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 [x_0, x_1] 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: d** – O número (inteiro) de algarismos significativos a ser usado internamente pelo programa. ($0 < x \leq 100$)

  Exemplo: ($x = 100$)

```
result: 1.4142135623730950488016887242096980785696718753769480731766797379907324784621070388503875343276415 73
```

  Exemplo: ($x = 5$)

```
result: 1.4142
```

- **Expression: $f(x)$** – A função matemática à qual se pretende encontrar o zero.
  - **Variáveis**: 'x'.
  - **Operadores aritméticos:** '+', '-', '/', '*'.
  - **Funções Trigonométricas:** '$cos(x)$', '$sin(x)$', '$tan(x)$', '$cot(x)$', '$sec(x)$', '$csc(x)$'.
  - **Expoentes:** 'x**y' (ou) 'x^y' (x levantado a y).
  - **Logaritmo:** '$log(x, n)$' (logaritmo de x, base n).
  - **Constantes:** 'E' (número de Euler), 'pi'.
  - **Raízes:** '$sqrt(x)$' (para raiz quadrada ou elevar um número a 1/raiz)
  - **Fatoriais:** 'x!' (AVISO: O método de Newton é extremamente ineficiente para fatoriais)

  Exemplo: $e^{\frac{1}{x}} - 5x^3 + 2x^2 - 2$

```
expression: E^(1/x)-5*x^3+2*x**2-2
```

  Exemplo: $x^5 + \frac{2}{3}x^2 + e^x - log_{10}(727x) + xlog_3(x) + \tan(x^2) - \frac{3}{2x} + \sqrt{2x^3}$

```
expression: x^5+(2/3)*x^2+E^x-log(727*x,10)+x*log(x,3)+tan(x^2)-3/(2*x)+sqrt(2*x^3)
```

- **Initial Range:** x_0 – O número real (incluí constantes e aritmética) que minora o zero a encontrar.
- **Final Range:** x_1 – O número real (incluí constantes e aritmética) que majora o zero a encontrar. ($x_0 \leq x_1$)
- **Error Margin:** ε – O número real (incluí constantes e aritmética) que define o erro do zero a calcular ($f(x) = 0 \Rightarrow x - x_{obtido} < \varepsilon$) 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**: n – O número que representa o máximo de iterações que o programa vai calcular até parar abruptamente, mesmo que $x - x_{obtido} > \varepsilon$. (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 $x_0 = x_1$ 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 x_0 e x_1 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 x_0 e x_1 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:

```
>-----------------------------------<
precision: -3
Error: Precision must be between 0 and 100.
>-----------------------------------<
```

Exemplo:

```
>-----------------------------------<
precision: 10
expression: x^3-3
initial range: 2
final range: 1
Error: final range should be greater than initial range.
>-----------------------------------<
```

Exemplo:

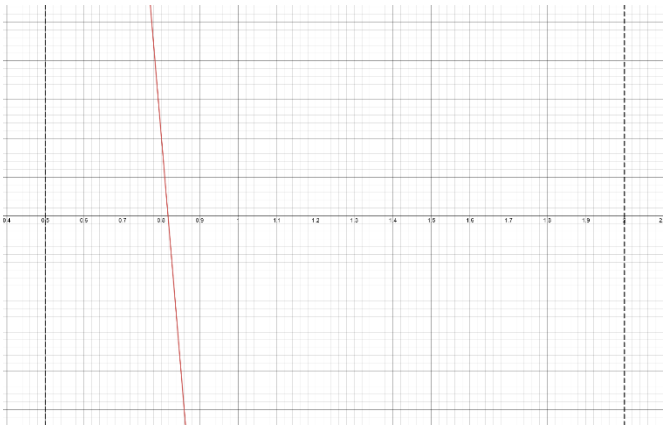
```
precision: 10
expression: cos(x)
initial range: 0
final range: 4*pi
error margin: 0.001
max iterations: 10
>-- root (bisector) --<
Error! Unable to find roots with cauchy's theorem.
Bisector method failed.
>-- root (Newton) --<
Iterations: 2
result: -9.752884716E+10
>-- root (Secant) --<
Invalid NaN comparison
Secant's method failed.
```

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

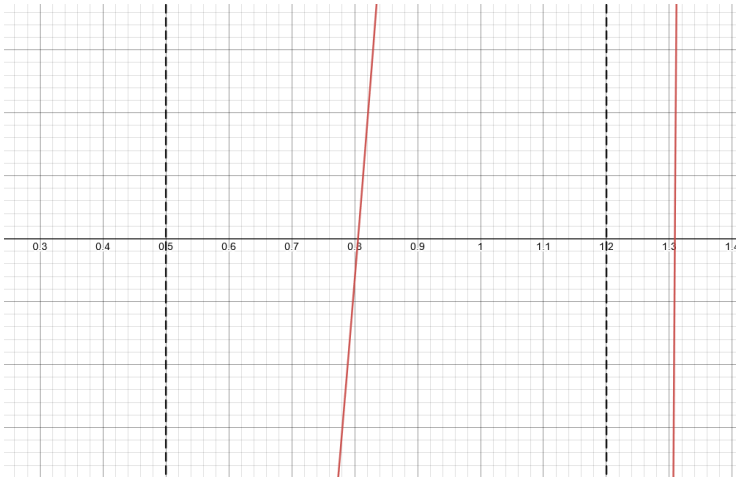
$$e^{\frac{1}{x}} - 5x^3 + 2x^2 - 2$$

```
>-----------------------------------<
precision: 10
expression: E^(1/x)-5*x^3+2*x**2-2
initial range: 0.5
final range: 2
error margin: 0.00000000001
max iterations: 100
>-- root (bisector) --<
Iterations: 37
result: 0.8177615098
>-- root (Newton) --<
Iterations: 4
result: 0.8177615098
>-- root (Secant) --<
Iterations: 6
result: 0.8177615098
>-----------------------------------<
```



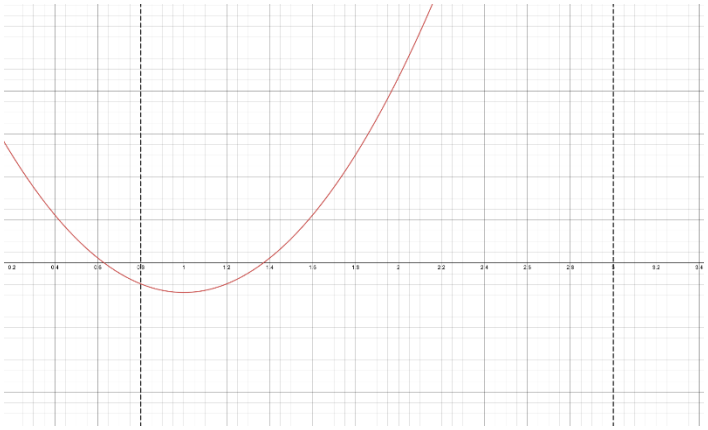
$$x^5 + \frac{2}{3}x^2 + e^x - log_{10}(727x) + xlog_3(x) + \tan(x^2) - \frac{3}{2x} + \sqrt{2x^3}$$

```
>-----------------------------------<
precision: 100
expression: x^5+(2/3)*x^2+E^x-log(727*x,10)+x*log(x,3)+tan(x^2)-3/(2*x)+sqrt(2*x^3)
initial range: 0.5
final range: 1.2
error margin: 0.0000000001
max iterations: 100
>-- root (bisector) --<
Iterations: 33
result: 0.80535181957529857754707336425781250000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
>-- root (Newton) --<
Iterations: 3
result: 0.80535181964761969502655695897412663846061557428858376751295158815072921097211855820172423539573839 93
>-- root (Secant) --<
Iterations: 7
result: 0.80535181964761969502651552479332842747134014574097103404421803820794737718597489933740557214821873 46
>-----------------------------------<
```



$$(x-1)^2 - 0.137$$

```
>-----------------------------------<
precision: 5
expression: (x-1)^2-0.137
initial range: 1.2
final range: 3
error margin: 10^(-5)
max iterations: 3
>-- root (bisector) --<
Reached max iterations, stopping.
Iterations: 3
result: 1.4250
>-- root (Newton) --<
Reached max iterations, stopping.
Iterations: 3
result: 1.3702
>-- root (Secant) --<
Reached max iterations, stopping.
Iterations: 3
result: 1.3671
>-----------------------------------<
```

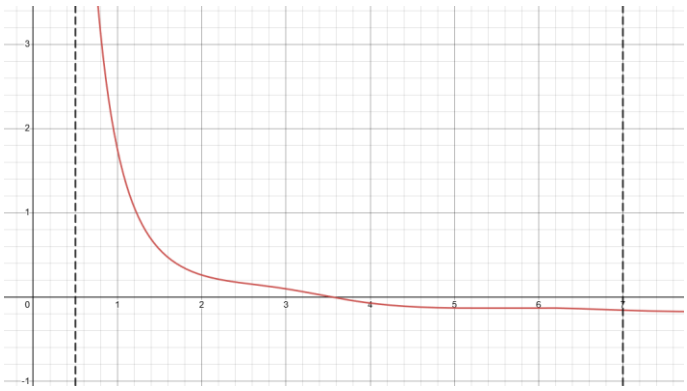


$(x-1)^2 - 0.137$ (cont.)

```
>-----------------------------------<
precision: 5
expression: (x-1)^2-0.137
initial range: 1.2
final range: 3
error margin: 10^(-5)
max iterations: 20
>-- root (bisector) --<
Iterations: 18
result: 1.3701
>-- root (Newton) --<
Iterations: 5
result: 1.3701
>-- root (Secant) --<
Iterations: 6
result: 1.3701
>-----------------------------------<
```

$$\frac{e^{\cos(\sin(x))}}{x^2} - 0.2$$

```
>-----------------------------------<
precision: 15
expression: E^(cos(sin(x)))*(1/x^2)-0.2
initial range: 0.5
final range: 7
error margin: 10^-10
max iterations: 100
>-- root (bisector) --<
Iterations: 36
result: 3.54750539649831
>-- root (Newton) --<
Iterations: 4
result: 3.54750539643728
>-- root (Secant) --<
Iterations: 9
result: 3.54750539643728
>-----------------------------------<
```



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
```