**MAT/CSC 433: Numerical Analysis**

**Project 2: Nonlinear Equations**

Anh N. Ngo

**1. Introduction**

Consider the solution of the equation:

*sin(x) = 1*

From our experiences with Precalculus, we would answer x = . However, I do believe that most of us will not be able to answer this question without remembering exactly what we were taught or without the support of Google. The answer to this question does not look too bad with a fraction of π, but what if we find the solution to this equation?

*sin(x) = 0.5*

It does not seem practical anymore to calculate it manually without the support of a calculator. To obtain a numerical solution of *sin(x) = 0.5*, we may produce a sequence of known values that converge to the equation's solution. There are multiple ways in which to produce such a sequence. This report will help us answer not only *sin(x) = 0.5* but generally

*sin(x) = y*.

The following section will introduce the *bisection* algorithm, which successively captures the solution within intervals of decreasing length. Section 3 presents the secant method, which sequentially estimates the solution using a secant line through pairs of points near the solution point. Section 4 introduces Newton's algorithm, which replaces the slope of the secant line with the derivative of *y = cos(x)* at a point. Section 5 presents a fixed-point algorithm that employs a recursively defined sequence xn+1 = g(xn) where the solution is a function's fixed point of the function g. In the final section, we compare the performance and complexity of the four algorithms.

**2. Bisection Algorithm**

The domain of sin(x) is all real numbers. Also, the value of sin(x), depending on the point on the circle, can go to a maximum of 1 at x = 90 degrees and a minimum of -1 at x = 270 degrees. So, the range of sin(x) is -1 to 1.

At the first approximation, let's try the midpoint x = 0. We find that sin(0) = 0, which is less than 1. Thus, by the same argument, we know that the solution lies in the interval [0, 1]. We may repeat the process, successively bisecting the interval [a, b] and identifying half which contains the solution, until the bounds of the interval, a and b, are sufficiently close together. The generalized algorithm is presented below.

Diagram

Description generated with very high confidence

The code below implements the algorithm in Octave. The user enters the value of y. The starting interval will be the range of . First, we will calculate the mid-point (mid), which initially is = 0. If sin(mid) > y, then our new interval is . However, if sin(mid) < y, our new interval is . To begin the iterative process, we test the relative difference between a and b against epsilon's allowable tolerance. We use the relative difference, rather than the absolute difference, to account for the magnitude of the solution. We will keep squeezing the interval until a and b no longer tolerate epsilon, which is also when we return mid – the value of x we have been looking for.

Using the code to solve the equation sin(x) = 0.5 with epsilon 0.00001, we obtain the solution x = 0.5236 after 19 iterations, which agrees with my calculator solution and Octave's built-in square root function. The efficiency of this algorithm will be discussed in the conclusion section.

% Using Bisection method to compute sin(x) = an enterred number

display("Calculate sin(x) = y");

y = input("Provide a number for this algorithm: ");

% enter tolerance

e = input("Enter your tolerance: ");

% create upper and lower bound that x must be in

if (-1 <= y && y <= 1)

a = -pi/2;

b = pi/2;

j = 0

while abs((b-a)/b) > e

j = j + 1

mid = (a+b)/2;

if sin(mid) < y

a = mid; % updates the lower bound to the midpoint

mid = (a + mid)/2;

else

b = mid; % updates the upper bound

mid = (b + mid)/2;

endif

endwhile

display(mid)

else

display("Only input between -1 and 1")

endif

**3. Secant Algorithm**

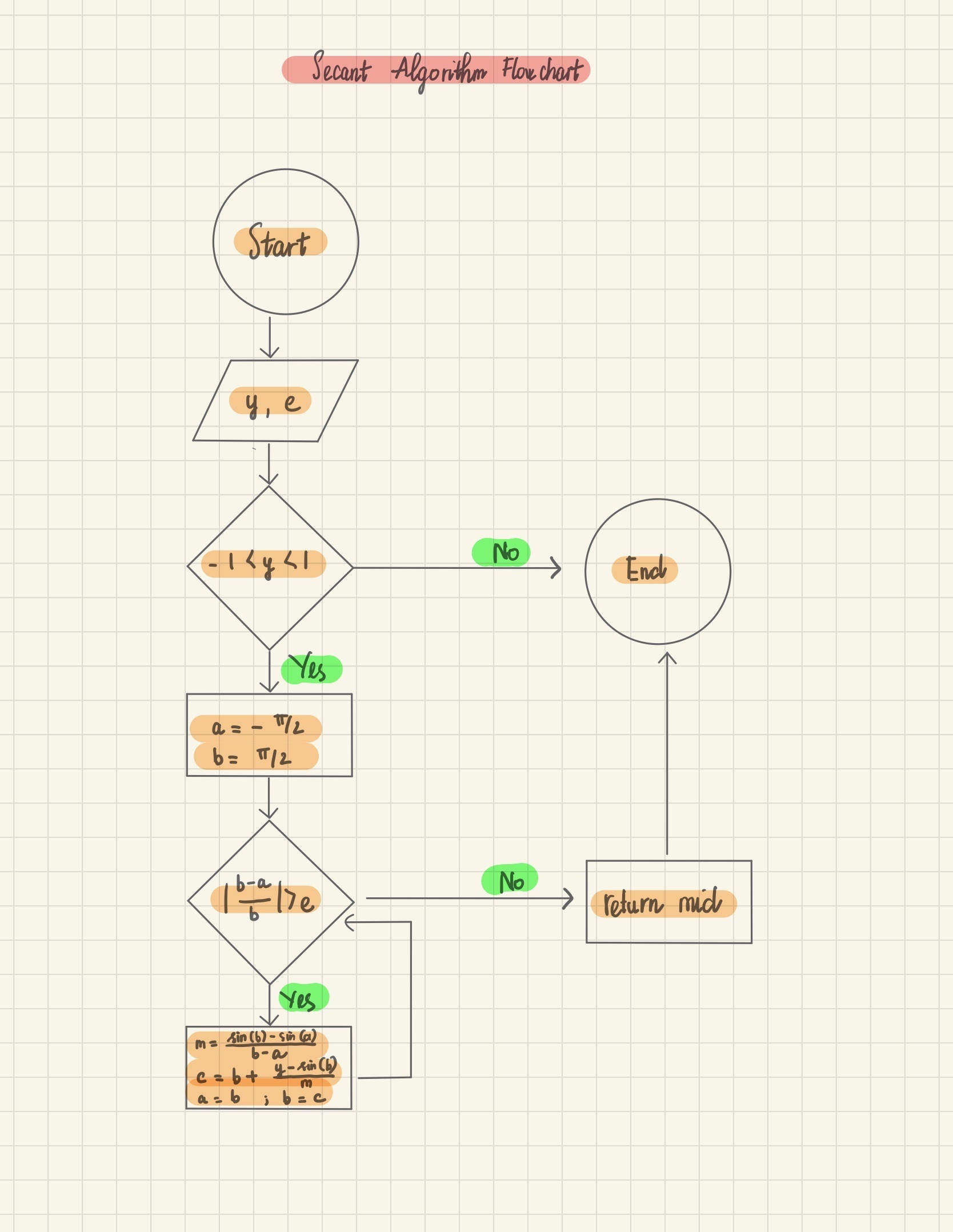
The [secant method](https://en.wikipedia.org/wiki/Secant_method) is very similar to the bisection method. Instead of dividing each interval by choosing the midpoint, the secant method divides each interval by connecting the endpoints. The secant method always converges to a root of  provided that is continuous on [a,b] and .

Let f(x) be a continuous function on a closed interval  such that . A solution of the equation  is guaranteed by the [Intermediate Value Theorem](https://en.wikipedia.org/wiki/Intermediate_value_theorem), which has been introduced in Calculus 2. Consider the line connecting the endpoint values  and . The line connecting these two points is called the secant line and is given by the formula

The point where the secant line crosses the x-axis is:

Which we solve for x:

Now, let's jump straight to build our algorithm. The generalized algorithm is presented below.



The code below implements the algorithm in Octave. The user enters the value of y. We need to make sure the entered value needs to be in the interval [-1,1], which is the range of sin(x). The starting domain will be the range of .

Compute f(x0) where the secant line gives x0

Which in this case is:

Then, determine the next subinterval :

If , then let  be the next interval with  and .

If , then let  be the next interval with  and .

Repeat those two steps until the interval  reaches some predetermined length, which is epsilon we have mentioned about.

Return the value , the x-intercept of the  subinterval. In the coding part, we will use c as a temporary variable to help with the updating process of a and b.

% compute sin(x) = an enterred number

display("Calculate sin(x) = y");

y = input("Provide a number for this algorithm: ");

% enter tolerance

e = input("Enter your tolerance: ");

% create upper and lower bound that x must be in

if (-1 <= y && y <= 1)

a = -pi/2;

b = pi/2;

j = 0

while abs((b-a)/b) > e

j = j + 1

m = (sin(b)-sin(a))/(b-a);

c = b + (y-sin(b))/m;

a = b;

b = c;

endwhile

display(b)

else

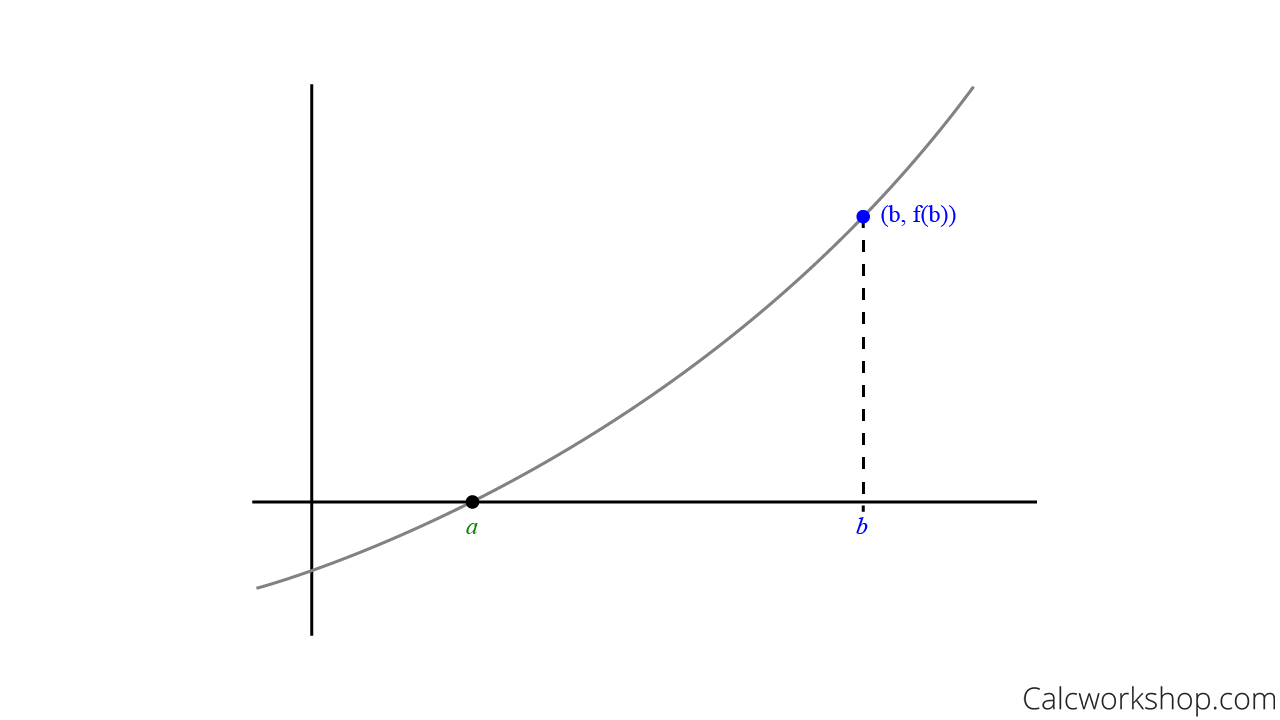
display("Only input between -1 and 1")

endif

Using the code to solve the equation sin(x) = 0.5 with epsilon 0.00001, we obtain the solution x = 0.5236 after 7 iterations, which agrees with my calculator solution and Octave's built-in square root function. The efficiency of this algorithm will be discussed in the conclusion section.

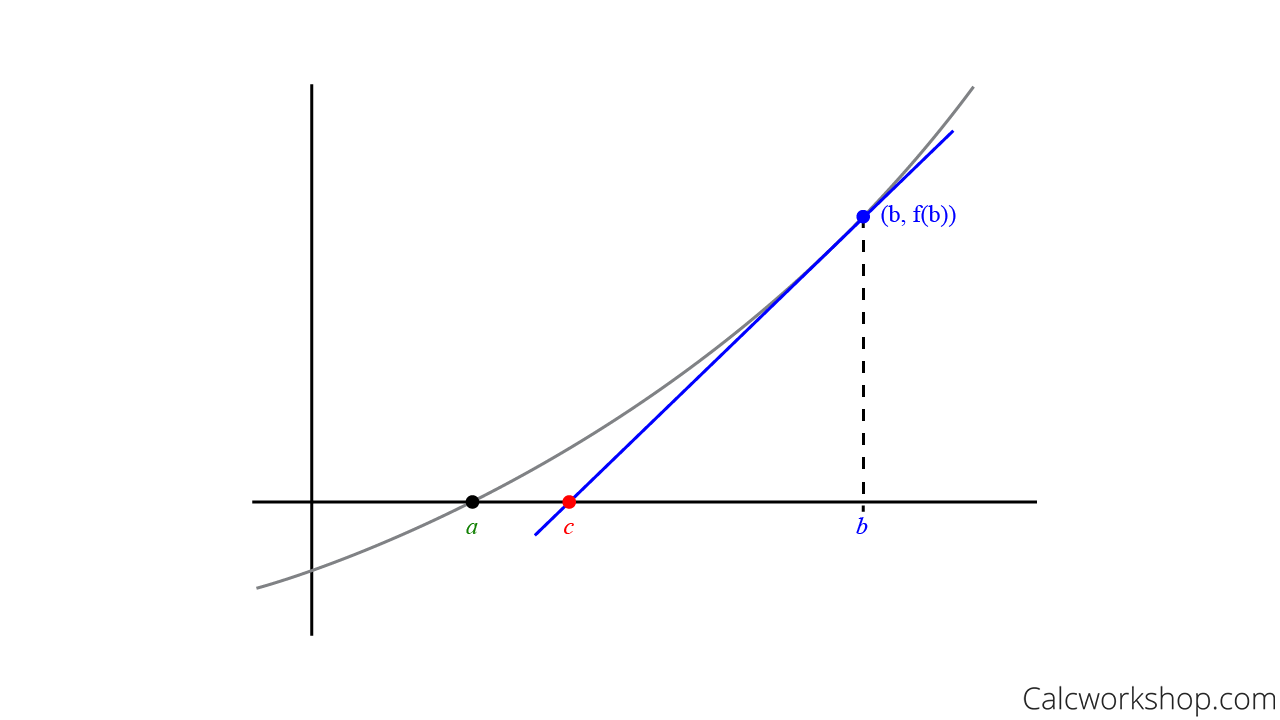
**4. Newton's Algorithm**

The idea is to start with an initial guess that is reasonably close to the true root (solution) and then use the tangent line to obtain another x-intercept that is even better than our initial guess or starting point. Let's look at this conceptually to make sense of what is happening. Assume we want to find the root (i.e., x-intercept) for . This means we want to find a in the picture below.



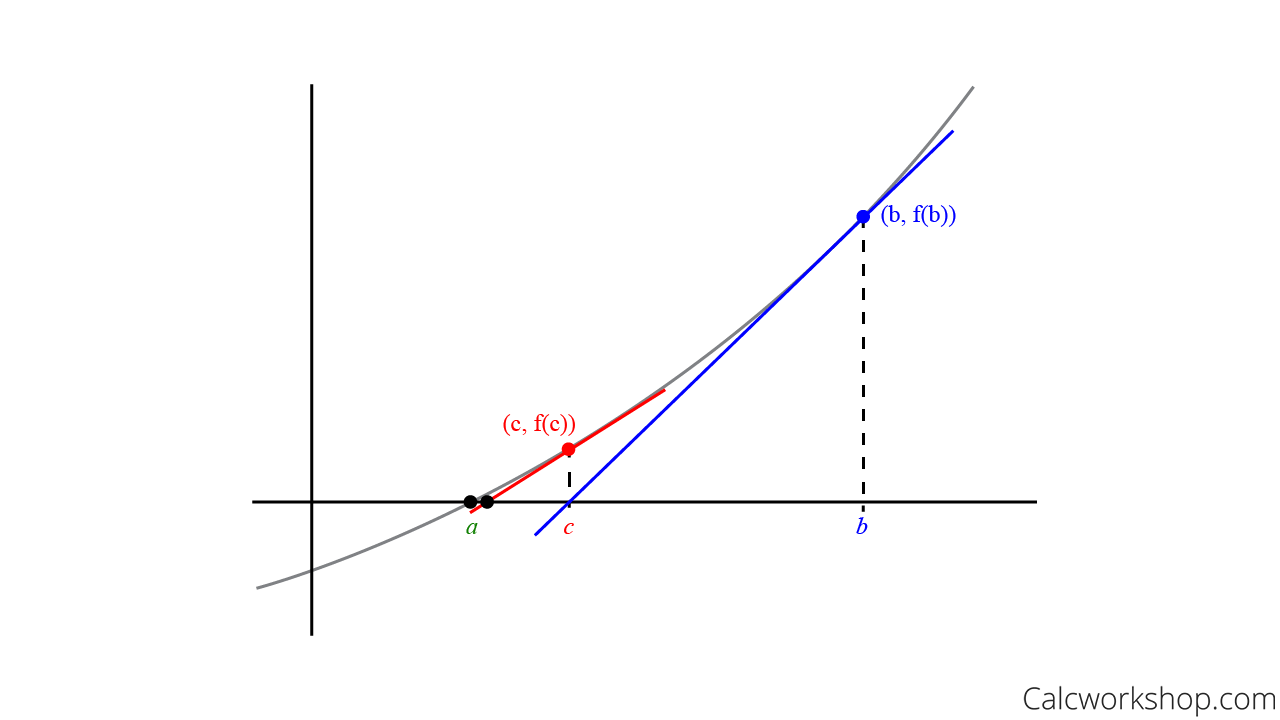
**Step One: Approximate Point B**

So, we begin with an initial guess that is relatively close to point a, which is indicated as point b, and substitute b into and see if it equals zero. If it does equal, we're done. However, more than likely, this will not be the case, and we will need to try for a better guess.



**Step Two: Drawing Tangent Lines For B**

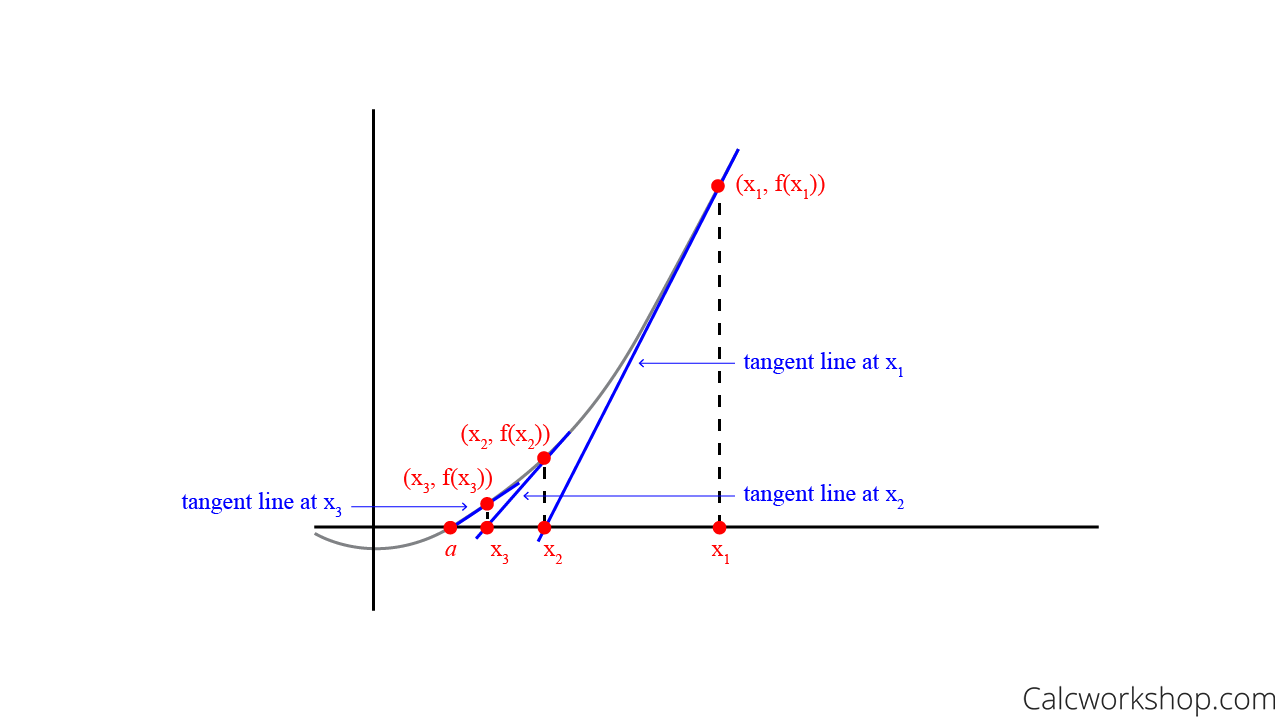
By using our knowledge of linear approximation. If we can create a tangent line to the graph at point b, then we can find another point even closer to a that also intersects the x-axis, at point c, as seen below.



**Step Three: Find the Tangent Line at C**

And we keep using this process until we can find two successive values that agree to the [desired decimal place](https://calcworkshop.com/decimal/decimal-place-value/). In other words, until we finally converge on the value of a.

This iterative process is called Newton's Method!



Newton's Method — Calculus

* **Newton's Method Formula**

And to help with our calculations, we can use the following formula:

If the nth approximation is  and , then the next approximation is given by:

In our case *sin(x) = y*, we have:

Now, let's jump straight to build our algorithm. The generalized algorithm is presented below.

Diagram

Description generated with very high confidence

The code below implements the algorithm in Octave. The user enters the value of y. We need to make sure the entered value needs to be in the interval [-1,1], which is the range of sin(x). The starting domain will be the range of .

Compute f(x0) where x0 is given by Newton's algorithm:

Which in this case is:

The initial guess for all cases is , a point inside the interval [-1,1]. The variable b acts as . The loop will continue replacing the values of a and b until the interval  reaches some predetermined length, which is epsilon in the code.

Return the value , the x-intercept of the  subinterval. Then, we are good to go!

% compute sin(x) = an enterred number

display("Calculate sin(x) = y");

y = input("Provide a number for this algorithm: ");

% enter tolerance

e = input("Enter your tolerance: ");

% create upper and lower bound that y must be in

if (-1 <= y && y <= 1)

a = 1;

m = cos(y);

b = a + (y-sin(a))/m;

j = 0

while (abs(b-a)/b) > e

j = j + 1

a = b;

b = a + (y-sin(a))/m;

endwhile

display(b)

else

display("Only input between -1 and 1")

endif

However, this algorithm works well if the input y is positive. If the input is a negative number, it will result incorrectly. Using the code to solve the equation sin(x) = 0.5 with epsilon 0.00001, we obtain the solution x = 0.5236 after four iterations, which agrees with my calculator solution and Octave's built-in square root function. The efficiency of this algorithm will be discussed in the conclusion section.

**5. Fixed-point Algorithm**

The idea of the fixed-point algorithm is first to reformulate an equation to an equivalent fixed-point problem:

and then to use the iteration: with an initial guess chosen, compute a sequence in the hope that → α. There are infinitely many ways to introduce an equivalent fixed-point problem for a given equation; e.g., for any function with the property , we can take . The resulting iteration method may or may not converge in any case.

In our case, let's first deal with .

If we add x to both sides, we will have:

Move to another side, we have:

First, assume , we have

, we have

By following the loop, we will stop at .

Now, let's jump straight to build our algorithm. The generalized algorithm is presented below.

Diagram

Description generated with very high confidence

The code below implements the algorithm in Octave. The user enters the value of y. We need to make sure the entered value needs to be in the interval [-1,1], which is the range of sin(x). The starting domain will be the range of . Compute f(x) where the Fixed-point algorithm gives x:

The loop will continue replacing the values of a and b until the interval  reaches some predetermined length, which is epsilon in the code.

Return the value , the x-intercept of the  subinterval. Then, we are good to go!

% compute sin(x) = an entered number

display("Calculate sin(x) = a");

y = input("Provide a number for this algorithm: ");

% enter tolerance

e = input("Enter your tolerance: ");

% create upper and lower bound that y must be in

if (-1 <= y && y <= 1)

a = -pi/2;

b = pi/2;

j = 0;

while abs((b-a)/b) > e

m = y + b - sin(b);

b = m;

m1 = y + m - sin(m);

a = m1;

j = j+2

endwhile

display(b)

else

display("Only input between -1 and 1")

endif

Using the code to solve the equation sin(x) = 0.5 with epsilon 0.00001, we obtain the solution x = 0.5236 after 16 iterations, which agrees with my calculator solution and Octave's built-in square root function. The efficiency of this algorithm will be discussed in the conclusion section.

**6. Conclusion**

As I have mentioned in section 4 – Newton's Algorithm, Newton's algorithm we built works perfectly with positive input y. However, with negative inputs, the algorithm fails to return correctly. In the current time limit that this project is written, I am not able to find out the reason behind it. However, I will continue to work on this and do more research for better understanding.

All algorithm Big O is O(n) in terms of performance since all of them utilize a while loop. However, the number of loops each algorithm takes varies depending on the input y and epsilon value. Here is the table that reflects the discrepancy:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| sin(x) | Epsilon | Bisection | Secant | Newton's | Fixed point |
| 0.25 | 0.01 | 11 | 4 | 3 | 8 |
| 0.001 | 14 | 5 | 3 | 8 |
| 0.0001 | 17 | 6 | 3 | 10 |
| 0.00001 | 21 | 6 | 4 | 12 |
| 0.5 | 0.01 | 10 | 5 | 2 | 8 |
| 0.001 | 13 | 6 | 3 | 10 |
| 0.0001 | 16 | 6 | 3 | 14 |
| 0.00001 | 20 | 7 | 4 | 16 |
| 0.75 | 0.01 | 9 | 6 | 2 | 12 |
| 0.001 | 12 | 7 | 3 | 16 |
| 0.0001 | 16 | 7 | 4 | 20 |
| 0.00001 | 19 | 8 | 5 | 24 |

Using Python and its visualization libraries, we have the following charts demonstrating the shown table:

Chart

Description generated with high confidence

Clearly, Newton's algorithm, regardless of its constrain of dealing with negative inputs, is the top scorer in terms of performance, followed by the Secant algorithm. In addition, when the value of y is smaller, the Fixed-point algorithm performs better than the Bisection method and ranks third. However, as the input y has a more significant value, the Fixed-point algorithm loses momentum and rates at the bottom. It also means that with more significant y, the Bisection algorithm can perform better.

Git Repository to the Charts: https://github.com/ahnngo/Project-2-Nonlinear-Equations

Work Cited

"Secant Method - Mathematical Python." *Mathematical Python*, personal.math.ubc.ca/%7Epwalls/math-python/roots-optimization/secant. Accessed 11 Mar. 2022.

"Fixed Point Iteration." *The University of Iowa*, homepage.divms.uiowa.edu/~whan/3800.d/S3-4.pdf. Accessed 10 Mar. 2022.

"Newton's Method." *Calc Workshop*, 21 Feb. 2021, calcworkshop.com/derivatives/newtons-method.