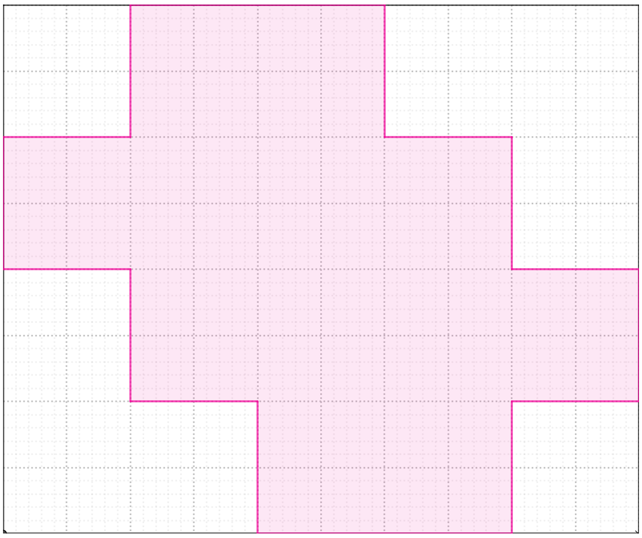
Содержание

1 Задание

Решить численно задачу Дирихле для уравнения Пуассона.

$$\begin{cases} -\Delta u = f, \; (x, y) \in \Omega, \\ u|_{\partial\Omega} = g. \end{cases}$$

Задачу необходимо решить для области $\Omega \subset \mathbb{R}$ в форме прямоугольника (с произвольными задаваемыми пользователем размерами), а также в форме, указанной в индивидуальном варианте:



Для решения СЛАУ следует использовать метод простой итерации (Якоби) и IOM(m).

Для задачи в прямоугольнике подготовить тестовые примеры, в каждом из которых необходимо выдавать норму ошибки (разности точного и вычисленного решений). В качестве тестовых, в том числе, следует использовать задачи, решением которых являются плоскости, параболоиды, комбинации тригонометрических функций.

Предусмотрите подсчет количества итераций, потребовавшихся для достижения заданной точности.

Сравните время работы двух методов при разном значении точности.

Примечание 1. Начало координат можно выбрать в любом удобном для заданной фигуры месте. Шаги сеток по обоим переменным можно взять равными.

Примечание 2. Структура данных для хранения решения (и, возможно, промежуточных переменных) должна быть оптимальной по памяти и не содержать фиктивные узлы, не входящие в заданную область.

2 Теоретический материал

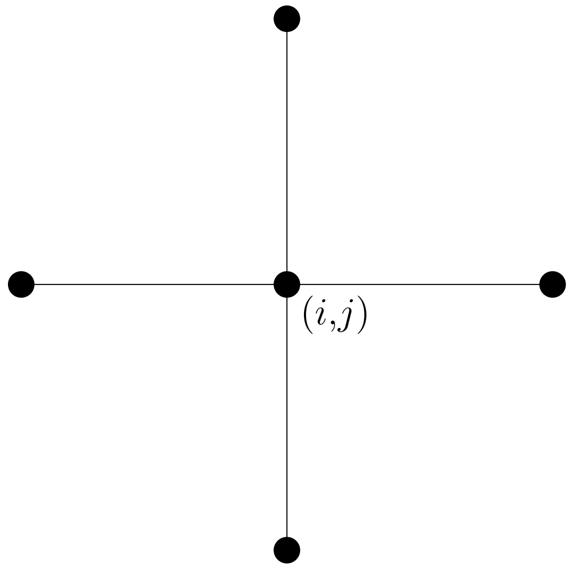
2.1 Метод Якоби

$$\frac{U_{i-1,j}^{(m)} - 2U_{i,j}^{(m+1)} + U_{i+1,j}^{(m)}}{h_1^2} + \frac{U_{i,j-1}^{(m)} - 2U_{i,j}^{(m+1)} + U_{i,j+1}^{(m)}}{h_2^2} = -F_{i,j}$$

Погрешность аппроксимации

$$\Psi = \mathcal{O}(h_1^2 + h_2^2)$$

Шаблон РС



Количество итераций

Зафиксируем произвольную точность ε, $h_1 = h_2 = h$, $\frac{1}{h} = N$

$$K(\varepsilon) \approx \frac{2\ln\left(\frac{1}{\varepsilon}\right)}{\pi^2}\left(\frac{1}{h}\right)^2 = \mathcal{O}(N^2)$$

Расчетные формулы для программирования:

$$U_{i,j}^{(m+1)} = C_0 F_{i,j} + C_1\left(U_{i,j-1}^{(m)} + U_{i,j+1}^{(m)}\right) + C_2\left(U_{i-1,j}^{(m)} + U_{i+1,j}^{(m)}\right)$$

где

$$C_0 = \frac{0.5h_1^2h_2^2}{h_1^2 + h_2^2},\ C_1 = \frac{0.5h_1^2}{h_1^2 + h_2^2},\ C_2 = \frac{0.5h_2^2}{h_1^2 + h_2^2}$$

2.2 Неполная ортогонализация с рестартами IOM(m)

Выберем в качестве подпространств $\mathcal{K}$ и $\mathcal{L}$ подпространство Крылова

$$\mathcal{K} = \mathcal{L} = \mathcal{K}_m(v_r, A),$$

для построения которого используется орт начальной невязки

$$v_1 = r_0/\|r_0\|_2.$$

Для построения базиса в $\mathcal{K}$ (и также в $\mathcal{L}$) будем использовать ортогонализацию Арнольди с начальным вектором v_1.

В соответствии с общим проекционным подходом решение должно уточняться по формуле

$$x = x_0 + Vy,$$

где y является решением системы

$$(W^T AV)y = W^T r_0,$$

в которой V и W - матрицы, составленные из базисных векторов подпространства $\mathcal{K}$ и $\mathcal{L}$ соответственно.

В нашем случае эти матрицы равны

$$V = W = V_m.$$

Следовательно,

$$W^T AV = V_m^T AV_m = H_m$$

Обозначим для краткости $\beta = \|r_0\|_2$.

$$W^T r_0 = V_m^T r_0 = V_m^T(\beta v_1) = \beta e_1,$$

где e_1 - единичный декартов орт, первая координата которого равна единице, остальные - нулю.

Таким образом, вспомогательная СЛАУ принимает вид

$$H_m y = \beta e_1.$$

Матрица H_m невырождена, а также легко обратима, поскольку имеет хессенбергову форму. Для решения системы с такой матрицей требуется занулить одну поддиагональ. Это можно сделать гауссовыми исключениями, либо ортогональными вращениями Гивенса.

Так как приходится хранить все базисные вектора пространства $\mathcal{K}$, то на это уходит много объема памяти.

Один из способов уменьшения затрат в алгоритме состоит в том, чтобы периодически обновлять алгоритм, а именно, останавливать раньше при некотором заданном значении m (как правило, небольшом). Поскольку в такой ситуации решение системы, скорее всего, найдено не будет, то процесс нужно повторять, используя в качестве нового начального приближения вектор x_m, полученный после неполного выполнения алгоритма.

Другим вариантом экономии ресурсов памяти в алгоритме является проведение неполной ортогонализации. Идея заключается в том, что очередной вектор v_{j+1} (получаемый на j-ом шаге основного цикла) делается ортогональным только к к предыдущим векторам, а не ко всем.

Алгоритм

1. Вычислить $r_0 := b - Ax_0$, $\beta := \|r_0\|_2$, $v_1 := r_0/\beta$
2. **For** $j = 1,2,\ldots,m$
3. Вычислить $w_j := Av_j$
4. **For** $i = \max\{1,j-k+1\}\ldots j$
5. $h_{i,j} := (w_j, v_i)$
6. $w_j := w_j - h_{i,j}v_i$
7. **EndFor**
8. Вычислить $h_{j+1,j} := \|w_j\|_2$
9. **If** $h_{j+1,j} = 0$ **then** положить $m := j$ и выйти из цикла **EndIf**
10. Вычислить $v_{j+1} := w_j/h_{j+1,j}$
11. **EndFor**
12. Преобразовать матрицу H (порядка m), исключив из нее поддиагональ.

Выполнить соответствующие преобразования с вектором $g = \beta e_1$

13. Решить треугольную СЛАУ $Hy = g$ порядка m
14. Вычислить $x_m = x_0 + \sum\limits_{i=1}^{m} y_i v_i$
15. **If** качество приближения удовлетворительное **then** выход
16. Положить $x_0 = x_m$, вернуться в шагу 1.

3 Тестовые задачи

3.1 Плоскость

$$\begin{cases} -\Delta u = 0, \\ u|_{\partial\Omega} = 2x + 3y - 5 \end{cases}$$

3.2 Параболоид

$$\begin{cases} -\Delta u = -2, \\ u|_{\partial\Omega} = 3x^2 - 2y^2 - 1 \end{cases}$$

3.3 Тригонометрическая функция

$$\begin{cases} -\Delta u = -e^{\sin^2 x + \cos^2 y}\left(\sin^2 2x + 2\cos 2x + \sin 2y - 2\cos 2y\right) \\ u|_{\partial\Omega} = e^{\sin^2 x + \cos^2 y} \end{cases}$$

4 Вычислительный эксперимент

4.1 Прямоугольная область

4.1.1 Метод Якоби

$h_1 = h_2 = h_0 = 0.1$, $\varepsilon_0 = 0.1$

Тестовая задача №1

Время расчета(в секундах) от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	0.0023	0.0086	0.036	0.1464
$\varepsilon_0/10$	0.0127	0.0725	0.3163	1.334
$\varepsilon_0/100$	0.0182	0.217	2.123	12.9
$\varepsilon_0/1000$	0.0274	0.3697	4.541	50.1

Число итераций от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	10	10	11	11
$\varepsilon_0/10$	50	94	95	101
$\varepsilon_0/100$	96	276	656	976
$\varepsilon_0/1000$	142	462	1400	3803

Тестовая задача №2

Время расчета(в секундах) от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	0.0018	0.0068	0.0263	0.105
$\varepsilon_0/10$	0.0069	0.0404	0.1966	0.8841
$\varepsilon_0/100$	0.0161	0.1614	1.304	6.852
$\varepsilon_0/1000$	0.026	0.3188	3.703	35.55

Число итераций от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	6	7	7	7
$\varepsilon_0/10$	33	50	58	65
$\varepsilon_0/100$	78	202	398	513
$\varepsilon_0/1000$	124	387	1100	2625

Тестовая задача №3

Время расчета(в секундах) от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	0.0086	0.0423	0.1827	0.7716
$\varepsilon_0/10$	0.0364	0.2554	1.389	6.422
$\varepsilon_0/100$	0.0993	0.9446	8.053	47.99
$\varepsilon_0/1000$	0.123	1.663	20.41	213.6

Число итераций от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	8	9	10	10
$\varepsilon_0/10$	38	61	78	89
$\varepsilon_0/100$	84	223	463	661
$\varepsilon_0/1000$	130	408	1188	2961

4.1.2 IOM(m)

$m = 90$, $k = 2$ - количество векторов, к которым происходит ортогонализациия

Тестовая задача №1

Время расчета(в секундах) от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	0.0657	0.2517	0.9379	14.4999
$\varepsilon_0/10$	0.069	0.2414	1.8968	14.4807
$\varepsilon_0/100$	0.0645	0.2288	1.8473	17.9485
$\varepsilon_0/1000$	0.0642	0.2434	1.8159	17.7229

Число итераций от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	1	1	1	4
$\varepsilon_0/10$	1	1	1	4
$\varepsilon_0/100$	1	1	2	5
$\varepsilon_0/1000$	1	1	2	5

Тестовая задача №2

Время расчета(в секундах) от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	0.0644	0.2404	0.9174	10.8944
$\varepsilon_0/10$	0.0643	0.2386	1.8933	14.3311
$\varepsilon_0/100$	0.0668	0.2391	1.8157	15.1238
$\varepsilon_0/1000$	0.0652	0.2349	1.8497	18.3193

Число итераций от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	1	1	1	3
$\varepsilon_0/10$	1	1	2	4
$\varepsilon_0/100$	1	1	2	4
$\varepsilon_0/1000$	1	1	2	5

Тестовая задача №3

Время расчета(в секундах) от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	0.0707	0.2348	1.9425	15.0457
$\varepsilon_0/10$	0.0648	0.2418	1.9311	14.8658
$\varepsilon_0/100$	0.0654	0.2288	1.8317	18.2153
$\varepsilon_0/1000$	0.0652	0.2411	1.858	22.4428

Число итераций от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	1	1	2	4
$\varepsilon_0/10$	1	1	2	4
$\varepsilon_0/100$	1	1	2	5
$\varepsilon_0/1000$	1	1	2	6

4.2 Индивидуальная область

4.2.1 Метод Якоби

$h_1 = h_2 = 0.5 = h_0, L_1 = 5, L_2 = 4, \varepsilon_0 = 0.1$

Тестовая задача №1

Время расчета(в секундах) от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	0.0071	0.0688	0.5821	6.0051
$\varepsilon_0/10$	0.0091	0.1013	0.8987	11.6856
$\varepsilon_0/100$	0.0112	0.116	1.3587	18.0
$\varepsilon_0/1000$	0.018	0.1969	1.7485	24.1388

Число итераций от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	16	53	163	490
$\varepsilon_0/10$	24	85	298	1018
$\varepsilon_0/100$	31	118	432	1561
$\varepsilon_0/1000$	39	150	566	2101

Тестовая задача №2

Время расчета(в секундах) от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	0.0051	0.0704	0.8356	9.1455
$\varepsilon_0/10$	0.0103	0.1015	1.2036	15.2191
$\varepsilon_0/100$	0.008	0.1599	1.6532	21.3458
$\varepsilon_0/1000$	0.0101	0.1979	1.9913	27.5801

Число итераций от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	21	71	241	789
$\varepsilon_0/10$	28	104	375	1332
$\varepsilon_0/100$	35	136	509	1873
$\varepsilon_0/1000$	43	169	642	2413

Тестовая задача №3

Время расчета(в секундах) от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	0.0146	0.106	1.25	11.0604
$\varepsilon_0/10$	0.0102	0.2424	2.9116	36.5327
$\varepsilon_0/100$	0.017	0.4943	4.7117	66.0623
$\varepsilon_0/1000$	0.0402	0.456	6.7337	96.749

Число итераций от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	9	27	77	193
$\varepsilon_0/10$	15	56	194	632
$\varepsilon_0/100$	23	88	324	1151
$\varepsilon_0/1000$	30	121	457	1687

4.2.2 IOM(m)

Тестовая задача №1

Время расчета(в секундах) от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	0.0783	0.1935	0.525	8.4078
$\varepsilon_0/10$	0.0868	0.1513	1.4902	10.294
$\varepsilon_0/100$	0.0764	0.189	1.5829	11.8592
$\varepsilon_0/1000$	0.1337	0.1829	1.4936	13.6049

Число итераций от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	1	1	1	5
$\varepsilon_0/10$	1	1	3	6
$\varepsilon_0/100$	1	1	3	7
$\varepsilon_0/1000$	2	1	3	8

Тестовая задача №2

Время расчета(в секундах) от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	0.1501	0.1936	3.3179	30.2108
$\varepsilon_0/10$	0.1375	0.358	3.8941	43.4151
$\varepsilon_0/100$	0.1552	0.4538	5.4603	43.4714
$\varepsilon_0/1000$	0.1048	0.5733	7.3711	92.1802

Число итераций от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	2	1	7	18
$\varepsilon_0/10$	2	2	9	26
$\varepsilon_0/100$	2	3	12	26
$\varepsilon_0/1000$	2	4	17	56

Тестовая задача №3

Время расчета(в секундах) от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	0.2309	0.2379	2.7948	49.0137
$\varepsilon_0/10$	0.1956	0.437	3.2799	54.093
$\varepsilon_0/100$	0.1995	0.4077	3.6902	72.4402
$\varepsilon_0/1000$	0.2247	0.7736	4.0087	87.2611

Число итераций от точности и шага сетки

	h_0	$h_0/2$	$h_0/4$	$h_0/8$
$\varepsilon_0/1$	3	1	6	30
$\varepsilon_0/10$	3	3	7	33
$\varepsilon_0/100$	3	3	8	44
$\varepsilon_0/1000$	4	5	9	53

5 Приложение

На прямоугольной области

```
import numpy as np
import time
import numpy.linalg as lin

L_x = 2  # размер области по x
L_y = 1  # размер области по y
h_x = 0.1  # шаг по x
h_y = 0.1  # шаг по y
eps = 10 ** (-8)  # точность решения

############################################
# ----------------Плоскость---------------#
```

```
# test = 1


# -----------------Параболоид--------------#
# test = 2


# ---------Индивидуальная функция---------#
# test = 3
############################################

# функция g
def func_u(x, y):
    match test:
        case 1:
            return 2 * x + 3 * y - 5
        case 2:
            return 3 * x ** 2 - 2 * y ** 2 - 1
        case 3:
            return np.exp((np.sin(x)) ** 2 + (np.cos(y)) ** 2)


# функция f
def func_f(x, y):
    match test:
        case 1:
            return 0
        case 2:
            return -2
        case 3:
            return -1 * np.exp(
                (np.sin(x)) ** 2 + (np.cos(y)) ** 2) * \
                   ((np.sin(2 * x)) ** 2 + 2 * np.cos(
                       2 * x) + np.sin(2 * y) -
                    2 * np.cos(2 * y))


# Метод простой итерации(Якоби)
def Cross_Jac(f, h_1, h_2, y0, ep):
    # счетчик времени
    tic = time.perf_counter()
    # коэффициенты в расчетной формуле
    C_0 = (0.5 * (h_1 ** 2) * (h_2 ** 2)) / (
```

```
            h_1 ** 2 + h_2 ** 2)
    C_1 = (0.5 * (h_1 ** 2)) / (h_1 ** 2 + h_2 ** 2)
    C_2 = (0.5 * (h_2 ** 2)) / (h_1 ** 2 + h_2 ** 2)
    # число точек разбиений по осям
    Nx = int(L_x / h_1) + 1
    Ny = int(L_y / h_2) + 1

    # матрица решений
    U = np.zeros((Ny, Nx))
    # промежуточная матрица решений
    U_m = np.zeros((Ny, Nx))

    # сетка по x
    x = np.linspace(0, L_x, Nx)
    # сетка по y
    y = np.linspace(0, L_y, Ny)

    # точное решение
    X, Y = np.meshgrid(x, y)
    Z = func_u(X, Y)

    # Заполняем начальное приближение с учетом нач.условий
    for i in range(0, Ny):
        for j in range(0, Nx):
            U[i][j] = y0(x[j], 0)
    for i in range(0, Nx):  # y = 1
        U[Ny - 1][i] = y0(x[i], L_y)
    for i in range(0, Ny):  # x = 1
        U[i][Nx - 1] = y0(L_x, y[i])
    for i in range(0, Ny):  # x = 0
        U[i][0] = y0(0, y[i])

    for i in range(0, Ny):
        for j in range(0, Nx):
            U_m[i][j] = y0(x[j], 0)
    for i in range(0, Nx):  # y = 1
        U_m[Ny - 1][i] = y0(x[i], L_y)
    for i in range(0, Ny):  # x = 1
        U_m[i][Nx - 1] = y0(L_x, y[i])
    for i in range(0, Ny):  # x = 0
        U_m[i][0] = y0(0, y[i])

    norm = 1  # значение нормы
```

```python
    k = 0  # число итераций
    while norm > ep:
        for i in range(Nx - 2, 0, -1):
            for j in range(1, Ny - 1):
                U_m[j][i] = C_0 * f(x[i], y[j]) + C_1 * (
                        U[j - 1][i] + U[j + 1][
                    i]) + C_2 * (U[j][i - 1] + U[j][
                    i + 1])
        norm = np.max(np.abs(U_m - U))
        k += 1
        for i in range(0, Ny):
            for j in range(0, Nx):
                U[i][j] = U_m[i][j]
        print(norm)

    # фиксируем счетчик времени
    toc = time.perf_counter()
    tme = round(toc - tic, 4)
    print("Время = ", tme)
    print("Погрешность с точным решением: ", lin.norm(Z - U_m))
    print("Количество итераций = ", k)
    return U_m


print("**********************Jacobi**********************")
sol = Cross_Jac(func_f, h_x, h_y, func_u, eps)


# Точное решение
def solve_of_task(U):
    trsol = np.zeros((N_x + 1, N_y + 1))
    for i in range(N_x + 1):
        for j in range(N_y + 1):
            trsol[i][j] = U(i * h_x, j * h_y)
    return trsol


# Вращения Гивенса
def givens(A, N):
    for l in range(N - 1):
        for i in range(N - 1, 0 + l, -1):
            j = i - 1
            if A[i][l] != 0:
```

```python
                alem = A[j][l]
                belem = A[i][l]
                if np.abs(belem) > np.abs(alem):
                    tau = alem / belem
                    S = 1 / np.sqrt(1 + tau ** 2)
                    C = S * tau
                else:
                    tau = belem / alem
                    C = 1 / np.sqrt(1 + tau ** 2)
                    S = C * tau
                A[i], A[j] = A[i] * C - A[j] * S, A[j] * C + A[i] * S
    return A


# обратный ход метода Гаусса
def Gauss_back_step(A, B, N):
    sol = np.zeros(N)
    for i in range(N - 1, -1, -1):
        s = 0
        if i == N - 1:
            sol[i] = B[i] / A[i][i]
        else:
            for j in range(i + 1, N, 1):
                s += A[i][j] * sol[j]
            sol[i] = (B[i] - s) / A[i][i]
    return sol


# произведение матрицы системы A на произвольный массив p
def multA(p):
    Ap = np.copy(p)
    for i in range(1, Ap.shape[0] - 1):
        for j in range(1, Ap.shape[1] - 1):
            Ap[i][j] = -1 * ((p[i - 1][j] - 2 * p[i][j] + p[i + 1][j]) / h_x ** 2 + (
                    p[i][j - 1] - 2 * p[i][j] + p[i][j + 1]) / h_y ** 2)
    return Ap


# скалярное произведение вектор-матриц
def Scr(a, b):
```

```
    sum = 0
    for i in range(a.shape[0]):
        for j in range(a.shape[1]):
            sum += a[i][j] * b[i][j]
    return sum


# число точек разбиений по осям
N_x = int(L_x / h_x)
N_y = int(L_x / h_x)
# задание правой части
B2 = np.zeros((N_x + 1, N_y + 1))
for i in range(N_y + 1):
    B2[0][i] = func_u(0 * h_x, i * h_y)
    B2[N_x][i] = func_u(N_x * h_x, i * h_y)
for i in range(N_x + 1):
    B2[i][0] = func_u(i * h_x, 0 * h_y)
    B2[i][N_y] = func_u(i * h_x, N_y * h_y)
for i in range(1, N_x):
    for j in range(1, N_y):
        B2[i][j] = func_f(i * h_x, j * h_y)


# Проекционный метод IOM(m)

def IOM_m(vec_b, m):
    solution = np.zeros((N_x + 1, N_y + 1))
    k = 1  # количество векторов, к которым будет ортогонален очередной вектор
    x0 = np.zeros((N_x + 1, N_y + 1))  # Начальное приближение
    # задание краевых значений
    for ik in range(N_y + 1):
        x0[0][ik] = func_u(0 * h_x, ik * h_y)
        x0[N_x][ik] = func_u(N_x * h_x, ik * h_y)
    for jk in range(N_x + 1):
        x0[jk][0] = func_u(jk * h_x, 0 * h_y)
        x0[jk][N_y] = func_u(jk * h_x, N_y * h_y)

    r0 = vec_b - multA(x0)  # вектор начальной невязки
    count_iter = 0
    while abs(lin.norm(r0)) > eps:
        V = np.zeros((N_x + 1, (m + 1) * (N_y + 1)))  # матрица базисных векторов из пространства K
```

```
        H = np.zeros((m + 1, m))  # матрица коэффициентов ортогонализации
        r0 = vec_b - multA(x0)  # вектор начальной невязки
        beta = lin.norm(r0)  # норма начальной невязки
        V[:, :N_y + 1] = r0 / beta  # первый базисный вектор пространства K
        for j in range(1, m + 1):
            omega_j = multA(V[:, (j - 1) * (N_y + 1): j * (N_y + 1)])  # базисный вектор пространства L
            for i in range(max(1, j - k + 1), j + 1):
                H[i - 1][j - 1] = Scr(omega_j,
                                      V[:, (i - 1) * (N_y + 1): i * (N_y + 1)])  # вычисление коэффициента орт-ции
                omega_j = omega_j - H[i - 1][j - 1] * V[:, (i - 1) * (N_y + 1): i * (
                    N_y + 1)]  # орт-ция очередного базисного вектора про-ва L
            H[j][j - 1] = lin.norm(omega_j)  # норма орт-го вектора
            if abs(H[j][j - 1]) < 10 ** (-8):
                m = j
                break
            V[:, j * (N_y + 1): (j + 1) * (N_y + 1)] = omega_j / H[j][j - 1]  # вычисление следующего вектора про-ва K
        e_1 = np.zeros(m + 1)  # орт
        e_1[0] = 1
        g = beta * e_1  # вектор правой части вспопогательной СЛАУ
        H = np.c_[H, g]  # добавление к матрице системы правой части
        H = givens(H, m + 1)  # зануляем поддиагональ вращениями Гивенса
        g = H[:m, m]  # перезаписываем измененую правую часть
        H = np.delete(np.delete(H, m, 1), m, 0)  # удаляем вектор правой части из системы
        y = Gauss_back_step(H, g, m)  # обратный ход метода Гауса
        # Уточнение решения
        sumyivi = np.zeros((N_x + 1, N_y + 1))  # уточняющий вектор
```

```
        for f in range(1, m + 1):
            sumyivi += y[f - 1] * V[:, (f - 1) * (N_y + 1): f *
             ↪ (N_y + 1)]  # вычисление уточняющего вектора
        solution = x0 + sumyivi  # уточнение
        r0 = vec_b - multA(solution)  # вычисление вектора
         ↪ начальной невязки
        x0 = solution  # изменение начального приближения
        count_iter += 1
    return solution, count_iter


print("**********************projection**********************")
ts = time.time()
lol, ver = IOM_m(B2, 100)
tf = time.time()
sol_anal = solve_of_task(func_u)
print('Время = ', tf - ts)
print('Погрешность с точным решением: = ', lin.norm(sol_anal -
 ↪ lol))
print('Количество итераций = ', ver)
```

На сложной области

```

import numpy as np
import time
import numpy.linalg as lin

dim = 1  # размер одного квадрата подобласти
L_x = 5 * dim  # размер области по x
L_y = 4 * dim  # размер области по y
h_x = 0.25  # шаг по x
h_y = 0.25  # шаг по y
eps = 0.00001  # точность

##########################################
# ----------------Плоскость---------------#
# test = 1


# ----------------Параболоид--------------#
```

```python
# test = 2


# ---------Индивидуальная функция---------#
test = 3


#############################################
# функция g
def func_u(x, y):
    match test:
        case 1:
            return 2 * x + 3 * y - 5
        case 2:
            return 3 * x ** 2 - 2 * y ** 2 - 1
        case 3:
            return np.exp((np.sin(x)) ** 2 + (np.cos(y)) ** 2)


# функция f
def func_f(x, y):
    match test:
        case 1:
            return 0 + x - x
        case 2:
            return -2 + x - x
        case 3:
            return -1 * np.exp(
                (np.sin(x)) ** 2 + (np.cos(y)) ** 2) * \
                   ((np.sin(2 * x)) ** 2 + 2 * np.cos(
                       2 * x) + np.sin(2 * y) -
                    2 * np.cos(2 * y))


# Метод простой итерации(Якоби)
def Cross_Jac(f, h_1, h_2, y0, ep):
    # коэффициенты в расчетной формуле
    C_0 = (0.5 * (h_1 ** 2) * (h_2 ** 2)) / (
            h_1 ** 2 + h_2 ** 2)
    C_1 = (0.5 * (h_2 ** 2)) / (h_1 ** 2 + h_2 ** 2)
    C_2 = (0.5 * (h_1 ** 2)) / (h_1 ** 2 + h_2 ** 2)
    # число точек разбиений по осям
    Nx = int(L_x / h_1) + 1
```

```
    Ny = int(L_y / h_2) + 1
    # сетка по x
    x = np.linspace(0, L_x, Nx)
    # сетка по y
    y = np.linspace(0, L_y, Ny)

    # 4 subareas
    # 1:
    Ny_1 = int((Ny - 1) / 4) + 1
    Nx_1 = int(2 * (Nx - 1) / 5) + 1
    U_sub_1 = np.zeros((Ny_1, Nx_1))
    # 2:
    Ny_2 = int((Ny - 1) / 4) + 1
    Nx_2 = int(4 * (Nx - 1) / 5) + 1
    U_sub_2 = np.zeros((Ny_2, Nx_2))
    # 3:
    Ny_3 = int((Ny - 1) / 4) + 1
    Nx_3 = int(4 * (Nx - 1) / 5) + 1
    U_sub_3 = np.zeros((Ny_3, Nx_3))
    # 4:
    Ny_4 = int((Ny - 1) / 4) + 1
    Nx_4 = int(2 * (Nx - 1) / 5) + 1
    U_sub_4 = np.zeros((Ny_4, Nx_4))

    # input initial values
    # First area
    for i in range(Ny_1):
        for j in range(Nx_1):
            U_sub_1[i][j] = y0(x[int((Nx - 1) / 5) + j], y[Ny - 1])
    for i in range(Ny_1):
        U_sub_1[i][0] = y0(x[int((Nx - 1) / 5)], y[Ny - 1 - i])
    for i in range(Ny_1):
        U_sub_1[i][Nx_1 - 1] = y0(x[int(3 * (Nx - 1) / 5)], y[Ny - 1 - i])

    # Second area
    for i in range(Ny_2):
        for j in range(Nx_2):
            U_sub_2[i][j] = y0(x[j], y[Ny - 1])
    for j in range(int((Nx - 1) / 5) + 1):
        U_sub_2[0][j] = y0(x[j], y[int(3 * (Ny - 1) / 4)])
    for i in range(Ny_2):
```

```python
        U_sub_2[i][0] = y0(x[0], y[int(3 * (Ny - 1) / 4) - i])
    for j in range(int((Nx - 1) / 5) + 1):
        U_sub_2[Ny_2 - 1][j] = y0(x[j], y[int(2 * (Ny - 1) / 4)])
    for j in range(int((Nx - 1) / 5) + 1):
        U_sub_2[0][j + int(3 * (Nx_2 - 1) / 4)] = y0(x[int(3 * (Nx - 1) / 5) + j], y[int(3 * (Ny - 1) / 4)])
    for i in range(int(Ny_2)):
        U_sub_2[i][Nx_2 - 1] = y0(x[int(4 * (Nx - 1) / 5)], y[int(3 * (Ny - 1) / 4) - i])

    # Third area
    for i in range(Ny_3):
        for j in range(Nx_3):
            U_sub_3[i][j] = y0(x[j + int((Nx - 1) / 5)], y[Ny - 1])
    for i in range(Ny_3):
        U_sub_3[i][0] = y0(x[int((Nx - 1) / 5)], y[int(2 * (Ny - 1) / 4) - i])
    for j in range(Ny_3):
        U_sub_3[Ny_3 - 1][j] = y0(x[int((Nx - 1) / 5) + j], y[int((Ny - 1) / 4)])
    for j in range(Ny_3):
        U_sub_3[Ny_3 - 1][j + int(3 * (Nx_3 - 1) / 4)] = y0(x[int(4 * (Nx - 1) / 5) + j], y[int((Ny - 1) / 4)])
    for i in range(Ny_3):
        U_sub_3[i][Nx_3 - 1] = y0(x[Nx - 1], y[int(2 * (Ny - 1) / 4) - i])
    for j in range(Ny_3):
        U_sub_3[0][j + int(3 * (Nx_3 - 1) / 4)] = y0(x[int(4 * (Nx - 1) / 5) + j], y[int(2 * (Ny - 1) / 4)])

    # fourth area
    for i in range(Ny_4):
        for j in range(Nx_4):
            U_sub_4[i][j] = y0(x[int(2 * (Nx - 1) / 5) + j], y[Ny - 1])
    for i in range(Ny_4):
        U_sub_4[i][0] = y0(x[int(2 * (Nx - 1) / 5)], y[int((Ny - 1) / 4) - i])
    for j in range(Nx_4):
        U_sub_4[Ny_4 - 1][j] = y0(x[int(2 * (Nx - 1) / 5) + j], y[0])
```

```python
    for i in range(Ny_4):
        U_sub_4[i][Nx_4 - 1] = y0(x[int(4 * (Nx - 1) / 5)], y[int((Ny - 1) / 4) - i])

    # обойдем с 1 по 4 области
    prev_sol1 = U_sub_1.copy()
    prev_sol2 = U_sub_2.copy()
    prev_sol3 = U_sub_3.copy()
    prev_sol4 = U_sub_4.copy()
    it = 0  # итераций
    nrm = 1
    while nrm > eps:
        # first area
        for i in range(1, Ny_1 - 1):
            for j in range(1, Nx_1 - 1):
                U_sub_1[i][j] = C_0 * f(x[j + int((Nx - 1) / 5)], y[Ny - 1 - i]) + C_1 * (
                        U_sub_1[i + 1][j] + U_sub_1[i - 1][j]) \
                                + C_2 * (U_sub_1[i][j - 1] + U_sub_1[i][j + 1])
        # border between first and second area
        for j in range(1, Nx_1 - 1):
            U_sub_1[Ny_1 - 1][j] = C_0 * f(x[j + int((Nx - 1) / 5)], y[Ny - Ny_1]) + C_1 * (
                    U_sub_2[1][int((Nx_2 - 1) / 4) + j] + U_sub_1[Ny_1 - 2][j]) \
                                   + C_2 * (U_sub_1[Ny_1 - 1][j - 1] + U_sub_1[Ny_1 - 1][j + 1])
            U_sub_2[0][int((Nx_2 - 1) / 4) + j] = U_sub_1[Ny_1 - 1][j]

        # second area
        for i in range(1, Ny_2 - 1):
            for j in range(1, Nx_2 - 1):
                U_sub_2[i][j] = C_0 * f(x[j], y[int(3 * (Ny - 1) / 4) - i]) + C_1 * (
                        U_sub_2[i + 1][j] + U_sub_2[i - 1][j]) \
                                + C_2 * (U_sub_2[i][j - 1] + U_sub_2[i][j + 1])
        # border between second and third area
        for j in range(int((Nx_2 - 1) / 4) + 1, Nx_2 - 1):
```

```
            U_sub_2[Ny_2 - 1][j] = C_0 * f(x[j], y[int(3 * (Ny - 1) / 4) - Ny_2 + 1]) + C_1 * (
                    U_sub_3[1][j - int((Nx_2 - 1) / 4)] + U_sub_2[Ny_2 - 2][j]) \
                                   + C_2 * (U_sub_2[Ny_2 - 1][j - 1] + U_sub_2[Ny_2 - 1][j + 1])
            U_sub_3[0][j - int((Nx_2 - 1) / 4)] = U_sub_2[Ny_2 - 1][j]
        # third area
        for i in range(1, Ny_3 - 1):
            for j in range(1, Nx_3 - 1):
                U_sub_3[i][j] = C_0 * f(x[int((Nx - 1) / 5) + j], y[int(2 * (Ny - 1) / 4) - i]) + C_1 * (
                        U_sub_3[i + 1][j] + U_sub_3[i - 1][j]) \
                                + C_2 * (U_sub_3[i][j - 1] + U_sub_3[i][j + 1])
        # border between third and fourth area
        for j in range(int((Nx_3 - 1) / 4) + 1, Nx_3 - int((Nx_3 - 1) / 4) - 1):
            U_sub_3[Ny_3 - 1][j] = C_0 * f(x[int((Nx - 1) / 5) + j], y[int(2 * (Ny - 1) / 4) - Ny_3 + 1]) + C_1 * (
                    U_sub_4[1][j - int((Nx_3 - 1) / 4)] + U_sub_3[Ny_3 - 2][j]) \
                                   + C_2 * (U_sub_3[Ny_3 - 1][j - 1] + U_sub_3[Ny_3 - 1][j + 1])
            U_sub_4[0][j - int((Nx_3 - 1) / 4)] = U_sub_3[Ny_3 - 1][j]
        # fourth area
        for i in range(1, Ny_4 - 1):
            for j in range(1, Nx_4 - 1):
                U_sub_4[i][j] = C_0 * f(x[int(2 * (Nx - 1) / 5) + j], y[int((Ny - 1) / 4) - i]) + C_1 * (
                        U_sub_4[i + 1][j] + U_sub_4[i - 1][j]) \
                                + C_2 * (U_sub_4[i][j - 1] + U_sub_4[i][j + 1])
        # calculate norm on all areas
        mx1 = lin.norm(np.abs(prev_sol1 - U_sub_1))
        mx2 = lin.norm(np.abs(prev_sol2 - U_sub_2))
        mx3 = lin.norm(np.abs(prev_sol3 - U_sub_3))
        mx4 = lin.norm(np.abs(prev_sol4 - U_sub_4))
```

```
        nrm = max(mx1, mx2, mx3, mx4)

        # change initial solve
        prev_sol1 = U_sub_1.copy()
        prev_sol2 = U_sub_2.copy()
        prev_sol3 = U_sub_3.copy()
        prev_sol4 = U_sub_4.copy()

        print(nrm)
        it += 1

    # r1 = lin.norm(sub1_z - U_sub_1)
    # r2 = lin.norm(sub2_z - U_sub_2)
    # r3 = lin.norm(sub3_z - U_sub_3)
    # r4 = lin.norm(sub4_z - U_sub_4)

    # r1 = lin.norm()

    # r = max(r1, r2, r3, r4)
    return nrm, it


print("**********************Jacobi***********************")
# start time
ts = time.time()
sol, pop = Cross_Jac(func_f, h_x, h_y, func_u, eps)
# finish time
tf = time.time()
print('Невязка = ', sol)
print('Итераций = ', pop)
print('Время = ', tf - ts)


# Точное решение
def solve_of_task(U):
    trsol = np.zeros((N_x + 1, N_y + 1))
    for i in range(N_x + 1):
        for j in range(N_y + 1):
            trsol[i][j] = U(i * h_x, j * h_y)
    return trsol


# Вращения Гивенса
```

```
def givens(A, N):
    for l in range(N - 1):
        for i in range(N - 1, 0 + l, -1):
            j = i - 1
            if A[i][l] != 0:
                alem = A[j][l]
                belem = A[i][l]
                if np.abs(belem) > np.abs(alem):
                    tau = alem / belem
                    S = 1 / np.sqrt(1 + tau ** 2)
                    C = S * tau
                else:
                    tau = belem / alem
                    C = 1 / np.sqrt(1 + tau ** 2)
                    S = C * tau
                A[i], A[j] = A[i] * C - A[j] * S, A[j] * C + A[i]
                 ↪  * S
    return A


# обратный ход метода Гаусса
def Gauss_back_step(A, B, N):
    sol = np.zeros(N)
    for i in range(N - 1, -1, -1):
        s = 0
        if i == N - 1:
            sol[i] = B[i] / A[i][i]
        else:
            for j in range(i + 1, N, 1):
                s += A[i][j] * sol[j]
            sol[i] = (B[i] - s) / A[i][i]
    return sol


def multA(area1, area2, area3, area4):
    # initialization mult
    mult_area1 = np.copy(area1)
    mult_area2 = np.copy(area2)
    mult_area3 = np.copy(area3)
    mult_area4 = np.copy(area4)

    # first area
    for i in range(1, mult_area1.shape[0] - 1):
```

```
        for j in range(1, mult_area1.shape[1] - 1):
            mult_area1[i][j] = -1 * ((area1[i - 1][j] - 2 * area1[i][j] + area1[i + 1][j]) / h_x ** 2 + (
                    area1[i][j - 1] - 2 * area1[i][j] + area1[i][j + 1]) / h_y ** 2)
    for j in range(1, mult_area1.shape[1] - 1):
        mult_area1[mult_area1.shape[0] - 1][j] = -1 * (
                (area1[mult_area1.shape[0] - 2][j] - 2 * area1[mult_area1.shape[0] - 1][j] + area2[1][int((area2.shape[1] - 1) / 4) + j]) / h_y ** 2 + (
                area1[mult_area1.shape[0] - 1][j - 1] - 2 * area1[mult_area1.shape[0] - 1][j] + area1[mult_area1.shape[0] - 1][j + 1]) / h_x ** 2)

    # second area
    for i in range(1, mult_area2.shape[0] - 1):
        for j in range(1, mult_area2.shape[1] - 1):
            mult_area2[i][j] = -1 * ((area2[i - 1][j] - 2 * area2[i][j] + area2[i + 1][j]) / h_x ** 2 + (
                    area2[i][j - 1] - 2 * area2[i][j] + area2[i][j + 1]) / h_y ** 2)
    for j in range(int((mult_area2.shape[1] - 1) / 4) + 1, mult_area2.shape[1] - 1):
        mult_area2[mult_area2.shape[0] - 1][j] = -1 * ((area2[mult_area2.shape[0] - 2][j] - 2 * area2[mult_area2.shape[0] - 1][j] + area3[1][
            j - int((mult_area2.shape[1] - 1) / 4)]) / h_x ** 2 + (area2[mult_area2.shape[0] - 1][j - 1] - 2 * area2[mult_area2.shape[0] - 1][j] + area2[mult_area2.shape[0] - 1][j + 1]) / h_y ** 2)
    # third area
    for i in range(1, mult_area3.shape[0] - 1):
        for j in range(1, mult_area3.shape[1] - 1):
            mult_area3[i][j] = -1 * ((area3[i - 1][j] - 2 * area3[i][j] + area3[i + 1][j]) / h_x ** 2 + (
                    area3[i][j - 1] - 2 * area3[i][j] + area3[i][j + 1]) / h_y ** 2)
    for j in range(int((mult_area3.shape[1] - 1) / 4) + 1, mult_area3.shape[1] - int((mult_area3.shape[1] - 1) / 4) - 1):
```

```
        mult_area3[mult_area3.shape[0] - 1][j] = -1 * ((area3[mult_area3.shape[0] - 2][j] - 2 * area3[mult_area3.shape[0] - 1][j] + area4[1][
            j - int((mult_area3.shape[1] - 1) / 4)]) / h_x ** 2 + (area3[mult_area3.shape[0] - 1][j - 1] - 2 * area3[mult_area3.shape[0] - 1][j] + area3[mult_area3.shape[0] - 1][j + 1]) / h_y ** 2)
    # fourth area
    for i in range(1, mult_area4.shape[0] - 1):
        for j in range(1, mult_area4.shape[1] - 1):
            mult_area4[i][j] = -1 * ((area4[i - 1][j] - 2 * area4[i][j] + area4[i + 1][j]) / h_x ** 2 + (
                    area4[i][j - 1] - 2 * area4[i][j] + area4[i][j + 1]) / h_y ** 2)

    return mult_area1, mult_area2, mult_area3, mult_area4


# calculate scalar product
def Scalar(a1, a2, a3, a4, b1, b2, b3, b4):
    sum = 0
    for i in range(a1.shape[0]):
        for j in range(a1.shape[1]):
            sum += a1[i, j] * b1[i, j]
    for i in range(a2.shape[0]):
        for j in range(a2.shape[1]):
            sum += a2[i, j] * b2[i, j]
    for i in range(a3.shape[0]):
        for j in range(a3.shape[1]):
            sum += a3[i, j] * b3[i, j]
    for i in range(a4.shape[0]):
        for j in range(a4.shape[1]):
            sum += a4[i, j] * b4[i, j]
    return sum


# calculate norm on all areas
def NNorma(r1, r2, r3, r4):
    e1 = np.linalg.norm(r1)
    e2 = np.linalg.norm(r2)
    e3 = np.linalg.norm(r3)
    e4 = np.linalg.norm(r4)
```

```
    return max(e1, e2, e3, e4)


# Проекционный метод IOM(m)

def IOM_m(m):
    # count points on axis
    Nx = int(L_x / h_x) + 1
    Ny = int(L_y / h_y) + 1
    # grid
    x = np.linspace(0, L_x, Nx)
    y = np.linspace(0, L_y, Ny)

    # initialization right part and initial approximation
    # 1:
    Ny_1 = int((Ny - 1) / 4) + 1
    Nx_1 = int(2 * (Nx - 1) / 5) + 1
    U_sub_1 = np.zeros((Ny_1, Nx_1))
    x_init_1 = U_sub_1.copy()
    # 2:
    Ny_2 = int((Ny - 1) / 4) + 1
    Nx_2 = int(4 * (Nx - 1) / 5) + 1
    U_sub_2 = np.zeros((Ny_2, Nx_2))
    x_init_2 = U_sub_2.copy()
    # 3:
    Ny_3 = int((Ny - 1) / 4) + 1
    Nx_3 = int(4 * (Nx - 1) / 5) + 1
    U_sub_3 = np.zeros((Ny_3, Nx_3))
    x_init_3 = U_sub_3.copy()
    # 4:
    Ny_4 = int((Ny - 1) / 4) + 1
    Nx_4 = int(2 * (Nx - 1) / 5) + 1
    U_sub_4 = np.zeros((Ny_4, Nx_4))
    x_init_4 = U_sub_4.copy()

    # filling in the area
    # First area
    for i in range(1, Ny_1):
        for j in range(1, Nx_1 - 1):
            U_sub_1[i][j] = func_f(x[int((Nx - 1) / 5) + j], y[Ny
             ↪   - 1 - i])
    for j in range(Nx_1):
```

```python
        U_sub_1[0][j] = func_u(x[int((Nx - 1) / 5) + j], y[Ny - 1])
        x_init_1[0][j] = U_sub_1[0][j].copy()
    for i in range(Ny_1):
        U_sub_1[i][0] = func_u(x[int((Nx - 1) / 5)], y[Ny - 1 - i])
        x_init_1[i][0] = U_sub_1[i][0].copy()
    for i in range(Ny_1):
        U_sub_1[i][Nx_1 - 1] = func_u(x[int(3 * (Nx - 1) / 5)], y[Ny - 1 - i])
        x_init_1[i][Nx_1 - 1] = U_sub_1[i][Nx_1 - 1].copy()

    # Second area
    for i in range(Ny_2):
        for j in range(1, Nx_2):
            U_sub_2[i][j] = func_f(x[j], y[int(3 * (Ny - 1) / 4) - i])
    for j in range(int((Nx - 1) / 5) + 1):
        U_sub_2[0][j] = func_u(x[j], y[int(3 * (Ny - 1) / 4)])
        x_init_2[0][j] = U_sub_2[0][j].copy()
    for i in range(Ny_2):
        U_sub_2[i][0] = func_u(x[0], y[int(3 * (Ny - 1) / 4) - i])
        x_init_2[i][0] = U_sub_2[i][0].copy()
    for j in range(int((Nx - 1) / 5) + 1):
        U_sub_2[Ny_2 - 1][j] = func_u(x[j], y[int(2 * (Ny - 1) / 4)])
        x_init_2[Ny_2 - 1][j] = U_sub_2[Ny_2 - 1][j].copy()
    for j in range(int((Nx - 1) / 5) + 1):
        U_sub_2[0][j + int(3 * (Nx_2 - 1) / 4)] = func_u(x[int(3 * (Nx - 1) / 5) + j], y[int(3 * (Ny - 1) / 4)])
        x_init_2[0][j + int(3 * (Nx_2 - 1) / 4)] = U_sub_2[0][j + int(3 * (Nx_2 - 1) / 4)].copy()
    for i in range(int(Ny_2)):
        U_sub_2[i][Nx_2 - 1] = func_u(x[int(4 * (Nx - 1) / 5)], y[int(3 * (Ny - 1) / 4) - i])
        x_init_2[i][Nx_2 - 1] = U_sub_2[i][Nx_2 - 1].copy()

    # Third area
    for i in range(Ny_3):
        for j in range(1, Nx_3 - 1):
            U_sub_3[i][j] = func_f(x[j + int((Nx - 1) / 5)], y[int(2 * (Ny - 1) / 4) - i])
```

```
    for i in range(Ny_3):
        U_sub_3[i][0] = func_u(x[int((Nx - 1) / 5)], y[int(2 * (Ny - 1) / 4) - i])
        x_init_3[i][0] = U_sub_3[i][0].copy()
    for j in range(Ny_3):
        U_sub_3[Ny_3 - 1][j] = func_u(x[int((Nx - 1) / 5) + j], y[int((Ny - 1) / 4)])
        x_init_3[Ny_3 - 1][j] = U_sub_3[Ny_3 - 1][j].copy()
    for j in range(Ny_3):
        U_sub_3[Ny_3 - 1][j + int(3 * (Nx_3 - 1) / 4)] = func_u(x[int(4 * (Nx - 1) / 5) + j], y[int((Ny - 1) / 4)])
        x_init_3[Ny_3 - 1][j + int(3 * (Nx_3 - 1) / 4)] = U_sub_3[Ny_3 - 1][j + int(3 * (Nx_3 - 1) / 4)].copy()
    for i in range(Ny_3):
        U_sub_3[i][Nx_3 - 1] = func_u(x[Nx - 1], y[int(2 * (Ny - 1) / 4) - i])
        x_init_3[i][Nx_3 - 1] = U_sub_3[i][Nx_3 - 1].copy()
    for j in range(Ny_3):
        U_sub_3[0][j + int(3 * (Nx_3 - 1) / 4)] = func_u(x[int(4 * (Nx - 1) / 5) + j], y[int(2 * (Ny - 1) / 4)])
        x_init_3[0][j + int(3 * (Nx_3 - 1) / 4)] = U_sub_3[0][j + int(3 * (Nx_3 - 1) / 4)].copy()

    # fourth area
    for i in range(Ny_4):
        for j in range(1, Nx_4 - 1):
            U_sub_4[i][j] = func_f(x[int(2 * (Nx - 1) / 5) + j], y[int((Ny - 1) / 4) - i])
    for i in range(Ny_4):
        U_sub_4[i][0] = func_u(x[int(2 * (Nx - 1) / 5)], y[int((Ny - 1) / 4) - i])
        x_init_4[i][0] = U_sub_4[i][0].copy()
    for j in range(Nx_4):
        U_sub_4[Ny_4 - 1][j] = func_u(x[int(2 * (Nx - 1) / 5) + j], y[0])
        x_init_4[Ny_4 - 1][j] = U_sub_4[Ny_4 - 1][j].copy()
    for i in range(Ny_4):
        U_sub_4[i][Nx_4 - 1] = func_u(x[int(4 * (Nx - 1) / 5)], y[int((Ny - 1) / 4) - i])
        x_init_4[i][Nx_4 - 1] = U_sub_4[i][Nx_4 - 1].copy()

```

```python
    k = 2  # количество векторов, к которым будет ортогонален очередной вектор

    mu1, mu2, mu3, mu4 = multA(x_init_1, x_init_2, x_init_3, x_init_4)

    # векторы начальных невязок
    r01 = U_sub_1 - mu1
    r02 = U_sub_2 - mu2
    r03 = U_sub_3 - mu3
    r04 = U_sub_4 - mu4

    iter = 0  # count iteration
    while NNorma(r01, r02, r03, r04) > eps:

        print(NNorma(r01, r02, r03, r04))

        # матрицы базисных векторов из пространства K
        V1 = np.zeros((Ny_1, (m + 1) * Nx_1))
        V2 = np.zeros((Ny_2, (m + 1) * Nx_2))
        V3 = np.zeros((Ny_3, (m + 1) * Nx_3))
        V4 = np.zeros((Ny_4, (m + 1) * Nx_4))

        # матрица коэффициентов ортогонализации
        H = np.zeros((m + 1, m))

        mu1, mu2, mu3, mu4 = multA(x_init_1, x_init_2, x_init_3, x_init_4)

        # векторы начальных невязок
        r01 = U_sub_1 - mu1
        r02 = U_sub_2 - mu2
        r03 = U_sub_3 - mu3
        r04 = U_sub_4 - mu4

        # нормы начальных невязок
        beta = NNorma(r01, r02, r03, r04)

        # первые базисные вектора пространства K
        V1[:, :Nx_1] = r01 / beta
        V2[:, :Nx_2] = r02 / beta
        V3[:, :Nx_3] = r03 / beta
        V4[:, :Nx_4] = r04 / beta
```

```

        for j in range(1, m + 1):
            # базисные вектора пространства L
            omega_j1, omega_j2, omega_j3, omega_j4 = multA(V1[:, (j - 1) * Nx_1: j * Nx_1], V2[:, (j - 1) * Nx_2: j * Nx_2],
                           V3[:, (j - 1) * Nx_3: j * Nx_3],
                           V4[:, (j - 1) * Nx_4: j * Nx_4])
            for i in range(max(1, j - k + 1), j + 1):
                # вычисление коэффициентов орт-ции
                H[i - 1][j - 1] = Scalar(omega_j1, omega_j2, omega_j3, omega_j4,
                                          V1[:, (j - 1) * Nx_1: j * Nx_1],
                                          V2[:, (j - 1) * Nx_2: j * Nx_2],
                                          V3[:, (j - 1) * Nx_3: j * Nx_3],
                                          V4[:, (j - 1) * Nx_4: j * Nx_4])

                # ортогонализация очередных базисных векторов про-ва L
                omega_j1 = omega_j1 - H[i - 1][j - 1] * V1[:, (i - 1) * Nx_1: i * Nx_1]
                omega_j2 = omega_j2 - H[i - 1][j - 1] * V2[:, (i - 1) * Nx_2: i * Nx_2]
                omega_j3 = omega_j3 - H[i - 1][j - 1] * V3[:, (i - 1) * Nx_3: i * Nx_3]
                omega_j4 = omega_j4 - H[i - 1][j - 1] * V4[:, (i - 1) * Nx_4: i * Nx_4]

            # норма орт-го вектора
            H[j][j - 1] = NNorma(omega_j1, omega_j2, omega_j3, omega_j4)

            if abs(H[j][j - 1]) < 10 ** (-8):
                m = j
                break

            # вычисление следующих векторов про-ва K
            V1[:, j * Nx_1: (j + 1) * Nx_1] = omega_j1 / H[j][j - 1]
```

```
        V2[:, j * Nx_2: (j + 1) * Nx_2] = omega_j2 / H[j][j - 1]
        V3[:, j * Nx_3: (j + 1) * Nx_3] = omega_j3 / H[j][j - 1]
        V4[:, j * Nx_4: (j + 1) * Nx_4] = omega_j4 / H[j][j - 1]

    e_1 = np.zeros(m + 1)  # орт
    e_1[0] = 1
    g = beta * e_1  # вектор правой части вспопогательной СЛАУ
    H = np.c_[H, g]  # добавление к матрице системы правой части
    H = givens(H, m + 1)  # зануляем поддиагональ вращениями Гивенса
    g = H[:m, m]  # перезаписываем измененую правую часть
    H = np.delete(np.delete(H, m, 1), m, 0)  # удаляем вектор правой части из системы
    y = Gauss_back_step(H, g, m)  # обратный ход метода Гауса

    # Уточнение решения
    sumyivi1 = np.zeros((Ny_1, Nx_1))  # уточняющий вектор
    sumyivi2 = np.zeros((Ny_2, Nx_2))  # уточняющий вектор
    sumyivi3 = np.zeros((Ny_3, Nx_3))  # уточняющий вектор
    sumyivi4 = np.zeros((Ny_4, Nx_4))  # уточняющий вектор

    for f in range(1, m + 1):
        sumyivi1 += y[f - 1] * V1[:, (f - 1) * Nx_1: f * Nx_1]  # вычисление уточняющего вектора
        sumyivi2 += y[f - 1] * V2[:, (f - 1) * Nx_2: f * Nx_2]  # вычисление уточняющего вектора
        sumyivi3 += y[f - 1] * V3[:, (f - 1) * Nx_3: f * Nx_3]  # вычисление уточняющего вектора
        sumyivi4 += y[f - 1] * V4[:, (f - 1) * Nx_4: f * Nx_4]  # вычисление уточняющего вектора

    solution1 = x_init_1 + sumyivi1  # уточнение
    solution2 = x_init_2 + sumyivi2  # уточнение
    solution3 = x_init_3 + sumyivi3  # уточнение
    solution4 = x_init_4 + sumyivi4  # уточнение
```

```
        musol1, musol2, musol3, musol4 = multA(solution1,
         ↪  solution2, solution3, solution4)
        r01 = U_sub_1 - musol1  # вычисление вектора невязки
        r02 = U_sub_2 - musol2  # вычисление вектора невязки
        r03 = U_sub_3 - musol3  # вычисление вектора невязки
        r04 = U_sub_4 - musol4  # вычисление вектора невязки

        # change initial solve
        x_init_1 = solution1.copy()
        x_init_2 = solution2.copy()
        x_init_3 = solution3.copy()
        x_init_4 = solution4.copy()
        iter += 1

    return NNorma(r01, r02, r03, r04), iter


print("**********************IOM(m)**********************")
# start time
ts = time.time()
nev, tr = IOM_m(30)
# finish time
tf = time.time()
print("Невязка = ", nev)
print("Итераций = ", tr)
print('Время = ', tf - ts)
```