1 Постановка задачи

Дано:

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

Задача:

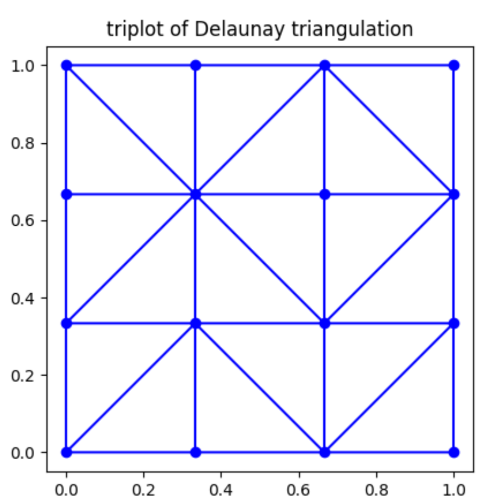
1. Построить линейный интерполянт I_h
2. Написать функцию, вычисляющую норму $L^2(\Omega)$ разности точного решения и интерполянта
3. Написать функцию, вычисляющую норму $L^2(\Omega)$ разности градиентов точного решения и интерполянта
4. Найти зависимость точности вычисления норм от шага h
5. Написать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

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

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)

$$||f||_{L_2(\Omega)} = \left(\int\limits_{\Omega} |f|^2 d\Omega\right)^{0.5} = \left(\sum_{i=1}^{N}\int\limits_{\Omega_i} |f|^2 d\Omega\right)^{0.5}$$

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 $|f|^2$ в центральной точке ребра jk конечного элемента i,
а Δ_i - площадь конечного элемента

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. $grad(u_I)$ находим как сумму значений $a_1 + a_2$, где a_1, a_2 находим из решения системы:

$$\begin{cases} a_1x_1 + a_2y_1 + a_3 = u(x_1, y_1) \\ a_1x_2 + a_2y_2 + a_3 = u(x_2, y_2) \\ a_1x_3 + a_2y_3 + a_3 = u(x_3, y_3) \end{cases}$$

Где $z = a_1x + a_2y + a_3$ - уравнение плоскости, в которой лежит конечный элемент,
а $(x_i, y_i, u(x_i, y_i))$ - координаты вершин конечного элемента (известны из постановки задачи и триангуляции области)

2.5 Зависимость ошибки от числа узлов

```
function:
n =  2 , error =  0.03608439182435161
n =  4 , error =  0.008242500184651272
n =  8 , error =  0.0015906731415815446
n =  16 , error =  0.00034528069349915264
n =  32 , error =  8.166999259286822e-05
n =  64 , error =  1.9929116899051134e-05
n =  128 , error =  4.90693971374586e-06
n =  256 , error =  1.21581684316745e-06
gradient:
n =  2 , error =  0.2041241452319315
n =  4 , error =  0.077795738993455
n =  8 , error =  0.035583665072484706
n =  16 , error =  0.01630394483187211
n =  32 , error =  0.007806450997176318
n =  64 , error =  0.0038594499909911373
n =  128 , error =  0.0019187120164667686
n =  256 , error =  0.0009538529259929061
```

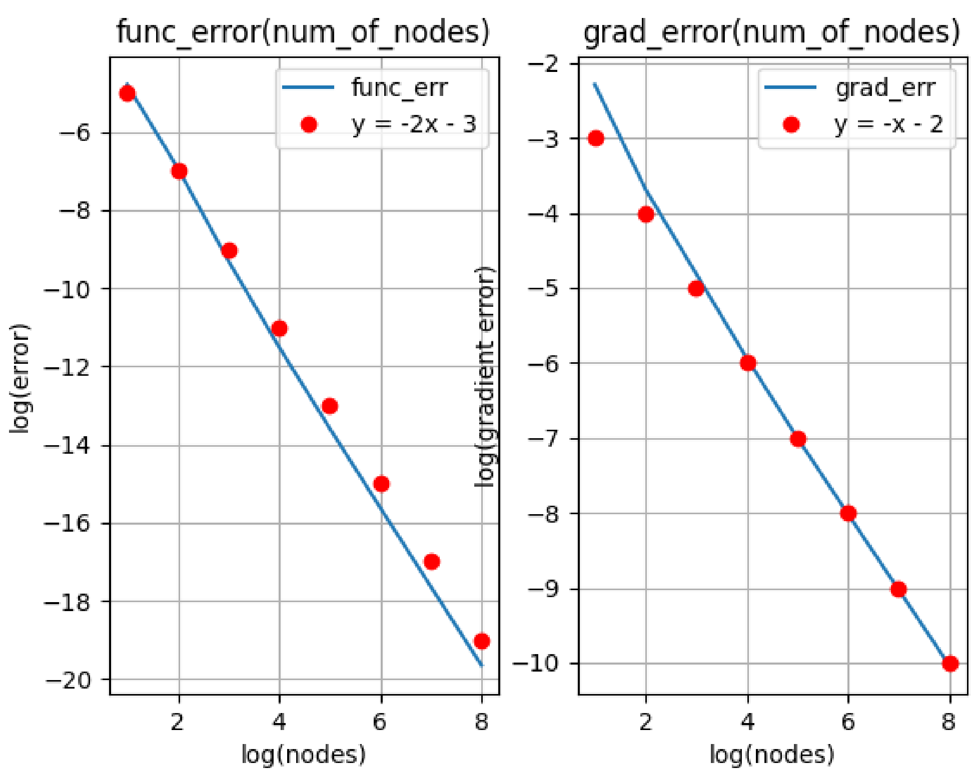


Рис. 1: Графики зависимости ошибки от числа узлов (на границе области) для функции $f(x, y) = x(1 - x)y(1 - y)$ в логарифмических осях.

2.6 Поиск значений сеточной функции посредством минимизации функционала ошибки

Запишем функционал ошибок:

$$J = \int_{\Omega} |u - \sum_{i=1}^{N} u_i \cdot \phi_i|^2 \tag{2}$$

Наше приближённое решение представлено в виде: $u_h = \sum_{i=1}^{N} u_i \cdot \phi_i$, где ϕ_i - базисные функции в узлах сетки, а N, соответственно, число узлов.

Далее составляем матричное уравнение, пользуясь условием минимума функционала:

$$\frac{\partial J}{\partial u_i} = 0 \tag{3}$$

Проделав очевидные преобразования над J, получаем матричное уравнение относительно u_i:

$$\sum_{j=i}^{N} u_i \int_{\Omega} \phi_i \cdot \phi_j = -\int_{\Omega} u \cdot \phi_i \tag{4}$$

Где базисные функции могут быть вычислены как сумма функций формы на тех конечных элементах, вершиной которых является узел с индексом i в глобальной нумерации узлов:

$$\phi_i = \sum_{k=1}^{M} \phi_i^k \tag{5}$$

а ϕ_i^k, в свою очередь, находятся из решения систем вида:

$$\begin{cases} a_1 x_1 + a_2 y_1 + a_3 = 1 \\ a_1 x_2 + a_2 y_2 + a_3 = 0 \\ a_1 x_3 + a_2 y_3 + a_3 = 0 \end{cases} \tag{6}$$

для каждого конечного элемента.

В формуле (5), M - число конечных элементов, в которые входит узел с индексом i в глобальной нумерации, а ϕ_i^k - функция формы на k-ом элементе - плоскость, проходящая через три точки:

1. $i : (x_1, y_1, 1)$
2. $(x_2, y_2, 0)$
3. $(x_3, y_3, 0)$

3 Код

Все материалы, использованные при подготовке работы, доступны тут:
`https://github.com/LanskovNV/fem-labs/tree/master/lab_3`
Также все исходные коды доступны в секции ‘Приложение‘.

4 Приложение

```
import numpy as np
import matplotlib.pyplot as plt
from math import log2

from src.interpolation import Interpolation

xa, xb = 0, 1
ya, yb = 0, 1
shape = np.array([[xa, ya], [xa, yb], [xb, yb], [xb, ya]])


def print_errors():
    print("function:")
    for i in range(1, 9):
        inter = Interpolation(2**i, shape)
        print("n = ", 2**i, ", error = ", inter.error(is_grad=False))

    print("gradient:")
    for i in range(1, 9):
        inter = Interpolation(2**i, shape)
        print("n = ", 2**i, ", error = ", inter.error(is_grad=True))


def draw_errors():
    n = [2**x for x in range(1, 9)]
    err = [Interpolation(i, shape).error() for i in n]
    err_grad = [Interpolation(i, shape).error(True) for i in n]

    f = np.vectorize(log2)
    def f1(x): return -2*x - 3
    def f2(x): return - x - 2
    f1 = np.vectorize(f1)
    f2 = np.vectorize(f2)
    z = range(1, 9)

    fig, (ax1, ax2) = plt.subplots(1, 2)
    x, y = n, err
    ax1.plot(f(x), f(y), label='func_err')
    ax1.plot(z, f1(z), 'ro', label='y = -2x - 3')
    x, y = n, err_grad
    ax2.plot(f(x), f(y), label='grad_err')
    ax2.plot(z, f2(z), 'ro', label='y = -x - 2')

    ax1.set_title("func_error(num_of_nodes)")
    ax1.set_xlabel("log(nodes)")
    ax1.set_ylabel("log(error)")
    ax1.grid()
    ax1.legend()

    ax2.set_title("grad_error(num_of_nodes)")
    ax2.set_xlabel("log(nodes)")
    ax2.set_ylabel("log(gradient error)")
    ax2.grid()
    ax2.legend()

    plt.savefig("pic/Figure_3.png")
    # plt.show()



if __name__ == "__main__":
    # print_errors()
    draw_errors()
```

```
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()
```

```
import numpy as np

from src.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)
        # n += 1
        #
        # def half_step(interval):
        #     return (interval[1] - interval[0]) / (n - 1) / 2
        #
        # def get_x(interval):
        #     first_layer = np.linspace(interval[0], interval[1], n)
        #     hs = half_step(interval)
        #     second_layer = np.linspace(interval[0] + hs, interval[1] + hs, n - 1, endpo
int=False)
        #
        #     second_layer = np.append(second_layer, first_layer[0])
        #     second_layer = np.append(second_layer, first_layer[-1])
        #
        #     segment = np.append(first_layer, second_layer)
        #     net_x = np.tile(segment, n - 1)
        #     return np.append(net_x, first_layer)
        #
        # def get_y(interval):
        #     net_y = np.linspace(interval[0], interval[1], 2*n - 1)
        #     first = np.repeat(net_y[::2], n)
        #     second = np.repeat(net_y[1::2], n + 1)
        #     return np.sort(np.append(first, second))
        #
        # def get_coords(x_int, y_int):
        #     xc = get_x(x_int)
        #     yc = get_y(y_int)
        #     return xc, yc
        #
        # # longest edge should have more points
        # if (self.x_int[1] - self.x_int[0]) > (self.y_int[1] - self.y_int[0]):
        #     self.y_coords, self.x_coords = get_coords(self.y_int, self.x_int)
        # else:
        #     self.x_coords, self.y_coords = get_coords(self.x_int, self.y_int)
```

```
import numpy as np
import matplotlib.pyplot as plt
from math import sqrt

from src.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])
            s = abs(s) / 2
            return s

        def middle_val(trc, a, b, x, y):
            xa, xb = trc[a][0], trc[b][0]
            ya, yb = trc[a][1], trc[b][1]
            za, zb = self.function(xa, ya), self.function(xb, yb)

            if xb - xa == 0:
                return za + (zb - za) * (y - ya) / (yb - ya)
            else:
                return za + (zb - za) * (x - xa) / (xb - xa)

        def middle_val_grad(trc):
            m = np.column_stack((trc, np.ones(3)))
            r = np.array([self.function(trc[0][0], trc[0][1]),
                          self.function(trc[1][0], trc[1][1]),
                          self.function(trc[2][0], trc[2][1])])
            a = np.linalg.solve(m, r)
            return a[0] + a[1]

        def middle(trc, a, b):
            x = (trc[b][0] + trc[a][0]) / 2
            y = (trc[b][1] + trc[a][1]) / 2
            if is_grad:
                mid_val = middle_val_grad(trc)
                return (abs(mid_val - self.grad(x, y))) ** 2
            else:
```

```
            mid_val = middle_val(trc, a, b, x, y)
            return (abs(mid_val - self.function(x, y))) ** 2

    def int_triang(tr):
        trc = get_coords(tr)
        f01 = middle(trc, 0, 1)
        f12 = middle(trc, 1, 2)
        f20 = middle(trc, 2, 0)
        return area(trc) / 3 * (f01 + f12 + f20)

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

    return sqrt(ans)
```