Elementos do grupo:

* Diogo Fonseca nº 79858
* Tomás Teodoro nº 80044
* Diogo Silva nº 79828
* Tiago Granja nº 79845

Resumo

O programa é capaz de calcular o zero de qualquer função com um máximo de 100 algarismos de precisão, dado que no intervalo [x0, x1] ela tem **um e um só** zero e é contínua. Para cada método do cálculo do zero (bissetriz, newton, secante) ela gera também o número de iterações necessários para cumprir as especificações.

Instruções de Utilização

Nota: pressione *Ctrl+C* a qualquer momento durante a execução para parar o programa.

* **precision:**  – O número (inteiro) de algarismos significativos a ser usado internamente pelo programa. ()

Exemplo: ()

Exemplo: ()

* **Expression:**  – A função matemática à qual se pretende encontrar o zero.
  + **Variáveis**: ‘x’.
  + **Operadores aritméticos:** ‘+’, ‘-‘, ‘/’, ‘\*’.
  + **Funções Trigonométricas:** ’’, ‘’, ‘’, ‘’, ‘’, ‘’.
  + **Expoentes:** ‘x\*\*y’ (ou) ‘x^y’ (x levantado a y).
  + **Logaritmo:** ‘’ (logaritmo de x, base n).
  + **Constantes:** ‘E’ (número de Euler), ‘pi’.
  + **Raízes:** ‘’ (para raiz quadrada ou elevar um número a 1/raiz)
  + **Fatoriais:** ‘’ (AVISO: O método de Newton é extremamente ineficiente para fatoriais)

Exemplo:

Exemplo:



* **Initial Range:**  – O número real (incluí constantes e aritmética) que minora o zero a encontrar.
* **Final Range:**  – O número real (incluí constantes e aritmética) que majora o zero a encontrar. ()
* **Error Margin:** – O número real (incluí constantes e aritmética) que define o erro do zero a calcular () Nota: se precisão > margem de erro, então os valores obtidos podem não estar precisos após a unidade da margem de erro. (é também possível definir esta variável a 0, mas o funcionamento pode ser irregular)
* **Max Iterations**: – O número que representa o máximo de iterações que o programa vai calcular até parar abruptamente, mesmo que . (é também possível definir esta variável a ‘oo’ (infinito), mas o funcionamento pode ser irregular)

Execução

Cada ficheiro aqui posto deve ser colocado no seu ficheiro particular com o mesmo nome que a sua classe, este projeto usa também o auxílio da biblioteca matemática SymPy (<https://www.sympy.org>). Para a sua execução é necessário instalar a mesma para o seu funcionamento correto. Com python instalado no computador, isto pode ser feito através do terminal com o comando “pip3 install sympy”.

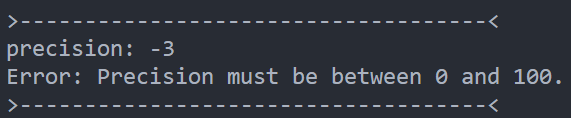
Observações

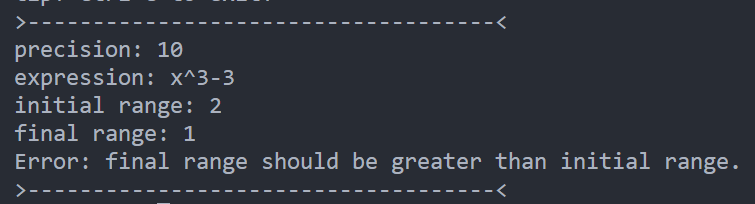
O ponto usado como ponto “guia” que o método de Newton utiliza é simplesmente o ponto médio entre o intervalo especificado, se desejado, é possível definir para controlar o ponto “guia” do método de Newton (fazer isto vai causar os outros dois métodos a darem erro).

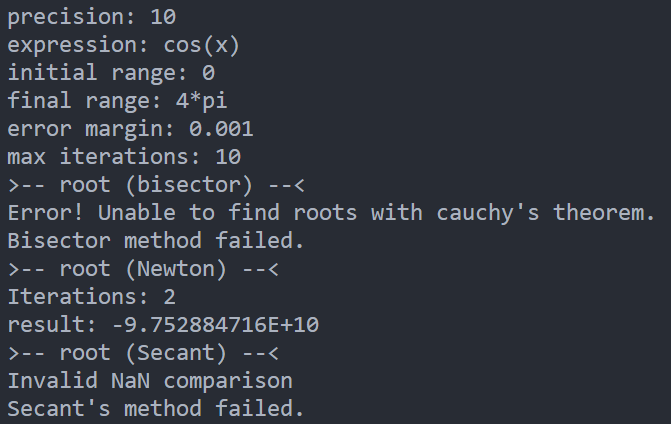
Os pontos utilizados para o método da secante são o e do intervalo, se estes forem muito próximos de outro zero, é possível que este método dê esse zero, para um maior controlo, é possível definir e só como pontos do método da secante, ignorando o output dos outros métodos.

O programa está preparado para obter qualquer tipo de input, mesmo que este viole as especificações previamente mencionadas, gerando uma mensagem de erro apropriada, mas continuando a execução.

Exemplo:

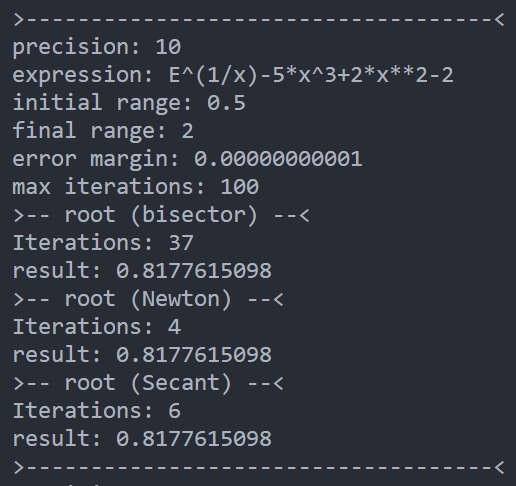
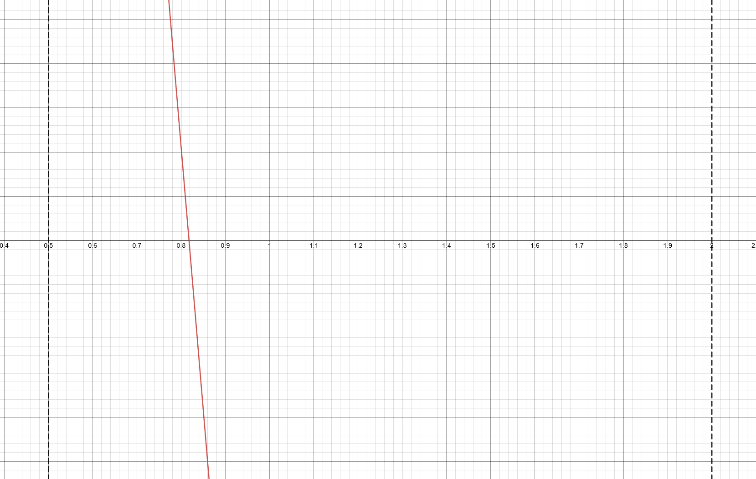


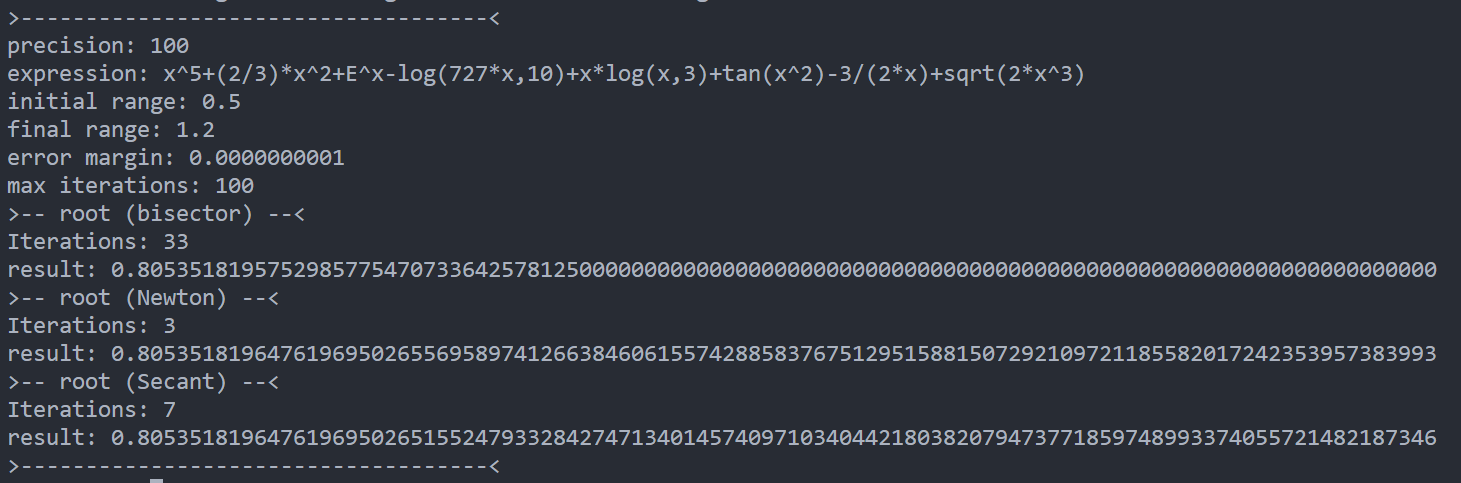
 Exemplo:

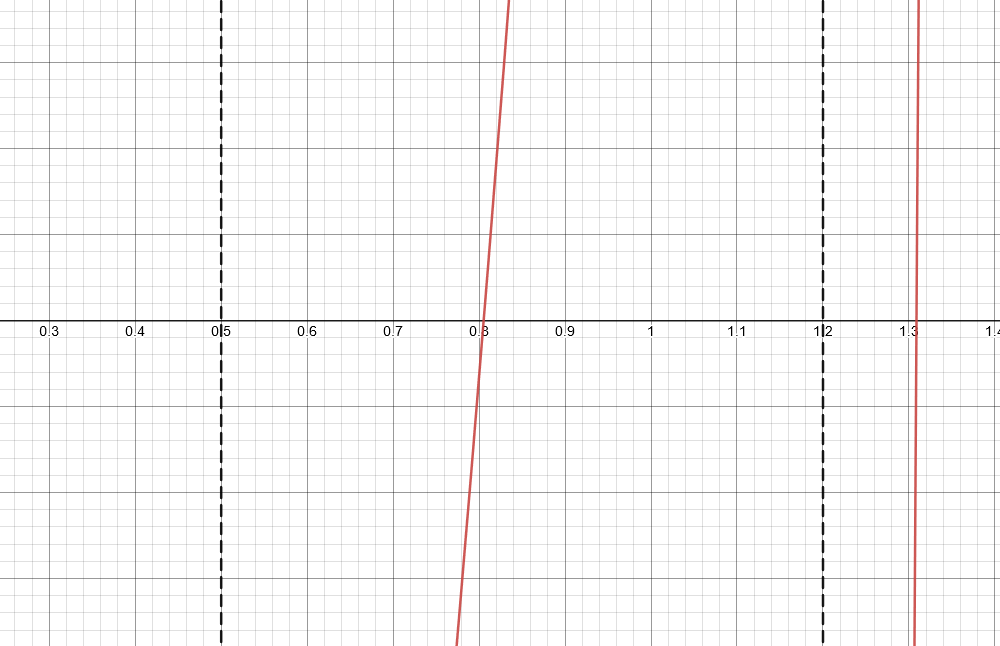
Exemplo:

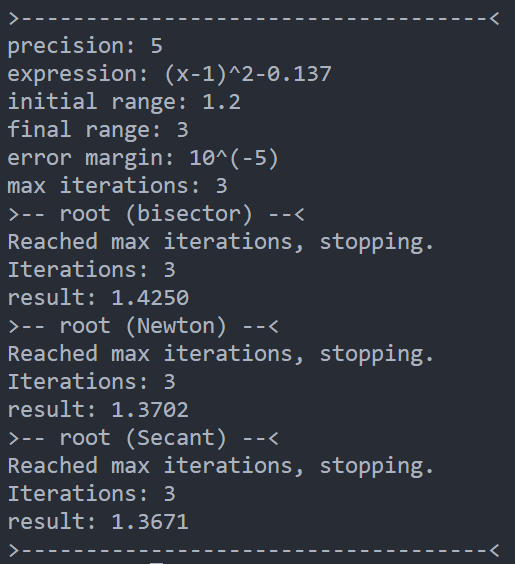
Explicação: A função apresentada em conjunto com o intervalo dados **estão fora do que o programa é capaz de calcular**: O método da bissetriz não pode encontrar zeros, devido à constante de sinais entre os pontos que foram selecionados (todos os pontos são positivos neste caso), o método de newton deu “lixo” (um valor incorreto), já que (simplesmente usando o método de Newton) este não tem forma de saber que existem vários zeros e faz contas erradas (para além de que ele tenta calcular a derivada num ponto em que a derivada dá 0), e o mesmo acontece para o método da secante.

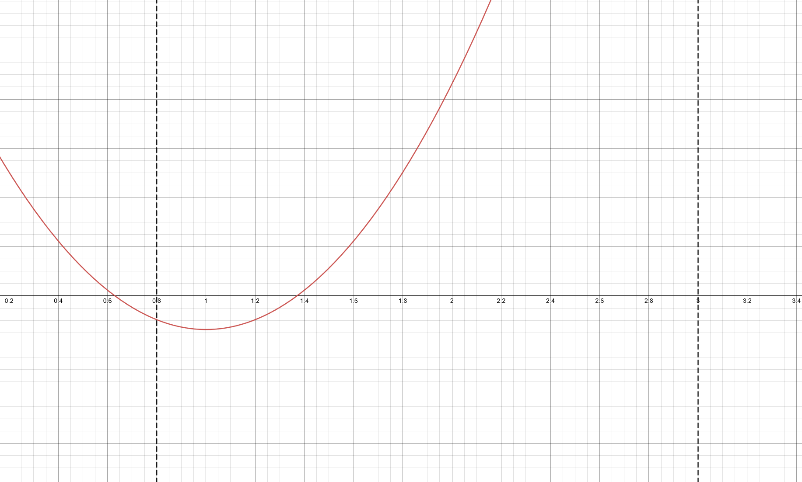
Screenshots



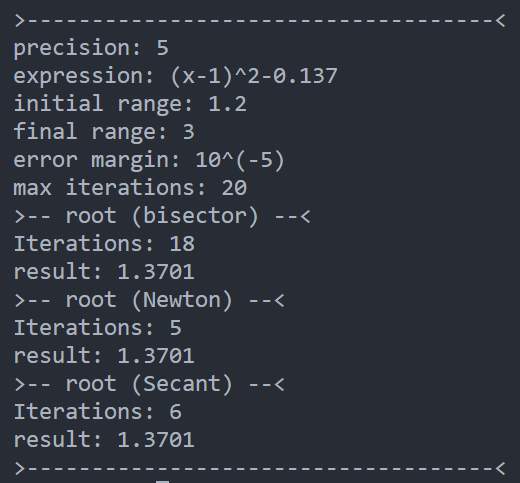


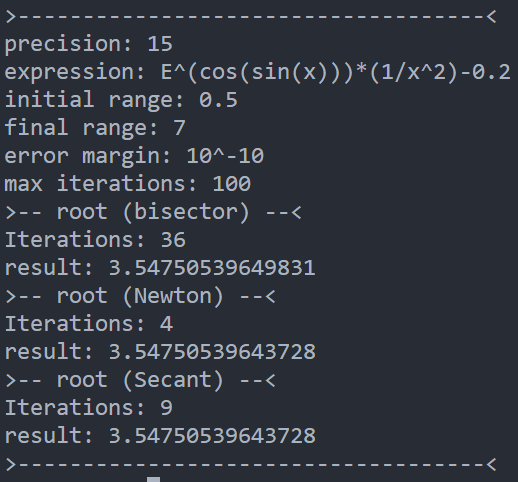


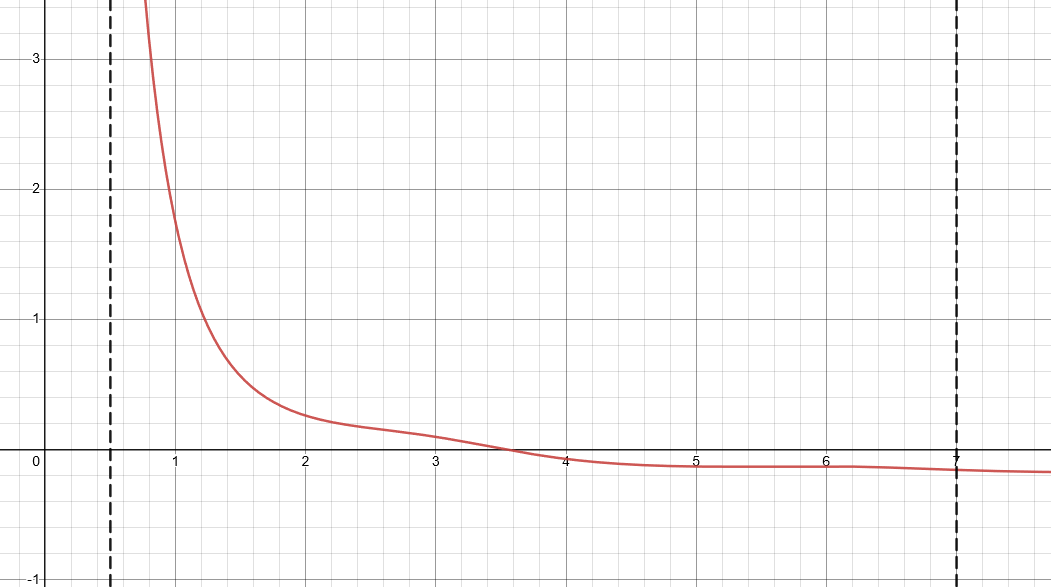




(cont.)







Código

main.py

import bisector

import newton

import secant

import math\_input

from sympy import \*

exit = false

print("tip: Ctrl+C to exit!")

while exit == false:

    print(">------------------------------------<")

    try:  *# ONLY WORKS IF THERE IS ONE AND ONLY ONE ROOT INSIDE RANGE*

        precision = math\_input.math\_input.get\_precision()  *# 0 < x <= 100*

        expression = math\_input.math\_input.get\_expression(precision)

        range0 = math\_input.math\_input.get\_range0(precision)

        range1 = math\_input.math\_input.get\_range1(range0, precision)

        error\_margin = math\_input.math\_input.get\_error\_margin(precision)  *# can be 0 (but should set max\_iterations)*

        max\_iterations = math\_input.math\_input.get\_max\_iterations(precision)  *# can be oo (but should set error\_margin)*

        try:

            print(">-- root (bisector) --<")

            result\_bisector = bisector.bisector.calc\_root(expression, range0, range1, error\_margin, max\_iterations)

            print(f"result: {round(result\_bisector, precision)}")

        except Exception as e:

            print(str(e))

            print("Bisector method failed.")

        try:

            print(">-- root (Newton) --<")

            result\_newton = newton.newton.calc\_root( expression, (range0 + range1) / 2, error\_margin, max\_iterations)  *# point is (range0 + range1) / 2*

            print(f"result: {round(result\_newton, precision)}")

        except:

            print("Newton's method failed.")

        try:

            print(">-- root (Secant) --<")

            result\_newton = secant.secant.calc\_root(expression, range0, range1, error\_margin, max\_iterations)

            print(f"result: {round(result\_newton, precision)}")

        except Exception as e:

            print(str(e))

            print("Secant's method failed.")

    except KeyboardInterrupt:

        exit = true

    except Exception as e:

        print(str(e))

print("")

print("Exiting...")

math\_input.py

from sympy import \*

class math\_input:

    @staticmethod

    def get\_precision():

        this\_input = input("precision: ")

        try:

            try:

                this\_input = int(this\_input)

            except:

                raise Exception("Error: precision must be an integer.")

            if not ask(Q.integer(this\_input)):

                raise Exception("Error: precision must be an integer.")

            if this\_input < 0 or this\_input > 100:

                raise Exception("Error: Precision must be between 0 and 100.")

            return int(this\_input)

        except Exception as e:

            raise e

    @staticmethod

    def get\_expression(*precision*):

        this\_input = input("expression: ")

        try:

            return N(this\_input, *precision*)

        except:

            raise Exception("Error: Invalid expression: '" + this\_input + "'.")

    @staticmethod

    def get\_range0(*precision*):

        this\_input = input("initial range: ")

        try:

            this\_input = N(this\_input, *precision*)

            if not ask(Q.real(this\_input)):

                raise Exception()

            return this\_input

        except:

            raise Exception("Error: Invalid range.")

    @staticmethod

    def get\_range1(*range0*, *precision*):

        this\_input = input("final range: ")

        try:

            this\_input = N(this\_input, *precision*)

        except:

            raise Exception("Error: Invalid range.")

        try:

            if not ask(Q.real(this\_input)):

                raise Exception("Error: Invalid range.")

            elif *range0* > this\_input:

                raise Exception(

                    "Error: final range should be greater than initial range."

                )

            return this\_input

        except Exception as e:

            raise e

    @staticmethod

    def get\_error\_margin(*precision*):

        this\_input = input("error margin: ")

        try:

            this\_input = N(this\_input, *precision*)

            if not ask(Q.real(this\_input)):

                raise Exception("Error: Invalid error margin.")

            if ask(Q.negative(this\_input)):

                raise Exception("Error: Margin should be positive.")

            return this\_input

        except Exception as e:

            raise e

    @staticmethod

    def get\_max\_iterations(*precision*):

        this\_input = input("max iterations: ")

        try:

            this\_input = N(this\_input, *precision*)

            if not ask(Q.real(this\_input)) and not ask(Q.positive\_infinite(this\_input)):

                raise Exception("Error: Invalid maximum iterations.")

            if ask(Q.negative(this\_input)):

                raise Exception("Error: iterations should be positive.")

            return this\_input

        except Exception as e:

            raise e

secant.py

from sympy import \*

class secant:

    @staticmethod

    def calc\_root(*expression*, *x0*, *x1*, *error\_margin*, *iterations*):

        startIterations = *iterations*

        new\_x = *x1* - (*expression*.subs(symbols("x"), *x1*) \* (*x1* - *x0*)) / (*expression*.subs(symbols("x"), *x1*) - *expression*.subs(symbols("x"), *x0*))

        while abs(new\_x - *x1*) > *error\_margin* and *iterations* > 0:

            if *expression*.subs(symbols("x"), new\_x) == 0:

                print(f"Iterations: {int(startIterations - *iterations*)}")

                return new\_x

*x0* = *x1*

*x1* = new\_x

*iterations* -= 1

            new\_x = *x1* - (*expression*.subs(symbols("x"), *x1*) \* (*x1* - *x0*)) / (*expression*.subs(symbols("x"), *x1*) - *expression*.subs(symbols("x"), *x0*))

        if *iterations* <= 0:

            print("Reached max iterations, stopping.")

        print(f"Iterations: {int(startIterations - *iterations*)}")

        return new\_x

newton.py

from sympy import \*

class newton:

    @staticmethod

    def calc\_root(*expression*, *x*, *error\_margin*, *iterations*):

        startIterations = *iterations*

        derivative = diff(*expression*, symbols("x"))

        new\_x = *x* - (*expression*.subs(symbols("x"), *x*) / derivative.subs(symbols("x"), *x*))

        while(abs(new\_x - *x*) > *error\_margin* and *iterations* > 0):

*x* = new\_x

            if (*expression*.subs(symbols("x"), *x*) == 0):

                print(f"Iterations: {int(startIterations - *iterations*)}")

                return *x*

            elif (derivative.subs(symbols("x"), *x*) == 0):

                print("Derivative was 0! Inflexion point found. Returning current value.")

                print(f"Iterations: {int(startIterations - *iterations*)}")

                return *x*

*iterations* -= 1

            new\_x = *x* - (*expression*.subs(symbols("x"), *x*) / derivative.subs(symbols("x"), *x*))

        if (*iterations* <= 0):

            print("Reached max iterations, stopping.")

        print(f"Iterations: {int(startIterations - *iterations*)}")

        return new\_x

bisector.py

from sympy import \*

class bisector:

    @staticmethod

    def get\_half(*x0*, *x1*):

        return (*x0* + *x1*) / 2

    @staticmethod

    def calc\_root(*expression*, *x0*, *x1*, *error\_margin*, *iterations*):

        startIterations = *iterations*

        half = bisector.get\_half(*x0*, *x1*)

        while(abs(*x1* - *x0*) > *error\_margin* and *iterations* > 0):

            half = bisector.get\_half(*x0*, *x1*)

*iterations* -= 1

            if *expression*.subs(symbols("x"), half) == 0:

                print(f"Iterations: {int(startIterations - *iterations*)}")

                return half

            elif *expression*.subs(symbols("x"), half) \* *expression*.subs(symbols("x"), *x0*) < 0:

*x1* = half

            elif *expression*.subs(symbols("x"), half) \* *expression*.subs(symbols("x"), *x1*) < 0:

*x0* = half

            else:

                if *expression*.subs(symbols("x"), *x0*) == 0:

                    print(f"Iterations: {int(startIterations - *iterations*)}")

                    return *x0*

                elif  *expression*.subs(symbols("x"), *x1*) == 0:

                    print(f"Iterations: {int(startIterations - *iterations*)}")

                    return *x1*

                raise Exception("Error! Unable to find roots with cauchy's theorem.")

        if (*iterations* <= 0):

            print("Reached max iterations, stopping.")

        print(f"Iterations: {int(startIterations - *iterations*)}")

        return half