we can assume that it'll take more iterations for it to converge when starting on the left side of the interval.

Figure 6 shows exactly what we would expect. The difference between results is rather big. This means that although the algorithm is very efficient compared to others, it is beneficial to know what initial value to set.

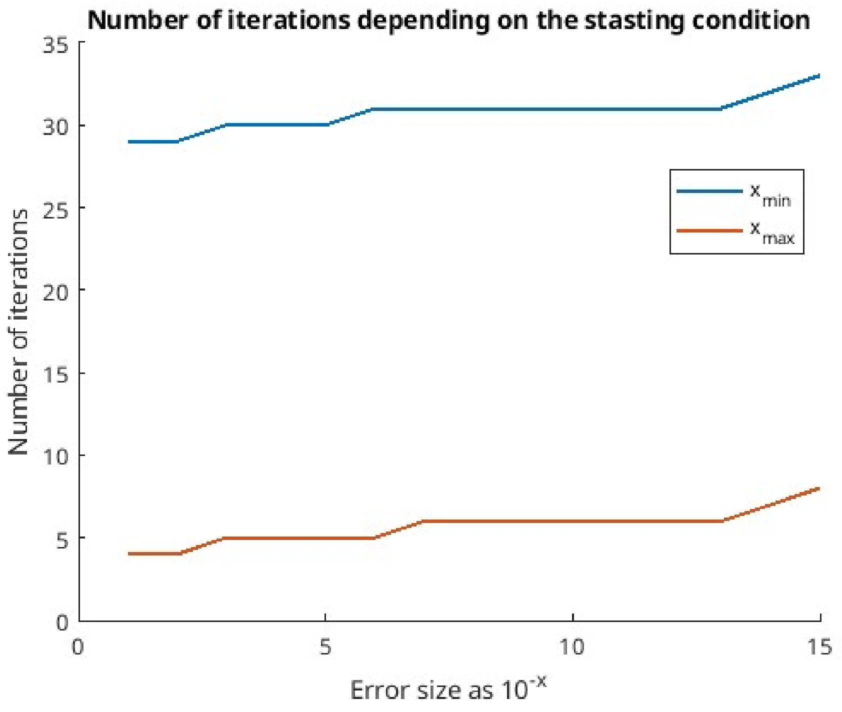

Figure 6: Comparison of iteration number depending on starting condition

References

[1] R. Z. Morawski: Numerical methods (ENUME) – *2. Accuracy and complexity of computing*, Warsaw University of Technology, Faculty of Electronics and Information Technology
[2] https://en.wikipedia.org/wiki/Fast$_{i}nverse_{s}quare_{r}oot$
[3]$https://epubs.siam.org/doi/10.1137/S003614459631241X$

Appendix: Listing of the developed programs

Source code for Task 1

```
% clear previous experiment results
clc, clearvars, close all

% define domain and function
x = linspace(0,10,100);
y = @(x) 2.5 * (cos (-x/7 - 1.5)).^3 - 0.01*(x/3).^3 + 2;
format long
xzero = fzero(y, [0,10])

hold on
% plot the function
plot(x, y(x));
% plot line y=0
line(xlim(), [0,0], 'Color', 'k');
% add the intersection point
plot(xzero, 0, 'o')
hold off
```

Source code for Task 4

```
% clear previous experiment results
clc, clearvars, close all

% define domain and function
x = linspace(0,10,100);
y = @(x) 2.5 * (cos (-x/7 - 1.5)).^3 - 0.01*(x/3).^3 + 2;
range = [0,10];

ref_xzero = fzero(y, range);
our_xzerob = bisection(y,range,1.0e-12);
our_xzeros = secant(y,range,1.0e-12);
our_xzeron = newton(y,10,1.0e-12);
our_xzerom = muler2(y,range,1.0e-12);
errorb = abs(our_xzerob - ref_xzero);
errors = abs(our_xzeros - ref_xzero);
errorn = abs(our_xzeron - ref_xzero);
errorm = abs(our_xzerom - ref_xzero);

% plot the consecutive aproximation errors
hold on
semilogy(1:length(errorb), errorb);
semilogy(1:length(errors), errors);
semilogy(1:length(errorn), errorn);
semilogy(1:length(errorm), errorm);
title("Comparison of error reduction using different methods");
xlabel("Iteration step"); ylabel("Absolute error");
legend("Bisection", "Secant", "Newton", "Muler2");
set(gca, 'YScale', 'log');
hold off
```

Source code for Task 5

```
% clear previous experiment results
clc, clearvars, close all

% define domain and function
x = linspace(0,10,100);
y = @(x) 2.5 * (cos (-x/7 - 1.5)).^3 - 0.01*(x/3).^3 + 2;
range = [0,10];

n_iter_bisection = zeros(15, 1);
for n = 1:15
    roots_bisection = bisection(y,range,10^(-n));
    n_iter_bisection(n) = length(roots_bisection);
end

n_iter_secant = zeros(15, 1);
for n = 1:15
    roots_secant = secant(y,range,10^(-n));
    n_iter_secant(n) = length(roots_secant);
end

n_iter_newton = zeros(15, 1);
for n = 1:15
    roots_newton = newton(y,range(2),10^(-n));
    n_iter_newton(n) = length(roots_newton);
end

n_iter_muller = zeros(15, 1);
for n = 1:15
    roots_muller = muler2(y,range,10^(-n));
    n_iter_muller(n) = length(roots_muller);
end

% plot the consecutive aproximation errors
hold on
plot(1:15, n_iter_bisection, 'LineWidth',1.5);
plot(1:15, n_iter_secant, 'LineWidth',1.5);
plot(1:15, n_iter_newton, 'LineWidth',1.5);
plot(1:15, n_iter_muller, 'LineWidth',1.5);
```

```
title("Number of iterations for each error size");
xlabel("Error size as 10^{-x}"); ylabel("Number of iterations");
legend("Bisection", "Secant", "Newton", "Muler2");
hold off
```

Source code for Task 6

```
% clear previous experiment results
clc, clearvars, close all

% define domain and function
x = linspace(0,10,100);
y = @(x) 2.5 * (cos (-x/7 - 1.5)).^3 - 0.01*(x/3).^3 + 2;
range = [0,10];

foo = @(n) log2(10./(10.^(-n)));
theoretical = feval(foo, 1:15);

n_iter_bisection = zeros(15, 1);
for n = 1:15
    roots_bisection = bisection(y,range,10^(-n));
    n_iter_bisection(n) = length(roots_bisection);
end

% plot the consecutive aproximation errors
hold on
plot(1:15, n_iter_bisection, 'LineWidth',1.5);
plot(1:15, theoretical, 'LineWidth',1.5);
title("Number of iterations for each error size");
xlabel("Error size as 10^{-x}"); ylabel("Number of iterations");
legend("True number", "Theoretical value");
hold off
```

Source code for Task 7

```
% clear previous experiment results
clc, clearvars, close all

% define domain and function
x = linspace(0,10,100);
y = @(x) 2.5 * (cos (-x/7 - 1.5)).^3 - 0.01*(x/3).^3 + 2;
range = [0,10];

n_iter_newton1 = zeros(15, 1);
for n = 1:15
    roots_newton1 = newton(y,range(1),10^(-n));
    n_iter_newton1(n) = length(roots_newton1);
end

n_iter_newton2 = zeros(15, 1);
for n = 1:15
    roots_newton2 = newton(y,range(2),10^(-n));
    n_iter_newton2(n) = length(roots_newton2);
end

% plot the consecutive aproximation errors
hold on
plot(1:15, n_iter_newton1, 'LineWidth',1.5);
plot(1:15, n_iter_newton2, 'LineWidth',1.5);
title("Number of iterations depending on the stasting condition");
xlabel("Error size as 10^{-x}"); ylabel("Number of iterations");
legend("x_{min}", "x_{max}");
hold off
```