|  | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |
| --- | --- |

ФАКУЛЬТЕТ Информатики и систем управления

КАФЕДРА Теоретической информатики и компьютерных технологий

**Домашнее задание № 3**

**Методы многомерного поиска**

**ПО КУРСУ:**

***«Теория искусственных нейронных сетей»***

Студент *Караник А. А.*

Преподаватель *Каганов Ю. Т.*

*Москва, 2024 г.*

**ОГЛАВЛЕНИЕ**

[Цель работы 3](#_Toc182539028)

[Постановка задачи 3](#_Toc182539029)

[Вариант 11 3](#_Toc182539030)

[Практическая реализация 3](#_Toc182539031)

[Результаты 10](#_Toc182539032)

[Вывод 10](#_Toc182539033)

# Цель работы

1. Изучение алгоритмов многомерного поиска 1-го и 2-го порядка.

2. Разработка программ реализации алгоритмов многомерного поиска 1-го и 2-го порядка.

3. Вычисление экстремумов функции.

# Постановка задачи

Требуется найти минимум тестовой функции Розенброка.

1. Методами сопряженных градиентов (методом Флетчера-Ривза и методом Полака-Рибьера).

2. Квазиньютоновским методом (Девидона-Флетчера-Пауэлла).

3. Методом Левенберга-Марквардта.

# Вариант 11

# Практическая реализация

Исходный текст программы на языке программирования Python:

import random

import numpy as np

from copy import deepcopy

from typing import Callable, Tuple, Union

def rosenbrock(a, b, n, f0, x):

    result = f0

    for i in range(0, n - 1):

        result += a \* (x[i]\*\*2 - x[i + 1])\*\*2 + b \* (x[i] - 1)\*\*2

    return result

def rosenbrock\_gradient(a, b, n, x):

    gradient = [0.0 for \_ in range(0, n)]

    gradient[0] = 2 \* a \* (x[0]\*\*2 - x[1]) \* 2 \* x[0] + 2 \* b \* (x[0] - 1)

    for i in range(1, n - 1):

        gradient[i] = 2 \* a \* (x[i - 1]\*\*2 - x[i]) \* (-1) + 2 \* a \* (x[i]\*\*2 - x[i + 1]) \* 2 \* x[i] + 2 \* b \* (x[i] - 1)

    gradient[n - 1] = 2 \* a \* (x[n - 2]\*\*2 - x[n - 1]) \* (-1)

    return gradient

def \_order(x: np.ndarray, fx: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:

    indices = np.argsort(fx)

    return x[indices], fx[indices]

def optimize\_1d(

    fun: Callable[[float], float],

    x0: float,

    maxiter: Union[int, None] = None,

    initial\_simplex: Union[np.ndarray, None] = None

) -> Tuple[float, float]:

    if initial\_simplex is not None:

        if initial\_simplex.ndim != 1 or len(initial\_simplex) != 2:

            raise ValueError("initial\_simplex must be a 1D array of length 2 for 1D optimization.")

        x = initial\_simplex.copy()

    else:

        h = 0.05 if x0 != 0 else 0.00025

        x = np.array([x0, x0 + h])

    if maxiter is None:

        maxiter = 200

    alpha = 1.0

    gamma = 2.0

    rho = 0.5

    sigma = 0.5

    fx = np.array([fun(x[0]), fun(x[1])])

    x, fx = \_order(x, fx)

    for \_ in range(maxiter):

        xo = x[0]

        xr = xo + alpha \* (xo - x[1])

        fxr = fun(xr)

        if fx[0] <= fxr < fx[1]:

            x[1] = xr

            fx[1] = fxr

        elif fxr < fx[0]:

            xe = xo + gamma \* (xr - xo)

            fxe = fun(xe)

            if fxe < fxr:

                x[1] = xe

                fx[1] = fxe

            else:

                x[1] = xr

                fx[1] = fxr

        else:

            if fxr < fx[1]:

                xc = xo + rho \* (xr - xo)

                fxc = fun(xc)

                if fxc < fxr:

                    x[1] = xc

                    fx[1] = fxc

            else:

                xc = xo + rho \* (x[1] - xo)

                fxc = fun(xc)

                if fxc < fx[1]:

                    x[1] = xc

                    fx[1] = fxc

                else:

                    x[1] = x[0] + sigma \* (x[1] - x[0])

                    fx[1] = fun(x[1])

        x, fx = \_order(x, fx)

        if abs(fx[1] - fx[0]) < 1e-6:

            break

    return x[0], fx[0]

def fletcher\_reeves(a, b, x, n=2, epsilon1=1e-6, delta2=1e-6, epsilon2=1e-6, M=10000):

    xs\_am = [None for \_ in range(n\_param)]

    gradient\_am = [None for \_ in range(n\_param)]

    result = []

    iterations = []

    def fn\_am(a):

        n\_xs = [None for \_ in range(0, len(xs\_am))]

        for i in range(0, len(xs\_am)):

            n\_xs[i] = xs\_am[i] - a \* gradient\_am[i]

        return rosenbrock(a\_param, b\_param, n\_param, f0\_param, n\_xs)

    def gam(xs, gradient):

        for i in range(0, len(xs)):

            xs\_am[i] = xs[i]

            gradient\_am[i] = gradient[i]

        p = optimize\_1d(fn\_am, x0=0)[0]

        return p

    k = 0

    previous\_x = [0 for \_ in range(n)]

    d\_k = 0

    d\_k\_1 = 0

    w\_k = 0

    while True:

        gradient = np.array(rosenbrock\_gradient(a, b, n, x), dtype=float)

        if np.linalg.norm(gradient) < epsilon1:

            return x, k, result, iterations

        if k >= M:

            return x, k, result, iterations

        if k == 0:

          d\_k = -gradient

        else:

          tmp1 = (np.linalg.norm(rosenbrock(a\_param, b\_param, n\_param, f0\_param, x)))\*\*2

          tmp2 = (np.linalg.norm(rosenbrock(a\_param, b\_param, n\_param, f0\_param, previous\_x)))\*\*2

          w\_k = tmp1 / tmp2

          d\_k = -gradient + w\_k \* d\_k\_1

        arg = gam(x, d\_k)

        previous\_x = deepcopy(x)

        x = x - arg \* d\_k

        result.append(rosenbrock(a\_param, b\_param, n\_param, f0\_param, x))

        if (np.linalg.norm(abs(x - previous\_x)) < delta2) and (abs(rosenbrock(a\_param, b\_param, n\_param, f0\_param, x) - rosenbrock(a\_param, b\_param, n\_param, f0\_param, previous\_x)) < epsilon2):

            return x, k, result, iterations

        iterations.append(k)

        k += 1

def polak\_ribiere(a, b, x, n=2, epsilon1=1e-7, delta2=1e-7, epsilon2=1e-7, M=10000):

    xs\_am = [None for \_ in range(n\_param)]

    gradient\_am = [None for \_ in range(n\_param)]

    result = []

    iterations = []

    def fn\_am(a):

        n\_xs = [None for \_ in range(0, len(xs\_am))]

        for i in range(0, len(xs\_am)):

            n\_xs[i] = xs\_am[i] - a \* gradient\_am[i]

        return rosenbrock(a\_param, b\_param, n\_param, f0\_param, n\_xs)

    def gam(xs, gradient):

        for i in range(0, len(xs)):

            xs\_am[i] = xs[i]

            gradient\_am[i] = gradient[i]

        p = optimize\_1d(fn\_am, x0=0)[0]

        return p

    k = 0

    previous\_x = [0 for \_ in range(n)]

    previous\_gradient = [0 for \_ in range(n)]

    d\_k = 0

    d\_k\_1 = 0

    while True:

        gradient = np.array(rosenbrock\_gradient(a, b, n, x), dtype=float)

        if np.linalg.norm(gradient) < epsilon1:

            return x, k, result, iterations

        if k >= M:

            return x, k, result, iterations

        if k > 0:

            grad\_diff = np.array(gradient) - np.array(previous\_gradient)

            if np.dot(previous\_gradient, previous\_gradient) != 0:

                beta\_k = np.dot(grad\_diff, grad\_diff) / np.dot(previous\_gradient, previous\_gradient)

                if np.isfinite(beta\_k):

                    d\_k = -gradient + beta\_k \* d\_k\_1

                else:

                    d\_k = -gradient

            else:

                d\_k = -gradient

        else:

            d\_k = -gradient

        arg = gam(x, d\_k)

        previous\_x = deepcopy(x)

        x = x - arg \* d\_k

        result.append(rosenbrock(a\_param, b\_param, n\_param, f0\_param, x))

        if (np.linalg.norm(abs(x - previous\_x)) < delta2) and (abs(rosenbrock(a\_param, b\_param, n\_param, f0\_param, x) - rosenbrock(a\_param, b\_param, n\_param, f0\_param, previous\_x)) < epsilon2):

            return x, k, result, iterations

        iterations.append(k)

        k += 1

def davidon\_fletcher\_powell(a, b, x, n=2, epsilon1=1e-6, delta2=1e-6, epsilon2=1e-6, M=10000):

    xs\_am = [None for \_ in range(n\_param)]

    gradient\_am = [None for \_ in range(n\_param)]

    result = []

    iterations = []

    def fn\_am(a):

        n\_xs = [None for \_ in range(0, len(xs\_am))]

        for i in range(0, len(xs\_am)):

            n\_xs[i] = xs\_am[i] - a \* gradient\_am[i]

        return rosenbrock(a\_param, b\_param, n\_param, f0\_param, n\_xs)

    def gam(xs, gradient):

        for i in range(0, len(xs)):

            xs\_am[i] = xs[i]

            gradient\_am[i] = gradient[i]

        p = optimize\_1d(fn\_am, x0=0)[0]

        return p

    G = np.eye(n)

    k = 0

    previous\_x = [0 for \_ in range(n)]

    previous\_gradient = [0 for \_ in range(n)]

    while True:

        gradient = np.array(rosenbrock\_gradient(a, b, n, x), dtype=float)

        if np.linalg.norm(gradient) < epsilon1:

            return x, k, result, iterations

        if k >= M:

            return x, k, result, iterations

        if k >= 1:

            delta\_g = gradient - previous\_gradient

            delta\_x = x - previous\_x

            delta\_G = np.outer(delta\_x, delta\_x) / np.dot(delta\_x, delta\_g) - np.outer(G @ delta\_g, G @ delta\_g) / np.dot(delta\_g, G @ delta\_g)

            G += delta\_G

        d = G @ gradient

        arg = gam(x, d)

        previous\_x = deepcopy(x)

        x = x - arg \* d

        previous\_gradient = deepcopy(gradient)

        result.append(rosenbrock(a\_param, b\_param, n\_param, f0\_param, x))

        if (np.linalg.norm(abs(x - previous\_x)) < delta2) and (abs(rosenbrock(a\_param, b\_param, n\_param, f0\_param, x) - rosenbrock(a\_param, b\_param, n\_param, f0\_param, previous\_x)) < epsilon2):

            return x, k, result, iterations

        iterations.append(k)

        k += 1

def levenberg\_marquardt(xs, epsilon1=1e-7, mu\_k=10000, M=10000):

    def find\_H(a, b, n, x):

        length = len(x)

        H = [[0.0 for \_ in range(0, length)] for \_ in range(0, length)]

        H[0][0] = 12 \* a \* x[0]\*\*2 - 4 \* a \* x[1] + 2 \* b

        H[0][1] = -4 \* a \* x[0]

        for i in range(1, n - 1):

            H[i][i - 1] = -4 \* a \* x[i - 1]

            H[i][i] = 12 \* a \* x[i]\*\*2 - 4 \* a \* x[i + 1] + 2 \* b + 2 \* a

            H[i][i + 1] = -4 \* a \* x[i]

        H[n - 1][n - 2] = -4 \* a \* x[n - 2]

        H[n - 1][n - 1] = 2 \* a

        return H

    iterations = []

    result = []

    for k in range(0, M):

        iterations.append(k)

        gradient = rosenbrock\_gradient(a\_param , b\_param , n\_param , xs)

        if np.linalg.norm(gradient) < epsilon1:

            break

        H = find\_H(a\_param , b\_param , n\_param , xs)

        ll = len(xs)

        matrix = np.zeros((ll, ll))

        for i in range(0, ll):

            for j in range(0, ll):

                if i == j:

                    matrix[i][j] = H[i][j] + mu\_k

                else:

                    matrix[i][j] = H[i][j]

        mat\_inv = np.linalg.inv(matrix)

        d = [0.0 for \_ in range(0, ll)]

        xs\_prev = deepcopy(xs)

        for i in range(0, ll):

            tmp = 0.0

            for j in range(0, ll):

                tmp += mat\_inv[i][j] \* gradient[j]

            d[i] = tmp

            xs[i] -= d[i]

        if rosenbrock(a\_param, b\_param, n\_param, f0\_param, xs) < rosenbrock(a\_param, b\_param, n\_param, f0\_param, xs\_prev):

            mu\_k = mu\_k / 2

        else:

            mu\_k = mu\_k \* 2

        result.append(rosenbrock(a\_param, b\_param, n\_param, f0\_param, xs))

    result.append(rosenbrock(a\_param, b\_param, n\_param, f0\_param, xs))

    return xs, iterations, result

a\_param = 300

b\_param = 5

n\_param = 2

f0\_param = 15

x0 = [random.uniform(-2.0, 2.0) for \_ in range(0, n\_param)]

print(f'a = {a\_param}\nb = {b\_param}\nf0 = {f0\_param}\nn = {n\_param}')

# Метод сопряженных градиентов Флетчера-Ривза

x, k, result, iterations = fletcher\_reeves(a\_param, b\_param, deepcopy(x0))

print(f'\nМетод Флетчера-Ривза')

print(f'Кол-во итераций: {k}')

print(f'x = {x}')

print(f'f(x) = {result[len(result) - 1]}')

# Метод сопряженных градиентов Полака-Рибьера

x, k, result, iterations = polak\_ribiere(a\_param, b\_param, deepcopy(x0))

print(f'\nМетод Полака-Рибьера')

print(f'Кол-во итераций: {k}')

print(f'x = {x}')

print(f'f(x) = {result[len(result) - 1]}')

# Квазиньютоновский метод Девидона-Флетчера-Пауэлла

x, k, result, iterations = davidon\_fletcher\_powell(a\_param, b\_param, deepcopy(x0))

print(f'\nМетод Девидона-Флетчера-Пауэлла')

print(f'Кол-во итераций: {k}')

print(f'x = {x}')

print(f'f(x) = {result[len(result) - 1]}')

# Метод Левенберга-Марквардта

x, k, result = levenberg\_marquardt(deepcopy(x0))

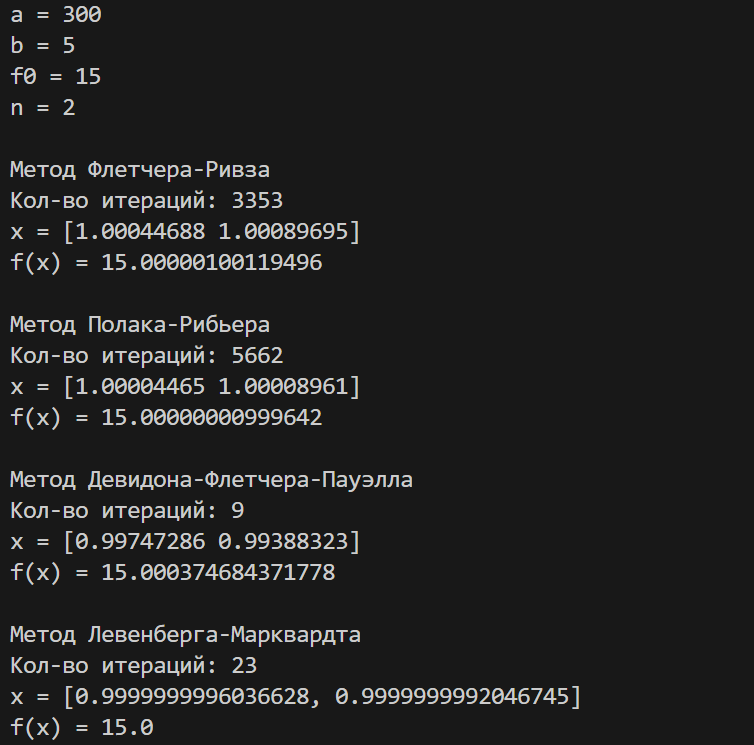
print(f'\nМетод Левенберга-Марквардта')

print(f'Кол-во итераций: {len(result) - 1}')

print(f'x = {x}')

print(f'f(x) = {result[len(result) - 1]}')

# Результаты



# Вывод

В ходе выполнения данной работы были изучены алгоритмы многомерного поиска, а также разработана программа для их реализации. Сравнение скорости достижения оптимизации для различных методов показало, что метод Девидона-Флетчера-Пауэлла требует меньше итераций для достижения результата.