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

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

$$a = 300, b = 5, f_0 = 15, n = 2$$

Практическая реализация

Исходный текст программы на языке программирования Python:

```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

```
a = 300
b = 5
f0 = 15
n = 2

Метод Флетчера-Ривза
Кол-во итераций: 3353
x = [1.00044688 1.00089695]
f(x) = 15.00000100119496

Метод Полака-Рибьера
Кол-во итераций: 5662
x = [1.00004465 1.00008961]
f(x) = 15.00000000999642

Метод Девидона-Флетчера-Пауэлла
Кол-во итераций: 9
x = [0.99747286 0.99388323]
f(x) = 15.000374684371778

Метод Левенберга-Марквардта
Кол-во итераций: 23
x = [0.9999999996036628, 0.9999999992046745]
f(x) = 15.0
```

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.