AM 2814G: Lab 2B

Aidan Sirbu, Katie Brown
asirbu@uwo.ca, kbrow327@uwo.ca

Western University — January 31, 2021

Abstract

The objective of this lab was to construct an algorithm that could approximate the roots of a function. Several functions were successfully implemented with increasing levels of automation, each taking a function, its first derivative, and some termination criteria as input parameters. The first, *mynewton*, uses Newton's Method to calculate an approximate root, given a sufficiently close initial guess, returning a vector containing the iterates. To analyze the error of successive iterations of *mynewton*, *mynewtonErrorPlot* plots the error, including a line of best fit using method of least squares, as well as determines the order of convergence of the input function. To allow for the approximation of multiple roots, the recursive function *myIntervalSolve* was constructed. This further automated the algorithm by taking an interval as input, which was repeatedly divided into sub-intervals. *mynewton2* was implemented as a subroutine, which approximates the root in each sub-interval. Each function was tested with various input functions, and all proved to successfully calculate the root(s) to the specified level of accuracy.

Introduction and Background

This investigation is based on the root-finding method called *Newton's method* or *Newton-Raphson method*. Newton's method works by first inputting an initial guess for the x-coordinate of a root x_0 into the iterative formula

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

for x_i. This will yield some other x_{i+1} that will be closer to the actual root. This method functions on the basis of finding a tangent of the function $f(x)$ at the initial guess, taking the intersection of that tangent with the x-axis, and then using the x-value of that intersection as the next guess. This method does not always converge towards a root, only for initial guesses in a reasonable interval surrounding the root.

Furthermore, Newton's method can converge towards a root either at a linear or quadratic rate. Simple roots, defined as a roots of multiplicity 1, obey quadratic convergence and are therefore Newton's method converges towards them qualitatively faster than the Bisection method or general Fixed-Point-Iteration methods [1]. For multiple roots, however, Newton's method obeys linear convergence. Quadratic convergence abides by the following property:

$$M = \lim_{i \to \infty} \frac{e_{i+1}}{e_i^2} \tag{1}$$

where e_i is the error at iteration *i*. Linear converge, however, obeys:

$$S = \lim_{i \to \infty} \frac{e_{i+1}}{e_i} \tag{2}$$

The order of convergence is 1 for a linearly convergent root and 2 for a quadratically convergent root.

Task 1 of this investigation shall focus on the construction of a Matlab algorithm which implements Newton's method as well as automating a secondary algorithm which plots the errors at every iteration.

From equations (1) and (2), one can write

$$\frac{|e_{k+1}|}{|e_k|^{\alpha}} \approx C \tag{3}$$

where α is the order of convergence and *C* is some constant. Rearranging and taking the natural logarithm of equation (3) allows one to transform it into a linear equation:

$$\ln|e_{k+1}| \approx \alpha \ln|e_k| + \ln(C) \tag{4}$$

The secondary algorithm will also approximate a line of best fit using the *Method of Least Squares*. From this, the order of convergence can be determined. Task 2 will focus on the automation of another algorithm which will implement the Bisection method alongside Newton's method to allow for the approximation of every root within any given interval.

Task 1

Task 1 begins with designing and implementing a Matlab function that can compute the roots of any equation. To achieve this, the function utilized Newton's method. The code begins with the header:

```
function [x, flag]=mynewton(f,fx,x0,tol,maxiter)
```

The .m file takes *'f'* as its first input to be the function which the algorithm will approximate a root for. *'fx'* is the first derivative of the function *f* as is needed by Newton's method. *'x0'* is the initial guess for a root. This initial guess will determine whether or not the algorithm converges on a solution as well as the amount of iterations that are needed. Parameter 'tol' is the error tolerance. Since Newton's method is a special case of Fixed-Point-Iteration (FPI), a stopping criterion must be determined as FPI's do not have a set error formula such as the Bisection Method has [1]. This tolerance number measures the absolute difference between x_i and x_{i+1} where i is an integer number denoting the iteration in which x was obtained. When the absolute difference, also known as the absolute error, is less than the set tolerance, the algorithm must terminate. The code which governs this behaviour can be seen here:

```
if abs(x0-xnew)<tol
    ...
    break
end
```

Furthermore, the *maxiter* parameter is an integer number denoting the maximum number of iterations the algorithm should perform before terminating. The algorithm is set to terminate when either the absolute error falls below the tolerance, or the maximum number of iterations has been performed. It will terminate as soon as one of those conditions is true. The *for* loop governs the *maxiter* parameter as it is set to run for $1 < i \leq maxiter$

```
for i=1:maxiter
    ...
end
```

The output for the function contains vector *x* which is a representation of all the iterates of Newton's method with the last value being the closest approximation of the root of the function *f*. The function also returns a flag which will denote the number of iterations it has performed before reaching the input tolerance level, or -1 if the maximum number of iterations was reached before reaching the desired accuracy. Refer to Listing 1 in the code appendix for the complete code of mynewton.m.

To continue, the function was tested for correctness by approximating the roots for 2 well known function. First the function $f(x) = (x-1)^2$ was tested as shown below:

```
[x, flag]=mynewton(@(x)(x-1).^2,@(x)2.*(x-1),0.5,0.001,10)
```

The approximated root was returned as $x = 0.9990 \approx 1$ as well as $flag = 9$ implying that 9 iterations were performed before falling below a tolerance level of $tol = 0.001$. Furthermore, the function $f(x) = (x-3)(x+2)(x-1)$ was also tested:

```
[x, flag]=mynewton(@(x)(x-3).*(x+2).*(x-1),@(x)3.*x.^2-4.*x-5,2.6,0.001,10)
```

The approximated root was returned as $x = 3.0000$ and $flag = 4$ showing a convergence to the root accurate to 4 decimal places after 4 iterations.

An error analysis was then performed with the purpose of determining the order of convergence of 2 different functions. To achieve this, a supplementary function titled *mynewtonErrorPlot* was engineered. The idea of this function was to perform the same algorithm as *mynewton*, but instead recording each consecutive iteration error into a matrix. To achieve this, the same code from *mynewton* was used, but the statement

```
errorMatrix = [errorMatrix, abs(x0-xnew)];
```

was added within the *for* loop as to build upon the previously declared empty matrix *errorMatrix*. Since the user may input any maxiter or tol parameter they chose, no preallocation of memory was declared for the matrix and was instead increased in size for every iteration. This, while being inefficient from a perspective of time complexity, allows for any tolerance and maximum number of iterations the user may choose. Next two subarrays labelled ek and ek1 were constructed from the values inside of errorMatrix so that for each value in ek, the value at the same index in ek1 is the subsequent value of ek inside of the errorMatrix. Now, yielding to the linear approximation in equation (4), the natural log of the absolute value of the vectors ek and ek1 represent the x and y coordinates, respectively, of the errors to be plotted.

Next, the challenge of determining constants α and C arose. Rather than hand-drawing a line of best fit and determining the slope and y-intercept, mynewtonErrorPlot was engineered to determine the best fitting line using the method of least-squares. To achieve this, the *Method of Least Squares* was used taken from *Theorem 5.6.4* [2]. This theorem states that given a vector representing the y-coordinates of a set of point, and the form of an approximating function, chosen in this case to be

$$r_0 x + r_1 \tag{5}$$

where $r_0 = \alpha$ and $r_1 = \ln(C)$, one can determine the coefficients of the approximating function using the equation:

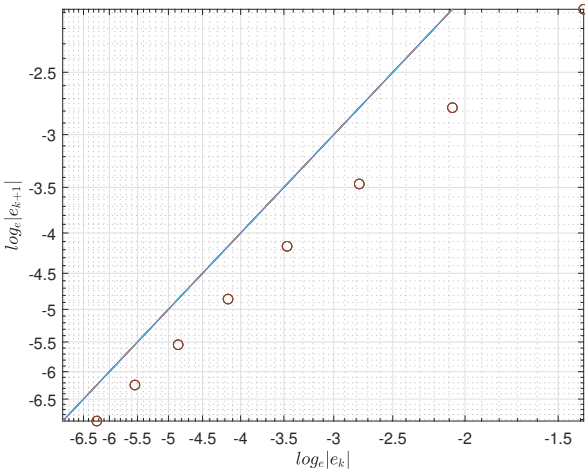
$$r = (M^T M)^{-1}(M^T y) \tag{6}$$

where $r = [r_0, r_1]$ and M is a matrix containing columns representing the approximating functions $r_0 x$ and r_1 evaluated at the x-coordinates of the aforementioned set of points. Details as well as the complete code for *mynewtonErrorPlot* can be found in Listing 2 in the code appendix.

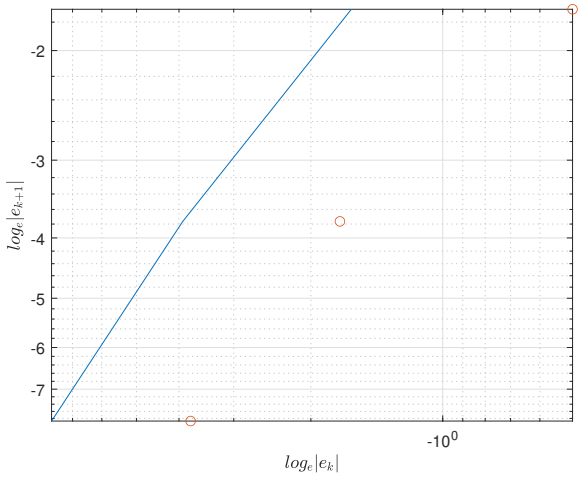
The function *mynewtonErrorPlot* was used to plot, determine a line of best fit for, and obtain the order of convergence α of the previously tested functions $f_1(x) = (x-1)^2$ with initial guess $x_0 = 0.8$ and $f_2(x) = (x-3)(x+2)(x-1)$ with initial guess $x_0 = 2.6$. The plots of the errors can be observed in Figure 1. Figure 1(a) was created using $f_1(x)$ with initial guess of $x_0 = 0.8$, $tol = 0.0001$, and $maxiter = 15$. The function returned a value of $\alpha = 1.0000$ and $C = 0.5000$. This is in accordance to the expected order of convergence for $f_1(1) = 0$. Since $f_1'(1) = 0$, due to $x = 1$ being a multiple root, the convergence is linear and therefore should have an order of 1 [1]. Furthermore, figure 1(b) was created using $f_2(x)$ with initial guess of $x_0 = 2.6$, $tol = 0.001$, and $maxiter = 10$. The function returned a value of $\alpha = 1.9034$ which ≈ 2 and $C = 0.5196$. Once again this is in accordance to the expected order of convergence for $f_2(3) = 0$. Since $f_2'(3) \neq 0$, it is a simple root and therefore obeys quadratic convergence with an order of convergence of 2. This shows that *mynewtonErrorPlot* can accurately plot the iteration errors as well as analyze the order of convergence of simple and multiple roots.

Task 2

The objective of this task if to construct an automated algorithm that uses the Bisection method in concordance with Newton's method to approximate the root of any function within any given interval



(a) Error plot of $f_1(x) = (x-1)^2$



(b) Error plot of $f_2(x) = (x-3)(x+2)(x-1)$

Figure 1: Scatter plots of errors for two test functions as well as their lines of best fit as computed by mynewtonErrorPlot.m.

$[a, b]$. To begin, consider the function

$$f(x) = x^3 - 2.0x - 2.71828182845905x^2 + 5.43656365691810 \tag{7}$$

within the given interval $[-2, 4]$. To motivate the progression of this algorithm, equation (7) was plotted using Matlab and points near the roots were located. This can be seen in figure 2. Using a natural approach to root finding, the three x-values found in figure 2 were fed to the *mynewton* algorithm with $tol = 0.0001$ and $maxiter = 20$. The results for the 3 roots were found as follows:

$$\begin{aligned} r_1 &= -1.41421\ , \quad flag = 3 \\ r_2 &= 1.41421\ , \quad flag = 3 \\ r_3 &= 2.71828\ , \quad flag = 2 \end{aligned} \tag{8}$$

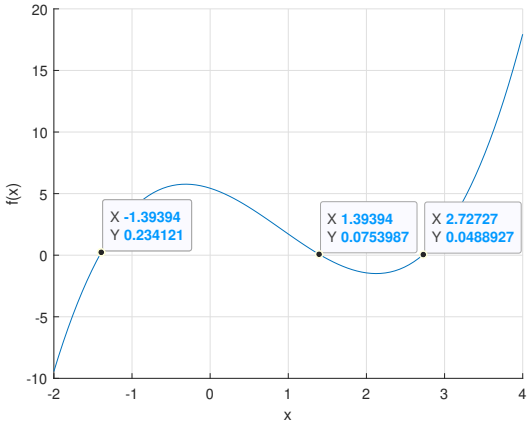


Figure 2: Plot of $f(x) = x^3 - 2.0x - 2.71828182845905x^2 + 5.43656365691810$ with the purpose of finding points near the roots

To begin the process of automating an algorithm to solve for a root within a given interval, function *mynewton2* was constructed taking an interval $[a, b]$ as input in place of an initial guess x_0, all other code remaining then same as *mynewton*. Furthermore, the following statement was inserted before the *for* loop

```
x0 = (a+b)/2;
```

as well as the following *if* statement at the start of the loop before the incrementation of the iteration counter. This code ensures that the guesses remain inside the input interval and will break the loop before incrementing the number of iterations recorded:

```
if (x0<a) || (x0>b)
    flag = -2;
    break
end
```

The routine was checked on equation (7). The routine found the root as $r = 1.41421356237309$. This was found with an accuracy of 4 decimal places since the tolerance was set to $tol = 0.0001$.

Continuing, one can divide the interval $[-2, 4]$ into subintervals $[-2, 1]$ and $[1, 4]$. Based in the *Intermediate Value Theorem (IVT)*, one can conclude that there will be at least one root in the sub interval $[-2, 1]$ since $f(-2)f(1) < 0$. However, applying IVT to the sub interval $[1, 4]$, $f(1)f(4) > 0$, one can not conclude that there exists a root within that interval. Dividing that interval further into $[1, 2.5]$ and $[2.5, 4]$, one may use IVT in the same manner to prove the existence of roots within both of those intervals. This motivates the next step in the generalization of the algorithm. A function called *myIntervalSolve* will be created which implements *mynewton2* as a subroutine. *myIntervalSolve* will divide the input interval repeatedly into a number of sub intervals until suitable termination criteria are achieved.

To begin the function *myIntervalSolve*, the header is coded as

```
function [subIntervals, rootList] = myIntervalSolve(f,fx,a,b,interLimit,tol,maxiter)
```

The function outputs an array of all the sub intervals it has divided the initial interval $[a, b]$ into as well as an array containing all of the approximated roots. The input of the function is identical to that of *mynewton2* save the parameter *interLimit*. This new parameter shall define the suitable termination criterion. The user may input any floating point or integer number so that when the sub intervals' ranges fall below this limit, the bisection of the intervals will cease. This is implemented by an if statement which terminates the function should $b - a < interLimit$. The interval is then divided by 2 with variable $c = (b + a)/2$ as the delimiter. Next, two recursive calls are made to the function such that it processes the new sub intervals $[a, c]$ and $[c, b]$, respectively and then add them to the list of sub intervals:

```
[subTemp1]=myIntervalSolve(f,fx,a,c,interLimit,tol,maxiter);
subIntervals=[subIntervals, subTemp1];
[subTemp2]=myIntervalSolve(f,fx,c,b,interLimit,tol,maxiter);
subIntervals=[subIntervals, subTemp2];
```

Finally, the sub interval array is traversed using a *for* loop in increments of 2 and processed each interval within it using the *mynewton2* subroutine. A challenge arose as the function would return a *rootList* containing many approximations of the same root due to the recursive nature of *myIntervalSolve*. To relieve this, the *rootList* was declared as an empty array in the line preceding the *for* loop.

```
rootList=[];
for i=1:2:length(subIntervals)
    [newtOut,flag] = mynewton2(f,fx,subIntervals(i),subIntervals(i+1),tol,maxiter);
    if (flag~=-2) && (subIntervals(i)~=newtOut)
        rootList = [rootList, newtOut];
    end
end
```

This allowed the function to re-declare the array as empty before analyzing it during every single iteration. The last iteration of the function contains a complete list of all the sub intervals with the specified limit set by the user. Therefore, the final analyzation of the sub intervals using *mynewton2* was recorded on an apriori tabula rasa so that the root list does not contain roots obtained from previous iterations. For the complete code refer to listing 3 in the code appendix. The function, when called as

```
[subIntervals, rootList]=myIntervalSolve(f,fx,-2,4,0.5,0.0001,20);
```

where *f* is defined as equation (7), and *fx* its first derivative, correctly returns a root list [-1.4142,1.4142,2.7183]. Therefore, the routine functions as needed.

Summary of Results

The purpose of Task 1 was to build a function, *mynewton*, that uses Newton's Method, a special case of Fixed-Point-Iteration, to approximate the roots of a function. The parameters include a function, its first derivative, and an initial guess of a root. Additionally, the error tolerance and maximum number of iterations must be input to signal the algorithm when to terminate. The function then outputs a vector containing the approximated root at every iteration (with the last being the most accurate root) and a flag denoting the number of iterations required to reach the tolerance level (or -1 if the maximum number of iterations was reached). *mynewton* was tested with numerous input functions, and proved to accurately approximate roots, given a reasonable initial guess.

To continue, the function *mynewtonErrorPlot* was constructed to perform error analysis. This algorithm functioned similarly to *mynewton*, but recorded each iteration error into a matrix. Two subarrays were constructed containing all consecutive errors, e_k and e_{k+1}. These were transformed into a logarithmic linear equation and used to plot the iteration errors on a log-log plot. Finally, the Least Squares Method was implemented to determine the order of convergence of the input function and the y-intercept of the plot. These two values were then used to produce a line of best fit for the error plots. *mynewtonErrorPlot* was tested using the same functions as *mynewton* and proved to accurately plot iteration error with a line of best fit and approximate the order of convergence of both simple and multiple roots.

The objective of Task 2 was to construct an automated algorithm that would combine the principles of Bisection method and Newton's method to approximate the roots of a function within a given interval. Firstly, the function *mynewton2* was built off of *mynewton*, by taking an interval instead of an initial point as input. This function would terminate if the algorithm diverged outside the initial interval. *mynewton2* was tested on the given equation and was found to accurately approximate the root within the specified error tolerance.

Next, the function *myIntervalSolve* was constructed to further automate the process with the capability of approximating multiple roots within the given interval. It was designed recursively, allowing it to repeatedly generate sub-intervals until the specified lower limit on interval size was reached. *mynewton2* was called as a subroutine to identify roots within these sub-intervals, with the list of roots declared empty before each *mynewton2* call to prevent roots from being repeated. This function was tested on the given equation and proved to return a correct list of approximated roots.

An interesting point for future research would be a deeper investigation into the recursive function *myIntervalSolve*. When tested on multiple roots, it would always return two roots. For the case of $f(x) = (x-1)^2$, for example, the returned roots were $[0.999, 1.001]$. While this of course is inaccurate as there is only one root for the function, it does in fact have a multiplicity of two. This could be caused by the recursive functionality, or by merely a consequence of dividing the sub intervals in the specific way it did. While this function still returns accurate roots, this interesting phenomenon deserves further investigation.

References

[1] Sauer, T. (2019). Numerical analysis (3rd ed., p. 42, 58). Hoboken, NJ: Pearson.

[2] Nicholson, W. K. (2018). Linear Algebra with Applications. Minneapolis, MN: Open Textbook Library.

Code Appendix

```
flag = -1;
iter = 0;
x=[];
for i=1:maxiter
    iter = iter + 1;
    deltax = -f(x0)./fx(x0);
    xnew = x0 + deltax;
    if abs(x0-xnew)<tol
        flag = iter;
        x0=xnew;
        break
    end
    x0=xnew;
    x=[x, xnew];
end
end
```

Listing 1: Function mynewton.m approximates the root of any once differentiable function

```
function [errorMatrix, a, C]=mynewtonErrorPlot(f,fx,x0,tol,maxiter)
iter = 0;
errorMatrix = [];
ek = [];
ek1 = [];

% Build error matrix
for i=1:maxiter
    iter = iter + 1;
    deltax = -f(x0)./fx(x0);
    xnew = x0 + deltax;
    % errorMatrix is a 1 row matrix (AKA array)
    errorMatrix = [errorMatrix, abs(x0-xnew)];
    if abs(x0-xnew)<tol
        break
    end
    x0=xnew;
end

% Build sub error matrices for e_k and e_k+1
ek = zeros(length(errorMatrix)-1,1);
ek1 = zeros(length(errorMatrix)-1,1);
% For each value in ek, the value at the same index in ek1 will be the
% subsequent value of ek inside the errorMatrix
for i=1:length(errorMatrix)-1
    ek(i,1) = errorMatrix(i);
end
for i=2:length(errorMatrix)
    ek1(i-1,1) = errorMatrix(i);
end

% Solve for alpha and C using least-squares approximation method
y = log(abs(ek1));
% M is matrix of functions with form y = mx + b where:
%   y = ln(ek1)
%   m = a
%   x = ln(ek)
%   b = ln(C)
M = zeros(length(ek),2);
% Least Squares function: r0*x+r1
```

```
%   r0 = alpha
%   r1 = ln(C)
leastSquaresFunc = @(x)x;
for i=1:length(ek)
    M(i,1) = leastSquaresFunc(log(abs(ek(i))));
    M(i,2) = 1;
end
% By Least Squares Method
r = (inv(transpose(M)*M)*(transpose(M)*y));

a = r(1);
C = exp(r(2));

% Plotting on loglog plot
loglog(r(1).*log(abs(ek))+r(2),log(abs(ek1)));
hold on;
scatter(log(abs(ek)),log(abs(ek1)));
grid on;
xlabel('$log_e|e_{k}|$', 'interpreter', 'latex')
ylabel('$log_e|e_{k+1}|$', 'interpreter', 'latex')
```

Listing 2: Function mynewtonErrorPlot.m plots the iterations errors on a loglog plot, find a line of best fit using least squares method, and returns the value of constant C and the order of convergence a

```
function [subIntervals, rootList] = myIntervalSolve(f,fx,a,b,interLimit,tol,maxiter)
subIntervals=[];

% Check if interval is smaller than limit
if (b-a)<interLimit
    subIntervals=[a,b];
    return
end

% Divide interval into 2
c=(b+a)/2;

% Call function recursively to build subInterval array
[subTemp1]=myIntervalSolve(f,fx,a,c,interLimit,tol,maxiter);
subIntervals=[subIntervals, subTemp1];
[subTemp2]=myIntervalSolve(f,fx,c,b,interLimit,tol,maxiter);
subIntervals=[subIntervals, subTemp2];

% Declare rootList output as empty before for loop to account for recursive
% calls evaluating in every instant
rootList=[];
for i=1:2:length(subIntervals)
    [newtOut,flag] = mynewton2(f,fx,subIntervals(i),subIntervals(i+1),tol,maxiter);
    % Make sure the root falls within the interval and account for case
    % where one interval ends on a root and the next one starts on that
    % same root
    if (flag~=-2) && (subIntervals(i)~=newtOut)
        rootList = [rootList, newtOut];
    end
end
```

Listing 3: Function myIntervalSolve solves for the roots of any given function within a specified interval

Plot Appendix

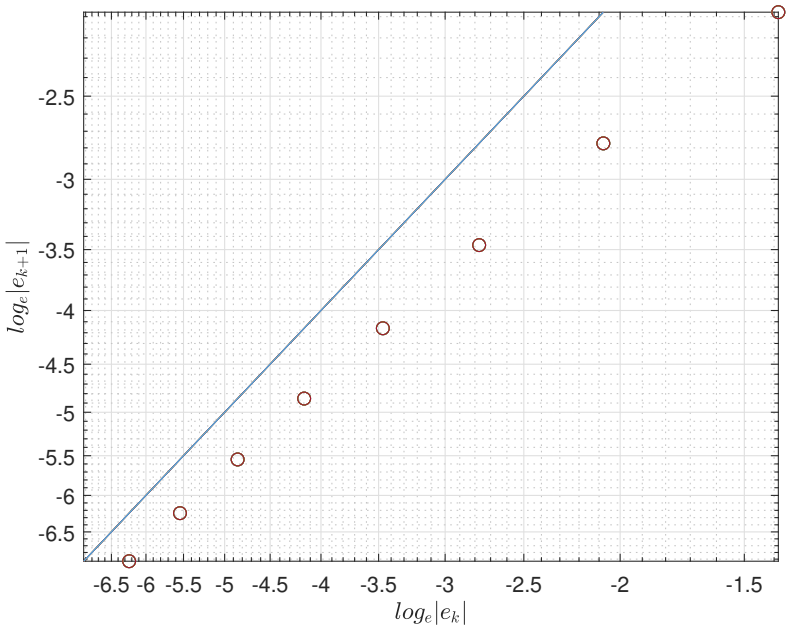


Figure 1(a): Error plot of $f_1(x) = (x-1)^2$ with the line of best fit as computed by *mynewtonErrorPlot*

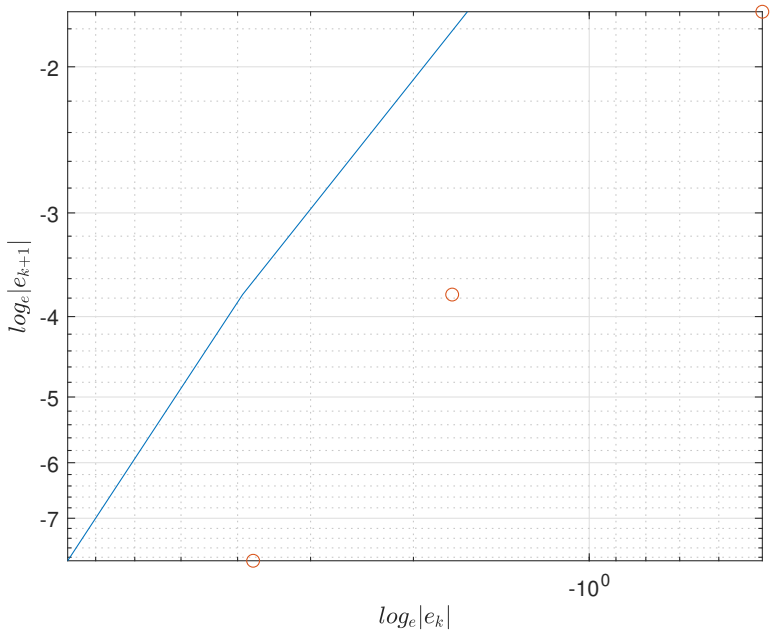


Figure 1(b): Error plot of $f_2(x) = (x-3)(x+2)(x-1)$ with the line of best fit as computed by *mynewtonErrorPlot*

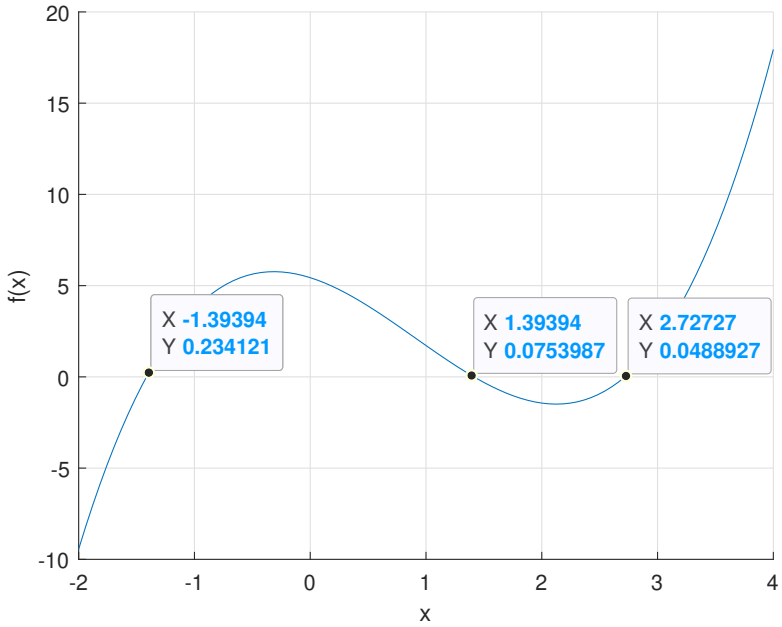


Figure 2: Plot of $f(x) = x^3 - 2.0x - 2.71828182845905x^2 + 5.43656365691810$ with points near the roots identified