Национальный исследовательский университет ИТМО
Прикладная математика и информатика

Методы оптимизации

Отчет по лабораторной работе №4
"Изучение алгоритмов метода Ньютона и его модификаций, в том числе квазиньютоновских методов"

Выполнили:

Михайлов Максим
Загребина Мария
Кулагин Ярослав

Команда:
$\forall \bar{R} \in \mathscr{R}^n : \mathbf{R}(\bar{R}) \in \mathscr{R}$
(КаМаЗ)

Группа: М3237

Санкт-Петербург, 7 июня 2021 г.

1 Цель

1. Разработать программы для безусловной минимизации функций многих переменных
2. Реализовать метод Ньютона
   - классический
   - с одномерным поиском
   - с направлением спуска
3. Продемонстрировать работу методов на 2-3 функциях, исследовать влияние выбора начального приближения на результат
4. Исследовать работу методов на двух функциях с заданным начальным приближением
   - $f(x) = x_1^2 + x_2^2 - 1.2x_1x_2,\ x^0 = (4,1)^T$
   - $f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2,\ x^0 = (-1.2,1)^T$
5. Реализовать метод Давидона-Флетчера-Пауэлла и метод Пауэлла и сравнить с наилучшим методом Ньютона

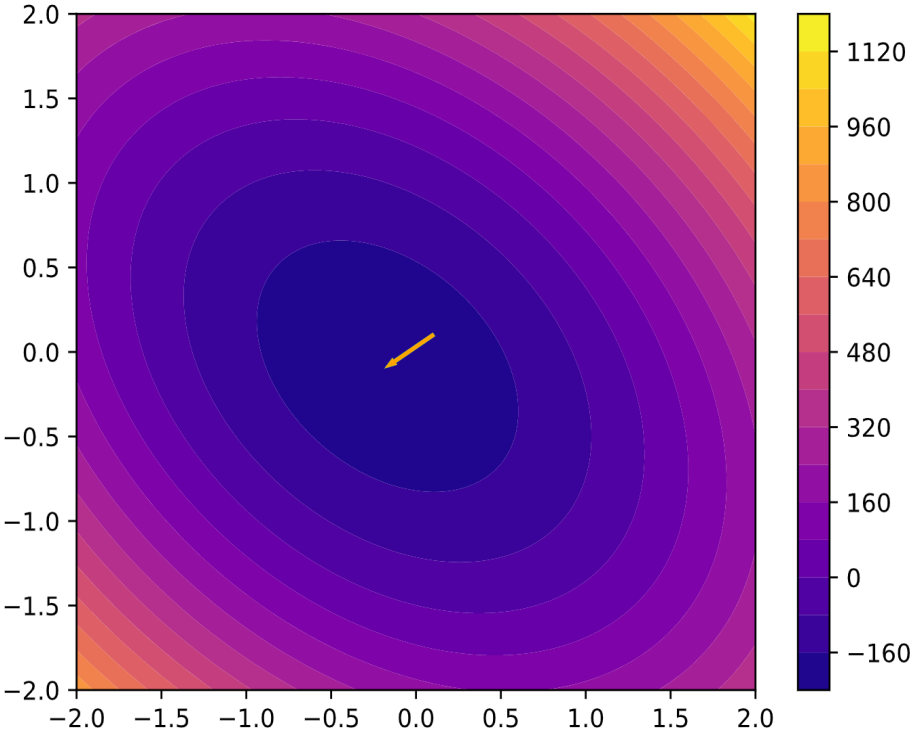
2 Ход работы

На всех иллюстрациях обозначены цвета:
Классический метод - зеленый
Метод Ньютона с одномерным поиском - голубой
Метод Ньютона с направлением спуска - оранжевый

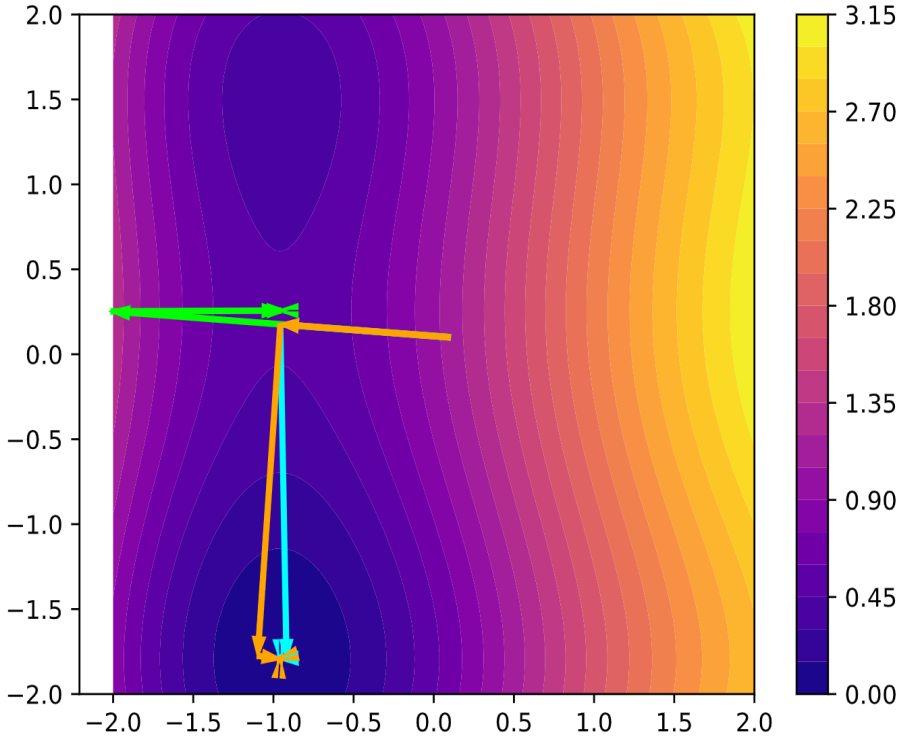
Во всех измерениях для одномерного поиска использовался метод Брента.

2.1 Метод Ньютона

$\varepsilon = 10^(-5)$
Начальная точка $(0.1, 0, 1)$ $f_1 = 108x^2 + 116y^2 + 80xy + 43x + 33y - 211$



$f_2 = sin(x) + cos(y) + 0.3y^2 + 0.3x^2 + 0.1y$



Классический метод Ньютона

Количество точек

Начальная точка	f_1	f_2
(0.1,0.1)	2	-
(1,1)	2	-
(2,2)	2	-
(3,3)	2	-
(10,10)	2	-

Метод Ньютона с одномерным поиском

Количество точек

Начальная точка	f_1	f_2
(0.1,0.1)	2	5
(1,1)	2	5
(2,2)	2	6
(3,3)	2	5
(10,10)	2	5

Метод Ньютона с направлением спуска

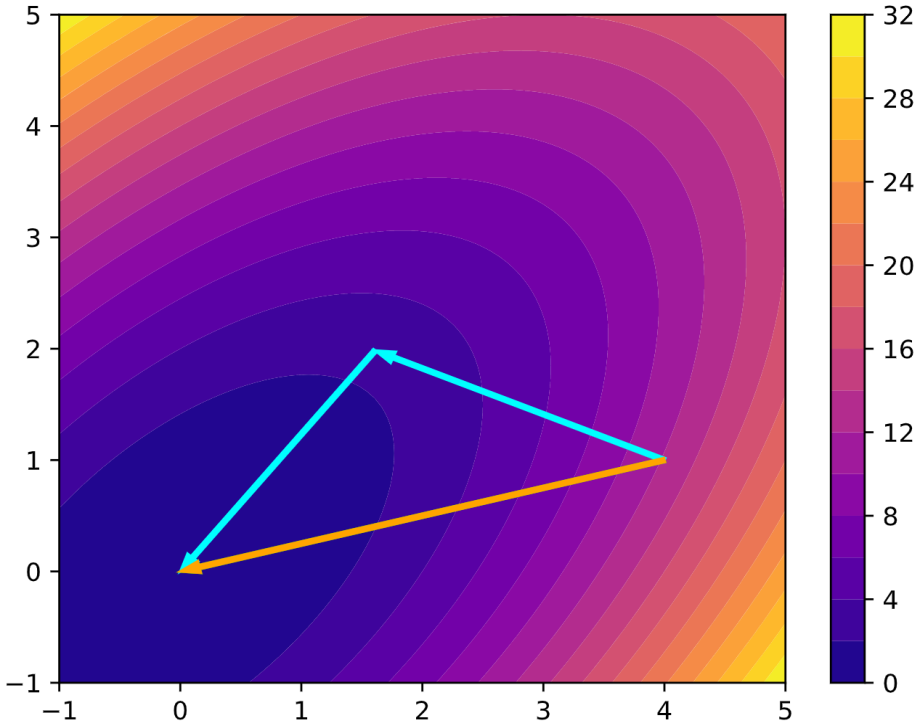
Количество точек

Начальная точка	f_1	f_2
(0.1,0.1)	3	5
(1,1)	3	5
(2,2)	3	6
(3,3)	3	5
(10,10)	3	6

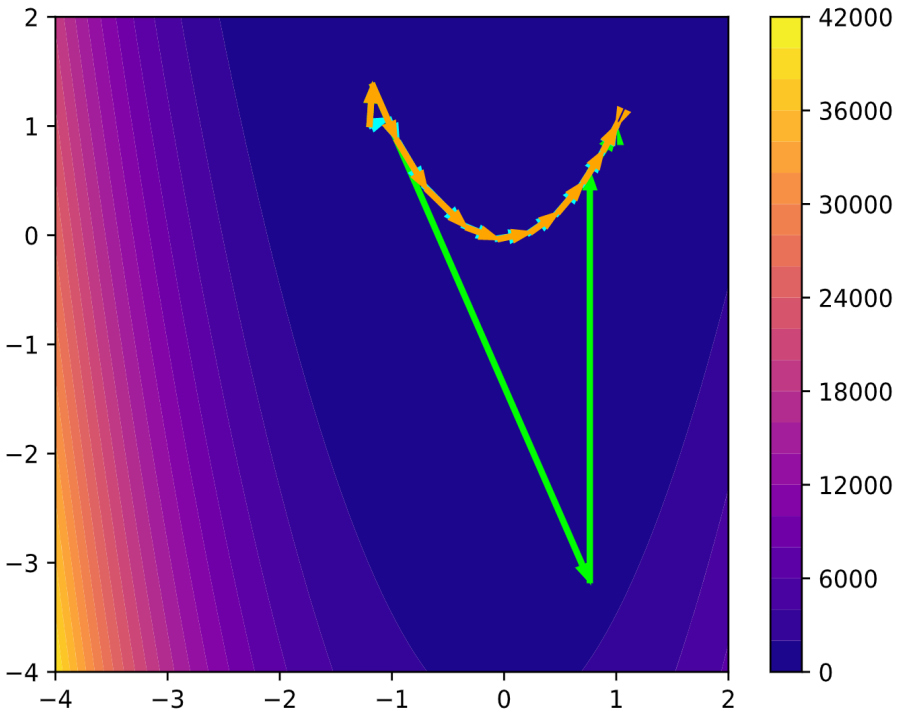
Классический метод Ньютона не всегда находит точку минимума на сложных функциях. Выбор начального приближения влияет на количество итераций методов. Так как матрица Гессе квадратичной функции положительно определена, все методы сходятся за одну итерацию и накладываются на графике.

2.2 Исследование на заданных функциях

$f_1 = x_1^2 + x_2^2 - 1.2x_1x_2,\ x^0 = (4,1)^T$



$f_2 = 100(x_2 - x_1^2)^2 + (1 - x_1)^2,\ x^0 = (-1.2, 1)^T$



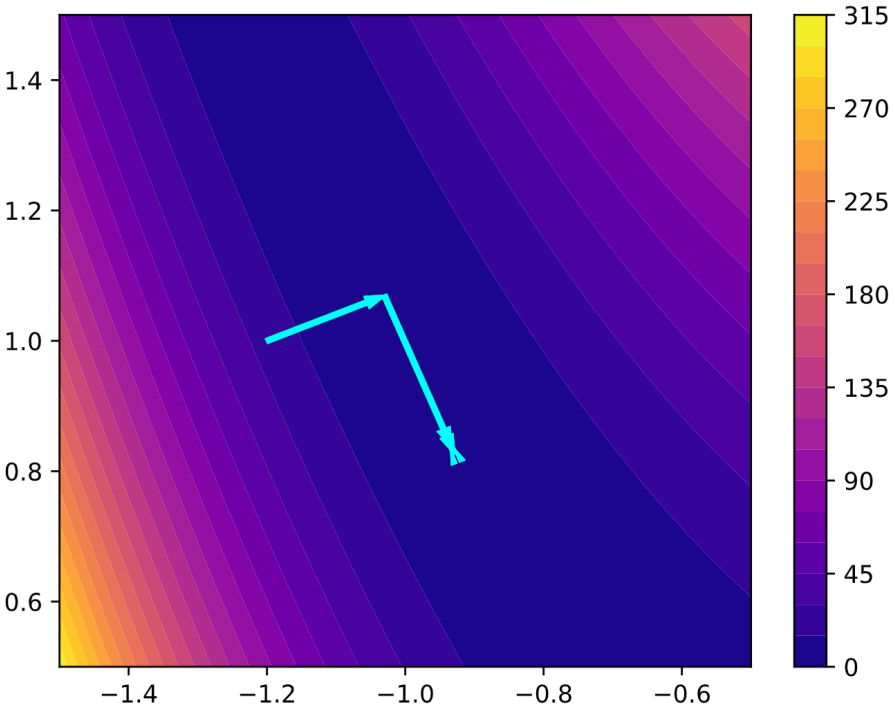
Количество точек

Метод	f_1	f_2
Классический	2	7
Одномерный поиск	2	13
С направлением спуска	2	13

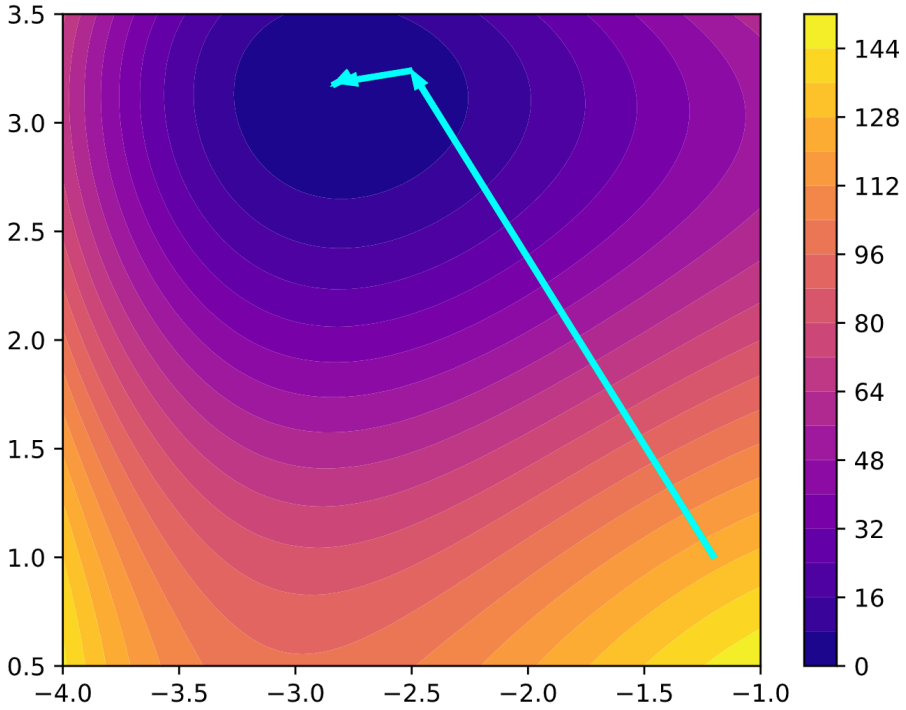
По результатам измерений на данных функциях наилучший метод Ньютона - классический. Все методы работают гораздо медленнее на овражной функции f_2. По сравнению с наискорейшим спуском из 2-ой лабораторной работы, методы используют меньшее число итераций, и не так сильно зависят от числа обусловленности.

2.3 Квазиньютоновские методы

Начальная точка $(-1.2, 1)$ $f_1 = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$,

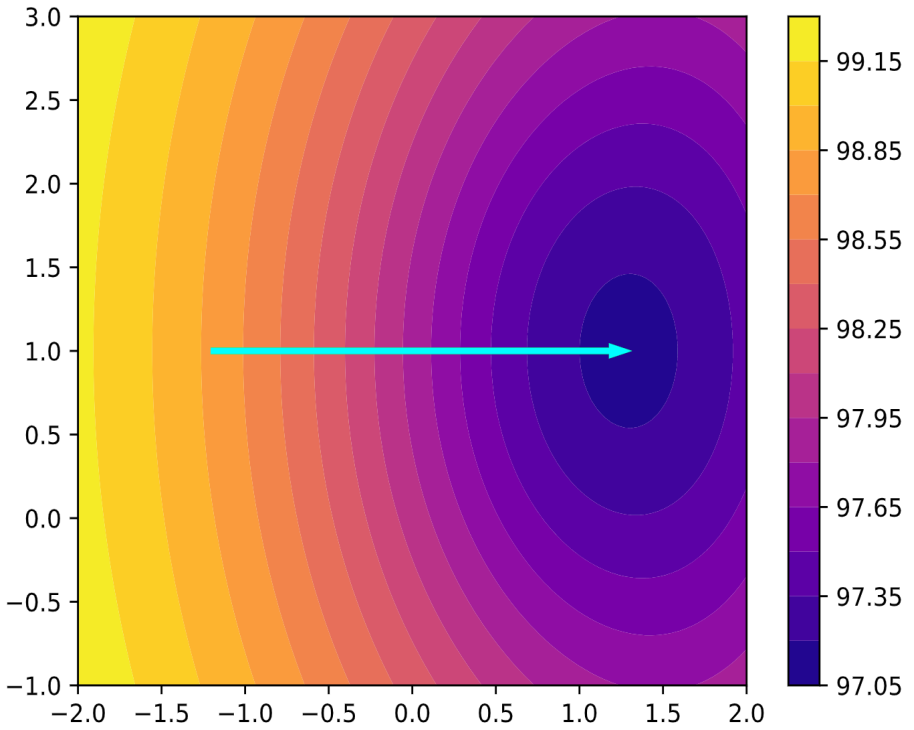


$f_2 = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2)$



$f_3 = (x_1 + 10x_2)^2 + 5(x_3 - x_4)^2 + (x_2 - 2x_3)^4 + 10(x_1 - x_4)^4$

$f_4 = 100 - \frac{2}{1+(\frac{x_1-1}{2})^2+(\frac{x_2-1}{3})^2} - \frac{1}{1+(\frac{x_1-2}{2})^2+(\frac{x_2-1}{3})^2}$



Классический метод Ньютона

Количество точек

Начальная точка	f_1	f_2	f_3	f_4
(0.1,0.1)	8	5	-	-
(-1.2,1)	8	9	-	-
(-2,2)	7	7	-	-
(3,-3)	7	6	-	-
(-5,-5)	6	6	-	-

Метод Давидона-Флетчера-Пауэлла

Количество точек

Начальная точка	f_1	f_2	f_3	f_4
(0.1,0.1)	-	16	31	31
(-1.2,1)	8	8	31	3
(-2,2)	6	4	15	31
(3,-3)	4	7	28	31
(-5,-5)	4	5	25	31

Метод Пауэлла

Количество точек

Начальная точка	f_1	f_2	f_3	f_4
(0.1,0.1)	-	20	31	16
(-1.2,1)	8	8	31	3
(-2,2)	6	4	31	31
(3,-3)	4	7	31	31
(-5,-5)	4	5	23	31

3 Выводы

1. Классический метод Ньютона работает быстро, но сходится не на всх функциях, так как в нем не оптимизации по выбору α. Остальные методы Ньютона более надежные и показали очень похожий результат.

2. Выбор начального приближения, как и число обусловленности, влият на количество итераций методов.

3. Методы ДФП и Пауэлла работают примерно с одинаковой скоростью и находят значения с заданной точностью на всех функциях, в отличие от классического метода, который был ограничен количеством итераций.

4 Исходный код

Функция

```cpp
#pragma once

#include <memory>
#include <vector>

#include "lab2/matrix.h"
#include "lab2/n_function.h"
#include "lab2/vector.h"

namespace lab4 {
    class NFunctionImpl : public lab2::NFunction {
      public:
        NFunctionImpl(std::size_t dim, std::function<double(lab2::Vector)> f,
                      std::function<lab2::Vector(lab2::Vector)> grad_f,
                      std::function<lab2::Matrix(lab2::Vector)> hessian_f);

        double operator()(lab2::Vector x) override;
        lab2::Vector grad(lab2::Vector x) override;
        std::size_t get_dim() override;
        lab2::Matrix hessian(lab2::Vector x) override;

      private:
        const std::size_t dim;
        const std::function<double(lab2::Vector)> f;
        const std::function<lab2::Vector(lab2::Vector)> grad_f;
        const std::function<lab2::Matrix(lab2::Vector)> hessian_f;
    };
}  // namespace lab4
```

Функция

```cpp
#include "lab4/n_function_impl.h"

#include <utility>

using namespace lab4;

NFunctionImpl::NFunctionImpl(
    std::size_t dim, std::function<double(lab2::Vector)> f,
    std::function<lab2::Vector(lab2::Vector)> grad_f,
    std::function<lab2::Matrix(lab2::Vector)> hessian_f)
    : dim(dim),
      f(std::move(f)),
      grad_f(std::move(grad_f)),
      hessian_f(std::move(hessian_f)) {}

lab2::Vector NFunctionImpl::grad(lab2::Vector x) { return grad_f(x); }

double NFunctionImpl::operator()(lab2::Vector x) { return f(x); }

lab2::Matrix NFunctionImpl::hessian(lab2::Vector x) { return hessian_f(x); }

std::size_t NFunctionImpl::get_dim() { return dim; }
```

Классический метод Ньютона

```hpp
#pragma once

#include "lab2/n_optimizer.h"

namespace lab4 {
    class ClassicNewton : public lab2::NOptimizer {
      public:
        ClassicNewton();

        lab2::Vector iteration(lab2::NFunction& f, double epsilon) override;
        bool is_done(lab2::NFunction& f, double epsilon) const override;

      private:
        lab2::Vector p;
    };
}  // namespace lab4
```

Классический метод Ньютона

```cpp
#include "lab4/classic_newton.h"

#include <lab3/solver.h>

#include <utility>

using namespace lab4;

ClassicNewton::ClassicNewton() : p(lab2::Vector({1})) {}

lab2::Vector ClassicNewton::iteration(lab2::NFunction& f, double) {
    lab2::Vector x = get_points().back();
    p              = lab3::Solver::solve(f.hessian(x), f.grad(x) * (-1));
    return x + p;
}

bool ClassicNewton::is_done(lab2::NFunction&, double epsilon) const {
    return p.norm() < epsilon;
}
```

метод Давидона-Флетчера-Пауэлла

```h
#pragma once

#include "lab2/matrix.h"
#include "lab2/n_optimizer.h"

namespace lab4 {
    class DFP : public lab2::NOptimizer {
      public:
        DFP();

        lab2::Vector iteration(lab2::NFunction& f, double epsilon) override;
        bool is_done(lab2::NFunction& f, double epsilon) const override;

      private:
        lab2::Vector w, p, delta_x;
        lab2::Matrix G;
        const double ONE_DIM_EPS = 1e-7;
        const int ONE_DIM_START = -100, ONE_DIM_END = 100;
    };
}  // namespace lab4
```

метод Давидона-Флетчера-Пауэлла

```cpp
#include "lab4/dfp.h"

#include "iostream"
#include "lab1/brent.h"
#include "lab3/solver.h"

using namespace lab4;

DFP::DFP()
    : w(lab2::Vector({1})),
      p(lab2::Vector({1})),
      delta_x(lab2::Vector({1})),
      G(lab2::Matrix({{1}}, std::nullopt)) {}

lab2::Vector DFP::iteration(lab2::NFunction &f, double) {
    const lab2::Vector x_k_1 = get_points().back(), grad = f.grad(x_k_1),
                       anti_grad = grad * (-1);
    if (iteration_count == 0) {
        w                = anti_grad;
        p                = w;
        const auto alpha = lab1::Brent(
                               [&f, x = x_k_1, p_ = p](double a) {
                                   return f(x + p_ * a);
                               },
                               ONE_DIM_EPS, ONE_DIM_START, ONE_DIM_END)
                               .optimize();
        delta_x = p * alpha;
        G       = lab2::Matrix::I(x_k_1.size());
        return x_k_1 + delta_x;
    }
    const auto w_k = anti_grad, delta_w_k = w_k - w, v_k = G * delta_w_k;
    G = G - lab2::Matrix::vector_mul(delta_x, delta_x) / (delta_w_k * delta_x)
        - lab2::Matrix::vector_mul(v_k, v_k) / (v_k * delta_w_k);
    p                = G * w_k;
    const auto alpha = lab1::Brent(
                           [&f, x = x_k_1, p_ = p](double a) {
                               return f(x + p_ * a);
                           },
                           ONE_DIM_EPS, ONE_DIM_START, ONE_DIM_END)
                           .optimize();
    delta_x = p * alpha;
    return x_k_1 + delta_x;
}

bool DFP::is_done(lab2::NFunction &, double epsilon) const {
    return delta_x.norm() < epsilon;
}
```

Метод Ньютона с направлением спуска

```h
#pragma once

#include "lab2/matrix.h"
#include "lab2/n_optimizer.h"

namespace lab4 {
    class DFP : public lab2::NOptimizer {
      public:
        DFP();

        lab2::Vector iteration(lab2::NFunction& f, double epsilon) override;
        bool is_done(lab2::NFunction& f, double epsilon) const override;

      private:
        lab2::Vector w, p, delta_x;
        lab2::Matrix G;
        const double ONE_DIM_EPS = 1e-7;
        const int ONE_DIM_START = -100, ONE_DIM_END = 100;
    };
}  // namespace lab4
```

Метод Ньютона с направлением спуска

```cpp
#include "lab4/dfp.h"

#include "iostream"
#include "lab1/brent.h"
#include "lab3/solver.h"

using namespace lab4;

DFP::DFP()
    : w(lab2::Vector({1})),
      p(lab2::Vector({1})),
      delta_x(lab2::Vector({1})),
      G(lab2::Matrix({{1}}, std::nullopt)) {}

lab2::Vector DFP::iteration(lab2::NFunction &f, double) {
    const lab2::Vector x_k_1 = get_points().back(), grad = f.grad(x_k_1),
                       anti_grad = grad * (-1);
    if (iteration_count == 0) {
        w                = anti_grad;
        p                = w;
        const auto alpha = lab1::Brent(
                               [&f, x = x_k_1, p_ = p](double a) {
                                   return f(x + p_ * a);
                               },
                               ONE_DIM_EPS, ONE_DIM_START, ONE_DIM_END)
                               .optimize();
        delta_x = p * alpha;
        G       = lab2::Matrix::I(x_k_1.size());
        return x_k_1 + delta_x;
    }
    const auto w_k = anti_grad, delta_w_k = w_k - w, v_k = G * delta_w_k;
    G = G - lab2::Matrix::vector_mul(delta_x, delta_x) / (delta_w_k * delta_x)
        - lab2::Matrix::vector_mul(v_k, v_k) / (v_k * delta_w_k);
    p                = G * w_k;
    const auto alpha = lab1::Brent(
                           [&f, x = x_k_1, p_ = p](double a) {
                               return f(x + p_ * a);
                           },
                           ONE_DIM_EPS, ONE_DIM_START, ONE_DIM_END)
                           .optimize();
    delta_x = p * alpha;
    return x_k_1 + delta_x;
}

bool DFP::is_done(lab2::NFunction &, double epsilon) const {
    return delta_x.norm() < epsilon;
}
```

Метод Пауэлла

```h
#pragma once

#include "lab2/matrix.h"
#include "lab2/n_optimizer.h"

namespace lab4 {
    class Powell : public lab2::NOptimizer {
      public:
        Powell();

        lab2::Vector iteration(lab2::NFunction& f, double epsilon) override;
        bool is_done(lab2::NFunction& f, double epsilon) const override;

      private:
        lab2::Vector w, p, delta_x;
        lab2::Matrix G;
        const double ONE_DIM_EPS = 1e-7;
        const int ONE_DIM_START = -100, ONE_DIM_END = 100;
    };
}  // namespace lab4
```

Метод Пауэлла

```cpp
#include "lab4/powell.h"

#include "iostream"
#include "lab1/brent.h"
#include "lab3/solver.h"

using namespace lab4;

Powell::Powell()
    : w(lab2::Vector({1})),
      p(lab2::Vector({1})),
      delta_x(lab2::Vector({1})),
      G(lab2::Matrix({{1}}, std::nullopt)) {}

lab2::Vector Powell::iteration(lab2::NFunction &f, double) {
    const lab2::Vector x_k_1 = get_points().back(), grad = f.grad(x_k_1),
                       anti_grad = grad * (-1);
    if (iteration_count == 0) {
        w                = anti_grad;
        p                = w;
        const auto alpha = lab1::Brent(
                               [&f, x = x_k_1, p_ = p](double a) {
                                   return f(x + p_ * a);
                               },
                               ONE_DIM_EPS, ONE_DIM_START, ONE_DIM_END)
                               .optimize();
        delta_x = p * alpha;
        G       = lab2::Matrix::I(x_k_1.size());
        return x_k_1 + delta_x;
    }
    const auto w_k = anti_grad, delta_w_k = w_k - w, v_k = G * delta_w_k,
               delta_x_wave_k = delta_x + G * delta_w_k;
    G                         = G
        - lab2::Matrix::vector_mul(delta_x_wave_k, delta_x_wave_k)
              / (delta_w_k * delta_x_wave_k);
    p                 = G * w_k;
    const auto alpha = lab1::Brent(
                           [&f, x = x_k_1, p_ = p](double a) {
                               return f(x + p_ * a);
                           },
                           ONE_DIM_EPS, ONE_DIM_START, ONE_DIM_END)
                           .optimize();
    delta_x = p * alpha;
    return x_k_1 + delta_x;
}

bool Powell::is_done(lab2::NFunction &, double epsilon) const {
    return delta_x.norm() < epsilon;
}
```