МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»**

Кафедра теоретических основ компьютерной безопасности и криптографии

**Методы решения системы линейных алгебраических уравнений**

ОТЧЕТ ПО ДИСЦИПЛИНЕ

«ЧИСЛЕННЫЕ МЕТОДЫ РЕШЕНИЯ ЗАДАЧ»

студента 4 курса 431 группы

специальности 10.05.01 Компьютерная безопасность

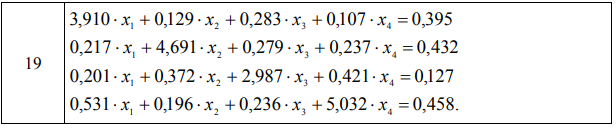
факультета компьютерных наук и информационных технологий

Сенокосова Владислава Владимировича

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
| Преподаватель  Аспирант | \_\_\_\_\_\_\_\_\_\_ | В.M.Шкатов |
|  | подпись, дата |  |

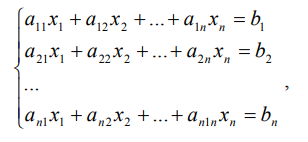
Саратов 2024

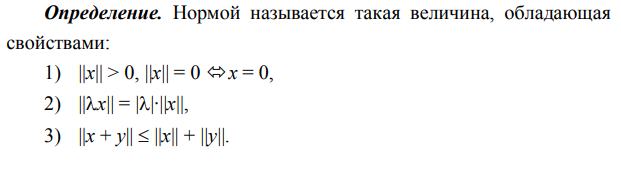
**Вариант 19**

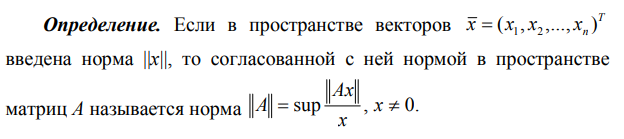


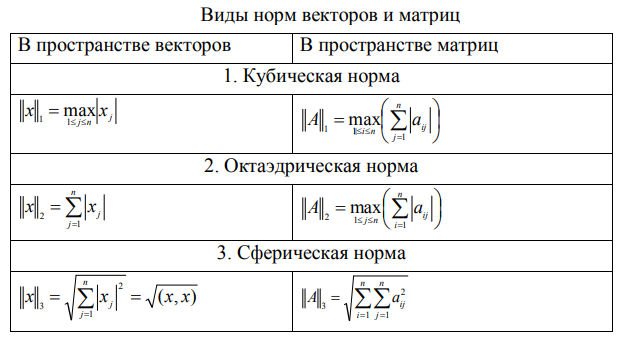
В процессе выполнения лабораторной работы необходимо реализовать 3 программы, позволяющие решать системы линейных уравнений.

Будем рассматривать системы уравнений вида:



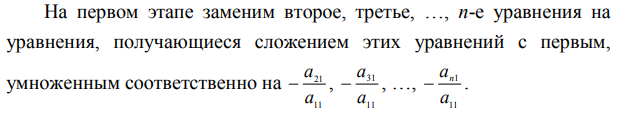


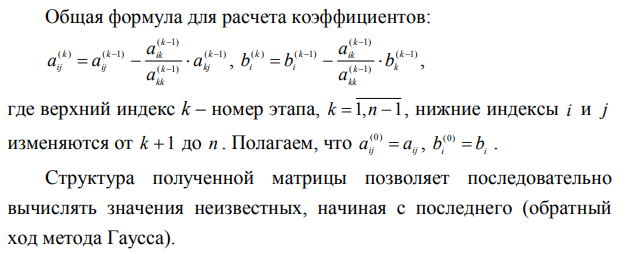


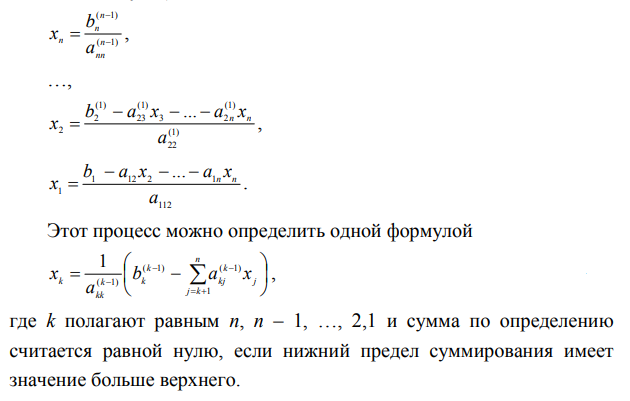


**Задание 1: «Решение системы линейных уравнений методом Гаусса»**

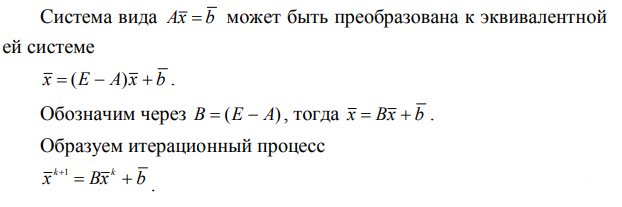
Один из методов решения системы уравнений – метод Гаусса. Суть метода Гаусса заключается в приведении исходной матрицы к треугольному виду. Будем постоянно приводить систему к треугольному виду, исключая последовательно сначала х1 из второго, третьего, …, -го уравнений, затем 2 из третьего, четвертого, …, -го уравнений преобразованной системы и т. д.

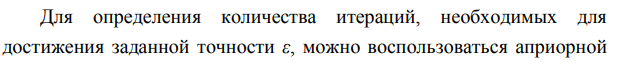


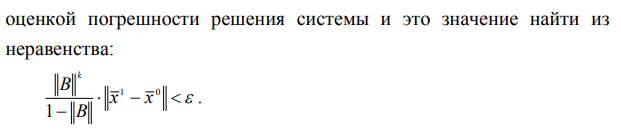


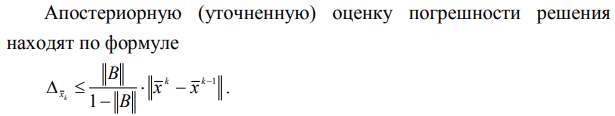


**Задание 2: «Решение системы линейных уравнений методом итераций»**

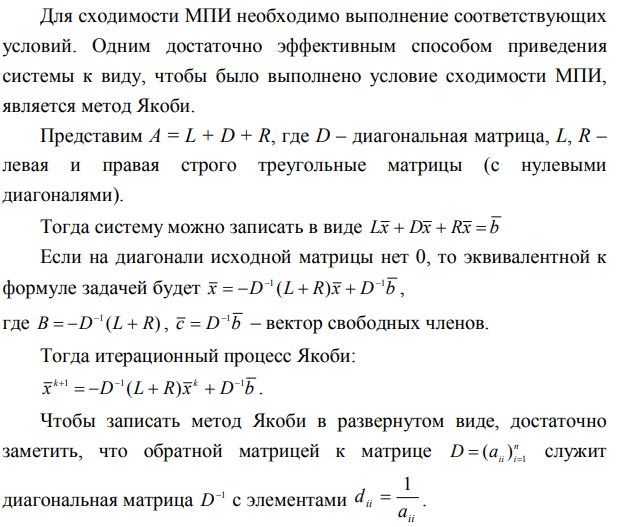
****

****

****

****

**Задание 3: «Решение линейных уравнений методом Якоби»**

****

**Программная реализация метода Гауса:**

Получение корней осуществляется с помощью следующей функции, которая сначала приводит матрицу системы уравнений к треугольному виду, а после по описанным выше формулам высчитывает корни. Также были использованы функции самостоятельно реализованные для умножения вектора на число и сложения векторов:

def gauss\_method(matrix):

*# Приведение к треугольному виду*

    for i in range(len(matrix)):

        for j in range(i + 1, len(matrix)):

            mul = mul\_vector\_to\_num(matrix[i], - matrix[j][i] / matrix[i][i])

            matrix[j] = add\_vectors(mul, matrix[j])

    roots = []

*# Получение корней системы уранений*

    for i in range(len(matrix) - 1, -1, -1):

        count = 0

        k = -1

        for j in range(i + 1, len(matrix)):

            count += matrix[i][j] \* roots[k]

            k -= 1

        roots.append(round((matrix[i][-1] - count) / matrix[i][i], 4))

    roots.reverse()

    return roots

Тестирование программы осуществлялось на двух системах линейных уравнений (из примера и варианта 19):

*# Система уранений из 19 варианта*

    matrix\_1 = [[3.910, 0.129, 0.283, 0.107, 0.395],

                [0.217, 4.691, 0.279, 0.237, 0.432],

                [0.201, 0.371, 2.987, 0.421, 0.127],

                [0.531, 0.196, 0.236, 5.032, 0.458]]

*# Система уравнений из примера*

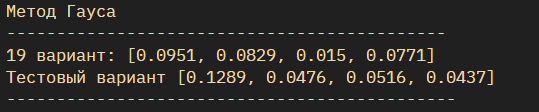
    matrix\_2 = [[5.526, 0.305, 0.887, 0.037, 0.774],

                [0.658, 2.453, 0.678, 0.192, 0.245],

                [0.398, 0.232, 4.957, 0.567, 0.343],

                [0.081, 0.521, 0.192, 4.988, 0.263]]

Результаты тестирования, следующие:



В результате ответ для тестового варианта полностью совпадает, с тем решением, которое представлено в методическом пособии.

**Программная реализация метода итераций:**

Для использования метода итераций необходимо осуществить, начальные преобразования над системой линейных уравнений, которые будут использоваться в дальнейшем. Для этого была реализована следующая функция:

def taransform\_for\_iteration(matrix):

    vec\_c = []

    matrix\_B = []

    for i in range(len(matrix)):

        vec = div\_vector\_to\_num(matrix[i], matrix[i][i])

        matrix\_B.append(vec[:-1])

        matrix\_B[i] = mul\_vector\_to\_num(matrix\_B[i], -1)

        matrix\_B[i][i] = 0

        vec\_c.append(vec[-1])

    if cubic\_norm(matrix\_B, system=False) < 1:

        return vec\_c, matrix\_B

    return None

Также для вычисления нормы матрицы системы уравнений использовалась следующая функция:

def cubic\_norm(matrix, system=True, vec=False):

    if vec:

        return max(matrix)

    max\_a = 0

    for row in range(len(matrix)):

        if system:

            sum\_row = abs(sum(matrix[row][:-1]))

        else:

            sum\_row = abs(sum(matrix[row]))

        if sum\_row > max\_a:

            max\_a = sum\_row

    if system:

        max\_b = 0

        for row in range(len(matrix)):

            if matrix[row][-1] > max\_b:

                max\_b = matrix[row][-1]

        return max\_a, max\_b

    return max\_a

Поиск реверсированной матрицы осуществлялся с помощью следующей функции:

def reverse\_matrix(matrix):

    res = []

    len\_m = len(matrix)

    for j in range(len(matrix[0])):

        buff = []

        for i in range(len\_m):

            buff.append(matrix[i][j])

        res.append(buff)

    return res

Умножение матриц и сложение матриц:

def mul\_matrix(matrix1, matrix2):

    res\_matrix = []

    for i in range(len(matrix1)):

        row = []

        for k in range(len(matrix2[0])):

            count = 0

            for j in range(len(matrix2)):

                count += matrix1[i][j] \* matrix2[j][k]

            row.append(round(count, 4))

        res\_matrix.append(row)

    return res\_matrix

def add\_matrix(matrix1, matrix2):

    res\_matrix = []

    for i in range(len(matrix1)):

        row = []

        for j in range(len(matrix1[i])):

            row.append(round(matrix1[i][j] + matrix2[i][j], 4))

        res\_matrix.append(row)

    return res\_matrix

Сам алгоритм вычисления корней с помощью метода итераций представлен представлен следующей функцией:

def iteration\_method(matrix):

    eps = 0.01

    val = taransform\_for\_iteration(matrix)

    if val != None:

        c, b = val

    norm\_c = cubic\_norm(c, system=False, vec=True)

    norm\_b = cubic\_norm(b, system=False)

    count\_interation =  ceil(log(round(eps \* (1 - norm\_b) / norm\_c, 4)) / log(norm\_b))

    c = reverse\_matrix([c])

    main\_c = deepcopy(c)

    arr\_steps = [c]

    for \_ in range(count\_interation):

        m = mul\_matrix(b, c)

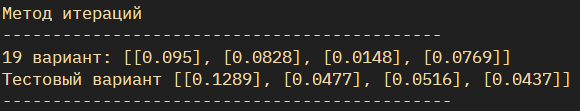
        buff = add\_matrix(m, main\_c)

        arr\_steps.append(buff)

        c = buff

    return arr\_steps[-1]

Результаты тестирования, следующие:



**Программная реализация метода Якоби:**

Для использования метода Якоби решения системы линейных уравнений необходимо сначала преобразовать входную систему, осуществляется это с помощью следующей функции:

def transform\_for\_jkobi(A):

    D, L, R, B, b = [], [], [], [], []

    for i in range(len(A)):

        row\_D, row\_L, row\_R = [], [], []

        b.append(A[i][-1])

        for j in range(len(A)):

            if i == j:

                row\_D.append(A[i][j])

                row\_L.append(0)

                row\_R.append(0)

            if j > i:

                row\_D.append(0)

                row\_L.append(0)

                row\_R.append(A[i][j])

            if j < i:

                row\_D.append(0)

                row\_L.append(A[i][j])

                row\_R.append(0)

        D.append(row\_D)

        L.append(row\_L)

        R.append(row\_R)

    rev\_D = []

    for i in range(len(D)):

        row\_rev\_D = []

        for j in range(len(D)):

            if i == j:

                row\_rev\_D.append(round(1 / D[i][j], 4))

            else:

                row\_rev\_D.append(0)

        rev\_D.append(row\_rev\_D)

    for row in mul\_matrix(rev\_D, add\_matrix(L, R)):

        B.append(mul\_vector\_to\_num(row, -1))

    b = reverse\_matrix([b])

    c = mul\_matrix(rev\_D, b)

    return D, L, R, B, b, c

В этой функции выводится множество различных матриц, которые можно представить в виде указанных выше формул.

Сам алгоритм вычисления корней методом Якоби представлен в следующей функцией с заранее заданными погрешностями:

def jkobi\_method(A):

*# Получаем преобразованные матрицы*

    D, L, R, B, b, c = transform\_for\_jkobi(A)

*# Проверим сходмость Якоби*

    for i in range(len(A)):

        sum\_elem = 0

        for j in range(len(A)):

            if i != j:

               sum\_elem += A[i][j]

        if A[i][i] < sum\_elem:

            print("Нарушена сходимость")

            break

*# Вычисляем корни*

    roots = []

    eps = 0.001

    start = reverse\_matrix([[1 for \_ in range(len(A))]])

    while True:

        copy\_start = deepcopy(start)

        start = add\_matrix(mul\_matrix(B, start), c)

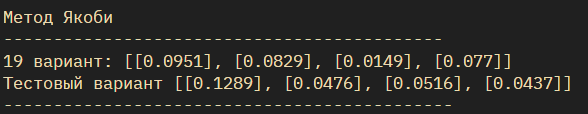
        roots.append(start)

        if copy\_start == start:

            break

    return roots[-1]

Результаты тестирования программы:



Для того чтобы проверить правильность найденных корней, воспользуемся функцией подстановки корней в систему, которой на вход поступает система и вектор столбец корней:

def check\_roots(system, vector):

    for row in range(len(system)):

        row\_res = 0

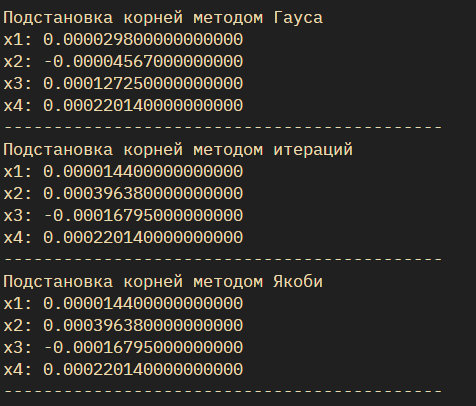
        for index1 in range(len(system)):

            row\_res += system[row][index1] \* vector[index1][0]

        row\_res -= system[row][index1 + 1]

        print(f"x{row + 1}:", "{:0<20.15f}".format(row\_res))

Результаты подстановки:



То есть корни верны с точностью до заданной погрешности.

Для вычисления абсолютной и относительных погрешностей использовались функции для нахождения обратной матрицы к системе и вычисления норм векторов и матриц:

def get\_matrix\_system\_reverse(matrix: list[list]) -> list[list]:

    res\_matrix = []

    for i in range(len(matrix)):

        matrix\_copy = deepcopy(matrix)

        for j in range(len(matrix)):

            if i == j:

                matrix\_copy[j][-1] = 1

            else:

                matrix\_copy[j][-1] = 0

        res\_matrix.append(gauss\_method(matrix\_copy))

*# Создаем из исходной системы матрицу (те исключаем свободные члены)*

    print("Исходная матрица:")

    m = [i[:len(i) - 1] for i in matrix]

    print\_matrix(m)

    print("Обратная матрица:")

    print\_matrix(reverse\_matrix(res\_matrix))

*# A \* A^-1 = E*

    print("Результат их перемножения:")

    print\_matrix(mul\_matrix(m, reverse\_matrix(res\_matrix)))

    return reverse\_matrix(res\_matrix)

def cubic\_norm(matrix: list[list], system=True, vec=False) -> float:

    if vec:

        return max(matrix)

    max\_a = 0

    for row in range(len(matrix)):

        if system:

            sum\_row = abs(sum(matrix[row][:-1]))

        else:

            sum\_row = abs(sum(matrix[row]))

        if sum\_row > max\_a:

            max\_a = sum\_row

    if system:

        max\_b = 0

        for row in range(len(matrix)):

            if matrix[row][-1] > max\_b:

                max\_b = matrix[row][-1]

        return max\_a, max\_b

    return max\_a

def abs\_and\_rel\_err(matrix: list[list], reversed\_matrix: list[list]) -> tuple[float, float]:

    delta = 0.001

    norm\_a, norm\_b = cubic\_norm(matrix)

    norm\_rev\_a = cubic\_norm(reversed\_matrix, system=False)

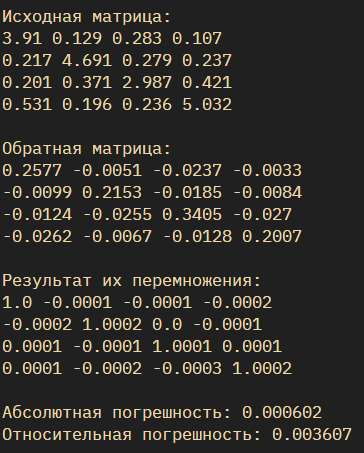
    abs\_err\_1 = delta / norm\_b

    abs\_err\_2 = abs\_err\_1 \* norm\_rev\_a

    rel\_err = norm\_a \* norm\_rev\_a \* abs\_err\_1

    return (round(abs\_err\_2, 6), round(rel\_err, 6))

В результате выполнения функции были получены следующие результаты, в которых можно заметить получение единичной матрицы, что свидетельствует о правильно найденной обратной матрицы:



**Выводы**

В результате проделанной работы можно сделать вывод, что все алгоритмы дают схожие результаты с заранее определённой погрешностью. В чем мы убедились путем проверки найденных значений.

**ПРИЛОЖЕНИЕ**

**Листинг программы**

from copy import deepcopy

from math import log, ceil

def print\_matrix(matrix: list[list]) -> print:

    for row in matrix:

        print(\*row)

    print()

def add\_vector\_to\_num(vec: list, num: int) -> list:

    new\_vec = []

    for elem in vec:

        new\_vec.append(elem + num)

    return new\_vec

def add\_vectors(vec1, vec2 : list) -> list:

    res\_vec = []

    for index in range(len(vec1)):

        res\_vec.append(round(vec1[index] + vec2[index], 4))

    return res\_vec

def mul\_vector\_to\_num(vec: list, num: int) -> list:

    res\_vec = []

    for index in range(len(vec)):

        if vec[index] != 0:

            res\_vec.append(vec[index] \* num)

        else:

            res\_vec.append(vec[index])

    return res\_vec

def div\_vector\_to\_num(vec: list, num: int) -> list:

    res\_vec = []

    for index in range(len(vec)):

        res\_vec.append(round(vec[index] / num, 4))

    return res\_vec

def reverse\_matrix(matrix: list[list]) -> list[list]:

    res = []

    len\_m = len(matrix)

    for j in range(len(matrix[0])):

        buff = []

        for i in range(len\_m):

            buff.append(matrix[i][j])

        res.append(buff)

    return res

def gauss\_method(matrix: list[list]) -> float:

*# Приведение к треугольному виду*

    for i in range(len(matrix)):

        for j in range(i + 1, len(matrix)):

            mul = mul\_vector\_to\_num(matrix[i], - matrix[j][i] / matrix[i][i])

            matrix[j] = add\_vectors(mul, matrix[j])

    roots = []

*# Получение корней системы уранений*

    for i in range(len(matrix) - 1, -1, -1):

        count = 0

        k = -1

        for j in range(i + 1, len(matrix)):

            count += matrix[i][j] \* roots[k]

            k -= 1

        roots.append(round((matrix[i][-1] - count) / matrix[i][i], 4))

    roots.reverse()

    return roots

def get\_matrix\_system\_reverse(matrix: list[list]) -> list[list]:

    res\_matrix = []

    for i in range(len(matrix)):

        matrix\_copy = deepcopy(matrix)

        for j in range(len(matrix)):

            if i == j:

                matrix\_copy[j][-1] = 1

            else:

                matrix\_copy[j][-1] = 0

        res\_matrix.append(gauss\_method(matrix\_copy))

*# Создаем из исходной системы матрицу (те исключаем свободные члены)*

    print("Исходная матрица:")

    m = [i[:len(i) - 1] for i in matrix]

    print\_matrix(m)

    print("Обратная матрица:")

    print\_matrix(reverse\_matrix(res\_matrix))

*# A \* A^-1 = E*

    print("Результат их перемножения:")

    print\_matrix(mul\_matrix(m, reverse\_matrix(res\_matrix)))

    return reverse\_matrix(res\_matrix)

def cubic\_norm(matrix: list[list], system=True, vec=False) -> float:

    if vec:

        return max(matrix)

    max\_a = 0

    for row in range(len(matrix)):

        if system:

            sum\_row = abs(sum(matrix[row][:-1]))

        else:

            sum\_row = abs(sum(matrix[row]))

        if sum\_row > max\_a:

            max\_a = sum\_row

    if system:

        max\_b = 0

        for row in range(len(matrix)):

            if matrix[row][-1] > max\_b:

                max\_b = matrix[row][-1]

        return max\_a, max\_b

    return max\_a

def abs\_and\_rel\_err(matrix: list[list], reversed\_matrix: list[list]) -> tuple[float, float]:

    delta = 0.001

    norm\_a, norm\_b = cubic\_norm(matrix)

    norm\_rev\_a = cubic\_norm(reversed\_matrix, system=False)

    abs\_err\_1 = delta / norm\_b

    abs\_err\_2 = abs\_err\_1 \* norm\_rev\_a

    rel\_err = norm\_a \* norm\_rev\_a \* abs\_err\_1

    return (round(abs\_err\_2, 6), round(rel\_err, 6))

def taransform\_for\_iteration(matrix: list[list]) -> tuple[list, list[list]]:

    vec\_c = []

    matrix\_B = []

    for i in range(len(matrix)):

        vec = div\_vector\_to\_num(matrix[i], matrix[i][i])

        matrix\_B.append(vec[:-1])

        matrix\_B[i] = mul\_vector\_to\_num(matrix\_B[i], -1)

        matrix\_B[i][i] = 0

        vec\_c.append(vec[-1])

    if cubic\_norm(matrix\_B, system=False) < 1:

        return vec\_c, matrix\_B

    return None

def mul\_matrix(matrix1, matrix2: list[list]) -> list[list]:

    res\_matrix = []

    for i in range(len(matrix1)):

        row = []

        for k in range(len(matrix2[0])):

            count = 0

            for j in range(len(matrix2)):

                count += matrix1[i][j] \* matrix2[j][k]

            row.append(round(count, 4))

        res\_matrix.append(row)

    return res\_matrix

def add\_matrix(matrix1, matrix2: list[list]) -> list[list]:

    res\_matrix = []

    for i in range(len(matrix1)):

        row = []

        for j in range(len(matrix1[i])):

            row.append(round(matrix1[i][j] + matrix2[i][j], 4))

        res\_matrix.append(row)

    return res\_matrix

def iteration\_method(matrix: list[list]) -> list[list]:

    eps = 0.01

    val = taransform\_for\_iteration(matrix)

    if val != None:

        c, b = val

    norm\_c = cubic\_norm(c, system=False, vec=True)

    norm\_b = cubic\_norm(b, system=False)

    count\_interation =  ceil(log(round(eps \* (1 - norm\_b) / norm\_c, 4)) / log(norm\_b))

    c = reverse\_matrix([c])

    main\_c = deepcopy(c)

    arr\_steps = [c]

    for \_ in range(count\_interation):

        m = mul\_matrix(b, c)

        buff = add\_matrix(m, main\_c)

        arr\_steps.append(buff)

        c = buff

    return arr\_steps[-1]

def transform\_for\_jkobi(A: list[list]):

    D, L, R, B, b = [], [], [], [], []

    for i in range(len(A)):

        row\_D, row\_L, row\_R = [], [], []

        b.append(A[i][-1])

        for j in range(len(A)):

            if i == j:

                row\_D.append(A[i][j])

                row\_L.append(0)

                row\_R.append(0)

            if j > i:

                row\_D.append(0)

                row\_L.append(0)

                row\_R.append(A[i][j])

            if j < i:

                row\_D.append(0)

                row\_L.append(A[i][j])

                row\_R.append(0)

        D.append(row\_D)

        L.append(row\_L)

        R.append(row\_R)

    rev\_D = []

    for i in range(len(D)):

        row\_rev\_D = []

        for j in range(len(D)):

            if i == j:

                row\_rev\_D.append(round(1 / D[i][j], 4))

            else:

                row\_rev\_D.append(0)

        rev\_D.append(row\_rev\_D)

    for row in mul\_matrix(rev\_D, add\_matrix(L, R)):

        B.append(mul\_vector\_to\_num(row, -1))

    b = reverse\_matrix([b])

    c = mul\_matrix(rev\_D, b)

    return D, L, R, B, b, c

def jkobi\_method(A: list[list]):

*# Получаем преобразованные матрицы*

    D, L, R, B, b, c = transform\_for\_jkobi(A)

*# Проверим сходмость Якоби*

    for i in range(len(A)):

        sum\_elem = 0

        for j in range(len(A)):

            if i != j:

               sum\_elem += A[i][j]

        if A[i][i] < sum\_elem:

            print("Нарушена сходимость")

            break

*# Вычисляем корни*

    roots = []

    eps = 0.001

    start = reverse\_matrix([[1 for \_ in range(len(A))]])

*# a = abs(cubic\_norm(start, system=False) - cubic\_norm(c, system=False))*

*# aa = round(((1 - cubic\_norm(B, system=False)) / cubic\_norm(B, system=False)) \* eps, 4)*

    while True:

        copy\_start = deepcopy(start)

        start = add\_matrix(mul\_matrix(B, start), c)

        roots.append(start)

        if copy\_start == start:

            break

    return roots[-1]

def check\_roots(system, vector):

    for row in range(len(system)):

        row\_res = 0

        for index1 in range(len(system)):

            row\_res += system[row][index1] \* vector[index1][0]

        row\_res -= system[row][index1 + 1]

        print(f"x{row + 1}:", "{:0<20.15f}".format(row\_res))

if \_\_name\_\_ == "\_\_main\_\_":

*# Тестовые примеры*

*# Система уранений из 19 варианта*

    matrix\_1 = [[3.910, 0.129, 0.283, 0.107, 0.395],

                [0.217, 4.691, 0.279, 0.237, 0.432],

                [0.201, 0.371, 2.987, 0.421, 0.127],

                [0.531, 0.196, 0.236, 5.032, 0.458]]

*# Система уравнений из примера*

    matrix\_2 = [[5.526, 0.305, 0.887, 0.037, 0.774],

                [0.658, 2.453, 0.678, 0.192, 0.245],

                [0.398, 0.232, 4.957, 0.567, 0.343],

                [0.081, 0.521, 0.192, 4.988, 0.263]]

*# Система уранений из 16 варианта*

    matrix\_16 = [[2.923, 0.220, 0.159, 0.328, 0.605],

                 [0.363, 4.123, 0.268, 0.327, 0.496],

                 [0.169, 0.271, 3.906, 0.295, 0.590],

                 [0.241, 0.319, 0.257, 3.862, 0.896]]

*# Тестовые матрицы для тестирования операций*

    matrix\_3 = [[1, 2, 2],

                [3, 1, 2],

                [3, 3, 1]]

    matrix\_4 = [[0, ],

                [2, ],

                [0, ]]

    matrix\_5 = [[1, 2, 2]]

    matrix\_6 = [[0, 2, 0]]

*# Создаем копии систем уравнений для тестирвоания*

    copy\_matrix\_1 = deepcopy(matrix\_1)

    copy\_matrix\_2 = deepcopy(matrix\_1)

    copy\_matrix\_3 = deepcopy(matrix\_1)

    copy\_matrix\_21 = deepcopy(matrix\_2)

    copy\_matrix\_22 = deepcopy(matrix\_2)

    copy\_matrix\_23 = deepcopy(matrix\_2)

    copy\_matrix\_31 = deepcopy(matrix\_16)

    copy\_matrix\_32 = deepcopy(matrix\_16)

    copy\_matrix\_33 = deepcopy(matrix\_16)

    copy\_matrix\_41 = deepcopy(matrix\_1)

*# Решение системы линейных уравнений методом Гауса*

*#-------------------------------------------------*

    print("Метод Гауса")

    print("--------------------------------------------")

    print("19 вариант:", gauss\_method(copy\_matrix\_1))

    print("16 вариант:", gauss\_method(copy\_matrix\_31))

    print("Тестовый вариант",gauss\_method(copy\_matrix\_21))

    print("---------------------------------------------")

    print(reverse\_matrix([gauss\_method(copy\_matrix\_31)]))

*#-------------------------------------------------*

*# Решение системы линейных уранений методом Итераций*

*#-------------------------------------------------*

    print("Метод итераций")

    print("--------------------------------------------")

    print("19 вариант:", iteration\_method(copy\_matrix\_2))

    print("16 вариант:", iteration\_method(copy\_matrix\_32))

    print("Тестовый вариант",iteration\_method(copy\_matrix\_22))

    print("---------------------------------------------")

*#-------------------------------------------------*

*# Решение системы линейных уранений методом Якоби*

*#-------------------------------------------------*

    print("Метод Якоби")

    print("--------------------------------------------")

    print("19 вариант:", jkobi\_method(copy\_matrix\_3))

    print("16 вариант:", jkobi\_method(copy\_matrix\_33))

    print("Тестовый вариант",jkobi\_method(copy\_matrix\_23))

    print("---------------------------------------------")

*#-------------------------------------------------*

*# Проверка найденных корней для системы уравнений*

*#-------------------------------------------------*

    print("Подстановка корней методом Гауса")

    check\_roots(matrix\_1, reverse\_matrix([gauss\_method(matrix\_1)]))

    print("--------------------------------------------")

    print("Подстановка корней методом итераций")

    check\_roots(matrix\_1, iteration\_method(matrix\_1))

    print("--------------------------------------------")

    print("Подстановка корней методом Якоби")

    check\_roots(matrix\_1, jkobi\_method(matrix\_1))

    print("--------------------------------------------")

*#-------------------------------------------------*

*# Высчитывание абсолютной и относительной погрешности*

*#-------------------------------------------------*

    reversed\_system = get\_matrix\_system\_reverse(copy\_matrix\_41)

    abs\_, rel\_ = abs\_and\_rel\_err(copy\_matrix\_41, reversed\_system)

    print(f"Абсолютная погрешность: {abs\_}")

    print(f"Относительная погрешность: {rel\_}")