Вычислительный практикум

Шеверев Сергей

Содержание

1 Задание 1

1.1 Постановка задачи

Необходимо для заданной матрицы A решить:

- Систему $Ax = b$
- Систему с измененной правой частью $A\bar{x} = \bar{b}$

где

$$b = \begin{pmatrix} 200 \\ -600 \end{pmatrix}, \bar{b} = \begin{pmatrix} 199 \\ -601 \end{pmatrix}$$

Найти число обусловленности $cond(A)$, посчитать фактическую погрешность $\Delta x = \frac{\|\bar{x}-x\|}{\|x\|}$ и оценку погрешности.
Реализовать три метода:

- метод Гаусса с выбором главного элемента (по желанию) для решения СЛАУ
- метод Жордана для нахождения обратной матрицы
- метод LU-разложения для нахождения определителя матрицы

1.2 Вариант задания

Использованы следующие матрицы из методички (Вариант 8):

- Для первой части

$$A = \begin{pmatrix} -402.94 & 200.02 \\ 1200.12 & -600.96 \end{pmatrix}$$

- Для второй части

$$\left(\begin{array}{ccc|c} 9.62483 & 1.15527 & -2.97153 & 8.71670 \\ 1.15527 & 7.30891 & 0.69138 & 5.15541 \\ -2.97153 & 0.69138 & 5.79937 & 0.27384 \end{array}\right)$$

1.3 Решение

Для решения СЛАУ воспользуемся встроенными функциями модуля numpy.

```
import numpy as np
x = np.linalg.solve(A, b)
x_ = np.linalg.solve(A, b_)
```

После решения СЛАУ:

- $x = \begin{pmatrix} -0.50036016 \\ -0.00807482 \end{pmatrix}$
- $\bar{x} = \begin{pmatrix} -0.50157129 \\ -0.01551414 \end{pmatrix}$

Найдем число обусловленности $cond(A)$:

```
np.linalg.cond(A)
```

$cond(A) = \|A^{-1}\|\|A\| = 7.507113499431776$

Посчитаем фактическую погрешность:

```
def solve_error(matrix, dmatrix, vector, dvector):
    x = np.linalg.solve(matrix, vector)
    print("reshenie ")
    print(x)
    x_ =  np.linalg.solve(matrix + dmatrix, vector + dvector)
    print("reshenie_ ")
    print(x_)
    return  np.linalg.norm(x - x_) / np.linalg.norm(x)
    ....
    error = solve_error(a, da, b, db)
```

$$\Delta x = 0.015061709699399828$$

Посчитаем оценку:

```
def error_estimate(matrix, dmatrix, vector, dvector):
    err_est = np.linalg.cond(matrix) / (1. - np.linalg.norm(dmatrix) / np
    .linalg.norm(matrix)) *\
                     (np.linalg.norm(dvector) / np.linalg.norm(vector) +
    np.linalg.norm(dmatrix) / np.linalg.norm(matrix))
    return err_est
    ...
    estimate = error_estimate(a, da, b, db)
```

Оценка фактической погрешности равна

$$0.016786416099535777$$

Реализация метода Гаусса:

```
def Gauss(matrix,  vector):
    a, b = Gauss_forward(matrix, vector)
    return Gauss_reverse(a, b)

#Прямой ход
def Gauss_forward(matrix, vector):
    n, _ = matrix.shape
    for k in range(0, n):
        #Выбираем номер строки для перестановки
        p = np.abs(matrix[k:, k]).argmax() + k
        if p != k:
            matrix[k], matrix[p] = matrix[p].copy(), matrix[k].copy()
            vector[k], vector[p] = vector[p].copy(), vector[k].copy()
        tmp = matrix[k, k]
        matrix[k, k + 1:] /= tmp
        vector[k] /= tmp

        for i in range(k + 1, n):
            tmp = matrix[i, k]
            matrix[i, k + 1:] -= matrix[k, k + 1:] * tmp
            vector[i] -= vector[k] * tmp
```

```
    return matrix, vector

#Обратный ход
def Gauss_reverse(matrix, vector):
    n, _  = matrix.shape
    res = np.array([0 for i in range(0, n)] , float)
    for i in range(n - 1, -1, -1):
        sum = np.sum(res[i + 1:] * matrix[i, i + 1:])
        res[i] = vector[i] - sum
    return res
    ...
    x = Gauss(a.copy(), b.copy())
```

$$x = \begin{pmatrix} 0.99999979 \\ 0.49999943 \\ 0.4999991 \end{pmatrix}$$

Компоненты вектора невязки:

$$R = b - Ax = \begin{pmatrix} -1.77635684e^{-15} \\ 0.00000000e^{00} \\ 1.11022302e^{-16} \end{pmatrix}$$

Реализация метода Жордана для поиска обратной матрицы:

```
Реализация
# метода Жордана
def Jordan(matrix):
    n, _ = matrix.shape
    Расширяем# матрицу единичной справа
    ext = np.hstack((matrix, np.identity(n, float)))

    for k in range(0, n):
        Выбираем# номер строки для перестановки
        p = np.abs(ext[k:, k]).argmax() + k
        if p != k:
            ext[k], ext[p] = ext[p].copy(), ext[k].copy()

        tmp = ext[k, k]
        ext[k, k + 1:] /= tmp

        Дополнительно# зануляем елементы выше диагонали
        for i in range(0, n):
            if i != k:
                tmp = ext[i, k]
                ext[i, k + 1:] -= ext[k, k + 1:] * tmp
    #ext = (A | B), Возвращаем только B
    return ext[:, n:]
    ...
    a_ = Jordan(a.copy())
```

$$A^{-1} = \begin{pmatrix} 0.1284203 & -0.02682544 & 0.06899911 \\ -0.02682544 & 0.14398335 & -0.03091022 \\ 0.06899911 & -0.03091022 & 0.21147187 \end{pmatrix}$$

Проверим, действительно ли это обратная матрица:

```
e = a@a_
```

$$e = \begin{pmatrix} 1.00000000e^{00} & 3.11452972e^{-17} & 5.54026190e^{-17} \\ -1.10232041e^{-17} & 1.00000000e^{+00} & -5.52709686e^{-18} \\ -4.33198229e^{-17} & 1.08347606e^{-17} & 1.00000000e^{00} \end{pmatrix}$$

Реализация LU-разложения:

```
Реализация
# LUразложения-
def LU_decomposition(matrix):
    n, _ = matrix.shape
    L = np.zeros_like(matrix)
    U = np.zeros_like(matrix)

    for i in range(0, n):
        for j in range(i, n):
            L[j, i] = matrix[j, i] - np.sum(L[j, :i] * U[:i, i])
            U[i, j] = (matrix[i, j] - np.sum(L[i, :i] * U[:i, j])) / L[i,
    i]

    return L, U
```

Посчитаем определитель матрицы:

```
Считаем
# определитель матрицы с помощью Luразложения-
def det_LU( L, U):
    n, _ = L.shape
    res = 1
    for i in range(0, n):
        res *= L[i, i]
    return res
    ..
    L, U = LU_decomposition(a.copy())
    det_LU(L, U)
```

Убедимся, что L и U найдены верно:

$$L = \begin{pmatrix} 9.62483 & 0 & 0 \\ 1.15527 & 7.17024274 & 0 \\ -2.97153 & 1.04805326 & 4.72876132 \end{pmatrix} U = \begin{pmatrix} 1 & 0.12003017 & -0.30873584 \\ 0 & 1 & 0.14616705 \\ 0 & 0 & 1 \end{pmatrix}$$

Как и ожидалось, L — нижнетреугольная матрица, а U — верхнетреугольная с единичной главной диагональю

$$det(A) = det(L) = 326.3430141905548$$

2 Задание 2

2.1 Постановка задачи

Необходимо решить СЛАУ $Ax = b$, реализовав итерационные методы.
Точное решение x^* найти методом Гаусса. Исходную систему преобразовать в систему вида

$$x = H_D x + g_D$$

где

- $H_D = E - D^{-1}A$
- $g_D = D^{-1}b$
- D - диагональная матрица, у которой на диагонали находятся диагональные элементы матрицы А

Вычислить $\|H_D\|_\infty$.
Вычислить $x^{(7)}$ методом простой итерации.
Вычислить апостериорную оценку погрешности $\|x^* - x^{(7)}\|_\infty$. Сравнить апостериорную оценку с фактической погрешностью.
Вычислить приближение $x^{(7)}$ к решению системы $x = H_D x + g_D$ методом Зейделя. Вывести его фактическую погрешность. Сравнить с решением, полученным методом простой итерации.
Определить спектральный радиус матрицы перехода. Уточнить последнее приближение по Люстернику. Вывести его фактическую погрешность. Вычислить $x^{(7)}$ методом верхней релаксации.
Сравнить фактические погрешности$x^{(7)}$, полученного различными методами.

2.2 Вариант задания

Использованы следующие матрицы из методички (Вариант 8):

$$\left(\begin{array}{ccc|c} 9.62483 & 1.15527 & -2.97153 & 8.71670 \\ 1.15527 & 7.30891 & 0.69138 & 5.15541 \\ -2.97153 & 0.69138 & 5.79937 & 0.27384 \end{array}\right)$$

2.3 Решение

Воспользуемся реализацией метода Гаусса для решения СЛАУ из Задачи 1.

$$x^* = \begin{pmatrix} -0.50036016 \\ -0.00807482 \end{pmatrix}$$

Преобразуем систему:

$$H_D = E - D^{-1}A$$

$$g_D = D^{-1}b$$

```
Преобразование
# системы к необходимому виду
def transform_equation(matrix, vector):
    Определим# матрицу D
    D = np.identity(matrix.shape[0], float)
    for i in range(0, matrix.shape[0]):
        D[i][i] = matrix[i][i];
    Определим# матрицу H_{D}
    H_D = np.identity(matrix.shape[0], float) - np.linalg.inv(D).dot(
    matrix)
```

```
    Определим# вектор g_{D}
    g_D = np.linalg.inv(D).dot(vector)

    return np.zeros(matrix.shape[0]), H_D, g_D
```

Функция возвращает кортеж, состоящий из нулевого приблежения $x^0 = 0$, матрицы H_D, вектора g_D.

$$H_D = \begin{pmatrix} 0 & -0.12003017 & 0.30873584 \\ -0.15806324 & 0. & -0.09459413 \\ 0.51238841 & -0.1192164 & 0 \end{pmatrix} g_D = \begin{pmatrix} 0.90564716 \\ 0.70535962 \\ 0.04721892 \end{pmatrix}$$

Вычислим $\|H_D\|_\infty$:

```
norm_H_D = np.linalg.norm(H_D, ord=np.inf)
```

$$\|H_D\|_\infty = 0.6316048122468475$$

Предъявим реализацию метода простой итерации:
Его расчетная формула на шаге:

$$x^{(k+1)} = H_D x^{(k)} + g_D$$

```
Метод
# простой итерации
def simple_iteration(H_D, g_D, x0, count):
    Список# приближений x^(k)
    iters = [x0]

    for k in range(0, count):
        temp = H_D.dot(iters[k]) + g_D
        iters += temp
    return iters
    ...
    iters = simple_iteration(H_D, g_D, x0, 7)
```

$$x^{(7)} = \begin{pmatrix} 0.99822136 \\ 0.50144807 \\ 0.49638199 \end{pmatrix}$$

Вычислим апостеорную оценку погрешности метода простой итерации:

$$\|x^* - x^{(k)}\| \leqslant \frac{\|H_D\|}{1 - \|H_D\|} \|x^{(k)} - x^{(k-1)}\|$$

```
def aprior_estimate(H_D, iters):
    return np.linalg.norm(H_D, np.inf) * np.linalg.norm(iters[-1] - iters
   [-2], np.inf) / (1 - np.linalg.norm(H_D, np.inf))
```

$$\|x^* - x^{(7)}\|_\infty = 0.007903752117469586$$

Сравним апостеорную оценку погрешности с фактической погрешностью:

```
Фактическая
# погрешность
np.linalg.norm(acc_sol - iters[-1], np.inf) / np.linalg.norm(acc_sol, np.
    inf)
```

$$\Delta x = 0.0036171105203740696$$

Видно, что фактическая погрешноть метода простой итерации меньше апостеорной оценки.

Реализуем метод Зейделя:

Расчетная формула на шаге:

$$x^{(k+1)} = (E - H_L)^{-1} H_R x^{(k)} + (E - H_L)^{-1} g_D$$

```
Представляем
# матрицу H_D в виде суммы
def decompose_H_D(H_D):
    H_L = np.zeros(H_D.shape, float)
    H_R = np.zeros(H_D.shape, float)

    for i in range(0, H_L.shape[0]):
        H_L[i][:i] = H_D[i][:i].copy();
        H_R[i][i:] = H_D[i][i:].copy()
    return H_L, H_Rметод
# Зейделя
def Zeidel(H_D, g_D, x0, count):
    Список# приближений решения, начинается с x0
    iters = [x0]
    H_L, H_R = decompose_H_D(H_D)
    #EH_L = (E - H_L)^-1
    EH_L = np.linalg.inv(np.identity(H_L.shape[0], float) - H_L)
    for k in range(0, count):
        temp = EH_L@H_R.dot(iters[k]) + EH_L.dot(g_D)
        iters.append(temp)

    return iters
    ...
    iters = Zeidel(H_D, g_D, x0, 7)
```

$$x^{(7)} = \begin{pmatrix} 0.9999871 \\ 0.50000449 \\ 0.49999199 \end{pmatrix}$$

Вычислим фактическую погрешность метода Зейделя:

```
Фактическая
# погрешность
np.linalg.norm(acc_sol - iters[-1], np.inf) / np.linalg.norm(acc_sol, np.
    inf)
```

$$\Delta x = 1.2687684652139433 e^{-05}$$

Можно увидеть, что метод Зейделя дает ощутимо меньшую фактическую погрешность в сравнении с методом простой итерации.

Матрица перехода метода Зейделя выглядит следубющим образом:

$$H_{seid} = (E - H_L)^{-1} H_R$$

Вычислим ее спектральный радиус с помощью встроенной функции в модуль numpy:

```
Матрица
# перехода метода Зейделя
H_seid = np.linalg.inv(np.identity(H_L.shape[0], float) - H_L)@H_R
np.abs(np.linalg.eigvals(zeidel_H)).max()
```

$$\rho(H_seid) = 0.22062881428014927$$

Теперь, зная спектральнйы радиус матрицы перехода метода Зейделя, можно применить метод ускорения сходимости Люстерника.
Условие для применения этого метода:

$$\rho(H_seid) < 1$$

Расчетная формула:

$$\bar{x} = x^{(k)} + \frac{1}{1 - \rho(H_seid)}(x^{(k)} - x^{(k-1)})$$

Реализация:

```
def Lusternik_speedup(H, iters):
    H_radius = np.abs(np.linalg.eigvals(H)).max()

    if(H_radius < 1):
        temp = iters[-2] + 1.0/(1 - H_radius)*(iters[-1] - iters[-2])
        iters.append(temp)
    return iters
    ...
    iters = Lusternik_speedup(zeidel_H, iters)
```

$$\bar{x} = \begin{pmatrix} 0.99999979 \\ 0.49999943 \\ 0.4999991 \end{pmatrix}$$

Фактическая погрешность уточнененного последнего приблежения:

$$\Delta x = 7.441460129632571e^{-11}$$

Точность решения действительно повысилась.
Реализуем метод верхней релаксации:

```
Метод
# верхней релаксации
def upper_relax(H_D, g_D, x0, count):
    iters = [x0]
    H_radius = np.abs(np.linalg.eigvals(H_D)).max()
    qopt = 2.0/(1 + sqrt(1 - H_radius**2))

    for k in range(0, count):
        temp = np.zeros(x0.shape, float)
```

```
        for i in range(0, x0.shape[0]):
            s = 0
            for j in range(0, i):
                s += H_D[i][j]*temp[j]
            for j in range(i+1, x0.shape[0]):
                s += H_D[i][j]*iters[k][j]
            s += g_D[i] - iters[k][i]
            temp[i] = iters[k][i] + qopt*s
        iters.append(temp)
    return iters
    ...
    iters = upper_relax(H_D, g_D, x0,  7)
```

$$x^{(7)} = \begin{pmatrix} 0.99999972 \\ 0.49999949 \\ 0.49999908 \end{pmatrix}$$

Фактическая погрешность метода верхней релаксации:

$$\Delta x = 7.049982307967775e^{-08}$$

2.4 Сравнение результатов

Ниже приведена таблица с фактическими погрешностями итерационных методов:

Название метода	Фактическая погрешность
Метод простой итерации	0.0036171105203740696
Метод Зейделя	$1.2687684652139433e^{-05}$
Метод Люстерника	$7.441460129632571e^{-11}$
Метод верхней релаксации	$7.049982307967775e^{-08}$

3 Задание 3

3.1 Постановка задачи

Необходимо решить частичную проблему собственных значений. Реализовать два метода:

- Степенной метод нахождения максимального по модулю собственного числа матрицы
- Метод скалярных произведений нахождения максимального по модулю собственного числа матрицы

Найти максимальное по модулю собственное число λ и соответсвующий ему собственный вектор x.

3.2 Вариант задания

Вариант 8 задания 13.8 из методички.
В качестве матрицы взят вариант 8(13.9).

$$A = \begin{pmatrix} -1.47887 & -0.09357 & 0.91259 \\ -0.09357 & 1.10664 & 0.03298 \\ 0.91259 & 0.03298 & -1.48225 \end{pmatrix}$$

3.3 Решение

Выберем начальный вектор Y_0. В коде программы:

```
Y0 = np.ones((A.shape[0], 1), float)
```

$$Y_0 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

По формуле $Y^{(k+1)} = AY^{(k)}$ строим последовательность векторов

$$Y^{(0)}, Y^{(1)}, Y^{(2)}...$$

Для обеспечения заданной наперед точности $\varepsilon = 0.001$ будем использовать апостериорную оценку погрешности

$$|\lambda - \bar{\lambda}| \leqslant \frac{\|A\bar{Y} - \bar{\lambda}\bar{Y}\|}{\|\bar{Y}\|_2}$$

```
Апостериорная
# оценка погрешности
def aposterior_err(A, eig_val, eig_vec):
    temp = np.linalg.norm(A.dot(eig_vec) - eig_val*eig_vec, 2) / np.
   linalg.norm(eig_vec, 2)
    return temp
```

Компоненты вектора $Y^{(k)}$ могут быстро расти или быстро убывать при $|\lambda| > 1$ или $|\lambda| < 1$, поэтому для избавления от этого нежелательного эффекта будем нормировать вектор $Y^{(k)}$:
Пусть $|y_p^{(0)}| = max|y_i^{(0)}|, 1 \leqslant i \leqslant n$. Тогда $Y_{norm}^{(k)} = \frac{Y^{(k)}}{y_p^{(k)}}$
Расчетная формула в степенном методе примет вид:

$$\lambda_{pow}^{(k)} = y_p^{(k+1)}$$

Реализация степенного метода:

```
Степенной
# метод
def power(A, Y0, eps):
    iters = 0
    temp = np.array([abs(i) for i in Y0], float).argmax()
    eig_vecs = [Y0.copy()/Y0[temp]]
    eig_vals = [ A.dot(eig_vecs[iters])[temp][0]]
```

```

    while aposterior_err(A, eig_vals[iters], eig_vecs[iters]) > eps:

        eig_vecs.append(A.dot(eig_vecs[iters]))
        ind = np.array([abs(i) for i in eig_vecs[-1]], float ).argmax()
        eig_vecs[-1] /= eig_vecs[-1][ind]
        eig_vals.append(A.dot(eig_vecs[-1])[ind][0])

        iters += 1
    return eig_vals, eig_vecs
    ...
    eig_vals, eig_vecs = power(A, Y0, eps)
```

Собственное число и собственный вектор, найденные с помощью степенного метода:

$$\lambda^{(k)} - 2.395436621163447$$

$$x^{(k)} = \begin{pmatrix} -0.9993384997374231 \\ -0.03639477525990835 \\ 1. \end{pmatrix}$$

Аналогично в методе скалярных произведений расчетная формула примет вид:

$$\lambda_{scal}^{(k)} = \frac{(AY_{norm}^{(k)}, Y_{norm}^{(k)})}{(Y_{norm}^{(k)}, Y_{norm}^{(k)})}$$

Его реализация:

```
Метод
# скалярных произведений
def scalar(A, Y0, eps):
    iters = 0;
    temp = np.array([abs(i) for i in Y0], float).argmax()
    eig_vecs = [Y0.copy()/Y0[temp]]
    eig_vals = [A.dot(eig_vecs[-1]).dot(eig_vecs[-1].transpose())[0][0]/
   eig_vecs[-1].dot(eig_vecs[-1].transpose())[0][0]]

    while aposterior_err(A, eig_vals[-1], eig_vecs[-1]) > eps:
        eig_vecs.append(A.dot(eig_vecs[-1]))
        ind = np.array([abs(i) for i in eig_vecs[-1]], float ).argmax()
        eig_vecs[-1] /= eig_vecs[-1][ind]
        eig_vals.append(A.dot(eig_vecs[-1]).dot(eig_vecs[-1].transpose())
   [0][0]/eig_vecs[-1].dot(eig_vecs[-1].transpose())[0][0])

    return eig_vals, eig_vecs
    ...
    eig_vals, eig_vecs  = scalar(A, Y0, eps)
```

Собственное число и собственный вектор, найденные с помощью метода скалярных произведений:

$$\lambda^{(k)} - 2.3954717914467905$$

$$x^{(k)} = \begin{pmatrix} -0.9993384997374231 \\ -0.03639477525990835 \\ 1. \end{pmatrix}$$

Собственное число, найденное библиотечным методом:

$$\lambda^{(k)} = -2.395436680511785$$

Ниже представлены две таблицы с протоколом работы методов

Таблица степенного метода:

	k	eig_val[k]	eig_val[k] - eig_val[k-1]	eig_val[k] - acc_eig_val	||vA*eig_vec[k] - eig_val[k]*eig_vec[k]||	Error_estimate
0	0	1.197965	0.000000e+00	3.593402e+00	3.999575	2.618213
1	1	-2.299761	-3.497727e+00	9.567534e-02	1.966496	1.347035
2	2	-2.371765	-7.200327e-02	2.367207e-02	0.966211	0.678874
3	3	-2.389756	-1.799125e-02	5.680815e-03	0.454147	0.320776
4	4	-2.389159	5.965959e-04	6.277411e-03	0.211032	0.149204
5	5	-2.395115	-5.955751e-03	3.216602e-04	0.097928	0.069244
6	6	-2.395360	-2.453453e-04	7.631495e-05	0.045387	0.032094
7	7	-2.395418	-5.813403e-05	1.818093e-05	0.021033	0.014873
8	8	-2.395432	-1.388533e-05	4.295593e-06	0.009747	0.006892
9	9	-2.395436	-3.264213e-06	1.031381e-06	0.004516	0.003194
10	10	-2.395436	-7.914183e-07	2.399624e-07	0.002093	0.001480
11	11	-2.395437	-1.806141e-07	5.934834e-08	0.000970	0.000686

Таблица метода скалярных произведений:

	k	eig_val[k]	eig_val[k] - eig_val[k-1]	eig_val[k] - acc_eig_val	||vA*eig_vec[k] - eig_val[k]*eig_vec[k]||	Error_estimate
0	0	-2.713753	0.000000	-0.318317	3.963561	2.594637
1	1	-2.588044	0.125709	-0.192608	2.144521	1.468981
2	2	-2.383807	0.204238	0.011630	0.969125	0.680921
3	3	-2.417797	-0.033990	-0.022360	0.458800	0.324062
4	4	-2.389159	0.028637	0.006277	0.211032	0.149204
5	5	-2.399303	-0.010144	-0.003866	0.098196	0.069434
6	6	-2.393873	0.005430	0.001563	0.045422	0.032118
7	7	-2.396215	-0.002342	-0.000779	0.021067	0.014896
8	8	-2.395089	0.001126	0.000348	0.009758	0.006900
9	9	-2.395601	-0.000512	-0.000164	0.004523	0.003198
10	10	-2.395361	0.000240	0.000075	0.002096	0.001482
11	11	-2.395472	-0.000111	-0.000035	0.000971	0.000687

Оба метода работают примерно с одинаковой скоростью, однако наблюдается несколько большая точность работы степенного метода.

4 Задание 4

4.1 Постановка задачи

Необходимо найти приближенное решение интегрального уравнения

$$u(x) - \int\limits_a^b H(x, y)u(y)dy,$$

используя одну из квадратурных формул:

- Составная формула трапеций
- Составная формула средних прямоугольников
- Составная формула Симпсона
- Формулы Гаусса с n = 2, 3, 4 ... узлами
- Составная формула Гаусса с двумя узлами
- Составная формула Гаусса с тремя узлами

и метод механических квадратур.
КФ и уравнение выбрать согласно варианту задания 4.3 из методички.

4.2 Вариант задания

Вариант 8:

$$u(x) - \int_0^1 e^{(x-0.5)y^2} u(y)dy = x + 0.5$$

Использовать квадратурную формулу Гаусса с n узлами.

4.3 Решение

Постараемся приближенно вычислить интеграл $\int_0^1 e^{(x-0.5)y^2} u(y)dy$, заменив его на квадратурную сумму вида:

$$\sum_{k=1}^{n} A_k H(x, x_k) u(x_k), \text{ где}$$

- $x_{1..n} \in [a, b]$ – попарно различные n узлов
- A_k - коэффициенты КФ
- $u(x_k)$ — значения искомой функции в узлах

Получим уравнение относительно "новой"неизвестной функции $u^n(x)$:

$$u^n(x) - \sum_{k=1} nA_k H(x, x_k) u^n(x_k) = f(x), x_k \in [a, b] = [0, 1]$$

Нам не известны искомой функции в узлах.
Положим x поочередно равным x_i, подставим его в уравнение и получим:

$$u^n(x_j) - \sum_{k=1} nA_k H(x_j, x_k) u^n(x_k) = f(x_j)$$

n неизвестных значений u^n и ровно n подстановок x_k вместо x образуют систему уравнений:

$$D = d_{jk} = \delta_{jk} - A_k H(x_j, x_k) u(x_k), j,k \in 1..n$$

$$g = f(x_j), z = u^n(x_k)$$

$$Dz = g$$

Для решения системы необходимо посчитать $H(x_j, x_k)$ и $f(x_j)$, поскольку они представлены в явном виде.
Приступим к вычислению коэффициентов A_k.
КФ Гаусса - это частный случай КФ типа Гаусса с весом $\rho(x) \equiv 1$. Узлами КФ являются корни ортогонального многочлена Лежандра, определенного на $[-1, 1]$.

$$P_n(x) = \frac{1}{2^n n!} \frac{d^n}{dx^n} (x^2 - 1)^n, x \in [-1, 1]$$

Вычисление значения многочлена, представленного в такой форме, не кажется удобным, поэтому воспользуемся следующим реккурентным соотношением:

- $P_0(x) = 1$
- $P_1(x) = x$
- $P_{n+1}(x) = \frac{2n+1}{n+1} x P_n(x) - \frac{n}{n+1} P_{n-1}(x)$

Реализуем соответствующий функционал в коде программы:

```
Вычисление
# значения многочлена Лежандра
def legp_eval(arg, n):
    lst = [1.0, float(arg)]
    for i in range(1, n+1):
        t = lst[i]*(2.0*float(i) + 1.0)/(float(i)+1.0)*arg - lst[i-1]*
   float(i)/float(i+1)
        lst.append(t)
    return lst
```

Возвращается список значений $[P_0(x), ..., P_n(x)]$
Предоставим функцию вычисления производной многочлена Лежандра в т. x:

$$P_n'(x) = \frac{n}{1-n^2}[P_{n-1}(x) - xP_n(x)]$$

```
Производная
# многочлена Лежандра
def legp_der(arg, n):
    lst = legp_eval(arg, n)
    return n/(1.0-arg**2)*(lst[n-1] - arg*lst[n])
```

Найдем корни $P_n(x)$, их ровно n штук и x^i — корень, значит $x^i \in [-1, 1]$
Вычислим корни $P_n(x)$ итеративно с помощью метода Ньютона:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(xn)}$$

или в наших обозначениях

$$x_i^{(k+1)} = x_i^k - \frac{P_n(x_i^k)}{P_n'(x_i^{(k)})}, \text{ i = 1,.., n}$$

В качестве начального приближения x_i^0 выберем

$$cos(\pi \frac{(4i-1)}{(4n+2)})$$

Итерационный процесс преращается при достижении

$$|x_i^{(k+1)} - x_i^{(k)}| < \varepsilon_{root}, \varepsilon_{root} \text{ задан наперед}$$

Программная реализация:

```
Нахождение
# корней многочлена Лежандра
def legp_roots(n, eps):
    roots = [cos(pi*(4*i -1.0)/(4*n+2)) for i in range(1, n+1)]
    for i in range(0, n):
        temp = roots[i] - legp_eval(roots[i], n)[n]/legp_der(roots[i], n)
        while abs(temp - roots[i]) > eps:
            roots[i] = temp
            temp = roots[i] - legp_eval(roots[i], n)[n]/legp_der(roots[i
], n)
        roots[i] = temp
    return roots
```

Теперь имеется все необходимое для подсчета $A_k = \frac{2}{(1-x_k^2)[P_n'(x_k)]^2}$:

```
Функция
# подготавливающая узлы и коэффициенты КФ Гаусса
|def Gauss_prepare(n,  sect, eps):
|    knots = legp_roots(n, eps)
--> coefs =  [2/(1-k**2)/(legp_der(k, n)**2) for k in knots]
    knots = list(map(lambda k: (sect[1] - sect[0])/2*k + (sect[1] + sect
[0])/2, knots))
    return coefs, knots
```

Заметим, что $[a, b] \neq [-1, -1]$. Поэтому выполним замену переменной:

$$x = \frac{(b-a)}{2}t + \frac{(b+a)}{2}, dx = \frac{(b-a)}{2}dt$$

Корни $P_n(x)$ после замены переменной являются узлами КФ Гаусса. Замена переменной выполняется тут:

```
#sect - это кортеж с информацией о промежутке интегрирования: sect = (a,
b)
def Gauss_prepare(n,  sect, eps):
    knots = legp_roots(n, eps)
    ...
--> knots = list(map(lambda k: (sect[1] - sect[0])/2*k + (sect[1] + sect
[0])/2, knots))
    ...
```

A_k не требуют преобразований.
Теперь можем решить систему уравнений относительно z:

$$Dz = g$$

```
def mech_quadrature(f, H, n, sect, eps, root_eps):
    ...
    coefs, knots = Gauss_prepare(n,sect, root_eps)
    D = np.array([[ kron(k, j) - coefs[k]*H(knots[j], knots[k]) for k in
range(0, n)] for j in range(0, n)])
    g =np.array([f(k) for k in knots])
    z = np.linalg.solve(D, g)
    ...
```

Воспользовавшись библиотечным методом, получили решение системы:

$$z = \begin{pmatrix} z_1 \\ . \\ . \\ . \\ z_n \end{pmatrix}$$

В методе mech_quadrature далее "собирается"новая функция, которая является приближенным решением интегрального уравнения:

```
def mech_quadrature(f, H, n, sect, eps, root_eps):
    ...
    def sol(arg):
        return np.sum([coefs[i]*H(arg, knots[i])*z[i] for i in range(0, n)]) + f(arg)
    return sol
```

После в методе find_eps_near_sol оцениваются пары решений:

$$(1)\ \max_{i=1,2,3} |u(x_i)^{n+m+1} - u(x_i)^{n+m+1}| < \varepsilon$$

При невыполнении оценки генерируется еще одно решение с на 1 большим количеством узлов КФ, т.е. $m = m + 1$

```
def find_eps_near_sol(n, sect, eps, root_eps):
(1)
--> def sols_diff(sols, sect):
        args = [sect[0], (sect[1]+sect[0])/2, sect[1]]
        return  np.array([abs(sols[-1](i) - sols[-2](i)) for i in args], float).max()


    sols = [mech_quadrature(f, H, n, sect, eps, root_eps), mech_quadrature(f, H, n+1, sect, eps, root_eps)]
    iters = 2
    while sols_diff(sols, sect)>eps:
        sols.append(mech_quadrature(f, H, n+iters,sect,  eps, root_eps))
        iters+=1

    return sols
```

Посмотрим на результаты, запустив программу $n = 2$, $\varepsilon = 10^{(-10)}$:

```
     x          a     (a+b)/2           b  max | u^{i}(xi) - u^{i-1}(xi)|
u^2+0 -93.663016 -110.737179 -133.843501                     0.000000e+00
u^2+1 -88.840225 -104.964072 -127.115532                     6.727969e+00
u^2+2 -88.793266 -104.909023 -127.053273                     6.225897e-02
u^2+3 -88.792759 -104.908419 -127.052592                     6.802552e-04
u^2+4 -88.792755 -104.908415 -127.052588                     4.640003e-06
u^2+5 -88.792755 -104.908415 -127.052588                     3.045940e-08
u^2+6 -88.792755 -104.908415 -127.052588                     1.599005e-10
u^2+7 -88.792755 -104.908415 -127.052588                     1.932676e-12
```

Посмотрим на промежуточные результаты($n = 2, 3, 7$):

```
Количество узлов КФ: 2

Узлы:
0.7886751345948129
0.21132486540518713

Коэффициенты КФ:
1.0000000000000002
1.0000000000000002

Матрица системы:
[[ 0.40165562 -0.50648758]
 [-0.41781958  0.50640448]]

Вектор правых частей:
[1.28867513 0.71132487]

Решение системы: [-123.21659425 -100.25776375]

Значения решения в a, (a + b)/2, b:
[-93.66301648877896, -110.73717900381395, -133.84350090077962]
```

```
Количество узлов КФ: 9

Узлы:
0.984080119753813
0.9180155536633179
0.8066857163502952
0.6621267117019045
0.5
0.33787328829809554
0.19331428364970482
0.08198444633668212
0.015919880246186957

Коэффициенты КФ:
0.08127438836157456
0.18064816069485734
0.2606106964029356
0.31234707704000286
0.3302393550012598
0.31234707704000286
0.2606106964029356
0.18064816069485734
0.08127438836157456

Решение системы: [-126.23031111 -122.90902222 -117.62850267 -111.31949217 -104.90841468
  -99.12448268  -94.4396776   -91.10640148  -89.2330506 ]

Значения решения в a, (a + b)/2, b:
[-88.79275489470186, -104.90841467748672, -127.05258761048131]
```

$$u^{n+7}(a) - u^{n+6}(a) = 1.4210854715202004e^{-12}$$

$$u^{n+7}((b+a)/2) - u^{n+6}((b+a)/2) = 1.6768808563938364e^{-12}$$

$$u^{n+7}(b) - u^{n+6}(b) = 1.9326762412674725e^{-12}$$

4.3.1

Проверим, насколько точно вычисляются значения $P_n(x)$.

```
def leg_acc(num , n = 6):
    res = []
    for i in range(0, num):
        arg = -1 + (1 -(-1))/num*i
        accs = [1, arg, (3*arg**2 - 1)/2,\
        (5*arg**3 - 3*arg)/2,\
        (35*arg**4 - 30*arg**2 +3)/8,\
        (63*arg**5 - 70*arg**3 +15*arg)/8,\
        (231*arg**6 - 315*arg**4 + 105*arg**2 -5)/16]
        appr = legp_eval(arg, n)
        res.append( list(map(lambda a, b: abs(a - b), accs, appr)))
    print(np.array([ np.array([k  for k in i], float).max() for i in res
   ], float).max())
...
leg_acc(100, n = 6)
```

В списке accs находятся значения $P_0(x), .., P_6(x)$, посчитанные по явным формулам многочленов соответсвующих степеней. Список appr хранит значения $P_0(x), ..., P_6(x)$, посчитанные функцией legp_eval (с применением реккурентного соотношения). Процедура выводит на экран $\Delta_i = max|acc_i[k] - appr_i[k]|$ при $k \in [0,.., 6]$, $i \in [0,..,num - 1]$

значения num	Δ_i
1	$1.1102230246251565e^{-16}$
10	$9.43689570931383e^{-16}$
100	$2.9976021664879227e^{-15}$
1000	$3.83026943495679e^{-15}$
10000	$4.3298697960381105e^{-15}$

Можно увидеть, что ошибки округления и нехватка точности, возникающие при счете, не оказывают большого влияния на результат.

Насколько хоршо находятся корни $P_n(x)$?

```
def leg_root_acc():
    reps  = 10**(-10)
    res = []
    for i in range(2, 7):
        temp = []
        for j in legp_roots(i, reps):
            arg = j
            accs = [1, arg, (3*arg**2 - 1)/2,...
            temp.append(abs(0 - accs[i]))
        res.append(temp)
    res = [np.array([i for i in k], float).max() for k in res]
    print(res)
```

У многочленов $P_2,..,6(x)$ находим корни, оцениваем $|P_i^{accurate}(x_ij) - 0|$:

Степень i	$max\lvert P_i^{accurate}(x_{ij}) - 0\rvert, j \in [1, ..., i]$
2	0.0
3	0.0
4	$2.4.440892098500626e^{-16}$
5	$2.4.440892098500626e^{-16}$
6	$8.881784197001252e^{-16}$

Видно, что корни многочлена Лежандра так же вычисляются с точностью, сопоставимой с машинным нулем.

5 Задача 5

5.1 Постановка задачи

Необходимо найти решение задачи

$$\frac{\delta u}{\delta x} = Lu + f(x,t), 0 < x < 1, 0 < t \leqslant 0.1$$

$$u(x, 0) = \varphi(x), 0 \leqslant x \leqslant 1$$

$$\left(\alpha_1(t)u - \alpha_2(t)\frac{\delta u}{\delta x}\right)\Bigg|_{x=0} = \alpha(t), 0 \leqslant t \leqslant 0.1$$

$$\left(\beta_1(t)u - \beta_2(t)\frac{\delta u}{\delta x}\right)\Bigg|_{x=1} = \beta(t), 0 \leqslant t \leqslant 0.1$$

методом сеток используя различные разностные схемы:

- Явную схему порядка $O(h^2 + \tau)$ с аппроксимацией производных в граничных условиях с порядком $O(h^2)$
- схему с весами при $\sigma = 0$ $\sigma = 0.5$ $\sigma = 1$ с аппроксимацией производных в гранич- ных условиях с порядком $O(h)$

5.2 Вариант задания

Тестировование будет проходить с использованием решения $u(x,t) = x^3 + t^3$

$$\frac{\delta u}{\delta x} = \frac{\delta}{\delta x}\left((x+1)\frac{\delta u}{\delta x}\right) + f(x,t)$$

$$u(x, 0) = \varphi(x), 0 \leqslant x \leqslant 1$$

$$\frac{\delta u}{\delta x}\Bigg|_{x=0} = \alpha(t), u(1,t) = \beta(t), 0 \leqslant t \leqslant 0.1$$

5.3 Решение

В условиях варианта задания оператор Lu выглядит так:

$$Lu = \frac{\delta u}{\delta x}\left(p(x)\frac{\delta u}{\delta x}\right) + b(x,t)\frac{\delta u}{\delta x} + c(x,t)u$$

Разобьем отрезок $[0,1]$ на N равных частей. $h = 1/N$, $x_i = hi$ $i = 0,..,N$.
Разобьем отрезок $[0,0.1]$ на M равных частей. $\tau = 1/M$, $t_k = \tau k$ $k = 0,..,M$.
Построим сетку узлов $\omega_{h\tau}$
Приближенное решение будем искать в виде таблицы значений искомой функции в узлах сетки $\omega_{h\tau}$.
$u_i^k = u(x_i, t_k)$
Заменим оператор Lu разностным оператором $L_h u_i^k$:

$$L_h u_i^k = p_{i+0.5}\frac{u_{i+1}^k - u_i^k}{h^2} - p_{i-0.5}\frac{u_i^k - u_{i-1}^k}{h^2} + b(x_i,t_k)\frac{u_{i+1}^k - u_{i-1}^k}{2h} + c(x_i, t_i)u_i^k$$

Здесь $k = 1,..,M$ $i = 1,..,N-1$ В коде программы:

```
def Lu(h, i, k, u_vals, x, t):
    return p((i+0.5)*h)*(u_vals[k][i+1] - u_vals[k][i])/(h**2) - p((i-0.5)*h)*(u_vals[k][i] - u_vals[k][i-1])/(h**2) + b(x(i), t(k))*(u_vals[k][i+1] - u_vals[k][i-1])/(2*h) + c(x(i), t(k))*u_vals[k][i];
```

Рассмотрим явную разностную схему:
Аппроксимируем уравнение в точке (x_i, t_{k-1})

$$\frac{u_i^k - u_i^{k-1}}{\tau} = L_h u_i^{k-1} + f(x_i,t_{k-1})$$

Из начального условия $u(x,0) = \varphi(x)$, $0 \leqslant x \leqslant 1$ получим

$$u_i^0 = \varphi(x_i),\, i = 0,..,N$$

Вычислим u_i^0 при $i = 0,..,N$

```
def find_u0i():
            return [phi(x(i)) for i in range(0, N+1)]
```

Теперь можем вычислить u_i^k при при $i = 1,..,N-1$

```
def find_uki(k):
            return [u_vals[k-1][i] + tau*(Lu(h, i, k-1, u_vals, x, t) + f(x(i), t(k-1))) for i in range(1,N)]
```

Аппроксимируя граничные условия порядком $O(h^2)$

$$\left(\alpha_1(t)u - \alpha_2(t)\frac{\delta u}{\delta x}\right)\bigg|_{x=0} = \alpha(t),\, 0 \leqslant t \leqslant 0.1$$

$$\left(\beta_1(t)u - \beta_2(t)\frac{\delta u}{\delta x}\right)\bigg|_{x=1} = \beta(t),\, 0 \leqslant t \leqslant 0.1$$

получим

$$\alpha_1(t_k)u_0^k - \alpha_2(t)\frac{-3u_0^k + 4u_1^k - u_2^k}{2h} = \alpha(t_k)$$

$$\beta_1(t_k)u_N^k - \beta_2(t)\frac{-3u_N^k + 4u_{N-1}^k - u_{N-2}^k}{2h} = \beta(t_k)$$

Вычислим:

```
def find_uk0(k):
            return (alph(t(k)) + alph2(t(k))*(4*u_vals[k][1] - u_vals[k][2])/(2*h))/(alph1(t(k)) + (3*alph2(t(k))/(2*h)))
        def find_ukN(k):
            return (betta(t(k)) - betta2(t(k))*(-4*u_vals[k][N-1] + u_vals[k][N-2])/(2*h))/(betta1(t(k)) + betta2(t(k))*3/(2*h))
```

Реализация явного разностного метода:

```
def expl_sub(N, M, h, tau, D):
        x = lambda i: D[0][0] + h*i
        t = lambda i: D[1][0] + tau*i

        u_vals = []
        def find_u0i():
            return [phi(x(i)) for i in range(0, N+1)]
        # i from 1 to N-1
        def find_uki(k):
            return [u_vals[k-1][i] + tau*(Lu(h, i, k-1, u_vals, x, t) + f(x(i), t(k-1))) for i in range(1, N)]
        def find_uk0(k):
            return (alph(t(k)) + alph2(t(k))*(4*u_vals[k][1] - u_vals[k][2])/(2*h))/(alph1(t(k)) + (3*alph2(t(k))/(2*h)))
        def find_ukN(k):
            return (betta(t(k)) - betta2(t(k))*(-4*u_vals[k][N-1] + u_vals[k][N-2])/(2*h))/(betta1(t(k)) + betta2(t(k))*3/(2*h))

        u_vals.append(find_u0i())
        вычисляем# значения на каждом слое
        for k in range(1, M+1):
            u_vals.append([0] + find_uki(k) + [0])
            u_vals[k][0] = find_uk0(k)
            u_vals[k][N] = find_ukN(k)

        return u_vals
```

Теперь рассмотрим схему с весами (неявный метод)

Рассмотрим семейство разностных систем с вещественным параметром σ

$$\frac{u_i^k - u_i^{k-1}}{\tau} = L_h(\sigma u_i^k + (1-\sigma)u_i^{k-1}) + f(x_i,\bar{t}_k)$$

Граничные условия аппроксимируем уже с первым порядком:

$$\alpha_1(t_k)u_0^k - \alpha_2(t)\frac{u_1^k - u_0^k}{h} = \alpha(t_k)$$

$$\beta_1(t_k)u_N^k - \beta_2(t)\frac{u_N^k - u_{N-1}^k}{h} = \beta(t_k)$$

Имея в виду, что на $k+1$ слое известны значения функции в узлах слоя k, можем переписать условия:

$$\sigma L_h u_i^k - \frac{1}{\tau}u_i^k = G_i^k$$

причем

$$G_i^k = -\frac{1}{\tau}u_i^{k-1} - (1-\sigma)L_h u_i^{k-1} - f(x_i, \bar{t}_k)$$

Граничные же условия представим в таком виде:

$$-B_0^k u_0^k + C_0^k u_1^k = G_0^k$$

$$A_N^k u_{N-1}^k - B_N^k u_N^k = G_N^k$$

Таким образом получим систему с трехдиагональной матрицей

$$\begin{array}{rrrcll} & -B_0 u_0^k & +C_0 u_1^k & = & G_0^k, & \\ A_i u_{i-1}^k & -B_i u_i^k & +C_i u_{i+1}^k & = & G_i^k, & i = \overline{1, N-1}, \\ A_N u_{N-1}^k & -B_N u_N^k & & = & G_N^k, & \end{array}$$

$k = \overline{1, M}$.

Эту систему будем решать методом прогонки.
Решение будем искать в виде $u_i^k = s_i^k u_{i+1}^k + t_i^k$
Из нулевого уравнения:

$$s_0^k = \frac{C_0^k}{B_0^k}$$

$$t_0^k = \frac{-G_0^k}{B_0^k}$$

Подставим u_{i-1}^k в i уравнение системы:

$$s_i^k = \frac{C_i^k}{B_i^k - A_i^k s_{i-1}^k}$$

$$t_i^k = \frac{A_i^k t_{i-1}^k}{B_i^k - A_i^k s_{i-1}^k}$$

Заметим, что $C_n^k = 0$, поэтому $u_n^k = t_n^k$. Последовательно вычисляем все остальные значения решения на текущем слое:

```
    ...Для
# вычисления значений решения на каждом слое используем метод прогонки
    def find_uki(k):Определили
# все коэффициенты в системе
        Ak = [0] + [sgm*(p(x(i-0.5))/(h**2) - b(x(i),t(k))/(2*h)) for i
    in range(1, N)] + [-betta2(t(k))/h ]
        Bk = [-alph1(t(k)) - alph2(t(k))/h] + [sgm*(p(x(i+0.5))/(h**2) +
    p(x(i-0.5))/(h**2) - c(x(i), t(k))) + 1/tau for i in range(1, N)] + [
    -betta1(t(k)) - betta2(t(k))/h]
        Ck = [-alph2(t(k))/h] + [sgm*(p(x(i+0.5))/(h**2) + b(x(i), t(k))
    /(2*h)) for i in range(1, N)] + [0]
        Gk = [alph(t(k))] +[-1/tau*u_vals[k-1][i] - (1-sgm)*Lu(h,i,k-1,
    u_vals, x, t) - f(x(i), t_(k)) for i in range(1, N)]+ [betta(t(k))]
        #ui = si*ui+1 +ti
        def sk_tk():
            sk = [Ck[0]/Bk[0]]
            tk = [-Gk[0]/Bk[0]]
```

```
            for i in range(1, N+1):
                sk.append(Ck[i]/(Bk[i] - Ak[i]*sk[-1]))
                tk.append((Ak[i]*tk[-1] - Gk[i])/(Bk[i] - Ak[i]*sk[-1]))
            return sk, tk
        sk, tk = sk_tk()

        Вычисляем# значения решения на текущем слое
        uk_vals = [tk[N]]
        for i in range (N-1, -1, -1):
            uk_vals.append(sk[i]*uk_vals[-1] + tk[i])
        return list(reversed(uk_vals))
        ...
```

Осталось вычислить значения решения на каждом слое последовательно. Закончим реализацию метода:

```
    def impl_schma(sgm, N, M, h, tau, D):
    x = lambda i: D[0][0] + h*i
    t = lambda i: D[1][0] + tau*i

    u_vals = []
    Значения# решения на нулевом слое
    def find_u0i():
        return [phi(x(i)) for i in range(0, N+1)]

    def t_(k):
        if sgm == 0.5: return t(k) - tau/2
        if sgm == 0: return t(k-1)
        return t(k)
        ...
    Вычисляем# значения решеня на каждом слое
    u_vals.append(find_u0i())
    for k in range(1, M+1):
        u_vals.append(find_uki(k))
    return u_vals
```

Для тестирования понадобится вычилислить таблицу точного решения $u(x,t) = x^3 + t^3$

```
Строит
# таблицу со значениями точного решения
def build_acc_vals(N, M, h, tau, D):
    x = lambda i: D[0][0] + h*i
    t = lambda i: D[1][0] + tau*i

    return [[u(x(i), t(k)) for i in range(0, N+1)] for k in range(0, M+1)
   ]
```

Напечатаем его ”крупную”сетку

```
def print_acc_sol_large_net(N, M, D):
    printТочное(” решение на крупной сетке:”)
    tau = (D[1][1] -  D[1][0])/M
    h = (D[0][1] -  D[0][0])/N
    t_data = build_acc_vals(N, M, h, tau, D)
    print(pd.DataFrame(t_data, index = [ f”{k*tau}” for k in range(0, M
   +1)], columns = [f”{i*h}” for i in range(0, N+1)]))
```

```
Точное решение на крупной сетке:
           0.0       0.2       0.4  0.6000000000000001       0.8       1.0
0.0   0.000000  0.008000  0.064000            0.216000  0.512000  1.000000
0.01  0.000001  0.008001  0.064001            0.216001  0.512001  1.000001
0.02  0.000008  0.008008  0.064008            0.216008  0.512008  1.000008
0.03  0.000027  0.008027  0.064027            0.216027  0.512027  1.000027
0.04  0.000064  0.008064  0.064064            0.216064  0.512064  1.000064
0.05  0.000125  0.008125  0.064125            0.216125  0.512125  1.000125
0.06  0.000216  0.008216  0.064216            0.216216  0.512216  1.000216
0.07  0.000343  0.008343  0.064343            0.216343  0.512343  1.000343
0.08  0.000512  0.008512  0.064512            0.216512  0.512512  1.000512
0.09  0.000729  0.008729  0.064729            0.216729  0.512729  1.000729
0.1   0.001000  0.009000  0.065000            0.217000  0.513000  1.001000
```

Также выведем на печать сетки для полученных решений:

```
Сетка
# решения через явный метод
def print_expl_large_net(N, M, D):

    print("\n\Крупнаян сетка для явной схемы:")
    tau = (D[1][1] -  D[1][0])/M
    h = (D[0][1] -  D[0][0])/N
    t_data = expl_sub(N, M, h, tau, D)
    print(pd.DataFrame(t_data, index = [ f"{k*tau}" for k in range(0, M+1)], columns = [f"{i*h}" for i in range(0, N+1)]))Сетка
# решения через неявный метод
def print_impl_schma_large_net(N, M, D, sgm):
    tau = (D[1][1] -  D[1][0])/M
    h = (D[0][1] -  D[0][0])/N
    t_data = impl_schma(sgm, N, M, h, tau, D)

    print("\n\Крупнаян сетка для неявной схемы:")
    print(f"sigma = {sgm}")
    print(pd.DataFrame(t_data, index = [ f"{k*tau}" for k in range(0, M+1)], columns = [f"{i*h}" for i in range(0, N+1)]))
```

```
Крупная сетка для явной схемы:
           0.0       0.2       0.4  0.6000000000000001       0.8       1.0
0.0   0.000000  0.008000  0.064000            0.216000  0.512000  1.000000
0.01 -0.010267  0.008400  0.064400            0.216400  0.512400  1.000001
0.02 -0.013775  0.005870  0.064803            0.216803  0.512613  1.000008
0.03 -0.016044  0.004033  0.064262            0.217134  0.512818  1.000027
0.04 -0.017863  0.002513  0.063642            0.217100  0.513004  1.000064
0.05 -0.019357  0.001224  0.062970            0.216960  0.513046  1.000125
0.06 -0.020632  0.000107  0.062324            0.216725  0.513047  1.000216
0.07 -0.021728 -0.000868  0.061712            0.216470  0.513024  1.000343
0.08 -0.022677 -0.001719  0.061155            0.216218  0.513012  1.000512
0.09 -0.023496 -0.002457  0.060661            0.215999  0.513029  1.000729
0.1  -0.024195 -0.003086  0.060243            0.215828  0.513092  1.001000
```

```
Крупная сетка для неявной схемы:
sigma = 0
            0.0       0.2       0.4  0.6000000000000001       0.8       1.0
0.0   0.000000  0.008000  0.064000            0.216000  0.512000  1.000000
0.01  0.008400  0.008400  0.064400            0.216400  0.512400  1.000001
0.02  0.011003  0.011003  0.064803            0.216803  0.512613  1.000008
0.03  0.012900  0.012900  0.065930            0.217134  0.512818  1.000027
0.04  0.014562  0.014562  0.067024            0.217725  0.513004  1.000064
0.05  0.016060  0.016060  0.068135            0.218354  0.513312  1.000125
0.06  0.017459  0.017459  0.069218            0.219054  0.513666  1.000216
0.07  0.018789  0.018789  0.070293            0.219783  0.514075  1.000343
0.08  0.020075  0.020075  0.071360            0.220545  0.514525  1.000512
0.09  0.021334  0.021334  0.072429            0.221334  0.515020  1.000729
0.1   0.022583  0.022583  0.073506            0.222154  0.515558  1.001000


Крупная сетка для неявной схемы:
sigma = 0.5
            0.0       0.2       0.4  0.6000000000000001       0.8       1.0
0.0   0.000000  0.008000  0.064000            0.216000  0.512000  1.000000
0.01  0.009339  0.009339  0.064571            0.216768  0.513041  1.000001
0.02  0.011275  0.011275  0.065460            0.217552  0.513670  1.000008
0.03  0.012812  0.012812  0.066532            0.218329  0.514151  1.000027
0.04  0.014120  0.014120  0.067620            0.219104  0.514583  1.000064
0.05  0.015279  0.015279  0.068672            0.219871  0.515006  1.000125
0.06  0.016335  0.016335  0.069680            0.220627  0.515437  1.000216
0.07  0.017316  0.017316  0.070646            0.221376  0.515882  1.000343
0.08  0.018244  0.018244  0.071580            0.222121  0.516348  1.000512
0.09  0.019133  0.019133  0.072491            0.222868  0.516840  1.000729
0.1   0.019999  0.019999  0.073390            0.223625  0.517366  1.001000


Крупная сетка для неявной схемы:
sigma = 1
            0.0       0.2       0.4  0.6000000000000001       0.8       1.0
0.0   0.000000  0.008000  0.064000            0.216000  0.512000  1.000000
0.01  0.010987  0.010987  0.065005            0.217069  0.513376  1.000001
0.02  0.013254  0.013254  0.066274            0.218166  0.514355  1.000008
0.03  0.015071  0.015071  0.067613            0.219249  0.515129  1.000027
0.04  0.016594  0.016594  0.068932            0.220297  0.515793  1.000064
0.05  0.017915  0.017915  0.070196            0.221303  0.516399  1.000125
0.06  0.019093  0.019093  0.071394            0.222270  0.516977  1.000216
0.07  0.020166  0.020166  0.072531            0.223202  0.517543  1.000343
0.08  0.021160  0.021160  0.073613            0.224107  0.518112  1.000512
0.09  0.022098  0.022098  0.074652            0.224994  0.518693  1.000729
0.1   0.022994  0.022994  0.075660            0.225873  0.519295  1.001000
```

Выведем на печать таблицы точностей найденных решений:

```
def print_expl_acc_check_table(D):
    printТаблица(" точности для явной схемы:")
    #h tau ||... || ||...||
    t_data = [[], [], [], []]
    M = 10
```

```
    tau =  (D[1][1] -  D[1][0])/M
    h_set = [0.2, 0.1, 0.05]
    N_set = list(map(lambda i: int((D[0][1] -  D[0][0])/i), h_set))
    for i in range(0, len(h_set)):

        tau, M = get_stable_tau(N_set[i], h_set[i], D)
        t_data[0].append(h_set[i])
        t_data[1].append(tau)
        t_data[2].append(snorm_diff(build_acc_vals(N_set[i], M, h_set[i], tau, D), expl_sub(N_set[i], M, h_set[i], tau, D), N, M))
        t_data[3] = None
    table = {
        ’h’: t_data[0],
        ’tau’: t_data[1],
        ’||J_ex - u(h, tau)||’: t_data[2],
        ’||u(h,tau) - u(2h, tau\’)||’:  t_data[3]
    }
    print(pd.DataFrame(data = table))


def print_impl_schma_acc_table(D):
    printТаблицы(” точности для неявной схемы:”)
    M = 10
    tau = (D[1][1] -  D[1][0])/M
    sigma_set = [1, 0.5, 0]
    h_set = [0.2, 0.1, 0.05]
    N_set = list(map(lambda i: int((D[0][1] -  D[0][0])/i), h_set))
    for sgm in sigma_set:
        t_data = [ [], [], [], []]
        for i in range(0, len(h_set)):
            t_data[0].append(h_set[i])
            t_data[1].append(tau)
            t_data[2].append(snorm_diff(build_acc_vals(N_set[i], M, h_set[i], tau, D), impl_schma(sgm, N_set[i], M, h_set[i], tau, D), N, M))
            t_data[3] = None
            table = {
                ’h’: t_data[0],
                ’tau’: t_data[1],
                ’||J_ex - u(h, tau)||’: t_data[2],
                ’||u(h,tau) - u(2h, tau\’)||’:  t_data[3]
        }
        print(f”\n\nsigma = {sgm}”)
        print(pd.DataFrame(data = table))
```

```
Таблица точности для явной схемы:
      h       tau  ||J_ex - u(h, tau)|| ||u(h,tau) - u(2h, tau')||
0  0.20  0.010000            0.025195                        None
1  0.10  0.002500            0.005929                        None
2  0.05  0.000625            0.001451                        None
Таблицы точности для неявной схемы:


sigma = 1
      h   tau  ||J_ex - u(h, tau)|| ||u(h,tau) - u(2h, tau')||
0  0.20  0.01            0.021994                        None
1  0.10  0.01            0.007934                        None
2  0.05  0.01            0.005301                        None


sigma = 0.5
      h   tau  ||J_ex - u(h, tau)|| ||u(h,tau) - u(2h, tau')||
0  0.20  0.01            0.018999                        None
1  0.10  0.01            0.006002                        None
2  0.05  0.01            0.005641                        None


sigma = 0
      h   tau  ||J_ex - u(h, tau)|| ||u(h,tau) - u(2h, tau')||
0  0.20  0.01        2.158298e-02                        None
1  0.10  0.01        2.933709e+01                        None
2  0.05  0.01        1.063887e+07                        None
```

6 Задание 6

6.1 Постановка задачи

Необходимо реализовать проекционный метод и метод коллокации для решения дифференциального уравнения вида

$$Ly = f(x)$$

с граничными условиями

$$\alpha_1 y(-1) - \alpha_2 y(-1)' = 0, |\alpha_1| + |\alpha_2| \neq , \alpha_1\alpha_2 \geqslant 0$$

$$\beta_1 y(1) + \beta_2 y(1)' = 0, |\beta_1| + |\beta_2| \neq , \beta_1\beta_2 \geqslant 0$$

6.2 Вариант задания

Вариант 8(2.9) из методички:

- Граничная задача

$$-\frac{4-x}{5-2x}u'' + \frac{1-x}{2}u' + \frac{1}{2}ln(3+x)u = 1 + \frac{x}{3}, , u(-1) = u(1) = 0$$

- Метод Галеркина
  Координатная система: $(1-x^2)P_i^{(1,1)}(x), i = 0, 1, 2, ...$
- Метод коллокации

6.3 Решение

Приближенное решение будем искать в виде:

$$u^n(x) = \sum_{i=1}^{n} c_i \omega_i(x)$$

Коэффициенты c_i разложения u^n по $\omega_i(x) = (1-x^2)P_i^{(1,1)}$ находим из решения системы:

$$\sum_{j=1}^{n} a_{ij} c_j = f_i, i = 1, 2, .., n$$

Способы построения матрицы $\{a_{ij}\}$ и вектора $F = (f_1, ..., f_n)$ определяются конкретным проекционным методом.
В методе Галеркина требуется условие ортогональности невязки $Lu^n - f$ всем координатным функциям.

$$Lu = p(x)u'' + q(x)u' + r(x)u$$

Определим соответсвующие функции в коде программы:

```
def p(x):
    return -(4-x)/(5-2*x)
def q(x):
    return +(1 - x)/2
def r(x):
    return 1/2*log(3+x)
def f(x):
    return 1 - x/3
```

Поэтому система уравнений может быть переписана следующим образом:

$$\sum_{j=1}^{n} (L\omega_j, \omega_i) c_j = (f, \omega_i)$$

$$L\omega_i = p(x)\omega_i''(x) + q(x)\omega_i'(x) + r(x)\omega_i(x)$$

$$(f, g) = \int_{-1}^{1} f(x)g(x)dx$$

Для вычисления значений координатных функций потребется реализовать частный случай многочленов Якоби $P_n^{(\alpha,\beta)} = P_n^{(k,k)}$
Воспользуемся рекурентным соотношением:

- $P_0^{(k,k)}(x) = 1$
- $P_1^{(k,k)}(x) = (k+1)x$

- $P_{n+2}^{(k,k)} = \frac{(n+k+2)(2n+2k+3)xP_{n-1}^{(k,k)}(x) - (n+k+2)(n+k+1)P_n^{(k,k)}(x)}{(n+2k+2)(n+2)}$

В коде:

```
Многочлены
# Якоби
def Jpoly(arg, n, k = 1):
    if n < 0:
        return 0
    poly = [1, (k+1)*arg]
    for i in range (2, n+1):
        temp = ((i+k)*(2*i+2*k-1)*arg*poly[i-1] - (i+k)*(i+k-1)*poly[i
    -2])/((i+2*k)*(i))
        poly.append(temp)
    return poly[n]
```

Определим координатные функции и их прозводные в программе:

```
Координатные
# функции
def omega(arg, n):
    k = 1
    return (1 - arg**2)*Jpoly(arg, n, 1)
#k>=1Производные
# координатных функий
def domega(arg, n):
    k = 1
    return -2*(n+1)*Jpoly(arg, n+1, k-1)

def ddomega(arg, n):
    k = 1
    return -2*(n+1)*(n+1 +2*(k-1)+1)/2*Jpoly(arg, n, k)
```

При вычислении производных использовались правила дифференцирования полиномов Якоби:

- $[P_n^{(k,k)}]' = \frac{n+2k+1}{2} P_{n-1}^{(k+1,k+1)}$
- $[(1-x^2)P_n^{(k,k)}]' = -2(n+1)(1-x^2)^{k-1}P_{n+1}^{(k-1,k-1)}$

Также следует определить действие $L\omega_i$:

```
def Lomega(j):
    return lambda arg: p(arg)*ddomega(arg, j) + q(arg)*domega(arg,j) + r(
    arg)*omega(arg, j)
```

Остается вычислить $\{a_{ij}\}$ и $F = (f_1,..,f_n)$ и решить систему $AC = F$.

```
Метод
# ГалеркинаМетод
# Галеркина
def Galerkin(n):
    def build_eq():
        F = [quad(lambda arg: f(arg)*omega(arg, i), -1, 1)[0] for i in
    range(1, n+1)]
        #for i in range(1, n+1):
        #    F.append(quad(lambda arg: f(arg)*omega(arg, i), -1, 1))
        def buildAi(i):
```

```
            return   [quad( lambda arg: Lomega(j)(arg)*omega(arg,i), -1,
    1)[0] for j in range(1, n+1)]
        A = [buildAi(i) for i in range(1, n+1)]
        return A, F
    A, F = build_eq()
    C = np.linalg.solve(A, F)
    return A, F, C
```

Для вычисления интегралов в скалярном произведении использовались встроенная функция $quad$ из модуля $scipy$

Остается только вычислить приближенное решение $u^n(x)$:

```
Вычисление
# приближенного решения
def build_solution( c):
    def u(arg):
        sum = 0
        for i in range(0, len(c)):
            sum += c[i]*omega(arg, i+1)
        return sum
    return u
```

В методе коллокаций требование ортогональности невязки координатным функциям заменяется на требование обращаться в ноль в некоторых точках $1 \leqslant t_1 < t_2, ..., t_{n-1} < t_n \leqslant 1$:

$$Lu^n(t_i) - f(t_i) = 0, , i = 1,.., n$$

Перепишем систему уравнений:

$$\sum_{j=1}^{n} (L\omega_j\big|_{x=t_i}) c_j = f(t_i)$$

$1 \leqslant t_1 < t_2, ..., t_{n-1} < t_n \leqslant 1$ — узлы коллокации. По рекомендации из методички будем использовать корни многочлена Чебышева первого рода:

```
Корни
# многочлена чебышева
def Chebyshev_roots(n):
    return [cos((2*k-1)/(2*n)*pi) for k in range(1, n+1)]
```

Решим систему уравнений:

```
def collocation(n):
    knots = Chebyshev_roots(n)
    F = [f(knot) for knot in knots]
    A = [ [Lomega(j)(knots[i-1]) for j in range(1, n+1)] for i in range
    (1, n+1)]
    C = np.linalg.solve(A, F)
    return A, F, C
```

Выведем на печать таблицы:

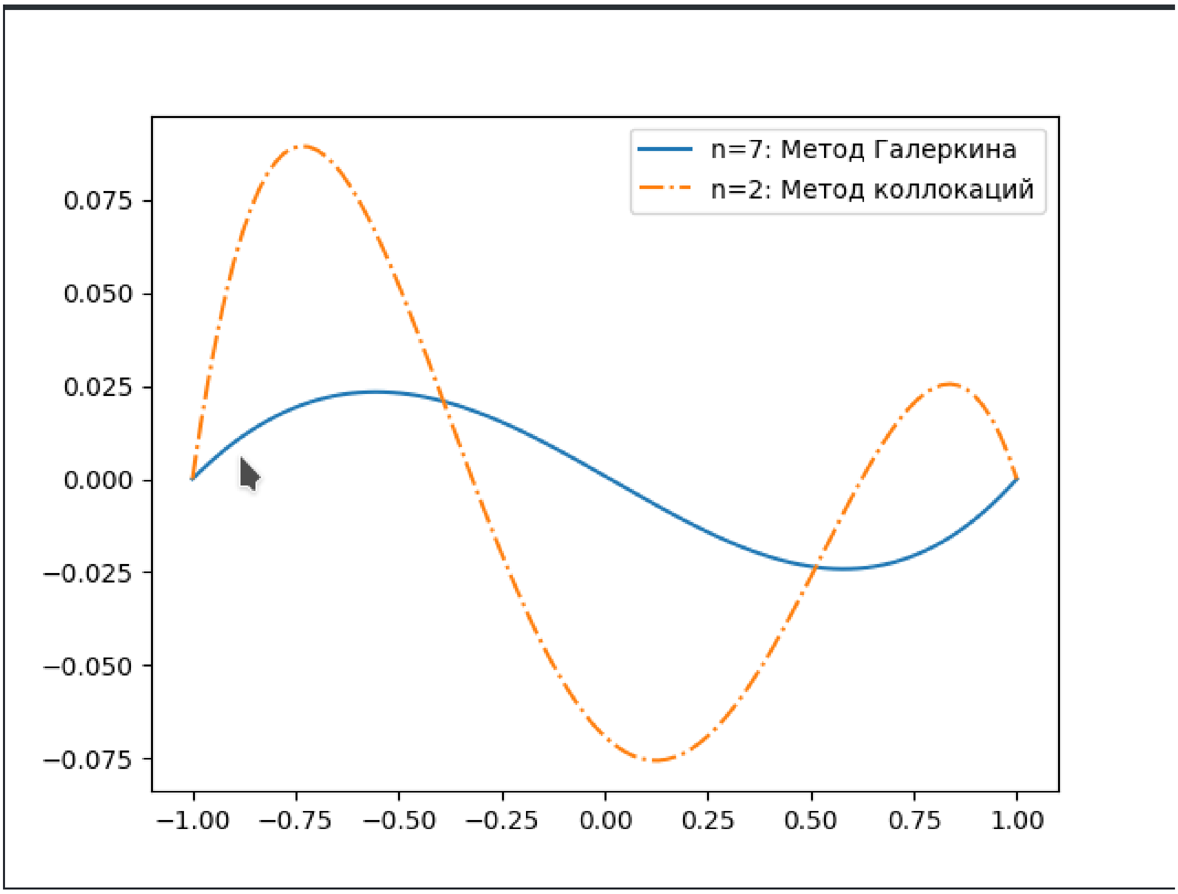
```
def print_table(f, n):
    conds = []
    args = [-0.5, 0, 0.5]
    vals = [[], [], []]
    for i in range(3, n+1):
```

```
        A, F, C = f(i)
        u = build_solution(C)
        conds.append(np.linalg.cond(A, np.infty))
        for k in range(0, 3):
            vals[k].append(u(args[k]))
    table = {
        'n': [i for i in range(3, n+1)],
        'cond(A)': conds,
        'u(-0.5)': vals[0],
        'u(0)': vals[1],
        'u(0.5)': vals[2]}
    print(pd.DataFrame(data = table))
```

```
метод Галеркина
   n   cond(A)   u(-0.5)      u(0)    u(0.5)
0  1  1.000000  0.023204  0.000000 -0.023204
1  2  1.835977  0.022871  0.000757 -0.023155
2  3  2.593218  0.023109  0.000856 -0.023430
3  4  3.315713  0.023097  0.000866 -0.023439
4  5  4.077312  0.023095  0.000867 -0.023438
5  6  4.840621  0.023095  0.000867 -0.023438
6  7  5.604827  0.023095  0.000867 -0.023438
Метод коллокаций
   n      cond(A)   u(-0.5)      u(0)    u(0.5)
0  1     1.000000 -0.750000  0.000000  0.750000
1  2     2.738189  0.052201 -0.069004 -0.026325
2  3    97.955136 -1.208319 -0.099879  1.245774
3  4   308.785446  1.043415 -1.800053  0.242493
4  5  2475.891462 -4.190627 -1.289439  5.133074
5  6  1278.101511 -1.754405  2.283630 -1.422314
6  7  9942.424462 -3.993299  1.173816  2.297436
```

Видно, что число обусловленности матрицы системы в методе коллокаций быстро растет, а следовательно вычисления становятся очень чувствительными к ошибкам округления.

Графики решений:



Выведем на печать расширенную матрицу системы, число обусловленности матрицы $cond_\infty(A)$, коэффициенты разложения приближенного решения по координатным функциям:

```
def print_info(A, F, C):
    printМатрица(" системы A:")
    print(np.array(A))
    print("\n Вектор F:")
    print(np.array(F))
    print(f"\Числоn обусловленности A: {np.linalg.cond(A, np.infty)}")
    printКоэффициенты(" разложения решения: ")
    print(np.array(C))

printМетод(" Галеркина: ")
for i in range(1, 7):
    print(f"\nn = {i}:")
    A, F, C = Galerkin( i)
    print_info(A, F, C)


print("\n\Методn коллокаций: ")
for i in range(1, 7):
    print(f"\nn = {i}:")
    A, F, C = collocation(i)
    print_info(A, F, C)
```

```
    Метод Галеркина:
```

```

n = 1:Матрица
 системы A:
[[5.74610509]]

 Вектор F:
[-0.17777778]Число

 обусловленности A: 1.0Коэффициенты
 разложения решения:
[-0.03093883]Значения

 решения в точках -0.5, 0, 0.5:
[ 0.02320412  0.         -0.02320412]

n = 2:Матрица
 системы A:
[[ 5.74610509  1.44889857]
 [-0.29111173  8.849551  ]]

 Вектор F:
[-1.77777778e-01 -2.77555756e-17]Число

 обусловленности A: 1.8359769645074955Коэффициенты
 разложения решения:
[-0.03068431 -0.00100938]Значения

 решения в точках -0.5, 0, 0.5:
[ 0.02287129  0.00075704 -0.02315518]

n = 3:Матрица
 системы A:
[[ 5.74610509  1.44889857 -0.29090203]
 [-0.29111173  8.849551    1.81360501]
 [ 0.26087245 -0.12334164 12.04144875]]

 Вектор F:
[-1.77777778e-01 -2.77555756e-17 -3.46944695e-18]Число

 обусловленности A: 2.593218297809861Коэффициенты
 разложения решения:
[-0.0306182  -0.00114075  0.00065164]Значения

 решения в точках -0.5, 0, 0.5:
[ 0.02310869  0.00085556 -0.02342952]

n = 4:Матрица
 системы A:
[[ 5.74610509e+00  1.44889857e+00 -2.90902025e-01  4.92459896e-02]
 [-2.91111735e-01  8.84955100e+00  1.81360501e+00 -2.34593796e-01]
 [ 2.60872450e-01 -1.23341643e-01  1.20414487e+01  2.14189737e+00]
 [-2.53190461e-03  3.68420337e-01  8.61630087e-02  1.52738070e+01]]

 Вектор F:
[-1.77777778e-01 -2.77555756e-17 -3.46944695e-18  5.37764278e-17]Число

```

```
 обусловленности A: 3.3157133070399563Коэффициенты
 разложения решения:
[-3.06188170e-02 -1.13959656e-03  6.48333470e-04  1.87552627e-05]Значения

 решения в точках -0.5, 0, 0.5:
[ 0.02309732  0.00086642 -0.02343871]

n = 5:Матрица
 системы A:
[[ 5.74610509e+00  1.44889857e+00 -2.90902025e-01  4.92459896e-02
   1.86555384e-02]
 [-2.91111735e-01  8.84955100e+00  1.81360501e+00 -2.34593796e-01
   6.57374513e-02]
 [ 2.60872450e-01 -1.23341643e-01  1.20414487e+01  2.14189737e+00
  -1.74201917e-01]
 [-2.53190461e-03  3.68420337e-01  8.61630087e-02  1.52738070e+01
   2.45199575e+00]
 [ 3.70280979e-03  9.19744618e-03  4.60128665e-01  3.16486602e-01
   1.85285352e+01]]

 Вектор F:
[-1.77777778e-01 -2.77555756e-17 -3.46944695e-18  5.37764278e-17
  1.73472348e-18]Число

 обусловленности A: 4.077311631537995Коэффициенты
 разложения решения:
[-3.06188695e-02 -1.13939916e-03  6.47917483e-04  2.03184747e-05
 -9.75255284e-06]Значения

 решения в точках -0.5, 0, 0.5:
[ 0.02309513  0.00086725 -0.0234382 ]

n = 6:Матрица
 системы A:
[[ 5.74610509e+00  1.44889857e+00 -2.90902025e-01  4.92459896e-02
   1.86555384e-02  4.40027616e-03]
 [-2.91111735e-01  8.84955100e+00  1.81360501e+00 -2.34593796e-01
   6.57374513e-02  2.16108256e-02]
 [ 2.60872450e-01 -1.23341643e-01  1.20414487e+01  2.14189737e+00
  -1.74201917e-01  8.05716595e-02]
 [-2.53190461e-03  3.68420337e-01  8.61630087e-02  1.52738070e+01
   2.45199575e+00 -1.13280040e-01]
 [ 3.70280979e-03  9.19744618e-03  4.60128665e-01  3.16486602e-01
   1.85285352e+01  2.75167580e+00]
 [ 3.56918789e-04  5.46569966e-03  2.12556068e-02  5.42345506e-01
   5.58804971e-01  2.17970317e+01]]

 Вектор F:
[-1.77777778e-01 -2.77555756e-17 -3.46944695e-18  5.37764278e-17
  1.73472348e-18 -2.77555756e-17]Число

 обусловленности A: 4.840620670415809Коэффициенты
 разложения решения:
[-3.06188693e-02 -1.13939939e-03  6.47918926e-04  2.03153256e-05
 -9.73759760e-06 -1.00580340e-07]Значения

```

```
 решения в точках -0.5, 0, 0.5:
[ 0.02309509  0.0008673  -0.02343824]Метод


 коллокаций:

n = 1:Матрица
 системы A:
[[1.]]

 Вектор F:
[1.]Число

 обусловленности A: 1.0Коэффициенты
 разложения решения:
[1.]Значения

 решения в точках -0.5, 0, 0.5:
[-0.75  0.    0.75]

n = 2:Матрица
 системы A:
[[ 8.10898629 12.92113511]
 [-7.37390867  9.23510633]]

 Вектор F:
[0.76429774 1.23570226]Число

 обусловленности A: 2.738189458164557Коэффициенты
 разложения решения:
[-0.05235048  0.09200485]Значения

 решения в точках -0.5, 0, 0.5:
[ 0.05220105 -0.06900364 -0.02632468]

n = 3:Матрица
 системы A:
[[ 10.09155383  23.95342817  37.69026262]
 [  1.          -7.61197961  -1.5       ]
 [-10.00835148  19.90311117 -28.52847817]]

 Вектор F:
[0.71132487 1.         1.28867513]Число

 обусловленности A: 97.95513605933759Коэффициенты
 разложения решения:
[ 1.36631625  0.13317249 -0.43159335]Значения

 решения в точках -0.5, 0, 0.5:
[-1.20831919 -0.09987937  1.24577395]

n = 4:Матрица
 системы A:
[[ 10.88500226  28.81130802  53.80614104  79.8122069 ]
 [  4.66693091  -1.35944571 -13.10626041 -10.70360514]
 [ -3.02993482  -3.71432477  12.24475653  -6.68684531]
```

```
 [-11.07314254  24.65833346 -42.45476623  60.06888503]]

 Вектор F:
[0.69204016 0.87243886 1.12756114 1.30795984]Число

 обусловленности A: 308.785446138744Коэффициенты
 разложения решения:
[-0.43581311  1.82085891  0.1570153  -0.69505467]Значения

 решения в точках -0.5, 0, 0.5:
[ 1.0434147  -1.80005335  0.2424932 ]

n = 5:Матрица
 системы A:
[[  11.27328564   31.26901584   62.46659382  101.95525096  143.56068849]
 [   6.76961814    6.53203348   -5.58580699  -22.34898737  -24.31021062]
 [   1.           -7.61197961   -1.5          15.34331634    1.875     ]
 [  -5.636604      3.23957267    7.70752253  -19.73951976   17.88445552]
 [ -11.59492745   27.06951024  -49.95478895   78.07306345 -106.97207662]]

 Вектор F:
[0.68298116 0.80407158 1.         1.19592842 1.31701884]Число

 обусловленности A: 2475.891461977278Коэффициенты
 разложения решения:
[ 2.20464985  1.28413944 -7.22251746 -0.52213442  3.06543506]Значения

 решения в точках -0.5, 0, 0.5:
[-4.1906269  -1.28943859  5.13307358]

n = 6:Матрица
 системы A:
[[  11.49022718   32.66388218   67.52188448  115.38976032  172.71561103
   233.2473696 ]
 [   8.10898629   12.92113511    7.08525327  -12.51726849  -35.83851385
   -43.42201709]
 [   3.4697738    -4.50385618  -11.72167955    1.23152853   21.50035125
    11.50829984]
 [  -1.63491351   -6.15708224    9.6987195     4.4667749   -20.36178172
     6.9180432 ]
 [  -7.37390867    9.23510633   -2.48844038  -13.62346617   30.31807568
   -32.92805063]
 [ -11.88650152   28.4387641   -54.33434985   89.0007846  -129.83886324
   172.35404581]]

 Вектор F:
[0.67802472 0.76429774 0.91372698 1.08627302 1.23570226 1.32197528]Число

 обусловленности A: 1278.1015114286386Коэффициенты
 разложения решения:
[ 0.06156303 -0.63604527 -0.28944561  2.06471978  0.12844281 -0.94381054]
   Значения

 решения в точках -0.5, 0, 0.5:
[-1.75440472  2.2836302  -1.42231389]
```

7 Репозиторий с кодом

Ссылка на репозиторий: https://github.com/rousewayse/Computational_methods