Национальный исследовательский университет ИТМО
Прикладная математика и информатика

Методы оптимизации

Отчет по лабораторной работе №1
“Алгоритмы одномерной минимизации функции”

Вариант 1

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, 3 марта 2021 г.

1 Цель

Реализовать алгоритмы одномерной минимизации функции:

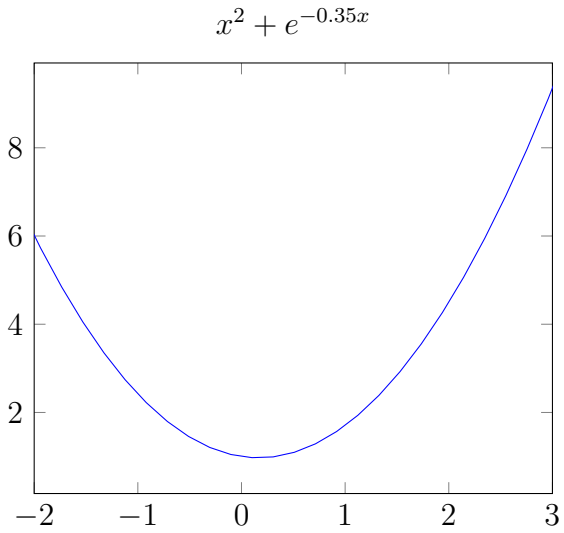
- Метод дихотомии
- Метод золотого сечения
- Метод Фибоначчи
- Метод парабол
- Комбинированный метод Брента

Протестировать алгоритмы на $f(x) = x^2 + e^{-0.35x}$ в интервале $[-2; 3]$ и других функциях, сравнить методы друг с другом.

2 Ход работы

2.1 Аналитическое решение

$$
\begin{aligned}
0 = f'(x) &= 2x - 0.35e^{-0.35x} \\
2x &= 0.35e^{-0.35x} \\
800 \cdot 0.35x &= 49\frac{1}{e^{0.35x}} \\
0.35x &:= W(z) \\
800 \cdot W(z) &= 49\frac{W(z)}{z} \\
\frac{49}{800} &= z \\
0.35x &= W\left(\frac{800}{49}\right) \\
x &= \frac{20}{7} \cdot W\left(\frac{800}{49}\right)
\end{aligned}
$$



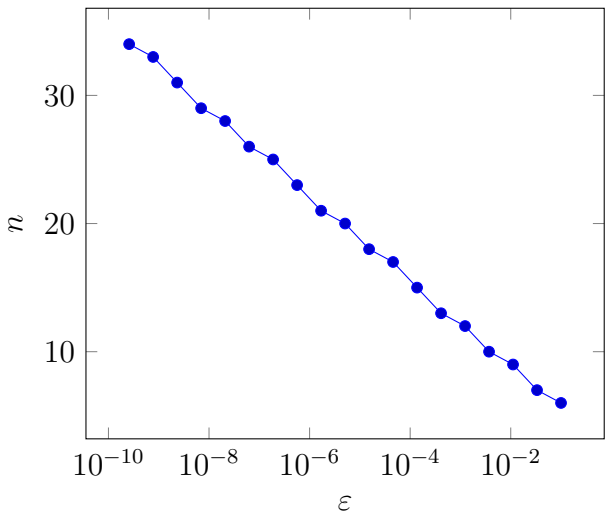
W-функция Ламберта не может быть выражена в элементарных функциях, поэтому аналитическое решение приближенное:

$$
\begin{cases}
x_{\min} \approx 0.16517 \\
y_{\min} \approx 0.9711
\end{cases}
$$

2.2 Метод дихотомии

N	a	b	% длины предыдущего отрезка	x_1	$f(x_1)$	x_2	$f(x_2)$
1	-2	3	100	0.5	1.08946	0.5	1.08946
2	-2	0.5	50	-0.75	1.86268	-0.75	1.86268
3	-0.75	0.5	50	-0.125	1.06035	-0.125	1.06035
4	-0.125	0.5	50	0.1875	0.971638	0.1875	0.971638
5	-0.125	0.1875	50	0.03125	0.990099	0.0312501	0.990099
6	0.03125	0.1875	50	0.109375	0.974405	0.109375	0.974405
7	0.109375	0.1875	50	0.148437	0.971407	0.148438	0.971407
8	0.148437	0.1875	50.0001	0.167969	0.971119	0.167969	0.971119
9	0.148437	0.167969	50.0001	0.158203	0.971162	0.158203	0.971162
10	0.158203	0.167969	50.0003	0.163086	0.971115	0.163086	0.971115
11	0.163086	0.167969	50.0005	0.165527	0.971111	0.165527	0.971111
12	0.163086	0.165527	50.001	0.164307	0.971112	0.164307	0.971112
13	0.164307	0.165527	50.002	0.164917	0.971111	0.164917	0.971111
14	0.164917	0.165527	50.0041	0.165222	0.971111	0.165222	0.971111
15	0.164917	0.165222	50.0082	0.16507	0.971111	0.16507	0.971111
16	0.16507	0.165222	50.0164	0.165146	0.971111	0.165146	0.971111
17	0.165146	0.165222	50.0327	0.165184	0.971111	0.165184	0.971111
18	0.165146	0.165184	50.0655	0.165165	0.971111	0.165165	0.971111
19	0.165165	0.165184	50.1307	0.165174	0.971111	0.165175	0.971111
20	0.165165	0.165175	50.2608	0.16517	0.971111	0.16517	0.971111
21	0.16517	0.165175	50.5188	0.165172	0.971111	0.165172	0.971111
22	0.16517	0.165172	51.027	0.165171	0.971111	0.165171	0.971111
23	0.16517	0.165171	52.0127	0.16517	0.971111	0.16517	0.971111

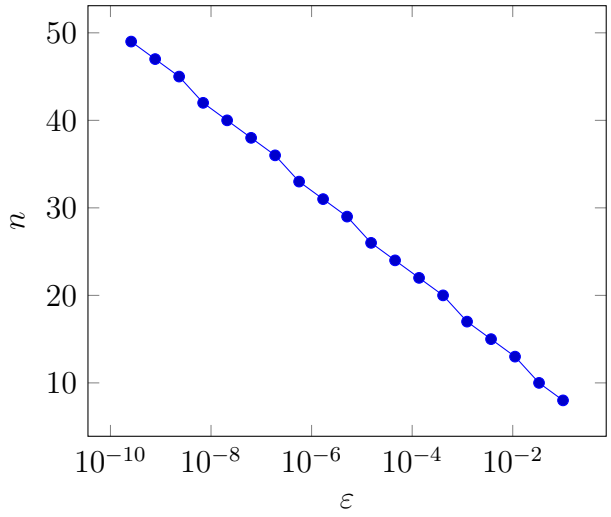
График зависимости количества измерений n от логарифма задаваемой точности ε



2.3 Метод золотого сечения

N	a	b	% длины предыдущего отрезка	x_1	$f(x_1)$	x_2	$f(x_2)$
1	-2	3	100	-0.0901699	1.04019	1.09017	1.87127
2	-2	1.09017	61.8034	-0.81966	2.00411	-0.0901699	1.04019
3	-0.81966	1.09017	61.8034	-0.0901699	1.04019	0.36068	1.0115
4	-0.0901699	1.09017	61.8034	0.36068	1.0115	0.63932	1.20824
5	-0.0901699	0.63932	61.8034	0.188471	0.971685	0.36068	1.0115
6	-0.0901699	0.36068	61.8034	0.0820393	0.978425	0.188471	0.971685
7	0.0820393	0.36068	61.8034	0.188471	0.971685	0.254249	0.9795
8	0.0820393	0.254249	61.8034	0.147817	0.971429	0.188471	0.971685
9	0.0820393	0.188471	61.8034	0.122692	0.97302	0.147817	0.971429
10	0.122692	0.188471	61.8034	0.147817	0.971429	0.163346	0.971114
11	0.147817	0.188471	61.8034	0.163346	0.971114	0.172942	0.971175
12	0.147817	0.172942	61.8034	0.157414	0.971174	0.163346	0.971114
13	0.157414	0.172942	61.8034	0.163346	0.971114	0.167011	0.971114
14	0.157414	0.167011	61.8034	0.16108	0.971129	0.163346	0.971114
15	0.16108	0.167011	61.8034	0.163346	0.971114	0.164746	0.971111
16	0.163346	0.167011	61.8034	0.164746	0.971111	0.165611	0.971111
17	0.163346	0.165611	61.8034	0.164211	0.971112	0.164746	0.971111
18	0.164211	0.165611	61.8034	0.164746	0.971111	0.165076	0.971111
19	0.164746	0.165611	61.8034	0.165076	0.971111	0.16528	0.971111
20	0.164746	0.16528	61.8034	0.16495	0.971111	0.165076	0.971111
21	0.16495	0.16528	61.8034	0.165076	0.971111	0.165154	0.971111
22	0.165076	0.16528	61.8034	0.165154	0.971111	0.165202	0.971111
23	0.165076	0.165202	61.8034	0.165124	0.971111	0.165154	0.971111
24	0.165124	0.165202	61.8034	0.165154	0.971111	0.165173	0.971111
25	0.165154	0.165202	61.8034	0.165173	0.971111	0.165184	0.971111
26	0.165154	0.165184	61.8034	0.165166	0.971111	0.165173	0.971111
27	0.165166	0.165184	61.8034	0.165173	0.971111	0.165177	0.971111
28	0.165166	0.165177	61.8034	0.16517	0.971111	0.165173	0.971111
29	0.165166	0.165173	61.8034	0.165168	0.971111	0.16517	0.971111
30	0.165168	0.165173	61.8034	0.16517	0.971111	0.165171	0.971111
31	0.165168	0.165171	61.8034	0.165169	0.971111	0.16517	0.971111
32	0.165169	0.165171	61.8034	0.16517	0.971111	0.16517	0.971111

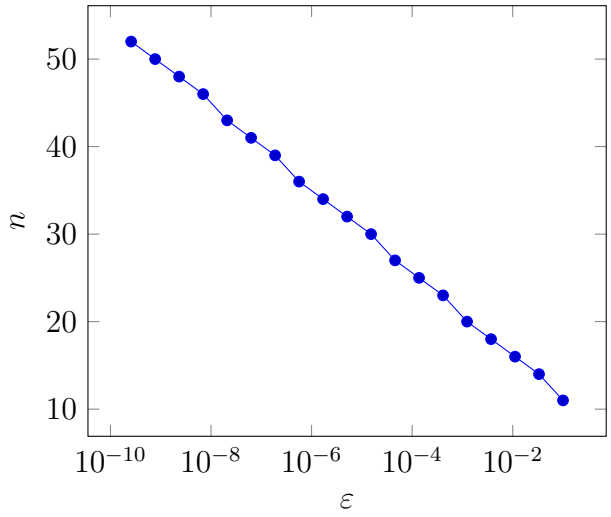
График зависимости количества измерений n от логарифма задаваемой точности ε



2.4 Метод Фибоначчи

N	a	b	% длины предыдущего отрезка	x_1	$f(x_1)$	x_2	$f(x_2)$
1	-2	3	100	-0.0901699	1.04019	1.09017	1.87127
2	-2	1.09017	61.8034	-0.81966	2.00411	-0.0901699	1.04019
3	-0.81966	1.09017	61.8034	-0.0901699	1.04019	0.36068	1.0115
4	-0.0901699	1.09017	61.8034	0.36068	1.0115	0.63932	1.20824
5	-0.0901699	0.63932	61.8034	0.188471	0.971685	0.36068	1.0115
6	-0.0901699	0.36068	61.8034	0.0820393	0.978425	0.188471	0.971685
7	0.0820393	0.36068	61.8034	0.188471	0.971685	0.254249	0.9795
8	0.0820393	0.254249	61.8034	0.147817	0.971429	0.188471	0.971685
9	0.0820393	0.188471	61.8034	0.122692	0.97302	0.147817	0.971429
10	0.122692	0.188471	61.8034	0.147817	0.971429	0.163346	0.971114
11	0.147817	0.188471	61.8034	0.163346	0.971114	0.172942	0.971175
12	0.147817	0.172942	61.8034	0.157414	0.971174	0.163346	0.971114
13	0.157414	0.172942	61.8034	0.163346	0.971114	0.167011	0.971114
14	0.157414	0.167011	61.8034	0.16108	0.971129	0.163346	0.971114
15	0.16108	0.167011	61.8034	0.163346	0.971114	0.164746	0.971111
16	0.163346	0.167011	61.8034	0.164746	0.971111	0.165611	0.971111
17	0.163346	0.165611	61.8034	0.164211	0.971112	0.164746	0.971111
18	0.164211	0.165611	61.8034	0.164746	0.971111	0.165076	0.971111
19	0.164746	0.165611	61.8034	0.165076	0.971111	0.16528	0.971111
20	0.164746	0.16528	61.8034	0.16495	0.971111	0.165076	0.971111
21	0.16495	0.16528	61.8034	0.165076	0.971111	0.165154	0.971111
22	0.165076	0.16528	61.8034	0.165154	0.971111	0.165202	0.971111
23	0.165076	0.165202	61.8034	0.165124	0.971111	0.165154	0.971111
24	0.165124	0.165202	61.8034	0.165154	0.971111	0.165173	0.971111
25	0.165154	0.165202	61.8033	0.165173	0.971111	0.165184	0.971111
26	0.165154	0.165184	61.8037	0.165166	0.971111	0.165173	0.971111
27	0.165166	0.165184	61.8026	0.165173	0.971111	0.165177	0.971111
28	0.165166	0.165177	61.8056	0.16517	0.971111	0.165173	0.971111
29	0.165166	0.165173	61.7978	0.165168	0.971111	0.16517	0.971111
30	0.165168	0.165173	61.8182	0.16517	0.971111	0.165171	0.971111
31	0.165168	0.165171	61.7647	0.165169	0.971111	0.16517	0.971111
32	0.165169	0.165171	61.9048	0.16517	0.971111	0.16517	0.971111
33	0.16517	0.165171	61.5385	0.16517	0.971111	0.165171	0.971111
34	0.16517	0.165171	62.5	0.16517	0.971111	0.16517	0.971111
35	0.16517	0.16517	60	0.16517	0.971111	0.16517	0.971111

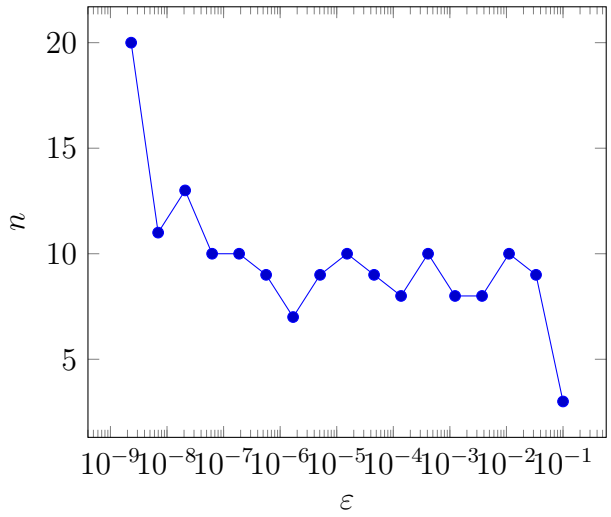
График зависимости количества измерений n от логарифма задаваемой точности ε



2.5 Метод парабол

N	a	b	% длины предыдущего отрезка	x_1	$f(x_1)$	x_2	$f(x_2)$	x_3	$f(x_3)$
1	-2	3	100	-2	6.01375	2.20094	5.30699	3	9.34994
2	-2	2.20094	84.0188	-2	6.01375	0.180922	0.971373	2.20094	5.30699
3	-2	0.180922	51.9151	-2	6.01375	0.179732	0.971335	0.180922	0.971373
4	-2	0.179732	99.9454	-2	6.01375	0.16542	0.971111	0.179732	0.971335
5	-2	0.16542	99.3434	-2	6.01375	0.165293	0.971111	0.16542	0.971111
6	-2	0.165293	99.9941	-2	6.01375	0.165173	0.971111	0.165293	0.971111
7	-2	0.165173	99.9945	-2	6.01375	0.165171	0.971111	0.165173	0.971111
8	-2	0.165171	99.9999	-2	6.01375	0.16517	0.971111	0.165171	0.971111
9	-2	0.16517	100	-2	6.01375	0.16517	0.971111	0.16517	0.971111
10	-2	0.16517	100	-2	6.01375	0.16517	0.971111	0.16517	0.971111

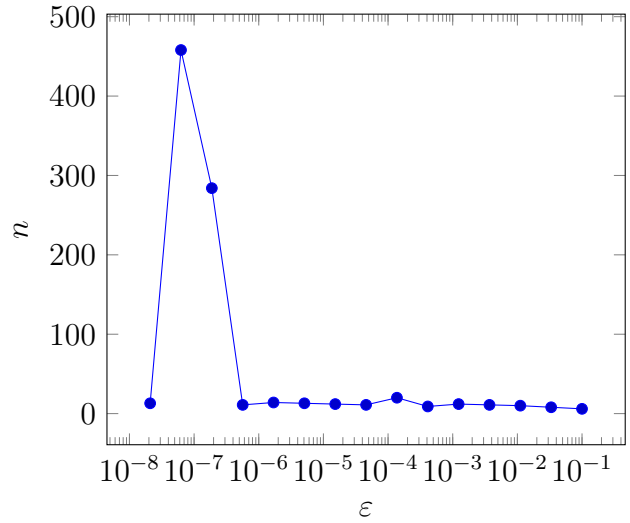
График зависимости количества измерений n от логарифма задаваемой точности ε



2.6 Комбинированный метод Брента

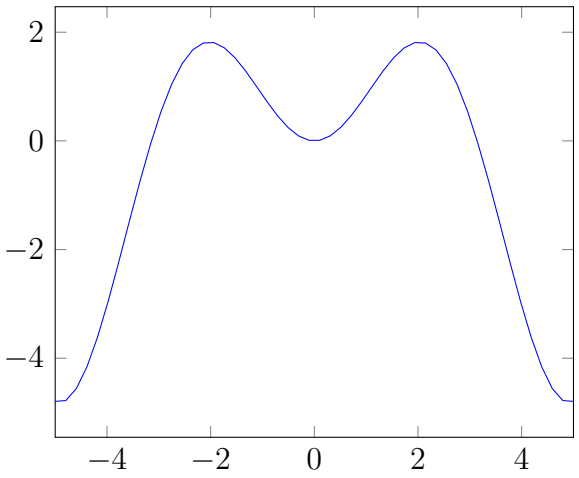
N	a	b	% длины предыдущего отрезка	u	$f(u)$	w	$f(w)$	x	$f(x)$
1	-2	3	100	1.45492	2.71774	0.5	1.08946	0.5	1.08946
2	-2	1.45492	69.0983	-1.63525	4.44647	1.45492	2.71774	0.5	1.08946
3	-1.63525	1.45492	89.4427	0.173569	0.971185	1.45492	2.71774	0.5	1.08946
4	-1.63525	0.5	69.0983	0.163923	0.971113	0.5	1.08946	0.173569	0.971185
5	-1.63525	0.173569	84.7123	0.165163	0.971111	0.173569	0.971185	0.163923	0.971113
6	0.163923	0.173569	0.53329	0.16517	0.971111	0.163923	0.971113	0.165163	0.971111
7	0.165163	0.173569	87.1423	0.16517	0.971111	0.165163	0.971111	0.16517	0.971111
8	0.165163	0.16517	0.0883378	0.165167	0.971111	0.16517	0.971111	0.16517	0.971111
9	0.165167	0.16517	38.0239	0.165169	0.971111	0.16517	0.971111	0.16517	0.971111

График зависимости количества измерений n от логарифма задаваемой точности ε



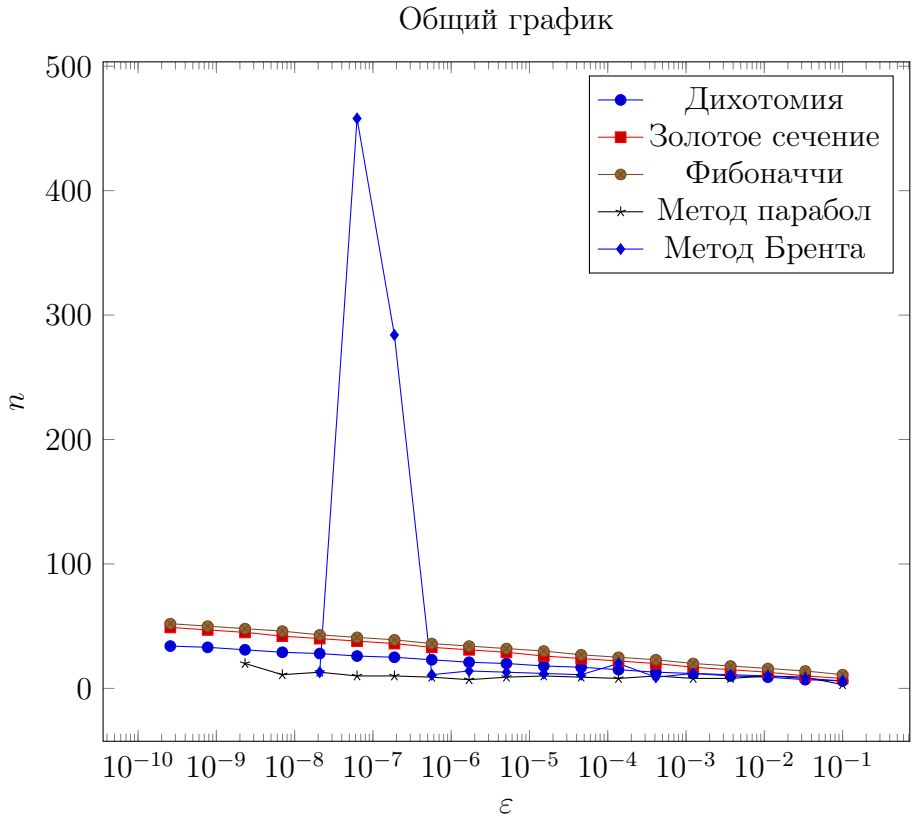
3 Тестирование на многомодальных функциях

3.1 $f(x) = \sin(x) \cdot x$



	$f(x) = \sin(x) \cdot x$	
Верный ответ	4.91318	
Дихотомия	2.2594e-16	
Золотое сечение	2.80886e-16	
Фибоначчи	2.2594e-16	
Параболы	4.91318	
Метод Брента	4.91318	

4 Выводы



Для каждого метода $\varepsilon = 10^{-6}$

Из рассмотренных алгоритмов быстрее всего сходится комбинированный алгоритм Брента, т.к. он сочетает в себе преимущества метода парабол и золотого сечения — квадратичная сходимость в окрестности решения и гарантированно линейная сходимость вне окрестности.

Рассмотренные алгоритмы не способны выполнять нахождение глобального многомодальных функций в общем случае, но находят локальный.

5 Исходный код

5.1 Рассматриваемый отрезок

```hpp
#pragma once

#include <map>
#include <optional>
#include <string>
#include <utility>
#include <vector>

namespace lab {
    /**
     * Отрезок [`start`, `end`]
     * (возможно) ответ `ans` для данного отрезка
     * (возможно) точка `mid`, по которой проходит аппроксимирующая парабола
     */
    class Segment {
      public:
        Segment(double start, double end);
        Segment(double start, double mid, double end);

        double get_start();
        double get_end();
        std::optional<double> get_ans();
        std::optional<double> get_mid();

        void set_ans(double ans);
        void set_mid(double mid);

        std::map<std::string, std::pair<double, double>> saved_points;

      private:
        double start;
        std::optional<double> mid;
        double end;
        std::optional<double> ans;
    };
}  // namespace lab
```

```cpp
#include "lab/segment.h"

using namespace lab;

Segment::Segment(double start, double end) : start(start), end(end) {}

Segment::Segment(double start, double mid, double end)
    : start(start), mid(mid), end(end) {}

double Segment::get_start() { return start; }

double Segment::get_end() { return end; }

std::optional<double> Segment::get_ans() { return ans; }

std::optional<double> Segment::get_mid() { return mid; }

```

```cpp
void Segment::set_ans(double ans) { this->ans = ans; }

void Segment::set_mid(double mid) { this->mid = mid; }
```

5.2 Общий класс оптимизаторов

```cpp
#pragma once

#include <functional>
#include <memory>
#include <ostream>
#include <vector>

#include "lab/segment.h"

namespace lab {

    using func = std::function<double(double)>;

    enum class Optimizers {
        DICHOTOMY,
        GOLDEN_RATIO,
        FIBONACCI,
        PARABOLA,
        BRENT
    };
    static std::unordered_map<std::string, Optimizers> const optimizers_table
        = {{"Дихотомия", Optimizers::DICHOTOMY},
           {"Золотое сечение", Optimizers::GOLDEN_RATIO},
           {"Фиббоначи", Optimizers::FIBONACCI},
           {"Параболы", Optimizers::PARABOLA},
           {"Брент", Optimizers::BRENT}};
    /**
     * Абстрактный класс оптимизатора
     */
    class Optimizer {
      public:
        /**
         * @param optimized_function Оптимизируемая функция
         * @param epsilon Необходимая точность ответа
         * @param start Левая граница отрезка, на котором происходит оптимизация
         * @param end Правая граница отрезка, на котором происходит оптимизация
         * @return Локальный минимум
         */
        Optimizer(const func& optimized_function, double epsilon, double start,
                  double end);

        /**
         * Процедура оптимизации
         *
         * @return результат оптимизации
         */
        double optimize();

        /**
         * @return Сегменты, которые рассматривались во время оптимизации
         */
        const std::vector<Segment>& get_segments();
```

```
53
54        protected:
55          /**
56           * @param current_segment Рассматриваемый сегмент
57           * @param epsilon Искомая точность
58           * @return Достигнута ли искомая точность
59           */
60          virtual bool is_done();
61
62          /**
63           * Выполняет один шаг
64           */
65          virtual void step() = 0;
66
67          /**
68           * @return Результат работы метода
69           */
70          virtual double answer();
71
72          void save_segment();
73
74          void unsave_segment();
75
76          func f;
77          Segment segment;
78          double epsilon;
79
80          /**
81           * Число выполненных шагов алгоритма
82           */
83          int steps_count;
84
85        private:
86          std::vector<Segment> calculated_segments;
87      };
88
89  }  // namespace lab
```

```
1   #include "lab/optimizer.h"
2
3   #include <iostream>
4
5   using namespace lab;
6
7   Optimizer::Optimizer(const func& f, const double epsilon, double start,
8                        double end)
9       : f(f), segment(start, end), epsilon(epsilon), steps_count(0) {}
10
11  double Optimizer::optimize() {
12      while (!is_done()) {
13          step();
14          steps_count++;
15      }
16      return answer();
17  }
18
19  const std::vector<Segment>& Optimizer::get_segments() {
20      return calculated_segments;
21  }
```

```
22
23  bool Optimizer::is_done() {
24      return segment.get_end() <= 2 * epsilon + segment.get_start();
25  }
26
27  double Optimizer::answer() {
28      return (segment.get_end() + segment.get_start()) / 2;
29  }
30
31  void Optimizer::save_segment() {
32      segment.set_ans(answer());
33      calculated_segments.push_back(segment);
34  }
```

5.3 Общий класс оптимизаторов, рассматривающих 2 точки

```hpp
#pragma once

#include <cmath>

#include "lab/optimizer.h"

namespace lab {
    /**
     * Общий класс оптимизаторов, которые на каждом шаге рассматривают две точки
     */
    class TwoPoint : public Optimizer {
        using Optimizer::Optimizer;

      protected:
        /**
         * Вычисляет `x1`, `x2`, `fx1` и `fx2`
         */
        void calc_points();

      protected:
        /**
         * Вычисляет первую точку для рассмотрения
         */
        virtual double get_x1(double start, double end) = 0;

        /**
         * Вычисляет вторую точку для рассмотрения
         */
        virtual double get_x2(double start, double end) = 0;

        /**
         * Рассматриваемые в отрезке точки
         */
        double x1, x2;
        /**
         * Значения оптимизируемой функции в рассматриваемых точках
         */
        double fx1, fx2;
    };

}  // namespace lab
```

```cpp
#include "lab/two_point.h"

#include "lab/segment.h"

using namespace lab;

void TwoPoint::calc_points() {
    double start = segment.get_start();
    double end = segment.get_end();
    x1 = get_x1(start, end);
    fx1 = f(x1);
    x2 = get_x2(start, end);
    fx2 = f(x2);
    segment.saved_points["x1"] = {x1, fx1};
    segment.saved_points["x2"] = {x2, fx2};
```

```
16      save_segment();
17  }
```

5.4 Метод дихотомии

```hpp
#pragma once

#include "lab/optimizer.h"
#include "lab/two_point.h"

namespace lab {
    /**
     * Оптимизатор на основе метода дихотомии
     */
    class Dichotomy : public TwoPoint {
        using TwoPoint::TwoPoint;

      protected:
        void step();
        double get_x1(double start, double end);
        double get_x2(double start, double end);

      private:
        double delta = epsilon / 2;
    };

}  // namespace lab
```

```cpp
#include "lab/dichotomy.h"

#include <iostream>

#include "lab/segment.h"

using namespace lab;

void Dichotomy::step() {
    calc_points();
    double new_start = segment.get_start();
    double new_end = segment.get_end();
    if (fx1 <= fx2) {
        new_end = x2;
    } else {
        new_start = x1;
    }
    segment = {new_start, new_end};
}

double Dichotomy::get_x1(double start, double end) {
    return (end + start - delta) / 2;
}

double Dichotomy::get_x2(double start, double end) {
    return (end + start + delta) / 2;
}
```

5.5 Метод золотого сечения

```hpp
#pragma once

#include <cmath>

#include "lab/two_point.h"

namespace lab {
    /**
     * Оптимизатор на основе метода золтого сечения
     */
    class GoldenRatio : public TwoPoint {
      public:
        GoldenRatio(const func& optimized_function, double epsilon,
                    double start, double end);

      protected:
        void step();

      protected:
        double get_x1(double start, double end);
        double get_x2(double start, double end);

      private:
        const double TAU = (std::sqrt(5) - 1) / 2;
    };

}  // namespace lab
```

```cpp
#include "lab/golden_ratio.h"

#include <cmath>

#include "lab/segment.h"

using namespace lab;

GoldenRatio::GoldenRatio(const func& optimized_function, double epsilon,
                         double start, double end)
    : TwoPoint(optimized_function, epsilon, start, end) {
    calc_points();
}

void GoldenRatio::step() {
    double start = segment.get_start();
    double end = segment.get_end();
    segment.saved_points["x1"] = {x1, fx1};
    segment.saved_points["x2"] = {x2, fx2};
    save_segment();
    if (fx1 <= fx2) {
        segment = {start, x2};
        double new_x1 = get_x1(start, x2);
        x2 = x1;
        fx2 = fx1;
        x1 = new_x1;
        fx1 = f(x1);
    } else {
        segment = {x1, end};
```

```
        double new_x2 = get_x2(x1, end);
        x1 = x2;
        fx1 = fx2;
        x2 = new_x2;
        fx2 = f(x2);
    }
}

double GoldenRatio::get_x1(double start, double end) {
    return start + (1 - TAU) * (end - start);
}

double GoldenRatio::get_x2(double start, double end) {
    return start + TAU * (end - start);
}
```

5.6 Метод Фибоначчи

```h
#pragma once

#include "lab/golden_ratio.h"

namespace lab {
    class Fibonacci : public GoldenRatio {
      public:
        Fibonacci(const func& optimized_function, double epsilon, double start,
                  double end);

      protected:
        bool is_done();

      private:
        double get_x1(double start, double end);
        double get_x2(double start, double end);
        /**
         * @return n-тое число Фибоначчи
         */
        double fib(int n);

        /**
         * Начальная левая граница
         */
        double initial_start;
        /**
         * Начальная правая граница
         */
        double initial_end;

        /**
         * Число итераций для получения ответа
         */
        int n;
    };
}  // namespace lab
```

```cpp
#include "lab/fibonacci.h"

#include <cmath>

using namespace lab;

Fibonacci::Fibonacci(const func& optimized_function, double epsilon,
                     double start, double end)
    : GoldenRatio(optimized_function, epsilon, start, end) {
    initial_start = start;
    initial_end = end;
    n = std::ceil(std::log((end - start) / epsilon * std::sqrt(5))
                  / std::log((1 + std::sqrt(5)) / 2));
}

bool Fibonacci::is_done() { return steps_count >= n; }

double Fibonacci::get_x1(double start, double end) {
    return start
           + fib(n - steps_count + 1) / fib(n - steps_count + 3)
```

```
                * (end - start);
}

double Fibonacci::get_x2(double start, double end) {
    return start
           + fib(n - steps_count + 2) / fib(n - steps_count + 3)
                * (end - start);
}

double Fibonacci::fib(int n) {
    return (std::pow((1 + std::sqrt(5)), n) - std::pow((1 - std::sqrt(5)), n))
           / std::pow(2, n) / std::sqrt(5);
}
```

5.7 Метод парабол

```hpp
#pragma once

#include "lab/optimizer.h"

namespace lab {
    /**
     * Оптимизатор на основе метода парабол
     */
    class Parabola : public Optimizer {
      public:
        Parabola(const func& optimized_function, double epsilon, double start,
                 double end);

      protected:
        void step();
        double answer();

      private:
        double f_start, f_mid, f_end;
    };
}  // namespace lab
```

```cpp
#include "lab/parabola.h"

#include <fmt/core.h>

using namespace lab;

Parabola::Parabola(const func& optimized_function, double epsilon, double start,
                   double end)
    : Optimizer(optimized_function, epsilon, start, end) {
    f_start = f(start);
    f_end = f(end);
    double mid;
    do {
        mid = ((double)rand()) / RAND_MAX * (end - start) + start;
        f_mid = f(mid);
    } while (f_start < f_mid || f_mid > f_end);
    segment.set_mid(mid);
}

double Parabola::answer() {
    double start = segment.get_start();
    double mid = *segment.get_mid();
    double end = segment.get_end();
    return mid
           - 0.5
                 * ((mid - start) * (mid - start) * (f_mid - f_end)
                    - (mid - end) * (mid - end) * (f_mid - f_start))
                 / ((mid - start) * (f_mid - f_end)
                    - (mid - end) * (f_mid - f_start));
}

void Parabola::step() {
    double ans = answer();
    double f_ans = f(ans);
    double start = segment.get_start();
```

```
    double mid = *segment.get_mid();
    double end = segment.get_end();

    segment.saved_points["x1"] = {start, f_start};
    segment.saved_points["x2"] = {mid, f_mid};
    segment.saved_points["x3"] = {end, f_end};
    save_segment();

    if (ans < mid) {
        if (f_ans >= f_mid) {
            segment = {ans, mid, end};
            f_start = f_ans;
        } else {
            segment = {start, ans, mid};
            f_end = f_mid;
            f_mid = f_ans;
        }
    } else {
        if (f_ans <= f_mid) {
            segment = {mid, ans, end};
            f_start = f_mid;
            f_mid = f_ans;
        } else {
            segment = {start, mid, ans};
            f_end = f_ans;
        }
    }
}
```

5.8 Комбинированный метод Брента

```
#pragma once

#include <cmath>

#include "lab/optimizer.h"

namespace lab {
    /**
     * Оптимизатор на основе комбинированного метода Брента
     */
    class Brent : public Optimizer {
      public:
        Brent(const func& optimized_function, double epsilon, double start,
              double end);

      protected:
        void step();
        bool is_done();

      private:
        const double TAU = (3 - std::sqrt(5)) / 2;
        double d, e;
        double x, w, v;
        double fx, fw, fv;
        bool m_is_done = false;

        bool areDistinct(double a, double b, double c) {
```

```
            return a != b && b != c;
        }

        double parabola(double start, double mid, double end, double f_start,
                        double f_mid, double f_end) {
            return mid
                   - 0.5
                         * ((mid - start) * (mid - start) * (f_mid - f_end)
                            - (mid - end) * (mid - end) * (f_mid - f_start))
                         / ((mid - start) * (f_mid - f_end)
                            - (mid - end) * (f_mid - f_start));
        }
    };
}  // namespace lab
```

```
#include "lab/brent.h"

#include <fmt/core.h>

#include <cmath>
#include <iostream>

using namespace lab;

Brent::Brent(const func& optimized_function, double epsilon, double start,
             double end)
    : Optimizer(optimized_function, epsilon, start, end) {
    x = end;
    w = x;
    v = x;
    fx = f(x);
    fw = fx;
    fv = fx;
    d = e = end - start;
}

bool Brent::is_done() { return m_is_done || Optimizer::is_done(); }

void Brent::step() {
    double min = segment.get_start();
    double max = segment.get_end();
    double mid = (min + max) / 2;
    double t = epsilon / 2 * std::abs(x) + epsilon / 10;
    if (std::abs(x - mid) <= (t * 2 - (max - min) / 2)) {
        m_is_done = true;
        return;
    }
    double u, fu;

    if (std::abs(e) > t) {
        double r = (x - w) * (fx - fv);
        double q = (x - v) * (fx - fw);
        double p = (x - v) * q - (x - w) * r;
        q = 2 * (q - r);
        if (q > 0) p = -p;
        q = std::abs(q);
        double g = e;
        e = d;
        // Проверка точности параболы:
```

```
        if ((std::abs(p) >= std::abs(q * g / 2)) || (p <= q * (min - x))
            || (p >= q * (max - x))) {
            // Парабола неточная, используем золотое сечение
            e = (x >= mid) ? min - x : max - x;
            d = TAU * e;
        } else {
            // Используем параболу
            d = p / q;
            u = x + d;
            if (((u - min) < t * 2) || ((max - u) < t * 2)) {
                d = std::copysign(t, mid - x);
            }
        }
    } else {
        // Используем золотое сечение
        e = (x >= mid) ? min - x : max - x;
        d = TAU * e;
    }
    if (std::abs(d) >= t) {
        u = x + d;
    } else {
        u = x + std::copysign(t, d);
    }
    fu = f(u);

    segment.saved_points["u"] = {u, fu};
    segment.saved_points["w"] = {w, fw};
    segment.saved_points["x"] = {x, fx};
    segment.saved_points["v"] = {v, fv};
    save_segment();

    if (fu <= fx) {
        if (u >= x) {
            min = x;
        } else {
            max = x;
        }
        v = w;
        w = x;
        x = u;
        fv = fw;
        fw = fx;
        fx = fu;
    } else {
        if (u < x) {
            min = u;
        } else {
            max = u;
        }
        if ((fu <= fw) || (w == x)) {
            v = w;
            w = u;
            fv = fw;
            fw = fu;
        } else if ((fu <= fv) || (v == x) || (v == w)) {
            v = u;
            fv = fu;
        }
    }

```

```
105         segment = {min, max};
106     }
```