**ИНСТИТУТ ТРАНСПОРТА И СВЯЗИ**



ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ НАУК И ТЕЛЕКОММУНИКАЦИЙ

КАФЕДРА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ КОМПЬЮТЕРНЫХ СИСТЕМ

**Лабораторная работа №1**

по дисциплине

„Численные методы”

Тема: «Методы решения системы линейных уравнений.

Число обусловленности матрицы.»

Выполнил: Денис Белов, Андрей Савкин, Евгений Хрущ

Рига.

2020.

**1)** **Формулировка задания**

В данной работе необходимо было реализовать метод исключения Гаусса с ведущим элементом и индивидуальный метод: метод Гаусса-Зейделя. В зависимости является ли метод итерационным или прямым, получить результат приведённых примеров.

Так же реализовать экспериментальный расчёт числа обусловленности матрицы и получить результаты двух систем.

**2) Метод исключения Гаусса с ведущим элементом**

class GaussElimination:

    def \_\_init\_\_(self, matrixAB: list, vars: list, print\_only\_results: bool = False, matrix\_name: str = "", matrixA=None, vectorB=None, print\_results=True, without\_print=False):

        self.matrix\_len = len(matrixAB)

        matrix\_copy = copy.deepcopy(matrixAB)

        def print\_matrix():

            for (var, row) in zip(vars, matrix\_copy):

                print(var, \*["%0.4f" % elem for elem in row], sep='\t')

        # Find the pivot element - we need to put it as the first row in the matrix

        for i in range(self.matrix\_len):

            for k in range(i + 1, self.matrix\_len):

                if abs(matrix\_copy[i][i]) < abs(matrix\_copy[k][i]):

                    for j in range(0, self.matrix\_len + 1):

                        # Swapping elements

                        matrix\_copy[i][j], matrix\_copy[k][j] = matrix\_copy[k][j], matrix\_copy[i][j]

        # Main Gauss Elimination loop

        for i in range(self.matrix\_len - 1):

            for k in range(i + 1, self.matrix\_len):

                coefficient = matrix\_copy[k][i] / matrix\_copy[i][i]

                for j in range(0, self.matrix\_len + 1):

                    matrix\_copy[k][j] -= coefficient \* matrix\_copy[i][j]

        vars\_values = [0] \* self.matrix\_len

        # Back substitution

        for i in range(self.matrix\_len - 1, -1, -1):

            vars\_values[i] = matrix\_copy[i][self.matrix\_len]

            for j in range(i + 1, self.matrix\_len):

                # Subtracting all values of matrixA except the coefficient of the variable

                if j != i:

                    vars\_values[i] -= matrix\_copy[i][j] \* vars\_values[j]

            # Finally, divide the rhs by the coefficient of the variable

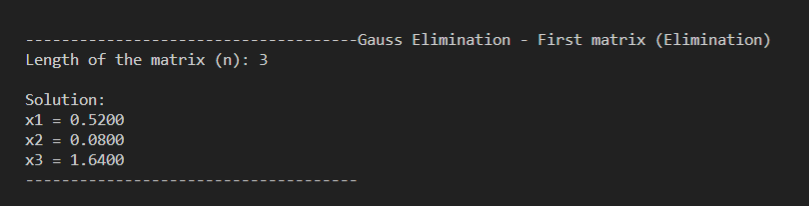
            vars\_values[i] /= matrix\_copy[i][i]

        self.vars\_values = vars\_values

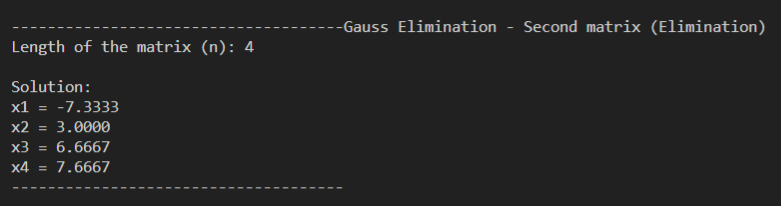
        self.matrix\_copy = matrix\_copy

        self.solution\_vectorX = vars\_values

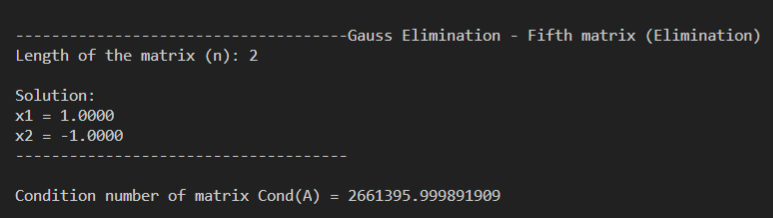
Метод 1:



Метод 2:



Метод 5:



**3)** **Индивидуальный метод: метод Гаусса-Зейделя**

class GaussSeidel:

    def \_\_init\_\_(self, equations: list, vars: list, e: float, print\_only\_results: bool = False, matrix\_name: str = "", matrixAB: list = None,

    print\_results=True, without\_print=False, max\_iterations=1000, show\_errors\_list=False, auto\_adjust\_matrix=True):

        condition = True

        # Initializing all variables values with zero if using equations

        # or set each X as B if using matrix, as the first approximation

        vars\_values = [0] \* len(equations) if matrixAB is None else [row[-1] for row in matrixAB]

        if not without\_print and not print\_only\_results:

            print(0, \*[f'{"%0.4f":<12}' % elem for elem in vars\_values], sep="\t")

        iteration = 1

        # Adjust matrix

        adjusted\_matrixAB = None

        if auto\_adjust\_matrix:

            adjusted\_matrixAB = self.adjust\_matrix(matrixAB)

            if not without\_print:

                print("(Auto adjustment of matrix is enabled)\nAdjusted matrix:\n", adjusted\_matrixAB)

        else:

            adjusted\_matrixAB = matrixAB

        while condition:

            e\_list = []

            # Calculating all variables

            if adjusted\_matrixAB is None:

                self.iterate\_equations(equations, e\_list, vars\_values)

            else:

                self.iterate\_matrix(adjusted\_matrixAB, e\_list, vars\_values)

            if not without\_print and not print\_only\_results:

                print(iteration, \*[f'{"%0.4f":<12}' % elem for elem in vars\_values], sep="\t")

            iteration += 1

            if show\_errors\_list:

                print("Errors:", \*[f'{"%0.4f":<12}' % elem for elem in e\_list], sep="\t")

            # Checking if all current errors are greater than required error e

            condition = self.check\_error\_rate(e\_list, e)

            if iteration > max\_iterations:

                break

        if not without\_print:

            print('\nSolution:')

            for (var, val) in zip(vars, vars\_values):

                print(var,'= %0.3f' %(val))

            print('-------------------------------------\n')

        self.vars\_values = vars\_values

        self.solution\_vectorX = vars\_values

    def iterate\_equations(self, equations, e\_list, vars\_values):

        # Calculating all variables

        for i, eq in enumerate(equations):

            new\_value = eq(\*vars\_values)

            e\_list.append(abs(vars\_values[i] - new\_value))

            vars\_values[i] = new\_value

    def iterate\_matrix(self, matrix, e\_list, vars\_values):

        # Calculating all variables

        for i, row in enumerate(matrix):

            # Initializing current X with B coefficient value

            new\_value = row[-1]

            for j in range(0, len(row) - 1):

                if i != j:

                    new\_value -= (row[j] \* vars\_values[j])

            new\_value \*= 1/row[i]

            e\_list.append(abs(vars\_values[i] - new\_value))

            vars\_values[i] = new\_value

    def check\_error\_rate(self, e\_list: list, e):

        return all([current\_e > e for current\_e in e\_list])

    def adjust\_matrix(self, matrixAB):

        if matrixAB is None:

            return None

        out\_matrix = copy.deepcopy(matrixAB)

        for i, row in enumerate(out\_matrix):

            current\_max = 0

            current\_max\_index = 0

            # Iterating over the i-th column values

            for j in range(i, len(out\_matrix)):

                if abs(current\_max) < abs(out\_matrix[j][i]):

                    current\_max = out\_matrix[j][i]

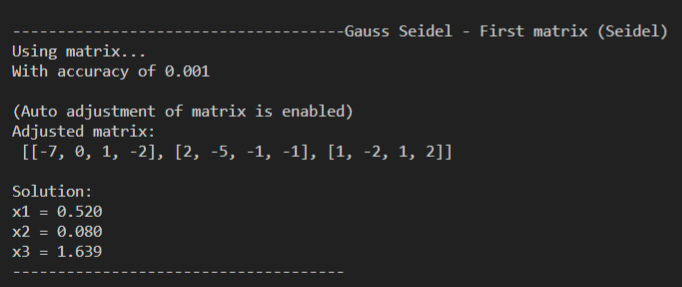
                    current\_max\_index = j

            # Swap max row with the current row

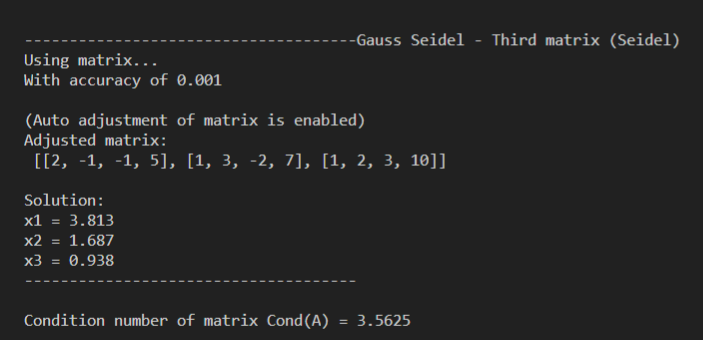
            out\_matrix[i], out\_matrix[current\_max\_index] = out\_matrix[current\_max\_index], out\_matrix[i]

        return out\_matrix

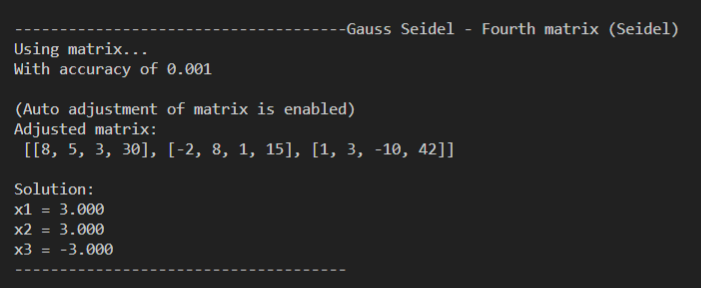
Метод 1:



Метод 3:



Метод 4:



**4) Экспериментальное определение числа обусловленности матрицы**

class ConditionNumber:

    def \_\_init\_\_(self, matrixAB=None, matrixA=None, vectorB=None, print\_result=False):

        self.whole\_matrix = matrixAB

        if matrixAB is None:

            self.matrixA = matrixA

            self.vectorB = vectorB

        else:

            self.matrixA = mh.get\_matrixA(matrixAB)

            self.vectorB = mh.get\_vectorB\_unpacked(matrixAB)

        # Get the condition number of the matrix

        if matrixAB is None:

            self.cond = mh.get\_matrix\_cond(matrixA)

        else:

            self.cond = mh.get\_matrix\_cond(mh.get\_matrixA(matrixAB))

    def experimental(self, vectorX, delta\_vectorX, printall=False):

        # Get the norms of the vectorB and delta\_vectorB

        vectorB\_norm = mh.get\_vector\_norm\_euc(self.vectorB)

        delta\_vectorB = mh.get\_deltaB\_from\_vectorB(self.vectorB)

        deltaB\_norm = mh.get\_vector\_norm\_euc(delta\_vectorB)

        # Get the norms of the vectorX and delta\_vectorX

        vectorX\_norm = mh.get\_vector\_norm\_euc(vectorX)

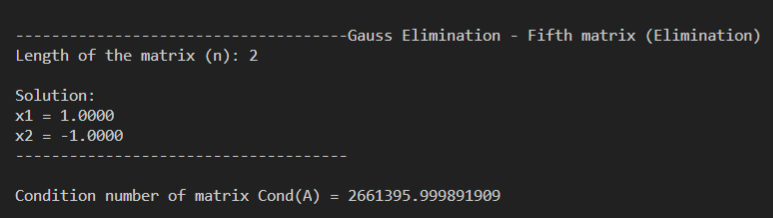
        deltaX\_norm = mh.get\_vector\_norm\_euc(delta\_vectorX)

        condition\_1 = self.cond >= (deltaX\_norm / vectorX\_norm) \* (vectorB\_norm / deltaB\_norm)

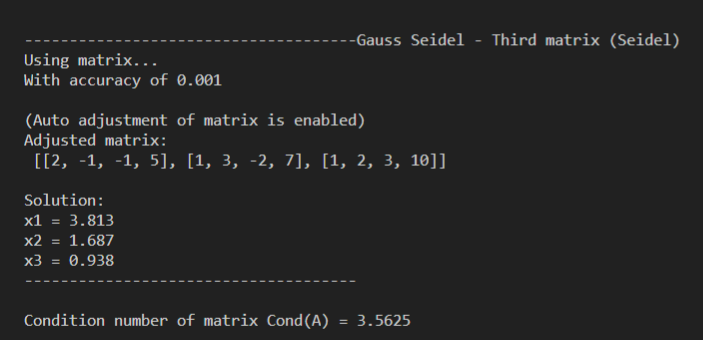
        condition\_2 = (deltaX\_norm / vectorX\_norm) <= self.cond \* (deltaB\_norm / vectorB\_norm)

        return condition\_1, condition\_2

Метод 5 (исключение Гауссом):

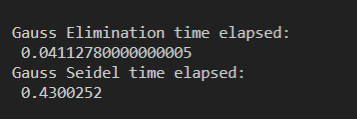


Метод 3 (Гаусса-Зейделя):



**5) Вывод**

Сравнивая метод исключения Гаусса с ведущим элементом и метод Гаусса-Зейделя для вычисления системы 2 из задания можно заметить, что метод исключения Гаусса с выбором ведущего элемента заметно быстрее.



Метод Гаусса имеет сложность О(n3), что означает быструю скорость решения для небольших матриц, но с увеличением размеров матриц, данный метод становится неоптимальным.

Так же если рассматривать расчёт числа обусловленности матрицы, то величина погрешности у систем 3 незначительна, меньше 10. Это означает что cond(A) хорошо обусловлена. У системы 5 огромное значение, больше 1000. Это значит, что она плохо обусловлена.