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

  Exemplo: $(x = 100)$

```
1.1038770053475935828877005347593582887700534759358288770053475935828877005347593582887700534759358288770053475935829
```

  Exemplo: $(x = 5)$

```
1.1039
```

- **Number of Variables:** n – O número de expressões (e variáveis) a ser introduzido. $(1 < x < \infty)$
- **Expression:** $\boldsymbol{f}(x)$ – A função matemática à qual se encontrar a solução
  - **Variáveis**: ‘xn’, onde xn = x_n.
  - **Operadores aritméticos:** ‘+’, ‘-‘, ‘/’, ‘*’.
  - **Funções Trigonométricas:** ’$cos(x)$’, ‘$sin(x)$’, ‘$tan(x)$’, ‘$cot(x)$’, ‘$sec(x)$’, ‘$csc(x)$’, ‘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:** ‘$log(x, n)$’ (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 uso de fatoriais pode tornar o programa bastante ineficiente)

  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 Values:** x_0 ... x_n – Os valores da iteração inicial (número real (incluí constantes e aritmética))
- **Final Range:** x_1 – O número real (incluí constantes e aritmética) ao qual o integral vai integrar até. ($x_0 \leq x_1$)
- **Error Margin:** ε – 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**: 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

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:

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

Exemplo (valores iniciais não estão perto da solução):

```
precision: 10
number of variables: 3
expression: 3*x0-cos(x1*x2)-1/2
expression: x0^2-81*(x1+0.1)^2+sin(x2)+1.06
expression: E^(-x0*x1)+20*x2+(10/3)*pi-(1/20)
starting values: 100 -4102 519240
error margin: 0.0000001
max iterations: 1000
>-- solve (Newton's method) --<
Jacobian:
⎡    3.0       x₂⋅sin(x₁⋅x₂)    x₁⋅sin(x₁⋅x₂)⎤
⎢                                            ⎥
⎢   2⋅x₀      -162.0⋅x₁ - 16.2     cos(x₂)   ⎥
⎢                                            ⎥
⎢     -x₀⋅x₁          -x₀⋅x₁                 ⎥
⎣-x₁⋅ℯ           -x₀⋅ℯ              20.0     ⎦
WARNING: diverging result detected! (maybe the initial values aren't close enough to the solution?)
finished in 2 iterations
x0=224.9907551
x1=-1025.614710
x2=1168245.767
```

Screenshots

$$\begin{cases} 3x_0 - cos(x_1 \times x_2) - \frac{1}{2} \\ x_0{}^2 - 81(x_1 + 0.1)^2 + sin(x_2) + 1.06 \\ e^{-(x_0 \times x_1)} + 20x_2 + \frac{10}{3}\pi - 1 \end{cases}$$

Valores iniciais: { 0.1; 0.1; -0.1 }
Margem de erro: 0.00000000001
Iterações máximas: 100

```
>-----------------------------------<
precision: 100
number of variables: 3
expression: 3*x0-cos(x1*x2)-1/2
expression: x0^2-81*(x1+0.1)^2+sin(x2)+1.06
expression: E^(-x0*x1)+20*x2+(10/3)*pi-(1/20)
starting values: 0.1 0.1 -0.1
error margin: 0.00000000001
max iterations: 100
>-- solve (Newton's method) --<
Jacobian:
⎡    3.0       x₂⋅sin(x₁⋅x₂)    x₁⋅sin(x₁⋅x₂)⎤
⎢                                            ⎥
⎢   2⋅x₀      -162.0⋅x₁ - 16.2     cos(x₂)   ⎥
⎢                                            ⎥
⎢     -x₀⋅x₁          -x₀⋅x₁                 ⎥
⎣-x₁⋅ℯ           -x₀⋅ℯ              20.0     ⎦
finished in 6 iterations
error margin reached.
x0=0.4999996494794774309795480359546842938916414582167950000948043784130717402600095922778405425604379032
x1=-0.002539059396729424903111993738429984712150436333988489399996646562638578836749782440808466879226479996
x2=-0.5711622923483586483227164495213774019688582304900692757854473337348121681576366884115989004213554609
>-----------------------------------<
```

$$\begin{cases} {x_0}^2 - x_1 \\ {x_1}^2 + x_0 \end{cases}$$

Valores iniciais: { 0; 0 }
Margem de erro: 0
Iterações máximas: 5

```
-------<
precision: 10
number of variables: 2
expression: x0^2-x1
expression: x1^2+x0
starting values: 0 0
error margin: 0
max iterations: 5
>-- solve (Newton's method) --<
Jacobian:
⎡2⋅x₀   -1 ⎤
⎢          ⎥
⎣ 1    2⋅x₁⎦
finished in 1 iterations
error margin reached.
x0=0
x1=0
>-------------------------------------<
```

$$\begin{cases} {x_0}^3 + {x_0}^2 x_1 - x_0 x_2 + 6 \\ e^{x_0} + e^{x_1} - x_2 \\ {x_1}^2 - 2x_0 x_2 - 4 \end{cases}$$

Valores iniciais: { -1; -2; 1 }
Margem de erro: 1×10^{-10}
Iterações máximas: 5

```
>-------------------------------------<
precision: 10
number of variables: 3
expression: x0^3+x0^2*x1-x0*x2+6
expression: E^x0+E^x1-x2
expression: x1^2-2*x0*x2-4
starting values: -1 -2 1
error margin: 1*10^-10
max iterations: 5
>-- solve (Newton's method) --<
Jacobian:
⎡     2                    2          ⎤
⎢3⋅x₀  + 2⋅x₀⋅x₁ - x₂   x₀       -x₀  ⎥
⎢                                     ⎥
⎢          x₀            x₁           ⎥
⎢         ℯ             ℯ         -1  ⎥
⎢                                     ⎥
⎣      -2.0⋅x₂          2⋅x₁   -2.0⋅x₀⎦
finished in 5 iterations
max iterations reached.
x0=-1.456042796
x1=-1.664230466
x2=0.4224934044
>-------------------------------------<
```

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

```python
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]))
```