

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:

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

Группа: М3237

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

Обозначение цветов на иллюстрациях:

Классический метод - зеленый

Метод Ньютона с одномерным поиском - голубой

Метод Ньютона с направлением спуска - оранжевый

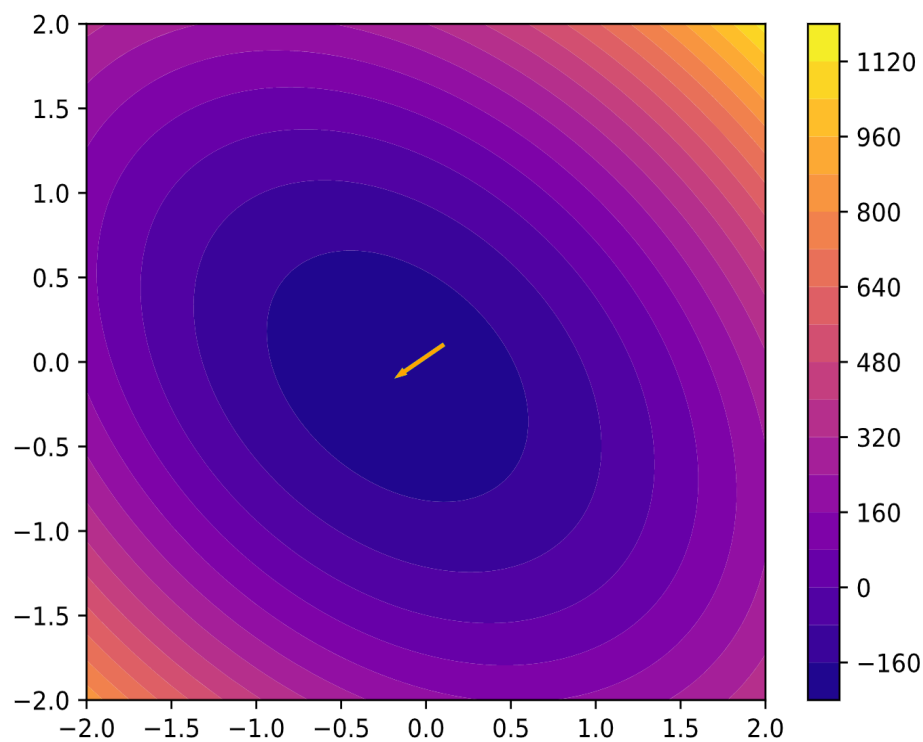
Во всех измерениях для одномерного поиска использовался метод Брента.

2.1 Метод Ньютона

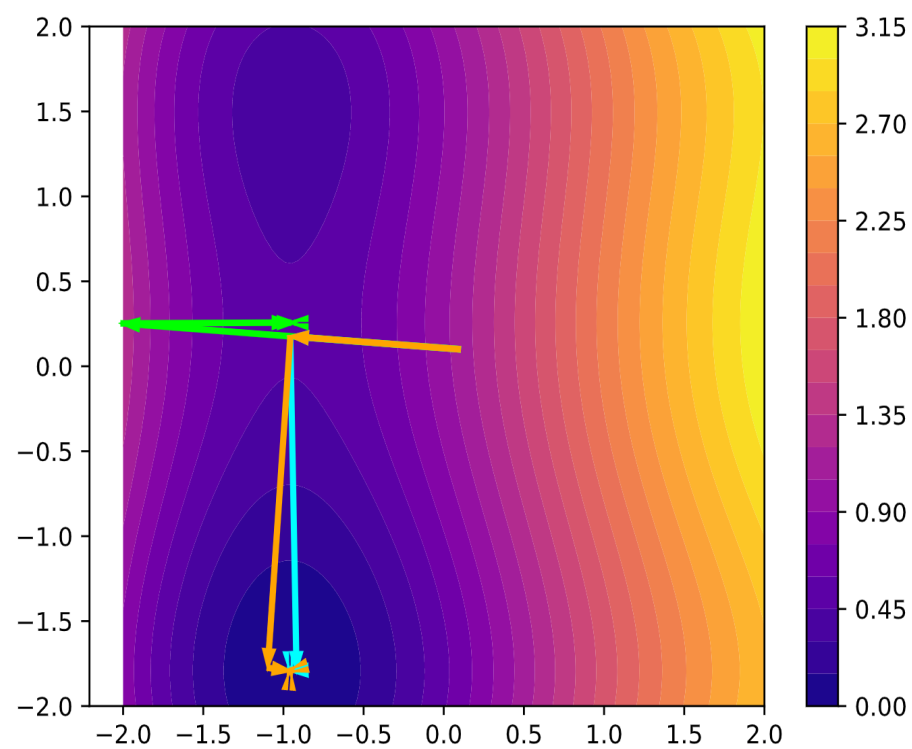
$$\varepsilon = 10^{-5}$$

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	-
(-5,-5)	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
(-5,-5)	2	5
(10,10)	2	5

α , полученные одномерным поиском для f_2

№	(0.1; 0.1)	(1; 1)	(2; 2)	(-5; -5)	(10; 10)
1	0.500975	-0.422133	-1.12804	0.157377	2.46763
2	-24.8939	-6.86726	6.07175	0.983785	0.72357
3	0.991506	1.03457	1.03309	1.11796	1.01044
4	1.00048	1.00347	1.01409	1.00082	1.00006
5	-	-	0.999954	-	-

Для f_1 $\alpha = 1$

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
(-5,-5)	3	5
(10,10)	3	6

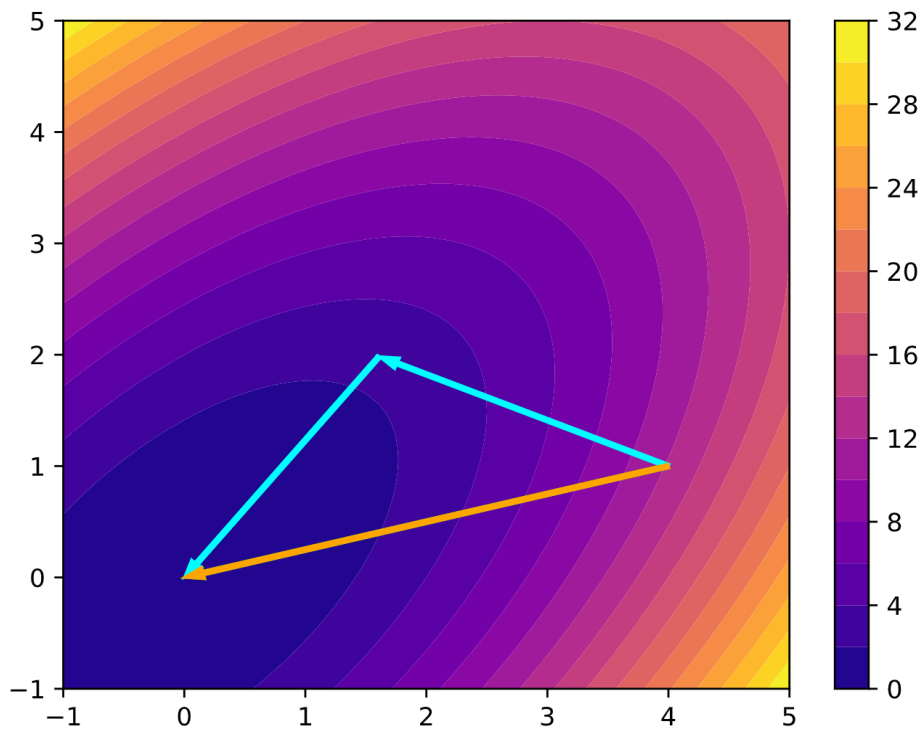
Если начальное приближение недостаточно близко к решению, то метод Ньютона может не сойтись.

Выбор начального приближения влияет на количество итераций методов.

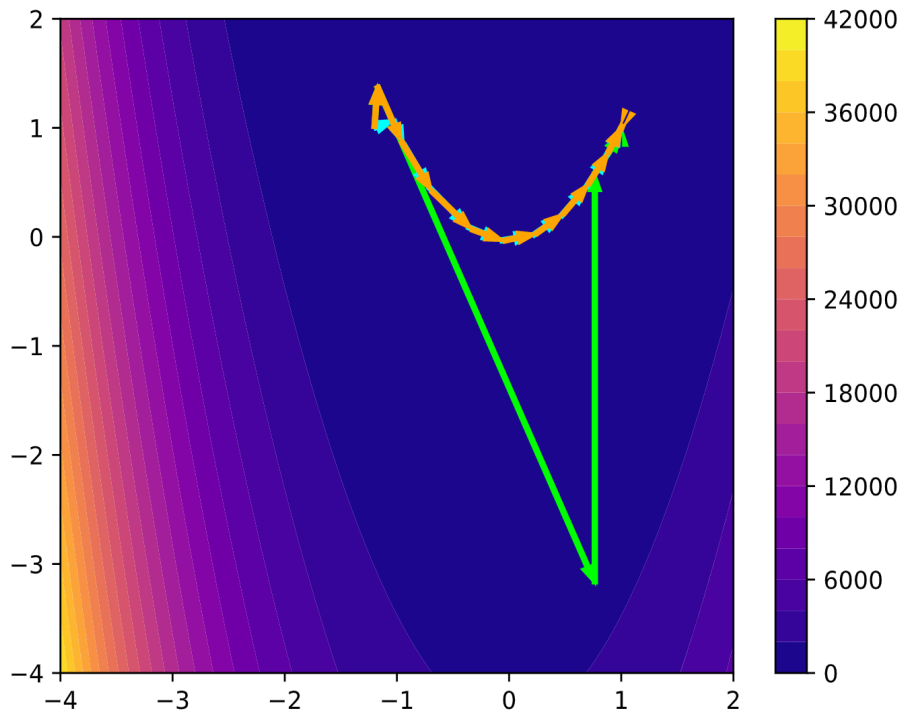
Так как матрица Гессе квадратичной функции положительно определена, все методы сходятся за одну итерацию и накладываются на графике.

2.2 Исследование на заданных функциях

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



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



Количество точек

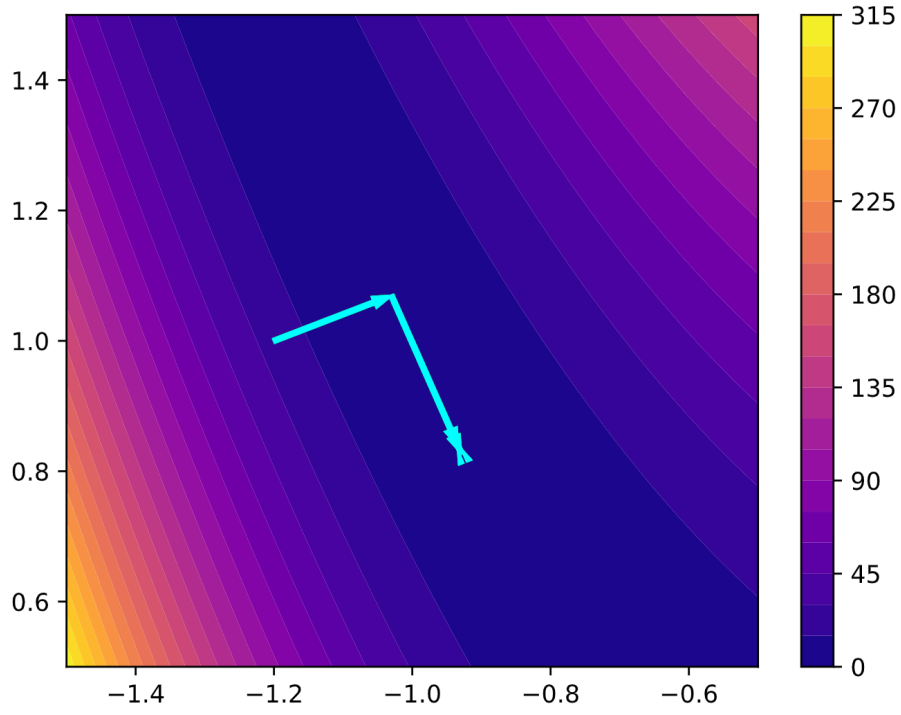
Метод	f_1	f_2
Классический	2	7
Одномерный поиск	2	14
С направлением спуска	2	13
Наискорейший спуск	18	97

По результатам измерений на данных функциях самый быстрый метод Ньютона - классический, но он не гарантирует сходимость, поэтому в следующем задании с квазиньютоновскими методами будет сравниваться метод с направлением спуска.

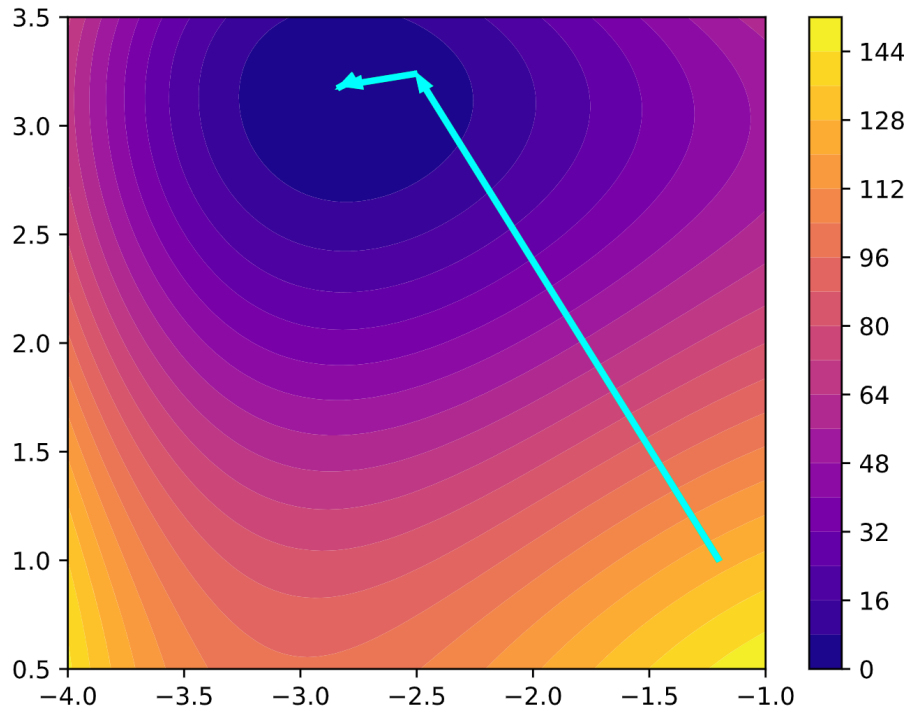
Все методы работают гораздо медленнее на овражной функции f_2 . По сравнению с наискорейшим спуском из 2-ой лабораторной работы, методы используют меньшее число итераций, и не так сильно зависят от числа обусловленности.

2.3 Квазиньютоновские методы

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

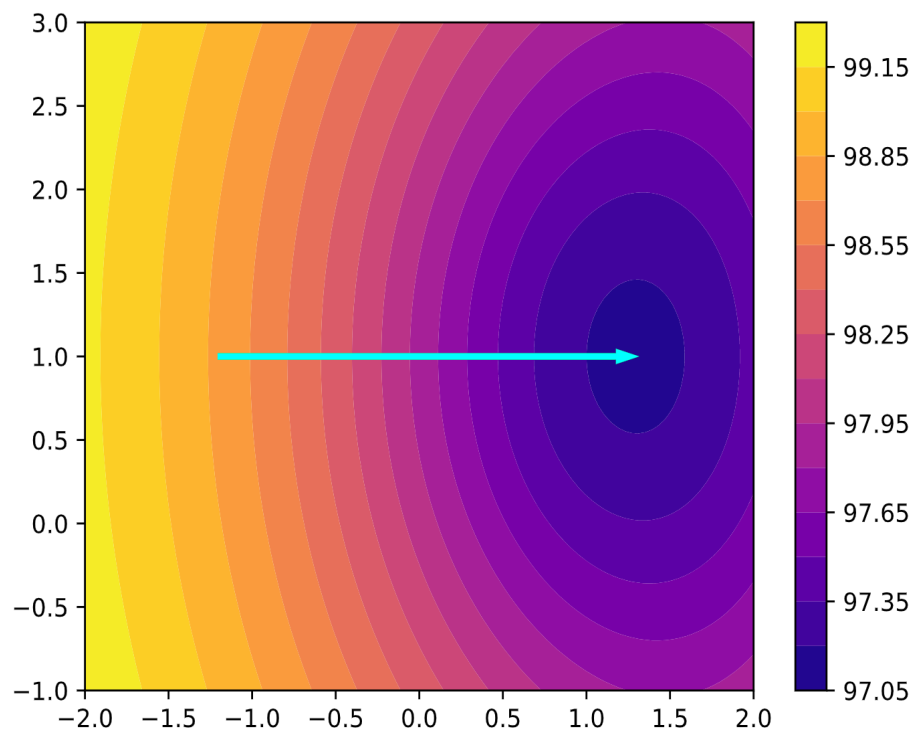


$$f_2 = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$$



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

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



Метод Ньютона с направлением спуска

Количество точек

Начальная точка	f_1	f_2	f_3	f_4
(0.1,0.1)	17	6	12	4
(-1.2,1)	13	5	26	3
(-2,2)	16	5	30	4
(3,-3)	12	5	10	5
(-5,-5)	11	5	31	7

Метод Давидона-Флетчера-Пауэлла

Количество точек

Начальная точка	f_1	f_2	f_3	f_4
(0.1,0.1)	31	10	31	7
(-1.2,1)	20	7	31	4
(-2,2)	27	6	31	8
(3,-3)	25	8	31	13
(-5,-5)	29	7	31	14

Метод Пауэлла

Количество точек

Начальная точка	f_1	f_2	f_3	f_4
(0.1,0.1)	31	10	31	9
(-1.2,1)	23	7	31	4
(-2,2)	27	6	31	15
(3,-3)	20	8	31	12
(-5,-5)	26	7	31	23

Все методы работают в несколько раз медленнее на функции с большим числом обусловленности и многомерной. Траектории всех методов оказались одинаковыми.

Метод Ньютона с направлением спуска сходится за меньшее количество итераций, чем квазиньютоновские методы, т.к. с большей точностью выбирает направление для движения.

3 Выводы

1. Классический метод Ньютона сходится не для каждого начального приближения, так как в нем нет оптимизации по выбору α . Остальные методы Ньютона более надежные и показали похожий результат.
2. Выбор начального приближения, как и число обусловленности, влияют на количество итераций методов.
3. Методы ДФП и Пауэлла работают примерно с одинаковой скоростью и находят значения с заданной точностью на всех рассмотренных функциях.
4. Метод Ньютона с направлением спуска находит минимум быстрее остальных методов, рассмотренных в данной работе.

4 Исходный код

Функция

```
1  #pragma once
2
3  #include <memory>
4  #include <vector>
5
6  #include "lab2/matrix.h"
7  #include "lab2/n_function.h"
8  #include "lab2/vector.h"
9
10 namespace lab4 {
11     class NFunctionImpl : public lab2::NFunction {
12     public:
13         NFunctionImpl(std::size_t dim, std::function<double(lab2::Vector)> f,
14                       std::function<lab2::Vector(lab2::Vector)> grad_f,
15                       std::function<lab2::Matrix(lab2::Vector)> hessian_f);
16
17         double operator()(lab2::Vector x) override;
18         lab2::Vector grad(lab2::Vector x) override;
19         std::size_t get_dim() override;
20         lab2::Matrix hessian(lab2::Vector x) override;
21
22     private:
23         const std::size_t dim;
24         const std::function<double(lab2::Vector)> f;
25         const std::function<lab2::Vector(lab2::Vector)> grad_f;
26         const std::function<lab2::Matrix(lab2::Vector)> hessian_f;
27     };
28 } // namespace lab4
```

Функция

```
1  #include "lab4/n_function_impl.h"
2
3  #include <utility>
4
5  using namespace lab4;
6
7  NFunctionImpl::NFunctionImpl(
8      std::size_t dim, std::function<double(lab2::Vector)> f,
9      std::function<lab2::Vector(lab2::Vector)> grad_f,
10     std::function<lab2::Matrix(lab2::Vector)> hessian_f)
11     : dim(dim),
12       f(std::move(f)),
13       grad_f(std::move(grad_f)),
14       hessian_f(std::move(hessian_f)) {}
15
16 lab2::Vector NFunctionImpl::grad(lab2::Vector x) { return grad_f(x); }
17
18 double NFunctionImpl::operator()(lab2::Vector x) { return f(x); }
19
20 lab2::Matrix NFunctionImpl::hessian(lab2::Vector x) { return hessian_f(x); }
21
22 std::size_t NFunctionImpl::get_dim() { return dim; }
```

Классический метод Ньютона

```
1  #pragma once
2
3  #include "lab2/n_optimizer.h"
4
5  namespace lab4 {
6      class ClassicNewton : public lab2::NOptimizer {
7      public:
8          ClassicNewton();
9
10         lab2::Vector iteration(lab2::NFunction& f, double epsilon) override;
11         bool is_done(lab2::NFunction& f, double epsilon) const override;
12
13     private:
14         lab2::Vector p;
15     };
16 } // namespace lab4
```

Классический метод Ньютона

```
1  #include "lab4/classic_newton.h"
2
3  #include <lab3/solver.h>
4
5  #include <utility>
6  #include "iostream"
7
8  using namespace lab4;
9
10 ClassicNewton::ClassicNewton() : p(lab2::Vector({1})) {}
11
12 lab2::Vector ClassicNewton::iteration(lab2::NFunction& f, double) {
13     lab2::Vector x = get_points().back();
14     p               = lab3::Solver::solve(f.hessian(x), f.grad(x) * (-1));
15     return x + p;
16 }
17
18 bool ClassicNewton::is_done(lab2::NFunction&, double epsilon) const {
19     return p.norm() < epsilon;
20 }
```

метод Давидона-Флетчера-Пауэлла

```
1  #pragma once
2
3  #include "lab2/matrix.h"
4  #include "lab2/n_optimizer.h"
5
6  namespace lab4 {
7      class DFP : public lab2::NOptimizer {
8      public:
9          DFP();
10
11          lab2::Vector iteration(lab2::NFunction& f, double epsilon) override;
12          bool is_done(lab2::NFunction& f, double epsilon) const override;
13
14      private:
15          lab2::Vector s, d;
16          lab2::Matrix G;
17          const double ONE_DIM_EPS = 1e-7;
18          const int ONE_DIM_START = -100, ONE_DIM_END = 100;
19      };
20  } // namespace lab4
```

метод Давидона-Флетчера-Пауэлла

```

1  #include "lab4/dfp.h"
2
3  #include "iostream"
4  #include "lab1/brent.h"
5  #include "lab3/solver.h"
6
7  using namespace lab4;
8
9  DFP::DFP()
10     : s(lab2::Vector({1})),
11       d(lab2::Vector({1})),
12       G(lab2::Matrix({{1}}, std::nullopt)) {}
13
14  lab2::Vector DFP::iteration(lab2::NFunction &f, double) {
15     lab2::Vector x_k_1 = get_points().back(), g_x = f.grad(x_k_1);
16     if (iteration_count == 0) {
17         G = lab2::Matrix::I(x_k_1.size());
18         d = g_x * (-1);
19         return x_k_1;
20     }
21     const auto r = lab1::Brent(
22         [&f, x = x_k_1, d_ = d](double a) {
23             return f(x + d_ * a);
24         },
25         ONE_DIM_EPS, ONE_DIM_START, ONE_DIM_END)
26         .optimize();
27     s = d * r;
28     auto x = x_k_1 + s, g_y = g_x;
29     g_x = f.grad(x);
30     auto p = g_x - g_y, v = G * p;
31     G = G + lab2::Matrix::vector_mul(s, s) / (s * p)
32         - lab2::Matrix::vector_mul(v, v) / (v * p);
33     d = (G * g_x) * (-1);
34     return x;
35 }
36
37 bool DFP::is_done(lab2::NFunction &, double epsilon) const {
38     return s.norm() < epsilon;
39 }

```


Метод Ньютона с направлением спуска

```
1  #pragma once
2
3  #include "lab2/matrix.h"
4  #include "lab2/n_optimizer.h"
5
6  namespace lab4 {
7      class DFP : public lab2::N0Optimizer {
8      public:
9          DFP();
10
11          lab2::Vector iteration(lab2::NFunction& f, double epsilon) override;
12          bool is_done(lab2::NFunction& f, double epsilon) const override;
13
14      private:
15          lab2::Vector s, d;
16          lab2::Matrix G;
17          const double ONE_DIM_EPS = 1e-7;
18          const int ONE_DIM_START = -100, ONE_DIM_END = 100;
19      };
20  } // namespace lab4
```

Метод Ньютона с направлением спуска

```
1  #include "lab4/dfp.h"
2
3  #include "iostream"
4  #include "lab1/brent.h"
5  #include "lab3/solver.h"
6
7  using namespace lab4;
8
9  DFP::DFP()
10     : s(lab2::Vector({1})),
11       d(lab2::Vector({1})),
12       G(lab2::Matrix({{1}}, std::nullopt)) {}
13
14  lab2::Vector DFP::iteration(lab2::NFunction &f, double) {
15     lab2::Vector x_k_1 = get_points().back(), g_x = f.grad(x_k_1);
16     if (iteration_count == 0) {
17         G = lab2::Matrix::I(x_k_1.size());
18         d = g_x * (-1);
19         return x_k_1;
20     }
21     const auto r = lab1::Brent(
22         [&f, x = x_k_1, d_ = d](double a) {
23             return f(x + d_ * a);
24         },
25         ONE_DIM_EPS, ONE_DIM_START, ONE_DIM_END)
26         .optimize();
27     s = d * r;
28     auto x = x_k_1 + s, g_y = g_x;
29     g_x = f.grad(x);
30     auto p = g_x - g_y, v = G * p;
31     G = G + lab2::Matrix::vector_mul(s, s) / (s * p)
32         - lab2::Matrix::vector_mul(v, v) / (v * p);
33     d = (G * g_x) * (-1);
34     return x;
35 }
36
37 bool DFP::is_done(lab2::NFunction &, double epsilon) const {
38     return s.norm() < epsilon;
39 }
```

Метод Пауэлла

```
1  #pragma once
2
3  #include "lab2/matrix.h"
4  #include "lab2/n_optimizer.h"
5
6  namespace lab4 {
7      class Powell : public lab2::NOptimizer {
8      public:
9          Powell();
10
11          lab2::Vector iteration(lab2::NFunction& f, double epsilon) override;
12          bool is_done(lab2::NFunction& f, double epsilon) const override;
13
14      private:
15          lab2::Vector s, d;
16          lab2::Matrix G;
17          const double ONE_DIM_EPS = 1e-7;
18          const int ONE_DIM_START = -100, ONE_DIM_END = 100;
19      };
20 } // namespace lab4
```

Метод Пауэлла

```
1  #include "lab4/powell.h"
2
3  #include "iostream"
4  #include "lab1/brent.h"
5  #include "lab3/solver.h"
6
7  using namespace lab4;
8
9  Powell::Powell()
10     : s(lab2::Vector({1})),
11       d(lab2::Vector({1})),
12       G(lab2::Matrix({{1}}, std::nullopt)) {}
13
14  lab2::Vector Powell::iteration(lab2::NFunction &f, double) {
15     lab2::Vector x_k_1 = get_points().back(), g_x = f.grad(x_k_1);
16     if (iteration_count == 0) {
17         G = lab2::Matrix::I(x_k_1.size());
18         d = g_x * (-1);
19         return x_k_1;
20     }
21     const auto r = lab1::Brent(
22         [&f, x = x_k_1, d_ = d](double a) {
23             return f(x + d_ * a);
24         },
25         ONE_DIM_EPS, ONE_DIM_START, ONE_DIM_END)
26         .optimize();
27     s = d * r;
28     auto x = x_k_1 + s, g_y = g_x;
29     g_x = f.grad(x);
30     auto p = g_x - g_y, v = G * p;
31     G = G + lab2::Matrix::vector_mul(s, s) / (s * p)
32         - lab2::Matrix::vector_mul(v, v) / (v * p);
33     d = (G * g_x) * (-1);
34     auto delta_x_wave = s + G * p;
35     G = G
36         - lab2::Matrix::vector_mul(delta_x_wave, delta_x_wave)
37         / (p * delta_x_wave);
38     return x;
39 }
40
41 bool Powell::is_done(lab2::NFunction &, double epsilon) const {
42     return s.norm() < epsilon;
43 }
```