Содержание

1 Часть A

Реализуйте указанный ниже проекционый метод решения СЛАУ и найдите решение системы $Ax = b$ из ЛР1. При этом матрица A должна храниться в памяти ЭВМ (и задаваться, например, в форме массива в программе).

1.1 Алгоритм IOM(m)

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.

1.2 Реализация IOM(m) на Python

```python
def IOM_m(matr_A, vec_b, m):
    k = 1  # количество векторов, к которым будет ортогонален очередной вектор
    x0 = np.zeros(N)  # Начальное приближение
    r0 = vec_b - np.dot(matr_A, x0)  # вектор начальной невязки
    while abs(LinAl.norm(r0)) > 10 ** (-12):
        V = np.zeros((N, m + 1))  # матрица базисных векторов из пространства K
        H = np.zeros((m + 1, m))  # матрица коэффициентов ортогонализации
        r0 = vec_b - np.dot(matr_A, x0)
        beta = LinAl.norm(r0)  # норма начальной невязки
```

```
        V[:, 0] = r0 / beta  # первый базисный вектор пространства K
        for j in range(1, m + 1):
            omega_j = np.dot(A, V[:, j - 1])  # базисный вектор пространства L
            for i in range(max(1, j - k + 1), j + 1):
                H[i - 1][j - 1] = np.dot(np.transpose(omega_j), V[:, i - 1])  # вычисление коэффициента орт-ции
                omega_j = omega_j - H[i - 1][j - 1] * V[:, i - 1]  # орт-ция очередного базисного вектора про-ва L
            H[j][j - 1] = LinAl.norm(omega_j)  # норма орт-го вектора
            if abs(H[j][j - 1]) < 10 ** (-8):
                m = j
                break
            V[:, j] = 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)  # уточняющий вектор
        for i in range(m):
            sumyivi += y[i] * V[:, i]  # вычисление уточняющего вектора
        solution = x0 + sumyivi  # уточнение
        r0 = vec_b - np.dot(A, solution)    # вычисление вектора начальной невязки
        x0 = solution  # изменение начального приближения
    return solution
```

1.3 Решение системы проекционным методом

$$A_{4\times 4} = \begin{pmatrix} 4 & 1 & 0.5 & 0 \\ 1 & 4 & 1 & 0.5 \\ 0.5 & 1 & 4 & 1 \\ 0 & 0.5 & 1 & 4 \end{pmatrix}, x = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}, b = \begin{pmatrix} 4 \\ 1.5 \\ 1.5 \\ 4 \end{pmatrix} \tag{1}$$

$x =$[1.00000000e+00 5.85306487e-15 5.83677963e-15 1.00000000e+00]
$\|r\|_2 =$1.4023659425114338e-13

2 Часть Б

В реализованной процедуре проекционого метода измените реализацию матрично-векторного умножения на случай матрицы системы $(*)$ (см. приложение).

Для задачи $(*)$ (см. приложение) необходимо иметь возможность задавать произвольные размеры прямоугольной области Ω и разные (не обязательно равные) шаги h_x и h_y. Функции $f(x,y)$ и $g(x,y)$ следует задавать в коде программы (и изменять с последующей перекомпиляцией).

Расчет следует провести для СЛАУ как можно большего порядка и предусмотреть фиксацию времени, затраченного непосредственно на решение системы.

Решение $\{u_{i\,j}\}$ визуализируйте в виде двумерного поля температур.

2.1 Реализация IOM(m) на Python. Матрично-векторный подход.

```python
# 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 / Pe * ((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)
            if v1_func(i, j) > 0:
                Ap[i][j] += v1_func(i, j) * (p[i][j] - p[i - 1][j]) / h_x
            else:
```

```python
                Ap[i][j] += v1_func(i, j) * (p[i + 1][j] - p[i][j]) / h_x
            if v2_func(i, j) > 0:
                Ap[i][j] += v2_func(i, j) * (p[i][j] - p[i][j - 1]) / h_y
            else:
                Ap[i][j] += v2_func(i, j) * (p[i][j + 1] - p[i][j]) / h_y
    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


# задание правой части
B2 = np.zeros((Nx + 1, Ny + 1))
for i in range(Ny + 1):
    B2[0][i] = g_func(0, i)
    B2[Nx][i] = g_func(Nx, i)
for i in range(Nx + 1):
    B2[i][0] = g_func(i, 0)
    B2[i][Ny] = g_func(i, Ny)
for i in range(1, Nx):
    for j in range(1, Ny):
        B2[i][j] = f_func(i, j)


def IOM_m(vec_b, m):
    solution = np.zeros((Nx + 1, Ny + 1))
    k = 1  # количество векторов, к которым будет ортогонален очередной вектор
    x0 = np.zeros((Nx + 1, Ny + 1))  # Начальное приближение
    # задание краевых значений
    for ik in range(Ny + 1):
        x0[0][ik] = g_func(0, ik)
        x0[Nx][ik] = g_func(Nx, ik)
    for jk in range(Nx + 1):
```

```
            x0[jk][0] = g_func(jk, 0)
            x0[jk][Ny] = g_func(jk, Ny)
    r0 = vec_b - multA(x0)  # вектор начальной невязки
    while abs(LA.norm(r0)) > 10 ** (-8):
        V = np.zeros((Nx + 1, (m + 1) * (Ny + 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 = LA.norm(r0)  # норма начальной невязки
        V[:, :Ny + 1] = r0 / beta  # первый базисный вектор пространства K
        for j in range(1, m + 1):
            omega_j = multA(V[:, (j - 1) * (Ny + 1): j * (Ny + 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) * (Ny + 1): i * (Ny + 1)])  # вычисление коэффициента орт-ции
                omega_j = omega_j - H[i - 1][j - 1] * V[:, (i - 1) * (Ny + 1): i * (
                          Ny + 1)]  # орт-ция очередного базисного вектора про-ва L
            H[j][j - 1] = LA.norm(omega_j)  # норма орт-го вектора
            if abs(H[j][j - 1]) < 10 ** (-8):
                m = j
                break
            V[:, j * (Ny + 1): (j + 1) * (Ny + 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 Гаусса
        # Уточнение решения
        sumyivi = np.zeros((Nx + 1, Ny + 1))  # уточняющий вектор
        for f in range(1, m + 1):
            sumyivi += y[f - 1] * V[:, (f - 1) * (Ny + 1): f * (Ny + 1)]  # вычисление уточняющего вектора
        solution = x0 + sumyivi  # уточнение
        r0 = vec_b - multA(solution)  # вычисление вектора начальной невязки
        x0 = solution  # изменение начального приближения
    return solution, r0
```

2.2 Вычислительные эксперименты

2.2.1 Тесты. Параболоид.

$h_x = 0.1$, $h_y = 0.1$
$Nx = 20$, $Ny = 20$

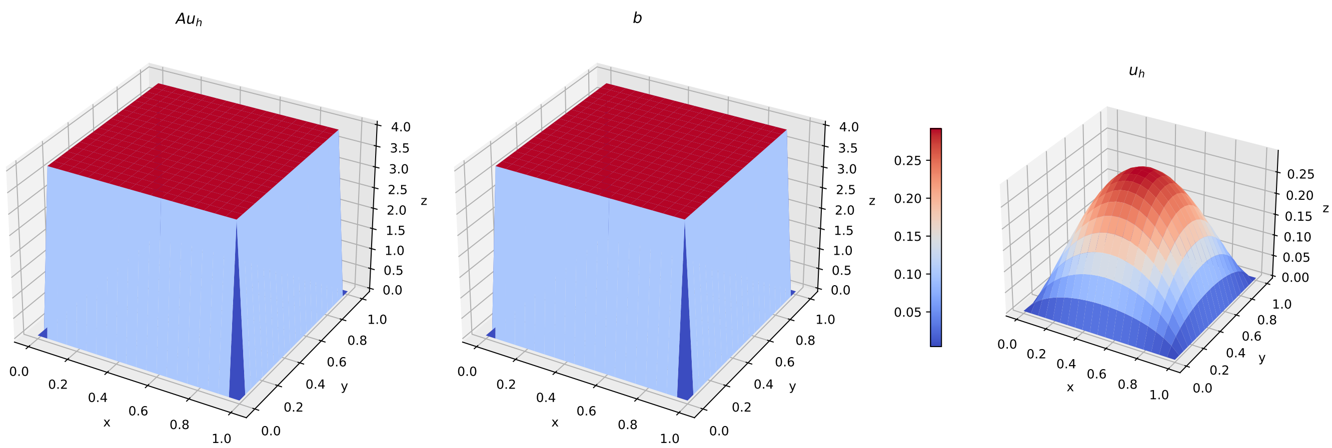
$Pe = 1$

$u = 1 - (x-1)^2 - (y-2)^2$

$g = 0$

$f = 4$

$v_1 = 0$, $v_2 = 0$



$\|r\|_2 \approx 9.23 \cdot 10^{-9}$
$time \approx 2.22$

2.2.2 Тесты. Квадратично-линейная функция.

$h_x = 0.1,\ h_y = 0.05$
$Nx = 50,\ Ny = 40$

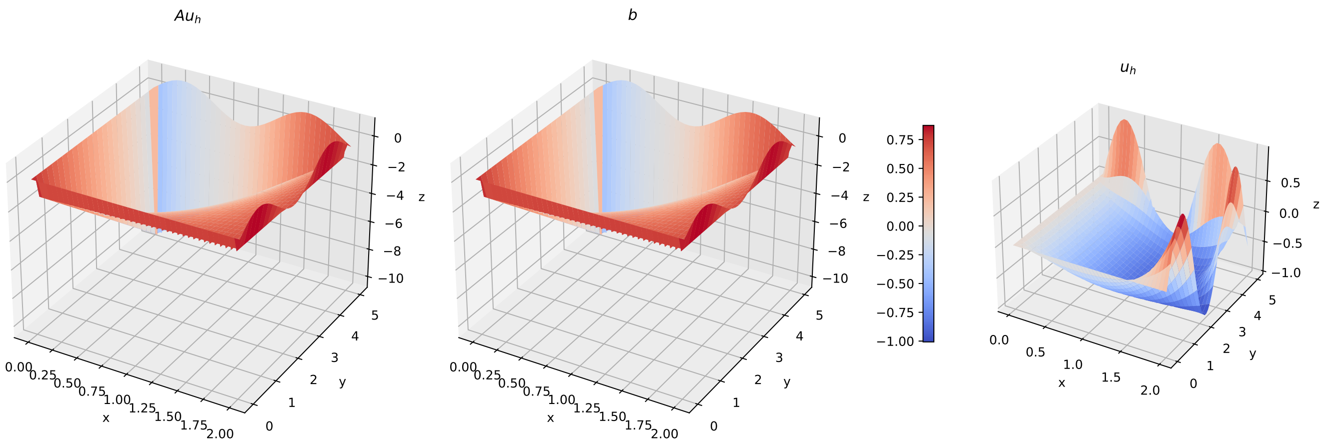
$Pe = 1$

$u = \frac{1}{2}x^2 - \frac{1}{2}y$

$g = \sin(xy)$

$f = -1 + xy - 2x$

$v_1 = y - 1,\ v_2 = 2x$



$\|r\|_2 \approx 8.8 \cdot 10^{-9}$
$time \approx 24.73$

2.2.3 Тесты. Индивидуальная функция.

$h_x = 0.1,\ h_y = 0.05$
$Nx = 50,\ Ny = 50$

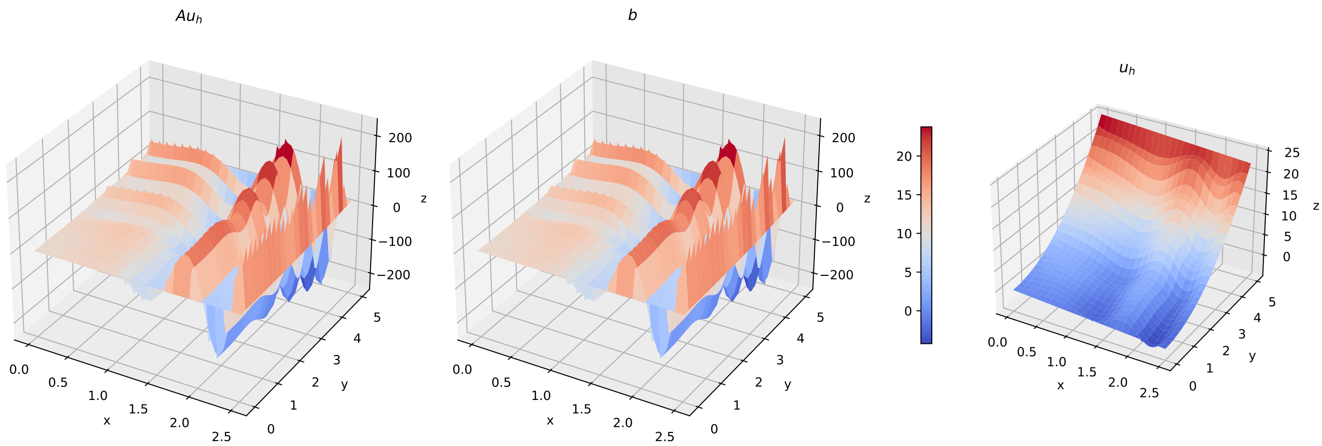
$Pe = 1$

$u = \sin(x^2) + 2\cos(2y^2)$

$g = x^2 - y$

$f = 4x^2\sin(x^2) - 2\cos(x^2) + 8\sin(2y^2) + 32y^2\cos(2y^2)$

$v_1 = 4y\sin(2y^2),\ v_2 = x\cos(x^2)$



$\|r\|_2 \approx 8.92 \cdot 10^{-9}$
$time \approx 173.3$

3 Приложение

Программа к части A.

```python
import numpy as np
import numpy.linalg as LinAl
import pandas as pd
import matplotlib as plt

num_list_group = 3  # номер в списке группы
N = 4  # размерность системы
K = N  # полуширина ленты
A = np.zeros((N, N))  # матрица A системы
elmin = max(N, 10)  # элемент главной диагонали
# формируем матрицу A
for i in range(N):
    for j in range(N):
        if i == j:
            A[i][j] = elmin
        if 0 <= i + j <= N - 1 and 1 <= j <= K:
            A[i][i + j] = 1 / j
        if 0 <= i - j <= N - 1 and 1 <= j <= K:
            A[i][i - j] = 1 / j
x = np.zeros(N)  # вектор решения
# формируем вектор решения
for i in range(N):
    if i == 0 or i == N - 1 or (i == num_list_group and 1 <=
    ↪ num_list_group < N - 1) or (
```

```python
            i == (N - 1) - 2 * num_list_group and 1 <= (N - 1) - 2 * num_list_group < N - 1):
        x[i] = 1
B = np.zeros(N)  # вектор правой части
# формируем вектор правой части
B = np.dot(A, x)


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


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 IOM_m(matr_A, vec_b, m):
```

```
    k = 1  # количество векторов, к которым будет ортогонален очередной вектор
    x0 = np.zeros(N)  # Начальное приближение
    r0 = vec_b - np.dot(matr_A, x0)  # вектор начальной невязки
    while abs(LinAl.norm(r0)) > 10 ** (-12):
        V = np.zeros((N, m + 1))  # матрица базисных векторов из пространства K
        H = np.zeros((m + 1, m))  # матрица коэффициентов ортогонализации
        r0 = vec_b - np.dot(matr_A, x0)
        beta = LinAl.norm(r0)  # норма начальной невязки
        V[:, 0] = r0 / beta  # первый базисный вектор пространства K
        for j in range(1, m + 1):
            omega_j = np.dot(A, V[:, j - 1])  # базисный вектор пространства L
            for i in range(max(1, j - k + 1), j + 1):
                H[i - 1][j - 1] = np.dot(np.transpose(omega_j), V[:, i - 1])  # вычисление коэффициента орт-ции
                omega_j = omega_j - H[i - 1][j - 1] * V[:, i - 1]  # орт-ция очередного базисного вектора про-ва L
            H[j][j - 1] = LinAl.norm(omega_j)  # норма орт-го вектора
            if abs(H[j][j - 1]) < 10 ** (-8):
                m = j
                break
            V[:, j] = 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)  # уточняющий вектор
        for i in range(m):
            sumyivi += y[i] * V[:, i]  # вычисление уточняющего
             ↪  вектора
        solution = x0 + sumyivi  # уточнение
        r0 = vec_b - np.dot(A, solution)    # вычисление вектора
         ↪  начальной невязки
        x0 = solution  # изменение начального приближения
    return solution, r0


sol, nv = IOM_m(A, B, 2)
print('Решение - ', sol)
print('Невязка - ', LinAl.norm(nv))
```

Программа к части Б.

```
import numpy as np
import numpy.linalg as LA
import matplotlib.pyplot as plt
from matplotlib import cm
import time

h_x = 0.1
h_y = 0.1
Nx = 40
Ny = 40
Pe = 1

# ------------------------
# Тест №1 - параболоид
# test = 1
# Тест №2 - квадратично-линейная функция
# test = 2
# Тест №3 - Индивидульаная функция
test = 3
# ------------------------


def u_func(i, j):
    x = i * h_x
```

```
    y = j * h_y
    match test:
        case 1:
            return 1 - ((x - 1) ** 2) - ((y - 2) ** 2)
        case 2:
            return 0.5 * x ** 2 - 0.5 * y
        case 3:
            return np.sin(x ** 2) + 2 * np.cos(2 * y ** 2)


def g_func(i, j):
    x = i * h_x
    y = j * h_y
    match test:
        case 1:
            return 0
        case 2:
            return np.sin(x * y)
        case 3:
            return x ** 2 - y


def f_func(i, j):
    x = i * h_x
    y = j * h_y
    match test:
        case 1:
            return 4
        case 2:
            return -1 + x * y - 2 * x
        case 3:
            return 4 * x ** 2 * np.sin(x ** 2) - 2 * np.cos(x ** 2) + 8 * np.sin(2 * y ** 2) + 32 * y ** 2 * np.cos(
                2 * y ** 2)


def v1_func(i, j):
    x = i * h_x
    y = j * h_y
    match test:
        case 1:
            return 0
```

```
        case 2:
            return y - 1
        case 3:
            return 4 * y * np.sin(2 * y ** 2)


def v2_func(i, j):
    x = i * h_x
    y = j * h_y
    match test:
        case 1:
            return 0
        case 2:
            return 2 * x
        case 3:
            return x * np.cos(x ** 2)


def solve_of_task(U):
    trsol = np.zeros((Nx + 1, Ny + 1))
    for i in range(1, Nx):
        for j in range(1, Ny):
            trsol[i][j] = U(i, j)
    return trsol


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


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 / Pe * ((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)
            if v1_func(i, j) > 0:
                Ap[i][j] += v1_func(i, j) * (p[i][j] - p[i -
                 ↪  1][j]) / h_x
            else:
                Ap[i][j] += v1_func(i, j) * (p[i + 1][j] -
                 ↪  p[i][j]) / h_x
            if v2_func(i, j) > 0:
                Ap[i][j] += v2_func(i, j) * (p[i][j] - p[i][j -
                 ↪  1]) / h_y
            else:
                Ap[i][j] += v2_func(i, j) * (p[i][j + 1] -
                 ↪  p[i][j]) / h_y
    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


# задание правой части
B2 = np.zeros((Nx + 1, Ny + 1))
for i in range(Ny + 1):
    B2[0][i] = g_func(0, i)
    B2[Nx][i] = g_func(Nx, i)
for i in range(Nx + 1):
    B2[i][0] = g_func(i, 0)
    B2[i][Ny] = g_func(i, Ny)
for i in range(1, Nx):
    for j in range(1, Ny):
        B2[i][j] = f_func(i, j)


def IOM_m(vec_b, m):
    solution = np.zeros((Nx + 1, Ny + 1))
    k = 1  # количество векторов, к которым будет ортогонален очередной вектор
    x0 = np.zeros((Nx + 1, Ny + 1))  # Начальное приближение
    # задание краевых значений
    for ik in range(Ny + 1):
        x0[0][ik] = g_func(0, ik)
        x0[Nx][ik] = g_func(Nx, ik)
    for jk in range(Nx + 1):
        x0[jk][0] = g_func(jk, 0)
        x0[jk][Ny] = g_func(jk, Ny)
    r0 = vec_b - multA(x0)  # вектор начальной невязки
    while abs(LA.norm(r0)) > 10 ** (-8):
        V = np.zeros((Nx + 1, (m + 1) * (Ny + 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 = LA.norm(r0)  # норма начальной невязки
        V[:, :Ny + 1] = r0 / beta  # первый базисный вектор пространства K
        for j in range(1, m + 1):
```

```
        omega_j = multA(V[:, (j - 1) * (Ny + 1): j * (Ny + 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) * (Ny + 1): i * (Ny + 1)])  # вычисление коэффициента орт-ции
            omega_j = omega_j - H[i - 1][j - 1] * V[:, (i - 1) * (Ny + 1): i * (
                        Ny + 1)]  # орт-ция очередного базисного вектора про-ва L
        H[j][j - 1] = LA.norm(omega_j)  # норма орт-го вектора
        if abs(H[j][j - 1]) < 10 ** (-8):
            m = j
            break
        V[:, j * (Ny + 1): (j + 1) * (Ny + 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((Nx + 1, Ny + 1))  # уточняющий вектор
    for f in range(1, m + 1):
        sumyivi += y[f - 1] * V[:, (f - 1) * (Ny + 1): f * (Ny + 1)]  # вычисление уточняющего вектора
    solution = x0 + sumyivi  # уточнение
    r0 = vec_b - multA(solution)  # вычисление вектора начальной невязки
    x0 = solution  # изменение начального приближения
```

```
            print(LA.norm(r0))
        return solution, r0


ts = time.time()
lol, nev = IOM_m(B2, 3)
tf = time.time()
print('Время = ', tf - ts)
print('Невязка = ', LA.norm(nev))
t = np.linspace(0, Nx * h_x, Nx + 1)
y = np.linspace(0, Ny * h_y, Ny + 1)
x_grid, y_grid = np.meshgrid(y, t)
fig = plt.figure(figsize=(15, 5))
ax = plt.subplot(131, projection='3d')
ax.set_title(r'$Au_h$')
ax.set_xlabel(r'x')
ax.set_ylabel(r'y')
ax.set_zlabel(r'z')
ax1 = plt.subplot(132, projection='3d')
ax2 = plt.subplot(133, projection='3d')
ax1.set_title(r'$b$')
ax1.set_xlabel(r'x')
ax1.set_ylabel(r'y')
ax1.set_zlabel(r'z')
surf = ax.plot_surface(x_grid, y_grid, multA(lol),
 ↪  cmap=cm.coolwarm,
                       linewidth=0, antialiased=False,
                        ↪  label=r'$\|r\|$')
surf1 = ax1.plot_surface(x_grid, y_grid, B2, cmap=cm.coolwarm,
                         linewidth=0, antialiased=False)
ax2.set_title(r'$u_h$')
ax2.set_xlabel(r'x')
ax2.set_ylabel(r'y')
ax2.set_zlabel(r'z')
surf2 = ax2.plot_surface(x_grid, y_grid, lol, cmap=cm.coolwarm,
                         linewidth=0, antialiased=False)
fig.colorbar(surf2, shrink=0.5, location='left')
plt.show()
```

Задача конвекции-диффузии

Рассмотрим классическую задачу конвекции – диффузии

$$\begin{cases} -\dfrac{1}{Pe}\left(\dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2}\right) + v_1 \dfrac{\partial u}{\partial x} + v_2 \dfrac{\partial u}{\partial y} = f, \\ u|_{\partial\Omega} = g \end{cases}$$

в прямоугольной области $\Omega = \big\{(x,y) \in \mathbb{R}^2 |\ 0 < x < X,\ 0 < y < Y\big\}$.
Здесь $v = (v_1(x,y), v_2(x,y))^T$ – заданный вектор скорости, Pe – безразмерный параметр (число Пекле). Искомой является температура $u = u(x,y)$.

Для численного решения задачи заменим область непрерывного изменения аргументов Ω конечномерной сеткой, а производные, входящие в уравнение, их разностными аппроксимациями. В результате получим систему сеточных уравнений

$$-\frac{1}{Pe}\left(\frac{u_{i-1\,j} - 2u_{i\,j} + u_{i+1\,j}}{h_x^2} + \frac{u_{i\,j-1} - 2u_{i\,j} + u_{i\,j+1}}{h_y^2}\right) +$$
$$+ v_{1i\,j} D_x u_{i\,j} + v_{2i\,j} D_y u_{i\,j} = f_{i\,j}, \qquad i = 1, \ldots, N_x - 1,\ \ j = 1, \ldots, N_y - 1. \qquad (*)$$

Для аппроксимации производных первого порядка будем использовать простейший вариант разностей против потока:

$$D_x u_{i\,j} = \begin{cases} \dfrac{u_{i\,j} - u_{i-1\,j}}{h_x} & \text{если} \quad v_{1i\,j} > 0, \\ \dfrac{u_{i+1\,j} - u_{i\,j}}{h_x} & \text{если} \quad v_{1i\,j} < 0, \end{cases}$$

$$D_y u_{i\,j} = \begin{cases} \dfrac{u_{i\,j} - u_{i\,j-1}}{h_y} & \text{если} \quad v_{2i\,j} > 0, \\ \dfrac{u_{i\,j+1} - u_{i\,j}}{h_y} & \text{если} \quad v_{2i\,j} < 0. \end{cases}$$

Система $(*)$ является системой линейных алгебраических уравнений вида

$$Au = b \qquad (**)$$

относительно неизвестных $u_{i\,j}$.