Elementos do grupo:

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

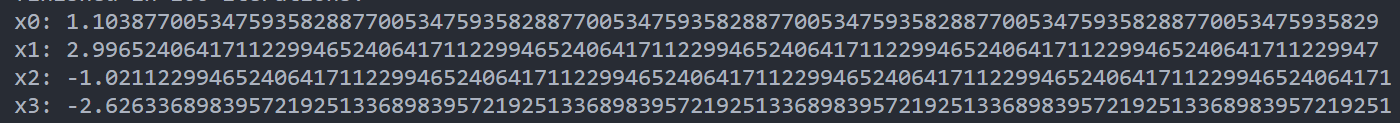
Resumo

O programa é capaz de aproximar a solução de um sistema de equações não linear. Isto é feito usando o método de Newton-Raphson. Este método só funciona quando os valores iniciais estão muito próximos da solução do sistema.

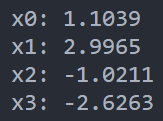
Instruções de Utilização

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: ()



* **Number of Variables:**  – O número de expressões (e variáveis) a ser introduzido. ()
* **Expression:**  – A função matemática à qual se encontrar a solução
  + **Variáveis**: ‘xn’, onde xn = xn.
  + **Operadores aritméticos:** ‘+’, ‘-‘, ‘/’, ‘\*’.
  + **Funções Trigonométricas:** ’’, ‘’, ‘’, ‘’, ‘’, ‘’, ‘acos(x)’, ‘asin(x)’, ‘atan(x)’, ‘asec(x)’, ‘acsc(x)’, ‘acot(x)’, ‘cosh(x)’, ‘sinh(x)’, ‘tanh(x)’, ‘sech(x)’, ‘csch(x)’, ‘coth(x)’, ‘acosh(x)’, ‘asinh(x)’, ‘atanh(x)’, ‘asech(x)’, ‘acsch(x)’, ‘acoth(x)’.
  + **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 uso de fatoriais pode tornar o programa bastante ineficiente)

Exemplo:

Exemplo:

* 
* **Initial Values:**  – Os valores da iteração inicial (número real (incluí constantes e aritmética))
* **Final Range:**  – O número real (incluí constantes e aritmética) ao qual o integral vai integrar até. ()
* **Error Margin:** – O número real (incluí constantes e aritmética) que define o erro máximo norma infinito da aproximação 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

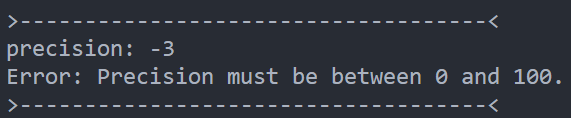
Cada ficheiro aqui posto deve ser colocado no seu ficheiro particular com o mesmo nome que a sua classe, este projeto usa também o auxílio da biblioteca matemática SymPy (<https://www.sympy.org>). Para a sua execução é necessário instalar a mesma para o seu funcionamento correto. Com python instalado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

Observações

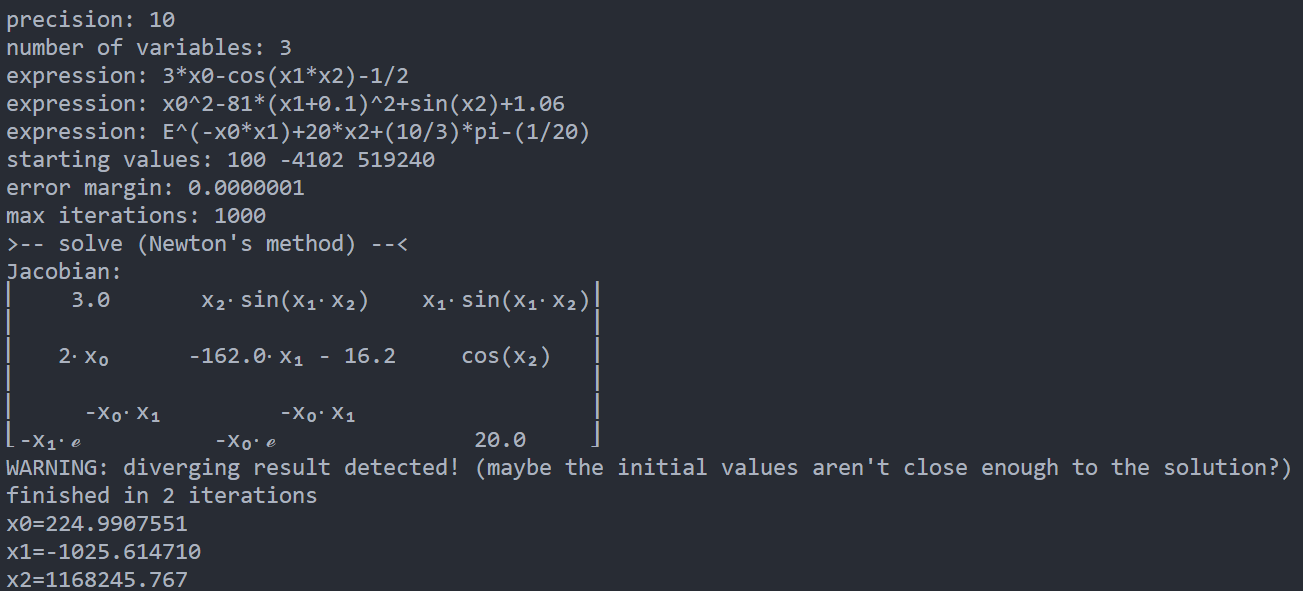
Pelo método de Newton-Raphson a aproximação só converge quando os valores iniciais estão muito perto da solução.

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 (valores iniciais não estão perto da solução):

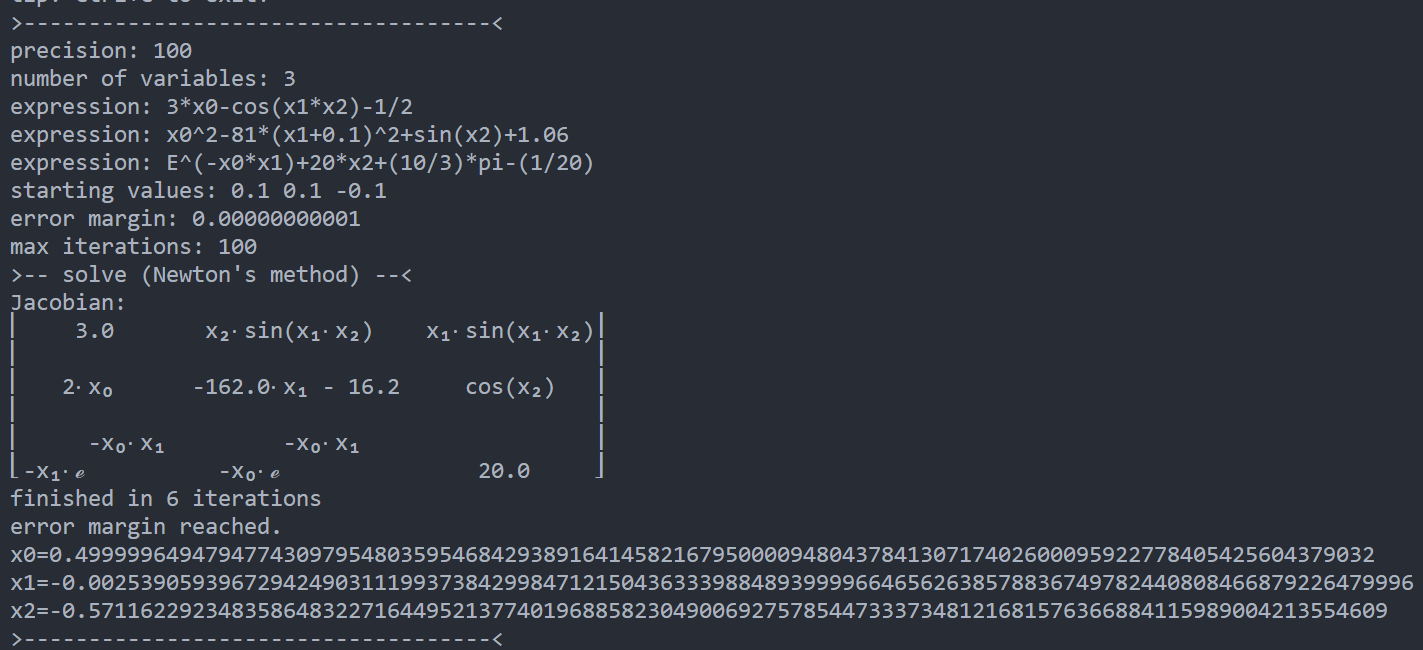


Screenshots

**Valores iniciais: { 0.1; 0.1; -0.1 }**

**Margem de erro: 0.00000000001**

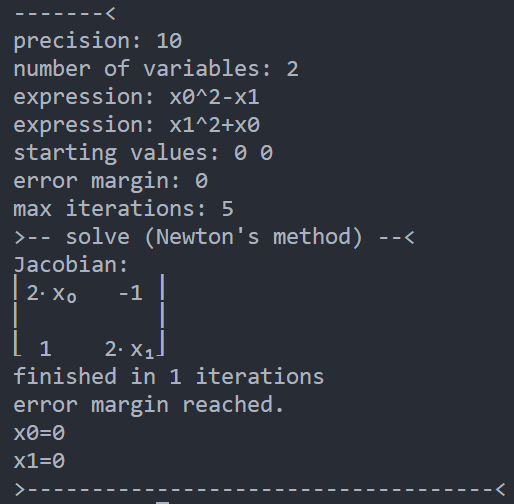
**Iterações máximas: 100**



**Valores iniciais: { 0; 0 }**

**Margem de erro: 0**

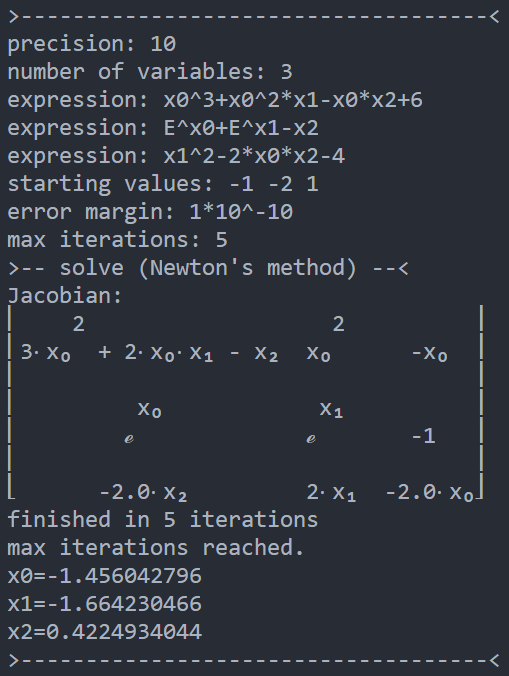
**Iterações máximas: 5**



**Valores iniciais: { -1; -2; 1 }**

**Margem de erro:**

**Iterações máximas: 5**



Código

main.py

import math\_input

import newton

from sympy import \*

def print\_approx(*approximations*, *precision*):

    for i in range(len(*approximations*)):

        print(f"x{i}: {round(*approximations*[i], *precision*)}")

exit = false

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

while exit == false:

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

    try:

        precision = math\_input.math\_input.get\_precision()

        num\_variables = math\_input.math\_input.get\_num\_variables()

        equations = []

        for i in range(num\_variables):

            equations.append(math\_input.math\_input.get\_expression(precision))

        initial\_values = math\_input.math\_input.get\_initial\_values(num\_variables, precision)

        error\_margin = math\_input.math\_input.get\_error\_margin(precision)

        max\_iterations = math\_input.math\_input.get\_max\_iterations(precision)

        try:

            print(">-- solve (Newton's method) --<")

            result = newton.newton.solve(equations, initial\_values, error\_margin, max\_iterations)

            newton.newton.print\_result(result)

        except Exception as e:

            print("Newton's method failed.")

            print(str(e))

    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\_num\_variables():

        this\_input = input("number of variables: ")

        try:

            try:

                this\_input = int(this\_input)

            except:

                raise Exception("Error: number of variables must be an integer.")

            if this\_input <= 0:

                raise Exception("Error: number of variables should be a number greater than zero.")

            return 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\_initial\_values(*num\_variables*, *precision*):

        row = []

        this\_input = input("starting values: ")

        try:

            tokens = this\_input.split(" ")

            if len(tokens) != num\_variables:

                raise Exception(f"Error: expected {num\_variables} values but got {len(tokens)}.")

            for token in tokens:

                cell = N(token, precision)

                if not ask(Q.real(cell)):

                    raise Exception(f"Error: Invalid value '{token}'.")

                row.append(N(token, precision))

            return row

        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

newton.py

from sympy import \*

class newton:

    @staticmethod

    def solve(*equations*, *values*, *error\_margin*, *max\_iterations*):

        if (*max\_iterations* == 0):

            return *values*

        J = newton.calc\_jacobian(*equations*)

        X = Matrix(*values*)

        eq\_matrix = Matrix(*equations*)

        F = newton.subs\_matrix(eq\_matrix, *values*)

        iteration\_J = newton.subs\_matrix(J, *values*).inv()

        result = X - iteration\_J \* F

        print("Jacobian:")

        pprint(J)

        old\_values = *values*.copy()

        for i in range(len(*values*)):

*values*[i] = result[i]

        iteration = 1

        error = newton.calc\_error(old\_values, *values*)

        last\_error = 0

        while(iteration < *max\_iterations* and error > *error\_margin*):

            X = Matrix(*values*)

            F = newton.subs\_matrix(eq\_matrix, *values*)

            iteration\_J = newton.subs\_matrix(J, *values*).inv()

            result = X - iteration\_J \* F

            old\_values = *values*.copy()

            for i in range(len(*values*)):

*values*[i] = result[i]

            iteration += 1

            last\_error = error

            error = newton.calc\_error(old\_values, *values*)

            if (last\_error < error):

                print("WARNING: diverging result detected! (maybe the initial values aren't close enough to the solution?)")

                break

        print(f"finished in {iteration} iterations")

        if (iteration >= *max\_iterations*):

            print("max iterations reached.")

        if (error <= *error\_margin*):

            print("error margin reached.")

        return *values*

    @staticmethod

    def calc\_jacobian(*equations*):

        rows = []

        for row in range(len(*equations*)):

            cols = []

            for col in range(len(*equations*)):

                cols.append(diff(*equations*[row], Symbol("x" + str(col))))

            rows.append(cols)

        return Matrix(rows)

    @staticmethod

    def subs\_matrix(*matrix*, *values*):

        rows = []

        for row in range(*matrix*.rows):

            cols = []

            for col in range(*matrix*.cols):

                cell = *matrix*[row, col]

                for valueIndex in range(len(*values*)):

                    cell = cell.subs(Symbol("x" + str(valueIndex)), *values*[valueIndex])

                cols.append(cell)

            rows.append(cols)

        return Matrix(rows)

    @staticmethod

    def calc\_error(*old\_values*, *values*):

        error = 0

        errors = []

        for i in range(len(*old\_values*)):

            errors.append(abs(*values*[i] - *old\_values*[i]))

        error = max(errors)

        return error

    @staticmethod

    def print\_result(*result*):

        for i in range(len(*result*)):

            print("x" + str(i) + "=" + str(*result*[i]))