# Домашнее задание № 3: Методы многослойной оптимизации 1-го и 2-го порядка

15 октября 2024 г.

Сергей Виленский, ИУ9-72Б

## Цель работы

Изучение и сравнение методов многомерной оптимизации 1-го и 2-го порядка.

## Постановка задачи

1. Реализовать следующие методы оптимизации:
    1. `GD` - метод градиентного спуска
    2. `CG` - метод сопряженных градиентов
        1. `F-R` - Флетчера-Ривза
        2. `P-R` - Полака-Рибьера
    3. `D-F-P` - квазиньютоновский метод
    4. `L-M` - метод Левенберга-Марквадрата
2. Провести сравнительный анализ эффективности работы реализованных методов на примере функции Розенброка.
3. Подготовить отчет с распечаткой текста программы, графиками результатов исследования и анализом результатов.

## Реализация

```python
import numpy as np
import matplotlib.pyplot as plt
from dataclasses import dataclass
from typing import Callable, Tuple

Func = Callable[[float], float]

@dataclass
```

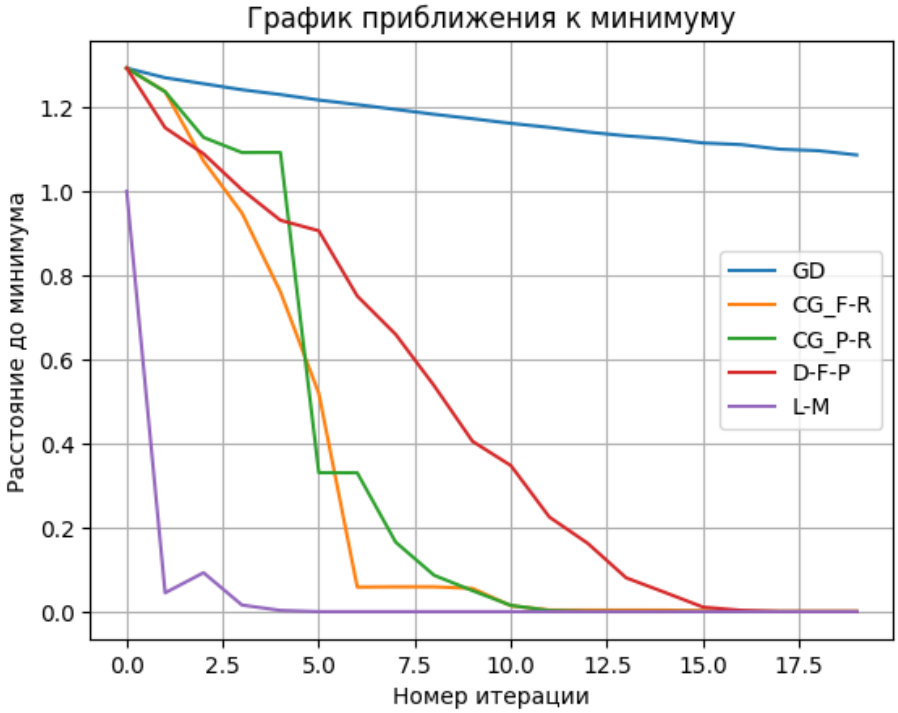
```python
class Function:
    f: Callable[[float, float], float]
    grad: Callable[[float, float], Tuple[float, float]]
    hessian: Callable[[float, float], Tuple[Tuple[float, float], Tuple[float, float]]]

Rosenbrock = Function(
    f = lambda x: 100 * (x[1] - x[0] ** 2) ** 2 + (1 - x[0]) ** 2,
    grad = lambda x: np.array((
        400 * x[0] * (x[0] ** 2 - x[1]) + 2 * (x[0] - 1),
        200 * (x[1] - x[0] ** 2),
    )),
    hessian = lambda x: np.array((
        (
            400 * (3 * x[0] ** 2 - x[1]) + 2,
            -400 * x[0],
        ),
        (
            -400 * x[0],
            200,
        ),
    ))
)

def optimal_by_golden_ratio(f, x_k, d_k):
    golden_ratio = (1 + 5 ** .5) / 2
    a_min, a_max = 0, 1

    while a_max - a_min > 1e-5:
        part = (a_max - a_min) / golden_ratio
        a_1 = a_min + part
        a_2 = a_max - part

        f_a_1 = f(x_k + a_1 * d_k)
        f_a_2 = f(x_k + a_2 * d_k)

        if f_a_1 <= f_a_2:
            a_min = a_1
        else:
            a_max = a_2

    return (a_max + a_min) / 2

def method_GD(function, x_k, k):
    result_path = []

    for _ in range(k):
```

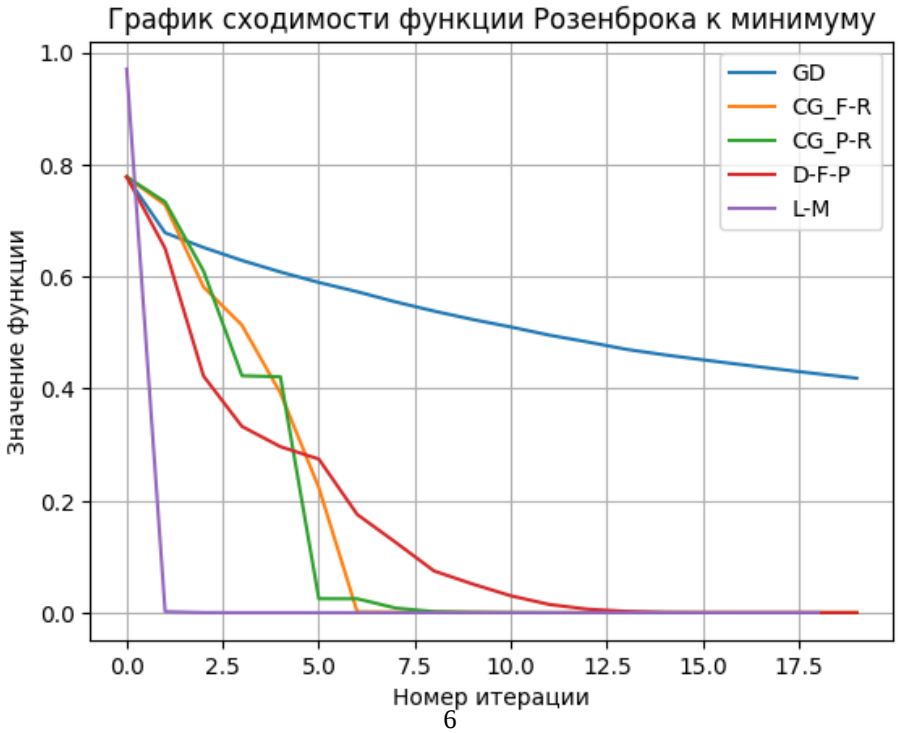
```
        d_k = -function.grad(x_k)
        a_k = optimal_by_golden_ratio(function.f, x_k, d_k)
        x_k = x_k + a_k * d_k
        result_path.append(x_k)

    return result_path

def method_CG_FR(function, x_k, k):
    result_path = []

    grad = function.grad(x_k)
    d_k = -grad

    for _ in range(k):
        a_k = optimal_by_golden_ratio(function.f, x_k, d_k)
        x_k = x_k + a_k * d_k
        result_path.append(x_k)

        grad_new = function.grad(x_k)
        beta = (np.linalg.norm(grad_new) / np.linalg.norm(grad)) ** 2

        d_k = -grad_new + beta * d_k
        grad = grad_new

    return result_path

def method_CG_PR(function, x_k, k):
    result_path = []

    grad = function.grad(x_k)
    d_k = -grad

    for i in range(k):
        a_k = optimal_by_golden_ratio(function.f, x_k, d_k)
        x_k = x_k + a_k * d_k
        result_path.append(x_k)

        grad_new = function.grad(x_k)

        if i % 3 == 1:
            beta = np.dot(grad_new, grad_new) / np.dot(grad, grad)
        else:
            beta = np.dot(grad_new - grad, grad_new) / np.dot(grad, grad)

        d_k = -grad_new + beta * d_k
        grad = grad_new
```

```
    return result_path

def method_DFP(function, x_k, k):
    result_path = []

    grad = function.grad(x_k)
    H_k = np.identity(2)

    for _ in range(k):
        d_k = -np.dot(H_k, grad)
        x_k_old = x_k

        a_k = optimal_by_golden_ratio(function.f, x_k, d_k)
        x_k = x_k_old + a_k * d_k
        result_path.append(x_k)

        grad_new = function.grad(x_k)

        delta_x_k = np.transpose(np.array((x_k - x_k_old,)))
        delta_grad = np.transpose(np.array((grad_new - grad,)))

        H_k = (
            H_k
            + np.dot(delta_x_k, np.transpose(delta_x_k))
                / np.dot(np.transpose(delta_x_k), delta_grad)[0][0]
            - np.linalg.multi_dot((H_k, delta_grad, np.transpose(delta_grad), H_k))
                / np.linalg.multi_dot((np.transpose(delta_grad), H_k, delta_grad))[0][0]
        )

        grad = grad_new

    return result_path

def method_LM(function, x_k, k):
    learning_rate = 1e-5
    result_path = []

    f_val = function.f(x_k)

    for _ in range(k):
        delta_x = - np.dot(
            np.linalg.inv(function.hessian(x_k)) + np.identity(2) * learning_rate,
            function.grad(x_k)
        )
```

```
            x_k = x_k + delta_x
            result_path.append(x_k)

            f_val_new = function.f(x_k)
            if f_val_new < f_val:
                learning_rate /= 10
            else:
                learning_rate *= 10

        return result_path


dist_to_min = lambda x: np.linalg.norm(x - np.array((1, 1)))
f_value = Rosenbrock.f

def prepare_result(result_path):
    return zip(*enumerate(map(dist_to_min, result_path)))

x_0_1 = np.array((-1.2, 1))
x_0_2 = np.array((0, 0))
x_0_3 = np.array((-2, 2))
x_0 = x_0_2
k = 20

plt.plot(*prepare_result(method_GD(Rosenbrock, x_0, k)), label="GD")
plt.plot(*prepare_result(method_CG_FR(Rosenbrock, x_0, k)), label="CG_F-R")
plt.plot(*prepare_result(method_CG_PR(Rosenbrock, x_0, k)), label="CG_P-R")
plt.plot(*prepare_result(method_DFP(Rosenbrock, x_0, k)), label="D-F-P")
plt.plot(*prepare_result(method_LM(Rosenbrock, x_0, k)), label="L-M")

plt.title("График сходимости функции Розенброка к минимуму")
plt.xlabel("Номер итерации")
plt.ylabel("Расстояние до минимума")
plt.grid(True)
plt.legend()
plt.show()
```

## Сравнение методов



## Выводы

В результате сравнения заданных методов 1-го порядка оптимизации можно заключить, что наиболее эффективным является метод сопряженных градиентов Флетчера-Ривье, в то время как метод скорейшего спуска очевидно оказался наименее эффективным. Однако метод 2-го порядка, метод Левенберга-Марквадрата, на порядок быстрее сходится к минимуму по сравнению со всеми методами 1-го порядка.