Отчет по лабораторной работе №1 Методы оптимизаций

Прямые методы одномерной оптимизации

Гарипов Роман M3238
Гарипов Эмиль M3238
Косогоров Евгений M3239

Март 2021

УНИВЕРСИТЕТ ИТМО

1 Постановка задачи

Реализовать алгоритмы одномерной оптимизации функции:

- Метод дихотомии
- Метод золотого сечения
- Метод Фиббоначи
- Метод парабол
- Комбинированный метод Брента

И протестировать их на следующей задаче оптимизации унимодальной функции:

$$f(x) = -3x \sin 0.75x + e^{-2x} \to \min \text{ на интервале } [0; 2\pi]$$

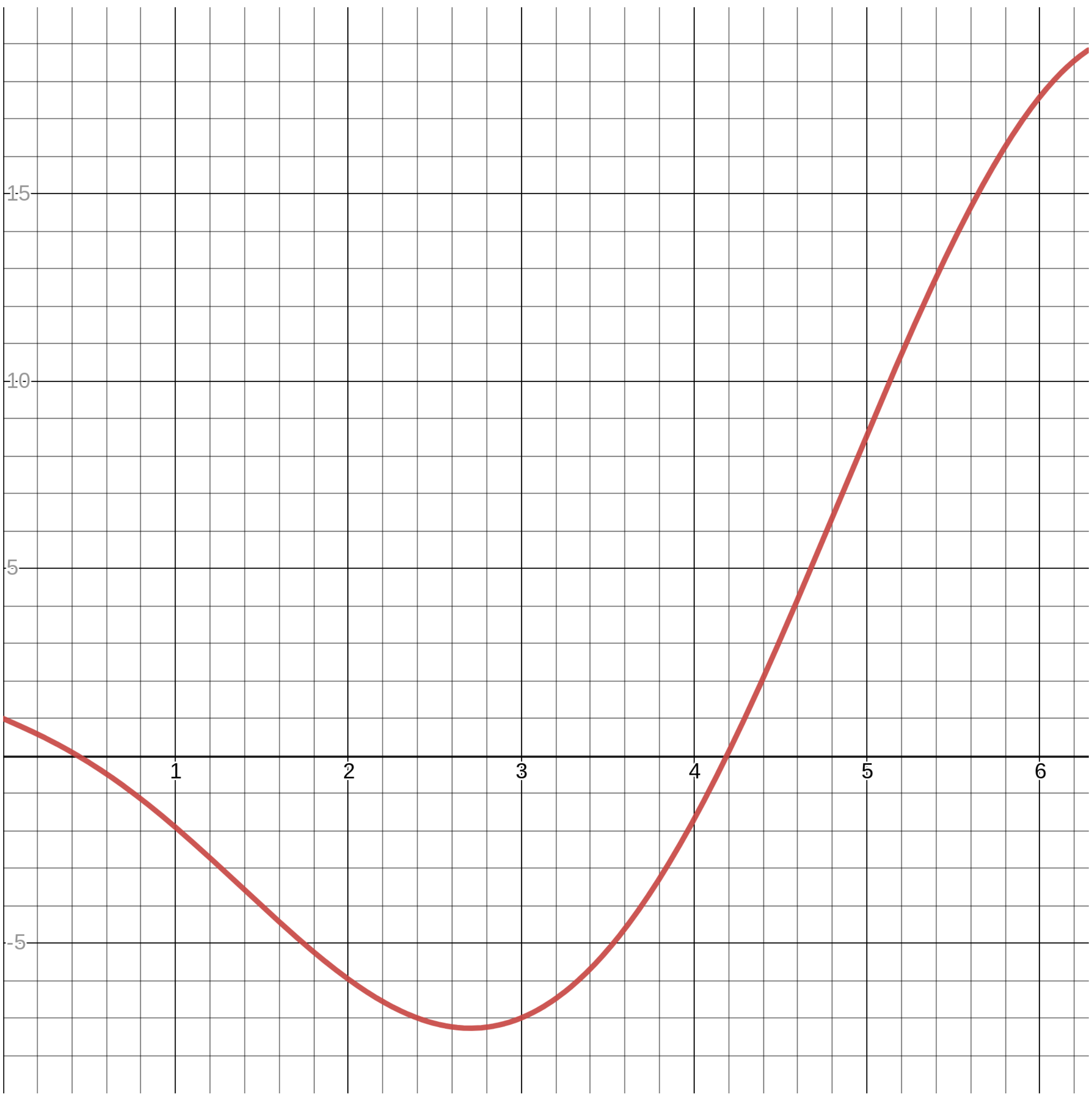


Рис. 1: График функции f

Аналитическое решение

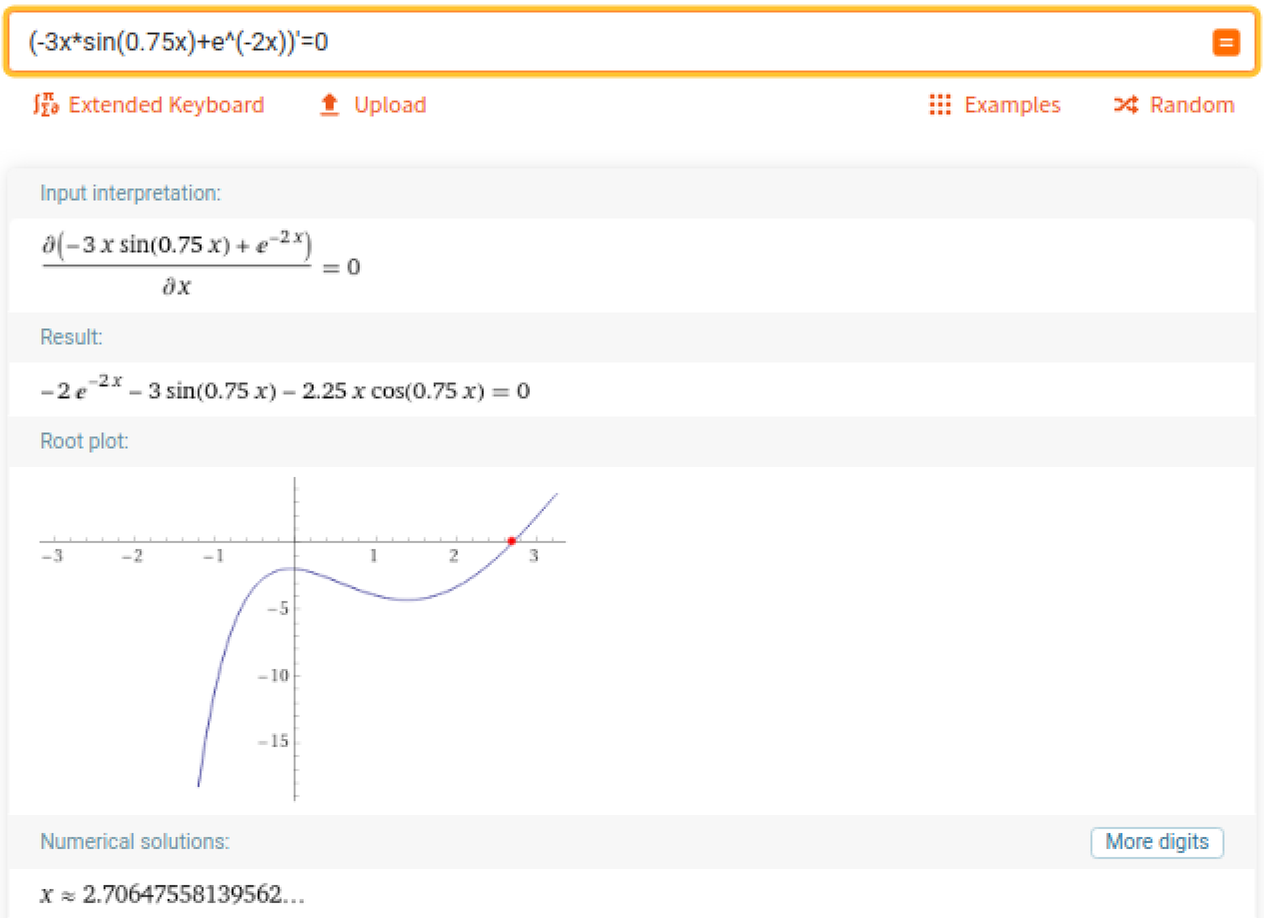


Рис. 2: Нахождение нулей производной функции f

Таким образом, минимальное значение функции f на интервале $[0; 2\pi]$ достигается в точке $x_{min} \approx 2.7065$.

2 Результаты исследований

Метод Дихотомии

l	r	x_1	x_2	fx_1	fx_2	$ratio$
0.0000	6.2832	3.1411	3.1421	-6.6639	-6.6610	
0.0000	3.1421	1.5705	1.5715	-4.3094	-4.3136	0.5000
1.5705	3.1421	2.3558	2.3568	-6.9231	-6.9250	0.5001
2.3558	3.1421	2.7485	2.7495	-7.2689	-7.2687	0.5003
2.3558	2.7495	2.5521	2.5531	-7.2036	-7.2045	0.5006
2.5521	2.7495	2.6503	2.6513	-7.2648	-7.2651	0.5012
2.6503	2.7495	2.6994	2.7004	-7.2742	-7.2742	0.5025
2.6994	2.7495	2.7239	2.7249	-7.2734	-7.2733	0.5050
2.6994	2.7249	2.7116	2.7126	-7.2743	-7.2742	0.5099
2.6994	2.7126	2.7055	2.7065	-7.2744	-7.2744	0.5195
2.7055	2.7126	2.7086	2.7096	-7.2743	-7.2743	0.5376
2.7055	2.7096	2.7070	2.7080	-7.2744	-7.2744	0.5700
2.7055	2.7080	2.7063	2.7073	-7.2744	-7.2744	0.6229
2.7055	2.7073	2.7059	2.7069	-7.2744	-7.2744	0.6973

$\mathbf{x} = 2.7067$ $\mathbf{y} = -7.2744$ $\boldsymbol{\varepsilon} = 0.001$ $\boldsymbol{\delta} = 0.00001$

Метод Золотого Сечения

l	r	x_1	x_2	fx_1	fx_2	$ratio$
0.0000	6.2832	2.4000	3.8832	-7.0034	-2.6461	
0.0000	3.8832	1.4833	2.4000	-3.9390	-7.0034	0.6180
1.4833	3.8832	2.4000	2.9665	-7.0034	-7.0600	0.6180
2.4000	3.8832	2.9665	3.3167	-7.0600	-6.0527	0.6180
2.4000	3.3167	2.7501	2.9665	-7.2685	-7.0600	0.6180
2.4000	2.9665	2.6164	2.7501	-7.2499	-7.2685	0.6180
2.6164	2.9665	2.7501	2.8328	-7.2685	-7.2247	0.6180
2.6164	2.8328	2.6990	2.7501	-7.2742	-7.2685	0.6180
2.6164	2.7501	2.6675	2.6990	-7.2697	-7.2742	0.6180
2.6675	2.7501	2.6990	2.7185	-7.2742	-7.2739	0.6180
2.6675	2.7185	2.6870	2.6990	-7.2732	-7.2742	0.6180
2.6870	2.7185	2.6990	2.7065	-7.2742	-7.2744	0.6180
2.6990	2.7185	2.7065	2.7111	-7.2744	-7.2743	0.6180
2.6990	2.7111	2.7036	2.7065	-7.2743	-7.2744	0.6180
2.7036	2.7111	2.7065	2.7082	-7.2744	-7.2743	0.6180
2.7036	2.7082	2.7054	2.7065	-7.2744	-7.2744	0.6180
2.7054	2.7082	2.7065	2.7072	-7.2744	-7.2744	0.6180
2.7054	2.7072	2.7061	2.7065	-7.2744	-7.2744	0.6180

$\mathbf{x} = 2.7063$ $\mathbf{y} = -7.2744$ $\boldsymbol{\varepsilon} = 0.001$

Метод Фибоначчи

l	r	x_1	x_2	fx_1	fx_2	$ratio$
0.0000	6.2832	2.4000	3.8832	-7.0034	-2.6461	
0.0000	3.8832	1.4833	2.4000	-3.9390	-7.0034	0.6180
1.4833	3.8832	2.4000	2.9665	-7.0034	-7.0600	0.6180
2.4000	3.8832	2.9665	3.3167	-7.0600	-6.0527	0.6180
2.4000	3.3167	2.7501	2.9665	-7.2685	-7.0600	0.6180
2.4000	2.9665	2.6164	2.7501	-7.2499	-7.2685	0.6180
2.6164	2.9665	2.7501	2.8328	-7.2685	-7.2247	0.6180
2.6164	2.8328	2.6990	2.7501	-7.2742	-7.2685	0.6180
2.6164	2.7501	2.6675	2.6990	-7.2697	-7.2742	0.6180
2.6675	2.7501	2.6990	2.7185	-7.2742	-7.2739	0.6180
2.6675	2.7185	2.6870	2.6990	-7.2732	-7.2742	0.6179
2.6870	2.7185	2.6990	2.7065	-7.2742	-7.2744	0.6181
2.6990	2.7185	2.7065	2.7111	-7.2744	-7.2743	0.6176
2.6990	2.7111	2.7037	2.7065	-7.2743	-7.2744	0.6190
2.7037	2.7111	2.7065	2.7083	-7.2744	-7.2743	0.6153
2.7037	2.7083	2.7055	2.7065	-7.2744	-7.2744	0.6250
2.7055	2.7083	2.7065	2.7074	-7.2744	-7.2744	0.6000

$\mathbf{x} = 2.7067$ $\mathbf{y} = -7.2744$ $\boldsymbol{\varepsilon} = 0.001$

Метод Парабол

x_1	x_2	x_3	fx_1	fx_2	fx_3	$\overline{x}$	$f\overline{x}$	$ratio$
0.0000	0.3142	6.2832	1.0000	0.3135	18.8496	1.4547	-3.8169	
0.3142	1.4547	6.2832	0.3135	-3.8169	18.8496	2.1842	-6.5249	0.95
1.4547	2.1842	6.2832	-3.8169	-6.5249	18.8496	2.7245	-7.2734	0.8089
2.1842	2.7245	6.2832	-6.5249	-7.2734	18.8496	2.7797	-7.2578	0.8489
2.1842	2.7245	2.7797	-6.5249	-7.2734	-7.2578	2.7017	-7.2743	0.1452
2.1842	2.7017	2.7245	-6.5249	-7.2743	-7.2734	2.7057	-7.2744	0.9073
2.7017	2.7057	2.7245	-7.2743	-7.2744	-7.2734	2.7065	-7.2744	0.0421

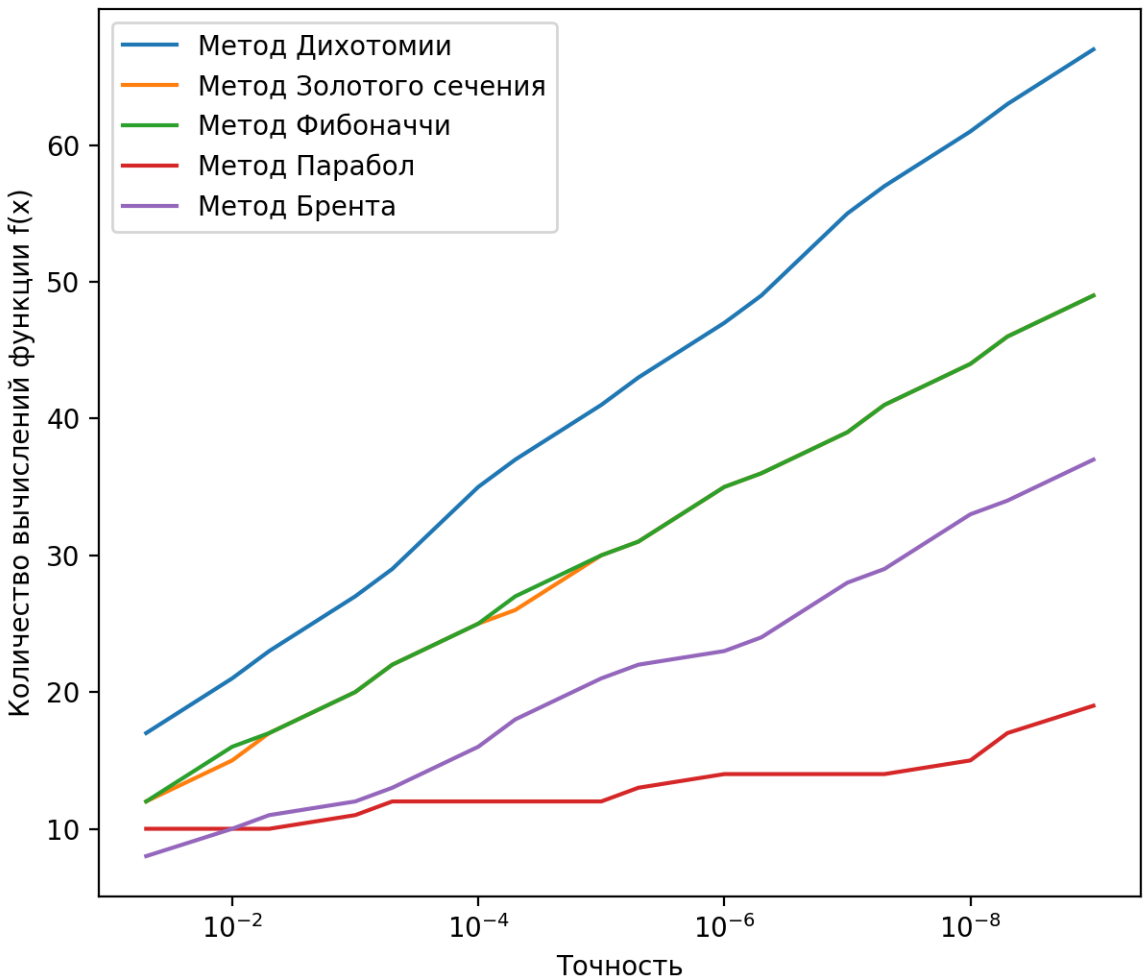
$\mathbf{x} = 2.7064$ $\mathbf{y} = -7.2743$

Метод Брента

l	r	x	w	v	fx	fw	fv	$ratio$
0.0000	6.2832	2.4000	2.4000	2.4000	-7.0034	-7.0034	-7.0034	
0.0000	3.8832	2.4000	3.8832	2.4000	-7.0034	-2.6461	-7.0034	0.6180
1.4833	3.8832	2.4000	1.4833	3.8832	-7.0034	-3.9390	-2.6461	0.6180
2.4000	3.8832	2.5803	2.4000	1.4833	-7.2268	-7.0034	-3.9390	0.6180
2.5803	3.8832	2.8130	2.5803	2.4000	-7.2391	-7.2268	-7.0034	0.8784
2.5803	3.2218	2.8130	2.5803	2.4000	-7.2391	-7.2268	-7.0034	0.4923
2.5803	2.8130	2.7059	2.8130	2.5803	-7.2744	-7.2391	-7.2268	0.3627
2.7031	2.8130	2.7059	2.7031	2.8130	-7.2744	-7.2743	-7.2391	0.4725
2.7031	2.7468	2.7059	2.7031	2.7468	-7.2744	-7.2743	-7.2694	0.3977
2.7031	2.7215	2.7059	2.7031	2.7215	-7.2744	-7.2743	-7.2737	0.4216
2.7031	2.7119	2.7059	2.7031	2.7119	-7.2744	-7.2743	-7.2743	0.4760

$\mathbf{x} = 2.7063$ $\mathbf{y} = -7.2744$ $\boldsymbol{\varepsilon} = 0.001$

3 Зависимость количества вычисления от точности



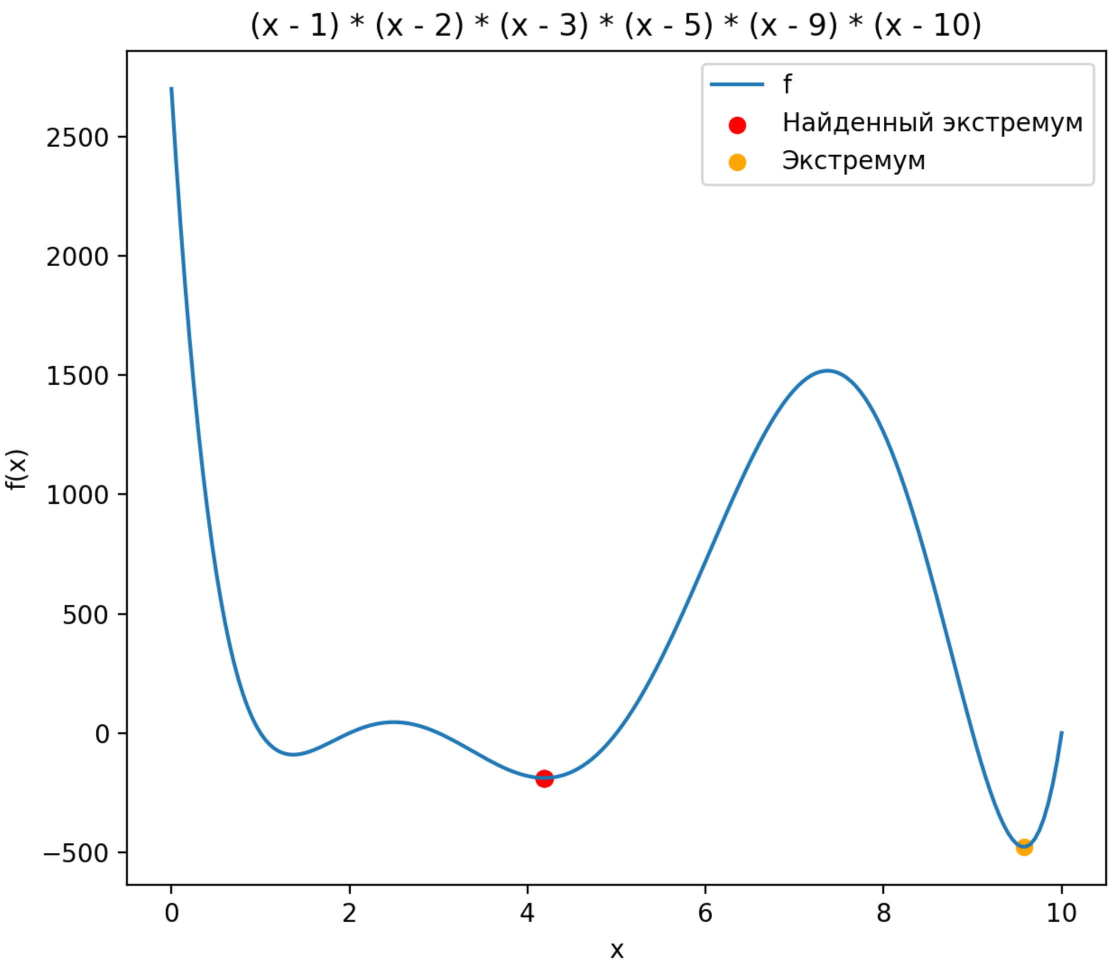
4 Тестирование на многомодальных функциях

Запустим методы на функции $f(x)$

$$f(x) = 2700 - 6060x + 4879x^2 - 1830x^3 + 340x^4 - 30x^5 + x^6 = (x-1) \cdot (x-2) \cdot (x-3) \cdot (x-5) \cdot (x-9) \cdot (x-10)$$

Результаты

Название	x	y
Метод Дихотомии	4.1888	-188.1868
Метод Золотого Сечения	4.1892	-188.1868
Метод Фибоначчи	4.1886	-188.1868
Метод Парабол	4.1882	-188.1869
Метод Брента	4.1888	-188.1868
Аналитическое решение	9.5756	-477.4996



Исходя из полученных результатов, делаем ожидаемый вывод, что не всегда будет найден глобальный экстремум, собственно, данные методы оптимизации и не гарантируют корректности на многомодальных функциях.

5 Сравнение методов и вывод

- Ожидалось, что наименьшее количество вычислений потребуется для Метода Брента, однако методу парабол потребовалось меньше всего вычислений, так как наша функция на заданном промежутке ведёт себя примерно как квадратичная
- Метод Брента – второй по количеству требуемых вычислений, требует весьма меньше вычислений чем остальные методы
- Метод Золотого Сечения и Метод Фибоначчи показали практически идентичные результаты и требуемые количества вычислений функции, особенно при ε требующих большее кол-во вычислений. Так же, им потребовалось вычислений меньше, чем методу Дихотомии.
- Метод Дихотомии, как и ожидалось, сходится медленнее и требует больше вычислений. Это обуславливается его простотой.
- Данные методы нельзя использовать для поиска глобального экстремума многомодальных функций (см. Пункт 4)

6 Программный код

Ссылка на репозиторий на github.com

```java
public class OptimizationMethodResult {
    private final Point extremum;
    private final int iterationCount;

    public OptimizationMethodResult(Point extremum, int iterationCount) {
        this.extremum = extremum;
        this.iterationCount = iterationCount;
    }

    public Point getExtremum() {
        return extremum;
    }

    public int getIterationCount() {
        return iterationCount;
    }

    @Override
    public String toString() {
        return "OptimizationMethodResult{" +
                "extremum=" + extremum +
                ", iterationCount=" + iterationCount +
                '}';
    }
}
```

Для всех методов были реализованы вспомогательные классы [название метода]Iteration

```java
public interface OptimizationMethodIteration {
    boolean hasNext();
    void next();
    Point getExtremum();
    DoubleFunction getFunction();
    double getLeft();
    double getRight();
    double getEps();
    String toTex();
}

public abstract class AbstractMethodIteration implements OptimizationMethodIteration {
    protected final DoubleFunction function;

    public DoubleFunction getFunction() {
        return function;
    }

    public double getLeft() {
        return left;
    }

    public double getRight() {
        return right;
    }

    public double getEps() {
        return eps;
    }
```

```java
    protected double left;
    protected double right;
    protected final double eps;

    protected AbstractMethodIteration(double left, double right,
        double eps, DoubleFunction function)
    {
        this.function = function;
        this.left = left;
        this.right = right;
        this.eps = eps;
    }

    protected abstract Point getExtremumImpl();

    protected double apply(double x) {
        return function.apply(x);
    }

    public Point getExtremum() {
        if (hasNext()) {
            throw new UnsupportedOperationException("Can't calculate extremum");
        }
        return getExtremumImpl();
    }
}

//Dichotomy iteration
public class DichotomyIteration extends AbstractMethodIteration {
    private final double delta;
    private double x1;
    private double x2;
    private double fx1;
    private double fx2;

    public DichotomyIteration(double left, double right, double eps,
        double delta, DoubleFunction func)
    {
        super(left, right, eps, func);
        this.delta = delta;
        x1 = (right + left - delta) / 2.0;
        x2 = (right + left + delta) / 2.0;
        fx1 = apply(x1);
        fx2 = apply(x2);
    }

    @Override
    public boolean hasNext() {
        return ((right - left) > eps * 2.0);
    }

    @Override
    public void next() {
        if (fx1 <= fx2) {
            right = x2;
        } else {
            left = x1;
        }
        x1 = (right + left - delta) / 2.0;
        x2 = (right + left + delta) / 2.0;
        fx1 = apply(x1);
```

```java
        fx2 = apply(x2);
    }

    public Point getExtremumImpl() {
        double x = (left + right) / 2.0;
        return new Point(x, apply(x));
    }
}

// Golden ration iteration
public class GoldenRatioIteration extends AbstractMethodIteration {
    public final static double TAU = (Math.sqrt(5.0) - 1.0) / 2.0;
    private double x1;
    private double x2;
    private double fx1;
    private double fx2;

    public GoldenRatioIteration(double left, double right, double eps, DoubleFunction func) {
        super(left, right, eps, func);
        this.x1 = left + (1.0 - TAU) * (right - left);
        this.x2 = left + TAU * (right - left);
        this.fx1 = apply(x1);
        this.fx2 = apply(x2);
    }

    @Override
    public boolean hasNext() {
        return ((right - left) > eps * 2.0);
    }

    @Override
    public void next() {
        if (fx1 <= fx2) {
            right = x2;
            double prevX1 = x1;
            x1 = x2 - TAU * (x2 - left);
            x2 = prevX1;
            fx2 = fx1;
            fx1 = apply(x1);
        } else {
            left = x1;
            double prevX2 = x2;
            x2 = x1 + TAU * (right - x1);
            x1 = prevX2;
            fx1 = fx2;
            fx2 = apply(x2);
        }
    }

    public Point getExtremumImpl() {
        double x = (left + right) / 2.0;
        return new Point(x, apply(x));
    }
}

//Fibonacci iteration
public class FibonacciIteration extends AbstractMethodIteration {
    private double x1;
    private double x2;
    private double fx1;
    private double fx2;
    private int k;
```

```java
    private final int n;
    private final double len;

    public FibonacciIteration(double left, double right, double eps, DoubleFunction func) {
        super(left, right, eps, func);
        this.n = FibonacciCalculator.calculateIterationsCount(left, right, eps);
        this.x1 = left + fib(n) / fib(n + 2) * (right - left);
        this.x2 = left + fib(n + 1) / fib(n + 2) * (right - left);
        this.fx1 = apply(x1);
        this.fx2 = apply(x2);
        this.k = 1;
        this.len = right - left;
    }

    @Override
    public boolean hasNext() {
        return (k < n);
    }

    @Override
    public void next() {
        double newLeft, newRight, newX1, newX2, newFx1, newFx2;
        if (fx1 > fx2) {
            newLeft = x1;
            newRight = right;
            newX1 = x2;
            newX2 = newLeft + fib(n - k + 2) / fib(n + 2) * (len);
            newFx1 = fx2;
            newFx2 = apply(newX2);
        } else {
            newLeft = left;
            newRight = x2;
            newX2 = x1;
            newX1 = left + fib(n - k + 1) / fib(n + 2) * (len);
            newFx1 = apply(newX1);
            newFx2 = fx1;
        }
        left = newLeft;
        right = newRight;
        x1 = newX1;
        x2 = newX2;
        fx1 = newFx1;
        fx2 = newFx2;
        k++;
    }

    public Point getExtremumImpl() {
        double x = (left + right) / 2.0;
        return new Point(x, apply(x));
    }
}

//Parabola iteration
public class ParabolaIteration extends AbstractMethodIteration {
    private final static int INITIAL_POINT_SEARCH_STEPS = 20;
    private double x1;
    private double x2;
    private double x3;
    private double fx1;
    private double fx2;
    private double fx3;
    private double pMinX;
```

```java
private double fOfMinX;
private DoubleFunction approximationParabola;
private boolean isFirst;
private double prevPMinX;

public static Point findParabolaMin(double x1, double x2, double x3, double fx1, double fx2,
     double fx3,
                                DoubleFunction func) {
    double x = findParabolaMinX(x1, x2, x3, fx1, fx2, fx3);
    double y = func.apply(x);
    return new Point(x, y);
}

public static double findParabolaMinX(double x1, double x2, double x3, double fx1, double fx2,
     double fx3) {
    double a1 = (fx2 - fx1) / (x2 - x1),
            a2 = ( (fx3 - fx1) / (x3 - x1) - (fx2 - fx1) / (x2 - x1) ) / (x3 - x2);
    return (x1 + x2 - a1 / a2) / 2;
}

private Point findParabolaMin() {
    return findParabolaMin(x1, x2, x3, fx1, fx2, fx3, function);
}

private static int compare(double x, double y) {
    return Double.compare(x, y);
}

private int compareWithEps(double x, double y) {
    if (Math.abs(x - y) < eps) {
        return 0;
    }
    if (x - y <= -eps) {
        return -1;
    } else {   /*if (x - y >= eps) {*/
        return 1;
    }
}

private double findInitialPoint(double l, double r, double fx1, double fx3) {
    double x2;
    for (int i = 0; i < INITIAL_POINT_SEARCH_STEPS; i++) {
        x2 = l + ((r - l) / INITIAL_POINT_SEARCH_STEPS) * (i + 1);
        double fx2 = apply(x2);
        if (compareWithEps(fx2, fx1) <= 0 && compareWithEps(fx2, fx3) <= 0) {
            return x2;
        }
    }
    throw new RuntimeException("Can't find initial x2 value");
}

public DoubleFunction getApproximationParabola() {
    return approximationParabola;
}

public double getpMinX() {
    return pMinX;
}

public double getFofMinX() {
    return fOfMinX;
}
```

```java
private static class Parabola {
    private final double a, b, c;

    private Parabola(double a, double b, double c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    public DoubleFunction toDoubleFunction() {
        return x -> a * x * x + b * x + c;
    }
}

public ParabolaIteration(double left, double right, double eps, DoubleFunction func) {
    super(left, right, eps, func);
    this.isFirst = true;
    this.x1 = left;
    this.x3 = right;
    this.fx1 = apply(x1);
    this.fx3 = apply(x3);
    this.x2 = findInitialPoint(left, right, fx1, fx3);
    this.fx2 = apply(x2);
    Parabola parabola = findApproximationParabola();
    Point pMin = findParabolaMin();
    this.pMinX = pMin.getX();
    this.fOfMinX = pMin.getY();
    this.approximationParabola = parabola.toDoubleFunction();
    this.prevPMinX = Double.NaN;
}

public static Parabola findApproximationParabola(double x1, double x2, double x3, double fx1,
     double fx2, double fx3) {
    double a0 = fx1, a1 = (fx2 - fx1) / (x2 - x1),
            a2 = ( (fx3 - fx1) / (x3 - x1) - (fx2 - fx1) / (x2 - x1) ) / (x3 - x2);
    return new Parabola(a2, a1 + a2 * (-x2) + a2 * (-x1), a0 + a1 * (-x1) + a2 * (-x1) * (-x2));
}

private Parabola findApproximationParabola() {
    return findApproximationParabola(x1, x2, x3, fx1, fx2, fx3);
}

@Override
public boolean hasNext() { return isFirst || compareWithEps(prevPMinX, pMinX) != 0; }

@Override
protected Point getExtremumImpl() {
    return new Point(getpMinX(), getFofMinX());
}

@Override
public void next() {
    double nx1 = x1, nx2 = x2, nx3 = x3;
    double nfx1 = fx1, nfx2 = fx2, nfx3 = fx3;
    if (compare(x1, pMinX) <= 0 && compare(pMinX, x2) < 0) { // x1 <= pMinX < x2
        if (compare(fOfMinX, fx2) >= 0) { // pMin >= f(x2)
            nx1 = pMinX;
            nfx1 = fOfMinX;
        } else { // pMin < f(x2)
            nx3 = x2;
            nfx3 = fx2;
```

```java
                nx2 = pMinX;
                nfx2 = fOfMinX;
            }
        } else if (compare(x2, pMinX) <= 0 && compare(pMinX, x3) <= 0) { // x2 <= pMinX <= x3
            if (compare(fx2, fOfMinX) >= 0) { // f(x2) >= pMin
                nx1 = x2;
                nfx1 = fx2;
                nx2 = pMinX;
                nfx2 = fOfMinX;
            } else { // f(x2) < pMin
                nx3 = pMinX;
                nfx3 = fOfMinX;
            }
        }
        left = nx1;
        right = nx3;
        x1 = nx1;
        x2 = nx2;
        x3 = nx3;
        fx3 = nfx3;
        fx2 = nfx2;
        fx1 = nfx1;
        isFirst = false;
        prevPMinX = pMinX;
        Parabola parabola = findApproximationParabola();
        Point pMin = findParabolaMin();
        pMinX = pMin.getX();
        fOfMinX = pMin.getY();
        approximationParabola = parabola.toDoubleFunction();
    }
}

//Brent iteration
public class BrentIteration extends AbstractMethodIteration {
    private static final double K = (3. - Math.sqrt(5.)) / 2.;
    private double x;
    private double w;
    private double v;
    private double fx;
    private double fw;
    private double fv;
    private double d;
    private double e;

    public BrentIteration(double left, double right, double eps, DoubleFunction function) {
        super(left, right, eps, function);
        x = w = v = left + K * (right - left);
        fx = fw = fv = function.apply(x);
        d = e = right - left;
    }

    @Override
    protected Point getExtremumImpl() {
        return new Point(x, apply(x));
    }

    @Override
    public boolean hasNext() {
        double tol = tol(x);
        return !(Math.abs(x - (left + right) / 2.0) + (right - left) / 2.0 < 2 * tol + eps);
    }
```

```java
private static boolean different(double a, double b, double c, double eps) {
    return Math.abs(a - b) > eps && Math.abs(a - c) > eps && Math.abs(c - b) > eps;
}

private double tol(double x) {
    return eps * Math.abs(x) + eps / 10.0;
}

@Override
public void next() {
    double tol = tol(x);
    double newE = d;
    boolean accepted = false;
    double u = 0.0;
    if (different(x, w, v, eps) && different(fx, fw, fv, eps)) {
        Point point = ParabolaIteration.findParabolaMin(x, w, v, fx, fw, fv, function);
        u = point.getX();
        if (u >= left && u <= right && Math.abs(u - x) < e / 2.0) {
            accepted = true;
        } else if (u - left < 2.0 * tol || right - u < 2.0 * tol) {
            u = x - Math.signum(x - (left + right) / 2.0) * tol;
            accepted = true;
        }
    }
    if (!accepted) {
        if (x < (left + right) / 2.0) {
            u = x + K * (right - x);
            newE = right - x;
        } else {
            u = x - K * (x - left);
            newE = x - left;
        }
    }
    if (Math.abs(u - x) < tol) {
        u = x + Math.signum(u - x) * tol;
    }
    double newD = Math.abs(u - x);
    double fu = apply(u);
    double newLeft = left;
    double newRight = right;
    if (fu <= fx) {
        if (u >= x) {
            newLeft = x;
        } else {
            newRight = x;
        }
        v = w;
        w = x;
        x = u;
        fv = fw;
        fw = fx;
        fx = fu;
    } else {
        if (u >= x) {
            newRight = u;
        } else {
            newLeft = u;
        }
        if (fu <= fw || Math.abs(w - x) < eps) {
            v = w;
            w = u;
            fv = fw;
```

```
                fw = fu;
            } else if (fu <= fv || Math.abs(v - x) < eps || Math.abs(v - w) < eps) {
                v = u;
                fv = fu;
            }
        }
        left = newLeft;
        right = newRight;
        d = newD;
        e = newE;
    }
}
```

Методы были реализованы при помощи соответствующих вспомогательных классов.

```
public abstract class AbstractOptimizationMethod {
    private final OptimizationMethodIteration iteration;

    AbstractOptimizationMethod(OptimizationMethodIteration iteration) {
        this.iteration = iteration;
    }

    public OptimizationMethodResult run() {
        int counter = 1;
        double lastLen = iteration.getRight() - iteration.getLeft();
        while (iteration.hasNext()) {
            iteration.next();
            lastLen = iteration.getRight() - iteration.getLeft();
            counter++;
        }
        return new OptimizationMethodResult(iteration.getExtremum(), counter);
    }
}

public class DichotomyMethod extends AbstractOptimizationMethod {
    public DichotomyMethod(double left, double right, double eps, double delta, DoubleFunction
         function) {
        super(new DichotomyIteration(left, right, eps, delta, function));
    }
}

public class GoldenRatioMethod extends AbstractOptimizationMethod {

    public GoldenRatioMethod(double left, double right, double eps, DoubleFunction function) {
        super(new GoldenRatioIteration(left, right, eps, function));
    }
}

public class FibonacciMethod extends AbstractOptimizationMethod {

    public FibonacciMethod(double left, double right, double eps, DoubleFunction function) {
        super(new FibonacciIteration(left, right, eps, function));
    }
}

public class ParabolaMethod extends AbstractOptimizationMethod {
    public ParabolaMethod(double left, double right, double eps, DoubleFunction function) {
        super(new ParabolaIteration(left, right, eps, function));
    }
}
```

```java
public class BrentMethod extends AbstractOptimizationMethod {
    public BrentMethod(double left, double right, double eps, DoubleFunction function) {
        super(new BrentIteration(left, right, eps, function));
    }
}
```