Национальный исследовательский университет ИТМО
Прикладная математика и информатика

Методы оптимизации

Отчет по лабораторной работе №2
“Алгоритмы минимизации многомерных функций”

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, 18 апреля 2021 г.

1 Цели

1. Реализовать алгоритмы минимизации функций:
   - Метод градиентного спуска
   - Метод наискорейшего спуска
   - Метод сопряженных градиентов
2. Исследовать скорость сходимости в зависимости от выбора алгоритма одномерной оптимизации в методе наискорейшего спуска.
3. Проанализировать траектории методов на различных квадратичных функциях
4. Исследовать зависимость числа итераций от следующих входных параметров:
   - Число обусловленности
   - Размерность пространства

2 Вычислительная схема методов

Обозначения:

- ε — искомая точность
- E_n — векторное пространство размерности n, в котором производится поиск точки минимума.
- f — оптимизируемая функция
- ∇f — градиент f
- $||\circ||$ — L_2 норма вектора, она же евклидово расстояние.
- $\langle \circ, \circ \rangle$ — векторное произведение

Для всех методов использовалось ограничение на 1000 итераций.

2.1 Метод градиентного спуска

Алгоритм метода:

1. Выбрать $\varepsilon > 0, \alpha > 0, x \in E_n$, вычислить $f(x)$.
2. Вычислить $\nabla f(x)$. Проверить условие $||\nabla f(x)|| < \varepsilon$. Если оно выполнено, то завершить процесс, иначе перейти к шагу 3.
3. Найти $y = x - \alpha \nabla f(x)$ и $f(y)$. Если $f(y) < f(x)$, принимаем $\alpha = \frac{\alpha}{2}$ и выполняем этот шаг ещё раз. Иначе положить $x = y, f(x) = f(y)$ и перейти к шагу 2.

2.2 Метод наискорейшего спуска

Алгоритм метода:

1. Выбрать $\varepsilon > 0, x \in E_n$, вычислить $f(x)$
2. Вычислить $\nabla f(x)$. Проверить условие $||\nabla f(x)|| < \varepsilon$. Если оно выполнено, то завершить процесс, иначе перейти к шагу 3.
3. Решить задачу одномерной оптимизации

$$\Phi(\alpha) \to \min \quad \Phi(\alpha) = f(x - \alpha \nabla f(x)), \alpha > 0$$

Положить $x = x - \alpha \nabla f(x)$, перейти к шагу 2.

2.3 Метод сопряженных градиентов

Алгоритм метода:

1. Выбрать $\varepsilon > 0, x \in E_n$, вычислить $\nabla f(x)$, задать $p = -\nabla f(x)$
2. Вычислить $\nabla f(x)$. Проверить условие $||\nabla f(x)|| < \varepsilon$. Если оно выполнено, то завершить процесс, иначе перейти к шагу 3.
3. Вычислить следующие значения:

$$t = Ap$$
$$\alpha = \frac{||\nabla f(x)||^2}{\langle t, p \rangle}$$
$$\tilde{x} = x + \alpha \cdot p$$
$$\nabla f(\tilde{x}) = \nabla f(x) + \alpha \cdot t$$
$$\beta = \frac{||\nabla f(\tilde{x})||^2}{||\nabla f(x)||^2}$$
$$p = -\nabla f(\tilde{x}) + p \cdot \beta$$

И принять $\tilde{x}$ за x, $\nabla f(\tilde{x})$ за $\nabla f(x)$

3 Ход работы

3.1 Скорость сходимости в зависимости от выбора алгоритма одномерной оптимизации

Следующие измерения проводились для функции $f = x^2 + y^2 - xy + 4x + 3y - 1$ с начальной точкой $(2, 2)$, искомой точностью $\varepsilon = 10^{-6}$.

Аналитическое решение задачи:

$$\begin{cases} f'_x = 2x - y + 4 \\ f'_y = 2y - x + 3 \end{cases}$$
$$\begin{cases} 0 = 2x - y + 4 \\ 0 = 2y - x + 3 \end{cases}$$
$$\begin{cases} x = -\frac{11}{3} \\ y = -\frac{10}{3} \end{cases}$$

1. Метод золотого сечения

№	x_1	x_2	α	Итераций одномерного поиска
0	2	2	0.666666	19
1	1.48785	1.57321	0.666666	19
2	0.982971	1.13785	0.666666	19
3	0.484755	0.69488	0.666666	19
4	-0.00744836	0.245237	0.666666	19
5	-0.494338	-0.210155	0.666666	19
6	-0.976651	-0.670391	0.666666	19
7	-1.45515	-1.13459	0.666666	19
8	-1.93063	-1.60188	0.666666	19
9	-2.4039	-2.07141	0.666666	19
10	-2.87578	-2.54234	0.666666	19
11	-3.34709	-3.01384	0.451891	19
12	-3.66675	-3.33325	0.000112783	19
13	-3.66667	-3.33333	8.85782e-07	19
14	-3.66667	-3.33333	0	0

2. Метод фибоначчи

№	x_1	x_2	α	Итераций одномерного поиска
0	2	2	0.666666	30
1	1.48785	1.57321	0.666666	30
2	0.98297	1.13785	0.666666	30
3	0.484754	0.694879	0.666666	30
4	-0.00745045	0.245235	0.666666	30
5	-0.494341	-0.210157	0.666666	30
6	-0.976654	-0.670394	0.666666	30
7	-1.45516	-1.13459	0.666666	30
8	-1.93064	-1.60188	0.666666	30
9	-2.40391	-2.07142	0.666666	30
10	-2.87578	-2.54235	0.666666	30
11	-3.3471	-3.01384	0.451883	30
12	-3.66675	-3.33325	0.000113101	30
13	-3.66667	-3.33333	8.85782e-07	0

3. Метод золотого сечения

№	x_1	x_2	α	Итераций одномерного поиска
0	2	2	0.666666	27
1	1.48785	1.57321	0.666666	27
2	0.982971	1.13785	0.666666	27
3	0.484755	0.69488	0.666666	27
4	-0.00744873	0.245237	0.666666	27
5	-0.494339	-0.210155	0.666666	27
6	-0.976651	-0.670392	0.666666	27
7	-1.45515	-1.13459	0.666666	27
8	-1.93063	-1.60188	0.666666	27
9	-2.4039	-2.07141	0.666666	27
10	-2.87578	-2.54234	0.666666	27
11	-3.34709	-3.01384	0.45189	27
12	-3.66675	-3.33325	0.000113103	27
13	-3.66667	-3.33333	8.85782e-07	0

4. Метод Брента

№	x_1	x_2	α	Итераций одномерного поиска
0	2	2	0.666666	14
1	1.48785	1.57321	0.666666	14
2	0.982971	1.13785	0.666666	14
3	0.484754	0.694879	0.666666	14
4	-0.00744959	0.245236	0.666666	14
5	-0.49434	-0.210156	0.666666	14
6	-0.976653	-0.670393	0.666666	14
7	-1.45515	-1.13459	0.666666	14
8	-1.93064	-1.60188	0.666666	14
9	-2.4039	-2.07142	0.666666	14
10	-2.87578	-2.54235	0.666666	14
11	-3.34709	-3.01384	0.451887	5
12	-3.66675	-3.33325	0.000113191	18
13	-3.66667	-3.33333	8.85782e-07	0

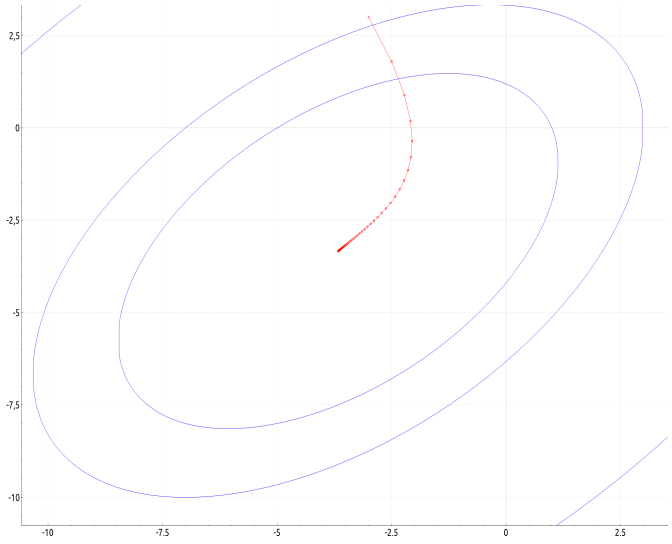
Таким образом, количество итераций метода наискорейшего спуска не зависит от выбора способа одномерного поиска, но это влияет на скорость работы. Метод парабол не работает для данной функции, т.к. при одномерной оптимизации используется функция с минимумом на границе отрезка допустимых решений. Такие функции не оптимизируются методом парабол, в силу невозможности найти начальное приближение $\tilde{x}$ такое, что $f(x_1) \leq f(\tilde{x})$, $f(x_2) \geq f(\tilde{x})$, $\tilde{x} \neq x_1$, $\tilde{x} \neq x_2$, где x_1, x_2 — границы отрезка, на котором производится оптимизация.

3.2 Траектории методов на различных функциях

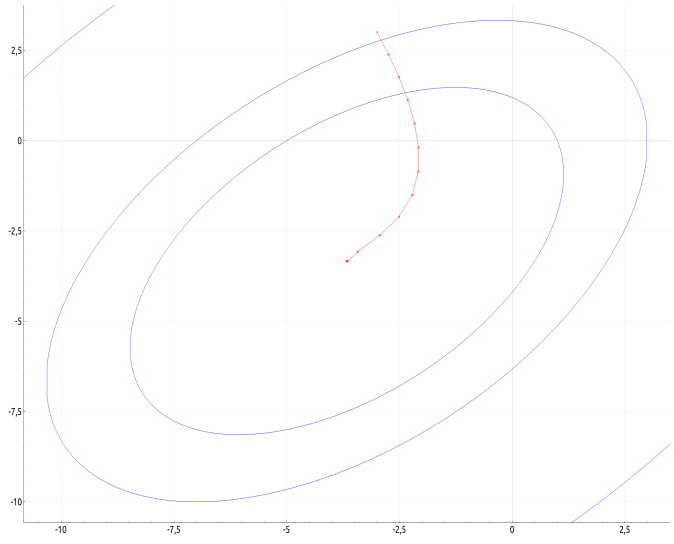
Во всех вычислениях для алгоритма наискорейшего спуска использовался метод Брента.
$\varepsilon = 10^{-6}$

1. $f_1 = x^2 + y^2 - xy + 4x + 3y - 1$
   Точка минимума (-3.66667 -3.33333), $\alpha = 0.5$ для метода градиентного спуска, начальная точка (-3, 3)

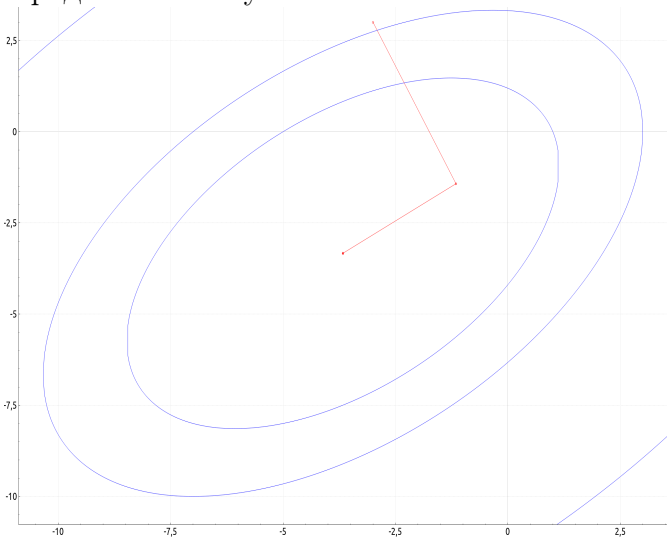
Метод	x_1	x_2	Количество итераций
Градиентный спуск	-3.66667	-3.33333	33
Наискорейший спуск	-3.66667	-3.33333	22
Сопряженные градиенты	-3.66667	-3.33333	3



Градиентный спуск



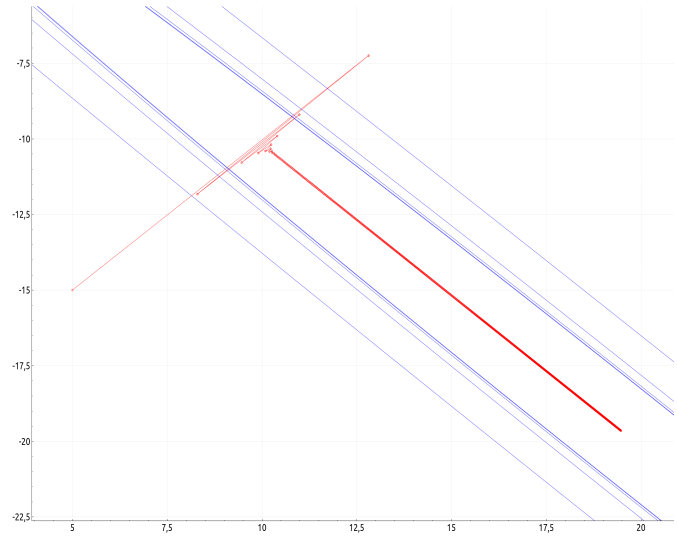
Наискорейший спуск



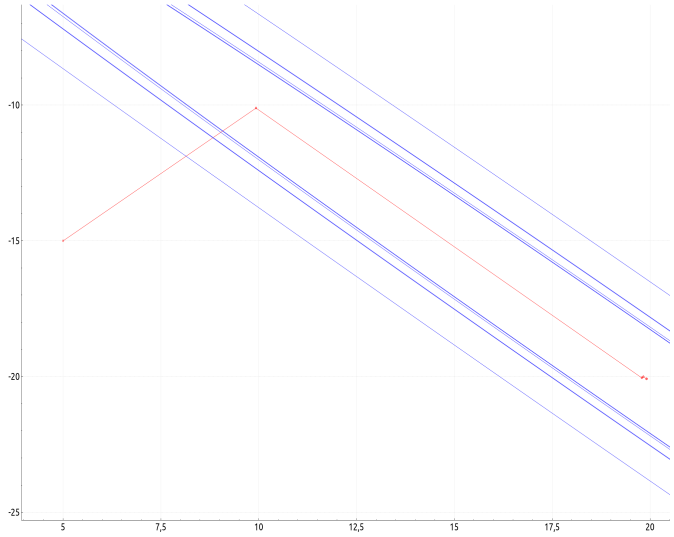
Сопряженные градиенты

2. $f_2 = 254x^2 + 506xy + 254y^2 + 50x + 130y - 111$
   Точка минимума (19.9112, -20.0888), $\alpha = 0.00196$, начальная точка (5, -15)

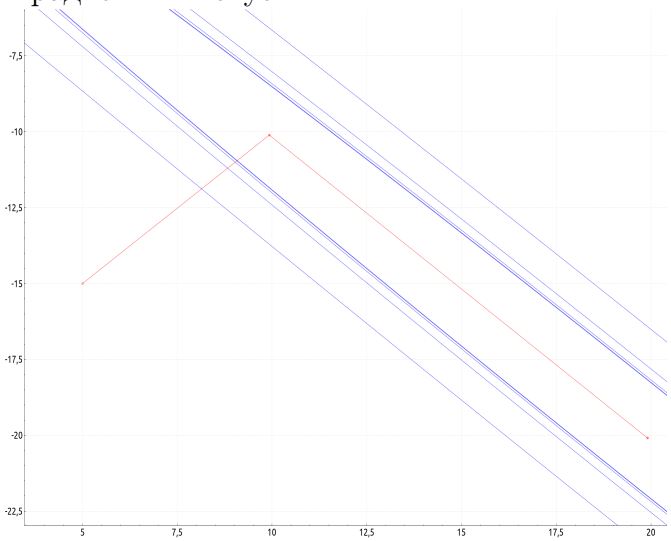
Метод	x_1	x_2	Количество итераций
Градиентный спуск	18.8385	-19.0161	1000
Наискорейший спуск	19.9112	-20.0888	9
Сопряженные градиенты	19.9112	-20.0888	3



Градиентный спуск



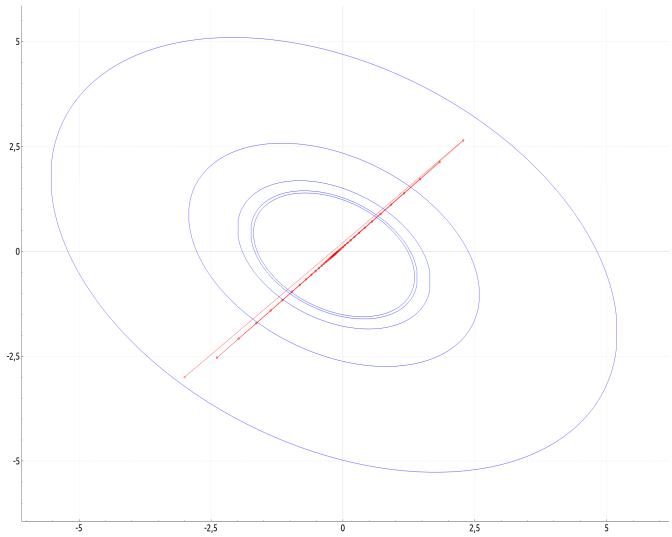
Наискорейший спуск



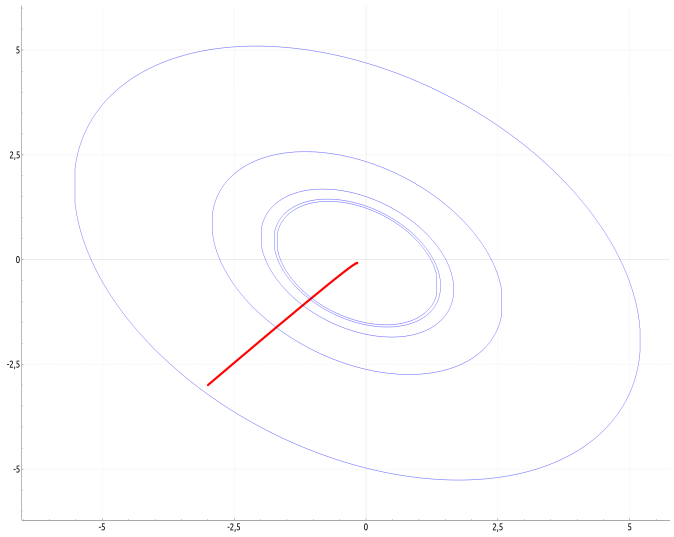
Сопряженные градиенты

3. $f_3 = 108x^2 + 116y^2 + 80xy + 43x + 33y - 211$
   Точка минимума (-0.195879, -0.13802), $\alpha = 0.00446$, начальная точка (-3, -3)

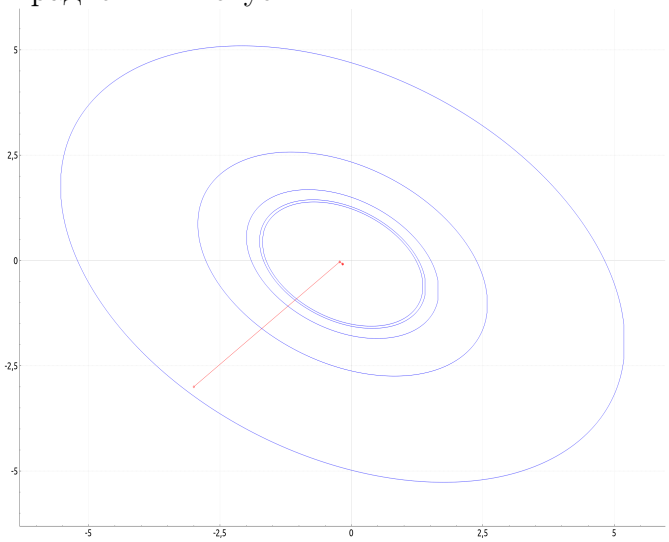
Метод	x_1	x_2	Количество итераций
Градиентный спуск	-0.167826	-0.0843708	157
Наискорейший спуск	-0.167826	-0.08437	626
Сопряженные градиенты	-0.167826	-0.0843704	3



Градиентный спуск



Наискорейший спуск

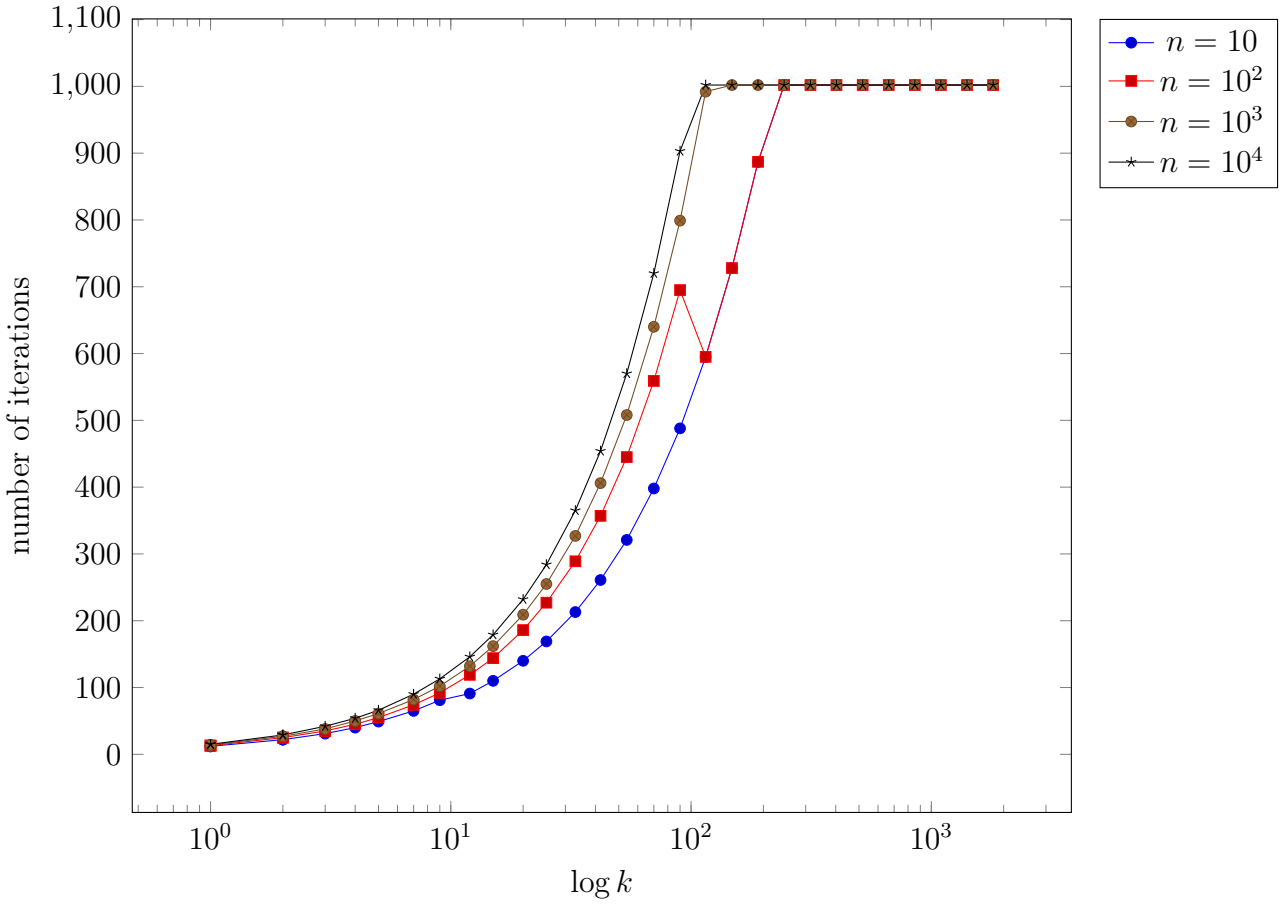


Сопряженные градиенты

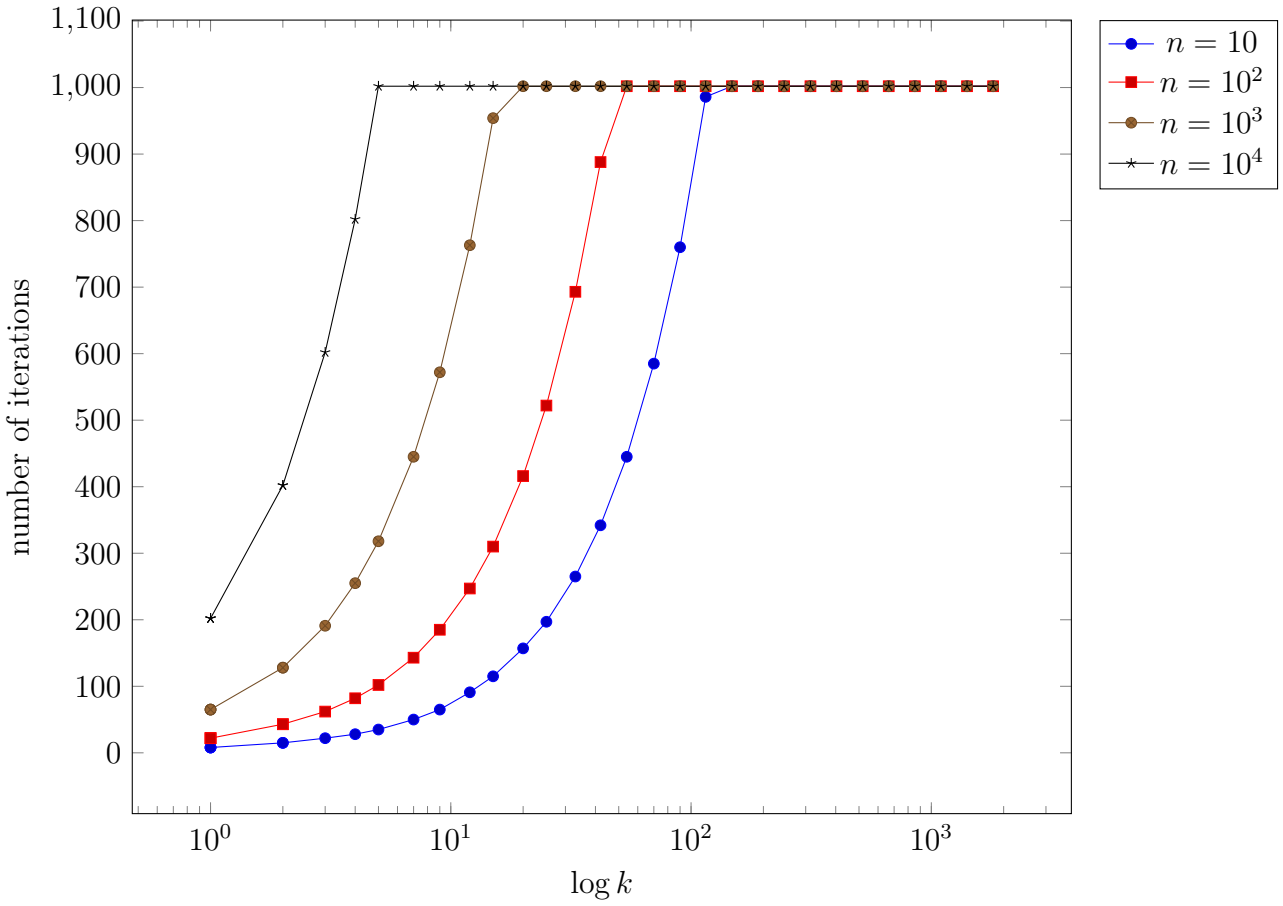
В общем случае работа каждого из методов на одной функции различается. Метод градиентного спуска подвержен зигзагообразным скачкам и не доходит до минимума из-за слишком маленьких шагов у функции с очень большим числом обусловленности. Метод сопряженных градиентов использует во много раз меньше итераций для произвольных функций, чем остальные.

3.3 Скорость сходимости в зависимости от числа обусловленности и размерности пространства

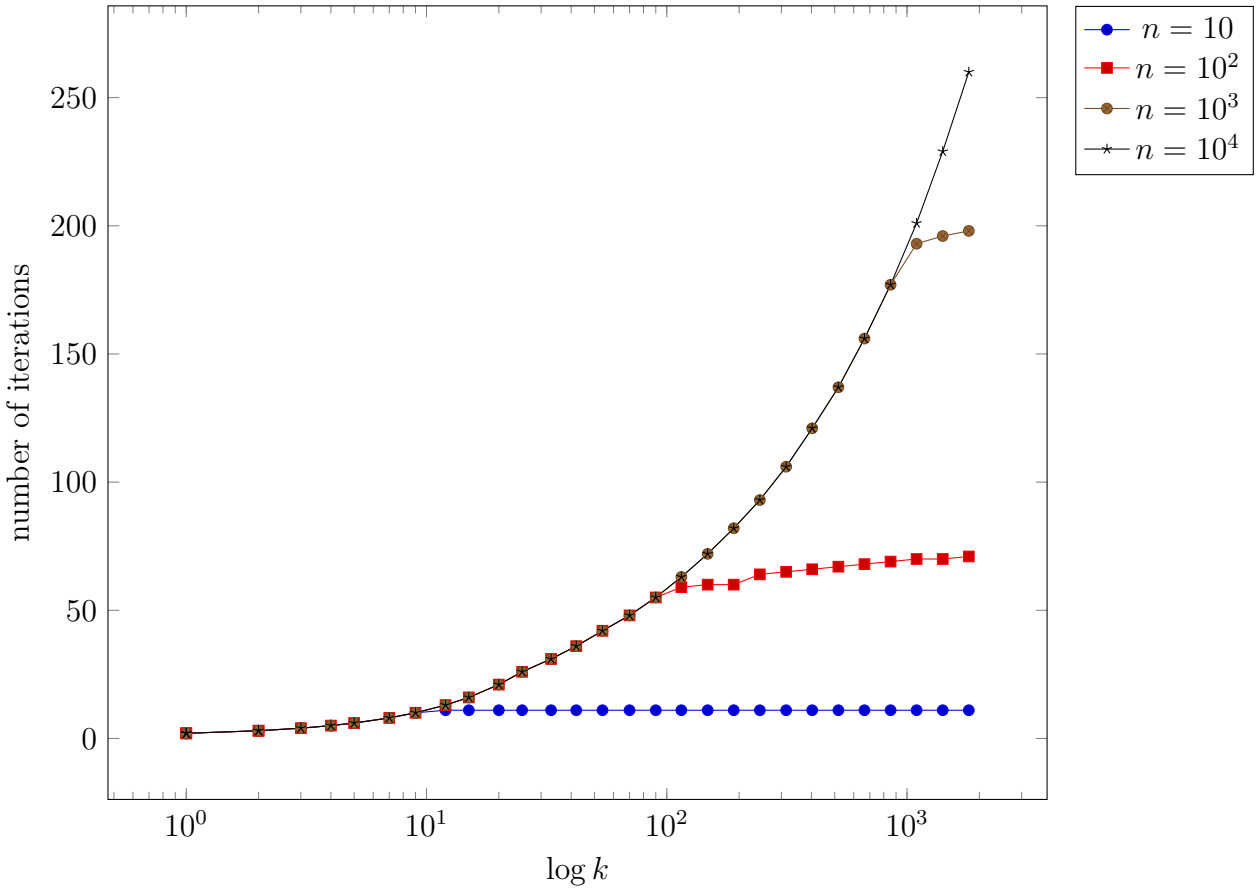
- Метод градиентного спуска



- Метод наискорейшего спуска



- Метод сопряженных градиентов



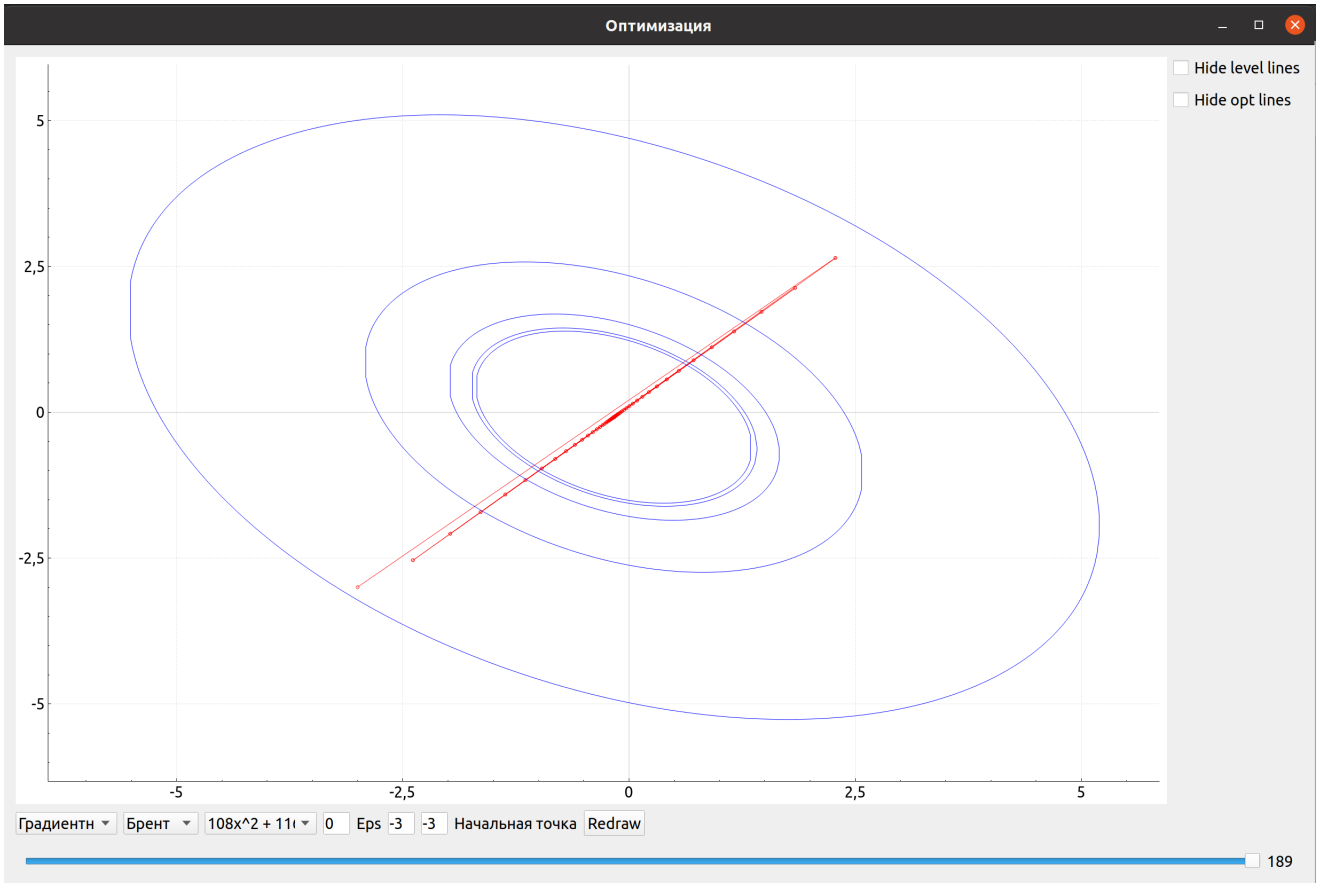
Скорость сходимости метода градиентного спуска и метода наискорейшего спуска зависит линейно от числа обусловленности k. При увеличении размерности пространства скорость сходимости обоих методов падает, но для метода наискорейшего спуска падает сильнее. Оба метода неспособны сойтись за 1000 итераций при $k > 300$ при всех рассмотренных размерностях.

Метод сопряженных градиентов также показывает линейную зависимость при $k < n$, но с меньшим коэффициентом пропорциональности. При $k > n$ зависимость становится сублогарифмической, близкой к константной. Этот метод сходится за < 1000 итераций при всех рассмотренных k и n.

4 Выводы

Метод сопряженных градиентов хорош, а все остальные не очень. Чего-то про сложность написания и оптимизации сопряженных.

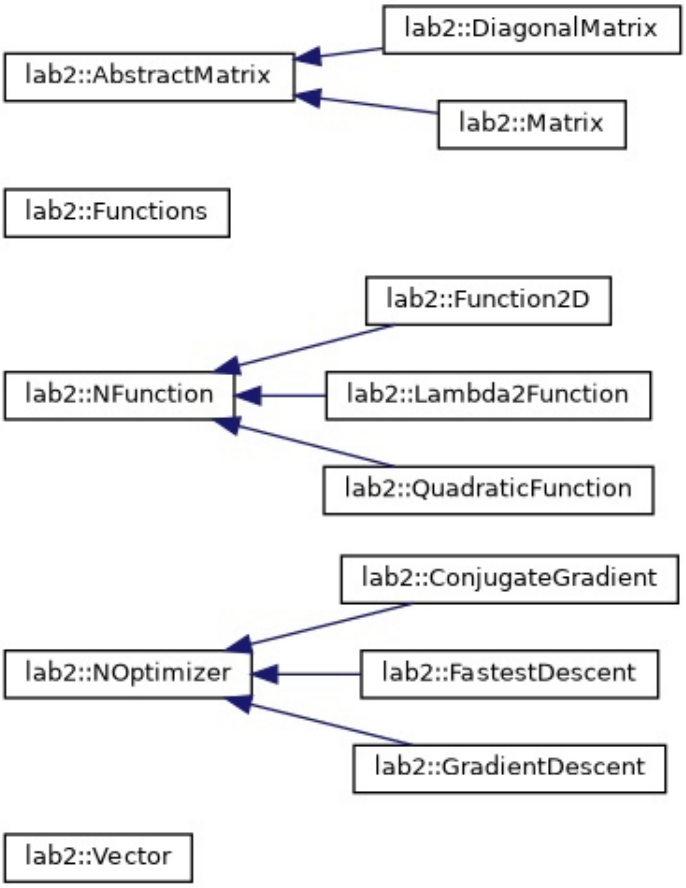
5 Графический интерфейс



Графический интерфейс позволяет выбрать метод оптимизации функций, способ одномерного поиска для наискорейшего спуска, одну из четырех предложенных квадратичных функций, а также указать начальное приближение и стартовую точку.
При передвижении ползунка внизу последовательно отображаются точки, вычисленные на каждой итерации.
Изображение можно масштабировать и передвигать, справа есть вариант скрыть линии уровня и прямые, соединяющие точки.

6 Исходный код

6.1 Диаграмма классов



6.2 Вектор

```cpp
#pragma once

#include <functional>
#include <vector>

namespace lab2 {
    /**
     * Вектор
     */
    class Vector {
      private:
        std::vector<double> data;

      public:
        Vector(const std::vector<double>& data);
        Vector(std::size_t size,
               const std::function<double(std::size_t)>& generator);

        double operator[](std::size_t idx) const;
        [[nodiscard]] std::size_t size() const;
        [[nodiscard]] double norm() const;

        Vector operator+(Vector other) const;
        Vector operator-(Vector other) const;
        double operator*(const Vector& other) const;
        Vector operator*(double val) const;
        bool operator==(const Vector& other) const;
    };

}  // namespace lab2
```

```cpp
#include "lab2/vector.h"

#include <cmath>
#include <utility>

using namespace lab2;

Vector::Vector(const std::vector<double>& data_) : data(data_) {
    if (size() == 0) {
        throw "Vector is empty";
    }
}

Vector::Vector(std::size_t size,
               const std::function<double(std::size_t)>& generator) {
    data.reserve(size);
    for (std::size_t i = 0; i < size; i++) {
        data.push_back(generator(i));
    }
}

std::size_t Vector::size() const { return data.size(); }

double Vector::operator[](std::size_t idx) const { return data[idx]; }

double Vector::norm() const { return std::sqrt((*this) * (*this)); }
```

```cpp

Vector Vector::operator+(Vector other) const {
    if (size() != other.size()) {
        throw "Size mismatch";
    }
    return Vector(size(), [this, &other](std::size_t i) {
        return (*this)[i] + other[i];
    });
}

Vector Vector::operator-(Vector other) const {
    if (size() != other.size()) {
        throw "Size mismatch";
    }
    return Vector(size(), [this, &other](std::size_t i) {
        return (*this)[i] - other[i];
    });
}

Vector Vector::operator*(double val) const {
    std::vector<double> tmp;
    return Vector(size(), [this, val](std::size_t i) {
        return (*this)[i] * val;
    });
}

double Vector::operator*(const Vector& other) const {
    if (size() != other.size()) {
        throw "Size mismatch";
    }
    double res = 0;
    for (std::size_t i = 0; i < size(); i++) {
        res += (*this)[i] * other[i];
    }
    return res;
}

bool Vector::operator==(const Vector& other) const {
    return data == other.data;
}
```

6.3 Общий класс матриц

```hpp
#pragma once

#include <optional>
#include <vector>

#include "lab2/vector.h"

namespace lab2 {
    /**
     * Абстрактный класс матрицы
     */
    class AbstractMatrix {
      public:
        [[nodiscard]] virtual Vector operator[](std::size_t idx) const = 0;
        [[nodiscard]] virtual ::size_t size() const                    = 0;

        // TODO: ref
        [[nodiscard]] virtual Vector operator*(Vector other) const = 0;

        const std::optional<double> max_eigenvalue;

        virtual ~AbstractMatrix() = default;

      protected:
        explicit AbstractMatrix(std::optional<double> max_eigenvalue);
    };

}  // namespace lab2
```

```cpp
#include "lab2/abstract_matrix.h"

using namespace lab2;

AbstractMatrix::AbstractMatrix(std::optional<double> max_eigenvalue)
    : max_eigenvalue(max_eigenvalue) {}
```

6.4 Матрица

```cpp
#pragma once

#include <optional>
#include <vector>

#include "lab2/abstract_matrix.h"
#include "lab2/vector.h"

namespace lab2 {
    /**
     * Произвольная матрица
     */
    class Matrix : public AbstractMatrix {
      public:
        explicit Matrix(const std::vector<std::vector<double>>& data,
                        std::optional<double> max_eigenvalue = std::nullopt);

        [[nodiscard]] Vector operator[](std::size_t idx) const override;
        [[nodiscard]] std::size_t size() const override;

        [[nodiscard]] Vector operator*(Vector other) const override;

        const std::optional<double> max_eigenvalue;

        ~Matrix() override = default;

      private:
        std::vector<Vector> data;

        void check_size() const;
    };

}  // namespace lab2
```

```cpp
#include "lab2/matrix.h"

using namespace lab2;

Matrix::Matrix(const std::vector<std::vector<double>>& data_,
               std::optional<double> max_eigenvalue)
    : AbstractMatrix(max_eigenvalue) {
    for (const auto& v : data_) {
        data.emplace_back(v);
    }
    check_size();
}

Vector Matrix::operator[](std::size_t idx) const { return data[idx]; }

std::size_t Matrix::size() const { return data.size(); }

Vector Matrix::operator*(Vector other) const {
    if (size() != other.size()) {
        throw "Size mismatch";
    }
    return Vector(size(), [this, &other](std::size_t i) {
        return (*this)[i] * other;
```

```
    });
}

void Matrix::check_size() const {
    if (size() == 0) {
        throw "Matrix is empty";
    }
    if (size() != (*data.begin()).size()
        || !std::equal(data.begin() + 1, data.end(), data.begin(),
                       [](const Vector& lhs, const Vector& rhs) {
                           return lhs.size() == rhs.size();
                       })) {
        throw "Matrix rows have different sizes";
    }
}
```

6.5 Диагональная матрица

```hpp
#pragma once

#include "lab2/abstract_matrix.h"

namespace lab2 {

    class DiagonalMatrix : public AbstractMatrix {
      public:
        explicit DiagonalMatrix(const std::vector<double>& diagonal,
                                std::optional<double> max_eigenvalue
                                = std::nullopt);

        [[nodiscard]] Vector operator[](std::size_t idx) const override;
        [[nodiscard]] std::size_t size() const override;

        [[nodiscard]] Vector operator*(Vector other) const override;

        ~DiagonalMatrix() override = default;

      private:
        const std::vector<double> diagonal;
    };

}  // namespace lab2
```

```cpp
#include "lab2/diagonal_matrix.h"

using namespace lab2;

DiagonalMatrix::DiagonalMatrix(const std::vector<double>& diagonal,
                               std::optional<double> max_eigenvalue)
    : AbstractMatrix(max_eigenvalue), diagonal(diagonal) {
    if (diagonal.size() == 0) {
        throw "Matrix is empty";
    }
}

Vector DiagonalMatrix::operator[](std::size_t idx) const {
    double val = diagonal[idx];
    return Vector(size(), [idx, val](std::size_t i) {
        if (i == idx) {
            return val;
        }
        return 0.;
    });
}

std::size_t DiagonalMatrix::size() const { return diagonal.size(); }

Vector DiagonalMatrix::operator*(Vector other) const {
    if (size() != other.size()) {
        throw "Size mismatch";
    }
    return Vector(size(), [this, &other](std::size_t idx) {
        return this->diagonal[idx] * other[idx];
    });
```

```
32  }
```

6.6 Функция

```hpp
#pragma once

#include "lab2/vector.h"

namespace lab2 {
    /**
     * Абстрактный класс N-мерной функции
     */
    class NFunction {
      public:
        /**
         * Вычисление функции
         * @param x Точка, в которой нужно вычислить функцию
         * @return Значение функции в точке x
         */
        [[nodiscard]] virtual double operator()(Vector x) = 0;
        /**
         * Вычисление градиента функции
         * @param x Точка, в которой нужно вычислить градиент функции
         * @return Значение градиента функции в точке x
         */
        [[nodiscard]] virtual Vector grad(Vector x) = 0;
        /**
         * @return Размерность функции
         */
        [[nodiscard]] virtual std::size_t get_dim() = 0;
        /**
         * @return Число вызовов функции
         */
        [[nodiscard]] std::size_t get_call_count() const;
        /**
         * @return Число вычислений градиента фукнции
         */
        [[nodiscard]] std::size_t get_grad_count() const;

      protected:
        /**
         * Увеличивает число вызовов функции на 1
         */
        void inc_call_count();
        /**
         * Увеличивает число вычислений градиента функции на 1
         */
        void inc_grad_count();

      private:
        std::size_t call_count = 0;
        std::size_t grad_count = 0;
    };
}  // namespace lab2
```

```hpp
#include "lab2/n_function.h"

using namespace lab2;

std::size_t NFunction::get_call_count() const { return call_count; }
std::size_t NFunction::get_grad_count() const { return grad_count; }
```

```
7
8   void NFunction::inc_call_count() { call_count++; }
9   void NFunction::inc_grad_count() { grad_count++; }
```

6.7 Квадратичная фунция

```h
#pragma once

#include <memory>

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

namespace lab2 {
    /**
     * Квадратичная функция вида f(x) = <A x, x> + <b, x> + c, где
     * <> обозначает векторное произведение
     */
    class QuadraticFunction : public NFunction {
      public:
        QuadraticFunction(AbstractMatrix* A, Vector&& b, double c);
        double operator()(Vector x) override;
        Vector grad(Vector x) override;
        std::size_t get_dim() override;

        std::unique_ptr<AbstractMatrix> A;
        const Vector b;
        const double c;
    };
}  // namespace lab2
```

```cpp
#include "lab2/quadratic_function.h"

using namespace lab2;

QuadraticFunction::QuadraticFunction(AbstractMatrix* A, Vector&& b, double c)
    : A(A), b(b), c(c) {
    if (A->size() != b.size()) {
        throw "Size mismatch";
    }
}

std::size_t QuadraticFunction::get_dim() { return A->size(); }

double QuadraticFunction::operator()(Vector x) {
    inc_call_count();
    return *A * x * x / 2 + b * x + c;
}

Vector QuadraticFunction::grad(Vector x) {
    inc_grad_count();
    return *A * x + b;
}
```

6.8 Абстрактный класс оптимизатора

```cpp
#pragma once

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

namespace lab2 {
    /**
     * Абстрактный класс оптимизатора N-мерных функций
     */
    enum class NOptimizers {
        CONJUGATE_GRADIENT,
        FASTEST_DESCENT,
        GRADIENT_DESCENT
    };

    static std::unordered_map<std::string, NOptimizers> const n_optimizers_table
        = {{"Градиентный спуск", NOptimizers::GRADIENT_DESCENT},
           {"Наискорейший спуск", NOptimizers::FASTEST_DESCENT},
           {"Сопряжённые градиенты", NOptimizers::CONJUGATE_GRADIENT}};

    class NOptimizer {
      public:
        /**
         * Процедура оптимизации.
         * @param f Оптимизируемая функция
         * @param starting_point Точка, из которой производится оптимизация
         * @param epsilon Искомая точность оптимизации
         * @return Минимум f
         */
        Vector optimize(QuadraticFunction& f, const Vector& starting_point,
                        double epsilon);
        /**
         * Возвращает вектор точек, рассмотренных в ходе оптимизации
         * @return Рассмотренные точки
         */
        [[nodiscard]] const std::vector<Vector>& get_points() const;

      protected:
        /**
         * @param f Оптимизируемая функция
         * @param epsilon Искомая точность
         * @return Достигнута ли искомая точность
         */
        [[nodiscard]] bool is_done(QuadraticFunction& f, double epsilon) const;
        /**
         * Итерация алгоритма оптимизации
         * @param f Оптимизируемая функция
         * @return Новая рассматриваемая точка
         */
        [[nodiscard]] virtual Vector iteration(QuadraticFunction& f,
                                               double epsilon)
            = 0;

        std::size_t iteration_count;

      private:
        /**
```

```
         * Вектор точек, рассмотренных в ходе оптимизации
         */
        std::vector<Vector> points;
    };

}  // namespace lab2
```

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

#include <lab2/quadratic_function.h>

using namespace lab2;

Vector NOptimizer::optimize(QuadraticFunction& f, const Vector& starting_point,
                            double epsilon) {
    iteration_count = 0;
    points.clear();
    points.push_back(starting_point);
    while (!is_done(f, epsilon) && iteration_count <= 1000) {
        points.push_back(iteration(f, epsilon));
        iteration_count++;
    }
    return points.back();
}

bool NOptimizer::is_done(QuadraticFunction& f, double epsilon) const {
    return f.grad(points.back()).norm() <= epsilon;
}

const std::vector<Vector>& NOptimizer::get_points() const { return points; }
```

6.9 Метод градиентного спуска

```cpp
#pragma once

#include "lab2/n_optimizer.h"

namespace lab2 {
    class GradientDescent : public NOptimizer {
      public:
        explicit GradientDescent(double step) : a(step) {}
        Vector iteration(QuadraticFunction &f, double epsilon) override;

      private:
        [[nodiscard]] Vector points_last() const;
        double a;
    };
}  // namespace lab2
```

```cpp
#include "lab2/gradient_descent.h"

using namespace lab2;

Vector GradientDescent::iteration(QuadraticFunction &f, double) {
    Vector x        = points_last();
    Vector f_x_grad = f.grad(x);
    Vector y        = x - f_x_grad * a;
    double f_x      = f(x);
    while (f(y) >= f_x && iteration_count <= 1000) {
        a /= 2;
        y = x - f_x_grad * a;
        iteration_count++;
    }
    return y;
}

lab2::Vector GradientDescent::points_last() const {
    return get_points().back();
}
```

6.10 Метод наискорейшего спуска

```cpp
#pragma once

#include <memory>

#include "lab1/optimizer.h"
#include "lab2/n_optimizer.h"

namespace lab2 {

    class FastestDescent : public NOptimizer {
        using Gen = std::function<std::unique_ptr<lab1::Optimizer>(
            const std::function<double(double)> &, double, double, double)>;

      public:
        FastestDescent(Gen generator);

        Vector iteration(QuadraticFunction &f, double epsilon) override;

      private:
        Gen generator;
    };

}  // namespace lab2
```

```cpp
#include "lab2/fastest_descent.h"

#include <utility>

lab2::FastestDescent::FastestDescent(
    std::function<std::unique_ptr<lab1::Optimizer>(
        const std::function<double(double)> &, double, double, double)>
        generator)
    : generator(std::move(generator)) {}

lab2::Vector lab2::FastestDescent::iteration(QuadraticFunction &f, double) {
    Vector x_k      = get_points().back();
    Vector grad_x_k = f.grad(x_k);
    grad_x_k        = grad_x_k * (1 / grad_x_k.norm());
    double alpha
        = generator(
              [&f, &x_k, &grad_x_k](double alpha) {
                  return f(x_k - grad_x_k * alpha);
              },
              1e-6, 0,
              f.A->max_eigenvalue.has_value() ? 2. / *f.A->max_eigenvalue
                                              : 1000)
              ->optimize();
    return x_k - grad_x_k * alpha;
}
```

6.11 Метод сопряженных градиентов

```hpp
#include "lab2/n_optimizer.h"

namespace lab2 {
    class ConjugateGradient : public NOptimizer {
      public:
        Vector iteration(QuadraticFunction &f, double epsilon) override;

      private:
        Vector p      = Vector({1, 1});
        Vector f_grad = p;
    };
}  // namespace lab2
```

```cpp
#include "lab2/conjugate_gradient.h"

#include <cmath>

using namespace lab2;
Vector ConjugateGradient::iteration(QuadraticFunction& f, double) {
    Vector x = get_points().back();
    if (iteration_count == 0) {
        f_grad = f.grad(x);
        p      = f_grad * (-1);
    }
    Vector t        = *(f.A) * p;
    double alpha    = std::pow(f_grad.norm(), 2) / (t * p);
    Vector new_x    = x + p * alpha;
    Vector new_grad = f_grad + t * alpha;
    double beta     = std::pow(new_grad.norm(), 2) / std::pow(f_grad.norm(), 2);
    p               = new_grad * (-1) + p * beta;
    f_grad          = new_grad;
    return new_x;
}
```

6.12 Используемые функции

```cpp
#pragma once

#include "lab2/n_function.h"
#include "lab2/quadratic_function.h"

namespace lab2 {
    class Functions {
      public:
        Functions() = delete;

        /**
         * Параболоид x^2 + y^2 + 1
         */
        static QuadraticFunction paraboloid;
        /**
         * x^2 + y^2 - xy + 4x + 3y - 1
         */
        static QuadraticFunction f1;

        /**
         * 254x^2 + 506xy + 254y^2 + 50x + 130y - 111
         */
        static QuadraticFunction f2;

        /**
         * 108x^2 + 116y^2 + 80xy + 43x + 33y - 211
         */
        static QuadraticFunction f3;
    };

    static std::unordered_map<std::string, QuadraticFunction*> const
        functions_table
        = {{"x^2 + y^2 + 1", &Functions::paraboloid},
           {"x^2 + y^2 - xy + 4x + 3y - 1", &Functions::f1},
           {"254x^2 + 506xy + 254y^2 + 50x + 130y - 111", &Functions::f2},
           {"108x^2 + 116y^2 + 80xy + 43x + 33y - 211", &Functions::f3}};
}  // namespace lab2
```

```cpp
#include <cmath>
#include "lab2/functions.h"

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

using namespace lab2;

QuadraticFunction Functions::paraboloid = QuadraticFunction(
    new Matrix({{2, 0}, {0, 2}}, 2), Vector(std::vector<double>{0, 0}), 1);

QuadraticFunction Functions::f1
    = QuadraticFunction(new Matrix({{2, -1}, {-1, 2}}, 3), Vector({4, 3}), -1);

QuadraticFunction Functions::f2
        = QuadraticFunction(new Matrix({{508, 506}, {506, 508}}),
                            Vector({50, 130}), -111);

QuadraticFunction Functions::f3
```

```
20          = QuadraticFunction(new Matrix({{216, 80}, {80, 232}}, 8*(28 + sqrt(101))),
21                              Vector({43, 33}), -211);
```