1 Постановка задачи

Дано:

1. Полигональная область Ω с треугольной сеткой
2. Заданная функция (например, u(x, y) = sin(x + y))

Задача:

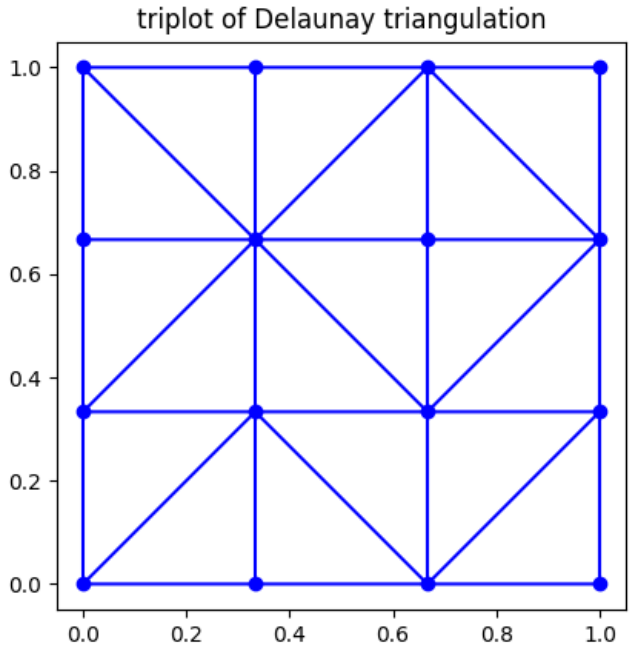
1. Построить линейный интерполянт I_h
2. Написать функцию, вычисляющую норму $L^2(\Omega)$ разности точного решения и интерполянта
3. Написать функцию, вычисляющую норму $L^2(\Omega)$ разности градиентов точного решения и интерполянта
4. Найти зависимость точности вычисления норм от шага h
5. Написать алгоритм минимизации функционала вида:

$$J = ||u_h - u||^2_{L^2} \tag{1}$$

2 Решение

2.1 Триангуляция области

Пока что для простоты рассматривается прямоугольная область, триангуляция выполняется следующим образом:



2.2 Построение интерполянта

Интерполянт задаётся в узлах сетки точными занчениями интерполируемой функции. На рёбрах конечного элемента он задаётся уравнением прямой в пространстве R_3, проходящей через точки ребра, на грани - уравнением плоскости, проходящей через вершины конечного элемента(в данной работе это впринципе не нужно, достаточно рассматривать рёбра).

2.3 Вычисление нормы разности точного решения и интерполянта

Норму считаем по следующей формуле (Ω_i - конечный элемент с индексом i, N - число конечных элементов)

$$||f||_{L_2(\Omega)} = \int\limits_{\Omega} |f|^2 d\Omega = \sum_{i=1}^{N} \int\limits_{\Omega_i} |f|^2 d\Omega$$

А интеграл на конечном элементе считаем как:

$$\int\limits_{\Omega_i} |f|^2 d\Omega = \frac{\Delta_i}{3}(\psi_{12} + \psi_{13} + \psi_{23})$$

Где ψ_{jk} - значение функции f в центральной точке ребра jk конечного элемента i

При этом, в данном случае функция $f = u - u_I$. Пример того, как в точности вычисляется норма, есть в приведённом ниже коде программы

2.4 Вычисление нормы разности градиентов точного решения и интерполянта

Тут действуем в точности также, как и в пункте выше. В данном случае, функция $f = grad(u) - grad(u_I)$. $grad(u)$ задаём ‘вручную‘ по определению градиента. $grad(u_I)$ считаем следующим образом:

$$\frac{x - x_b}{x_a - x_b} = \frac{y - y_b}{y_a - y_b} = \frac{z - z_b}{z_a - z_b}$$

Из этой формулы для ребра, заданного точками a и b, спроецировав равенства на плоскости ‘Ozy‘, ’Ozx’ получим константное выражение вида:

$$grad(u_I) = \frac{z_a - z_b}{y_a - y_b} + \frac{z_a - z_b}{x_a - x_b}$$

Если знаменатель в каком-то слагаемом обращается в нуль - заменяем всё слагаемое на нуль.

(Мне очень сильно кажется, что я всё делаю совершенно не так, но на сегодня ничего лучше я не придумал)

2.5 Зависимость ошибки от числа шагов

```
function:
n =  2 , error =  0.0
n =  4 , error =  5.691237405802064e-10
n =  8 , error =  2.7314080965089306e-10
n =  16 , error =  5.920631616878284e-11
n =  32 , error =  1.4515131676229756e-11
n =  64 , error =  3.5421490352017987e-12
n =  128 , error =  8.357825257952179e-13
n =  256 , error =  2.0823515947018266e-13
gradient:
n =  2 , error =  0.0791015625
n =  4 , error =  2.3494280425288075e-06
n =  8 , error =  1.5144356455031636e-08
n =  16 , error =  6.59748600754445e-10
n =  32 , error =  7.958758452422896e-11
n =  64 , error =  1.4587119129461523e-11
n =  128 , error =  3.2438375060047228e-12
n =  256 , error =  7.893913730348227e-13
```

3 Код

Прикладываю скриншоты программы. Весь код в более удобном виде можно посмотреть тут `https://github.com/LanskovNV/fem-labs/tree/master/lab_3`

```python
import numpy as np
import matplotlib.pyplot as plt

from interpolation import Interpolation


if __name__ == "__main__":
    num_of_nodes = 2
    x0, x1 = 0, 1
    y0, y1 = 0, 1
    shape = np.array([[x0, y0], [x0, y1], [x1, y1], [x0, y1]])

    inter = 0
    print("function:")
    for i in range(8):
        inter = Interpolation(num_of_nodes, shape)
        print("n = ", num_of_nodes, ", error = ", inter.error(is_grad=False))
        num_of_nodes *= 2

    num_of_nodes = 2
    print("gradient:")
    for i in range(8):
        inter = Interpolation(num_of_nodes, shape)
        print("n = ", num_of_nodes, ", error = ", inter.error(is_grad=True))
        num_of_nodes *= 2
```

```python
from abc import ABC, abstractmethod
import matplotlib.pyplot as plt
import matplotlib.tri as tri
import numpy as np


class Region(ABC):
    triangles = []
    x_coords = []
    y_coords = []
    shape = []

    def __init__(self, shape):
        self.shape = shape

    def triangulate(self, n):
        self._flatten(n)
        x = self.x_coords
        y = self.y_coords
        self.triangles = tri.Triangulation(x, y)

    # generate flatten coords of the net points
    # with specified discretization steps
    @abstractmethod
    def _flatten(self, n):
        pass

    def draw(self):
        fig = plt.figure()
        ax = fig.add_subplot(1, 1, 1)
        shape = plt.Polygon(self.shape, color='g', alpha=0.3)
        ax.add_patch(shape)
        plt.plot()
        plt.show()

    def draw_net(self):
        plt.figure()
        plt.gca().set_aspect('equal')
        plt.triplot(self.triangles, 'bo-')
        plt.title('triplot of Delaunay triangulation')
        plt.show()
```

```python
import numpy as np

from region import Region


class Rectangle(Region):
    def __init__(self, shape=np.array([[0, 0], [0, 1], [1, 1], [1, 0]])):
        super().__init__(shape)
        self.x_int = [shape[:, 0].min(), shape[:, 0].max()]
        self.y_int = [shape[:, 1].min(), shape[:, 1].max()]

    def _flatten(self, n):
        self.x_coords = np.linspace(self.x_int[0], self.x_int[1], n)
        self.y_coords = np.linspace(self.y_int[0], self.y_int[1], n)

        self.x_coords = np.tile(self.x_coords, n)
        self.y_coords = np.repeat(self.y_coords, n)
        ...
```

```python
import numpy as np
import matplotlib.pyplot as plt

from rectangle import Rectangle


class Interpolation:
    def __init__(self, n, shape=np.array([[0, 0], [0, 1], [1, 1], [1, 0]])):
        # omega setup
        self.shape = shape
        self.omega = Rectangle(self.shape)
        self.omega.triangulate(n)

        # function set up manually
        self.function = lambda x, y: x*(1 - x)*y*(1 - y)
        self.grad = lambda x, y: ((1 - 2*x)*(y - y**2) + (1 - 2*y)*(x - x**2))

    def draw(self):
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')

        x = self.omega.x_coords
        y = self.omega.y_coords
        func = np.vectorize(self.function)
        z = func(self.omega.x_coords, self.omega.y_coords)

        ax.plot_trisurf(x, y, z)

        plt.show()

    def error(self, is_grad=False):
        def get_coords(tr):
            trc = np.zeros((3, 2))
            trc[0][0], trc[0][1] = self.omega.x_coords[tr[0]], self.omega.y_coords[tr[0]]
            trc[1][0], trc[1][1] = self.omega.x_coords[tr[1]], self.omega.y_coords[tr[1]]
            trc[2][0], trc[2][1] = self.omega.x_coords[tr[2]], self.omega.y_coords[tr[2]]
            return trc

        def area(trc):
            s = (trc[1, 0] - trc[0, 0])*(trc[2, 1] - trc[0, 1]) - \
                (trc[1, 1] - trc[0, 1])*(trc[2, 0] - trc[0, 0])
```

```
def area(trc):
    s = (trc[1, 0] - trc[0, 0])*(trc[2, 1] - trc[0, 1]) - \
        (trc[1, 1] - trc[0, 1])*(trc[2, 0] - trc[0, 0])
    s = abs(s) / 2
    return s

def int_middle(trc, a, b,  x, y):
    ff = self.function
    x0, x1 = trc[a][0], trc[b][0]
    y0, y1 = trc[a][1], trc[b][1]
    z0, z1 = ff(x0, y0), ff(x1, y1)
    dx, dy = x1 - x0, y1 - y0

    if is_grad:
        res = 0
        if x1 - x0 != 0:
            res += (z1-z0) / (x1-x0)
        if y1 - y0 != 0:
            res += (z1-z0) / (y1-y0)
        return res
    else:
        if dx == 0:
            return z0 + (z1 - z0) * (y - y0) / dy
        else:
            return z0 + (z1 - z0) * (x - x0) / dx

def middle(trc, a, b):
    if is_grad:
        f = self.grad
    else:
        f = self.function
    x = (trc[a][0] - trc[b][0]) / 2
    y = (trc[a][1] - trc[b][1]) / 2

    return abs(int_middle(trc, a, b, x, y) - f(x, y))

def int_triang(tr):
    trc = get_coords(tr)
    f01 = middle(trc, 0, 1)
```

```
        y = (trc[a][1] - trc[b][1]) / 2

        return abs(int_middle(trc, a, b, x, y) - f(x, y))

    def int_triang(tr):
        trc = get_coords(tr)
        f01 = middle(trc, 0, 1)
        f12 = middle(trc, 1, 2)
        f02 = middle(trc, 0, 2)
        return area(trc) / 3 * (f01 * f12 * f02)

    ans = 0
    for _ in self.omega.triangles.triangles:
        ans += (int_triang(_))**2

    return ans
```