Comparisons Of Numerical Methods

Introduction & Rationale

Every year, millions of middle and high-schoolers study some of the hundreds of approaches to solving equations for zeros ($f(x) = 0$), first learning to solve linear equations, then quadratic equations, then learning to group higher-order polynomials, and finally learning to solve trigonometric equations. What sparked interest in this topic for me was 'what comes after that?' We currently have no explicit solution-finding formulae that can generalize for all polynomials of degree 5+ [Abel-Ruffini Theorem], so what do calculators use in lieu of a 'quintic formula'? If we're assuming 'just plot it on your GDC' doesn't count, probably the best and most straightforward option will be numerical methods. These are algorithms specifically designed to find functions' zeros, approximating them to a degree of accuracy proportional to the number of iterations used. Numerical methods are used in a wealth of use cases, including our very own TI-NSpire calculators on functions like 'n-solve' and for finding *n*th-roots (as will be explained later).

This investigation aims to explore and compare three distinct numerical methods (Bisection, Newton, Secant) with regard to efficiency and reliability, as well as investigate how they can be used in a variety of scenarios. This will be conducted by first introducing and explaining each of the methods, then investigating approaches to compare their relative zero-finding speeds including their order of convergence, and finally delving into their applications in the context of mathematics including root-finding.

Note on terminology:

a) Given the possibility of confusion between the terms 'zeros' and 'roots', I will exclusively use 'zero' or 'x-intercept' to refer to a solution to $f(\alpha) = 0,\ \alpha \in R$ and exclusively use 'root' to refer to $r = \sqrt[n]{x}$.

b) Newton's Method vs. Newton's Iteration vs. Newton-Raphson. In this investigation 'Newton's Method' and the 'Newton-Raphson method' will be used interchangeably, while Newton's Iteration would refer only to a context in which Newton's method has been applied to solve for the square root of a constant.

Bisection method

The bisection method relies on a fundamental property of continuous functions: the Continuous Value Theorem. This theorem postulates that between two points of opposite signs on a continuous function, there must be a point at which $f(c) = 0$:

$$c \in [a, b],\ \ sign\, f(a) \neq sign\, f(b)$$

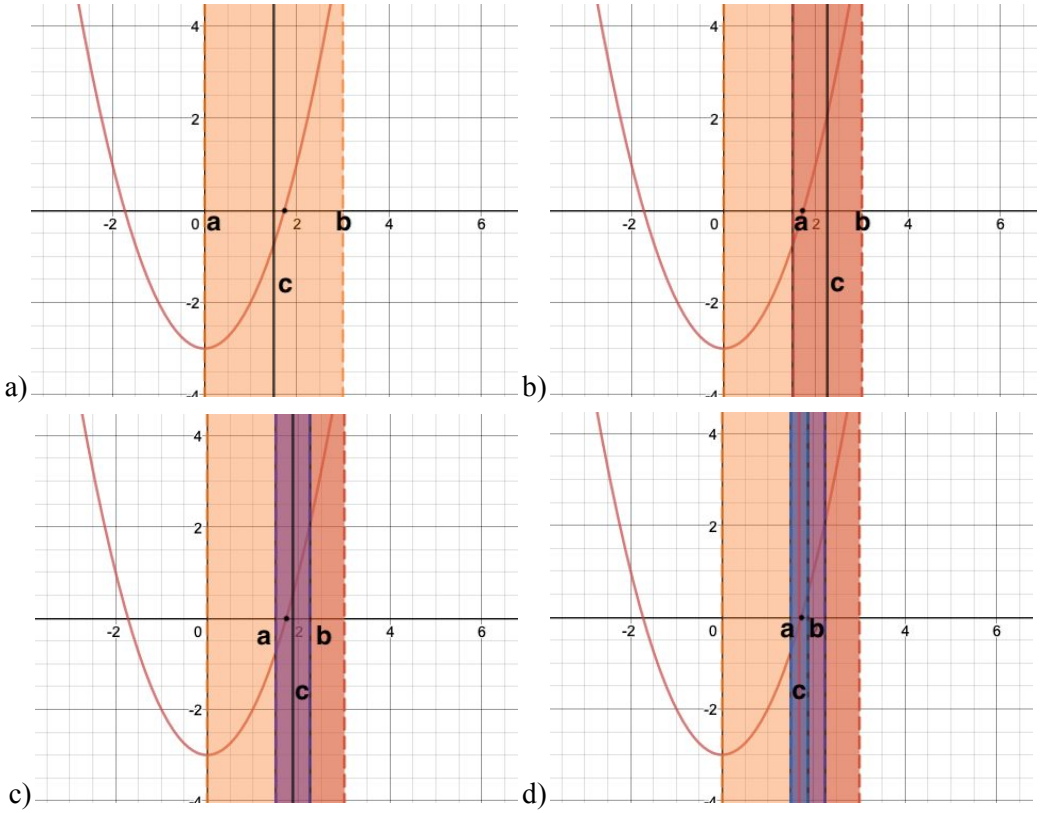
And so there must therefore be a value c for which

$$f(c) = 0$$

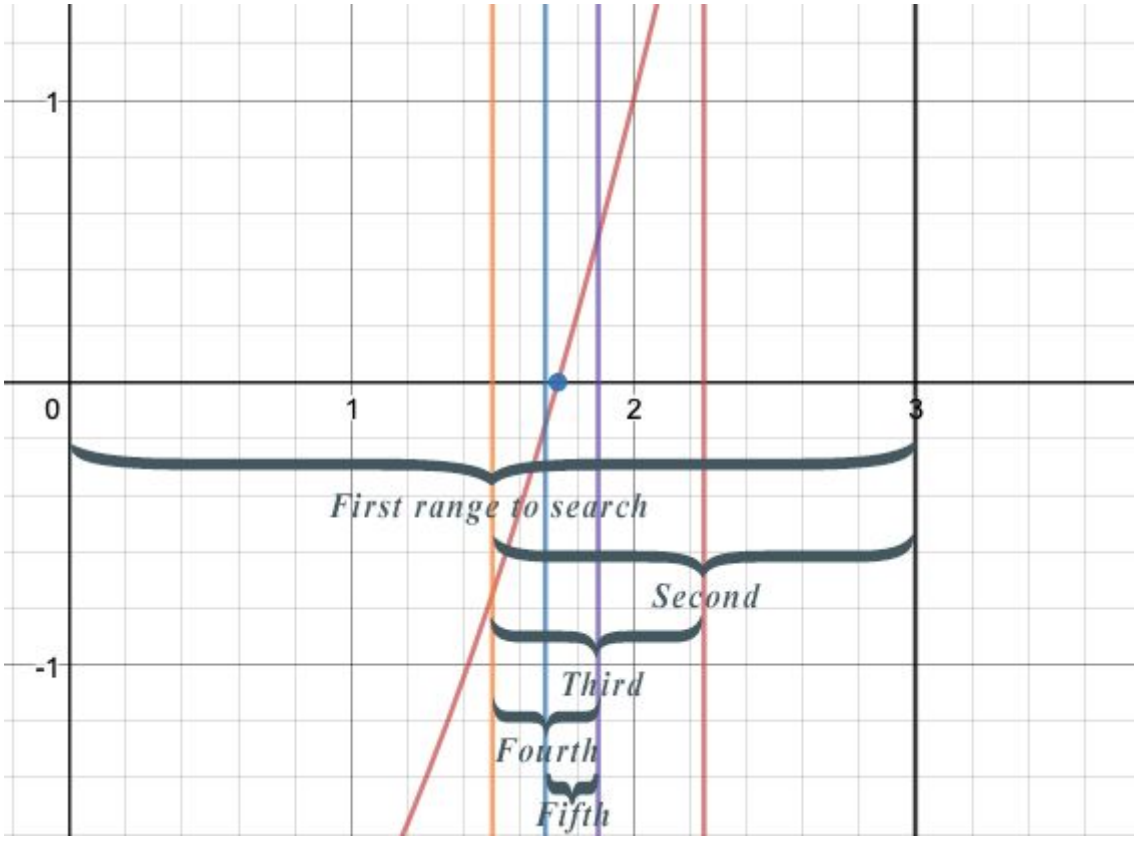
The bisection method exploits this, by subdividing a closed interval $[a, b]$ into two equal halves with midpoint c

$$c = \frac{|b-a|}{2}$$

It then checks which of the two halves contains a change in sign between a and b, given that the half in which $sign\, f(a) \neq sign\, f(b)$ will contain the zero. Once done, it repeats itself and subdivides that half with midpoint c, then checks the signs, etc.



The graphs above show 4 iterations of the Bisection method on the function $f(x) = x^2 - 3$ with a zero at $x = \sqrt{3}$ and bounded in a closed domain of $x \in [0, 3]$ such that it approximates the positive root/zero. As can be seen, on each iteration the possible interval containing the zero is one half from the previous iteration (represented by the layers of shaded area).



until it reaches an approximation of the root within a certain tolerance.

Drawbacks

Unfortunately, the main drawback associated with the Bisection method is it's slowness compared to other numerical methods. From a theoretical standpoint each new iteration of the bisection method should half its absolute error

$\varepsilon = |r - x_n|$ where r is the true zero being approximated because the interval that could contain r is halved each iteration. This can be represented by the inequality below, where a and b represent the initial interval, and c the initial midpoint.

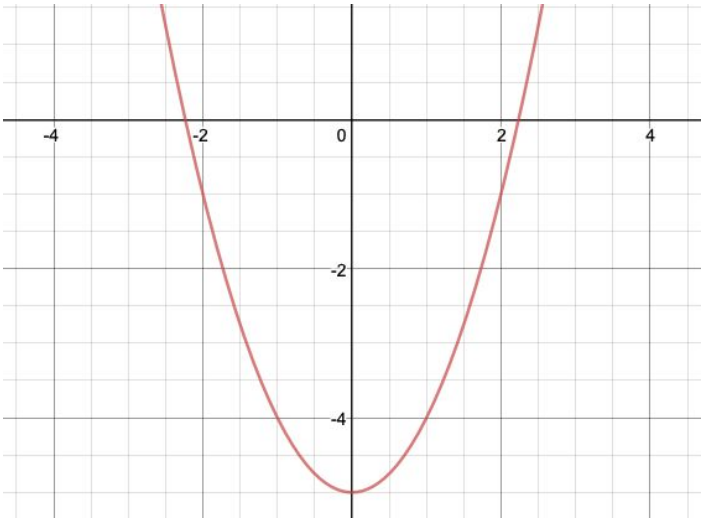
$$\varepsilon_n = |c_n - c| \leq \frac{|b-a|}{2^n},\ n = [0, 1, 2...]$$

This relationship must be modeled as an inequality due to the fact that $\frac{|b-a|}{2^n}$ represents the absolute error from the midpoint (positive or negative) of the iteration, while the the actual distance from the root could be significantly lower (e.g. Fig. 7 on iteration 1, the absolute error is $\frac{3}{2}$ while the midpoint's actual distance from the zero is only $\sqrt{3} - \frac{3}{2} \approx 0.232$.

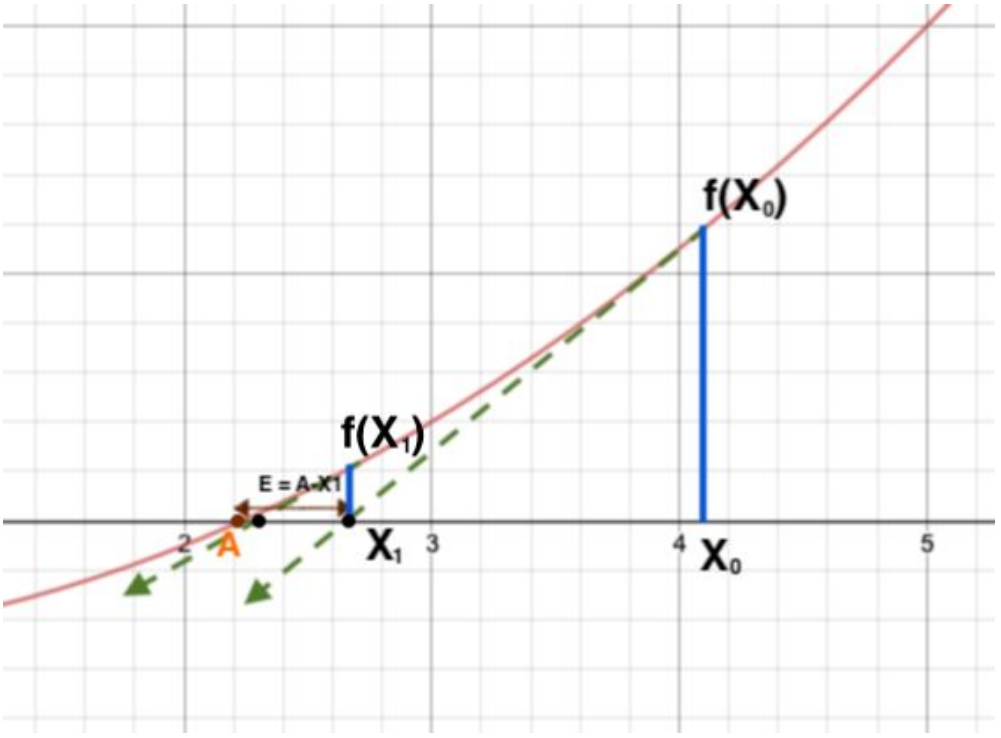
Newton's method

A good rule of thumb for numerical methods is that the more information about the function is provided, the faster the method's rate of convergence. Newton's method takes a wonderful approach to solving for $f(x) = 0$, by using calculus to find the gradient of a point on the function, and using that gradient/tangent line to guess where the function is going to contain a zero.

Consider the curve $f(x),\ y = x^2 - 5$

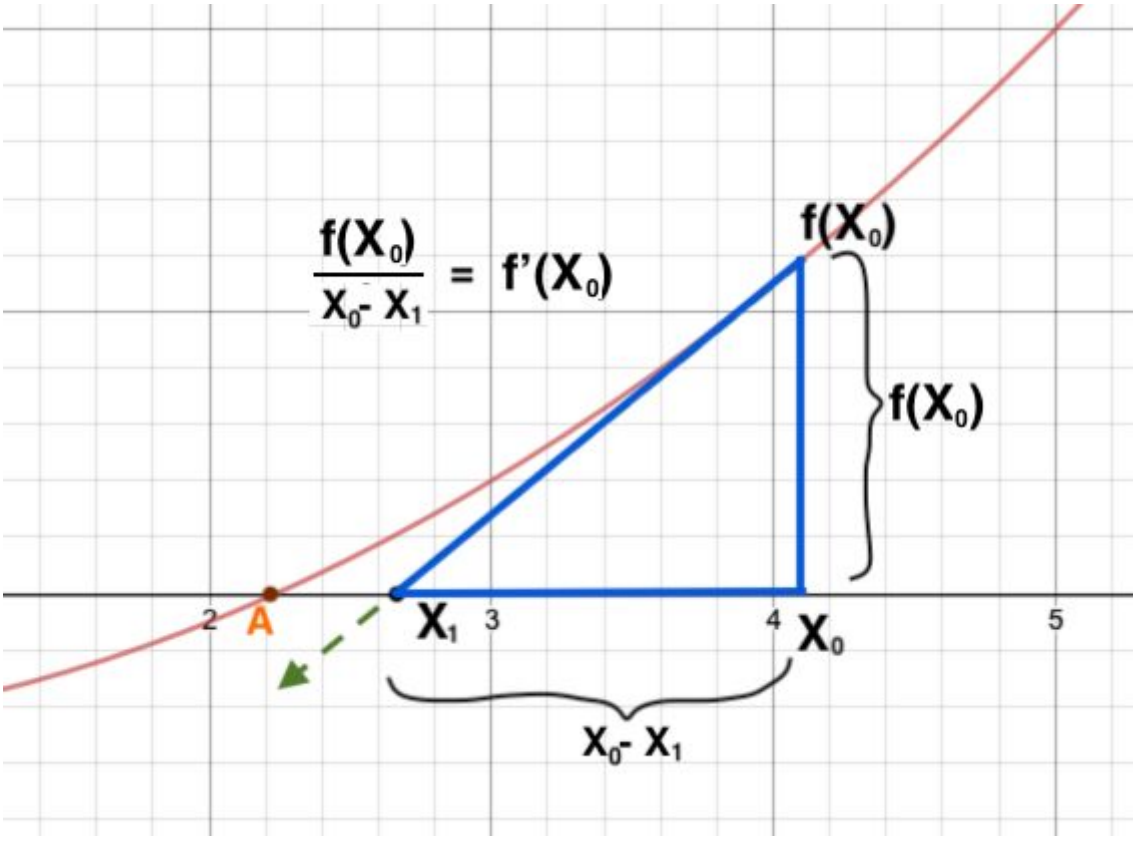


It can be factorized as $y = (x + \sqrt{5})(x - \sqrt{5})$ to give the two zeroes of the graph: $\pm\sqrt{5}$



The initial guess x_0 will have a corresponding point on the function $f(x_0)$. The tangent of the curve at the point $f(x_0)$ will cross the x-axis with an error $e_1 = |\alpha - x_1|$, $\alpha = \sqrt{5}$, wherein x_1 denotes the new intersection of the x-axis, or in the context of Newton's method, the second iteration's estimation of the function's zero.
In this current state though, we don't know what x_1 is, which is where the proof for the Newton's method comes in handy:

If one were to draw a right-angled triangle over our curve, $f(x_0)$ could represent the vertical component and the difference $x_0 - x_1$ the horizontal component.



In order to find the unknown x_1 we could use the equation of the line and solve it for $x = 0$, although that would be inefficient. What's so cool about this method is that it doesn't even require all those calculations. Instead, we can first find the gradient of the function at x_0 using the rise over the run:

$$gradient = \frac{f(x_0)}{x_0 - x_1}$$

Which we know is equal to the derivative of a function at a point, so:

$$f'(x_0) = gradient = \frac{f(x_0)}{x_0 - x_1}$$

And finally this can be rearranged to solve for x_1 as such:

$$x_0 - x_1 = \frac{f(x_0)}{f'(x_0)}$$
$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

This is the essence of Newton's method. A new approximation is found by subtracting the quotient of $f(x)$ and $f'(x)$ at the previous point from the previous x estimation. Here it is written with domains and generalized:

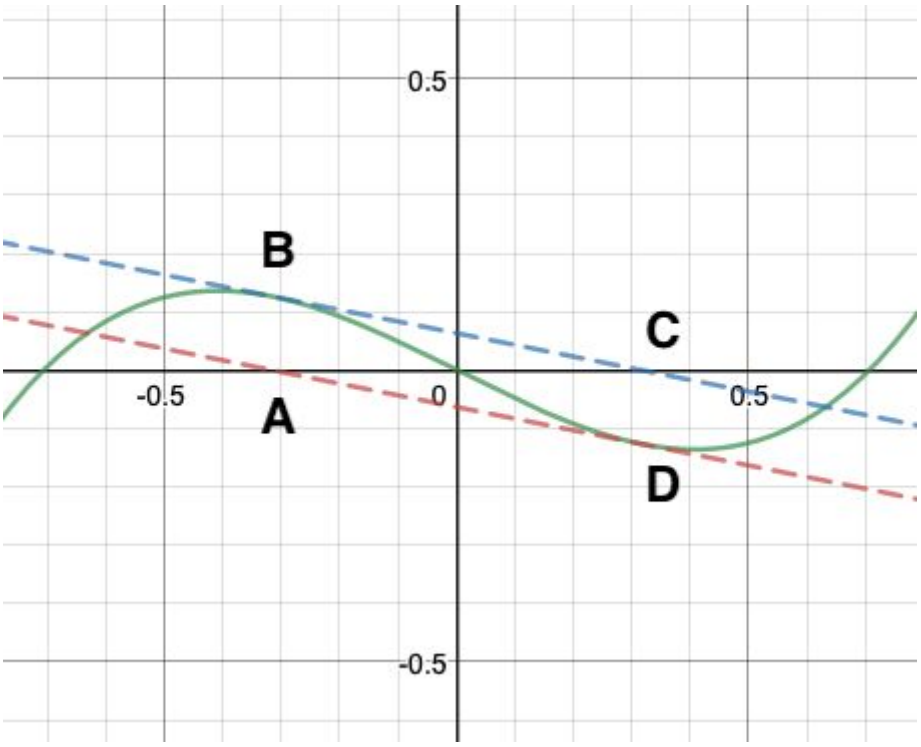
$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)},\ k \in Z+,\ f'(x) \neq 0$$

Drawbacks

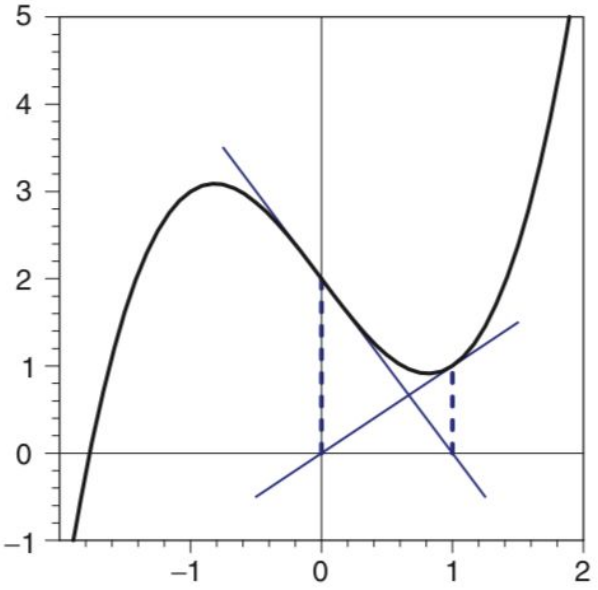
That denominator, $f'(x) \neq 0$, is a hint at the main problem with Newton's method: its instability. Newton's method can 'break' in 3 main ways:

a) The function's derivative is equal to 0 ($f'(x) = 0$), resulting in a division by zero error in the formula. From a geometrical perspective, this can be understood as a horizontal tangent at a point, which will therefore not intersect the x-axis and prevent us from finding an x-intercept.

b) The target zero is a point of inflection (POI), causing the method to alternate between a positive and negative value near the POI. (See diagram below).



Here, if an iteration of x_k returns point A, the subsequent iteration will determine point C as x_{k+1} (since B's tangent intersects the x-axis at C), which in turn will yield point A as x_{k+2} ad infinitum. The method will cycle between the two opposite points on the POI indefinitely. This is also known as a 'superattracting cycle' (which can occur outside of POIs as well), which, as the name suggests, 'attract' the attention of the method, and lock it indefinitely in a cycle of 2 points or more.



This diagram from the paper on Newton's method for finding complex roots by M. Hubbard [Hubbard] is quite effective at illustrating a superattracting 2-cycle. The gradient of the left point intersects the x-axis at the right point, whose gradient intersects the x-axis at the left point, and so it continues until the max iteration count is exceeded in the algorithm.

c) The function has multiple zeros, and the method converges on the incorrect one. The method may converge on an unwanted solution if it contains multiple zeros, yielding a possibly useless zero answer (e.g. a negative zero when solving an optimization problem for a physical property like volume). This can be somewhat alleviated thanks to real-root isolation, which is the process of identifying closed intervals in which each root can be found, before

even starting the numerical method. It goes hand-in-hand with other observations that can be made on the zeros of polynomials using techniques like Descartes' rule of signs.

Secant method

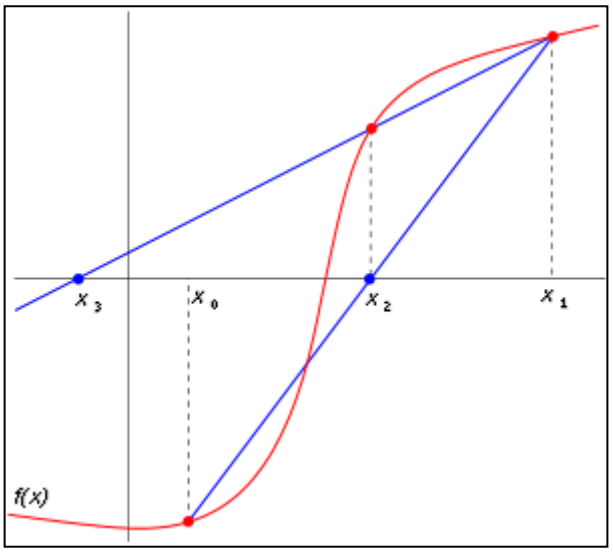
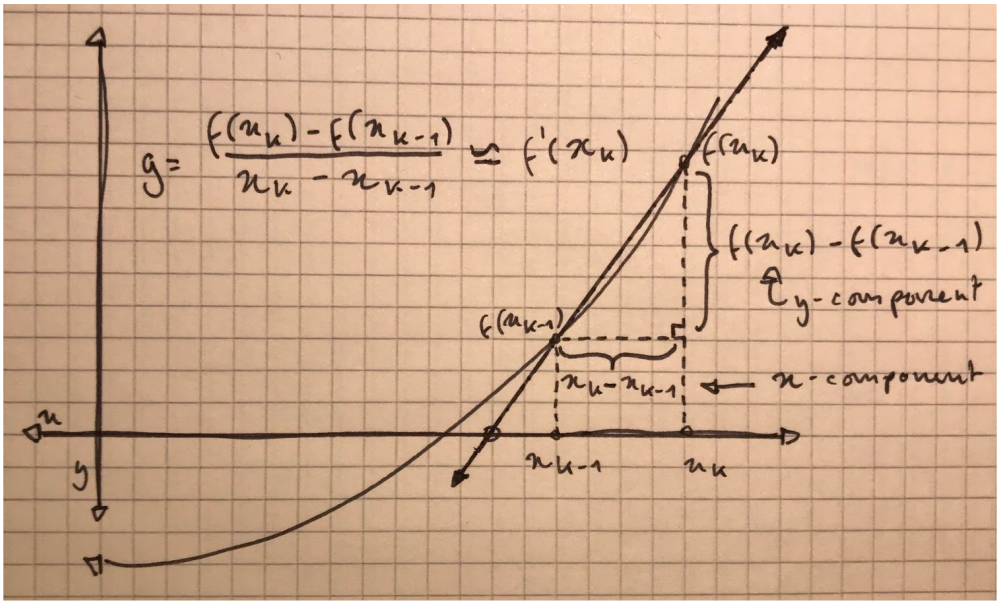


Diagram sourced from wikimedia commons [Jitse Niesen]

The secant method is derived from Newton's method, which is part of the reason the pair are so closely linked.

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

The bottom term in Newton's method is the derivative of the function, which normally we would need to compute symbolically and then carefully chose starting points so that $f'(x) \neq 0$ for reasons outlined earlier. The Secant method, meanwhile, allows us to estimate this derivative using a process called the 'backward divided difference', both making the numerical method more stable and its initial parameters easier to calculate (given that it removes the need to algebraically find the derivative).



The diagram above shows two points x_{k-1} and x_k on the x-axis, with their associated $f(x_{k-1})$ and $f(x_k)$. If we are trying to estimate the gradient at point x_k we can use the horizontal and vertical components of the right-angled triangle formed between the two coordinates to find the gradient, which should approximate the actual tangent at x_k. This gradient can therefore be expressed as:

$$gradient = \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}} \simeq f'(x_k)$$

Where the numerator is the y-component and the denominator the x-component. Now that we have a relatively solid estimation for $f'(x_k)$, we can substitute it into Newton's method as such:

$$x_{k+1} = x_k - \frac{f(x_k)}{\frac{f(x_k)-f(x_{k-1})}{x_k - x_{k-1}}}$$

$$x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k)-f(x_{k-1})}$$ Q.E.D.

We now have a working expression of the Secant method. One aspect to note is that the user must provide both an x_0 and x_1 as the first two terms for the equation, since this allows a secant (a line that intersects a curve in two or more places) to be drawn between them and then the gradient computed in the steps outlined above. Useful values for these terms can be

Drawbacks

a) While the secant method is overall a highly reliable numerical method, it can still be broken if both $f(x_k) = f(x_{k-1})$ which will raise a division by zero error--although this is significantly less likely than in Newton's method.

b) The secant method makes the small sacrifice of convergence speed for a more stable overall numerical method. It has an order of convergence of ~1.618, while Newton's method has an order of 2, showing the difference in speeds between them.

Nth-Root-finding (Newton's Iteration)

Now that we are equipped with powerful zero-finding numerical methods, we can now use them on almost any equation that has been rearranged to solve for $f(x) = 0$. One such example of this is the square root of a number α where $\alpha \in R^+$ using a function of form $f(x) = x^2 - \alpha$.

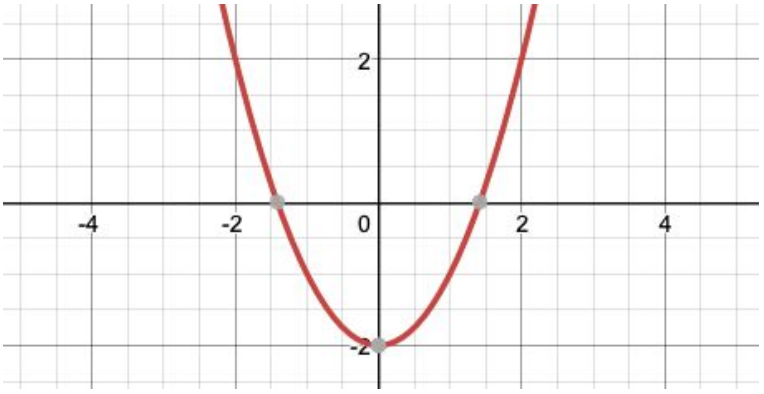
$y = x^2 - \alpha$ can be factorized into

$$y = (x + \sqrt{\alpha})(x - \sqrt{\alpha})$$

If we solve f(x) = 0, it will yield zeroes at $x = \pm\sqrt{\alpha}$ which give us the square root of our number α

This is where our numerical methods come in handy; they excel at finding zeroes, and hence can be used to rapidly find $\sqrt{\alpha}$.

Let's take the example of $\alpha = 2$. $f(x) = x^2 - 2$ has the following graph, on which one can clearly that the function intersects the x-axis at $\pm\sqrt{2}$ (≈ 1.4121).



Now, we can use our numerical methods to attempt to find this value:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \rightarrow 1.5 = 2 - \frac{2}{4}$$

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \rightarrow 1.41667 = 1.5 - \frac{0.25}{3}$$

$$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)} \rightarrow 1.41422 = 1.41667 - \frac{0.006944}{2.833333}$$

$$x_4 = x_3 - \frac{f(x_3)}{f'(x_3)} \rightarrow 1.41421 = 1.41422 - \frac{0.000006}{2.82843}$$

After just four iterations with Newton's method, we arrive at 1.41421--the literature value for $\sqrt{2}$ for 6 s.f.

Next I wondered whether it would be possible to use this method for higher-index roots. I knew that I would be looking for a function with at least one zero of $(x - \sqrt[n]{\alpha}) = 0$, since this worked for square roots. After a bit of testing I realized that actually I only had to change my formula slightly to accommodate higher-index roots:

$$f(x) = x^n - \alpha \text{ where } n \in R$$

Which could be generalized in a numerical method like Newton's Method for the *n*th root as:

$$x_{k+1} = x_k - \frac{{x_k}^n - \alpha}{n{x_k}^{(n-1)}}$$

This is most likely the algorithm used in calculators to rapidly find the *n*th root of a number. Using a python script which can be seen in the appendices [Nth Root Code], my computer took only 0.1 milliseconds to find $\sqrt[3]{197}$ to 5 d.p. (the standard on a TI-Nspire calculator):

```
(base) luca-macbook-pro:zero_algorithms lucamehl$ python3 nth_root.py
root value:  2.876691209017467
time taken:  0.00012493133544921875
(base) luca-macbook-pro:zero_algorithms lucamehl$
```

Rate and Order of convergence

Now that we know how to use our three numerical methods and some of their pitfalls, we can proceed to compare them in terms of speed/efficiency. This is one of the most important factors to consider when choosing one method over another, given that what may take 10 steps for one to find may take thousands in another.

The main differences in solving speed can be expressed through two factors: rate of convergence μ and order of convergence *q*. In a sequence (resulting from a numerical method) that converges linearly to a term $r = x_{k\to\infty}$, the rate of convergence can be expressed as:

$$\lim_{k\to\infty} \frac{|x_{k+1} - r|}{|x_k - r|} = \mu$$

Explained in words, the expression outlines that as the sequence approaches infinite terms, the rate of convergence will be equal to the next distance from the zero *r* over the previous distance from *r*. Since $|x_{k+1} - r| \leq |x_k - r|$ in all three methods (unless they broke because they were ill-suited to their target function), μ must be a number in the range [0,1]. It also follows that if $\mu_k \to 0$ for $k \to \infty$, it must mean that the x_{k+1} term is increasingly closer to *r* than the previous iteration x_k, hence branding the rate of convergence as 'superlinear'. The converse is $\mu_k \to 1$ for $k \to \infty$, where the ratio tending to 1 indicates that the current and previous term become more similar, and therefore make slower strides toward approximating the zero. This is referred to as sublinear convergence.

The other factor--order of convergence *q*--is what makes Newton's method so appreciated for zero-finding. Numerical methods like Newton's method have the added benefit of an exponent on the bottom term, allowing for exceptionally fast convergence on the zero, which is referred to as Q-quadratic convergence. This can be shown by the expression below.

$$\lim_{k\to\infty} \frac{|x_{k+1} - r|}{|x_k - r|^q} = \mu,\ q \geq 1$$

Normal proofs of Newton's method converging quadratically require multivariable calculus and Taylor series expansions (e.g. see [Gockenbach]), so to avoid that I will attempt to prove it using my own proof using the above definition of q-order convergence:

$$\frac{|x_{k+1} - r|}{|x_k - r|^q} = \mu$$

$$|x_{k+1} - r| = \mu |x_k - r|^q$$

$$|x_k - r| = \mu |x_{k-1} - r|^q$$

$$\frac{|x_{k+1} - r|}{|x_k - r|} = \left(\frac{|x_k - r|}{|x_{k-1} - r|}\right)^q$$

$$log\left(\frac{|x_{k+1} - r|}{|x_k - r|}\right) = q \cdot log\left(\frac{|x_k - r|}{|x_{k-1} - r|}\right)$$

$$q = \frac{log(\frac{|x_{k+1}-r|}{|x_k-r|})}{log(\frac{|x_k-r|}{|x_{k-1}-r|})}$$

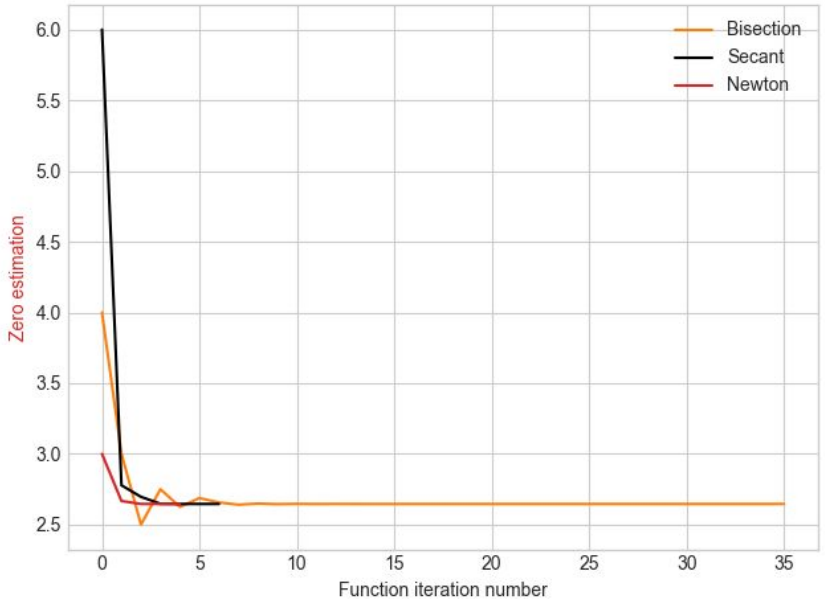
Then we can substitute in some values from an iterations table for the function $f(x) = x^2 - 5$ which has a root we know to be $\sqrt{5}$.

Iteration number (I)	Iteration number in equation	Zero approximation
1	x_{k-1}	2
2	x_k	2.25
3	x_{k+1}	2.23611

$$\frac{log(\frac{|2.23611-\sqrt{5}|}{|2.25-\sqrt{5}|})}{log(\frac{|2.25-\sqrt{5}|}{|2-\sqrt{5}|})} = 2.05084$$

The final expression yields a result of 2.05--not bad at all given that it should give an answer of 2. The 0.05084 error could have originated from an insufficient number of decimal places available for x_{k+1}, and may also have been exacerbated by imprecision at the start of the iteration / an imperfect first guess. I would theoretically get a better result if I used three concurrent values from later iterations (e.g. I=40,41,42), since the function would have stabilized a lot more by that time. Regardless, this value of q=2.05 serves as an effective demonstration that the rate of convergence q in Newton's method is quadratic.

Lastly, I will compare the 3 methods using the Python library Matplotlib and the python script listed under the appendices as [Graph Code]:



As can be seen on the graph, the Newton and secant methods converge significantly faster to the desired root (in this case $\sqrt{7}$ given a function $f(x) = x^2 - 7$) than their bisection counterpart. As expected, Newton's method [Newton Code] converges the fastest to the desired 6 d.p. in only 4 iterations, while the secant method [Secant Code] takes 6 and bisection method [Bisection Code] a massive 35.

Conclusion and Evaluation

I was intrigued to find that when I attempted to use the proof I derived for isolating the order of convergence of a method on either of the other two methods (secant and bisection), I received answers different from those quoted in literature. In future investigations I would be curious to see whether performing the expression at 10 or 20 intervals and then averaging the answers could give me a more complete solution for their orders of magnitude. I'm especially interested to do this for the secant method, given that it has an order of convergence equal to the golden ratio [Thompson], and I would be interested to find out why.

While perhaps beyond the scope of my exploration, I discovered two highly extensive topics that are still being researched in numerical analysis: numerical method acceleration, and real-root isolation. Currently the methods I've discussed are very classical, but for real computing purposes they must be adapted and merged with other methods (e.g. Miller method) in order to maximize efficiency and be compatible with preliminary real-root isolation techniques, such as to identify discrete intervals for each real root.
The one thing I really wish I had had time to research and talk about in my exploration was using / adapting numerical methods for complex zeros. A lot of the research I read ended up pointing towards papers in complex zero-finding, and many seems to suggest that the graphs of complex functions are not longer linear/continuous, but follow fractal patterns like those in papers published by [Downs] and [Hubbard].

Lastly, Newton's Method, the Secant Method, and the Bisection method are only the tip of the iceberg; hundred of other algorithms exist, with applications ranging from plotting mathematical topography to solving Markov chains for neural nets. Further exploration should definitely be conducted into the depth and applications of these numerical models.

References:

[Abel-Ruffini Theorem]
"Abel–Ruffini Theorem." *Wikipedia*, Wikimedia Foundation, 30 Mar. 2019, en.wikipedia.org/wiki/Abel%E2%80%93Ruffini_theorem.

[Jitse Niesen]
Niesen, Jitse. "Illustration of the Secant Method." *Wikimedia Commons*, 19 June 2006, commons.wikimedia.org/wiki/File:Secant_method.svg.

[Gockenbach]
Gockenbach, Mark S. "Proof of Quadratic Convergence of Newton's Method." *Proof of Quadratic Convergence of Newton's Method*, 23 Jan. 2003, pages.mtu.edu/~msgocken/ma5630spring2003/lectures/newton/newton/node4.html.

[Downs]
Downs, Michael R.. "Math 56 Newton Fractals Michael Downs 1 Newton's Method." (2014).

[Hubbard]
Hubbard, John, et al. "How to Find All Roots of Complex Polynomials by Newton's Method." *Inventiones Mathematicae*, vol. 146, no. 1, 2001, pp. 1–33., doi:10.1007/s002220100149.

[Thompson]
Thompson, Skip. "Convergence of the Secant Method." *Radford Department of Mathematics*, 2010, www.radford.edu/~thompson/Fall10/434/Chapter4/secant_convergence.pdf.

Appendices:

[Bisection Code] Bisection method python algorithm:

```
import numpy as np

def Bisection_clean(f, r1, r2, tol, maxIt=50):
    """
    [f] is the function to be used
    [r1] is the first initial guess
    [r2] is the second initial guess
    [tol] is the tolerance
    [maxIt] is the maximum number of iterations
    """
    a = r1
    b = r2
    c = 0

    i = 0
    while i < maxIt:
        co = c
        c = ( (a + b) / 2 )
        print(b,c)
        if abs(b-c) < tol:
            break
        if np.sign( f(b) ) * np.sign( f(c) ) <= 0:
            a = c
            print("true")
        else:
            b = c
        i+=1
    return c
```

[Newton Code] Newton's method python algorithm:

```
import numpy as np

def Newton_clean(f, fp, inita, tol, maxIt=50):
    """
    [f] is the function to be used
    [fp] is the derivative of the function to be used
    [init] is the initial guess
    [tol] is the tolerance
    [maxIt] is the maximum number of iterations
    """
    x = inita

    i = 0
    while i < maxIt:
        xn = x - ( f(x) / fp(x) )
        if abs(xn - x) < tol:
            break
        x = xn
        i+=1
    return x
```

[Secant Code] Secant method python algorithm:

```
import numpy as np

def Secant_clean(f, inita, initb, tol, maxIt=50):
    """
    [f] is the function to be used
    [init] is the initial guess
    [tol] is the tolerance
    [maxIt] is the maximum number of iterations
    """
    xo = inita
    x = initb
    xn = 0

    i = 0
    while i < maxIt:
        xn = x - ( f(x) * (x - xo) ) / ( f(x) - f(xo) )
        if abs(xn - x) < tol:
            break
        xo = x
        x = xn
        i+=1
    return x
```

[Nth Root Code] Newton's method used for finding the *n*th root of a number python algorithm:

```
import numpy as np
import time


def f(x):
  return x**5 - 197

def fp(x):
  return 5*x**4

def Newton_clean(f, fp, inita, tol, maxIt=50):
  """
  [f] is the function to be used
  [fp] is the derivative of the function to be used
  [init] is the initial guess
  [tol] is the tolerance
  [maxIt] is the maximum number of iterations
  """
  x = inita
   i = 0
  while i < maxIt:
      xn = x - ( f(x) / fp(x) )
      if abs(xn - x) < tol:
          break
      x = xn
      i+=1
  return x

init_guess = 5
tolerance = 0.00001
```

```
timea = time.time()
print("root value: ",Nth_root_clean(f,fp,init_guess,tolerance))
timeb = time.time()
print("time taken: ",timeb-timea)
```

[Graph Code] A graph comparing the zero estimations of the three numerical methods:

```
from secant import Secant
from bisection import Bisection
from newton import Newton
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
fig, ax = plt.subplots()

def f(x):
   return x**2 - 7

def fp(x):
   return 2*x

def myplot(x,y,y2,color='tab:black',label="yes"):
   plt.xlabel("Function iteration number")
   # ax.set_yscale('log')
   ax2 = ax
   ax2.set_ylabel('Zero estimation', color=color)
   ax2.plot(x, y2, color=color,label=label)
   fig.tight_layout()

x, y, y2 = Bisection(f, 2, 6, 1e-10,maxIt=1000,test=True)
myplot(x, y, y2,color='tab:orange',label="Bisection")

x, y, y2 = Secant(f, 3, 6, 1e-10,maxIt=1000,test=True)
myplot(x, y, y2,color='black',label="Secant")

x, y, y2 = Newton(f, fp, 3, 1e-10,maxIt=1000,test=True)
myplot(x, y, y2,color='tab:red',label="Newton")

plt.legend(loc='upper right')
plt.show()
```