Методы оптимизации
Лабораторная работа №2
Изучение градиентных методов оптимизации многомерных функций на примере квардатичных функций

Гранкин Максим M3237 Назаров Георгий Панов Иван M3239

Университет ИТМО, КТ.
Четвертый семестр. апрель 2021

1 Постановка задачи

1. Реализовать алгоритмы:

   - метод градиентного спуска;
   - метод наискорейшего спуска;
   - метод сопряженных градиентов.

   Оценить, как меняется скорость сходимости, если для поиска величины шага использовать различные методы одномерного поиска.

2. Проанализировать траектории реализованных методов.

3. Найти зависимость числа итераций, необходимое методам для сходимости, от числа обусловленности $k \geq 1$ оптимизируемой функции и от размерности простанства оптимизируеммых переменных $\mathbb{R}^n$.

2 Цели работы

- Реализовать градиентные методы;
- Проанализировать траектории методов для нескольких квадратичных функций;
- Исследовать, как зависит число итераций, необходимое методам для сходимости, от числа обусловленности оптимизируемой функции и размерности пространства n оптимизируемых переменных;

3 Введение

3.1 Постановка задачи

Пусть $U \subset \mathbb{R}^n$, $f : U \longrightarrow \mathbb{R}$. Тогда f — минимизируемая на U функция. Алгоритм минимизации должен выдавать на выходе пару $(x^*, f(x^*))$, где $f(x^*)$ — минимум функции f (не обязательно точный) на множестве U, а x^* — аргумент, при котором он достигается.

Градиентом $\nabla f(x)$ в точке x функции f называется вектор-столбец частных производных 1-ого порядка в этой точке. Градиент направлен по нормали к поверхности уровня (перпендикулярно к касательной плоскости в точке x)

Градиентные методы минимизации — методы минимизации, использующие градиент функции, как вспомогательную информацию.

В дальнейшем, если не оговорено иное, считаем, что функция f — квадратичная, заданная следующим образом:

$$f(x) = \frac{1}{2}\langle Ax, x\rangle + \langle b, x\rangle + c$$

Здесь A — симметричная матрица коэфициентов при $x_i \cdot x_j$, b — вектор-столбец при x_i, $c \in \mathbb{R}$.

3.2 Общая вычислительная схема многомерной оптимизации

Пусть f — минимизуемая на $\mathbb{R}^n$ функция. Задается начальное приближение $x^{(0)} \in \mathbb{R}^n$. На каждом шаге (k — номер итерации) алгоритма вычисляется

$$x^{(k+1)} = \Phi(x^{(k)}, x^{(k-1)}, ..., x^{(0)})$$

Возможные критерии остановки итерационного процесса:

- $\rho(x^{(k+1)}, x^{(k)}) < \varepsilon_1$ — шаг стал слишком мал;
- $|f(x^{(k+1)}) - f(x^{(k)})| < \varepsilon_2$ — значения функции на очередном шаге меняются слишком мало;
- $\left\|\nabla f(x^{(k)})\right\| < \varepsilon_3$ (для градиентных методов) — градиент функции стал слишком мал.

3.2.1 Итерационный процесс

В рассматриваемых нами методах запись итерационной процедуры можно упростить (определив достаточно простую Φ):

$$x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)}$$

Здесь α_k — величина шага, а $p^{(k)}$ — направление поиска.
Для выбора коэфициентов α_k следует воспользоваться условием $f(x^{(k+1)}) < f(x^{(k)})$.

3.2.2 Исчерпывающий спуск

В итерационном процессе производится **исчерпывающий спуск**, если коэфициенты α_k находятся из решения задачи одномерной минимизации следующей функции:

$$\Phi_k(\alpha_k) = f(x^{(k)} + \alpha_k p^{(k)})$$

4 Метод градиентного спуска

4.1 Вычислительная схема

- Задаются:
  - $\varepsilon > 0$ — допустимый порог нормы градиента для остановки метода;
  - $\alpha > 0$ — величина шага на каждой итерации. При положительно определенной матрице A и $\alpha < \frac{2}{L}$, где L — наибольшее собственное значение матрицы A, метод гарантированно сходится к единственной точке глобального минимума, причем линейно. В таком случае, наиболее теоретически оптимальное по количеству шагов значение $\alpha = \frac{2}{L+l}$, где l — наименьшее собственное число матрицы A.
  - $x^{(0)} \in \mathbb{R}^n$ — начальное приближение. Вычисляется $f(x^{(0)})$.
- Вычисляется $\nabla f(x^{(k)})$. Если $\left\|\nabla f(x^{(k)})\right\| < \varepsilon$, то метод завершается.
-
  - $y = x^{(k)} - \alpha \nabla f(x^{(k)})$, $f(y)$
  - Если $f(y) >= f(x^{(k)})$, повторить с $\alpha := \frac{\alpha}{2}$
  - $x^{(k+1)} = y$, $f(x^{(k+1)}) = f(y)$

5 Метод наискорейшего спуска

5.1 Вычислительная схема

Схема вычислений похожа на метод градиентного спуска, за исключением того, что α выбирается как ответ на задачу одномерной оптимизации вдоль антиградиента.

- Задаются:
  - $\varepsilon > 0$ — допустимый порог нормы градиента для остановки метода;
  - $x^{(0)} \in \mathbb{R}^n$ — начальное приближение. Вычисляется $f(x^{(0)})$.
- Вычисляется $\nabla f(x^{(k)})$. Если $\left\|\nabla f(x^{(k)})\right\| < \varepsilon$, то метод завершается.
- Решается задача одномерной оптимизации относительно α_k:
  - $\Phi_k(\alpha_k) = f(x^{(k)} - \alpha_k \nabla f(x^{(k)}))$
  - α^* — минимум функции Φ_k.
- $x^{(k+1)} = x^{(k)} - \alpha^* \nabla f(x^{(k)})$

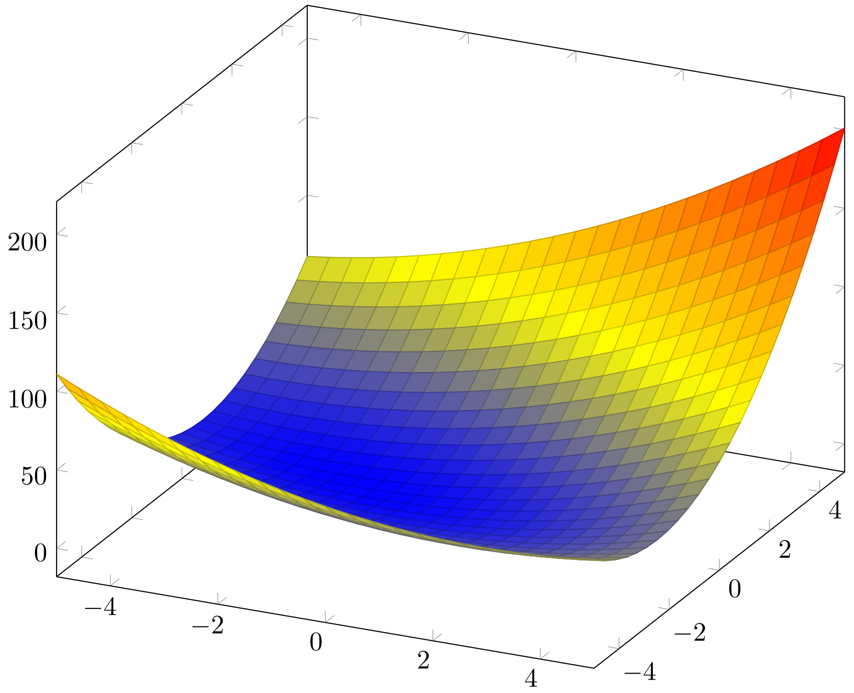
6 Метод сопряженных градиентов

6.1 Вычислительная схема

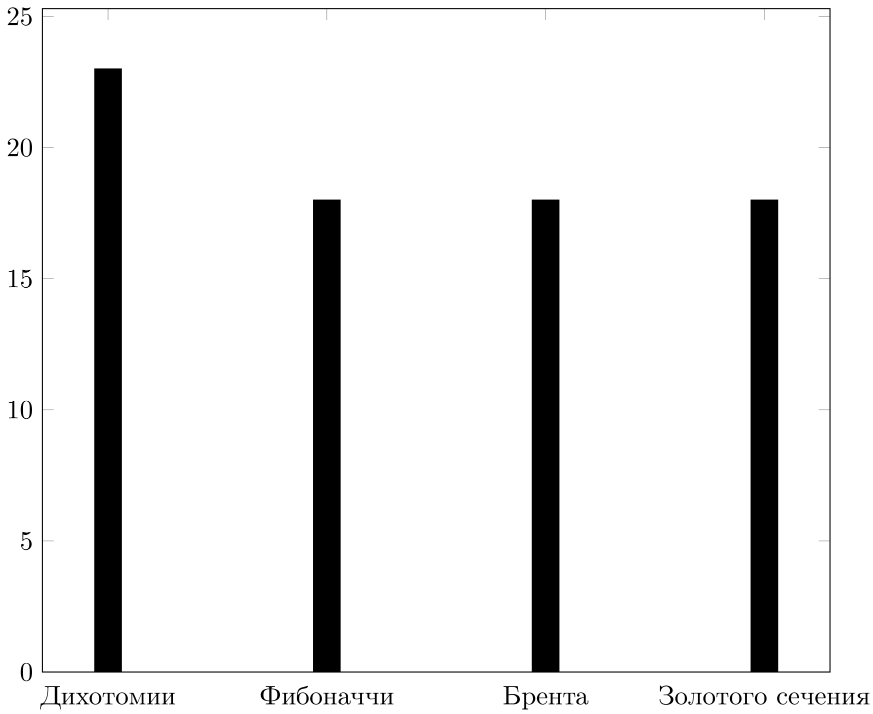
- Задаются $x^{(0)}$ — начальное приближение и $p^{(0)} = -\nabla f(x^{(0)})$ — начальное направление поиска;
- На каждой итерации:
  - $x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)}$
  - $p^{(k+1)} = -\nabla f(x^{(k+1)}) + \beta_k p^{(k)}$
  - $$\alpha_k = \frac{\left\|\nabla f(x^{(k)})\right\|^2}{\langle Ap^{(k)}, p^{(k)}\rangle}$$
  - β_k выбирается так, чтобы последовательность векторов $p^{(0)}, p^{(1)}, \ldots$ была A-ортогональной.
  - $$\beta_k = \frac{\langle A\nabla f(x^{k+1}), p^{(k)}\rangle}{\langle Ap^{(k)}, p^{(k)}\rangle} = \frac{\left\|\nabla f(x^{(k+1)})\right\|^2}{\left\|\nabla f(x^{(k)})\right\|^2}$$
- Для квадратичных функций число итераций метода можно ограничить сверху числом n, в дополнение к критериям остановки, описанным выше.

7 Оценка скорости сходимости при использовании различных методов одномерного поиска

Функция для подсчета: $f(x, y) = x^2 + 2 \cdot x \cdot y + 3 \cdot y^2 + 4 \cdot x + 5 \cdot y + 6$



Минимум функции $\min f(x, y) = 1.875$ в точке $(x, y) = (-1.75, -0.25)$
Выбран $\epsilon = 10^{-5}$ **для всех методов**
Для метода дихотомии выбрано $\delta = 10^{-6}$



Результаты: метод парабол не сошелся.
Остальные методы показали примерно одинаковые результаты.

8 Анализ траекторий методов

Выбранные для анализа функции:

- $f(x,y) = x^2 + y^2$;
- $f(x,y) = x^2 + 2xy + 3y^2 + 4x + 5y + 6$;
- $f(x,y) = 64x^2 + 126xy + 64y^2 - 10x + 30y + 13$;

8.1 Аналитическое решение

Для функции $f(x,y) = x^2 + y^2$, $x^* = (0,0)$, $f(x^*) = 0$.

Для функции $f(x,y) = x^2 + 2xy + 3y^2 + 4x + 5y + 6$:

$$f'_x = 2x + 2y + 4$$
$$f'_y = 2x + 6y + 5$$

$$\begin{cases} f'_x = 0 \\ f'_y = 0 \end{cases} \quad \begin{cases} 2x + 2y + 4 = 0 \\ 2x + 6y + 5 = 0 \end{cases} \quad \begin{cases} x^* = -\frac{7}{4} = -1.75 \\ y^* = -\frac{1}{4} = -0.25 \end{cases}$$

То есть, $(-1.75, -0.25)$ — критическая точка. Проверим наличие экстремума через гессиан:

$$f''_{xx} \cdot f''_{yy} - f''_{xy} \cdot f''_{yx} = 2 \cdot 6 - 2 \cdot 2 > 0$$

Таким образом, $(-1.75, -0.25)$ **— точка минимума функции** f**.**

Для функции $f(x,y) = 64x^2 + 126xy + 64y^2 - 10x + 30y + 13$:

$$f'_x = 128x + 126y - 10$$

$$f'_y = 126x + 128y + 30$$

$$\begin{cases} f'_x = 0 \\ f'_y = 0 \end{cases} \quad \begin{cases} 128x + 126y - 10 = 0 \\ 126x + 128y + 30 = 0 \end{cases} \quad \begin{cases} x = -\frac{1265}{127} \approx 9.96062992 \\ y = -\frac{1275}{127} \approx 10.03937008 \end{cases}$$

То есть, $(-\frac{1265}{127}, -\frac{1275}{127})$ — критическая точка. Проверим наличие экстремума через гессиан:

$$f''_{xx} \cdot f''_{yy} - f''_{xy} \cdot f''_{yx} = 128 \cdot 128 - 126 \cdot 126 > 0$$

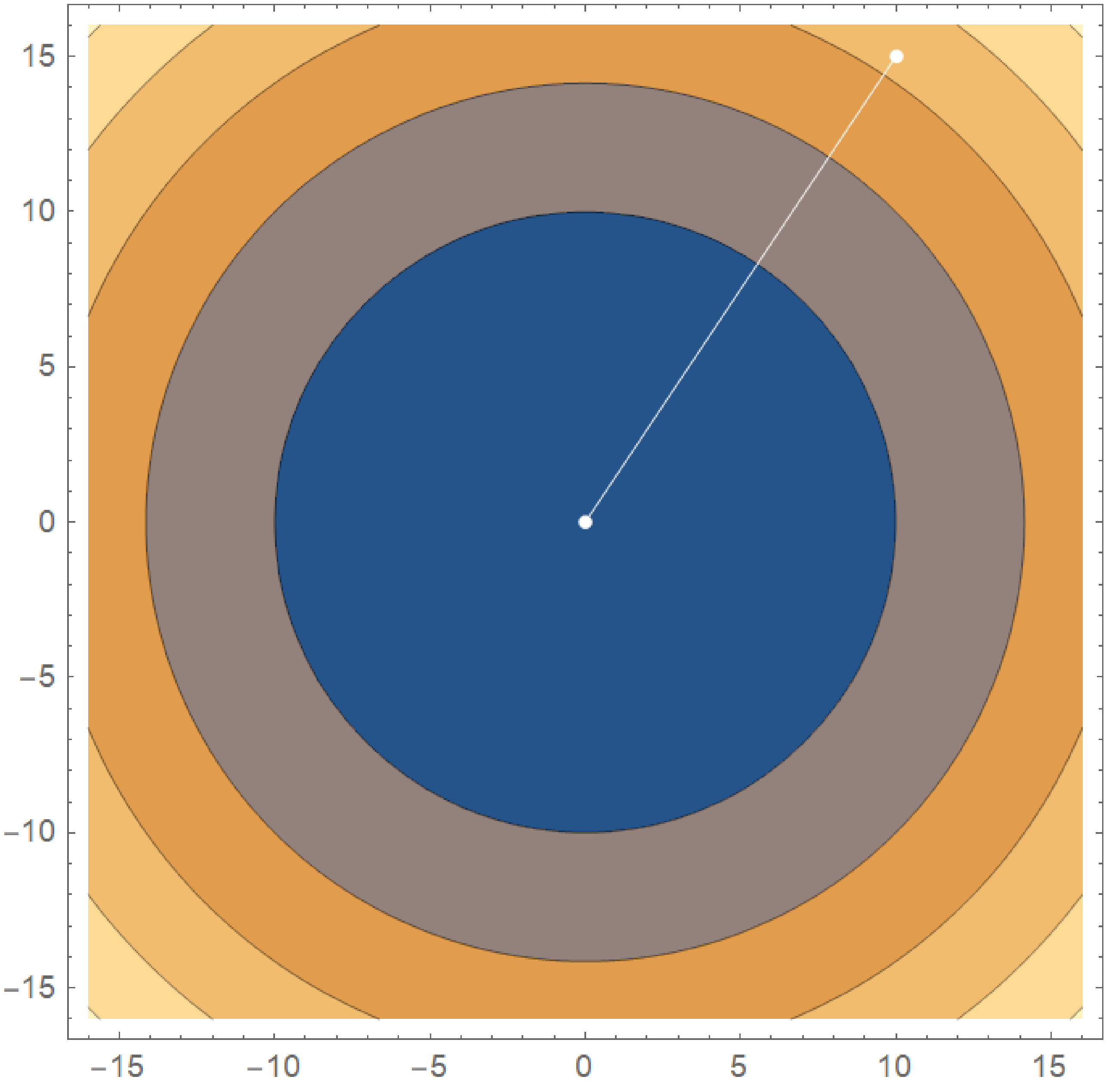
Таким образом, $(-\frac{1265}{127}, -\frac{1275}{127})$ **— точка минимума функции** f**.**

8.2 Метод градиентного спуска

Во всех функциях начальная точка: $(x, y) = (10.0, 15.0)$

Функция 1: $x^2 + y^2$

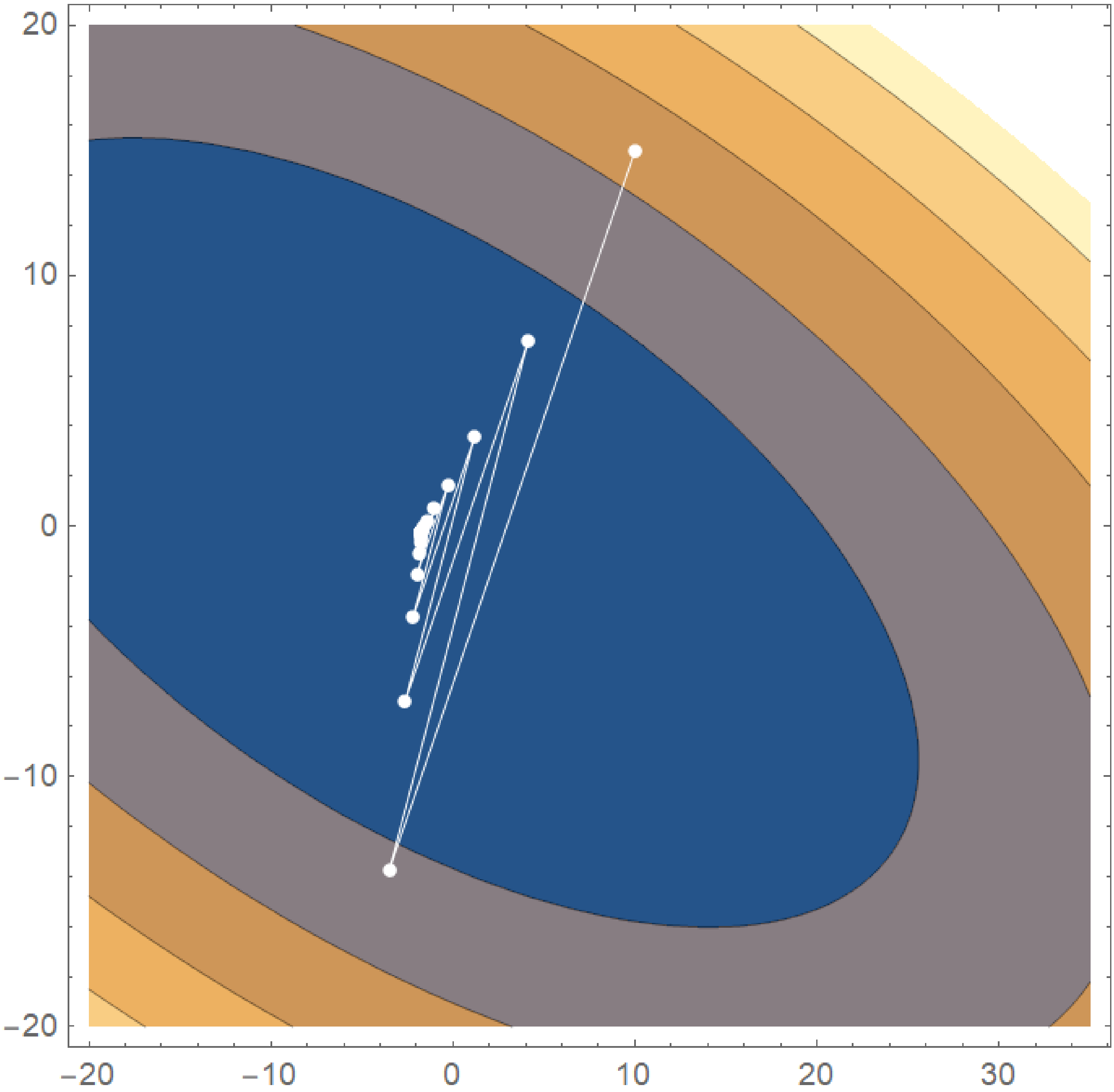
$\alpha = 1$, $\varepsilon = 10^{-5}$



Метод нашёл точку $x^* = (0, 0)$ за 1 шаг.
В данном случае метод сразу находит точку минимума.

Функция 2: $x^2 + 2 \cdot x \cdot y + 3 \cdot y^2 + 4 \cdot x + 5 \cdot y + 6$

$\alpha = \frac{1}{4}$, $\varepsilon = 10^{-5}$

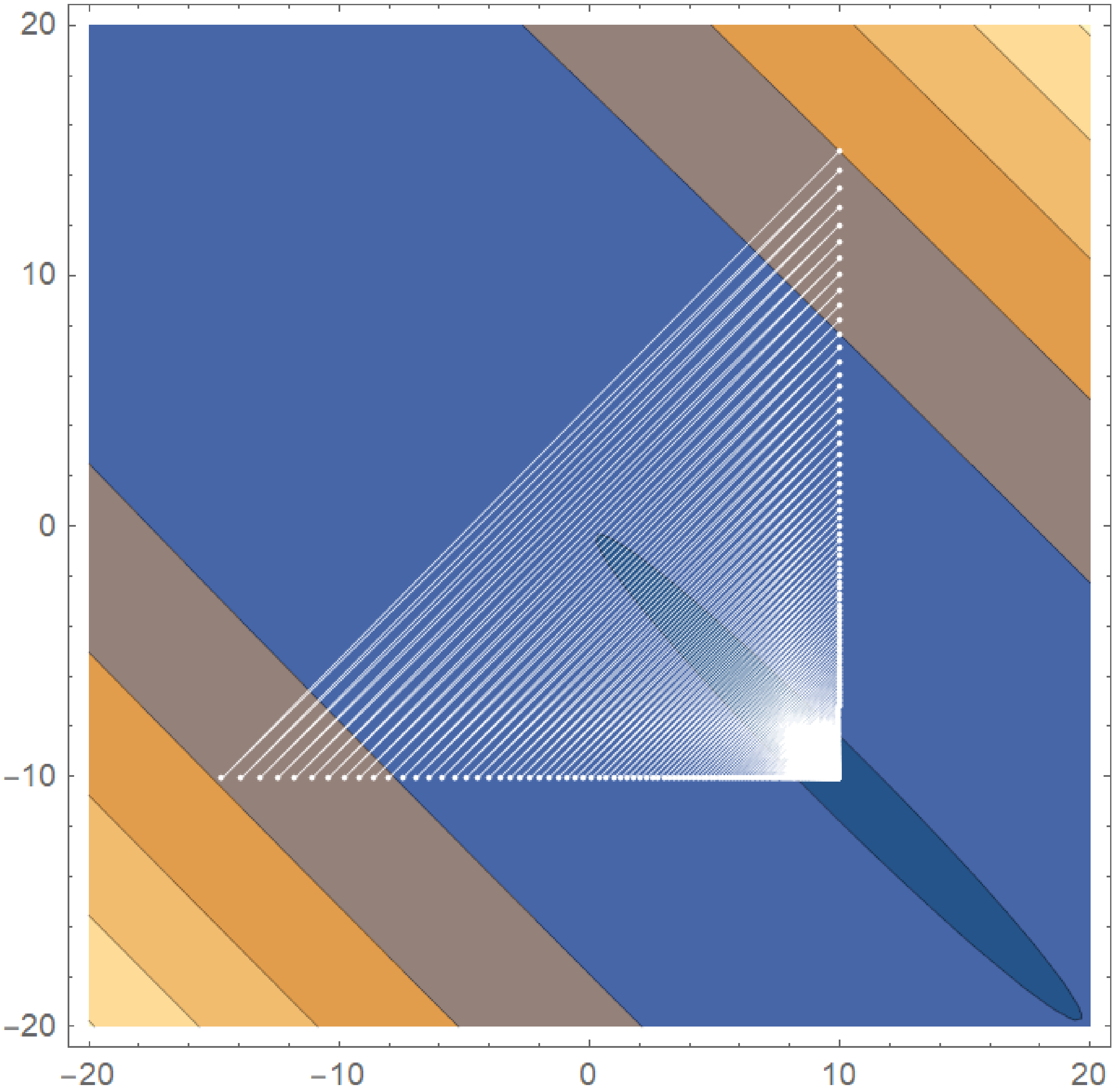


Метод находит точку $x^* = (-1.7499993, -0.2499991)$ за 49 шагов.

В данном случае метод каждый раз перескакивает область минимума, но находит точку минимума аз приемлемое количество шагов.

Функция 3: $64 \cdot x^2 + 126 \cdot x \cdot y + 64 \cdot y^2 - 10 \cdot x + 30 \cdot y + 13$

$\alpha = \frac{1}{128}$, $\varepsilon = 10^{-5}$



Метод находит точку $x^* = (9.9606296, -10.0393697)$ за 1109 шагов.

В данном случае функция имеет ярко выраженный овражный характер и градиентный спуск производит более 1000 **итераций, так как функция имеет овражный характер.** $q*$ **близка к** 1**.**

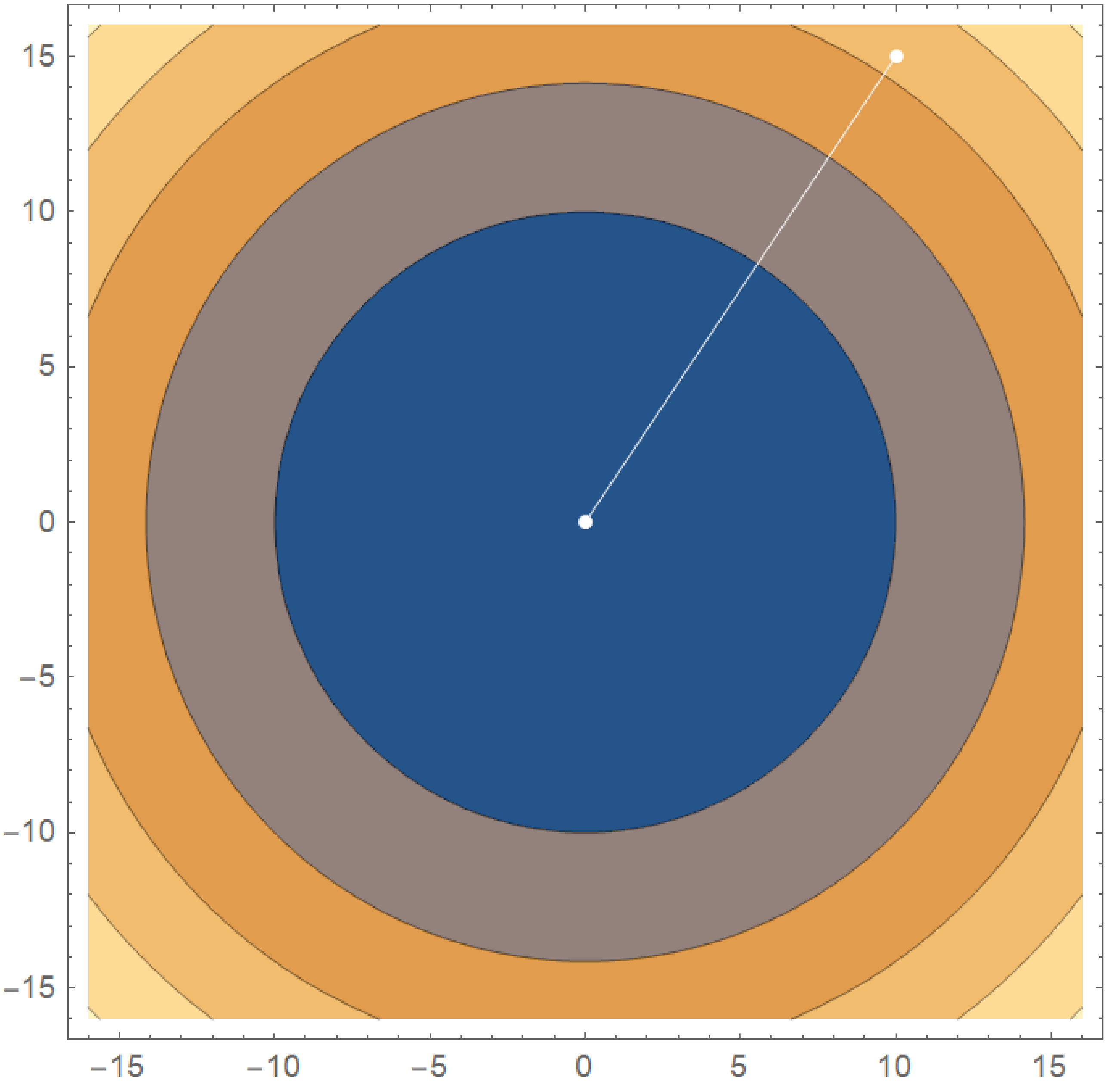
8.3 Метод наискорейшего спуска

8.3.1 Метод золотого сечения

Во всех функциях начальная точка: $(x, y) = (10.0, 15.0)$

Функция 1: $x^2 + y^2$

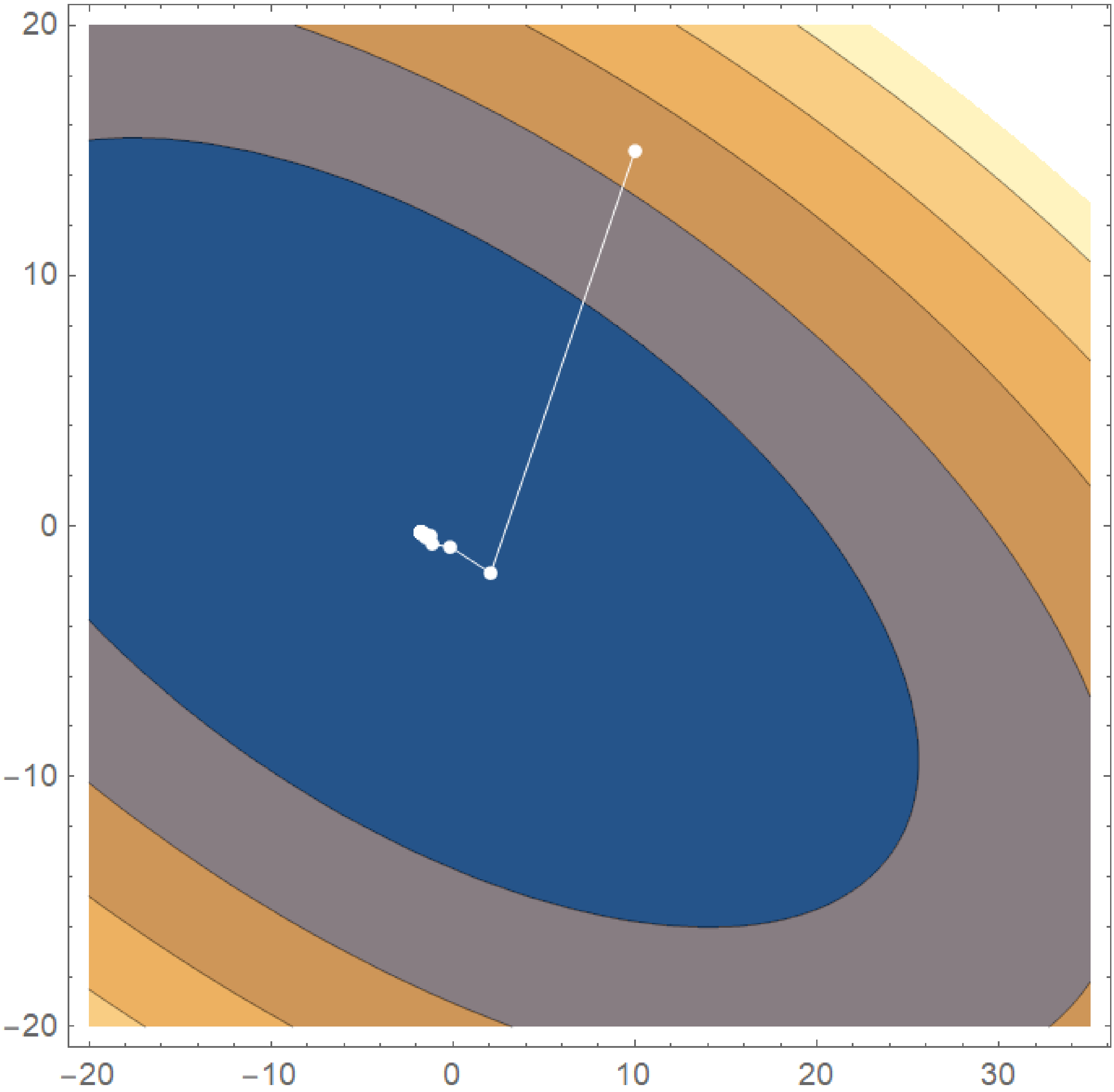
$\varepsilon = 10^{-5}$



Метод нашёл точку $x^* = (0, 0)$ за 1 шаг.
Аналогично, метод находит точку минимума за одну итерацию

Функция 2: $x^2 + 2 \cdot x \cdot y + 3 \cdot y^2 + 4 \cdot x + 5 \cdot y + 6$

$\varepsilon = 10^{-5}$

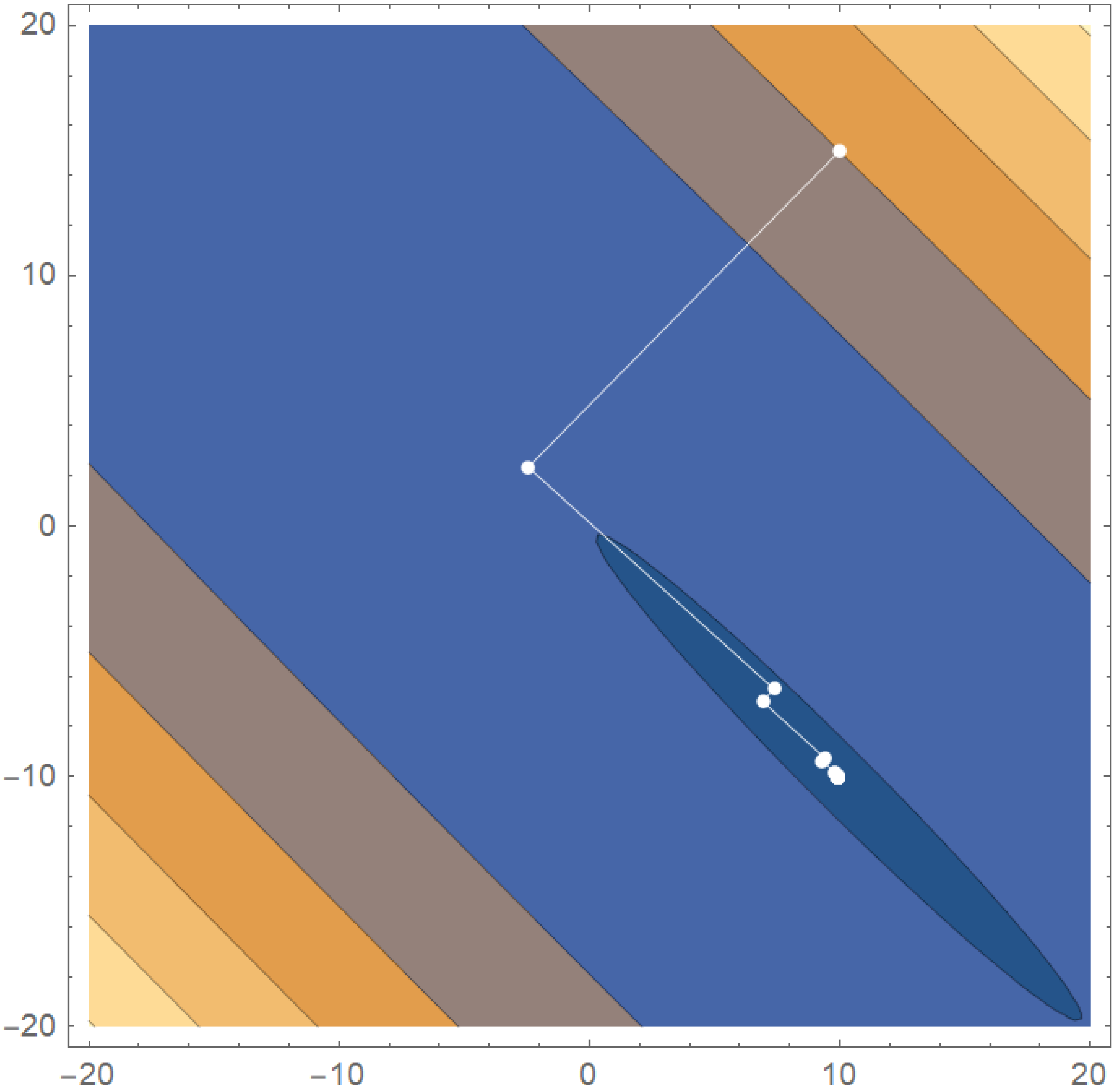


Метод находит точку $(-1.7499977, -0.2500010)$ за 8 шагов.

По сравнению с методом градиентного спуска, метод наискорейшего спуска не перескакивает через область минимума (нет зигзагообразных траекторий), поэтому траектория более «спокойная»

Функция 3: $64 \cdot x^2 + 126 \cdot x \cdot y + 64 \cdot y^2 - 10 \cdot x + 30 \cdot y + 13$

$\varepsilon = 10^{-5}$



Метод находит точку $(9.9606286, -10.0393688)$ за 20 шагов.

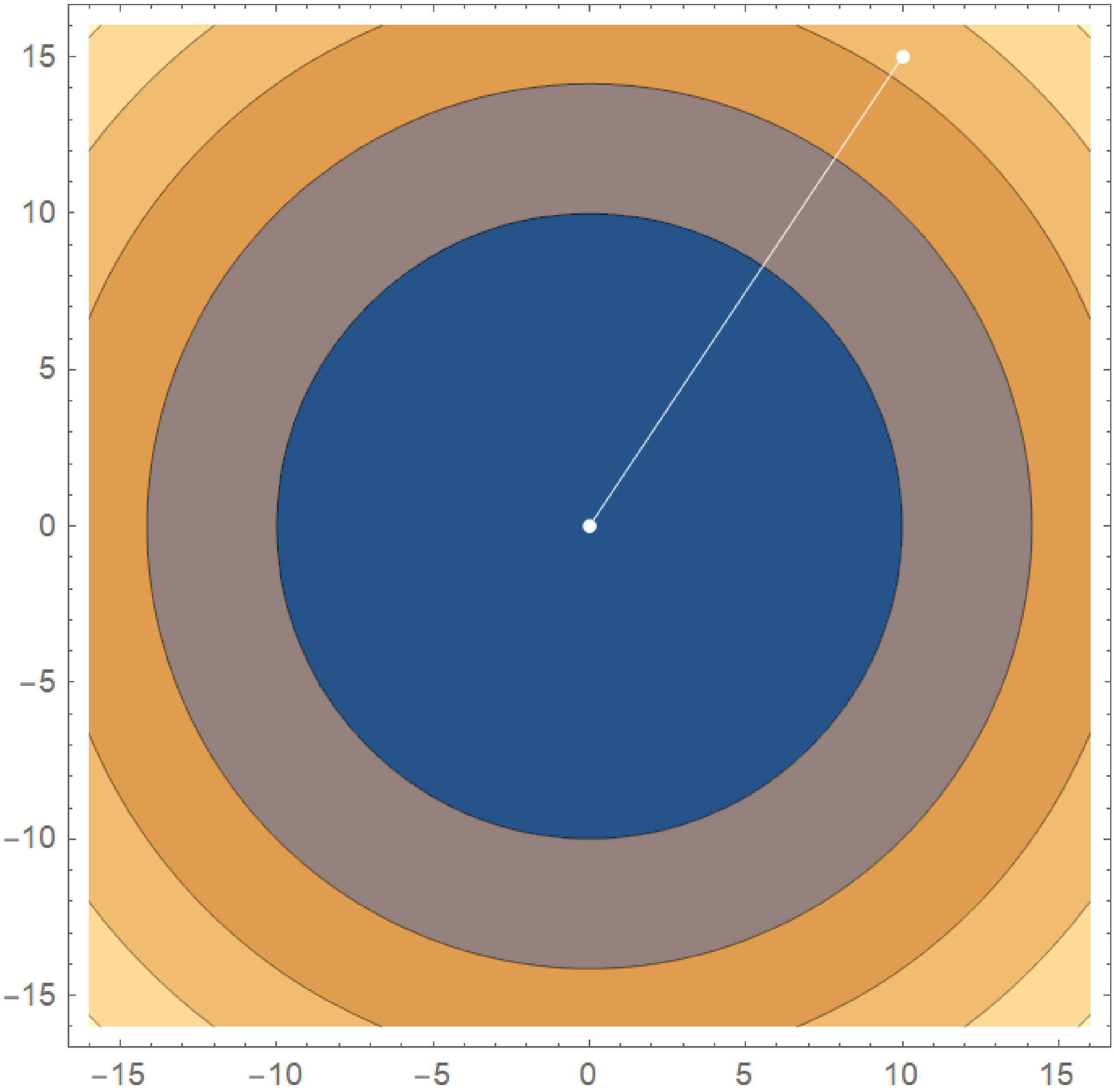
Функция имеет ярко выраженный овражный характер. По сравнению с методом градиентного спуска, первой итерацией метод попадает (примерно) на линию, содержащую минимум ($y = -x$), а затем скачет по направлению к минимуму. Так как после первой итерации метод попадает не точно на прямую $y = -x$, градиент направлен не строго в направлении глобального минимума, из-за этого возникают небольшие скачки.

8.4 Метод сопряженных градиентов

Во всех функциях начальная точка: $(x, y) = (10.0, 15.0)$

Функция 1: $x^2 + y^2$

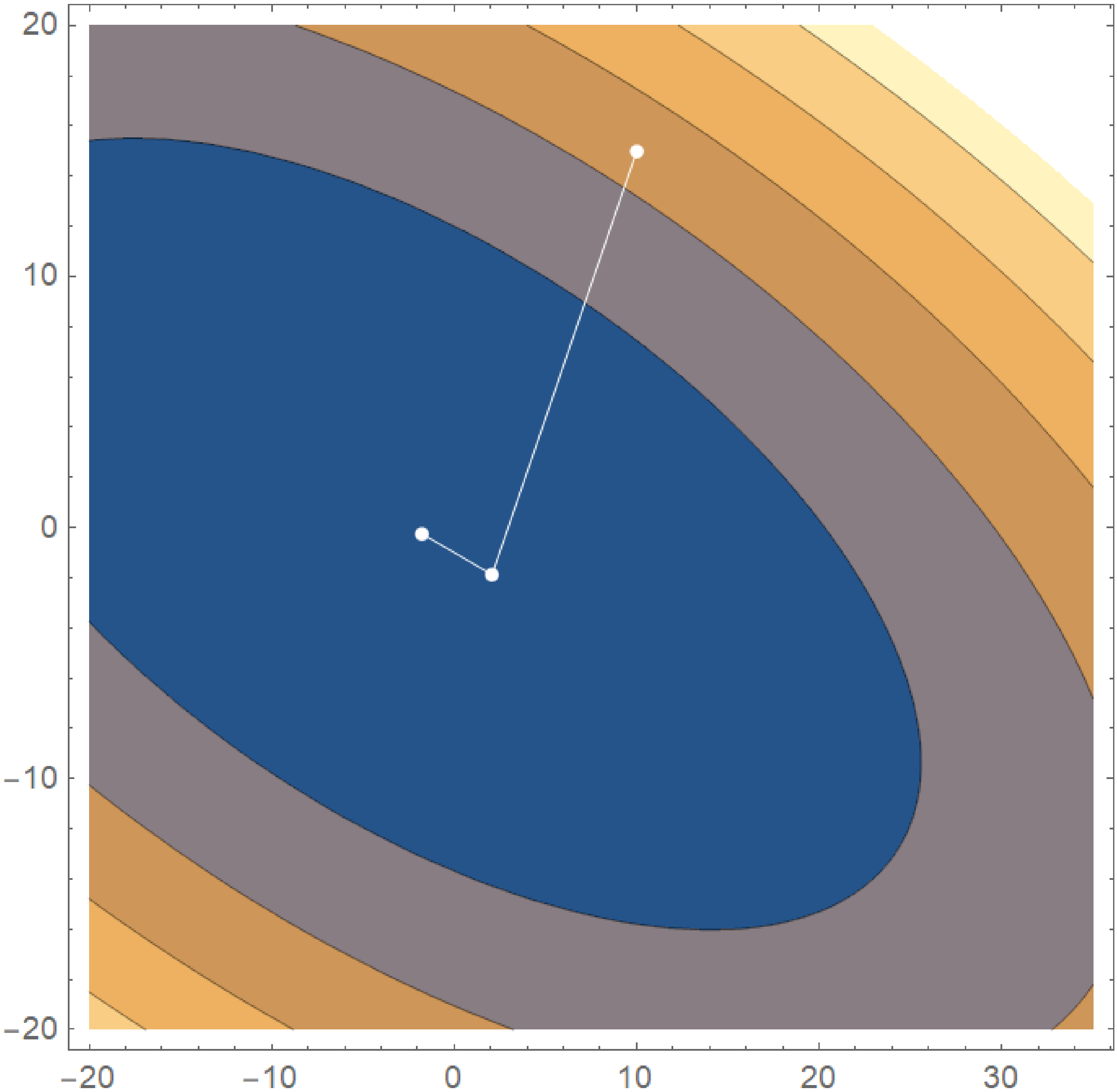
$\varepsilon = 10^{-5}$



Метод находит точку $(0, 0)$ за 1 шаг.
В данном случае метод сразу находит точку минимума.

Функция 2: $x^2 + 2 \cdot x \cdot y + 3 \cdot y^2 + 4 \cdot x + 5 \cdot y + 6$

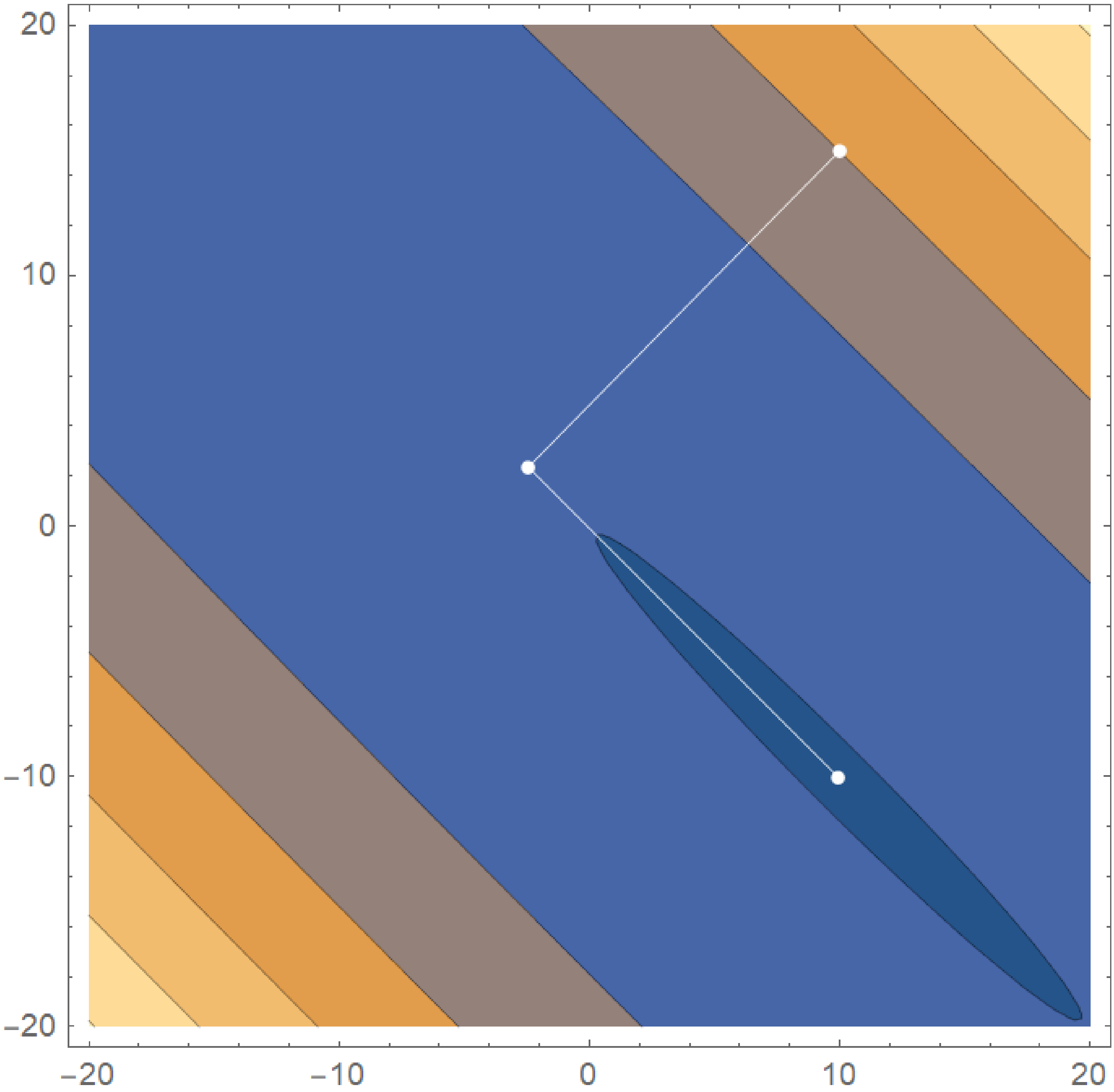
$\varepsilon = 10^{-5}$



Метод находит точку $(-1.75, -0.25)$ за 2 шага.
Метод находит точку минимума за два шага, как и предполагалось

Функция 3: $64 \cdot x^2 + 126 \cdot x \cdot y + 64 \cdot y^2 - 10 \cdot x + 30 \cdot y + 13$

$\varepsilon = 10^{-5}$



Метод находит точку $(9.9606299, -10.0393701)$ за 2 шага.
Метод находит точку минимума за два шага, как и предполагалось

8.5 Вывод

Метод сопряженных градиентов имеет наиболее «спокойные» траектории на квадратичных функциях. Наглядно видно, что методы градиентного и наискорейшего спусков ведут себя зигзагообразно на квадратичных функциях, у которых число обусловленности матрицы A велико.

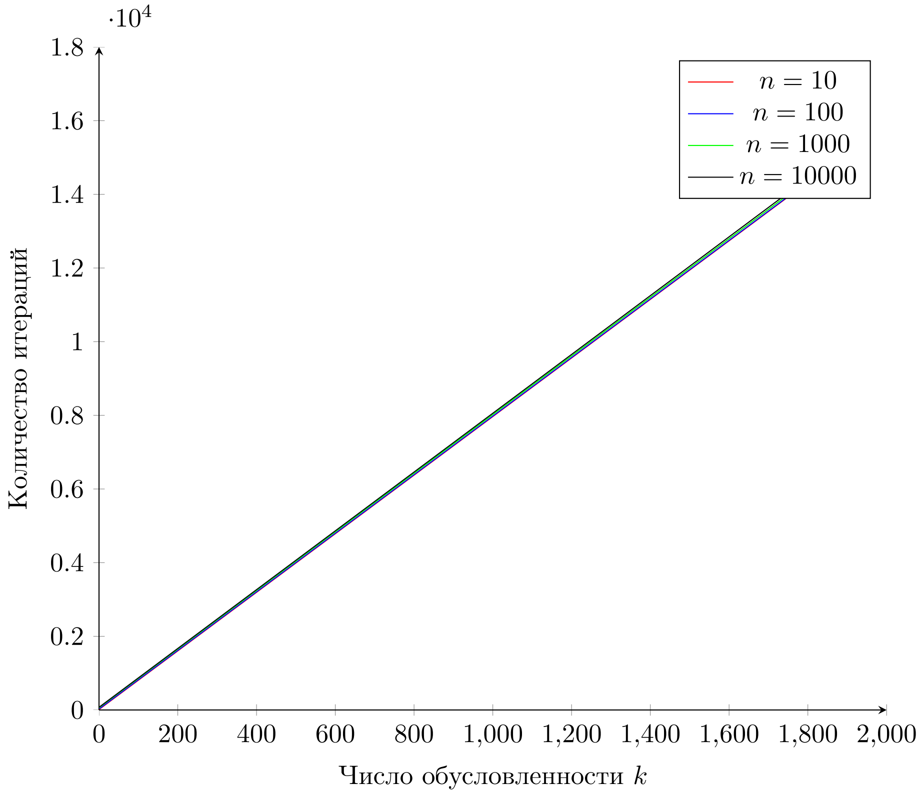
9 Оценка сходимости

10 Зависимость числа итераций от

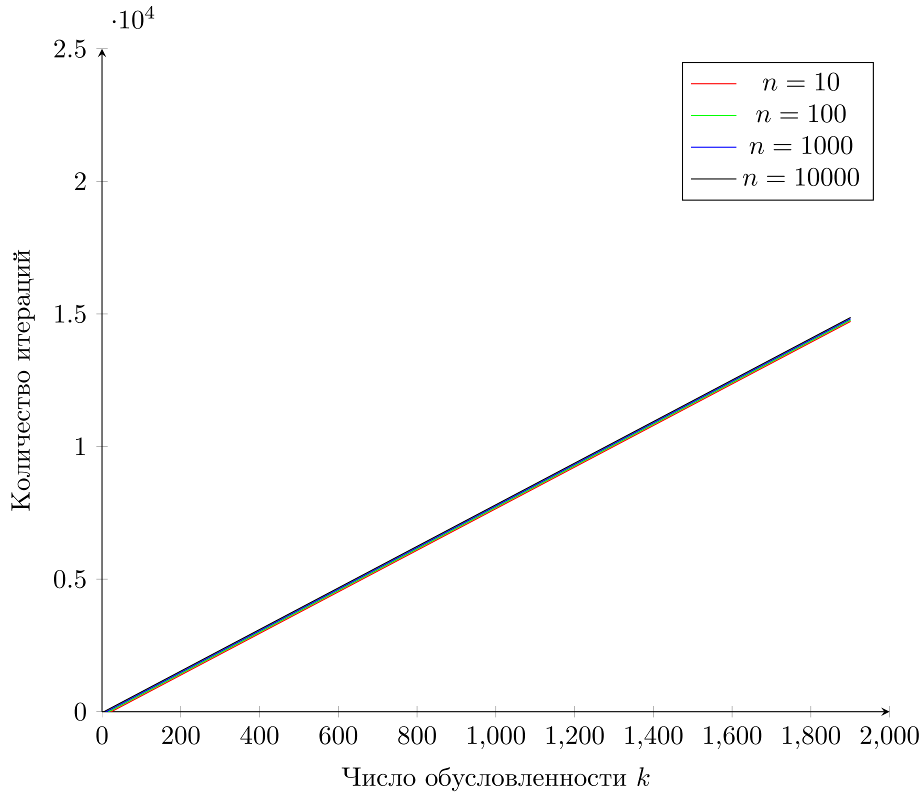
Оценим количество итераций $T(n, k)$ в зависимости от размерности пространства $\mathbb{R}^n$ аргумента x функции f и от числа обусловленности k матрицы A.

Число обусловленности $k = \frac{L}{l}$, где L и l — наибольшее и наименьшее собственное значение матрицы A соответственно. Оно характеризует степень «вытянутости» квадратичной функции или то, насколько сильно может измениться значение при малом изменении аргумента.

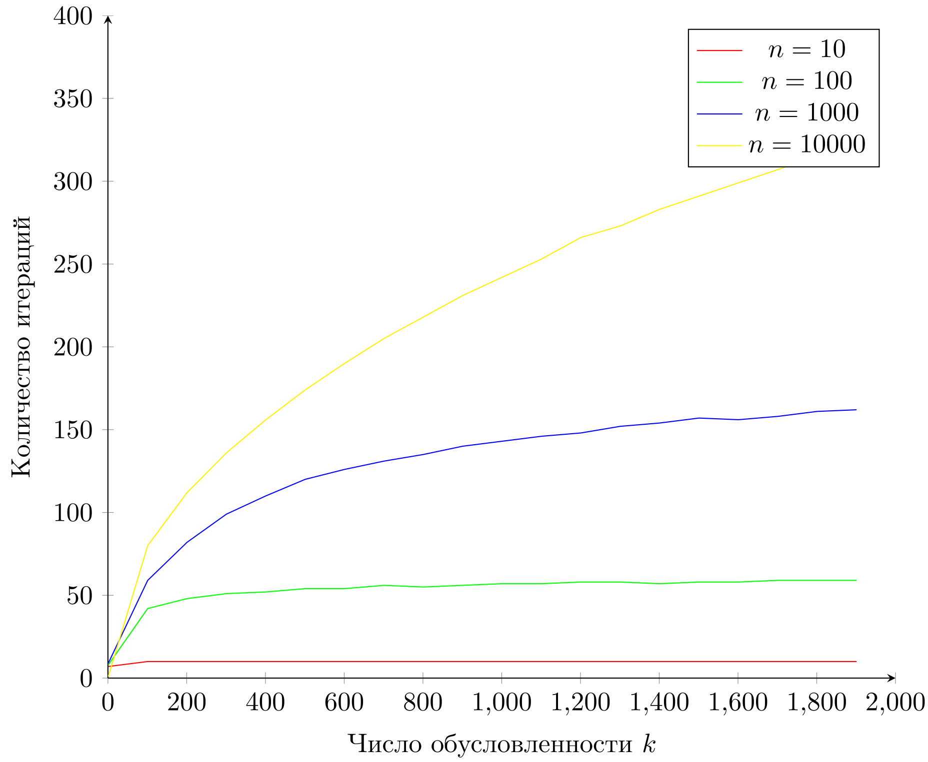
10.1 Метод градиентного спуска



10.2 Метод наискорейшего спуска с методом золотого сечения



10.3 Метод сопряженных градиентов



10.4 Вывод

Метод градиентного спуска и метод наискорейшего спуска ведут себя примерно одинаково по количеству шагов, необходимых для сходимости, причем количество итераций растет линейно с ростом числа обусловленности k. Однако, количество итераций не зависит от размерности пространства аргумента минимизируемой функции.

Количество итераций, необходимых для метода сопряженных градиентов, растет логарифмически с увеличением числа обусловленности и логарифмически с увеличением размерности n.

11 Программный код

11.1 Класс для хранения итераций

```java
/**
 * Iteration step POJO.
 */
public class IterationStep {

    /**
     * Step number. i.
     */
    private final long stepNumber;

    /**
     * Vector (point) x[i].
     */
    private final Vector vector;

    /**
     * Function result in {@link IterationStep#vector}. f(x[i])
     */
    private final double functionResult;

    public IterationStep(final long stepNumber, final Vector vector, final
        double functionResult) {
        this.stepNumber = stepNumber;
        this.vector = vector;
        this.functionResult = functionResult;
    }

    /**
     * Get step number.
     *
     * @return step number.
     */
    public long getStepNumber() {
        return stepNumber;
    }

    /**
     * Get vector.
     *
     * @return vector.
     */
    public Vector getVector() {
        return vector;
    }

    /**
     * Get function result in {@link IterationStep#vector}.
     *
     * @return function result.
     */
    public double getFunctionResult() {
        return functionResult;
    }

}
```

11.2 Программное представление матрицы

```java
/**
 * Matrix representation.
 */
public class Matrix {

    /**
     * Underlying matrix.
     */
    private final double[][] a;

    /**
     * Transposed flag.
     */
    private final boolean transposed;

    /**
     * Constructor for matrix. Transposed {@code false} by default.
     *
     * @param a given matrix.
     */
    public Matrix(final double[][] a) {
        this(a, false);
    }

    /**
     * Constructor for matrix.
     *
     * @param a          given matrix.
     * @param transposed given transposed flag.
     */
    private Matrix(double[][] a, boolean transposed) {
        if (a.length == 0) {
            throw new IllegalArgumentException();
        }

        final int n = a[0].length;
        if (n == 0) {
            throw new IllegalArgumentException();
        }

        for (double[] line : a) {
            if (line.length != n) {
                throw new IllegalArgumentException();
            }
        }

        this.a = a;

        this.transposed = transposed;
    }

    /**
     * Get element from matrix.
     *
     * @param i row index.
     * @param j column index.
     * @return matrix element.
     */
    public double get(int i, int j) {
```

```java
        return (transposed ? a[j][i] : a[i][j]);
    }

    /**
     * Number of columns.
     *
     * @return number of columns.
     */
    public int verticalLength() {
        return (transposed ? a[0].length : a.length);
    }

    /**
     * Number of rows.
     *
     * @return number of rows.
     */
    public int horizontalLength() {
        return (transposed ? a.length : a[0].length);
    }

    /**
     * Multiply matrix on given scalar number.
     *
     * @param alpha given scalar number.
     * @return multiplied matrix.
     */
    public Matrix mul(final Number alpha) {
        double t = alpha.doubleValue();
        int n = verticalLength();
        int m = horizontalLength();
        double[][] a = new double[n][m];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                a[i][j] = t * get(i, j);
            }
        }
        if (this instanceof Vector) {
            return new Vector(a);
        } else {
            return new Matrix(a);
        }
    }


    /**
     * Add matrix.
     *
     * @param right matrix to be added.
     * @return new matrix.
     */
    public Matrix add(final Matrix right) {
        int n = verticalLength();
        int m = horizontalLength();
        if (n != right.verticalLength() || m != right.horizontalLength()) {
            throw new IllegalArgumentException();
        }

        double[][] a = new double[n][m];
        for (int i = 0; i < n; i++) {
```

```java
            for (int j = 0; j < m; j++) {
                a[i][j] = get(i, j) + right.get(i, j);
            }
        }
        if (this instanceof Vector) {
            return new Vector(a);
        } else {
            return new Matrix(a);
        }
    }

    /**
     * Multiply matrix on another matrix.
     *
     * @param right given matrix.
     * @return multiplied matrix.
     */
    public Matrix mul(final Matrix right) {
        int n = verticalLength();
        int m = horizontalLength();
        if (right.verticalLength() != m) {
            throw new IllegalArgumentException();
        }
        int k = right.horizontalLength();

        double[][] a = new double[n][k];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < k; j++) {
                a[i][j] = 0.0;
                for (int l = 0; l < m; l++) {
                    a[i][j] += get(i, l) * right.get(l, j);
                }
            }
        }

        Matrix result = new Matrix(a);
        if (result.horizontalLength() == 1) {
            return new Vector(result.a);
        } else {
            return result;
        }
    }

    /**
     * Transpose matrix.
     * @return transposed matrix.
     */
    public Matrix transpose() {
        return new Matrix(a, !transposed);
    }

}
```

11.3 Программное представление вектора. Наследуется от Matrix

```
/**
 * One dimensional matrix representation.
 */
public class Vector extends Matrix {
    /**
     * Vector constructor.
     *
     * @param vector given args.
     */
    public Vector(final double... vector) {
        super(prepareMatrix(vector));
    }

    /**
     * Create vector from matrix.
     *
     * @param matrix given matrix.
     */
    public Vector(final double[][] matrix) {
        super(matrix);
    }

    /**
     * Get vectors element by index.
     *
     * @param i given index
     * @return element.
     */
    public double get(final int i) {
        return get(i, 0);
    }

    /**
     * Vectors length.
     *
     * @return vectors length.
     */
    public int length() {
        return verticalLength();
    }

    /**
     * Multiply vector by scalar number.
     *
     * @param alpha given scalar number.
     * @return multiplied vector.
     */
    public Vector mul(final Number alpha) {
        return (Vector) super.mul(alpha);
    }

    /**
     * Add two vectors.
     *
     * @param right given vector.
     * @return added vector.
     */
    public Vector add(final Vector right) {
```

```java
        return (Vector) super.add(right);
    }

    /**
     * Subtract two vectors.
     *
     * @param right given vector
     * @return subtracted vector.
     */
    public Vector sub(final Vector right) {
        return (Vector) super.add(right.mul(-1.0));
    }

    /**
     * Multiply vector on another vector.
     *
     * @param right given vector.
     * @return multiplied vector.
     */
    public double scalarMul(final Vector right) {
        return transpose().mul(right).get(0, 0);
    }

    /**
     * Square rate of vector.
     *
     * @return square rate.
     */
    public double sqrRate() {
        double sum = 0.0;
        for (int i = 0; i < length(); i++) {
            sum += get(i) * get(i);
        }
        return sum;
    }

    /**
     * Vectors rate.
     *
     * @return rate.
     */
    public double rate() {
        return Math.sqrt(sqrRate());
    }

    /**
     * Create matrix from vector.
     *
     * @param vector given vector.
     * @return matrix with one row.
     */
    private static double[][] prepareMatrix(double[] vector) {
        if (vector.length == 0) {
            throw new IllegalArgumentException();
        }
        double[][] a = new double[vector.length][1];
        IntStream.range(0, vector.length).forEach(i -> a[i][0] = vector[i]);
        return a;
    }}
```

11.4 Программное представление квадратичной функции

```java
/**
 * Quadratic function representation.
 */
public class QuadraticFunction implements Function<Vector, Double> {

    /**
     * Matrix a. {@link Matrix}
     */
    private final Matrix a;

    /**
     * Matrix b. {@link Vector}
     */
    private final Vector b;

    /**
     * Coefficient c.
     */
    private final double c;

    /**
     * Quadratic function constructor.
     *
     * @param a matrix.
     * @param b vector
     * @param c coefficient.
     */
    public QuadraticFunction(final Matrix a, final Vector b, final double c) {
        int n = a.verticalLength();
        int m = a.horizontalLength();

        if (n != m || n != b.length()) {
            throw new IllegalArgumentException();
        }

        double[][] aSymmetric = new double[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                aSymmetric[i][j] = a.get(i, j) + a.get(j, i);
            }
        }

        this.a = new Matrix(aSymmetric);
        this.b = b;
        this.c = c;
    }

    /**
     * Get function result in point.
     *
     * @param point vector representation of point
     * @return function result.
     */
    @Override
    public Double apply(final Vector point) {
        return 0.5 * ((Vector) a.mul(point)).scalarMul(point) +
            b.scalarMul(point) + c;
    }
```

```
/**
 * Getter for a.
 *
 * @return a.
 */
public Matrix getA() {
    return a;
}

/**
 * Getter for b.
 *
 * @return b.
 */
public Vector getB() {
    return b;
}

/**
 * Getter for c.
 *
 * @return c.
 */
public double getC() {
    return c;
}

/**
 * Find gradient for function in point.
 *
 * @param point given point
 * @return gradient.
 */
public Vector gradient(final Vector point) {
    return ((Vector) getA().mul(point)).add(getB());
}

/**
 * Find anti gradient for function in point
 *
 * @param point given point.
 * @return anti gradient.
 */
public Vector antiGradient(final Vector point) {
    return gradient(point).mul(-1);
}

/**
 * Dimensions.
 *
 * @return dimensions.
 */
public int dimensions() {
    return a.verticalLength();
}

/**
 * Create quadratic function from 2d. {@code Ax^2 + Bxy + C y^2 + Dx + Ey +
 F}
 *
```

```java
     * @param a coefficient.
     * @param b coefficient.
     * @param c coefficient.
     * @param d coefficient.
     * @param e coefficient.
     * @param f coefficient.
     * @return created quadratic function.
     */
    public static QuadraticFunction from2d(double a, double b, double c, double
        d, double e, double f) {
        double[][] A = new double[][]{
                {a, b},
                {0.0, c}
        };
        double[] B = new double[]{d, e};
        return new QuadraticFunction(new Matrix(A), new Vector(B), f);
    }
}
```

11.5 Абстрактный класс для методов градиентного и наискорейшего спуска

Метод iteration переходит на новую итерацию для метода оптимизации, записывает данные в список для логирования каждой итерации.
Метод newPoint возвращает переходит на новую точку. Каждый метод оптимизации наследующий данный класс обязан реализовывать данный метод.

```java
/**
 * Abstract gradient minimizer.
 */
public abstract class AbstractGradientMinimizer {

    /**
     * New iteration function.
     *
     * @param f         given function.
     * @param rateValue rate value (alpha or value to calculate alpha).
     * @param eps       epsilon.
     * @param point     given point.
     * @return returns the point where the function reaches its minimum.
         {@link MinimizationResult}
     */
    public MinimizationResult iteration(
            final QuadraticFunction f,
            final double rateValue,
            final double eps,
            final Vector point,
            final long numberOfIterations,
            final List<IterationStep> steps
    ) {
        final double fPoint = f.apply(point);
        steps.add(new IterationStep(numberOfIterations + 1, point, fPoint));
        final Vector gradient = f.gradient(point);
        if (gradient.rate() < eps || numberOfIterations >= 1000) {
            return MinimizationResult.of(point, fPoint, numberOfIterations,
                steps);
        } else {
            return this.newPoint(f, point, eps, rateValue, fPoint,
                gradient, numberOfIterations + 1, steps);
        }
    }

    /**
     * Finds new point (xk).
     *
     * @param f           given function.
     * @param point       given point.
     * @param eps         epsilon.
     * @param rateValue rate value (alpha or value to calculate alpha).
     * @param fPoint      function result in given point.
     * @param gradient    gradient for given function and point.
     * @return new point.
     */
    public abstract MinimizationResult newPoint(
            final QuadraticFunction f,
            final Vector point,
            final double eps,
            final double rateValue,
            final double fPoint,
            final Vector gradient,
            final long numberOfIterations,
            final List<IterationStep> steps
```

```
    );
}
```

11.6 Метод градиентного спуска

Главная идея метода градиентного спуска заключается в оптимизации функции в направлении наискорейшего спуска. Направление спуска находится при помощи антиградиента $-\nabla f(x)$.

Алгоритм находит приближение точки минимума итеративно: на каждой итерации алгоритма находится антиградиент f относительно текущего приближения x^k, после чего выбирается шаг – α, на который в направлении антиградиента совершается переход к следующему приближению – x^{k+1}. Шаг α выбирается следующим образом: уменьшается в два раза пока $f(x^k - \alpha \cdot \nabla f(x))$ не станет меньше чем $f(x)$. Перед переходом на следующую итерацию α уменьшается в два раза.

Изначальное значение α определяется как $\frac{2}{l+L}$, где l – минимальное собственное число, L – максимальное собственное число.

Итерации продолжаются до тех пор пока не выполнится критерий остановки, в нашем случаее критерий остановки выглядит так – $\|\nabla f(x)\| < \varepsilon$.

```
/**
 * Gradient descent method.
 */
public class GradientDescentMinimizer extends AbstractGradientMinimizer {
    public MinimizationResult newPoint(
            final QuadraticFunction f,
            final Vector point,
            final double eps,
            final double alpha,
            final double fX,
            final Vector gradient,
            final long numberOfIterations,
            final List<IterationStep> steps
    ) {
        final Vector y = point.sub(gradient.mul(alpha));
        final double fY = f.apply(y);
        if (fY < fX) {
            return iteration(f, alpha, eps, y, numberOfIterations, steps);
        } else {
            return newPoint(f, point, eps, alpha / 2, fX, gradient,
                numberOfIterations, steps);
        }
    }
}
```

11.7 Метод наискорейшего спуска

Метод наискорейшего спуска аналогичен методу градиентного спуска, за исключением поиска шага α. По направлению вектора градиента ищется точка минимума функции и из этого уже вычисляется α методом одномерной оптимизации. Границы для поиска α задаются интервалом $(0, \frac{2}{L})$, где L – максимальное собственное число.

Стоит заметить, что все используемые методы одномерной оптимизации были реализованы в ходе Лабораторной работы №1 и добавлены в данную лабораторную работу как зависимость модуля, таким образом вся кодовая база одномерных методов сохранила изначальное состояние.

```
/**
 * Gradient fastest descent minimizer.
 */
public class GradientDescentFastestMinimizer extends AbstractGradientMinimizer {
    public MinimizationResult newPoint(
            final QuadraticFunction f,
            final Vector point,
            final double eps,
            final double maxEigenvalue,
            final double fPoint,
            final Vector gradient,
            final long numberOfIterations
    ) {
        final var method = new FibonacciMethod(
                0.0,
                maxEigenvalue,
                alpha -> f.apply(point.sub((f.gradient(point)).mul(alpha))),
                eps
        );
        method.calculate();
        final double learningRate = method.getMinimumArgument();
        final Vector y = point.sub(gradient.mul(learningRate));
        return minimize(f, maxEigenvalue, eps, y, numberOfIterations);
    }

}
```

11.8 Метод сопряженных градиентов

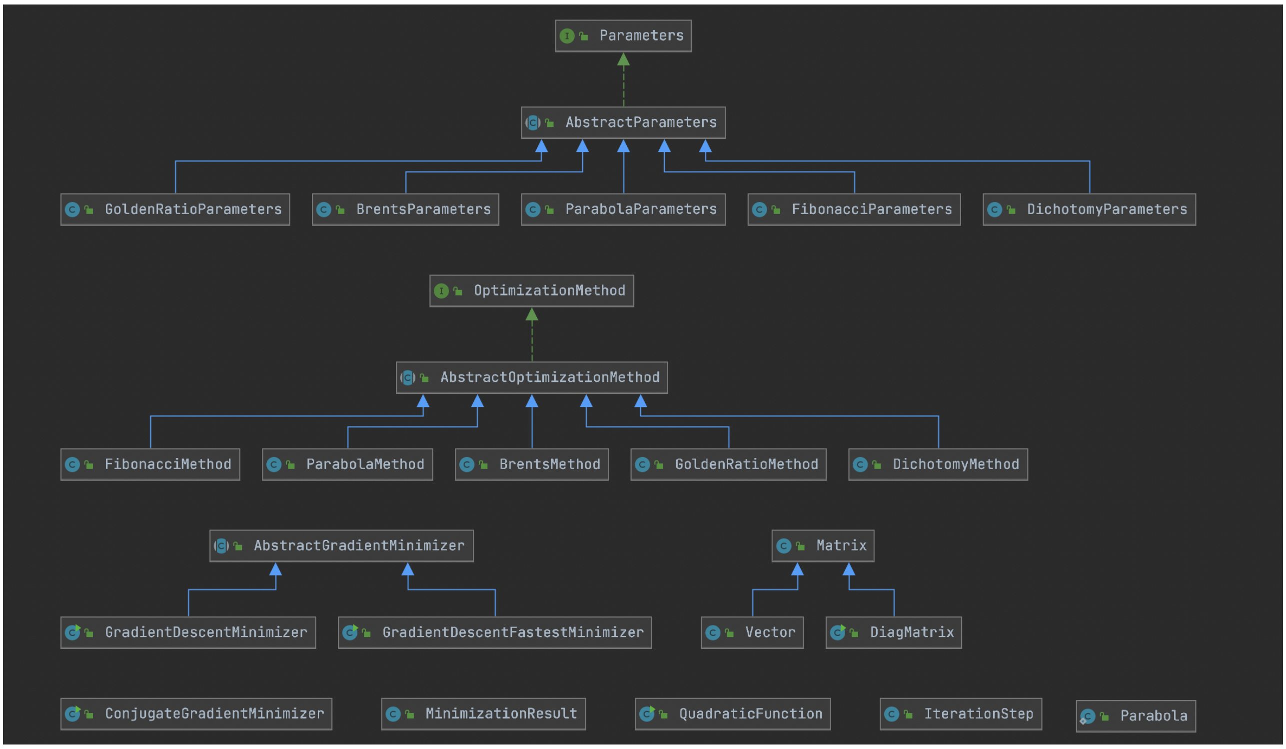
На вход алгоритму подается минимизируемая функция, начальная точка и ε. На каждом шаге (функция iteration) реализуется схема из пункта 6.1.

```
/**
 * Conjugate gradient minimizer.
 */
public class ConjugateGradientMinimizer {

    /**
     * Given function.
     */
    private final QuadraticFunction f;

    /**
     * Iteration steps.
     */
    private final List<IterationStep> steps = new ArrayList<>();

    /**
     * Search direction.
     */
    private final List<Vector> p = new ArrayList<>();

    /**
     * Alpha coefficient.
     */
    private final List<Double> alpha = new ArrayList<>();

    /**
     * Beta coefficient.
     */
    private final List<Double> beta = new ArrayList<>();

    /**
     * Gradients.
     */
    private final List<Vector> gradients = new ArrayList<>();

    /**
     * Given epsilon.
     */
    private final double eps;

    /**
     * Constructs conjugate gradient method.
     *
     * @param f given function.
     * @param startPoint given start point.
     * @param eps given epsilon.
     */
    public ConjugateGradientMinimizer(
            final QuadraticFunction f,
            final Vector startPoint,
            final double eps
    ) {
        this.f = f;
        this.steps.add(new IterationStep(0, startPoint, f.apply(startPoint)));

        final Vector gradient0 = f.gradient(startPoint);
        this.gradients.add(gradient0);
```

```java
        this.p.add(gradient0.mul(-1));

        this.eps = eps;
    }

    /**
     * New iteration.
     *
     * @param k last iteration step.
     */
    private void iteration(final int k) {
        final Vector pk = p.get(k);
        final Vector aPk = (Vector) f.getA().mul(pk);
        final double alphaK = gradients.get(k).sqrRate() / aPk.scalarMul(pk);
        alpha.add(alphaK);

        final Vector xk = steps.get(k).getVector();
        final Vector xk1 = xk.add(pk.mul(alphaK));
        steps.add(new IterationStep(k + 1, xk1, f.apply(xk1))); // calculating
            function only for step logging.

        final Vector gradientK = gradients.get(k);
        final Vector gradientK1 = gradientK.add(aPk.mul(alphaK));
        gradients.add(gradientK1);

        final double betaK = gradientK1.sqrRate() / gradientK.sqrRate();
        beta.add(betaK);

        final Vector pk1 = gradientK1.mul(-1).add(pk.mul(betaK));
        p.add(pk1);
    }

    /**
     * Minimize given function.
     */
    public void minimize() {
        final double epsSqr = eps * eps;
        for (int i = 0; i <= f.dimensions(); i++) {
            iteration(i);
            if (gradients.get(gradients.size() - 1).sqrRate() < epsSqr) {
                break;
            }
        }
    }

    /**
     * Get minimum argument.
     *
     * @return minimum argument.
     */
    public Vector getMinX() {
        return steps.get(steps.size() - 1).getVector();
    }

    /**
     * Get function result in minimum argument.
     *
     * @return function result.
     */
```

```java
    public double getMinF() {
        return f.apply(getMinX());
    }

    /**
     * Get iteration steps.
     *
     * @return iteration steps.
     */
    public List<IterationStep> getSteps() {
        return steps;
    }

}
```

11.9 Диаграмма классов



12 Выводы

В ходе выполнения лабораторной работы мы провели доскональный анализ каждого из градиентных методов. Наилучшие результаты показал метод сопряженных градиентов, однако его использование ограничивается квадратичными функциями, в отличии от метода градиентного спуска и метода наискорейшего спуска.

Также мы провели анализ траекторий методов для трех выбранных квадратичных функций. С более подробными результатами анализа можно ознакомиться в пункте 8.2-8.5

Мы исследовали зависимость числа итераций от числа обусловленности оптимизируемой функции и размерности пространства n оптимизируемых переменных. Исследование проводилось в различных условиях, таких как:

- Запуск в одном потоке, ноутбук с 8-16ГБ ОЗУ DDR4. Для JVM было отведено 12ГБ ОЗУ DDR4, остальное под операционную систему. Исследование успешно завершено.

- Запуске в восьми потоках, персональный компьютер с 32ГБ ОЗУ DDR4, 30ГБ было отведено для JVM. В ходе исследования персональный компьютер был нагрет до критических значений (порядка 100^oC). Исследование прервано.

- Запуск в трех потоках на бумаге. Было израсходовано около 4 гелевых ручек. Исследование прервано.

- Запуск на 1000 потоков на арендованном сервере EC2 AWS, 256ГБ ОЗУ. После исследования №2 было принято решение отдать не более 240ГБ ОЗУ под нужды JVM. Исследование успешно завершено.

Результаты исследований были усреднены и добавлены в отчет.