МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ «ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Компьютерная математика»

Тема: Решение уравнения Пуассона методом конечных элементов в двумерном пространстве.

Студентки гр. 0382	____________________	Деткова А.С.
		Здобнова К.Д.
Преподаватель	____________________	Коптелов Я.Ю.

Санкт-Петербург

2022

Цель работы.

Изучить метод конечных элементов на примере уравнения Пуассона.

Задание.

Методом конечных элементов построить аппроксимацию дифференциального уравнения Пуассона. Составить программу, отображающую аппроксимацию уравнения в двумерном пространстве.

Основные теоретические положения.

Уравнение Пуассона - эллиптическое дифференциальное уравнение в частных производных, описывающее электростатическое поле, стационарное поле температуры, поле давления, поле потенциала скорости в гидродинамике. Это уравнение имеет вид:

$$-\Delta \boldsymbol{u} = \boldsymbol{f}, \boldsymbol{x} \in \Omega$$

с граничным условием:

$$\boldsymbol{u}(\boldsymbol{x}) = \boldsymbol{0}, \boldsymbol{x} \in \Gamma(\boldsymbol{\partial}\Omega)$$

где

$$\Omega = \{\boldsymbol{x} = (\boldsymbol{x}, \boldsymbol{y}) : \ \boldsymbol{x}, \boldsymbol{y} \in (\boldsymbol{0}, \boldsymbol{1})\}$$

$$\boldsymbol{u}(\boldsymbol{x}) = \boldsymbol{u}(\boldsymbol{x}, \boldsymbol{y})$$

$$\Delta \boldsymbol{u} = \frac{\boldsymbol{\partial}^2 \boldsymbol{u}}{\boldsymbol{\partial} \boldsymbol{x}^2} + \frac{\boldsymbol{\partial}^2 \boldsymbol{u}}{\boldsymbol{\partial} \boldsymbol{y}^2}$$

Требуется найти функцию u, решающую заданное уравнение.

Выполнение работы.

1. Приведение уравнения к слабой форме.

Определим функциональное пространство: $\boldsymbol{V} := \boldsymbol{H}_{\boldsymbol{0}}^{\boldsymbol{1}}(\Omega)$

Введем также гладкую функцию $\boldsymbol{\phi}$ – непрерывная, кусочно непрерывно-дифференцируема, $\boldsymbol{\phi} \in \boldsymbol{V}$.

$$-\boldsymbol{u}'' = \boldsymbol{f}$$

$$-\int_{\Omega} \boldsymbol{u}''\boldsymbol{\phi}\boldsymbol{dx} = \int_{\Omega} \boldsymbol{f}\boldsymbol{\phi}\boldsymbol{dx}, \qquad \forall\boldsymbol{\phi} \in \boldsymbol{V}$$

Проинтегрируем левый интеграл по частям:

$$\int_{\Omega} \boldsymbol{u}'\boldsymbol{\phi}'\boldsymbol{dx} - \int_{\Omega} \boldsymbol{\partial_n u\phi ds} = \int_{\Omega} \boldsymbol{f}\boldsymbol{\phi}\boldsymbol{dx}, \qquad \forall\boldsymbol{\phi} \in \boldsymbol{V}$$

$$\int_{\Omega} \boldsymbol{u}'\boldsymbol{\phi}'\boldsymbol{dx} - \big(\boldsymbol{u}'(\boldsymbol{1})\boldsymbol{\phi}(\boldsymbol{1}) - \boldsymbol{u}'(\boldsymbol{0})\boldsymbol{\phi}(\boldsymbol{0})\big) = \int_{\Omega} \boldsymbol{f}\boldsymbol{\phi}\boldsymbol{dx}, \qquad \forall\boldsymbol{\phi} \in \boldsymbol{V}$$

Так как $\boldsymbol{\phi}$ принадлежит нашей области *V*, на которой определены граничные условия $\boldsymbol{u}(\boldsymbol{x}) = \boldsymbol{0}, \boldsymbol{x} \in \Gamma(\boldsymbol{\partial}\Omega)$, то $\boldsymbol{\phi}$ так же должны удовлетворять им, поэтому $\boldsymbol{\phi}(\boldsymbol{0}) = \boldsymbol{\phi}(\boldsymbol{1}) = \boldsymbol{0}$. Уберем обнулившиеся слагаемые из уравнения:

$$\int_{\Omega} \boldsymbol{u}'\boldsymbol{\phi}'\boldsymbol{dx} = \int_{\Omega} \boldsymbol{f}\boldsymbol{\phi}\boldsymbol{dx}, \qquad \forall\boldsymbol{\phi} \in \boldsymbol{V}$$

Теперь для удобства введем обозначение: $(\boldsymbol{f}, \boldsymbol{g}) := \int_{\Omega} \boldsymbol{f} \cdot \boldsymbol{g}\boldsymbol{dx}$, и получаем слабую формулу уравнения:

$$(\boldsymbol{u}', \boldsymbol{\phi}') = (\boldsymbol{f}, \boldsymbol{\phi}), \qquad \forall\boldsymbol{\phi} \in \boldsymbol{V}$$

2. Сетка.

В данной работе уравнение Пуассона рассматривается в ограниченной квадратной области 1 x 1. Но программа написана таким образом, что областью может быть прямоугольник произвольного размера.

Разбиваем нашу плоскость на сетку шага *h* – параметр может быть произвольным положительным числом, не более размера исходного поля.

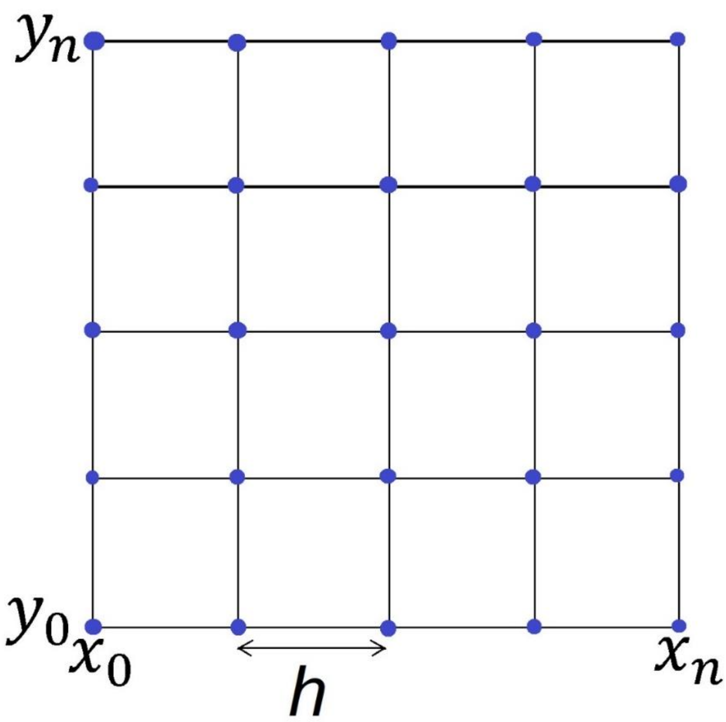


Рисунок 1. Разбиение квадратной области на конечные элементы

Конечный элемент K представляет собой квадрат (или прямоугольник), на котором задан полином $\boldsymbol{P_K}$ и координаты этого элемента в сетке. Полином $\boldsymbol{P_K}$ задается базисными функциями.

3. Базисные функции для одномерного случая.

Для понимания, как выбираются базисные функции в двумерном случае, рассмотрим для начала одномерный случай.

Нам нужно получить аппроксимацию $\boldsymbol{u_h(x) = \sum_{i=0}^{n} u_j \, \phi_j(x) \in V_h^{(1)}, u_j \in}$ $\mathbb{R}$

Базисные функции задаются соответствием:

$$\boldsymbol{\phi_j(x_i) = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}}$$

Это означает, что базисная функция не нулевая только на соответствующей ей точке конечного элемента.

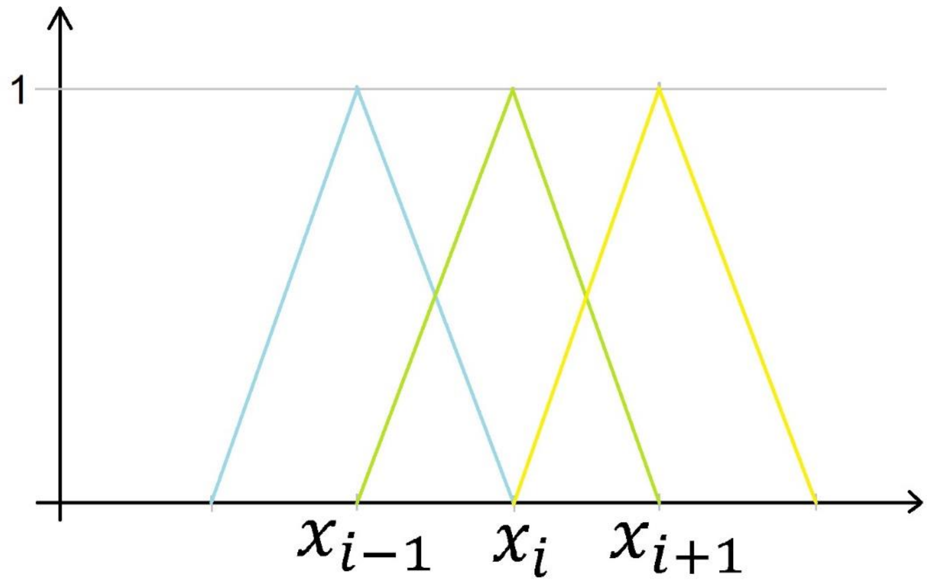


Рисунок 2. Пример построения базисных функций.

На рис. 2 представлены базисные функции $\boldsymbol{\phi_i(x) = x + i}$. Для данного базиса решение $\boldsymbol{u_h}(\boldsymbol{x})$ будет выглядеть следующим образом:

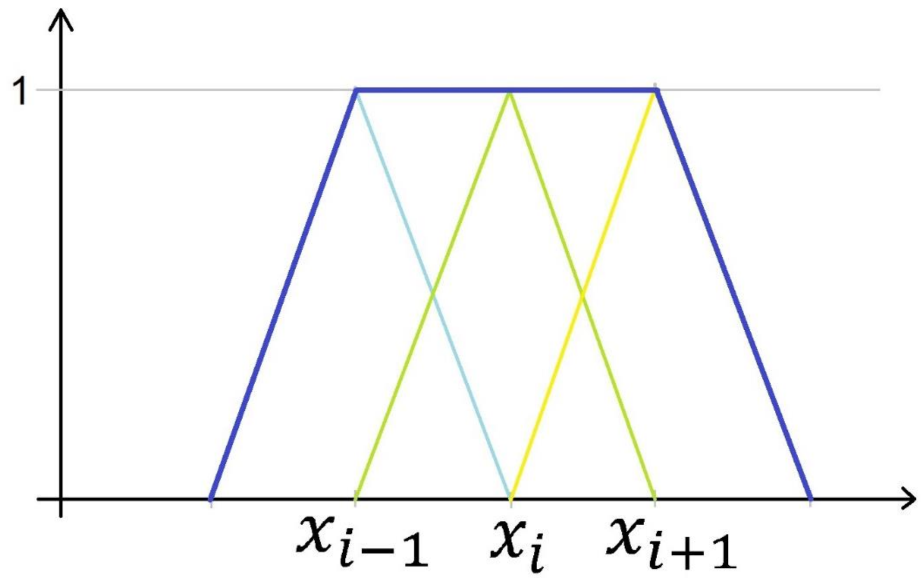


Рисунок 3. Пример решения $\boldsymbol{u_h}(\boldsymbol{x})$.

$$\boldsymbol{u_h} = \mathbf{1.00}\boldsymbol{\phi_1} + \mathbf{1.00}\boldsymbol{\phi_2} + \mathbf{1.00}\boldsymbol{\phi_3}$$

Пример решения с другими коэффициентами:

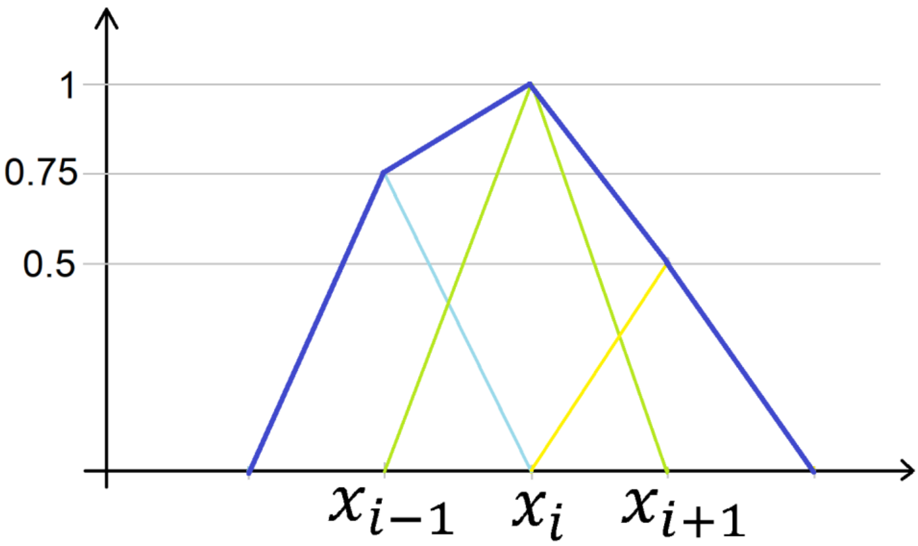


Рисунок 4.

$$u_h = 0.75\phi_1 + 1.00\phi_2 + 0.50\phi_3$$

4. Базисные функции для двумерного случая.

Для двумерного случая возьмем базисные функции: $z = x, z = y$.

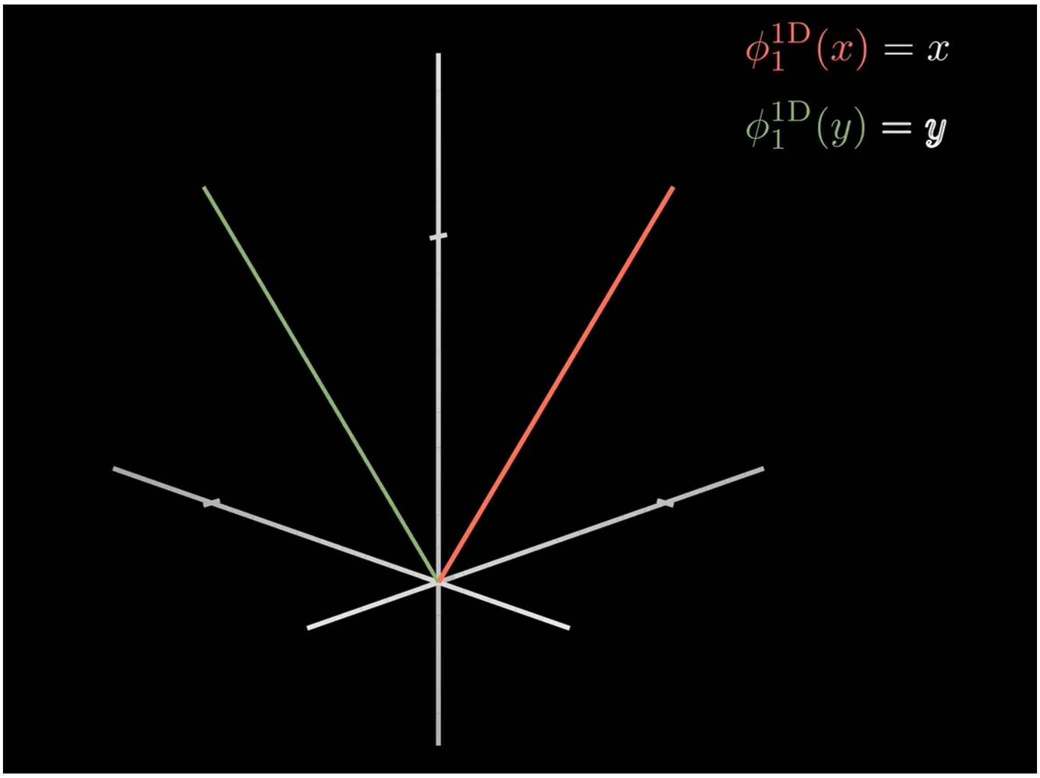


Рисунок 5.

Базисная ("шляпная") функция в 2D это произведение двух базисных функций из 1D.

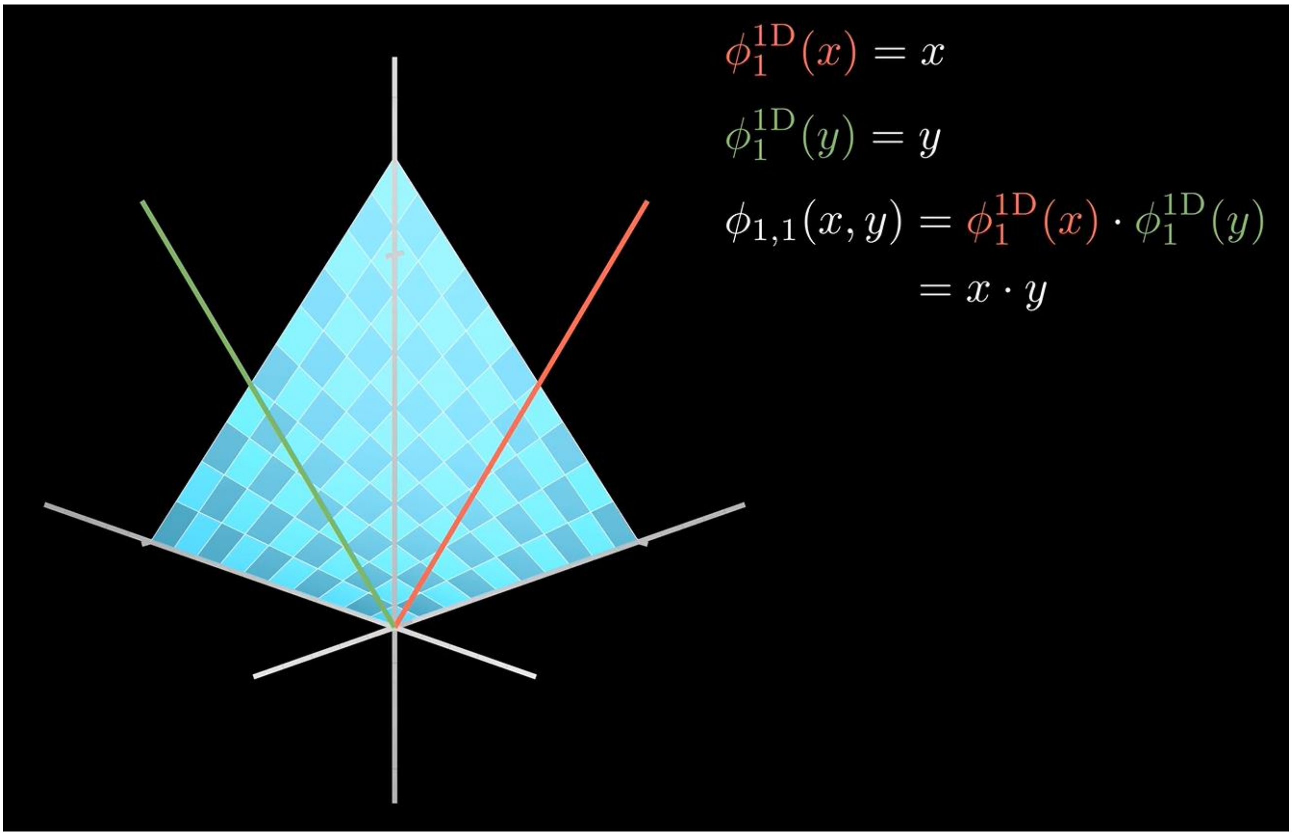


Рисунок 6.

5. Построение Матриц

Нам нужно найти решение уравнения:

$$(\boldsymbol{u}', \boldsymbol{\phi}') = (\boldsymbol{f}, \boldsymbol{\phi}), \qquad \forall \boldsymbol{\phi} \in \boldsymbol{V}$$

Так как мы не можем решить его в явном виде, мы будем искать аппроксимацию этого уравнения. Для этого для слабой формулы уравнения мы переходим к конечномерному подпространству $\boldsymbol{V_h} \subset \boldsymbol{V}$, областью решения будет $\boldsymbol{u_h}$.

$$(\boldsymbol{u_h}', \boldsymbol{\phi_h}') = (\boldsymbol{f}, \boldsymbol{\phi_h}), \qquad \forall \boldsymbol{\phi_h} \in \boldsymbol{V_h}$$

Так как решение $\boldsymbol{u_h}$ – это сумма точек КЭ, то:

$$\leftrightarrow \left(\left(\sum_{j=1}^{n} \boldsymbol{u_j} \boldsymbol{\phi_j} \right)', \boldsymbol{\phi_h'} \right) = (\boldsymbol{f}, \boldsymbol{\phi_h}), \qquad \forall \boldsymbol{\phi_h} \in \boldsymbol{V_h}$$

$$\leftrightarrow \sum_{j=1}^{n} \boldsymbol{u_j} (\boldsymbol{\phi_j'}, \boldsymbol{\phi_h'}) = (\boldsymbol{f}, \boldsymbol{\phi_h}), \qquad \forall \boldsymbol{\phi_h} \in \boldsymbol{V_h}$$

$$\leftrightarrow \sum_{j=1}^{n} \boldsymbol{u_j}(\boldsymbol{\phi}'_j, \boldsymbol{\phi}'_i) = (\boldsymbol{f}, \boldsymbol{\phi_i}), \qquad \forall\, \mathbf{1} \le \boldsymbol{i} \le \boldsymbol{n}$$

Это уравнение перепишем в матричный вид:

$$\underbrace{\begin{pmatrix} (\boldsymbol{\phi}'_1, \boldsymbol{\phi}'_1) & (\boldsymbol{\phi}'_1, \boldsymbol{\phi}'_2) & \cdots & (\boldsymbol{\phi}'_1, \boldsymbol{\phi}'_n) \\ (\boldsymbol{\phi}'_2, \boldsymbol{\phi}'_1) & (\boldsymbol{\phi}'_2, \boldsymbol{\phi}'_2) & & (\boldsymbol{\phi}'_2, \boldsymbol{\phi}'_n) \\ \vdots & & \ddots & \vdots \\ (\boldsymbol{\phi}'_n, \boldsymbol{\phi}'_1) & (\boldsymbol{\phi}'_n, \boldsymbol{\phi}'_2) & \cdots & (\boldsymbol{\phi}'_n, \boldsymbol{\phi}'_n) \end{pmatrix}}_{=\boldsymbol{A}} \underbrace{\begin{pmatrix} \boldsymbol{u_1} \\ \boldsymbol{u_2} \\ \vdots \\ \boldsymbol{u_n} \end{pmatrix}}_{=\boldsymbol{U}} = \underbrace{\begin{pmatrix} (\boldsymbol{f}, \boldsymbol{\phi_1}) \\ (\boldsymbol{f}, \boldsymbol{\phi_2}) \\ \vdots \\ (\boldsymbol{f}, \boldsymbol{\phi_n}) \end{pmatrix}}_{=\boldsymbol{F}}$$

Решением уравнения будет вектор *U*.

Теперь нужно определить элементы матриц *A* и *F*. Для левой матрицы так как $\boldsymbol{\phi}'_i = \frac{\partial \phi_i}{\partial x}\vec{\boldsymbol{\imath}} + \frac{\partial \phi_i}{\partial y}\vec{\boldsymbol{\jmath}} + \frac{\partial \phi_i}{\partial z}\vec{\boldsymbol{k}}$, то ее элемент определяется:

$$\boldsymbol{A_{ij}} = (\boldsymbol{\phi}'_i, \boldsymbol{\phi}'_j) = \left(\nabla\boldsymbol{\phi_i}, \nabla\boldsymbol{\phi_j}\right) \coloneqq \int\limits_{\Omega} \nabla\boldsymbol{\phi_i} \cdot \nabla\boldsymbol{\phi_j}\boldsymbol{dx}$$

Где $\boldsymbol{\nabla}$ - оператор набла. Для правой матрицы *F*:

$$\boldsymbol{F_i} = (\boldsymbol{f}, \boldsymbol{\phi_i}) \coloneqq \int\limits_{\Omega} \boldsymbol{f\phi_i dx}$$

Выразив из уравнения вектор *U* получим, что $\boldsymbol{U} = \boldsymbol{A^{-1}F}$, тогда решением дискретной слабой формы будет:

$$\boldsymbol{u_h} = \sum_{i=1}^{n} \boldsymbol{u_i\phi_i} = \sum_{i=1}^{n} (\boldsymbol{A^{-1}F})_{\boldsymbol{i}}\boldsymbol{\phi_i}$$

6. Сборка и решение системы линейных уравнений.

Метод конечных элементов заключается в том, что вместо интегрирования по области Ω, эта область разбивается на отдельные ячейки *K*, по которым и будет происходить интегрирование, а затем результат суммируется. В формулах это означает, что

$$\int\limits_{\Omega} \ldots \boldsymbol{dx} = \sum_{\boldsymbol{cell\,K} \in \Omega} \int\limits_{\boldsymbol{K}} \ldots \boldsymbol{dx}$$

Так как базисная функция $\boldsymbol{\phi_i}$ равна 1 на i конечном элементе и 0 на всех остальных элементах, то в программе мы будем высчитывать только ненулевые элементы. Это означает, что $\boldsymbol{F(j)} \mathrel{+}= \int_{\boldsymbol{K}} \boldsymbol{f\phi_j dx}$ и $\boldsymbol{A(i,j)} \mathrel{+}= \int_{\boldsymbol{K}} \boldsymbol{\phi_i' \cdot \phi_j' \, dx}$.

Теперь заменим подынтегральные функции на более простые, чтобы можно было их посчитать. Для этого используем формулу Симпсона:

$$\int_a^b g(x)dx \approx \left(\frac{b-a}{6}\right) \cdot \left(g(a) + 4g\left(\frac{a+b}{2}\right) + g(b)\right)$$

Для того, чтобы уметь численно считать интегралы по формуле Симпсона мы должны для каждого конечного элемента перейти к мастер-элементу.

$$J = h^2$$

Для матрицы F:

$$\int_K f(x)\phi_j^K(x)dx = \int_M f(T(\xi))\phi_j^M(\xi)h^2 d\xi$$

Для матрицы A:

$$\int_K \nabla_x \phi_i^K(x) \cdot \nabla_x \phi_j^K(x)dx = \int_M \nabla_\xi \phi_i^M(\xi) \cdot \nabla_\xi \phi_j^M(\xi)d\xi$$

Где мастер-элемент $\boldsymbol{M} \coloneqq (\boldsymbol{0}, \boldsymbol{1}) \times (\boldsymbol{0}, \boldsymbol{1})$ и T – аффинное преобразование из M к определенной клетке K.

На рисунке 7 показана биекция перехода из КЭ к мастер-элементу.

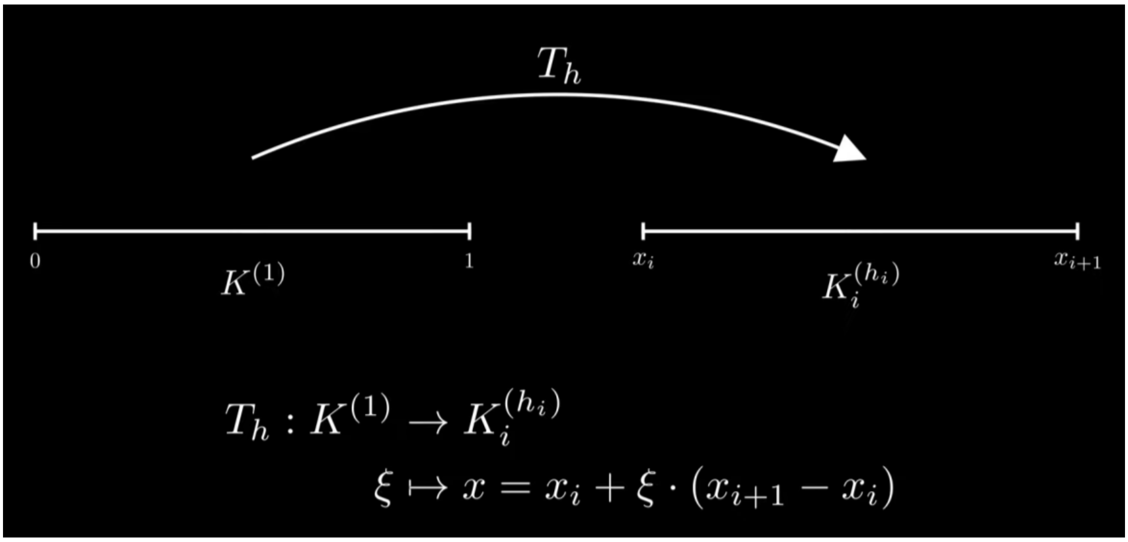


Рисунок 7.

7. LUP – разложение.

Для нахождения вектора U нужно решить систему линейных алгебраических уравнений. В данной работе применяется LUP-разложение, так как оно устойчивое – в алгоритме не используется нахождение обратной матрицы, что реализуемо только для вырожденных матриц.

СЛАУ – это матричное уравнение с матрицей A и векторами x и b.

$$\boldsymbol{Ax = b}$$

Для решения уравнения мы будем использовать LUP метод с поворотом для разложения нашей матрицы A в $PA = LU$, где L – нижняя треугольная матрица, U – верхняя треугольная матрица, P – матрица перестановок.

Для вычисления элементов верхней треугольной матрицы мы используем следующую формулу:

$$\boldsymbol{u_{ij} = a_{ij} - \sum_{k=1}^{i-1} u_{kj} l_{ik}}$$

Для вычисления значений элементов нижней диагональной матрицы используется формула:

$$\boldsymbol{l_{ij} = \frac{1}{u_{jj}} (a_{ij} - \sum_{k=1}^{j-1} u_{kj} l_{ik})}$$

Чтобы гарантировать численную стабильность алгоритма при $\boldsymbol{u_{jj} = 0}$ используется матрица перестановок P (иначе может возникнуть ситуация деления на 0). Каждый раз, меняя строчки в A происходит перестановка строчек в P.

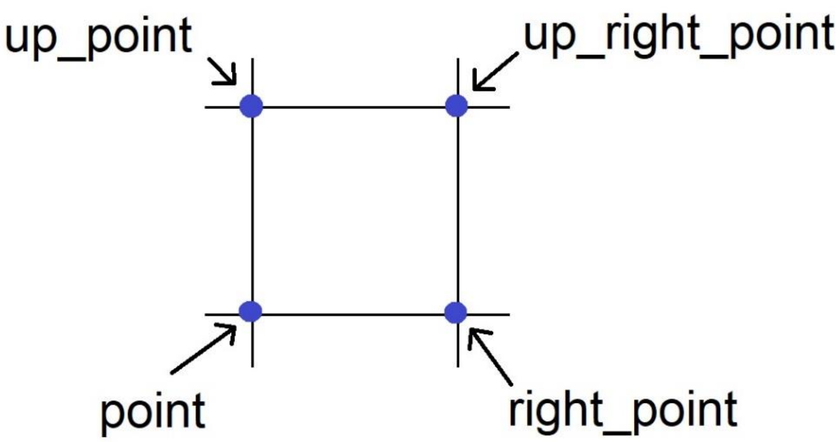
8. Написание кода.

- В <u>классе *Point*</u> хранится информация о точке сетки – координаты x, y и порядковый номер точки *ind*.

```
class Point:
    def __init__(self, x, y, ind):
        self.x, self.y, self.index = x, y, ind
```

- В <u>классе *Cell*</u> хранится информация о 4-х точках клетки: левая нижняя – главная. Точки хранятся в виде списка.

```
class Cell:
    def __init__(self, point, right_point, up_point, up_right_point):
        self.points = [point, right_point, up_point, up_right_point]
```



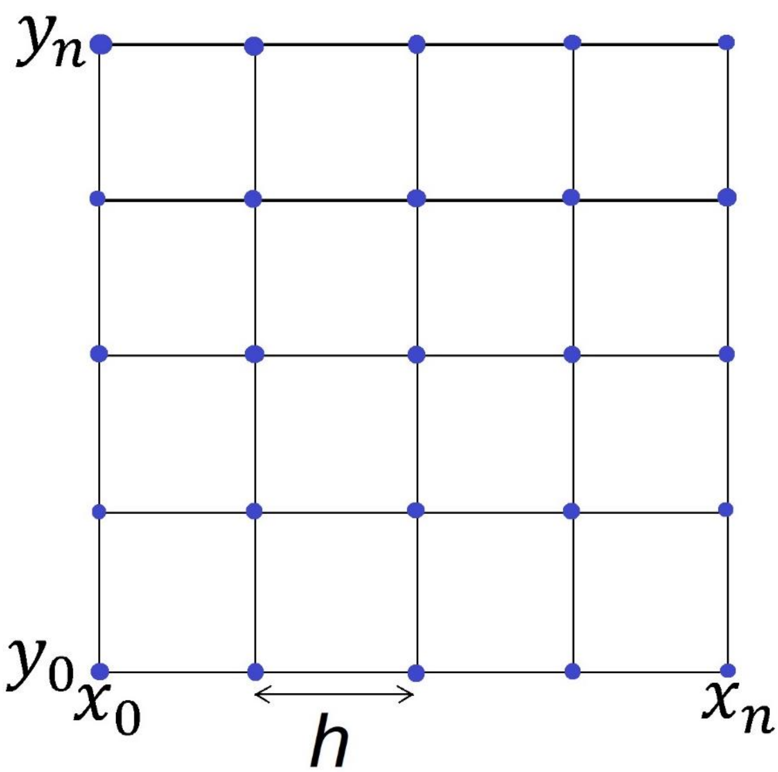
- В <u>классе *Mesh*</u> задается квадратная сетка.

```python
class Mesh:
    def __init__(self, xMin=0.0, xMax=1.0, yMin=0.0, yMax=1.0, stepSize=0.5):
        self._xNum, self._yNum = 0, 0
        self._xMesh, self._yMesh = [], []
        self._xMin, self._xMax, self._yMin, self._yMax = xMin, xMax, yMin, yMax
        self._h = stepSize
        self._points, self._cells = [], []
```

Поля *_xMin*, *_xMax*, *_yMin*, *_yMax* – размеры сетки; *_h* – шаг сетки; *_xNum*, *_yNum* – количество точек по оси *x* и *y*; *_xMesh*, *_yMesh* – списки точек по *x* и *y*; *_points* – список точек; *_cells* – список клеток.

Функция класса *get_coordinates()* находит координаты сетки для заданного шага *h*, округляя до последнего несущего знака после запятой шага *h*. Получаем два списка значений точек $x_0, x_1, \ldots, x_n$ и $y_0, y_1, \ldots, y_n$.

```python
def get_coordinates(self, coord_list, cMin, cMax, rounding):
    ind = cMin
    while ind < cMax:
        coord_list.append(round(ind, rounding))
        ind += self._h
    if coord_list[len(coord_list) - 1] < cMax:
        coord_list.append(cMax)
```



В функции *createMesh()* строится сама сетка.

Сначала решаем проблему с точностью вещественных чисел – находим количество знаков после запятой шага *h*:

```
def createMesh(self):
    h_tmp = self._h
    sights = 0
    while h_tmp % 10 > 0:
        h_tmp *= 10
        sights += 1
```

Далее используем функцию *get_coordinates()*, чтобы найти точки сетки. В поля *_xNum*, *_yNum* записываем количество точек по оси *x* и *y*:

```
x_coordinates = []
self.get_coordinates(x_coordinates, self._xMin, self._xMax, sights)
self._xNum = len(x_coordinates)
y_coordinates = []
self.get_coordinates(y_coordinates, self._yMin, self._yMax, sights)
self._yNum = len(y_coordinates)
```

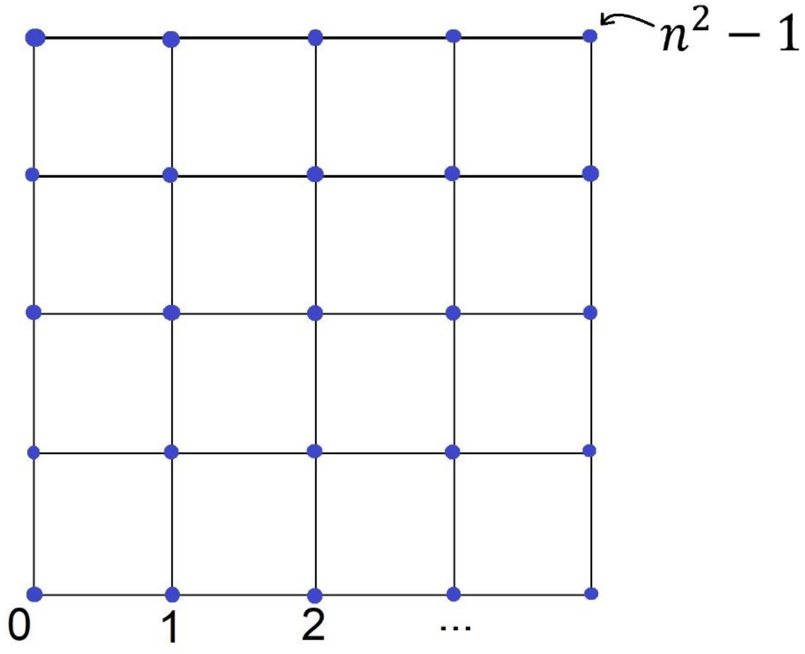
Затем заполним списки *_xMesh* и *_yMesh* получившимися значениями:

```
for ind in range(self._yNum):
    self._xMesh.append(x_coordinates)
    self._yMesh.append([y_coordinates[ind]] * self._xNum)
```

То есть *_xMesh* будет выглядеть как $[[\boldsymbol{x_0}, \boldsymbol{x_1}, \dots, \boldsymbol{x_n}], \dots, [\boldsymbol{x_0}, \boldsymbol{x_1}, \dots, \boldsymbol{x_n}]]$, *_yMesh* - $[[\boldsymbol{y_0}, \boldsymbol{y_0}, \dots, \boldsymbol{y_0}], \dots, [\boldsymbol{y_n}, \boldsymbol{y_n}, \dots, \boldsymbol{y_n}]]$.

Теперь заполним поле *_points* значениями узлов сетки: *(x, y, ind)*

```
for i in range(len(self._yMesh)):
    for j in range(self._xNum):
        self._points.append(Point(self._xMesh[i][j], self._yMesh[i][j], i * self._xNum + j))
```



У узла с *ind=0* координаты $(\boldsymbol{x_0}, \boldsymbol{y_0})$, *ind=1* будут $(\boldsymbol{x_1}, \boldsymbol{y_0})$ и так далее.

Теперь осталось заполнить поле *_cells* – информация о четырех точках для каждого прямоугольника из сетки.

```
for i, point in enumerate(self._points):
    if point.x != self._xMax and point.y != self._yMax:
        self._cells.append(
            Cell(
                point,
                self._points[i + 1],
                self._points[i + self._xNum],
                self._points[i + 1 + self._xNum]
            )
        )
```

Например, для левого нижнего треугольника значения будут следующими: *point* = $(\boldsymbol{x_0}, \boldsymbol{y_0}, \boldsymbol{0})$, *right_point* = $(\boldsymbol{x_1}, \boldsymbol{y_0}, \boldsymbol{1})$, *up_point* = $(\boldsymbol{x_0}, \boldsymbol{y_1}, \boldsymbol{n})$, *up_right_point* = $(\boldsymbol{x_1}, \boldsymbol{y_1}, \boldsymbol{n+1})$.

- Теперь нужно разобраться с базисными функциями.

Сначала введем базисные функции для одномерного случая (т.к. базисная «шляпная» функция в 2D это произведение двух базисных функций из 1D:

```
hat0 = {"eval": lambda x: x, "nabla": lambda x: 1}
hat1 = {"eval": lambda x: 1 - x, "nabla": lambda x: -1}
hatFunction = [hat0, hat1]
```

Eval – сама функция, *nabla* – производная от функции.

Теперь создадим класс *Basis2D* – базисные функции для двумерного случая. Сделаем два поля *xBasis* и *yBasis* – для базисных функций по *x* и *y*.

```
def __init__(self, x0, y0):
    self.xBasis = hatFunction[x0]
    self.yBasis = hatFunction[y0]
```

Функция *evaluation()* возвращает значение базисной в точке *(x, y).*

```
def evaluation(self, x, y):
    return self.xBasis["eval"](x) * self.yBasis["eval"](y)
```

Функция *nabla()* дает производную базисной функции по x и по y. Возвращает список, нулевой элемент которого - частная производная базисной функции по x, первый элемент - частная производная по y.

```
def nabla(self, x, y):
    return [self.xBasis["nabla"](x) * self.yBasis["eval"](y),
            self.yBasis["nabla"](y) * self.xBasis["eval"](x)]
```

Функция *scalar()* возвращает произведение двух базисных функций.

```
def scalar(self, x_self, y_self, other, x_other, y_other):
    return (self.nabla(x_self, y_self)[0] * other.nabla(x_other, y_other)[0] +
            self.nabla(x_self, y_self)[1] * other.nabla(x_other, y_other)[1])
```

Таким образом, линейная базисная функция в 2D это будет комбинации *xy*, *(1-x)y*, *x(1-y)*, *(1-x)(1-y).*

```
basisFunction = {0: Basis2D(0, 0), 1: Basis2D(1, 0), 2: Basis2D(0, 1), 3: Basis2D(1, 1)}
```

- В функции *createSystem()* вычисляются значения матриц *A* и *F*.

Сначала создадим пустые матрицы: *A* – двумерная, *F* – одномерная:

```
a = [[0 for _ in range(len(self._points))] for _ in range(len(self._points))]
_f = [0] * len(self._points)
```

Далее посчитаем значения матриц. Как было написано в теории ранее вместо нахождения каждого элемента матрицы с помощью базиса конкретного конечного элемента, делается переход к мастер-элементу. На псевдокоде это будет выглядеть следующим образом:

```
for element K in mesh.elements:
    for DoF j in K.DoFs:
        F(j) += $\int_K f\phi_j\,dx$
        for DoF i in K.DoFs:
            A(i,j) += $\int_K \phi_i'\phi_j'\,dx$
```

Цикл проходит по всем конечным элементам, а в каждом КЭ проходим по всем его 4-м точкам. Значения интегралов считаем с помощью правила Симпсона.

```
for cell in self._cells:
    for j, point_j in enumerate(cell.points):
        _f[point_j.index] += (((1 / 6) ** 2) * f * (self._h ** 2) *
                              (basisFunction[j].evaluation(0, 0) + 4 * basisFunction[j].evaluation(0, 1 / 2) +
                               basisFunction[j].evaluation(0, 1) + 4 * basisFunction[j].evaluation(1 / 2, 0) +
                               16 * basisFunction[j].evaluation(1 / 2, 1 / 2) +
                               4 * basisFunction[j].evaluation(1 / 2, 1) + basisFunction[j].evaluation(1, 0) +
                               4 * basisFunction[j].evaluation(1, 1 / 2) + basisFunction[j].evaluation(1, 1)))
        for i, point_i in enumerate(cell.points):
            a[point_i.index][point_j.index] += (
                    ((1 / 6) ** 2) * (basisFunction[i].scalar(0, 0, basisFunction[j], 0, 0) +
                                      4 * basisFunction[i].scalar(0, 1 / 2, basisFunction[j], 0, 1 / 2) +
                                      basisFunction[i].scalar(0, 1, basisFunction[j], 0, 1) +
                                      4 * basisFunction[i].scalar(1 / 2, 0, basisFunction[j], 1 / 2, 0) +
                                      16 * basisFunction[i].scalar(1 / 2, 1 / 2, basisFunction[j], 1 / 2,
                                                                   1 / 2) +
                                      4 * basisFunction[i].scalar(1 / 2, 1, basisFunction[j], 1 / 2, 1) +
                                      basisFunction[i].scalar(1, 0, basisFunction[j], 1, 0) +
                                      4 * basisFunction[i].scalar(1, 1 / 2, basisFunction[j], 1, 1 / 2) +
                                      basisFunction[i].scalar(1, 1, basisFunction[j], 1, 1))
```

Осталось применить граничное условие Дирихле – в матрице A граничные точки = 0, на главной диагонали 1. В матрице F граничные точки также равны 0:

```
for point in self._points:
    if point.x in [self._xMin, self._xMax] or point.y in [self._yMin, self._yMax]:
        for j in range(len(self._points)):
            a[point.index][j] = 0.0
        a[point.index][point.index] = 1.0
        _f[point.index] = 0.0

return a, _f
```

$$\boldsymbol{A} = \begin{pmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ \boldsymbol{a_{21}} & \mathbf{1} & \boldsymbol{a_{23}} & & \boldsymbol{a_{2n}} \\ & \vdots & & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{1} \end{pmatrix}, \boldsymbol{F} = (\mathbf{0}, \boldsymbol{f_2}, \dots, \boldsymbol{f_{n-1}}, \mathbf{0})$$

- Функция для LUP-разложения - *lup_decomposition():*

```python
def lup_decomposition(a):
    n = len(a)
    l = [[0.0] * n for i in range(n)]
    u = [[0.0] * n for i in range(n)]
    p = [[float(i == j) for i in range(n)] for j in range(n)]
    pa = multiply_matrix(p, a)
    for j in range(n):
        max_elem = a[j][j]
        max_ind = (j, j)
        for k in range(j, n):
            if abs(a[k][j]) > max_elem:
                max_elem = abs(a[k][j])
                max_ind = (k, j)
        pa[j], pa[max_ind[0]] = pa[max_ind[0]], pa[j]
        p[j], p[max_ind[0]] = p[max_ind[0]], p[j]
        u[j], u[max_ind[0]] = u[max_ind[0]], u[j]
        l[j], l[max_ind[0]] = l[max_ind[0]], l[j]
        l[j][j] = 1.0
        for i in range(j + 1):
            s1 = sum(u[k][j] * l[i][k] for k in range(i))
            u[i][j] = pa[i][j] - s1
        for i in range(j, n):
            s2 = sum(u[k][j] * l[i][k] for k in range(j))
            l[i][j] = (pa[i][j] - s2) / u[j][j]
    return p, l, u
```

Сначала создаем пусты матрицы *l* и *u*, затем создаем матрицу перестановок *p* с единицами на главной диагонали. Матрица *pa* – произведение матрицы *p* на *a* – это сама матрица *a*, т.к. *p* – единичная матрица. Затем в цикле делаем само разложение – в каждом столбце ставим максимальный по модулю элемент на главную диагональ, далее делаем перестановки в матрице *pa*, *p*, *u* и *l*. В последних двух вложенных циклах вычисляются значения элементов матриц *u* и *l*. Функция возвращает матрицы *p*, *l* и *u*.

Также написаны функции *Ly_b()*, *Ux_y()*, *multiply_matrix()*, *multiply_matrix_and_vector()*.

Функция *Ly_b()* решает систему $ly = b$, находит значения вектора *y*, где *l* – нижняя треугольная матрица с единицами на главной диагонали. Размерности матриц не проверяются, так как в нашей задаче размерности подходят.

```python
def Ly_b(l, b):
    y = [0.0 for i in range(len(b))]
    # yi = (bi - sum(lij * yj)) / lii, проходим по матрице l сверху вниз
    for i in range(len(l)):
        summa = sum(l[i][j] * y[j] for j in (range(i)))
        y[i] = (b[i] - summa) / l[i][i]
    return y
```

Функция *Ux_y()* решает систему *ux* = *y*, находит значения вектора *x*, *u* – верхняя треугольная матрица.

```python
def Ux_y(u, y):
    x = [0.0 for i in range(len(y))]
    # xi = (yi - sum(uij * xj)) / uii, проходим по матрице u снизу вверх
    for i in range(len(u) - 1, -1, -1):
        summa = sum(u[i][j] * x[j] for j in range(len(u) - 1, i, -1))
        x[i] = (y[i] - summa) / u[i][i]
    return x
```

Функции *multiply_matrix()* реализована для вычисления произведения двух квадратных матриц *m* и *n* одинакового размера.

```python
def multiply_matrix(m, n):
    size = len(m)
    result = [[0 for _ in range(len(n))] for _ in range(size)]

    for i in range(size):
        for j in range(len(n)):
            summa = 0
            for k in range(size):
                summa += m[i][k] * n[k][j]
            result[i][j] = summa

    return result
```

Функция *multiply_matrix_and_vector()* реализована для произведения матрицы *m* на вектор *vec*.

```python
def multiply_matrix_and_vector(m, vec):
    size = len(m)
    result = [0 for _ in range(len(vec))]
    for i in range(size):
        summa = 0
        for j in range(len(vec)):
            summa += m[i][j] * vec[j]
        result[i] = summa
    return result
```

Отрисовка результатов происходит с помощью библиотеки *pyplot*:

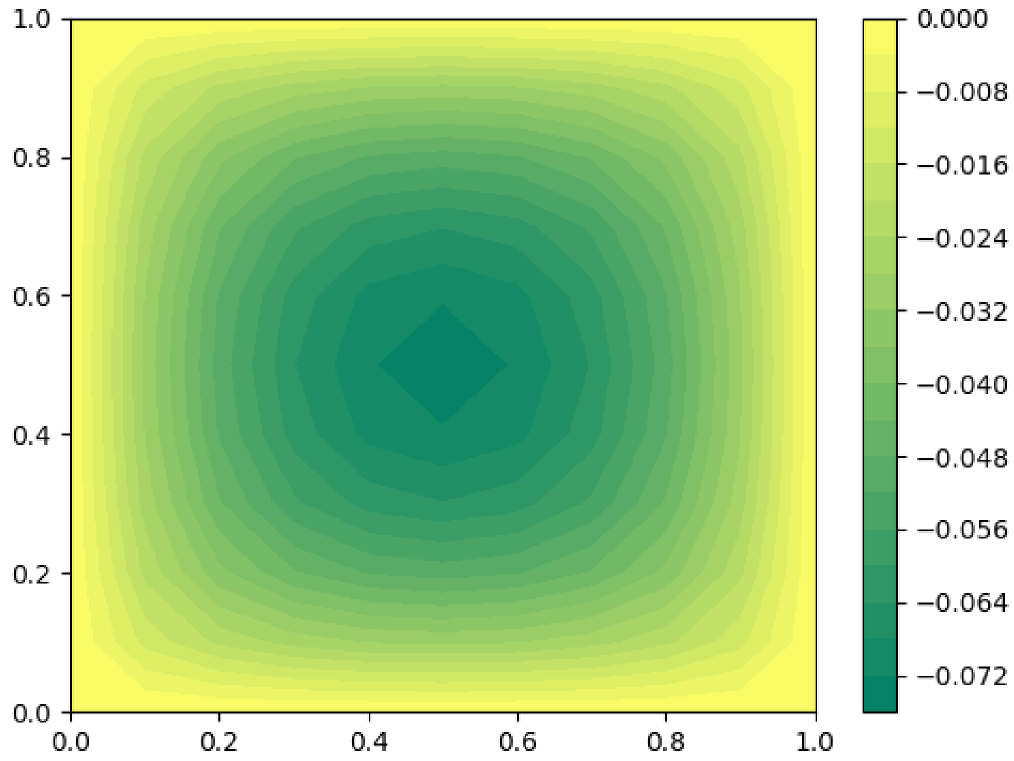
```python
def plotSolution(self, solution):
    # 2D рисунок решенного уравнения методом КЭ
    plt.contourf(self._xMesh, self._yMesh, solution, 20, cmap="summer",)
    plt.colorbar()
    plt.show()

def solution_in_2D(self, solution):
    # преобразует вектор решения в матрицу 2D,
    # где каждое решение располагается над своей точкой конечного элемента,
    # нужно для отрисовки
    matrix_solution = [[0 for _ in range(self._xNum)] for _ in range(self._yNum)]
    for i in range(self._yNum):
        for j in range(self._xNum):
            matrix_solution[i][j] = solution[i * self._xNum + j]
    return matrix_solution
```

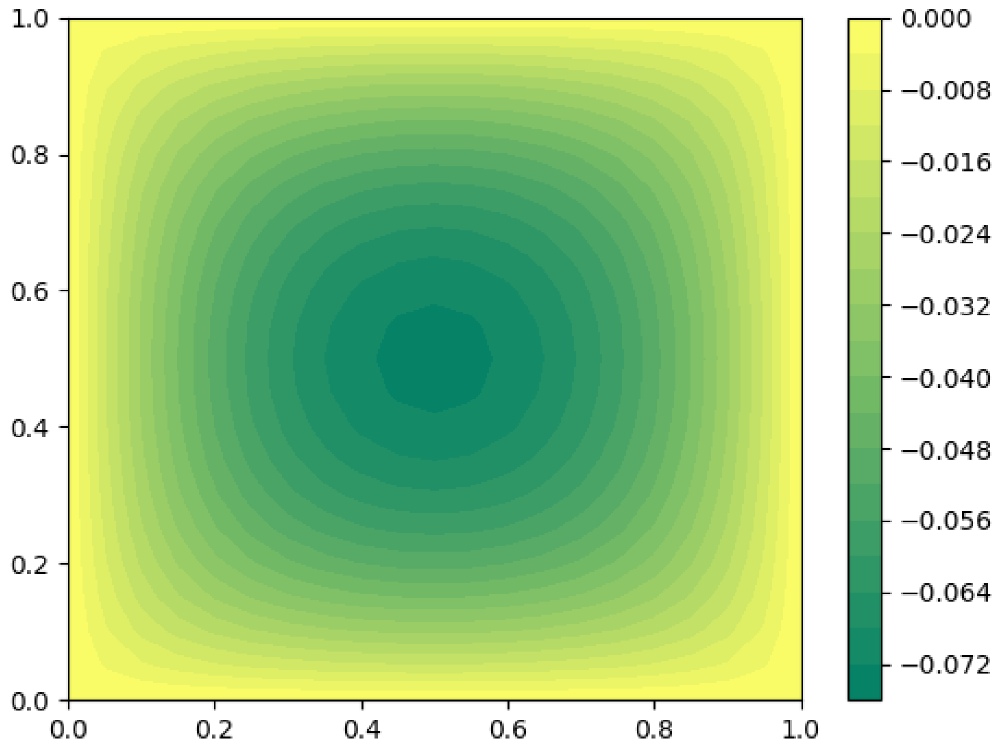
Результат программы

В результате работы программы получаем 2D картинку решения уравнения Пуассона методом конечных элементов. Программа была протестирована на разных данных. В отчете предоставлены решения области квадрата 1 на 1 с различным шагом сетки.

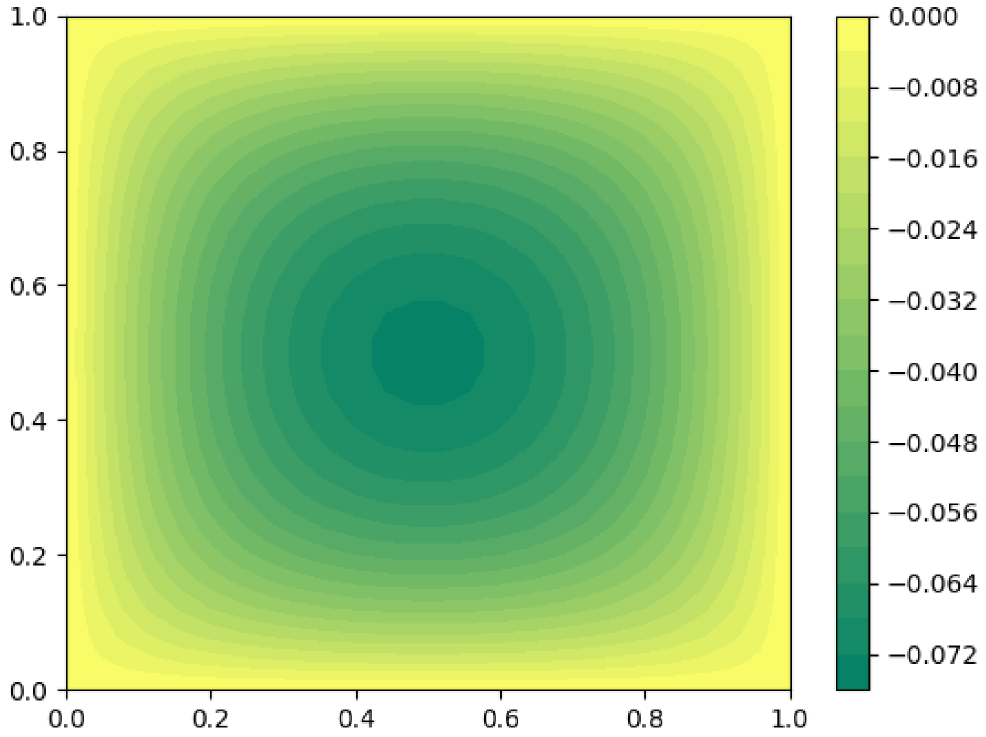
При h = 0.1:


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
0.000
−0.008
−0.016
−0.024
−0.032
−0.040
−0.048
−0.056
−0.064
−0.072

При h = 0.05:


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
0.000
−0.008
−0.016
−0.024
−0.032
−0.040
−0.048
−0.056
−0.064
−0.072

При h = 0.04:



Вывод.

В данной работе показано решение уравнения Пуассона с помощью методов конечных элементов. Был изучен метод МКЭ, а также LUP-разложение для вычисления системы линейных алгебраических уравнений. Составлена программа на языке Python, результатом которой является графическое решение уравнения Пуассона в области квадрата 1 на 1. Из результатов вывода программы видно, что чем меньше шаг сетки, тем точнее будет рисунок.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: compmath.py

```
from matplotlib import pyplot as plt

# левая часть дифференциального уравнения, уравнения Пуассона
f = -1

# базисные ("шляпные") функции в 1D: x или 1-x (если по оси OY, то y и 1-y)
hat0 = {"eval": lambda x: x, "nabla": lambda x: 1}
hat1 = {"eval": lambda x: 1 - x, "nabla": lambda x: -1}
hatFunction = [hat0, hat1]


# базисная ("шляпная") функция в 2D = произведение двух базисных функций из
1D
class Basis2D:
    def __init__(self, x0, y0):
        self.xBasis = hatFunction[x0]
        self.yBasis = hatFunction[y0]

    def evaluation(self, x, y):
        # значение базисной функции в точке x, y
        return self.xBasis["eval"](x) * self.yBasis["eval"](y)

    def nabla(self, x, y):
        # производная базисной функции в точке x, y (на самом деле оператор
набла (Гамильтона)
        # дает сумму двух значений, произведение двух сумм (a+b)*(c+d) = ac
+ ad + bc + bd,
        # поэтому возвращаем вектор, состоящий из двух элементов суммы
        return [self.xBasis["nabla"](x) * self.yBasis["eval"](y),
                self.yBasis["nabla"](y) * self.xBasis["eval"](x)]

    def scalar(self, x_self, y_self, other, x_other, y_other):
        # скалярное произведение двух базисных функций
        return (self.nabla(x_self, y_self)[0] * other.nabla(x_other,
y_other)[0] +
                self.nabla(x_self, y_self)[1] * other.nabla(x_other,
y_other)[1])


# линейная базисная функция в 2D: xy, (1-x)y, x(1-y), (1-x)(1-y)
basisFunction = {0: Basis2D(0, 0), 1: Basis2D(1, 0), 2: Basis2D(0, 1), 3:
Basis2D(1, 1)}


# Point - класс, хранит точку сетки (одна точка и ее координаты),
# а также ее индекс в сетке
# (порядок - 1, 2, ... 22 (до (n+1)^2, n - количество КЭ))
class Point:
    def __init__(self, x, y, ind):
        self.x, self.y, self.index = x, y, ind


# Cell - класс, хранит 4 точки клетки: левая нижняя - главная. Хранится в
```

```
виде списка
class Cell:
    def __init__(self, point, right_point, up_point, up_right_point):
        self.points = [point, right_point, up_point, up_right_point]


# Mesh - класс, сетка.
class Mesh:
    def __init__(self, xMin=0.0, xMax=1.0, yMin=0.0, yMax=1.0,
stepSize=0.1):
        self._xNum, self._yNum = 0, 0
        self._xMesh, self._yMesh = [], []
        self._xMin, self._xMax, self._yMin, self._yMax = xMin, xMax, yMin,
yMax
        self._h = stepSize
        self._points, self._cells = [], []

    def get_coordinates(self, coord_list, cMin, cMax, rounding):
        # находит координаты от мин до макс, округляя, делит сетку по одной
координатной оси с шагом h
        ind = cMin
        while ind < cMax:
            coord_list.append(round(ind, rounding))
            ind += self._h
        if coord_list[len(coord_list) - 1] < cMax:
            coord_list.append(cMax)

    def createMesh(self):
        # решаем проблему точности, находим количество знаков после запятой
в шаге,
        # округляем до этого количества
        h_tmp = self._h
        sights = 0
        while h_tmp % 10 > 0:
            h_tmp *= 10
            sights += 1

        # нашли координаты по оси X с заданным шагом
        x_coordinates = []
        self.get_coordinates(x_coordinates, self._xMin, self._xMax, sights)
        # количество точек по оси X
        self._xNum = len(x_coordinates)
        # нашли координаты по оси У с заданным шагом
        y_coordinates = []
        self.get_coordinates(y_coordinates, self._yMin, self._yMax, sights)
        # количество точек по оси Y
        self._yNum = len(y_coordinates)

        # задает сетку x-координат и y-координат (сетка размером x*y),
        # где значение это x-координаты для соответствующих x и y
        for ind in range(self._yNum):
            self._xMesh.append(x_coordinates)
            self._yMesh.append([y_coordinates[ind]] * self._xNum)

        # заполняем список точек
        for i in range(len(self._yMesh)):
            for j in range(self._xNum):
                self._points.append(Point(self._xMesh[i][j],
```

```
self._yMesh[i][j], i * self._xNum + j))

        # заполняем список клеток (каждая состоит из 4 точек)
        for i, point in enumerate(self._points):
            if point.x != self._xMax and point.y != self._yMax:
                self._cells.append(
                    Cell(
                        point,
                        self._points[i + 1],
                        self._points[i + self._xNum],
                        self._points[i + 1 + self._xNum]
                    )
                )

    def createSystem(self):
        # создает СЛАУ, которую в последствии решаем с помощью LUP-
разложения

        # матрица A (двумерная)
        a = [[0 for _ in range(len(self._points))] for _ in
range(len(self._points))]
        # матрица F (одномерная)
        _f = [0] * len(self._points)

        # считаем матрицу A и F, причем вместо нахождения значения каждого
элемента матрицы с помощью базиса конкретного
        # КЭ, делаем переход к master элементу
        # проходим по всем клеткам (конечным элементам) сетки
        for cell in self._cells:
            # также в каждом конечном элементе проходим по всем его 4-м
точкам
            for j, point_j in enumerate(cell.points):
                # считаем значение Fj для каждой точки КЭ, используем
правило Симпсона
                _f[point_j.index] += (((1 / 6) ** 2) * f * (self._h ** 2) *
                                      (basisFunction[j].evaluation(0, 0) +
4 * basisFunction[j].evaluation(0, 1 / 2) +
                                       basisFunction[j].evaluation(0, 1) +
4 * basisFunction[j].evaluation(1 / 2, 0) +
                                       16 * basisFunction[j].evaluation(1 /
2, 1 / 2) +
                                       4 * basisFunction[j].evaluation(1 /
2, 1) + basisFunction[j].evaluation(1, 0) +
                                       4 * basisFunction[j].evaluation(1, 1
/ 2) + basisFunction[j].evaluation(1, 1)))
                for i, point_i in enumerate(cell.points):
                    # находим Aij также с помощью правила Симпсона,
проходим еще раз по всем точкам одного КЭ,
                    # так как Aij ищется как интеграл от произведения двух
базисных функций от разных КЭ
                    a[point_i.index][point_j.index] += (
                            ((1 / 6) ** 2) * (basisFunction[i].scalar(0, 0,
basisFunction[j], 0, 0) +
                                              4 *
basisFunction[i].scalar(0, 1 / 2, basisFunction[j], 0, 1 / 2) +
                                              basisFunction[i].scalar(0, 1,
basisFunction[j], 0, 1) +
                                              4 * basisFunction[i].scalar(1
```

```
/ 2, 0, basisFunction[j], 1 / 2, 0) +
                                                     16 *
basisFunction[i].scalar(1 / 2, 1 / 2, basisFunction[j], 1 / 2,

1 / 2) +
                                                     4 * basisFunction[i].scalar(1
/ 2, 1, basisFunction[j], 1 / 2, 1) +
                                                     basisFunction[i].scalar(1, 0,
basisFunction[j], 1, 0) +
                                                     4 *
basisFunction[i].scalar(1, 1 / 2, basisFunction[j], 1, 1 / 2) +
                                                     basisFunction[i].scalar(1, 1,
basisFunction[j], 1, 1)))

        # добавили граничные условия Дирихле, точки, которые на границе =
0, а на главной диагонали = 1 в матрице А,
        # в матрице F также точки с границы равны 0
        for point in self._points:
            if point.x in [self._xMin, self._xMax] or point.y in
[self._yMin, self._yMax]:
                for j in range(len(self._points)):
                    a[point.index][j] = 0.0
                a[point.index][point.index] = 1.0
                _f[point.index] = 0.0

        return a, _f

    def plotSolution(self, solution):
        # 2D рисунок решенного уравнения методом КЭ
        plt.contourf(
            self._xMesh,
            self._yMesh,
            solution,
            20,
            cmap="summer",
        )
        plt.colorbar()
        plt.show()

    def solution_in_2D(self, solution):
        # преобразует вектор решения в матрицу 2D, где каждое решение
располагается над своей точкой конечного элемента,
        # нужно для отрисовки
        matrix_solution = [[0 for _ in range(self._xNum)] for _ in
range(self._yNum)]
        for i in range(self._yNum):
            for j in range(self._xNum):
                matrix_solution[i][j] = solution[i * self._xNum + j]
        return matrix_solution


def lup_decomposition(a):
    # LUP разложение для матрицы a
    n = len(a)

    # пустые матрицы l и u
    l = [[0.0] * n for i in range(n)]
    u = [[0.0] * n for i in range(n)]
```

```
    # матрица перестановок p, в начале на главной диагонали 1
    p = [[float(i == j) for i in range(n)] for j in range(n)]
    # pa - матрица, является произведением матрицы p на a, по сути это сама
матрица a
    pa = multiply_matrix(p, a)

    # само LUP-разложение
    for j in range(n):
        # в каждом столбце ставим максимальный по модулю элемент на главную
диагональ
        max_elem = a[j][j]
        max_ind = (j, j)
        # находим максимальный
        for k in range(j, n):
            if abs(a[k][j]) > max_elem:
                max_elem = abs(a[k][j])
                max_ind = (k, j)
        # делаем перестановки в матрице pa, p, u и l
        pa[j], pa[max_ind[0]] = pa[max_ind[0]], pa[j]
        p[j], p[max_ind[0]] = p[max_ind[0]], p[j]
        u[j], u[max_ind[0]] = u[max_ind[0]], u[j]
        l[j], l[max_ind[0]] = l[max_ind[0]], l[j]

        # Все значения на главной диагонали матрицы l равны 1
        l[j][j] = 1.0

        for i in range(j + 1):
            # вычисляем значения элементов матриц u и l на текущем шаге
            s1 = sum(u[k][j] * l[i][k] for k in range(i))
            u[i][j] = pa[i][j] - s1
        for i in range(j, n):
            s2 = sum(u[k][j] * l[i][k] for k in range(j))
            l[i][j] = (pa[i][j] - s2) / u[j][j]

    # возвращаем матрицы p - матрица перестановок, l - нижне треугольная
матрица с единицами на главной диагонали,
    # u - верхне треугольная матрица
    return p, l, u


def Ly_b(l, b):
    # решает систему ly = b, находит значения вектора y, где l - нижне
треугольная матрица с единицами на
    # главной диагонали, на то, что размерности вектора и матрицы подходят
проверка не выполняется, так как функция
    # писалась для нашей задачи, то размерности подходят
    y = [0.0 for i in range(len(b))]

    # yi = (bi - sum(lij * yj)) / lii, проходим по матрице l сверху вниз
    for i in range(len(l)):
        summa = sum(l[i][j] * y[j] for j in (range(i)))
        y[i] = (b[i] - summa) / l[i][i]
    return y


def Ux_y(u, y):
    # решает систему ux = y, находит значения вектора x, u - верхне
```

```
треугольная матрица
    x = [0.0 for i in range(len(y))]

    # xi = (yi - sum(uij * xj)) / uii, проходим по матрице и снизу вверх
    for i in range(len(u) - 1, -1, -1):
        summa = sum(u[i][j] * x[j] for j in range(len(u) - 1, i, -1))
        x[i] = (y[i] - summa) / u[i][i]
    return x


def multiply_matrix(m, n):
    # произведение двух квадратных матриц m и n одинакового размера
    size = len(m)
    result = [[0 for _ in range(len(n))] for _ in range(size)]

    for i in range(size):
        for j in range(len(n)):
            summa = 0
            for k in range(size):
                summa += m[i][k] * n[k][j]
            result[i][j] = summa

    return result


def multiply_matrix_and_vector(m, vec):
    # произведение матрицы m и вектора vec, размеры подходящие
    size = len(m)
    result = [0 for _ in range(len(vec))]

    for i in range(size):
        summa = 0
        for j in range(len(vec)):
            summa += m[i][j] * vec[j]
        result[i] = summa

    return result


if __name__ == '__main__':
    # создаем сетку и систему ЛАУ
    mesh = Mesh()
    mesh.createMesh()
    a, _f = mesh.createSystem()

    # делаем lup-разложение
    p, l, u = lup_decomposition(a)

    # теперь pa = lu
    # решим систему ax = y, домножим слева на p обе части -> pax = py, y =
_f, b = p*_f
    # lux = b, ux = y, найдем решение для ly = b, подставим в ux = y,
найдем x - искомый вектор
    b = multiply_matrix_and_vector(p, _f)
    y = Ly_b(l, b)
    solution = Ux_y(u, y)

    # найденное решение сделаем удобным для отрисовки в 2D и нарисуем,
```

итоговый рисунок - результат работы программы

```
sol = mesh.solution_in_2D(solution)
mesh.plotSolution(sol)
```