МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КУБГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра вычислительных технологий**

**Отчет**

**по лабораторной работе №2 по курсу**

**«МЕТОДЫ ПОИСКОВОЙ ОПТИМИЗАЦИИ»**

Работу выполнил

Студенты 39 группы

Козлов Э.Д. Кличенко Д.А

Преподаватель:

Нигодин Е.А.

Краснодар

2025

**Цель работы:** изучить методы оптимизации с использованием квадратичной целевой функции и линейными ограничениями, и применить один из них.

**Ход работы:** для реализации был выбран симплекс-метод с условием Куна-Таккера

**Шаги алгоритма**

*Шаг 1*. Преобразовать ограничения исходной задачи к виду g(x) <=0

*Шаг 2.* Составить функцию Лагранжа для получения линейного уравнения

*Шаг 3.* Находим частные производные и составляем условия Куна-Таккера для получения линейных ограничений:

а) Стационарность (равенство градиента нулю)

б) Допустимость

в) Дополняющая нежёсткость

*Шаг 4.* Преобразуем задачу в задачу линейного программирования

*Шаг 5.* Решаем задачу линейного программирования симплекс-методом с учетом условий дополняющей нежесткости

а) Создаем симплекс-таблицу

б) Определяем разрешающий столбец

в) Определяем разрешающую строку

г) Пересчёт таблицы

д) Проверка на оптимальность

**Программа**

Для создания программы используется язык программирования Python. Для графической визуализации используется графический фреймворк tkinter & matplotlib. В приложение первой лабораторной был добавлен симплекс-метод.

В созданной программе одно главное активное окно.

Изображение выглядит как снимок экрана, программное обеспечение, текст, диаграмма

Автоматически созданное описание

Рисунок 1 – Главное окно программы.

В данном окне можно выбрать необходимую лабораторную работу (Задание 1,2….,7), в данном случае «Задание 2», задать задержку между этими итерациями в секундах (используется для наглядного представления работы алгоритма).

Также, в интерфейсе программы есть панель управления, в которой можно выбрать необходимый алгоритм с функцией и задать параметры:

1. Скорость анимации

На рисунке 2 показана «Панель управления»

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Рисунок 2 – Панель управления.

Для выбора задания используется выпадающий список с номером лабораторной работы. Для запуска алгоритма необходимо нажать кнопку «Start», при этом на отображаемой функции можно наглядно увидеть функционирование алгоритма – искомые точки, соединенные линией, как показано на рисунке 3

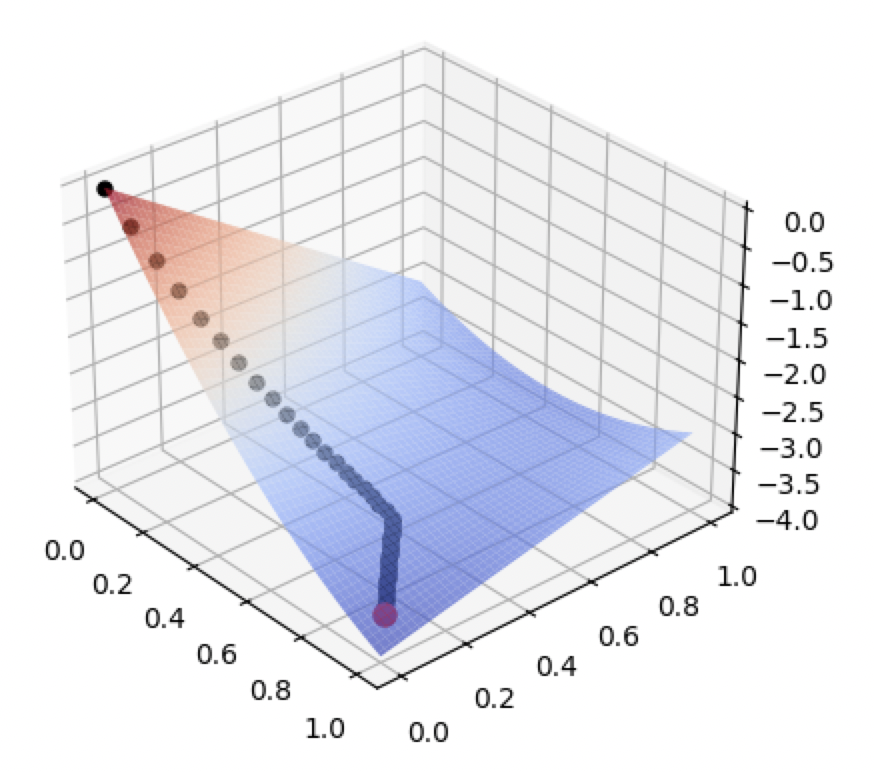


Рисунок 3 – Результат работы программы

**Вывод:** в ходе работы были изучены различные методы с использованием квадратичной целевой функции и линейными ограничениями, реализован симплекс-метод с условием Куна-Таккера.

**Листинг программы:**

import numpy as np  
from numba import jit  
  
@jit(nopython=True)  
def quadratic\_function(x, y):  
 return 2 \* x\*\*2 + 4 \* x \* y - 6 \* x - 3 \* y  
  
@jit(nopython=True)  
def quadratic\_gradient(x, y):  
 df\_dx = 4 \* x + 4 \* y - 6  
 df\_dy = 4 \* x - 3  
 return np.array([df\_dx, df\_dy])  
  
def is\_within\_constraints(x, y):  
 return x >= 0 and y >= 0 and (x + y) <= 1 and (2 \* x + 3 \* y) <= 4  
  
def project\_to\_constraints(x, y):  
 x = max(0, min(x, 1))  
 y = max(0, min(y, 1))  
 if (x + y) > 1:  
 x, y = x / (x + y), y / (x + y)  
 if (2 \* x + 3 \* y) > 4:  
 scale = 4 / (2 \* x + 3 \* y)  
 x, y = x \* scale, y \* scale  
 return x, y

import numpy as np  
from model.base\_optimizer import BaseOptimizer  
from model.quadratic\_task import quadratic\_function, quadratic\_gradient, project\_to\_constraints  
  
class QuadraticOptimizer(BaseOptimizer):  
 def optimize(self, x0, y0):  
 tolerance = 0.001  
 points = np.zeros((self.iterations + 1, 2))  
 current\_point = np.array([x0, y0])  
 points[0] = current\_point  
 actual\_iter = 0  
  
 for i in range(self.iterations):  
 grad = quadratic\_gradient(current\_point[0], current\_point[1])  
 if np.linalg.norm(grad) < tolerance:  
 break  
 new\_point = current\_point - self.step\_size \* grad  
 new\_point[0], new\_point[1] = project\_to\_constraints(new\_point[0], new\_point[1])  
 points[i + 1] = new\_point  
 if np.linalg.norm(new\_point - current\_point) < tolerance:  
 break  
 current\_point = new\_point  
 actual\_iter = i + 1  
  
 return points[:actual\_iter + 1]

import numpy as np  
from model.base\_optimizer import BaseOptimizer  
from model.quadratic\_task import quadratic\_function  
  
class NelderMeadOptimizer(BaseOptimizer):  
 def \_\_init\_\_(self, func, step\_size, iterations, tolerance=1e-6):  
 super().\_\_init\_\_(func, None, step\_size, iterations)  
 self.tolerance = tolerance  
  
 def optimize(self, x0, y0):  
 *# Начальный симплекс (треугольник в 2D)* simplex = np.array([  
 [x0, y0],  
 [x0 + self.step\_size, y0],  
 [x0, y0 + self.step\_size]  
 ])  
 values = [self.func(x, y) for x, y in simplex]  
 history = []  
  
 for \_ in range(self.iterations):  
 indices = np.argsort(values)  
 simplex = simplex[indices]  
 values = [values[i] for i in indices]  
 best = simplex[0]  
 worst = simplex[-1]  
 second\_worst = simplex[-2]  
 history.append((best[0], best[1], self.func(best[0], best[1])))  
  
 centroid = np.mean(simplex[:-1], axis=0)  
 reflected = centroid + (centroid - worst)  
 f\_reflected = self.func(reflected[0], reflected[1])  
  
 if f\_reflected < values[0]:  
 expanded = centroid + 2 \* (centroid - worst)  
 f\_expanded = self.func(expanded[0], expanded[1])  
 if f\_expanded < f\_reflected:  
 simplex[-1] = expanded  
 values[-1] = f\_expanded  
 else:  
 simplex[-1] = reflected  
 values[-1] = f\_reflected  
 elif f\_reflected < values[-2]:  
 simplex[-1] = reflected  
 values[-1] = f\_reflected  
 else:  
 contracted = centroid + 0.5 \* (worst - centroid)  
 f\_contracted = self.func(contracted[0], contracted[1])  
 if f\_contracted < values[-1]:  
 simplex[-1] = contracted  
 values[-1] = f\_contracted  
 else:  
 simplex[1:] = best + 0.5 \* (simplex[1:] - best)  
 values = [self.func(x, y) for x, y in simplex]  
  
 if np.std(values) < self.tolerance:  
 break  
  
 return np.array(history)