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: M3237

Санкт-Петербург, 8 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 *(первый вариант)*.

2 Ход работы

Обозначение цветов на иллюстрациях:

- Зеленый — классический метод Ньютона.
- Голубой — метод Ньютона с одномерным поиском.
- Оранжевый — метод Ньютона с направлением спуска.

Во всех измерениях для одномерного поиска использовался метод Брента; для решения СЛАУ — метод Ньютона с выбором ведущего элемента.

2.1 Исследование на произвольных функциях

2.1.1 Условия

Исследуемые функции:

$$f_1 = 108x^2 + 116y^2 + 80xy + 43x + 33y - 211$$
$$f_2 = \sin x + \cos y + 0.3y^2 + 0.3x^2 + 0.1y$$

Целевая точность $\varepsilon = 10^{-5}$. Для других значений точности методы демонстрировали аналогичное поведение.
Начальная точка: $(0.1, 0.1)$.

2.1.2 Число итераций

Метод Ньютона

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

С одномерным поиском

Начальная точка	f_1	f_2
(0.1,0.1)	1	4
(1,1)	1	4
(2,2)	1	5
(-5,-5)	1	4
(10,10)	1	4

С направлением спуска

Начальная точка	f_1	f_2
(0.1,0.1)	1	3
(1,1)	1	3
(2,2)	1	4
(-5,-5)	1	3
(10,10)	1	4

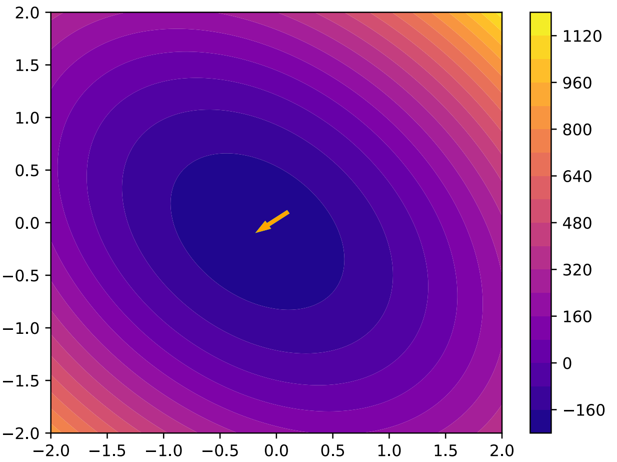
2.1.3 Найденные значения параметра α

α, полученные одномерным поиском для f_2

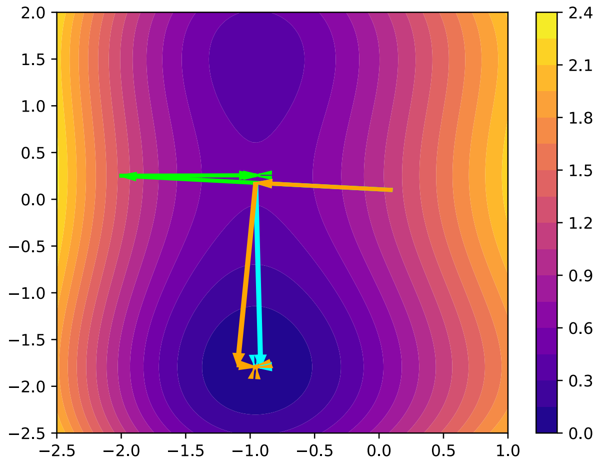
№	(0.1; 0.1)	(1; 1)	(2; 2)	(-5; -5)	(10; 10)
1	0.500975	-0.422133	-1.12804	0.157377	2.46763
2	-24.8939	-6.86726	6.07175	0.983785	0.72357
3	0.991506	1.03457	1.03309	1.11796	1.01044
4	1.00048	1.00347	1.01409	1.00082	1.00006
5	-	-	0.999954	-	-

Для f_1 все найденные $\alpha \approx 1$.

2.1.4 Траектории



f_1



f_2

2.1.5 Промежуточные выводы

- Если начальное приближение недостаточно близко к решению, то метод Ньютона может не сойтись к глобальному минимуму. Например, для f_2 метод Ньютона пришёл в седловую точку.
- Выбор начального приближения влияет на количество итераций методов.

- Так как матрица Гессе квадратичной функции положительно определена, все методы сходятся за одну итерацию, что было показано для функции f_1. Кроме того, траектории всех трёх методов в данном случае одинаковы и найденные одномерной оптимизацией $\alpha = 1$, т.е. модификации метода Ньютона эквивалентны исходному методу.

2.2 Исследование на заданных функциях

2.2.1 Условия

Исследуемые функции и начальные приближения:

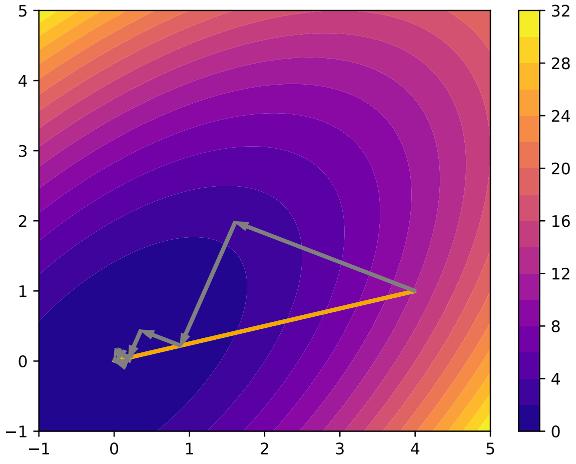
$$f_1 = x_1^2 + x_2^2 - 1.2x_1x_2,\ x^0 = (4, 1)^T$$
$$f_2 = 100(x_2 - x_1^2)^2 + (1 - x_1)^2,\ x^0 = (-1.2, 1)^T$$

2.2.2 Число итераций

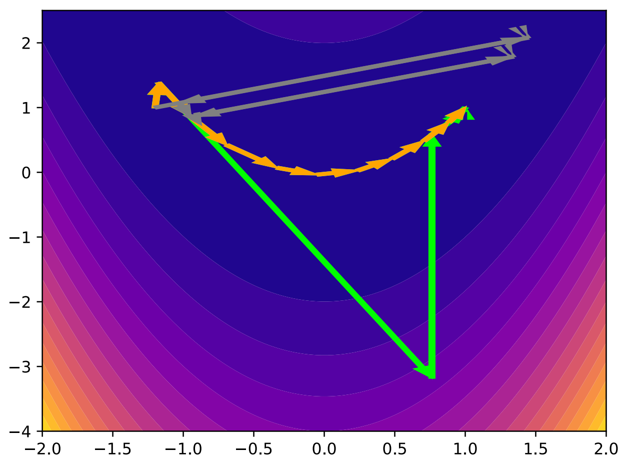
Метод	f_1	f_2
Классический	1	6
Одномерный поиск	1	12
С направлением спуска	1	12
Наискорейший спуск	10	1000

2.2.3 Траектории

Серым цветом обозначен метод наискорейшего спуска.



f_1



f_2

2.2.4 Промежуточные выводы

- Так как f_1 является квадратичной функцией, все вариации метода Ньютона сходятся за одну итерацию. Метод наискорейшего спуска таким свойством не обладает.
- Все методы работают гораздо медленнее на овражной функции f_2. Кроме того, метод наискорейшего спуска не сходится к точке минимума, т.к. f_2 не является квадратичной.

- По результатам измерений на данных функциях самый быстрый метод Ньютона — классический, но он демонстрирует нестабильное поведение *(см. траекторию)*, поэтому в следующем задании с квазиньютоновскими методами будет сравниваться метод с направлением спуска.

2.3 Квазиньютоновские методы

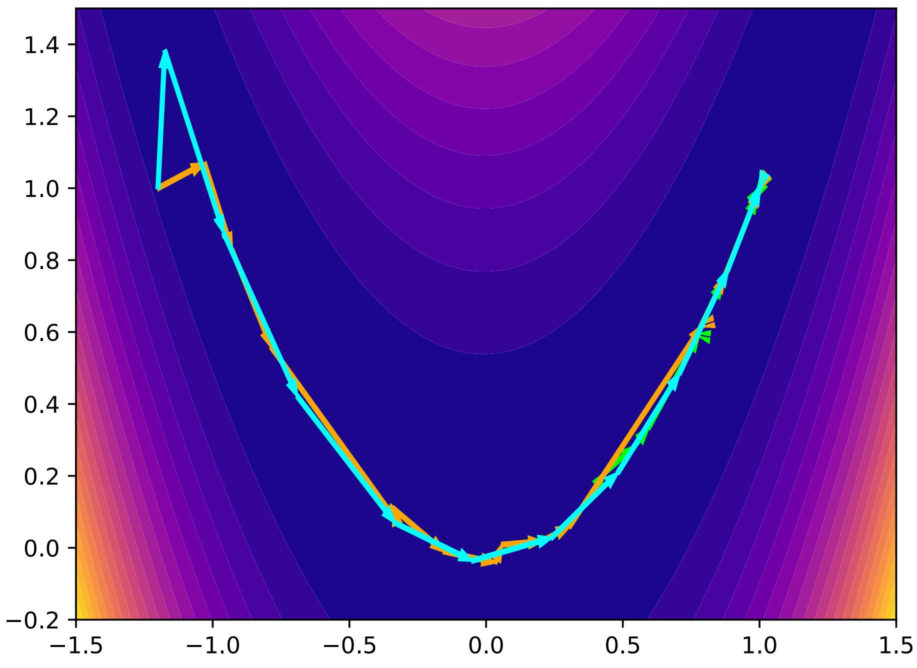
2.3.1 Условия

$$
\begin{aligned}
&f_1 = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \\
&f_2 = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2) \\
&f_3 = (x_1 + 10x_2)^2 + 5(x_3 - x_4)^2 + (x_2 - 2x_3)^4 + 10(x_1 - x_4)^4 \\
&f_4 = 100 - \frac{2}{1 + \left(\frac{x_1-1}{2}\right)^2 + \left(\frac{x_2-1}{3}\right)^2} - \frac{1}{1 + \left(\frac{x_1-2}{2}\right)^2 + \left(\frac{x_2-1}{3}\right)^2}
\end{aligned}
$$

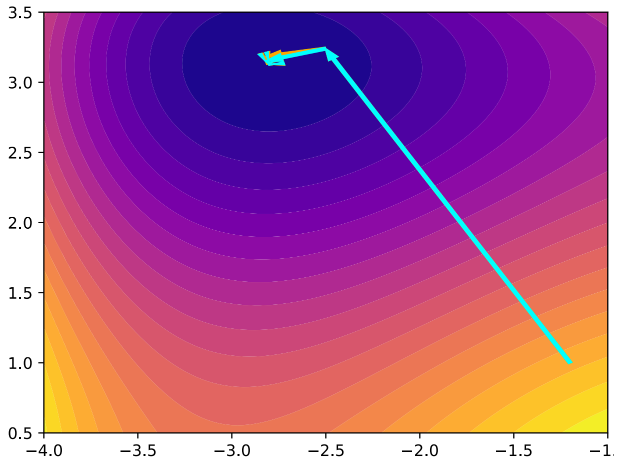
2.3.2 Траектории

Начальная точка $(-1.2, 1)$.
Цвета траекторий:

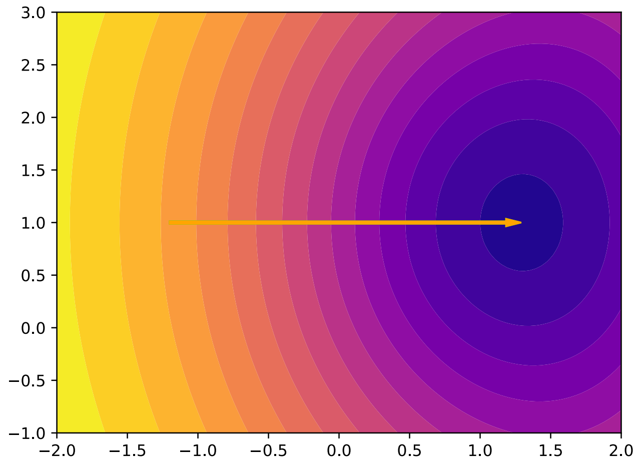
- Голубой — метод Ньютона с одномерным поиском.
- Оранжевый — метод Давидона-Флетчера-Пауэлла.
- Зеленый — метод Пауэлла.



f_1



f_2



f_3

2.3.3 Число итераций

Метод Ньютона с направлением спуска

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

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

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

2.3.4 Промежуточные выводы

- Все методы работают в несколько раз медленнее на функции с большим числом обусловленности и на функции большей размерности.
- Траектории всех методов схожи.
- 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, влияет на количество итераций методов.

3. Методы ДФП и Пауэлла работают примерно за одинаковое число итераций 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

```
#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:
        // all fns copy input :(
        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

```
#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>
#include "iostream"

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;
}
```

```cpp
#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 s, d;
        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()
    : s(lab2::Vector({1})),
      d(lab2::Vector({1})),
      G(lab2::Matrix({{1}}, std::nullopt)) {}

lab2::Vector DFP::iteration(lab2::NFunction &f, double) {
    lab2::Vector x_k_1 = get_points().back(), g_x = f.grad(x_k_1);
    if (iteration_count == 0) {
        G = lab2::Matrix::I(x_k_1.size());
        d = g_x * (-1);
        return x_k_1;
    }
    const auto r = lab1::Brent(
                       [&f, x = x_k_1, d_ = d](double a) {
                           return f(x + d_ * a);
                       },
                       ONE_DIM_EPS, ONE_DIM_START, ONE_DIM_END)
                       .optimize();
    s      = d * r;
    auto x = x_k_1 + s, g_y = g_x;
    g_x    = f.grad(x);
    auto p = g_x - g_y, v = G * p;
    G = G + lab2::Matrix::vector_mul(s, s) / (s * p)
        - lab2::Matrix::vector_mul(v, v) / (v * p);
    d = (G * g_x) * (-1);
    return x;
}

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

```cpp
#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 s, d;
        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()
    : s(lab2::Vector({1})),
      d(lab2::Vector({1})),
      G(lab2::Matrix({{1}}, std::nullopt)) {}

lab2::Vector DFP::iteration(lab2::NFunction &f, double) {
    lab2::Vector x_k_1 = get_points().back(), g_x = f.grad(x_k_1);
    if (iteration_count == 0) {
        G = lab2::Matrix::I(x_k_1.size());
        d = g_x * (-1);
        return x_k_1;
    }
    const auto r = lab1::Brent(
                       [&f, x = x_k_1, d_ = d](double a) {
                           return f(x + d_ * a);
                       },
                       ONE_DIM_EPS, ONE_DIM_START, ONE_DIM_END)
                       .optimize();
    s       = d * r;
    auto x = x_k_1 + s, g_y = g_x;
    g_x     = f.grad(x);
    auto p = g_x - g_y, v = G * p;
    G = G + lab2::Matrix::vector_mul(s, s) / (s * p)
        - lab2::Matrix::vector_mul(v, v) / (v * p);
    d = (G * g_x) * (-1);
    return x;
}

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

Метод Пауэлла

```cpp
#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 s, d;
        lab2::Matrix G;
        const double ONE_DIM_EPS = 1e-7;
        const int ONE_DIM_START = -100, ONE_DIM_END = 100;
    };
}  // namespace lab4
```

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

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

using namespace lab4;

Powell::Powell()
    : s(lab2::Vector({1})),
      d(lab2::Vector({1})),
      G(lab2::Matrix({{1}}, std::nullopt)) {}

lab2::Vector Powell::iteration(lab2::NFunction &f, double) {
    lab2::Vector x_k_1 = get_points().back(), g_x = f.grad(x_k_1);
    if (iteration_count == 0) {
        G = lab2::Matrix::I(x_k_1.size());
        d = g_x * (-1);
        return x_k_1;
    }
    const auto r = lab1::Brent(
                       [&f, x = x_k_1, d_ = d](double a) {
                           return f(x + d_ * a);
                       },
                       ONE_DIM_EPS, ONE_DIM_START, ONE_DIM_END)
                       .optimize();
    s      = d * r;
    auto x = x_k_1 + s, g_y = g_x;
    g_x    = f.grad(x);
    auto p = g_x - g_y, v = G * p;
    G = G + lab2::Matrix::vector_mul(s, s) / (s * p)
        - lab2::Matrix::vector_mul(v, v) / (v * p);
    d                 = (G * g_x) * (-1);
    auto delta_x_wave = s + G * p;
    G                 = G
        - lab2::Matrix::vector_mul(delta_x_wave, delta_x_wave)
              / (p * delta_x_wave);
    return x;
}

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