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.М.Шкатов

подпись, дата

Саратов 2024

Вариант 19

19	$3{,}910 \cdot x_1 + 0{,}129 \cdot x_2 + 0{,}283 \cdot x_3 + 0{,}107 \cdot x_4 = 0{,}395$ $0{,}217 \cdot x_1 + 4{,}691 \cdot x_2 + 0{,}279 \cdot x_3 + 0{,}237 \cdot x_4 = 0{,}432$ $0{,}201 \cdot x_1 + 0{,}372 \cdot x_2 + 2{,}987 \cdot x_3 + 0{,}421 \cdot x_4 = 0{,}127$ $0{,}531 \cdot x_1 + 0{,}196 \cdot x_2 + 0{,}236 \cdot x_3 + 5{,}032 \cdot x_4 = 0{,}458.$

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

Будем рассматривать системы уравнений вида:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{n1n}x_n = b_n \end{cases},$$

Определение. Нормой называется такая величина, обладающая свойствами:

1) $\|x\| > 0$, $\|x\| = 0 \Leftrightarrow x = 0$,
2) $\|\lambda x\| = |\lambda| \cdot \|x\|$,
3) $\|x + y\| \le \|x\| + \|y\|$.

Определение. Если в пространстве векторов $\bar{x} = (x_1, x_2, \ldots, x_n)^T$ введена норма $\|x\|$, то согласованной с ней нормой в пространстве матриц A называется норма $\|A\| = \sup \frac{\|Ax\|}{x}$, $x \ne 0$.

Виды норм векторов и матриц

В пространстве векторов	В пространстве матриц
1. Кубическая норма	
$\|x\|_1 = \max\limits_{1\le j\le n} \lvert x_j \rvert$	$\|A\|_1 = \max\limits_{1\le i\le n}\left(\sum\limits_{j=1}^{n} \lvert a_{ij} \rvert\right)$
2. Октаэдрическая норма	
$\|x\|_2 = \sum\limits_{j=1}^{n} \lvert x_j \rvert$	$\|A\|_2 = \max\limits_{1\le j\le n}\left(\sum\limits_{i=1}^{n} \lvert a_{ij} \rvert\right)$
3. Сферическая норма	
$\|x\|_3 = \sqrt{\sum\limits_{j=1}^{n} \lvert x_j \rvert^2} = \sqrt{(x,x)}$	$\|A\|_3 = \sqrt{\sum\limits_{i=1}^{n}\sum\limits_{j=1}^{n} a_{ij}^2}$

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 x_1 из второго, третьего, …, n-го уравнений, затем x_2 из третьего, четвертого, …, n-го уравнений преобразованной системы и т. д.

На первом этапе заменим второе, третье, …, n-е уравнения на уравнения, получающиеся сложением этих уравнений с первым, умноженным соответственно на $-\frac{a_{21}}{a_{11}}, -\frac{a_{31}}{a_{11}}, \ldots, -\frac{a_{n1}}{a_{11}}$.

Общая формула для расчета коэффициентов:

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}} \cdot a_{kj}^{(k-1)}, \quad b_i^{(k)} = b_i^{(k-1)} - \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}} \cdot b_k^{(k-1)},$$

где верхний индекс k – номер этапа, $k = \overline{1, n-1}$, нижние индексы i и j изменяются от $k+1$ до n. Полагаем, что $a_{ij}^{(0)} = a_{ij}$, $b_i^{(0)} = b_i$.

Структура полученной матрицы позволяет последовательно вычислять значения неизвестных, начиная с последнего (обратный ход метода Гаусса).

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}},$$

$\ldots,$

$$x_2 = \frac{b_2^{(1)} - a_{23}^{(1)} x_3 - \ldots - a_{2n}^{(1)} x_n}{a_{22}^{(1)}},$$

$$x_1 = \frac{b_1 - a_{12} x_2 - \ldots - a_{1n} x_n}{a_{112}}.$$

Этот процесс можно определить одной формулой

$$x_k = \frac{1}{a_{kk}^{(k-1)}} \left(b_k^{(k-1)} - \sum_{j=k+1}^{n} a_{kj}^{(k-1)} x_j \right),$$

где k полагают равным n, $n - 1$, …, 2,1 и сумма по определению считается равной нулю, если нижний предел суммирования имеет значение больше верхнего.

Задание 2: «Решение системы линейных уравнений методом итераций»

Система вида $A\overline{x} = \overline{b}$ может быть преобразована к эквивалентной ей системе

$$\overline{x} = (E - A)\overline{x} + \overline{b}.$$

Обозначим через $B = (E - A)$, тогда $\overline{x} = B\overline{x} + \overline{b}$.

Образуем итерационный процесс

$$\overline{x}^{k+1} = B\overline{x}^{k} + \overline{b}.$$

Для определения количества итераций, необходимых для достижения заданной точности ε, можно воспользоваться априорной

оценкой погрешности решения системы и это значение найти из неравенства:

$$\frac{\|B\|^k}{1 - \|B\|} \cdot \left\| \overline{x}^1 - \overline{x}^0 \right\| < \varepsilon.$$

Апостериорную (уточненную) оценку погрешности решения находят по формуле

$$\Delta_{\bar{x}_k} \le \frac{\|B\|}{1-\|B\|} \cdot \|\bar{x}^k - \bar{x}^{k-1}\|.$$

Задание 3: «Решение линейных уравнений методом Якоби»

Для сходимости МПИ необходимо выполнение соответствующих условий. Одним достаточно эффективным способом приведения системы к виду, чтобы было выполнено условие сходимости МПИ, является метод Якоби.

Представим $A = L + D + R$, где D – диагональная матрица, L, R – левая и правая строго треугольные матрицы (с нулевыми диагоналями).

Тогда систему можно записать в виде $L\bar{x} + D\bar{x} + R\bar{x} = \bar{b}$

Если на диагонали исходной матрицы нет 0, то эквивалентной к формуле задачей будет $\bar{x} = -D^{-1}(L+R)\bar{x} + D^{-1}\bar{b}$,

где $B = -D^{-1}(L+R)$, $\bar{c} = D^{-1}\bar{b}$ – вектор свободных членов.

Тогда итерационный процесс Якоби:

$\bar{x}^{k+1} = -D^{-1}(L+R)\bar{x}^k + D^{-1}\bar{b}$.

Чтобы записать метод Якоби в развернутом виде, достаточно заметить, что обратной матрицей к матрице $D = (a_{ii})_{i=1}^n$ служит диагональная матрица D^{-1} с элементами $d_{ii} = \dfrac{1}{a_{ii}}$.

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:

```python
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):

```python
    # 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:

```
Метод Гауса
-----------------------------------------------
19 вариант: [0.0951, 0.0829, 0.015, 0.0771]
Тестовый вариант [0.1289, 0.0476, 0.0516, 0.0437]
-----------------------------------------------
```

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:

```
Метод итераций
---------------------------------------------
19 вариант: [[0.095], [0.0828], [0.0148], [0.0769]]
Тестовый вариант [[0.1289], [0.0477], [0.0516], [0.0437]]
---------------------------------------------
```

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:

```
Метод Якоби
----------------------------------------------
19 вариант: [[0.0951], [0.0829], [0.0149], [0.077]]
Тестовый вариант [[0.1289], [0.0476], [0.0516], [0.0437]]
----------------------------------------------
```

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:

```
Подстановка корней методом Гауса
x1: 0.000029800000000000
x2: -0.00004567000000000
x3: 0.000127250000000000
x4: 0.000220140000000000
---------------------------------------------
Подстановка корней методом итераций
x1: 0.000014400000000000
x2: 0.000396380000000000
x3: -0.00016795000000000
x4: 0.000220140000000000
---------------------------------------------
Подстановка корней методом Якоби
x1: 0.000014400000000000
x2: 0.000396380000000000
x3: -0.00016795000000000
x4: 0.000220140000000000
---------------------------------------------
```

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
```

```python
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:

```
Исходная матрица:
3.91 0.129 0.283 0.107
0.217 4.691 0.279 0.237
0.201 0.371 2.987 0.421
0.531 0.196 0.236 5.032

Обратная матрица:
0.2577 -0.0051 -0.0237 -0.0033
-0.0099 0.2153 -0.0185 -0.0084
-0.0124 -0.0255 0.3405 -0.027
-0.0262 -0.0067 -0.0128 0.2007

Результат их перемножения:
1.0 -0.0001 -0.0001 -0.0002
-0.0002 1.0002 0.0 -0.0001
0.0001 -0.0001 1.0001 0.0001
0.0001 -0.0002 -0.0003 1.0002

Абсолютная погрешность: 0.000602
Относительная погрешность: 0.003607
```

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:
```

```python
        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
```

```python
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_}")
```