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 resolve qualquer matriz possível determinada com até 100 algarismos significativos de precisão, que seja resolvível pelo método de Jacobi/Gauss-Seidel. Isto pode requerer que a matriz seja diagonalmente dominante, isto é, só garante resolver matrizes as quais a diagonal seja maior que a soma dos outros elementos da mesma linha.

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: $(x = 100)$

```
x0: 1.10387700534759358288770053475935828877005347593582887700534759358288770053475935829
x1: 2.99652406417112299465240641711229946524064171122994652406417112299465240641711229947
x2: -1.02112299465240641711229946524064171122994652406417112299465240641711229946524064171
x3: -2.62633689839572192513368983957219251336898395721925133689839572192513368983957219251
```

  Exemplo: $(x = 5)$

```
x0: 1.1039
x1: 2.9965
x2: -1.0211
x3: -2.6263
```

- **Number of Variables:** n – O número de variáveis do sistema de equações. Este número vai definir o tamanho da matriz a introduzir $(n \times n)$. $(0 < x < \infty)$

  Exemplo: $n = 2$

```
[1.0  2.0]
[       ]
[3.0  4.0]
```

  Exemplo: $n = 4$

```
[1.0  2.0  3.0  4.0]
[                   ]
[5.0  6.0  7.0  8.0]
[                   ]
[9.0   0   1.0  2.0]
[                   ]
[3.0  4.0  5.0  6.0]
```

- **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

Como o programa funciona com base no método de Jacobi e de Gauss-Seidel, ele tem as mesmas limitações que estes métodos têm: só garantem a convergência do resultado quando a matriz é diagonalmente dominante. O uso de uma matriz não diagonalmente dominante pode levar a um funcionamento irregular, dito isto o programa está pronto para detetar quando os resultados estão a divergir, quando possível.

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
number of variables: 3
row1: 2 -1 1
row2: 1 1 1
row3: 1 1 1
equalities: 1 2 3
[2.0  -1.0  1.0] [1.0]
[              ] [   ]
[1.0  1.0   1.0]*[2.0]
[              ] [   ]
[1.0  1.0   1.0] [3.0]
error margin: 0
max iterations: 100
>-- solve (Jacobi) --<
Jacobi's method failed.
Error: Matrix is impossible. Determinant is 0.
>-- solve (Gauss) --<
Gauss's method failed.
Error: Matrix is impossible. Determinant is 0.
>---------------------------------<
```

Exemplo:

```
>-----------------------------------<
precision: 10
number of variables: 3
row1: 2 -1 1
row2: 3 3 9
row3: 3 3 5
equalities: -1 0 4
[2.0  -1.0  1.0] [-1.0]
[              ] [    ]
[3.0  3.0   9.0]*[ 0  ]
[              ] [    ]
[3.0  3.0   5.0] [4.0 ]
error margin: 0
max iterations: 100
>-- solve (Jacobi) --<
Jacobi's method failed.
Error: Matrix diverges with Jacobi's method.
>-- solve (Gauss) --<
Gauss's method failed.
Error: Matrix diverges with Gauss's method.
>-----------------------------------<
```

Explicação: A matriz apresentada não é diagonalmente dominante, isto pode ser evidenciado pelas linhas 2 e 3: $(3 < 3 + 9)$ $(5 < 3 + 3)$. Isto causa os métodos a divergirem, o programa ao detetar esta situação para a execução e dá uma mensagem de erro apropriada.

Screenshots

$$\begin{bmatrix} 10 & -1 & 2 & 0 \\ -1 & 11 & -1 & 3 \\ 2 & -1 & 10 & 0 \\ 0 & 3 & -1 & 8 \end{bmatrix} \begin{bmatrix} 6 \\ 25 \\ -11 \\ -11 \end{bmatrix}$$

```
>-----------------------------------<
precision: 100
number of variables: 4
row1: 10 -1 2 0
row2: -1 11 -1 3
row3: 2 -1 10 0
row4: 0 3 -1 8
equalities: 6 25 -11 -11
[10.0  -1.0  2.0    0 ] [ 6.0 ]
[                     ] [     ]
[-1.0  11.0  -1.0  3.0] [25.0 ]
[                     ]*[     ]
[2.0   -1.0  10.0   0 ] [-11.0]
[                     ] [     ]
[ 0    3.0   -1.0  8.0] [-11.0]
error margin: 10^(-10)
max iterations: 1000
>-- solve (Jacobi) --<
finished in 27 iterations.
x0: 1.10387700534399702672346515932716402911660052390677280781342679562966165225452649054204512350573452 4
x1: 2.99652406417781225652528064651260033625703767001921902043422538991063084776428000093185970664140150 6
x2: -1.02112299465600297299132216867283597088339947609322719218657320437033834774547350945795487649426547 6
x3: -2.62633689838775091992204576815074667247574677432802499407326500241466820270245254535139946986284631 7
>-- solve (Gauss) --<
finished in 13 iterations.
x0: 1.10387700534588545987919318458457583658666047402744797758806970720526066933095020468053048932327698 2
x1: 2.99652406416971720260263318928235699545465592760720741354510777603564553930217017373082812047756207 0
x2: -1.02112299465220537171557531798867946777186650204476885416310316383748757993597302356302328581689918 9
x3: -2.62633689839516962244043436072946880676697928560829888684980331149305302473031044309443845590619817 5
>-----------------------------------<
```

$$\begin{bmatrix} 3 & 2 \\ 1 & 5 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

```
>-----------------------------------<
precision: 5
number of variables: 2
row1: 3 2
row2: 1 5
equalities: 1 -1
[3.0  2.0] [1.0 ]
[       ]*[    ]
[1.0  5.0] [-1.0]
error margin: 0.1
max iterations: 30
>-- solve (Jacobi) --<
finished in 3 iterations.
x0: 0.51111
x1: -0.29333
>-- solve (Gauss) --<
finished in 3 iterations.
x0: 0.53482
x1: -0.30696
>-----------------------------------<
>-----------------------------------<
precision: 5
number of variables: 2
row1: 3 2
row2: 1 5
equalities: 1 -1
[3.0  2.0] [1.0 ]
[       ]*[    ]
[1.0  5.0] [-1.0]
error margin: 0.00001
max iterations: 30
>-- solve (Jacobi) --<
finished in 12 iterations.
x0: 0.53846
x1: -0.30769
>-- solve (Gauss) --<
finished in 7 iterations.
x0: 0.53846
x1: -0.30769
>-----------------------------------<
```

$$\begin{bmatrix} 0.5 & 1 \\ 1 & 5 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

```
>-----------------------------------<
precision: 10
number of variables: 2
row1: 0.5 1
row2: 1 5
equalities: 1 -1
[0.5  1.0] [1.0 ]
[       ]*[    ]
[1.0  5.0] [-1.0]
error margin: 10^(-20)
max iterations: 100
>-- solve (Jacobi) --<
Jacobi's method failed.
Error: Matrix diverges with Jacobi's method.
>-- solve (Gauss) --<
finished in 30 iterations.
x0: 4.000000000
x1: -1.000000000
>-----------------------------------<
```

$$\begin{bmatrix} 3 & \frac{1}{4} & 1 \\ 1 & 5 & 3 \\ 9 & 1 & 15 \end{bmatrix} \begin{bmatrix} 6 \\ 2 \\ 1 \end{bmatrix}$$

```
>-----------------------------------<
precision: 10
number of variables: 3
row1: 3 1/4 1
row2: 1 5 3
row3: 9 1 15
equalities: 6 2 1
[3.0  0.25  1.0 ] [6.0]
[               ] [   ]
[1.0  5.0   3.0 ]*[2.0]
[               ] [   ]
[9.0  1.0   15.0] [1.0]
error margin: 10^(-50)
max iterations: 15
>-- solve (Jacobi) --<
max iterations reached.
finished in 15 iterations.
x0: 2.412955207
x1: 0.7773085433
x2: -1.432722823
>-- solve (Gauss) --<
max iterations reached.
finished in 15 iterations.
x0: 2.412857143
x1: 0.7771428572
x2: -1.432857143
>-----------------------------------<
```

Código

main.py

```
import math_input
import jacobi as my_jacobi
import gauss
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()
        # num_equations =
math_input.math_input.get_num_equations(num_variables)
        rows = []
        for i in range(num_variables): # changed to num_variables
            rows.append(math_input.math_input.get_row(num_variables, i + 1,
precision))
        equalities = math_input.math_input.get_equalities(num_variables,
precision)
        matrix = Matrix(rows)
        equalities_matrix = Matrix(equalities)
        pprint(MatMul(matrix, equalities_matrix), use_unicode=False)
        error_margin = math_input.math_input.get_error_margin(precision)
        max_iterations = math_input.math_input.get_max_iterations(precision)
        try:
            print(">-- solve (Jacobi) --<")
            result_jacobi = my_jacobi.jacobi.solve(matrix, equalities,
error_margin, max_iterations)
            print_approx(result_jacobi, precision)
        except Exception as e:
            print("Jacobi's method failed.")
            print(str(e))
        try:
            print(">-- solve (Gauss) --<")
            result_gauss = gauss.gauss.solve(matrix, equalities, error_margin,
max_iterations)
            print_approx(result_gauss, precision)
        except Exception as e:
```

```
            print("Gauss'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_row(num_variables, row_num, precision):
        row = []
        this_input = input(f"row{row_num}: ")
        try:
            tokens = this_input.split(" ")
            if len(tokens) != num_variables:
                raise Exception(f"Error: expected {num_variables} variables
but got {len(tokens)}.")
            for token in tokens:
                cell = N(token, precision)
                if not ask(Q.real(cell)):
                    raise Exception(f"Error: Invalid cell '{token}'.")
                row.append(N(token, precision))
            return row
        except Exception as e:
            raise e

    @staticmethod
    def get_equalities(num_variables, precision):
        row = []
        this_input = input("equalities: ")
        try:
            tokens = this_input.split(" ")
            if len(tokens) != num_variables:
                raise Exception(f"Error: expected {num_variables} equalities
but got {len(tokens)}.")
            for token in tokens:
                cell = N(token, precision)
                if not ask(Q.real(cell)):
                    raise Exception(f"Error: Invalid equality '{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
```

matrix_utils.py

```
from sympy import *

class matrix_utils:
    @staticmethod
    def substitute_values(expression, values):
        for i in range(len(values)):
            expression = expression.subs(Symbol("x" + str(i)), values[i])
        return expression


    @staticmethod
    def get_expressions(matrix, equalities):
        expressions = []
        for i in range(matrix.rows):
            current_expression = equalities[i]
            current_diag = matrix[i, i]
            if current_diag == 0:
                raise Exception("Error: Invalid matrix. Diagonal should not
contain zeros.")
            for j in range(matrix.cols):
                if i != j: # if not in the diagonal
                    current_expression += Symbol("x" + str(j)) * -1 *
matrix[i, j]
            current_expression /= current_diag
            expressions.append(current_expression)
        return expressions

    @staticmethod
```

```
    def calc_error(approx, new_approx):
        error = 0
        for i in range(len(approx)):
            error += abs(new_approx[i] - approx[i])
        return error
```

jacobi.py

```
from sympy import *
import matrix_utils

class jacobi:
    @staticmethod
    def solve(matrix, equalities, error_margin, max_iterations):
        if matrix.det() == 0:
            raise Exception("Error: Matrix is impossible. Determinant is 0.")

        expressions = matrix_utils.matrix_utils.get_expressions(matrix,
equalities)
        approximations = [0] * matrix.rows
        new_approx = []
        iterations = 0
        error = 0
        if iterations < max_iterations:
            for i in range(matrix.rows):
                new_approx.append(matrix_utils.matrix_utils.substitute_values(
expressions[i], approximations))
            error = matrix_utils.matrix_utils.calc_error(approximations,
new_approx)
            approximations = new_approx
            new_approx = []
            iterations += 1
        while (iterations < max_iterations and error > error_margin):
            for i in range(matrix.rows):
                new_approx.append(matrix_utils.matrix_utils.substitute_values(
expressions[i], approximations))
            new_error = matrix_utils.matrix_utils.calc_error(approximations,
new_approx)
            if (new_error > error):
                raise Exception("Error: Matrix diverges with Jacobi's
method.")
            error = new_error
            approximations = new_approx
            new_approx = []
            iterations += 1
```

```
        if (iterations >= max_iterations):
            print("max iterations reached.")

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

        return approximations
```

gauss.py

```
from sympy import *
import matrix_utils

class gauss:
    @staticmethod
    def solve(matrix, equalities, error_margin, max_iterations):
        if matrix.det() == 0:
            raise Exception("Error: Matrix is impossible. Determinant is 0.")

        expressions = matrix_utils.matrix_utils.get_expressions(matrix,
equalities)
        approximations = [0] * matrix.rows
        new_approx = []

        iterations = 0
        error = 0
        if iterations < max_iterations:
            new_approx = approximations.copy()
            for i in range(matrix.rows):
                new_approx[i] =
matrix_utils.matrix_utils.substitute_values(expressions[i], new_approx)
            error = matrix_utils.matrix_utils.calc_error(approximations,
new_approx)
            approximations = new_approx
            iterations += 1
        while (iterations < max_iterations and error > error_margin):
            new_approx = approximations.copy()
            for i in range(matrix.rows):
                new_approx[i] =
matrix_utils.matrix_utils.substitute_values(expressions[i], new_approx)
            new_error = matrix_utils.matrix_utils.calc_error(approximations,
new_approx)
            if (new_error > error):
                raise Exception("Error: Matrix diverges with Gauss's method.")
            error = new_error
            approximations = new_approx
            iterations += 1
```

```
    if (iterations >= max_iterations):
        print("max iterations reached.")

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

    return approximations
```