Elementos do grupo:

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

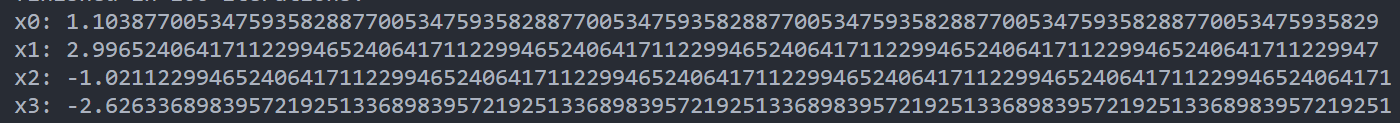
Resumo

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.

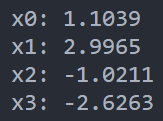
Instruções de Utilização

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

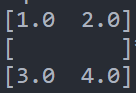
* **Precision:**  – O número (inteiro) de algarismos significativos a ser usado internamente pelo programa. ()

Exemplo: ()

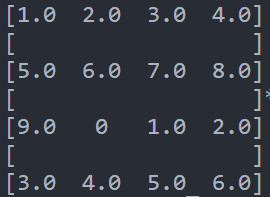
Exemplo: ()



* **Number of Variables:**  – O número de variáveis do sistema de equações. Este número vai definir o tamanho da matriz a introduzir (). ()

Exemplo:

Exemplo:



* **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

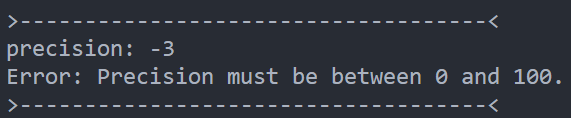
Cada ficheiro aqui posto deve ser colocado no seu ficheiro particular com o mesmo nome que a sua classe, este projeto usa também o auxílio da biblioteca matemática SymPy (<https://www.sympy.org>). Para a sua execução é necessário instalar a mesma para o seu funcionamento correto. Com python instalado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

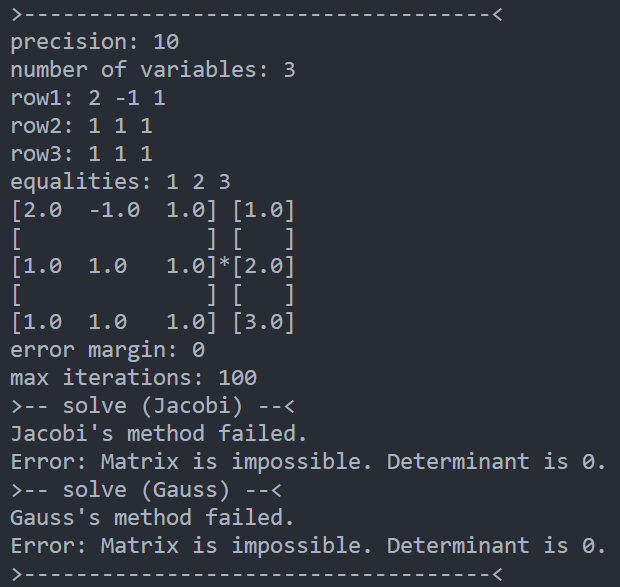
Observações

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.

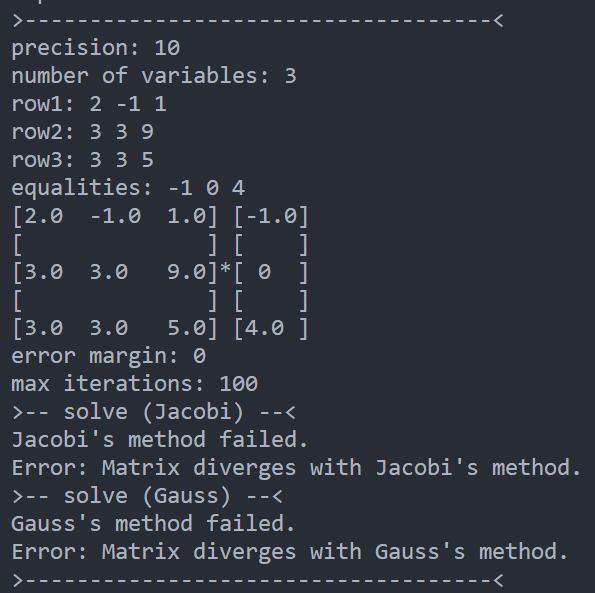
O programa está preparado para obter qualquer tipo de input, mesmo que este viole as especificações previamente mencionadas, gerando uma mensagem de erro apropriada, mas continuando a execução.

Exemplo:



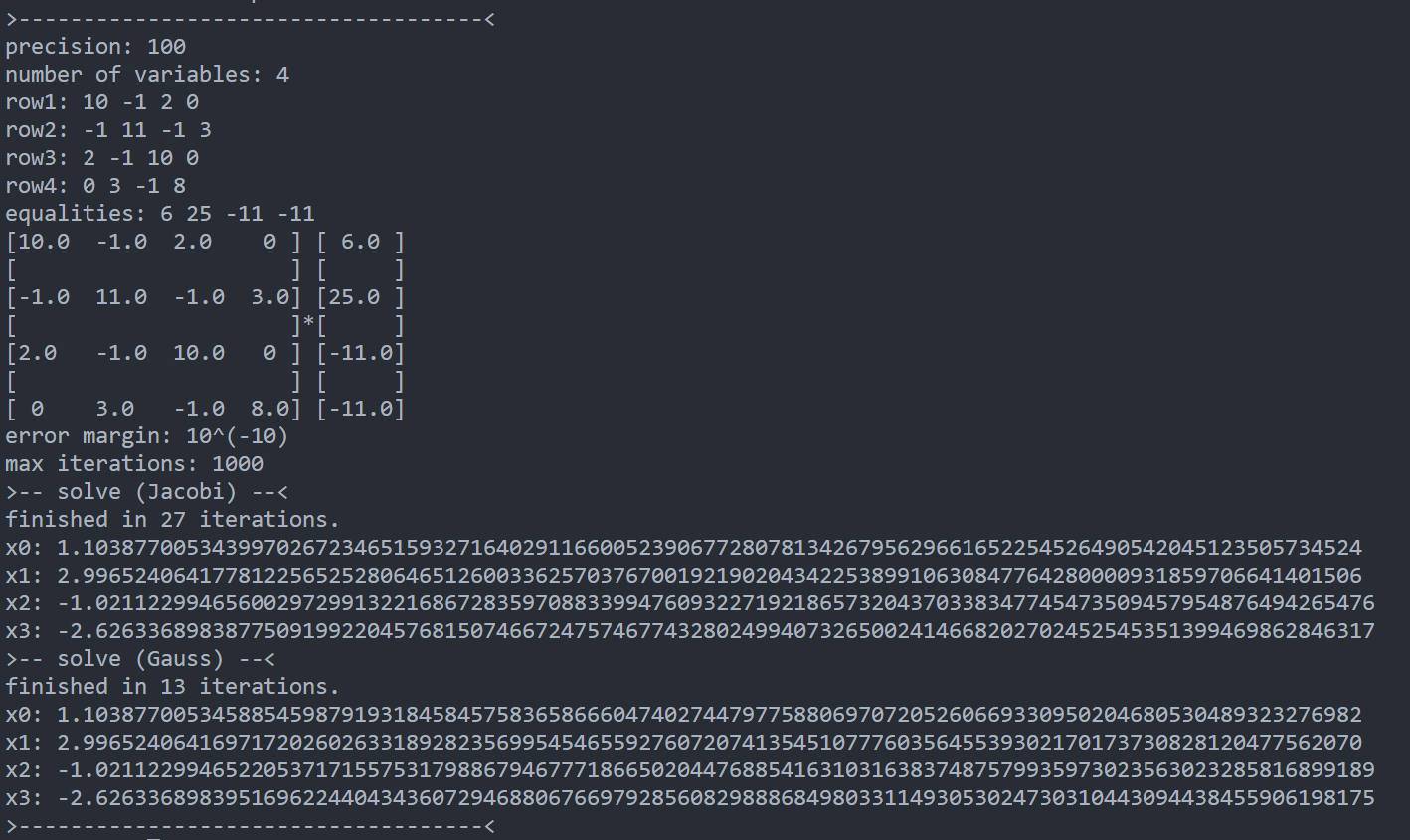
 Exemplo:

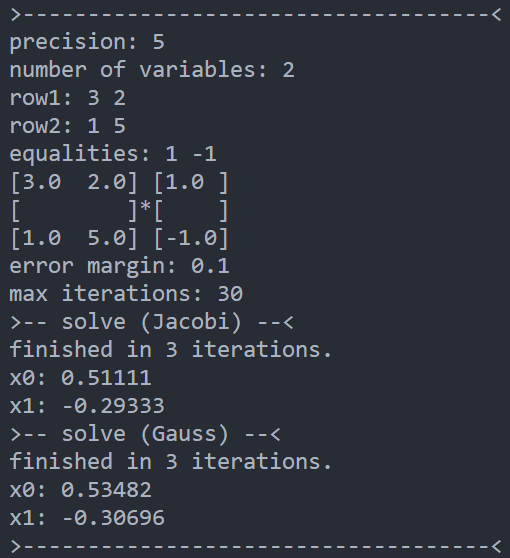
Exemplo:

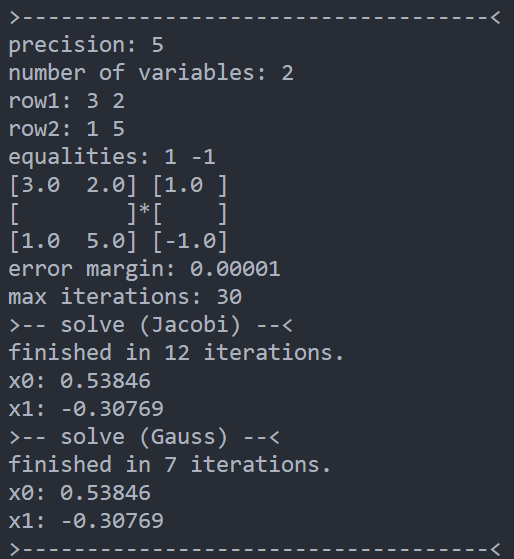


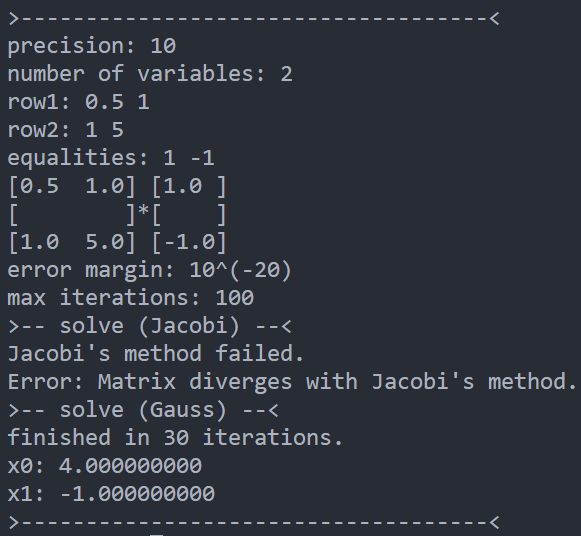
Explicação: A matriz apresentada não é diagonalmente dominante, isto pode ser evidenciado pelas linhas 2 e 3: () (). Isto causa os métodos a divergirem, o programa ao detetar esta situação para a execução e dá uma mensagem de erro apropriada.

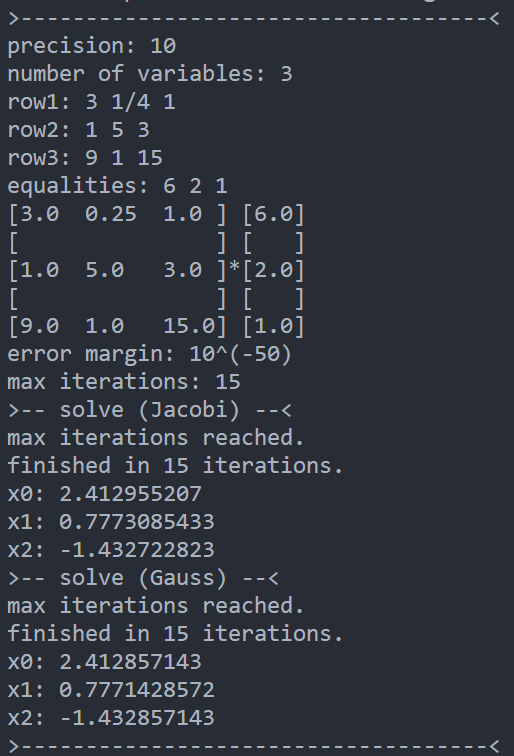
Screenshots











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