```
import numpy as np
import matplotlib.pyplot as plt
from math import log2
import csv
from src.interpolation import Interpolation
from src.minimization import Minimization
from src.source import draw
import matplotlib

name = "./out2.csv"
xa, xb = 0, 1
ya, yb = 0, 1
shape = np.array([[xa, ya], [xa, yb], [xb, yb], [xb, ya]])

matplotlib.use('TkAgg')


def print_errors():
    dicts = []
    for i in range(2, 5):  # 1, 9
        print(i)
        inter = Minimization(2**i, shape)  # Interpolation(2**i, shape)
        dicts.append({"n": 2**i,
                      "function error": inter.error(is_grad=False),
                      "gradient error": inter.error(is_grad=True)})

    with open(name, mode="w") as csv_file:
        fieldnames = ["n", "function error", "gradient error"]
        writer = csv.DictWriter(csv_file, fieldnames=fieldnames)
        writer.writeheader()
        writer.writerows(dicts)


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()


def test(n):
    inter = Minimization(n, shape)
    print("Minimization error: ", inter.error(False))


def test2(n):
    inter = Interpolation(n, shape)
    print("Interpolation error: ", inter.error(False))


if __name__ == "__main__":
    n = 10
    test(n)
    test2(n)
    # 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)
```

```
from abc import ABC, abstractmethod
import numpy as np
from math import sqrt
from src.rectangle import Rectangle
from src.source import area, middle_val, middle_val_grad


class BaseFem(ABC):
    def __init__(self, n, shape):
        # 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)
)

        self.is_grad = False
        self.node_values = []

    def error(self, is_grad=False):
        ans = 0

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

        return sqrt(ans)

    def set_error_type(self, is_grad):
        self.is_grad = is_grad

    def get_coords(self, tr):
        omega = self.omega
        trc = np.zeros((3, 3))
        trc[0][0], trc[0][1], trc[0][2] = omega.x_coords[tr[0]], omega.y_coords[tr[0]], s
elf.node_values[tr[0]]
        trc[1][0], trc[1][1], trc[1][2] = omega.x_coords[tr[1]], omega.y_coords[tr[1]], s
elf.node_values[tr[1]]
        trc[2][0], trc[2][1], trc[2][2] = omega.x_coords[tr[2]], omega.y_coords[tr[2]], s
elf.node_values[tr[2]]
        return trc

    def middle(self, trc, a, b):
        x = (trc[b][0] + trc[a][0]) / 2
        y = (trc[b][1] + trc[a][1]) / 2
        if self.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(self, tr):
        trc = self.get_coords(tr)
        f01 = self.middle(trc, 0, 1)
        f12 = self.middle(trc, 1, 2)
        f20 = self.middle(trc, 2, 0)
        return area(trc) / 3 * (f01 + f12 + f20)

    @abstractmethod
    def set_node_values(self):
        pass
```

```
from src.base_fem import BaseFem


class Interpolation(BaseFem):
    def __init__(self, n, shape):
        super().__init__(n, shape)
        self.set_node_values()

    def set_node_values(self):
        f = self.function
        x = self.omega.x_coords
        y = self.omega.y_coords
        for i in range(len(x)):
            self.node_values.append(f(x[i], y[i]))
```

```
import numpy as np
from src.base_fem import BaseFem
from src.source import area


class Minimization(BaseFem):
    def __init__(self, n, shape):
        super().__init__(n, shape)
        self.set_node_values()

    def get_elems(self, i):
        ans = []
        for _ in self.omega.triangles.triangles:
            if i in _:
                ans.append(_)
        return ans

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

    def get_shape(self, tr, i):
        trc = self.get_coords_xy(tr)
        m = np.column_stack((trc, np.ones(3)))
        r = np.zeros(3)

        for j in range(3):
            if tr[j] == i:
                r[j] = 1

        a = np.linalg.solve(m, r)
        return lambda x, y: a[0] * x + a[1] * y + a[2]

    def integrate(self, tr, func, s):
        trc = self.get_coords_xy(tr)
        f12 = func((trc[0, 0] + trc[1, 0]) / 2, (trc[0, 1] + trc[1, 1]) / 2)
        f13 = func((trc[0, 0] + trc[2, 0]) / 2, (trc[0, 1] + trc[2, 1]) / 2)
        f23 = func((trc[1, 0] + trc[2, 0]) / 2, (trc[1, 1] + trc[2, 1]) / 2)
        return s / 3 * (f12 + f13 + f23)

    def set_node_values(self):
        rank = len(self.omega.x_coords)
        A = np.zeros((rank, rank))
        R = np.zeros(rank)

        for i in range(rank):
            elements = self.get_elems(i)
            for e in elements:
                si = self.get_shape(e, i)
                trc = self.get_coords_xy(e)
                a = area(trc)
                for ind in e:
                    sj = self.get_shape(e, ind)
                    if i == ind:
                        A[i, ind] += self.integrate(e, lambda x, y: si(x, y) * sj(x, y),
a)
                    else:
                        A[i, ind] += self.integrate(e, lambda x, y: si(x, y) * sj(x, y),
a) / 2

        self.node_values = np.linalg.solve(A, R)
```

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


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

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

    ax.plot_trisurf(x, y, z)

    plt.show()


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 = trc[a][2], trc[b][2]

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


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
```