Fractal Visualization of the Generalized Newton-Raphson Method for Non-Linear Equations

William Christopher

Section 1. Introduction

The necessity of solving equations is not at all obvious nor a practical concern for the vast majority of the population, most of the times when dealing with equations, we see raw, arid mathematical text and formulas. It is therefore, the purpose of this paper to highlight a highly visually appealing aspect of numerically solving equations and systems of equations.

We will begin in Section 2 to introduce the famous Newton-Raphson's method for solving equations, how the formula is derived using a bit of calculus, with useful intuitions behind the method.

In Section 3, we see how Newton-Raphson's method could be extended to the multi-dimensional case of determined system of equations(when the number of equations equals the number of variables). Some knowledge of linear algebra and multi-variables calculus we be required here.

And finally, the highlight of this article, we visually show that sometimes the method will make "guesses of solutions" converge to the true solutions, while some other times the method fails(when these "guesses" diverge). The success of Newton-Raphson's method depends greatly on the choice of initial guesses, one slight change to a guess could make the method take the guess to a completely different true solution or make it diverge. It is due to this complex dynamics that visualizations of this technique provide intricate fractal patterns.

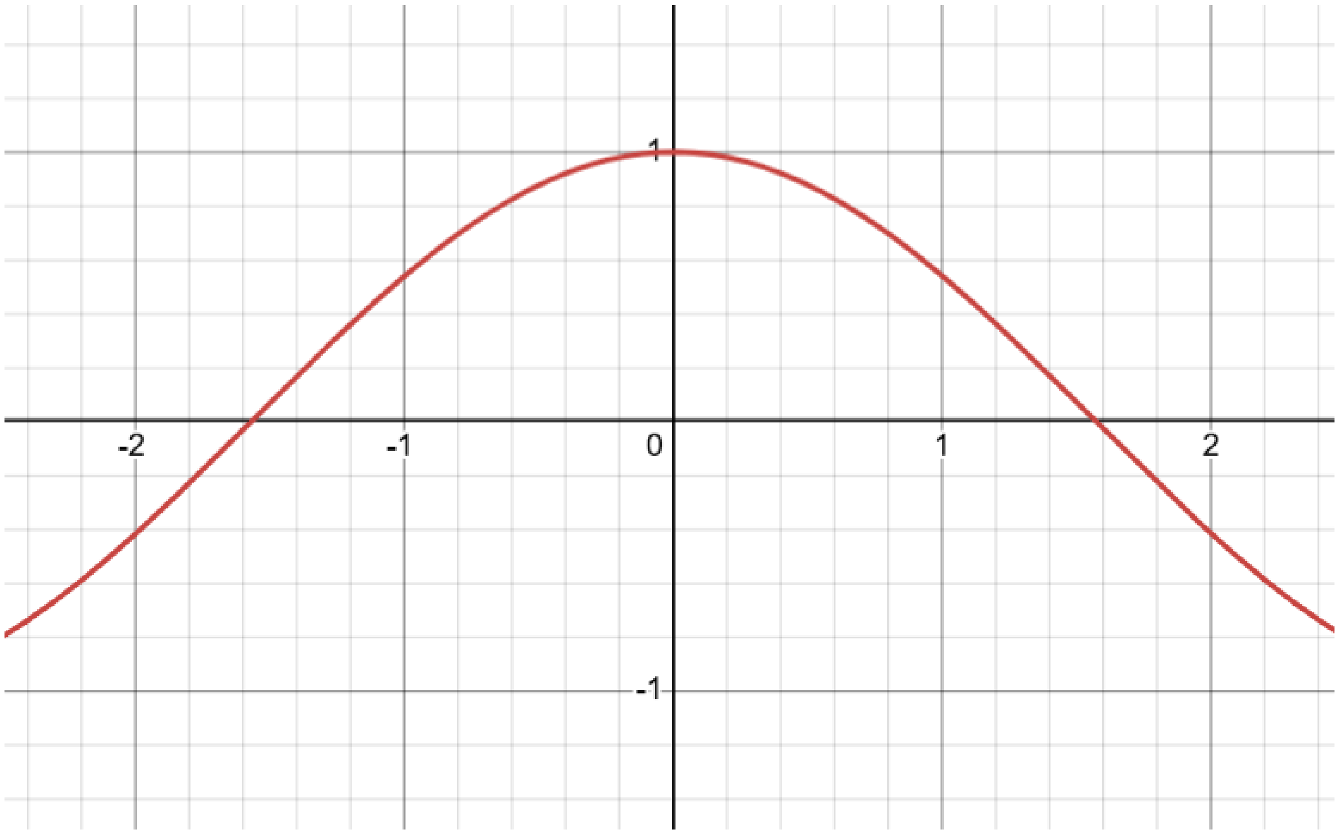
Section 2. Newton-Raphson's roots finding method

Let x be a real number, f(x) is a function that takes x to a different number, we say this explicitly to emphasize that, later on, we will see x and f(x) as any arbitrary dimensional vector. Now, for some reasons, smart people around the world are obsessed with the task of figuring out when f(x) equals 0. We will not discuss what these "practical" reasons here, our motivation right now is knowing how to solve this(or rather, approximately solving) will leads to interesting visualizations.

Formally, we wish to find what value of x makes f(x) equals 0.

Let's pick a function as a concrete example:

$$f(x) = \cos(x)$$



And above is the graph the cosine function(made with Desmos).

We know from high school that cosine(x) = 0 have a solution at x=pi/2(approximately 1.57), but this requires explicit prior knowledge of the function, but this is almost always not the case in practice. What Newton-Raphson's method do instead requires only the following restrictions:

1. The function is differentiable everywhere.
2. The function can be computed everywhere.
3. The derivative of the function can be computed everywhere.

And that is all that needed to find the roots of the function, these conditions are not at all abitrary, since what it boils down to is computing the function and its derivative, and with those two information alone, you can figure out the roots, no manually handling required.

Of course, for simple function that is massively used everywhere, like the quadratic function, there exists cheaper, direct solutions, but what we want is an automated pipeline to solve for any arbitrary kind of exotic equations.

The formula

As is the case with any numerical methods to solve equations, we begin by making a guess of the solution, it doesn't have to be too precise, but the success of most of numerical methods out there depends heavily on the choice of the initial guess, preferably as close to the true solution as possible. We denote this guess x_0.

Next, again, just like other methods out there, we iteratively refine the guess, by applying some certain operation on it. This operation is where the methods differ, some require cheap calculations but yield mediocre convergence, some make guesses converge faster to the true solutions but require more expensive computations, the Newton-Raphson method yields quadratic convergence(pretty fast) but require the computation of the derivative, this requirement is not trivial and sometimes isn't practically possible since the number of operations of a derivative could sometimes grow exponentially compared to the original function(This is especially true for complicated functions).

Here is the update rule to refine the guess to make $f(x^{(k)})$ closer to 0:

$$x^{(k+1)}=x^{(k)}-\frac{f(x^{(k)})}{f'(x^{(k)})}$$

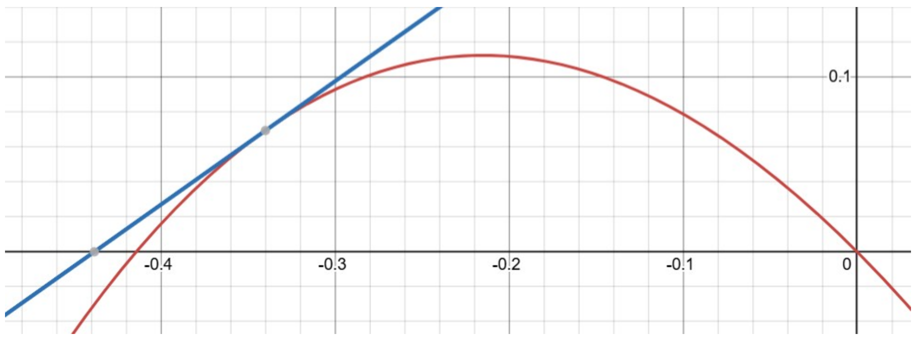
A full understanding of the formula requires understanding the linear approximation of functions, we will get into the intuition of the formula later on, but right now I think it's best to just formally see how the formula is derived.

Suppose we have a function $f(x)$, it is well-known that we could approximate this function locally near any input, hereby denoted α with the tangent line:

$$f(x)\approx l(x)=f(\alpha)+f'(\alpha)(x-\alpha)$$

The reason is that when we zoom in really close to a function around an input, the function looks like a straight line and may as well be the line itself, and what's even better, we can express this line in explicitly!

We should see that when α is close the the root of the function, where the linear approximation(i.e the tangent line) intersects $y=0$ should be close to where the function itself intersects $y=0$ as well. In other words, the root of the tangent line looks really close to the root of the function:

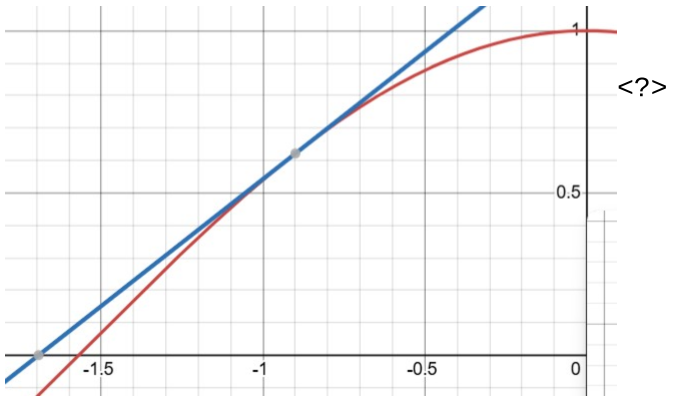


The key take-away is that by updating our next guess to be the root of $l(x)$, the hope is that it should lie closer to the root of the original function, because the root of $l(x)$ is close to the root of $f(x)$. We now proceed to derive the formula for this new(improved) guess by letting x be our next guess that we must find x_{k+1}, and the approximation is around x_k:

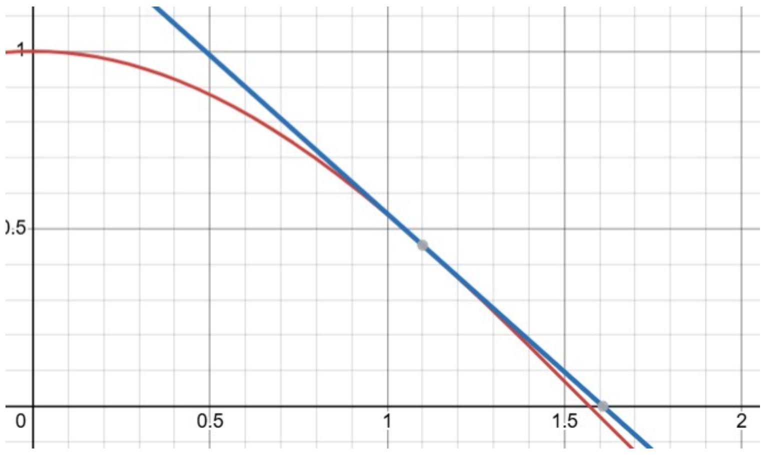
$$f(x^{(k)}) + f'(x^{(k)})(x^{(k+1)} - x^{(k)}) = 0 \Leftrightarrow x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}$$

Here is another intuition:

- Say, your guess $x^{(k)}$ resulted in $f(x^{(k)})$ producing a positive value, and $f'(x^{(k)})$ also producing a positive value (The function is increasing), you'd want to nudge it to the left by subtracting a positive amount:



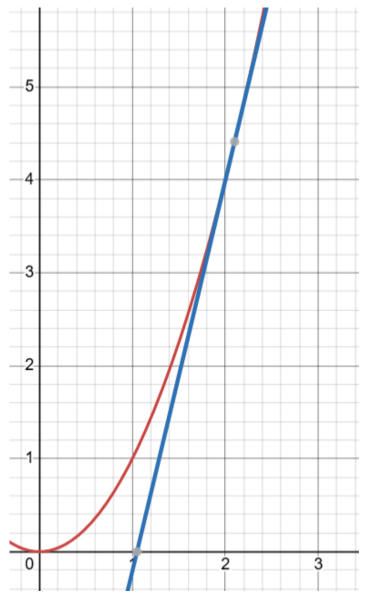
- And if your guess $x^{(k)}$ resulted in $f(x^{(k)})$ producing a positive value, and $f'(x^{(k)})$ producing a negative value (The function is decreasing), you'd want to nudge it to the right by subtracting a negative amount:



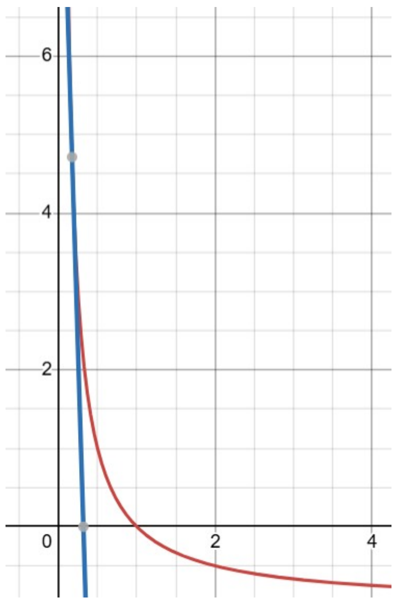
As you can see, after taking a

I encourage the reader to think about the case when $f\left(x^{(k)}\right)$ producing a negative value to get a better intuition on how Newton-Raphson's method works. Above is the intuition on how we should choose the *direction* to update the guess, but how about the *size* of the step?

Well, if is producing a very high value, that means you are still quite far away from 0 and should take a bigger step, so you take a step size *proportional* to the



But watch out! If your function is very steep(high derivative), you should ease off on how much you take the step(since a small step could cause a huge change in the function), we could say the step size is inversely proportional to the derivative:



This concludes the basic Newton-Raphson's method for solving an equation of 1 variable. In the next section, we will see how to generalize the method to system of n equations with n variables.

Section 3. Generalizing to system of equations

Suppose now we have a system of n equations with n variables:

$$\begin{cases} f_1(x_1, x_2, \cdots, x_n) = 0 \\ f_2(x_1, x_2, \cdots, x_n) = 0 \\ \vdots \\ f_n(x_1, x_2, \cdots, x_n) = 0 \end{cases} \Leftrightarrow \mathbf{f}(\mathbf{x}) = \mathbf{0}$$

Notice that from here onward, we denote $\mathbf{f}$ as the vector of functions on the left hand side of the system of equations, as you can see, it has multiple inputs and multiple output, and $\mathbf{x}$ represent the multiple inputs. More formally:

$$\mathbf{x} \in \mathbb{R}^n$$

$$\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^n$$

Taking inspiration from the previous section from the previous section, we start by taking the linear approximation of the function at a guess, preferably near a root, we then set the next guess to be the root of the linear approximation and hopefully it should be closer the root of the function. For a scalar function of multiple variables, its linear approximation at input $\boldsymbol{\alpha}$ is given by the formula:

$$f(x_1, x_2, \ldots, x_n) = f(\mathbf{x}) \approx f(\boldsymbol{\alpha}) + \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}(\boldsymbol{\alpha}) \cdot (x_i - \alpha_i)$$

As you can see, it reminisces the 1 dimensional for tangent line for a one variable function, it's just that now we have to account for the different partial derivatives of the multi-input function. Now is when the magic happens, if we take the linear approximation of all the functions in our system of equations:

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(x_1, x_2, \cdots, x_n) \\ f_1(x_1, x_2, \cdots, x_n) \\ \vdots \\ f_1(x_1, x_2, \cdots, x_n) \end{bmatrix} \approx \begin{bmatrix} f_1(\boldsymbol{\alpha}) + \sum_{i=1}^{n} \frac{\partial f_1}{\partial x_i}(\boldsymbol{\alpha}) \cdot (x_i - \alpha_i) \\ f_2(\boldsymbol{\alpha}) + \sum_{i=1}^{n} \frac{\partial f_2}{\partial x_i}(\boldsymbol{\alpha}) \cdot (x_i - \alpha_i) \\ \vdots \\ f_n(\boldsymbol{\alpha}) + \sum_{i=1}^{n} \frac{\partial f_n}{\partial x_i}(\boldsymbol{\alpha}) \cdot (x_i - \alpha_i) \end{bmatrix}$$

We can then represent this succinctly as simple linear algebra calculations:

$$\mathbf{f}(\mathbf{x}) \approx \begin{bmatrix} f_1(\boldsymbol{\alpha}) \\ f_2(\boldsymbol{\alpha}) \\ \vdots \\ f_n(\boldsymbol{\alpha}) \end{bmatrix} + \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(\boldsymbol{\alpha}) & \frac{\partial f_1}{\partial x_2}(\boldsymbol{\alpha}) & \cdots & \frac{\partial f_1}{\partial x_n}(\boldsymbol{\alpha}) \\ \frac{\partial f_2}{\partial x_1}(\boldsymbol{\alpha}) & \frac{\partial f_2}{\partial x_2}(\boldsymbol{\alpha}) & \cdots & \frac{\partial f_2}{\partial x_n}(\boldsymbol{\alpha}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1}(\boldsymbol{\alpha}) & \frac{\partial f_n}{\partial x_2}(\boldsymbol{\alpha}) & \cdots & \frac{\partial f_n}{\partial x_n}(\boldsymbol{\alpha}) \end{bmatrix} \begin{bmatrix} (x_1 - \alpha_1) \\ (x_2 - \alpha_2) \\ \vdots \\ (x_n - \alpha_n) \end{bmatrix} = \mathbf{f}(\boldsymbol{\alpha}) + \mathbf{J}(\boldsymbol{\alpha})(\mathbf{x} - \boldsymbol{\alpha})$$

The giant matrix in the middle is the Jacobian matrix $\mathbf{J}$ of the function $\mathbf{f}(\mathbf{x})$, evaluated at $\boldsymbol{\alpha}$. If you don't know what's a Jacobian matrix is, it is essentially a mathematical structure to hold the information of the multiple partial derivatives of a multi-valued function with multiple inputs, where:

$$\mathbf{J}_{ik} = \frac{\partial f_i}{\partial x_j}$$

If $\mathbf{x}^{(k)}$ is the local point at which we want to approximate our function at, and $\mathbf{x}^{(k+1)}$ is the input that makes our linear approximation equals 0, that means $\mathbf{x}^{(k+1)}$ should approximately makes the result of our function closer to 0, we now solve for $\mathbf{x}^{(k+1)}$:

$$\mathbf{f}(\mathbf{x}^{(\mathbf{k})}) + \mathbf{J}(\mathbf{x}^{(\mathbf{k})})(\mathbf{x}^{(\mathbf{k+1})} - \mathbf{x}^{(\mathbf{k})}) \Rightarrow \mathbf{x}^{(\mathbf{k+1})} = \mathbf{x}^{(\mathbf{k})} - \mathbf{J}(\mathbf{x}^{(\mathbf{k})})^{-1}\mathbf{f}(\mathbf{x}^{(\mathbf{k})})$$

One of the classical problem solving skills in math should be clear to us now: We want to solve for when an arbitrary function equals zero, but this is maddeningly complicated so we approximate it with a simpler linear function and solve for this linear approximation instead. Look at the formula again and we see a very close resemblance to the original Newton-Raphson's formula, it is precisely how division in the 1D case is now generalized to inverting a matrix in multiple dimensions.

Section 4. Fractal Visualization

We now have a method to solve a system of any arbitrary equations, but this all presumed two important conditions:

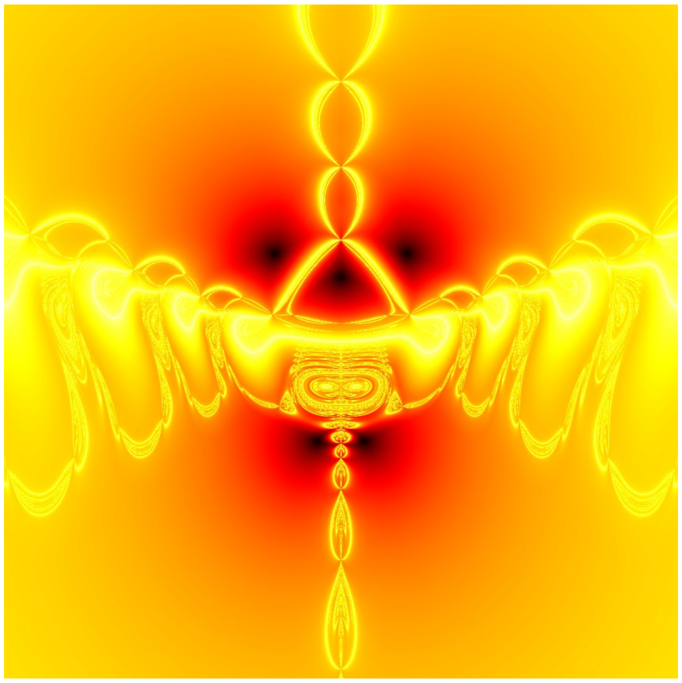
1. The initial guess must be close to a root of the function.
2. The Jacobian matrix must be invertible.

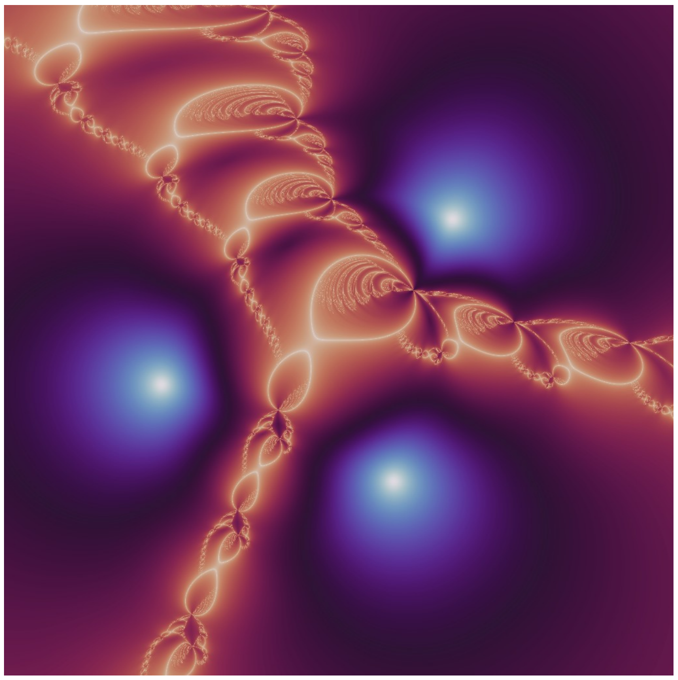
Regarding condition 2, in practice you would observe that this is always satisfied, but to account for the pathological cases where the determinant of $\mathbf{J}$ is zero, we could devise a scheme for regularizing $\mathbf{J}$ or use a pseudo-inverse instead. Still this remains an ugly hole in our method.

For condition 1, finding an initial root that is close to a root of the function is a complex art in and of itself, we could test a bunch of sample points and select the one closest to 0. But usually we would perform the method for multiple sample points and discard any points that diverge.

Because of this inherent instability of Newton-Raphson's method, some initial guess will converge to a true root, others may take hundreds of iterations to converge, stuck in a loop, or diverge to infinity. Newton-Raphson's method is not without its caveat, but could take advantage of this and visualize how chaotic the method is for certain initial guesses. The following images visualize this in the 2D case, for 2 variables and 2 equations, each pixel coordinates are the initial guesses, and the brightness value measures how much each guess must traverse before settling to a root(except for figure 2 whose color palette is using a bright color for stability)

$$\begin{cases} x^3-3xy^2+\exp(-x^2-y^2)\sin(3x)=0 \\ y^3-3x^2y+\exp(-x^2-y^2)\cos(3y)=0 \end{cases}: \qquad \begin{cases} x^3-3xy^2+\sin y=0 \\ y^3-3x^2y+\cos x=0 \end{cases}:$$





$$\begin{cases} x^4-x^2y^2+y-1+0.5\cdot\cos(3x)=0 \\ x^3y-x+0.5\cdot\sin(3y)=0 \end{cases}: \qquad \begin{cases} x^4-x^2y^2+y-1+0.5\cdot\cos(3x)=0 \\ x^3y-x+0.5\cdot\sin(3y)=0 \end{cases}:$$