# Root finding with the Newton-Raphson Method (Project 1: Numerical Methods for Engineers)

McQuade, Michael and Vargas, Vincent and McMahan, Josh and Luna, José
{A01677104, A01335644, A01677115, A01337885} @itesm.mx
Instituto Tecnológico y de Estudios Superiores de Monterrey, Campus Ciudad de México

***Abstract*- Studying an engineering major implies the usage of mathematical tools along with creativity and ingenuity. This is the reason why this project aims to determine the implications and utility of the Newton-Raphson method as well as to know its benefits and failings. At the moment of calculating a value within a descriptive model, such as the motion of a planet around its orbit or a simple root finding problem, we can use this method to compute the solution of any of these problems as it will be seen through this report. The results that we have obtained so far were the successful modeling of the orbits of two bodies following the Kepler's planetary motion law and the root finding for two polynomial functions, the graphs obtained for the orbits where sketched point by point, resulting in two perfect ellipses, the root values found in the polynomial problems by Newton-Raphson where correct compared to the theoretical values found through algebra. Finally, some particularities found on the method were discussed; we found that the Newton-Raphson Method can be used to find a root within a very small number of iterations. However, we found that it was difficult, if not impossible with this method alone, to control the method in order to find a specific root, or a root in a specific direction. This conclusion coincides with previous research discussing basins of attraction and attractors.**

**Keywords - Newton-Raphson method, descriptive model, Kepler's planetary motion Law, root, polynomial function.**

## I. INTRODUCTION

### *A. Problem statement*

The main problem being discussed in this report is the implications of using the Newton-Raphson method to approximate the value of the roots of a given function. The Newton-Raphson method is known as the most used method of its kind; nevertheless, there are several reasons that make this method useless. For example, this method often requires using a large number of iterations to accurately converge to a root. These reasons are due to be explored on this report, as well as searching alternatives for correcting the breaches of this method.

In order to investigate this hypothesis, we have sought to solve three questions within this report:

1. Firstly, calculate the position of $x$ planet at time $t$ and then discover the elliptic path that said planet makes around the sun.

2. Secondly, find the roots of the function:

$$f(x) = x^3 - \frac{31}{10}x^2 + \frac{1}{10}x + \frac{21}{5}$$

Use the starting guess x-values $x = 0.0161$ , $x = 2.051$, and $x = 0.5$ in order to find roots at the left and right of each initial starting value.

3. Finally, find all of the roots of a polynomial function between the interval (-3,4).:

$$f(x) = x^4 + 10x^3 + 27x^2 - 2x - 40$$

### *B. Theoretical framework*

The Newton-Raphson method is a mathematical model that was used for the first time by sir Isaac Newton in his book "The Method of Fluxions and infinite series", in 1671; however it wasn't published until 1736. Another mathematician, Joseph Raphson, used the same exact method in his book "Aequationum Universalis"; despite Newtons, Raphsons book was first published in 1609. It was stated as a method to find the roots of a polynomial with an iterative process, requiring exclusively the use of algebra. This method is most commonly used for its effectiveness and short time to compute all the solutions inside a function.

The Newton-Raphson method states that, given an initial value (a first guess), the position of that point, set by a pair of coordinates (x, y), can determine the roots of the whole function in which that point is. This is done by approximating values of the roots using an initial point, the function that is describing the model and its derivative. With these parameters, the new value in which the method starts a new iteration is computed.

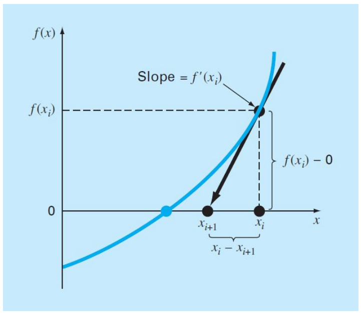


Figure 1. Graphical representation for Newton-Raphson

Graphically, the Newton-Raphson method focuses on using several tangent lines per iteration, searching for a new value every time the error frame proposed is not satisfied. The equation to find a new value for each approximation is given by Eq.(2), which is calculated from the geometric meaning of the derivative in the Fundamental Theorem of Calculus (Eq.(1)). Essentially, evaluating a point in the original function and in its derivative leads us to two points which are used to calculate the slope designated to cross through the horizontal axis of the graph; this process is repeated multiple times until the slope crosses through or is approximately at a root, where the error of the method is minimal.

$$f'(x_i) = \frac{f(x_i) - 0}{x_i - x_{i+1}} \qquad (1)$$

Which yields

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \qquad (2)$$

### C. Possible solution

By applying the Newton-Raphson method and adapting it to each individual problem, we can obtain results within our parameters. Within the first problem, we expect that each point found will form the path of a planet's movement, which will result in an ellipse when plotted. The second problem we hypothesize that finding a root in a specific direction will only be possible if contrive the initial value of $x$. We expect that if we can manipulate the value of $x$ based off of the current sign of the function multiplied by the functions derivative, then we can obtain a new $x$ value that will assist in finding a root in the other direction. Another possible solution, is to factor out the root we find upon each iteration, and then search for additional roots. Finally, we believe that if we use a quadratic approximation instead of a linear approximation, then we can find two new x values instead of a single new value. With this method of quadratic approximation approximation we may be able to find two roots at the same time.

For the third and final problem, we believe that the method and the number of iterations will strongly depend on the initial values chosen as well as the calculated x-values within the process, which will result in a process that is attached to the parameters of a function given. In our trials, we expect to justify this.

### D. Importance of the research

In the case of an engineering major or co-related knowledge areas, the correct usage of mathematical tools is a fundamental task; this tools include numericals methods such as Newton-Raphson's. We decided to start this research in order to know how to correct apply the method to different situations, as it will be seen later on the report with Kepler's planetary motion law and predicting the root values for a function. The reason behind numerical methods is to provide the user different ways to approach a problem and to solve it, even if the difficulty of this make obsolete other mathematical tools, like calculus.

## II. METHODOLOGY

In order to solve the problems presented, three programs which implemented the Newton-Raphson method in various ways were created.

### A. Problem approach

*1. First problem*

The following formulae are important in order to solve the first problem presented:

$$x = acos(E - e)$$

$$y = a\sqrt{1 - e^2}sin\ E$$

Within these formulae, $e$ represents the eccentricity of the ellipse which can be defined as the relation $e = \frac{c}{a}$. In this relation, $c$ is the distance between the center and the focus, which is given by a Pythagorean Triangle inside the parameters of the ellipse (Figure 2.), and $a$ is the distance between the center and the vertex. Finally, $E$ is the eccentric anomaly.

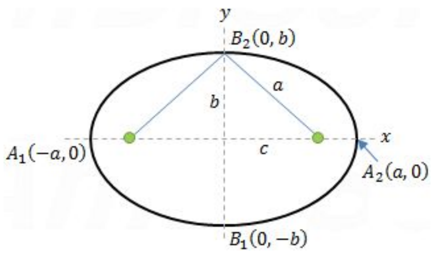


Figure 2. Schematic of the parameters of an ellipse

Within the implementation, two values of $a$ and $b$ were used. The values set during this implementation were $a = 5 * 10^{11}$, $a = 6 * 10^{11}$ and $b = 4 * 10^{11}$, $b = 3 * 10^{11}$. Also, a relative error frame of $\varepsilon = 1 \times 10^{-6}$ and a limit of up to 20 iterations was established. The values for $M$ and $e$ were fixed inside the intervals $(0, 2\pi)$ and $(0, 1)$ respectively. For the interval of M a step size of .1 was chosen. A smaller step size could have been chosen, but diminishing returns of precision and increased processing time would follow. It was determined that an accurate ellipse was able to be produced at this step size.

In order to draw the orbit that the planet follows in the given problem we had to solve the main incognita, which was to compute the value of $E$ in the Kepler's nonlinear equation for planetary motion (Eq. 4)

$$f(E) = M - E + e\,sin(E) = 0 \quad (4)$$

For achieving this objective we set $E$ as our $x_i$ value on the Newton-Raphson method. Since $a$ and $b$ are fixed, $e$ can be calculated easily following a simple Pythagorean Triangle (Eq. 5.). The value for $M$ shall be set differently for each point that conforms our ellipse, which implies that, for each calculation of the points within our ellipse, a new Newton-Raphson method should be done, varying the $M$ value only.

$$e = \frac{c}{a} = \frac{\sqrt{a^2 - b^2}}{a} \quad (5)$$

*2. Second problem*

In the second problem a function and three starting values were used as a guess. To start, it was necessary to derive the function and store both within a variable as symbolic matlab expressions. For derivation, we manually derived the function, but later, when within our looping structure, make use of the Matlab `diff()` method which provides the same result.

While no arbitrary limit of iterations was imposed for this project, a hard limit of 100 iterations was chosen. It was found that if the program began to use many iterations then it was likely it would not find the answer and there was a mistake within the programming.

On the first iteration of the looping structure, the function and its derivative are evaluated at the starting guess value for $x$ and stored. Afterwards, it was important to ensure that the result of the derivative was not zero, and if so, change the value. Next, the value for $X_{i+1}$ was obtained using the previously derived formula (2). The error is found by subtracting the starting value of $x$ from the newly obtained $X_{i+1}$ and dividing the result by $X_{i+1}$. If this resultant error is within a desired tolerance, then the root was considered found and added to the solution set, if it was not within tolerance then the process is repeated with $X = X_{i+1}$. The original $x$ value was stored in order to determine if the root found was to the left or right of the original value. When the program found a root, a comparison was done and a boolean flag was set based on the direction of the newly discovered root. Additionally, another boolean flag was set to signal the program that it a root was just found. This allowed the loop to have an additional control structure at the beginning that checked for these conditions. Upon entering this structure, the program would factor out the most recent root that it had found from the existing function, and store a new function and new derivative function to continue the Newton-Raphson method with. Afterwards, it would reset the flags to off. Finally, to go through all the values with just a single run of the program, the entire loop structure was wrapped in an additional loop which would change the original value of $x$ on each of its iterations. Printing the results on each iteration.

*3. Third problem*

In the final problem, the roots of the polynomial function $f(x) = x^4 - 10x^3 + 27x^2 - 2x - 40$ over the interval (-3,4) had to be found using Newton-Raphson; the Figure 3. shows its graph. In order to do this, it is necessary to start the iterations at some point of the graph where we could estimate the behaviour of the method. Using this strategy will make our program follow a pre established route.

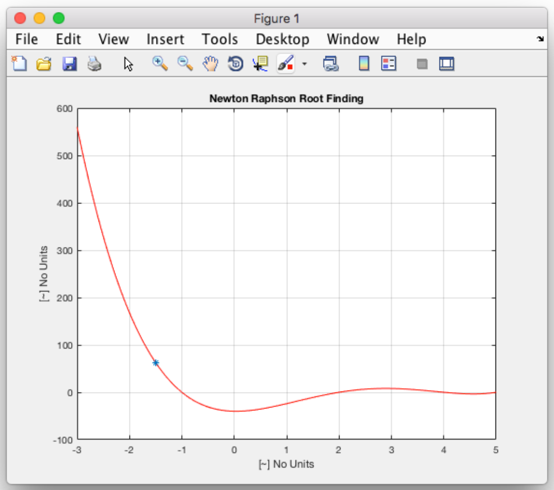


Figure 3. Graph of the original polynomial in the interval (-3, 4).

First of all, the initial point of the function need to be proposed inside the limits given by the problem ((-3, 4)). Then, using the graphical model of the polynomial we can infer the behaviour of our method by looking at the sign of the slope that crosses tangent to our point. This implementation strongly depends on the form of the graph and the location of our initial point. This phenomenon can be seen in Figure 3. As the curve descends, the derivative at that point will be negative signed until an inflexion point is reached; in the other hand, as the curve rises, the derivative will be positive signed until it reaches a new inflexion point.

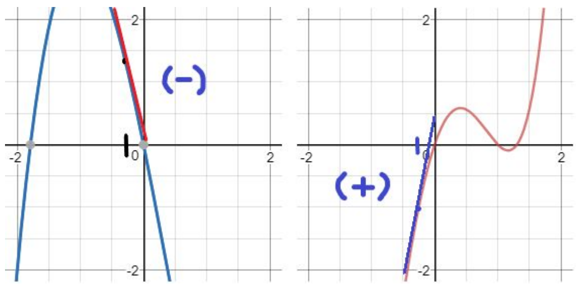


Figure 3. Sign change of the derivative of a polynomial function

Thus, in order to find all the roots the program must have at maximum fifty iterations to search all the possible roots within the interval given and, in our case, in only one direction. That is reason why we have decided to set our initial point at $x = -1.5$ , for the number of iterations shall be less than if we had started at a more negative value. Also, we start at the limit on the left of the graph so it is possible to implement an automatic method at the moment of coding the program on matlab. This method is given by the following:

Taking into account that the derivative at our initial point is negative, the first root that the method will find is $x = -1$. The next root should be found when the sign of the derivative at a new initial point changes, as saw in Figure 3. Thus, until an inflection point is found, the new Newton-Raphson process for the next root search should not start. The change in the sign of the slope can be verified by using the Eq. 6. Furthermore, since the Newton-Raphson method is very sensitive to the values used on it and the form of the graph, we shall take into account the behaviour of the method at the moment of the calculation of the new starting values $x_{i+1}$ throughout the whole process and add a "break" to prevent the program from non-desirable outputs.

### *B. Matlab Implementation*

Within the second problem there were a few Matlab functions used within the implementation that will be explained below.

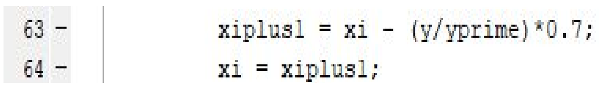
```
if (numberOfSolutions > 0 && flag == 0)
    flag = 1;
    syms x;
    currentSolution = solutions(end);
    funF = simplify(funF/(x - currentSolution));
    funDF = diff(funF);
    x = Xoriginal;
end
```

In the above code excerpt, the implementation for obtaining a new function without the previously found root can be seen. This implementation makes use of symbolic expressions and the matlab simplify function. On paper, the simplification is merely dividing the root in the form of $x - root = 0$ from the polynomial. This provides a new function which we store back into the symbolic expressions `funF` and `funDF`. These expressions are later evaluated by storing a new value into $x$ and using the Matlab `subs()` method to substitute the value of x into the symbolic expression and retrieve the value of the function at that point. The flag in this control structure is used in order to ensure that the factorization process here only happens after we find a root.

Some Matlab-specific explanations are offered below for the third problem. In order to make this program we needed to give the initial parameters of the problem with which the code would start with. This way, the program would be more easy and understandable for the user as well as for the language. In this case, we firstly declared the variables that contained the polynomial and its derivative, as well as the interval of our target function and its plot (Figure 4.). Also, all the fixed values on the problem were included as variables in order to start the method.

First, the initial guess was established at $x_i = -1.5$. The original function and its derivative were set as intrinsic Matlab functions in order to evaluate the $x_i$ value inside them. Then inside a secondary while loop, the Newton-Raphson process was constructed and plotted, so that for a new $x_i$ value formulated, the process would start again until all the roots of the polynomial were found.

Since Newton-Raphson method is unstable in terms of root finding, it is necessary to consider a “break”. This break is given by multiplying the method by a constant; in the case of the third problem, we pose a constant with a value of 0.7 as it can be seen in Figure 4.



```
        xiplus1 = xi - (y/yprime)*0.7;
        xi = xiplus1;
```

Figure 4. Problem 3 code fragment: Newton-Raphson break constant.

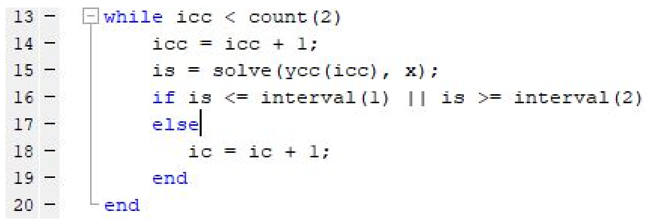
The reason behind the value relies on the reduction of the number of iterations in the process, but also, having a number that is small enough to reduce the jumping amount of the method. The difference between using the method without the break and with it is given in the following example:

$$x_{i+1} = -1.5 - \frac{f(-1.5)}{f'(-1.5)} = -1.5 - \frac{62.5625}{-164} = -1.1185 \quad (6.a)$$

$$x_{i+1} = -1.5 - \frac{f(-1.5)}{f'(-1.5)} * (0.7) = -1.5 - \frac{62.5625}{-164} * (0.7) = -1.2329 \quad (6.b)$$

$$\Delta x = |-1.1185 - (-1.2329)| = 0.1144$$

As it can be seen, the usage of a break previse the Newton-Raphson method from giving big steps along the entire iterative process, which prevents the method from jumping from one root into another, skipping important values within an interval.



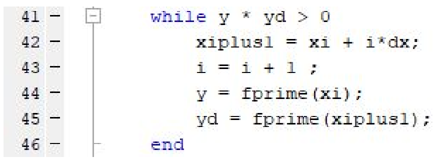
```
while icc < count(2)
    icc = icc + 1;
    is = solve(ycc(icc), x);
    if is <= interval(1) || is >= interval(2)
    else
       ic = ic + 1;
    end
end
```

figure 5. problem 3 fragment: interval function detector

The purpose of this implementation is to remain between the interval and with any change in the interval the program will stay working only in the given interval but is useful if we want to experiment with that interval and find more roots to the left or to the right .

‘is’ is the function that prints the root and only if the root is between the interval it will printed if not it will be ignored so if we want to get more roots in a polynomial function the interval needs to be modified but it will find every root until the end of the interval.

The last consideration that has to be taken into account is the sign of the slope. To verify this change in the sign it is necessary to detect it. With a simple division or multiplication given of the values $f'(x_i)$ and $f'(x_{i+1})$, we can obtain a value, which could be less or greater than 0; this method only cares if the value is positive or negative since the program is looking only for the sign of that computed point. The best way to implement this test is a simple while loop within an if inside, as it is shown in Figure 5.

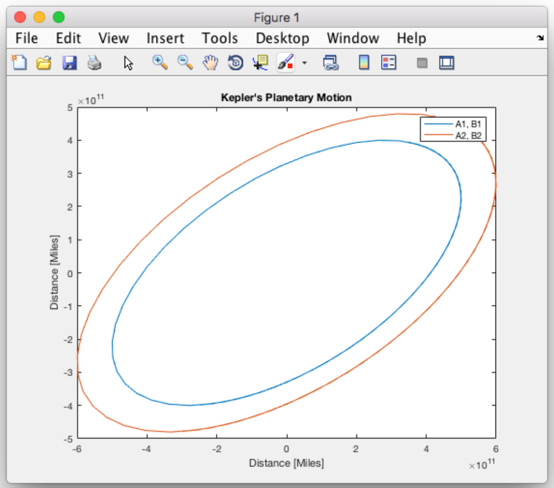


```
while y * yd > 0
    xiplus1 = xi + i*dx;
    i = i + 1 ;
    y = fprime(xi);
    yd = fprime(xiplus1);
end
```

Figure 5. Method used to detect the change on the sign of the slope at a certain point of the graph.

Until the slope changes its sign, a new value of $x_i$ is searched until the method overpasses the inflexion point. When the condition is satisfied, the $x_{i+1}$ value calculated inside the loop is assigned to $x_i$, and a new Newton-Raphson method is started in order to find a new root.

## III. RESULTS



Graph 1. *Problem 1, Path of a planet's movement*

Above is the graph obtained from the first problem. This graph of two perfect ellipses represents the entire path of a planet's movement with the chosen values of $a_1 = 5 * 10^{11}$, $b_1 = 4 * 10^{11}$ and $a_2 = 6 * 10^{11}$, $b_2 = 3 * 10^{11}$. The outer ellipse

The results of the second problem concluded that the Newton-Raphson method can not be made to give the zero that is closest to the chosen guess value. The result is highlighted by the third guess of $x$ where $x = 0.5$. This value results in Newton-Raphson finding a root after only a single iteration.

$$f(0.5) = 0.5^3 - \frac{31}{10}(0.5)^2 + \frac{1}{10}(0.5) + \frac{21}{5} = 3.6$$

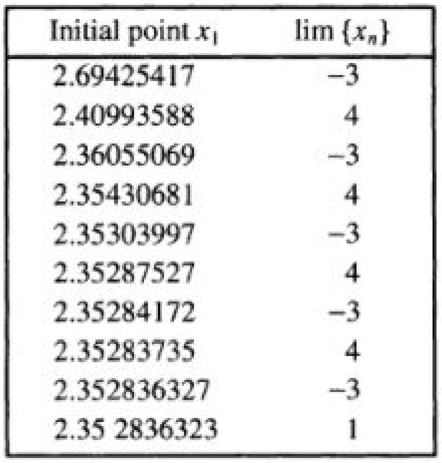
$$f'(0.5) = 3(0.5)^2 - \frac{31}{5}(0.5) + \frac{1}{10} = -2.25$$

$$x_{i+1} = x_i - \frac{f(x)}{f'(x)} = 0.5 - \frac{3.6}{-2.25} = 2.1$$

As seen above, the first root of the function was immediately found. However, the first root found is not the closest root to the original guess value. This is because 0.5 lies within a basin of attraction to the root at $x = 2.1$. Results such as this are not first of their kind. In 1997, Thomas Dence published *Cubics, Chaos and Newton's Method* in *The Mathematical Gazette*, over this very topic. He discovered that with the following function:

$$f(x) = x^3 - 2x^2 - 11x + 12$$

Considering this function, his article analyzes the following guesses (excerpted from the aforementioned journal):

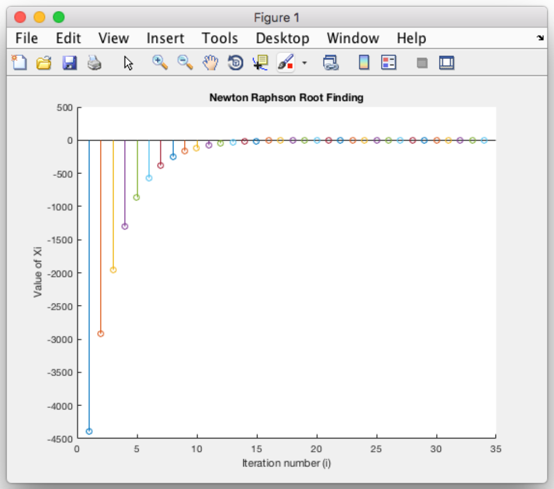


| Initial point $x_1$ | $\lim \{x_n\}$ |
|---|---|
| 2.69425417 | −3 |
| 2.40993588 | 4 |
| 2.36055069 | −3 |
| 2.35430681 | 4 |
| 2.35303997 | −3 |
| 2.35287527 | 4 |
| 2.35284172 | −3 |
| 2.35283735 | 4 |
| 2.352836327 | −3 |
| 2.35 2836323 | 1 |

Figure 6. Table showing the different roots for x with small changes in starting points.

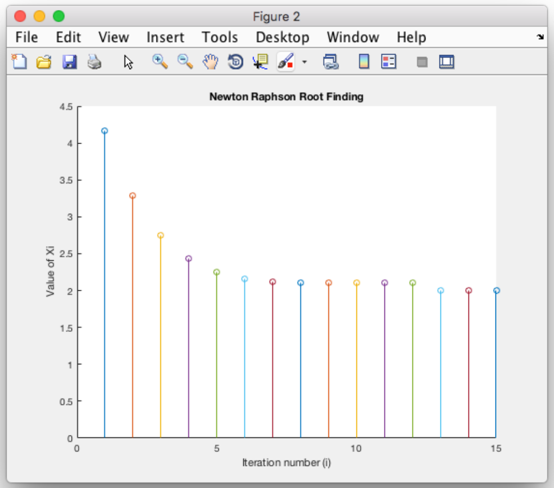
As seen above, even with very small changes in the initial guess of $x$, a different root is found. This is because these initial points lie within different basins of attractions. These basins of attractions are not easy to predict and thus in a programmatic implementation such as the one in this report it can cause some unexpected results.

The following are graphs that were obtained by plotting the iterations in relation to the values of $x_i$ during this implementation of our second problem:



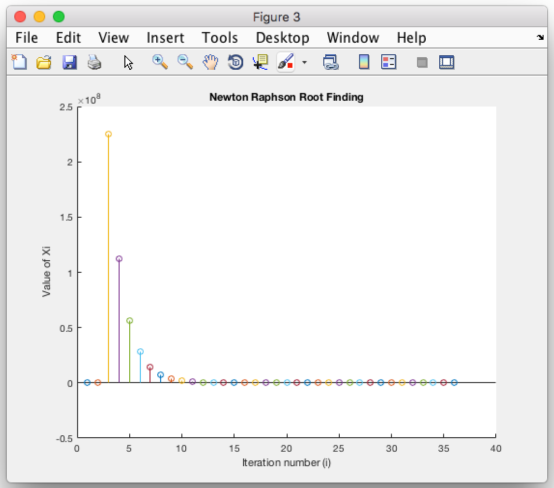
Graph 2. *Problem 2, where $x = 0.0161$*

In the above figure, it is seen that after a single iteration the initial value of $x$ at a starting value of $x = 0.0161$ becomes a very negative value, far away from any roots. This is due to the value of the the derivative function evaluated at the initial guess. The result was a very small fractional number being in the divisor of the Newton Raphson method, thus multiplying the function by that factor. However, this did not break the method, as within 10 iterations the program returned to the origin and found its first root at $x = -1$.



Graph 3. *Problem 2, where x = 2.051*

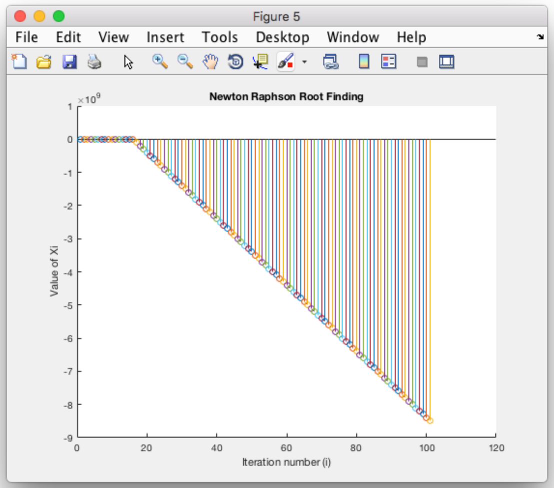
In the next figure seen above where the initial guess value of $x = 2.051$ is between two roots, it can be observed that the initial value of $x$ produces a jump to the right, to somewhere around 4. Within 5 iterations the method returns to the range between the two roots and quickly converges to the roots.



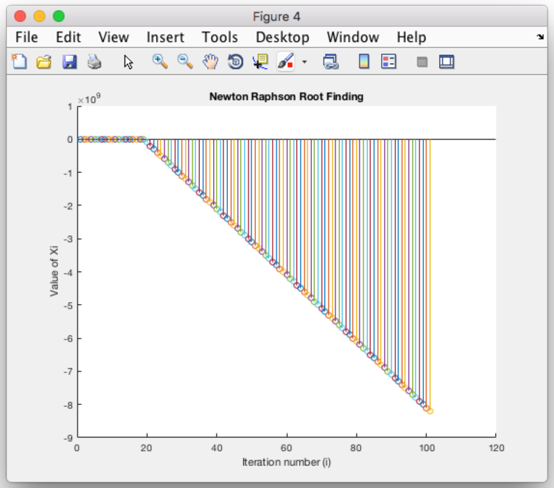
Graph 4. *Problem 2, where x = 0.5*

Above, is a very interesting graph, which comes from a starting value of $x = 0.5$. This is perhaps the most interesting case that was observed within our results, owing to the fact that after the first iteration, the program converged to the root of 2.1. As stated previously within our methodology, we did not allow division by zero thus if there was a time where we would divide by zero, we instead divided by the same number that was used for tolerance $\varepsilon = 1 * 10^{-6}$. This resulted in an astronomically large initial jump for the $x_{next}$. Even with this extreme jump causing a very inaccurate guess, it was observed that the Newton-Raphson method successfully returned to the proper range needed to find the root within less than 10 iterations. This result provided the conclusion that the initial guess is actually not that important, as the method will find the root in a short number of iterations regardless of the initial guess.

In the next two figures, something different can be observed:



Graph 5. *Problem 2, where x = -2*



Graph 6. *Problem 2, where x = 3*

The first figure is $x$ with an initial guess of $x = -2$ and the second figure is $x$ with an initial guess of $x = 3$. Both of these cases have the same issue that is due to the arbitrary limit we set upon this problem. Since it was decided to find a root to the right and a root to the left of each starting value, after a root is found in a single direction the program will keep searching for a root in the other direction. It does so with more and more distant(larger or smaller) values of $x$ until it finds a root. However, in this case, no roots were found within the maximum iteration limit because there are no roots in the other direction of our initial guesses.

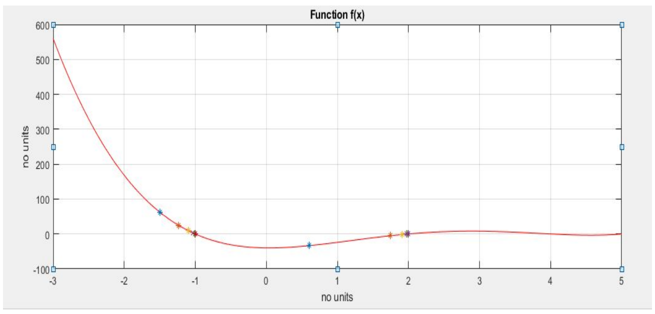
The above figure for a starting guess value of $x = 0.5$ showed another interesting case. The first iteration of this method converged to a root immediately and then made a

Results of problem 3.

| I | Xi | f(xi) | f'(xi) | Epsilon |
|---|---|---|---|---|
| 1 | -1.2330 | 24.5660 | -121.6836 | 21.658 |
| 2 | -1.0916 | 8.7881 | -101.9032 | 12.9455 |
| 3 | -1.0313 | 2.8771 | -93.9823 | 5.8537 |
| 4 | -1.0098 | 0.89253 | -91.245 | 2.122 |
| 5 | -1.0030 | 0.27073 | -90.3786 | 0.68267 |
| 6 | -1.0009 | 0.081495 | -90.1141 | 0.20949 |
| 7 | -1.0003 | 0.024474 | -90.0343 | 0.063288 |
| 8 | -1.0001 | 0.0073444 | -90.0103 | 0.019026 |
| 9 | -1 | 0.0022035 | -90.0031 | 0.0057115 |
| 10 | -1 | 0.00066108 | -90.0009 | 0.0017138 |
| 11 | -1 | 0.00019833 | -90.0003 | 0.00051417 |
| **12** | **-1** | **1.7849e-05** | **-90** | **4.6276e-05** |
| 13 | 0.6 | -33.5104 | 20.464 | ----------- |
| 14 | 1.7463 | -5.1097 | 22.1155 | 65.6411 |
| 15 | 1.908 | -1.7305 | 19.602 | 8.4765 |
| 16 | 1.9698 | -0.55176 | 18.538 | 3.1372 |
| 17 | 1.9906 | -0.16937 | 18.1681 | 1.0466 |
| 18 | 1.9972 | -0.051193 | 18.0511 | 0.32675 |
| 19 | 1.9991 | -0.015393 | 18.0154 | 0.099302 |
| 20 | 1.9997 | -0.0046212 | 18.0046 | 0.02991 |
| 21 | 1.9999 | -0.0013866 | 18.0014 | 0.0089837 |
| 22 | 2 | -0.00041602 | 18.0004 | 0.0026961 |
| 23 | 2 | -0.00012481 | 18.0001 | 0.00080891 |
| 24 | 2 | -3.7443e-05 | 18 | 0.00024268 |
| **25** | **2** | **-1.1233e-05** | **18** | **7.2805e-05** |

Table 1. Results from problem 2 with all the iterations that the program make to find the roots.

In this table we show the results we got and all the values that the program obtained searching the roots that the function has, we can see that the values highlighted in the table are a high approximation of the root for each one this values have a error that is below under the error minimum error we stipulated in the program , the program did twelve and thirteen iterations until it found the two roots.



Graph 7. *Problem 3, all iterations*

Graphic of the function with all the iterations, in the graph we can see that the iterations until it find one root are so close and have a small difference between them so they will not variate a lot as the firsts iterations do.

## IV. DISCUSSION AND CONCLUSIONS

Considering function 3, and an initial starting point of $x = 0.0161$ applying the Newton-Raphson method will result in the following:

$$x_{i+1} = 0.0161 - \frac{4.2008}{9.5763 * 10^{-4}} \cong -4386$$

This large result produces a jump to the far left, resulting in the first zero to be -1. The problem lies in that since the denominator is so small. A large jump to the left will always be observed when the derivative evaluated at the initial point produces a number that is close to zero.

Since the Newton-Raphson method always converges to the nearest root, we considered this problem in our method. In order to find a root to the opposite direction of the first root that we find, we take the difference and decide if the first root was to the left or right. Afterwards, we factor out the root that we found from the original function, creating a new function with which we can reevaluate with the Newton-Raphson method once more. This will provide us the second nearest root, but not necessarily the one to the opposite direction in cases where there were two or more roots close to the initial point, and in the initial direction. To solve this, we continue to factor out these roots and create new functions until the root that we find is in the desired direction.

This method works perfectly on paper, because we can factor the polynomial without evaluating it first. However, we were not able to reproduce these results within our Matlab applications. By first function handles of the functions in matlab, we are able to substitute in the values on each iteration. Assuming the first root that we find is $x = -1$, we can produce the following function that does not have $-1$ as a root:

$$f(x) = \frac{x^3 - \frac{31}{10}x^2 + \frac{1}{10}x + \frac{21}{5}}{x+1}$$

Which simplifies to:

$$x^2 - \frac{41x}{10} + \frac{21}{5}$$

Since, before simplification, evaluating this function at the root $x = -1$ provides a zero in the denominator, we were not able to proceed with this method without further knowledge of symbolic polynomial simplification within Matlab. In order to solve this, we converted the function handles into symbolic expressions, and then we were able to manipulate them as we would on paper, simplifying the equation before evaluating it at the current value of $x = -1$. After we have this new function, we restore $x$ to its original starting value, and reiterate through the Newton-Raphson method until we are within error tolerance again. This will produce a new root. We then check this root is in the opposite direction of the first root from the original value. In the second problem, we record the direction in which the original root is found from the original value, and then we continue factoring and applying the method until we find a root on the other side.

One danger that we discovered, is if we reach a stationary point of the function. That is to say, if the derivative of the function at our value of $x$ is equal to zero. In this case, we will encounter a problem where we are attempting to divide by zero. In fact, this actually happens in the method we have implemented when evaluating the function at $x = 0.5$, after factoring it the first time. The original equation (3), with $x - 2.1$ divided out of it results in:

$$x^2 - x - 2$$

The derivative of said function is:

$$2x - 1$$

Thus, if we use evaluate the derivative at the value of $x = 0.5$, we obtain a zero, which will then be the denominator when we apply the Newton-Raphson method. In order to resolve this issue within our application, we decided that in the case the denominator would be zero, we would instead use a very small number as the denominator, thus not changing the value by much, but excluding zero from our denominator. In this case, we reused the tolerance value, $x = 0.00000001$ that we are using for error-checking as the new value of the denominator.

Referring to problem three we found an efficient mode to resolve the problem; however, that our first program only works with the function we had, another thing we had to solve was the "jumps" the method used. We conclude that our program is designed only to run in one direction and the values used as a "break" are specifically thought to reduce the movement that the method does. Thus, it is highly probable that our program would not work with polynomials with multiple roots within a very small interval. This conclusion coincides with the problems that we uncovered when attempting to solve the second problem. Thus, the break constants should be modified in a convenient way to uncover roots in the desired direction.

## V. REFERENCES

1. T. Dence, "Cubics, Chaos and Newton's Method," *The Mathematical Gazette*, vol. 81, no. 492, p. 403, Nov. 1997.
2. S. C. Chapra and R. P. Canale, *Numerical methods for engineers*, 7th ed. Boston: McGraw-Hill Higher Education, 2010.
3. S. C. Chapra, Applied numerical methods with MATLAB for engineers and scientists. New York, NY: McGraw-Hill Education, 2018.
4. R. W. Hamming, *Numerical methods for scientists and engineers*. New York, NY: Dover Pub., 1988.
5. J. Kiusalaas, *Numerical methods in engineering with MATLAB*. Cambridge: Cambridge University Press, 2016.

## VII. Attachments

*Code for problem 1:*

```
close all; clear all; clc;


M = 0:.1:2*pi;
%%%e = 0:.1/(2*pi):1; %% /2pi in order to be same size
array .1

E = 1;

a = 5 * 10^11;
b = 4 * 10^11;
c = sqrt(a^2 - b^2);
e = c/a;

a2 = 6 * 10^11;
b2 = 3 * 10^11;

tolerance = 0.000001;
iterations = 20;

f = M - E + e * sin(E);
df = -1 + e*cos(E);

i = 0;
while ( i <= iterations )
   Enext = E - (f)/(df);
   Error = abs((Enext - E)/Enext);
   E = Enext;
   if(Error <= tolerance)
      break;
   end

   f = M - E + e * sin(E);
   df = -1 + e*cos(E);

   i = i+1;
end

x = a.*cos(E - e);
y = a.*sqrt(1-e^2).*sin(E);
```

```
x2 = a2.*cos(E - e);
y2 = a2.*sqrt(1-e^2).*sin(E);



plot(x,y), hold on;
plot(x2,y2)
legend("A1, B1", "A2, B2");
ylabel('Distance [Miles]');
xlabel('Distance [Miles]');
title("Kepler's Planetary Motion");
```

*Code for problem 2:*

```
close all; clear *; clc;

x1 = 0.0161;
x2 = 2.051;
x3 = 0.5;
x4 = -2;
x5 = 3;

figureCount = 1;

Xs = [ x1 x2 x3 x4 x5 ]
for ( K = 1:size(Xs,2))


   syms x;



   funF = x^3 - (31/10)*x^2 + (1/10)*x + (21/5);
   funDF = 3*x^2 - 2*(31/10)*x + (1/10);

   x = Xs(K);
   f = subs(funF);
   df = subs(funDF);

   Xnext = x - (f/df);
   tolerance = 0.00000001;
   iterations = 100;
   i = 0;
   Xoriginal = x;
   solutions = [];
   numberOfSolutions = 0;
   flag = 0;
   originalLeft = 0;
   originalRight = 0;
   left = 0;
   right = 0;

   originalF = funF;
   originalDF = funDF;
   title('Newton Raphson Root Finding');
   xlabel('Iteration number (i)');
   ylabel('Value of Xi');


   while ( i <= iterations )
      if ( left == 1 || right == 1)
         flag = 0;
         right = 0;
         left = 0;
      end

      if (numberOfSolutions > 0 && flag == 0)            %%
Flag controls that we only enter this block when we need to
factor out a root
          flag = 1;
          syms x;
          currentSolution = solutions(end);
          funF = simplify(funF/(x - currentSolution));
          funDF = diff(funF);
          x = Xoriginal;
      end

      f = eval(subs(funF));
      df = eval(subs(funDF));
      if ( df == 0)
         df = tolerance;
      end

      ppd = f/df;
      Xnext = x - (ppd);
      Error = abs((Xnext - x)/Xnext);
      x = Xnext;

      if ( Error <= tolerance)
         if (x < Xoriginal ) %% we went left
            if (numberOfSolutions == 0)
               originalLeft = 1;
            end
            left = 1;
            right = 0;
         else
            if ( x > Xoriginal ) %% we went right
```

```
                if (numberOfSolutions == 0)
                    originalRight = 1;
                end
                right = 1;
                left = 0;
            end
        end
        solutions = [solutions x]; %% solutions is equal to
itself + next solution (x)
        numberOfSolutions=size(solutions,2); %% get size
of solution set
        flag = 0;
        if( originalLeft == 1 && right == 1)
                %%%finish root finding
            break;
        else
            if ( originalRight == 1 && left == 1 )
                %%%finish root finding
                break;
            end

        end
    end
    i = i + 1;
    hold on
    stem(i, x)
  end
  if figureCount < size(Xs,2) %% make a figure for each
value of X
    figureCount = figureCount + 1;
    figure;
  end

  leftSolutions = [];
  rightSolutions = [];

  %%% Build right solutions and left solutions matrix
  for  J = 1 : numberOfSolutions
    if ( solutions(J) < Xoriginal)
        leftSolutions = [leftSolutions solutions(J)];
    else
        rightSolutions = [rightSolutions solutions(J)];
    end
  end

  leftSolution = max(leftSolutions);
  rightSolution = min(rightSolutions);
  if( ~isempty(leftSolution))
      disp(strcat("Zero 1 found to the left of ",
num2str(Xoriginal), " with value of ", num2str(leftSolution),
" for function ", char(originalF)));
  end
  if ( ~isempty(rightSolution))
      disp(strcat("Zero 2 found to the right of ",
num2str(Xoriginal), " with value of ",
num2str(rightSolution), " for function ", char(originalDF)));
  end

  disp('------------------------------------');
end
```

*Code for problem 3:*

```
clc, close all, clear all
f = @(a) a.^4 - 10*a.^3 + 27*a.^2 - 2*a - 40;
fprime = @(b) 4*b.^3 - 30*b.^2 + 54*b - 2;


syms x;
ycc = factor(x.^4 - 10*x.^3 + 27*x.^2 - 2*x - 40);
count = size(ycc);
ic = 0;
icc = 0;
interval = [-3, 4];

while icc < count(2)
  icc = icc + 1;
  is = solve(ycc(icc), x);
  if is <= interval(1) || is >= interval(2)
  else
    ic = ic + 1;
  end
end

x = -3 : 0.01 : 5;
EpsilonTolerance = 10E-05;
y1 = f(x);

plot(x, y1, 'r'), hold on, grid on;

Iter = 0;
EpsilonCalcu = inf;
xprev = inf;
totalIter = 0;
xiplus1 = -1.5;
y = 1;
```

```
yd = -1;
it = 0;

while it < ic
    while y * yd > 0
        xiplus1 = xi + i*dx;
        i = i + 1 ;
        y = fprime(xi);
        yd = fprime(xiplus1);
    end

    xi = xiplus1;
    while EpsilonCalcu > EpsilonTolerance
        if Iter > 0
            EpsilonCalcu = abs(1 - (xprev/xi))*100;
            %disp(EpsilonCalcu);
        end

        xprev = xi;
        y = f(xi);
        %disp(y);
        yprime = fprime(xi);
        %disp(yprime);

        plot(xi, y,'*');
        pause();

        xiplus1 = xi - (y/yprime)*0.7;
        xi = xiplus1;
        %disp(xi);
        Iter = Iter + 1;
        if EpsilonCalcu < EpsilonTolerance
            disp(xi);
        end
        totalIter = totalIter + 1;
    end

    Iter = 0;
    EpsilonCalcu = 1;
    xiplus1 = xi + 0.8;
    y = fprime(xi);
    yd = fprime(xiplus1);
    i = 1;
    dx = 0.8;
    it = it +1;

end
```

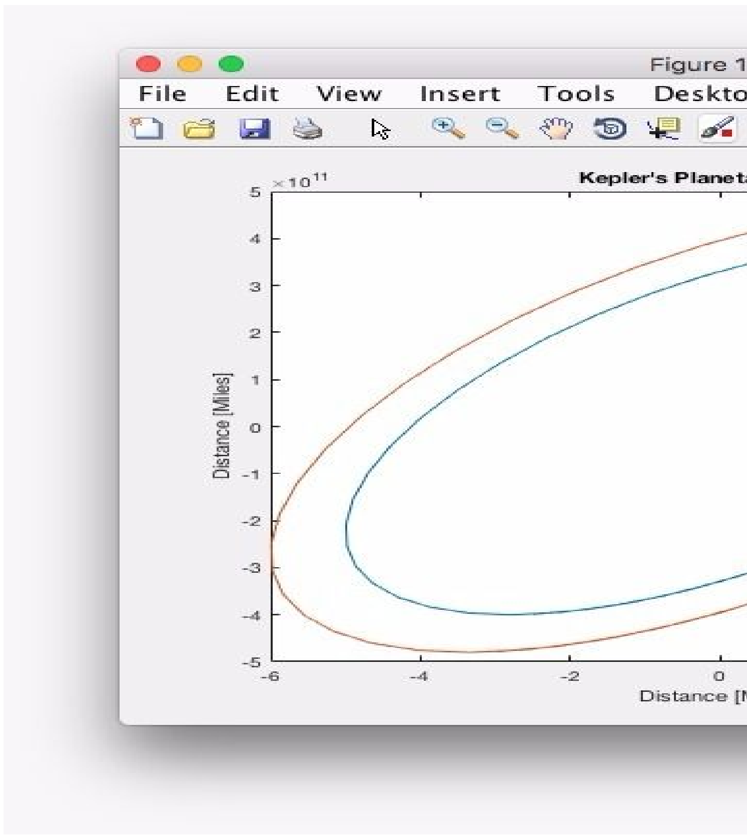


Figure 1
File Edit View Insert Tools Deskto
Kepler's Planet
×10¹¹
Distance [Miles]
Distance [M

Graph 1

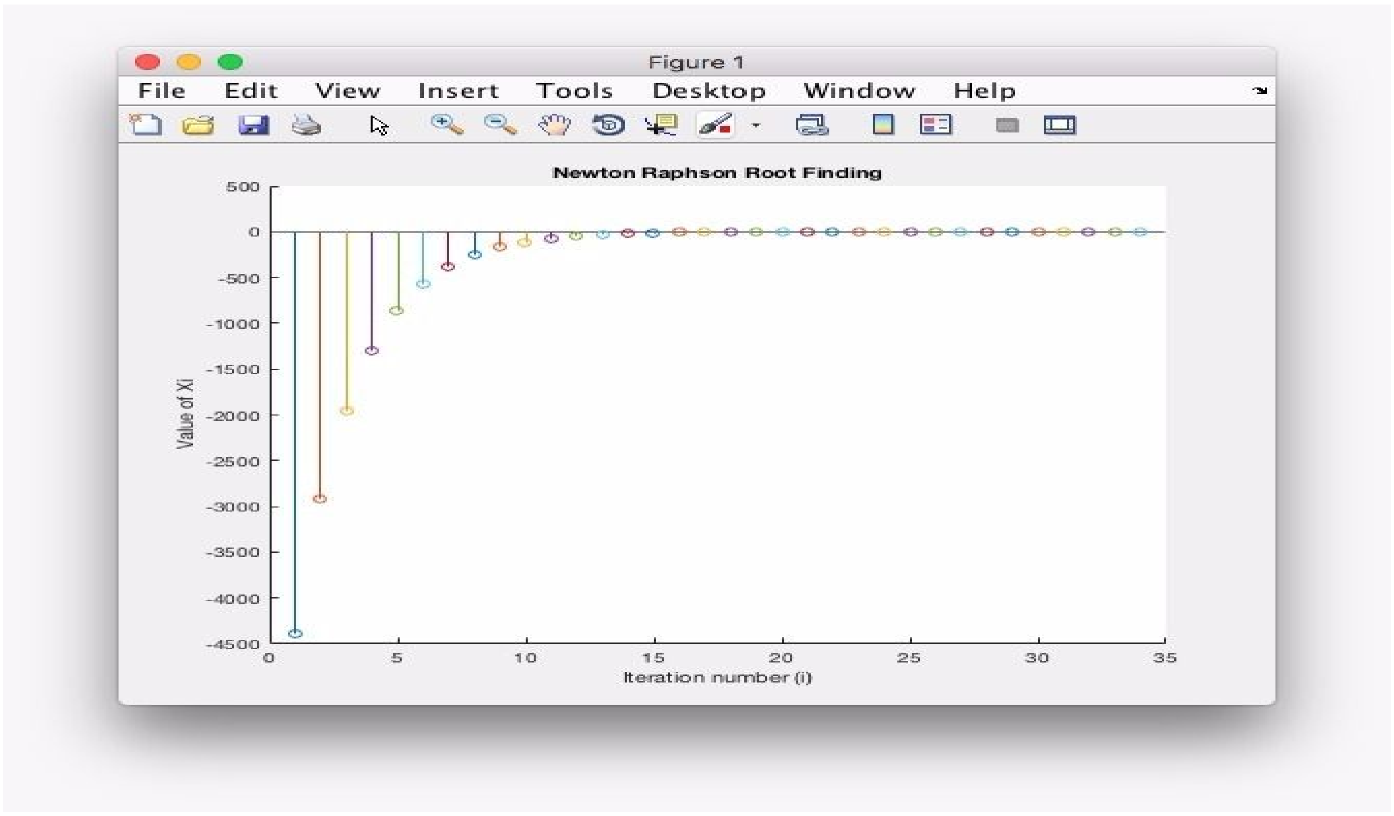
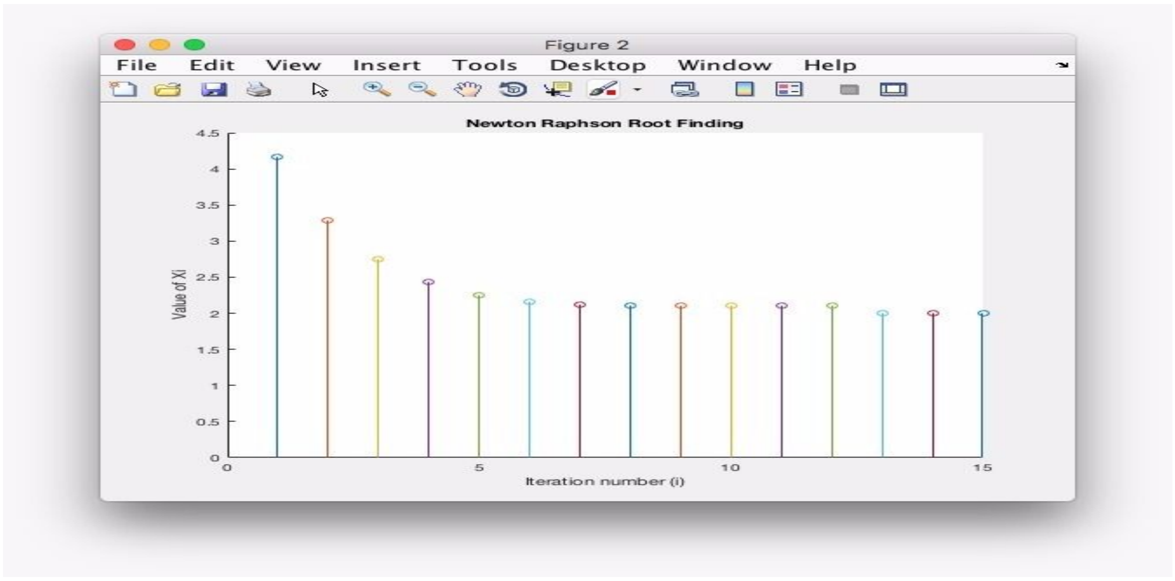


Figure 1
File Edit View Insert Tools Desktop Window Help
Newton Raphson Root Finding
Value of Xi
Iteration number (i)

Graph 2



Graph 3

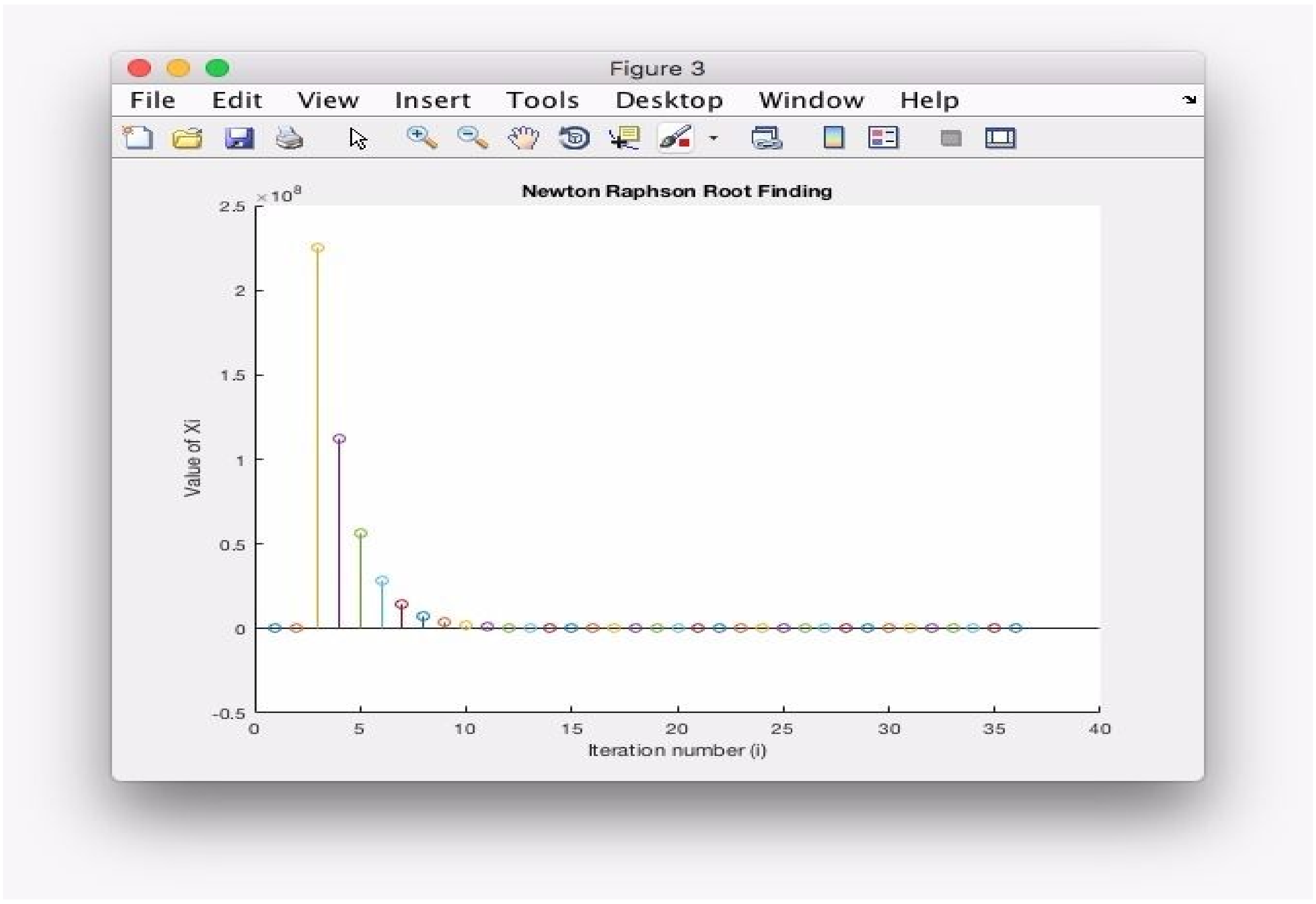
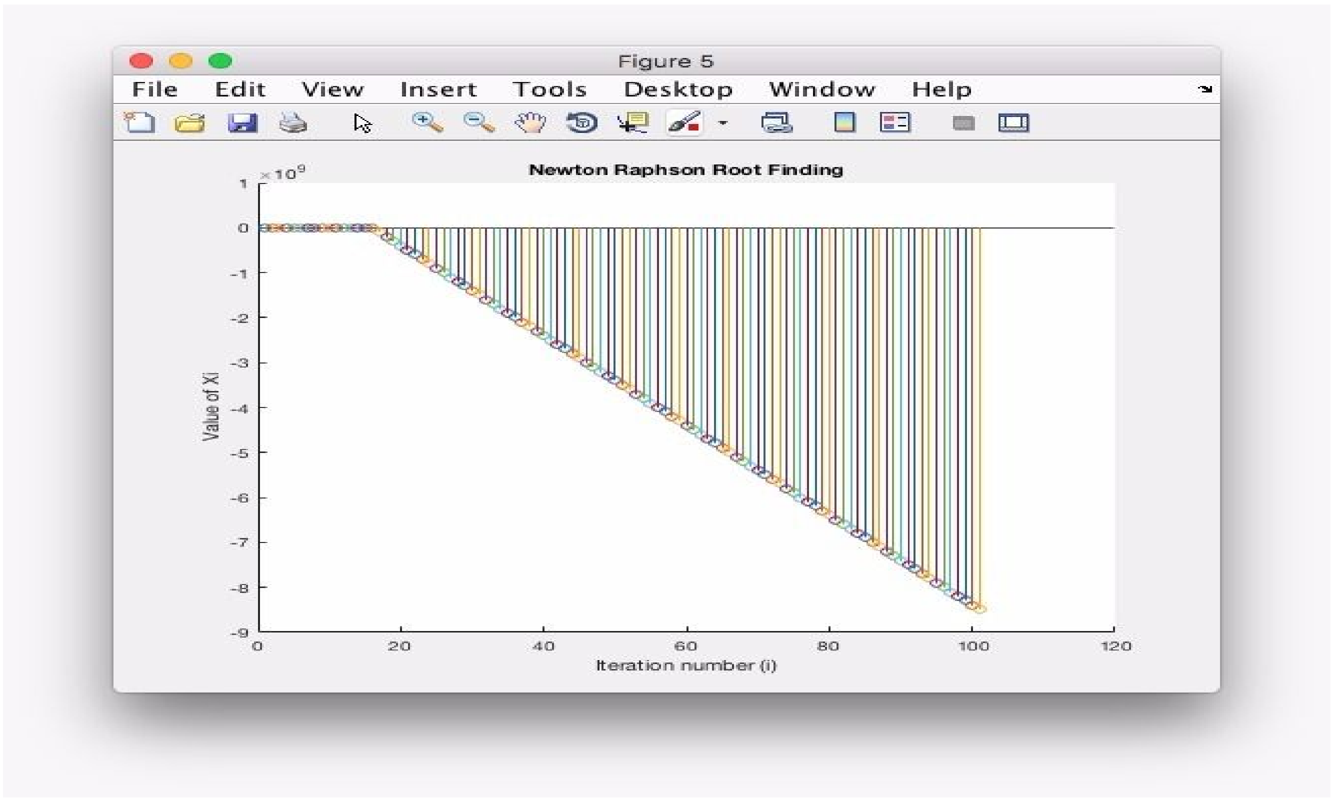
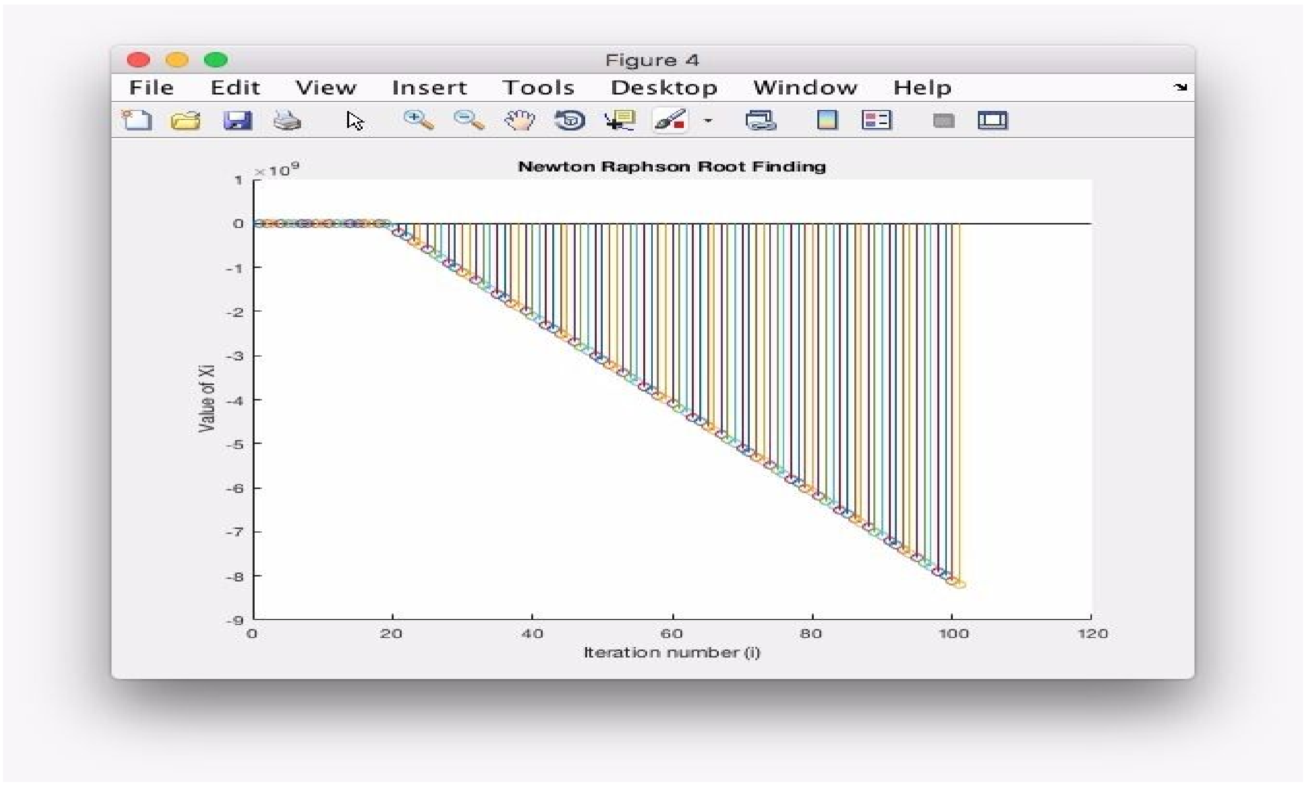


Figure 3
File
Edit
View
Insert
Tools
Desktop
Window
Help
Newton Raphson Root Finding
×10^8
2.5
2
1.5
1
0.5
0
-0.5
Value of Xi
0
5
10
15
20
25
30
35
40
Iteration number (i)

Graph 4



Graph 5



Graph 6

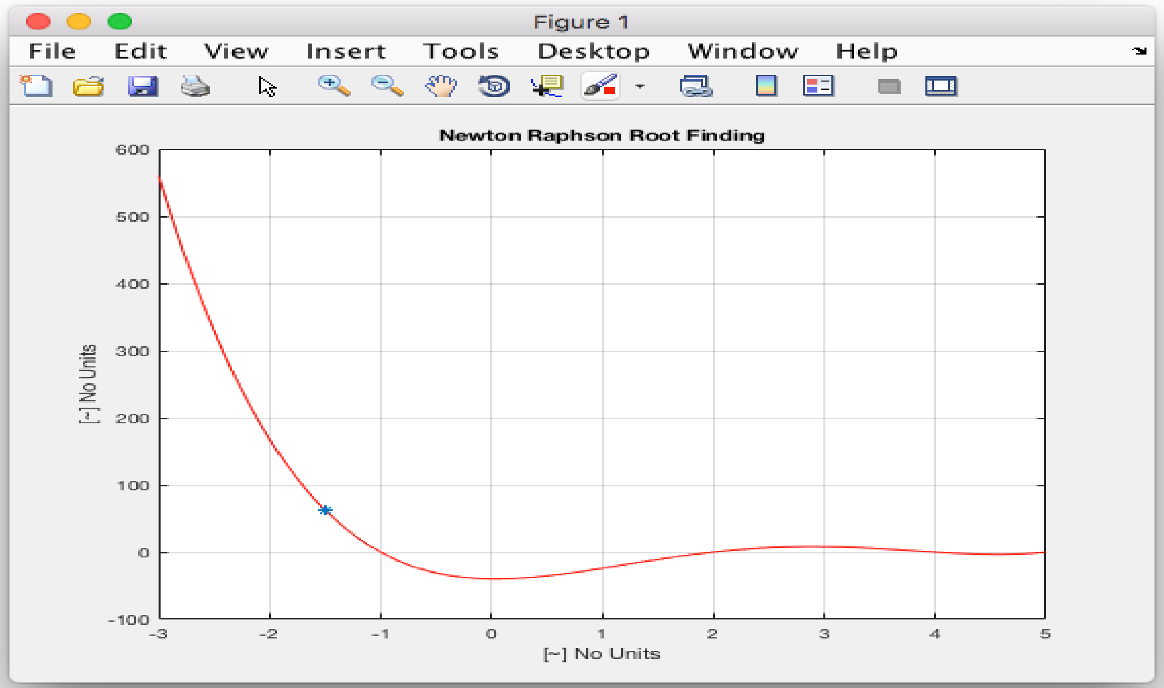


Figure 1
File
Edit
View
Insert
Tools
Desktop
Window
Help
Newton Raphson Root Finding
600
500
400
300
200
100
0
-100
[-] No Units
-3
-2
-1
0
1
2
3
4
5
[-] No Units

Graph 7