Министерство науки и высшего образования Российской Федерации
федеральное государственное автономное образовательное учреждение
высшего образования
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт

По лабораторной работе №1

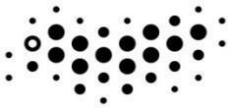
По дисциплине: прикладная математика

Факультет: ИТиП

Группа: М32001

Авторы:

Андреев Артём Русланович
Слюсаренко Сергей Владимирович
Шевченко Валерий Владимирович



Санкт-Петербург 2022

1. Решить задачу в соответствии с номером варианта. Для решениея реализовать алгоритмы одномерной минимизации функции без производной: метод дихотомии, метод золотого сечения, метод Фиббоначи, метод парабол и комбинированный метод Брента.

2. Сравните методы по количеству итераций и количеству вычислений функции в зависимости от разной точности. Для каждого метода обязательно указывайте, как изменяется отрезок при переходе к следующей итерации.

3. Протестировать
реализованные алгоритмы для задач минимизации многомодальных функций, например, на различных полиномах. Могут ли метод золотого сечения/Брента не найти локальный минимум многомодальной функции?

4. По результатам выполнения лабораторной работы необходимо подготовить отчет. Отчет должен содержать описание реализованных вами алгоритмов, ссылку на реализацию, необходимые тесты и таблицы.

5. Для защиты лабораторной работы необходимо знать описание методов на языке математики, пояснять полученные результаты, а также уметь обосновать разумность примененных Вами методов для данных функций.

Вариант 1

Горный хребет

Геодезист, изучая профиль хребта некой горной системы пришел к выводу, что “в разрезе” его можно приближенно описать функцией, заданной на промежутке:

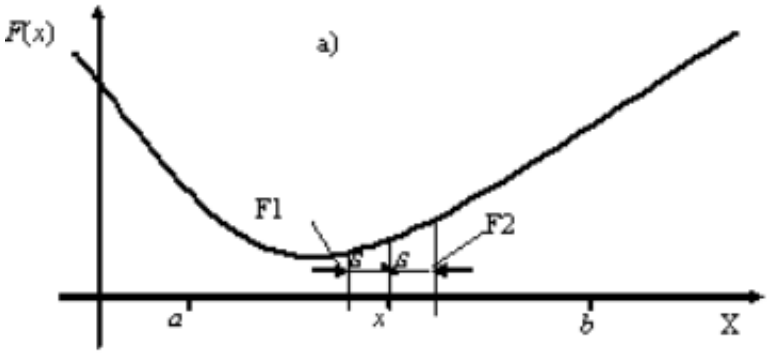
$$y(x) = \sin(x) \cdot x^3$$

По данной модели профиля хребта и промежутку, на которой этот профиль задан, найдите самую низкую точку рассматриваемого участка системы гор.

Алгоритмы:

Метод дихотомии

Этот метод оптимизации функции основывается на том, что если взять две очень близкие точки, на функции с одним экстремумом, то, посчитав значения функции в этих точках и сравнив эти значения, можно определить, с какой стороны от данной точки находится экстремум функции. Приведу пример с поиском минимума функции. Пусть у нас есть какая точка x внутри промежутка [a, b] и заданная точность eps. Тогда, нужно взять две точки: x_1 и x_2, которые будут равны x – delta и x + delta (delta < eps / 2) соответственно, и посчитать значения функции в этих точках (F_1 и F_2). Если $F_1 < F_2$, то функция возрастает на промежутке от x до b, следовательно мы можем отбросить этот промежуток, так как нас интересует минимум функции, то есть уменьшить промежуток поиска до [a, x], снова взять внутри этого промежутка точку и повторять всё, что мы уже делали, пока разница между границами промежутка поиска не станет меньше eps, тогда промежуток сойдётся и это будет нашим ответом. Конкретно в методе дихотомии в качестве точки x, в промежутке поиска, берётся середина отрезка [a, b]. Так как на каждой итерации отбрасывается одна из половин отрезка, то отрезок уменьшается в примерно 2 раза.

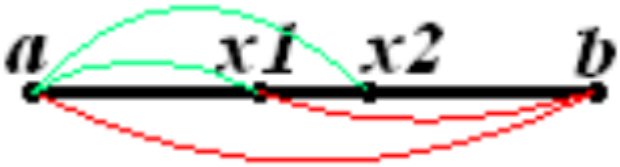


Примерный алгоритм:
1. Если разница a и b меньше, чем заданный eps, то ответ найден, это середина между a и b.
2. Иначе, на промежутке [a, b] берём середину – x, находим точки x_1 и x_2 в окрестности x, и их значения F_1 и F_2.
3. В зависимости от того, какое значение больше, возвращаемся к пункту 1 с параметрами [x, b] или [a, x].

Особенности метода дихотомии:
- Гарантируется коэффициент сужения отрезка – 0.5, что является неплохой константной.
- За одну итерацию происходит 2 вызова оракула (функции), что может быть проблемой, если процесс вычисления значения функции трудоёмок.

Метод золотого сечения

Этот метод во многом схож с методом дихотомии, но главная его идея – это то, что точки для сравнения выбираются не в окрестности середины отрезка, а симметрично относительно границ промежутка с помощью золотого сечения. Мы проигрываем относительно дихотомии в скорости сходимости, так как константа теперь не 0.5, а примерно 0.62, но благодаря тому, что точки x_1 и x_2 были выбраны симметрично с соблюдением золотого сечения, то при обработке новой итерации окажется так, что одна из двух новых точек совпадает с одной точкой предыдущей итерации, благодаря этому нам можно посчитать значение только одной из двух внутренних точек. Значение другой точки можно взять из предыдущей итерации.



Примерный алгоритм:
1. На промежутке [a, b] берём две точки x_1 и x_2, по формуле $b - (b - a)/K$ и $a + (b - a)/K$, где K – значение золотого сечения.
2. Находим значения функции F_1 и F_2.
3. В зависимости от того, какое значение больше, сохраняем одну внутреннюю точку с её значением в памяти и приравниваем одну из границ отрезка другой внутренней точке.
4. Если разница a и b меньше, чем заданный eps, то ответ найден, это середина между a и b.
5. На промежутке [a, b] находим оставшуюся внутреннюю точку и считаем её значение.
6. В зависимости от того, какое значение больше (значение в новой точке или старой), сохраняем одну внутреннюю точку с её значением в памяти и приравниваем одну из границ отрезка другой внутренней точке и переходим к пункту 4.

Особенности метода золотого сечения:
- Гарантируется коэффициент сужения отрезка – 0.62 (Значение золотого сечения).
- На каждой итерации, кроме начальной, выполняется только 1 вызов к оракулу, что может выиграть много времени, если процесс высчитывания значения функции трудоёмок.

Метод Фибоначчи

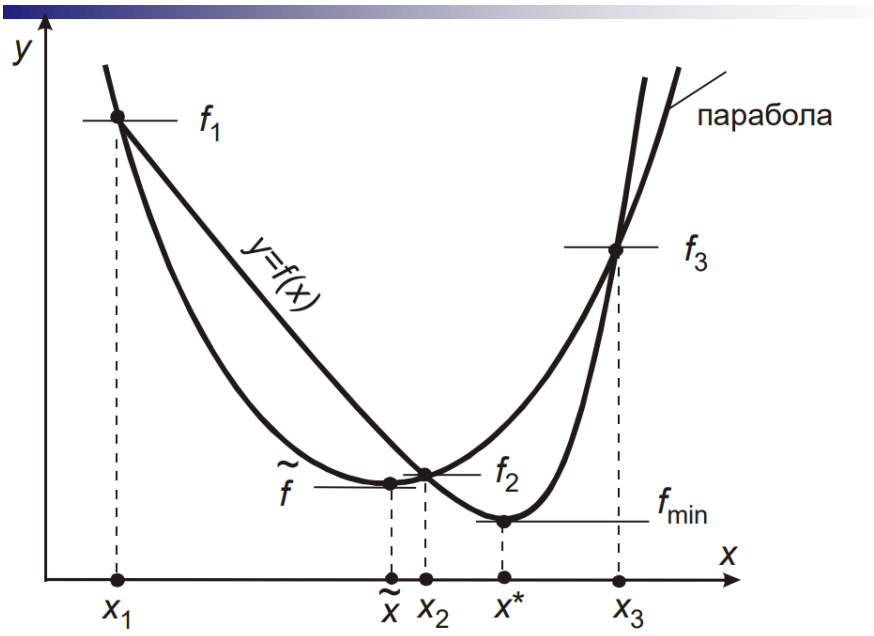
Этот метод основан на том, что в пределе отношение n-ого числа Фибоначчи к следующему равняется значению золотого сечения. Пользуясь этим фактом, в методе золотого сечения можно значение константы на отношение чисел Фибоначчи. Когда нам заранее известно количество итераций, мы можем сразу посчитать n-ое, (n-1)-ое и (n-2)-ое числа Фибоначчи, используя это, посчитать на начальной итерации две внутренние точки, дальше на каждой итерации одна точка будет сохраняться, а вторую можно посчитать по формуле через нужное число Фибоначчи. Но также, можно получить эту вторую точку и другой формулой, в которой нет чисел Фибоначчи, её можно вывести.
Однако чаще всего нам неизвестно число итераций заранее, поэтому нужно его оценивать согласно условию, что n-ое число Фибоначчи должно быть больше, чем длина изначального отрезка делённого на eps. Эта оценка займёт какое-то время.

Примерный алгоритм: (схож с золотым сечением)
0. Делаем оценку количества итераций.
1. На промежутке [a, b] берём две точки x_1 и x_2, по формуле $b – (b – a) * F_{n-1}/F_n$ и $a + (b – a)* F_{n-2}/F_n$, где F_n – соответствующие числа Фибоначчи.
2. Находим значения функции F_1 и F_2.
3. В зависимости от того, какое значение больше, сохраняем одну внутреннюю точку с её значением в памяти и приравниваем одну из границ отрезка другой внутренней точке.
4. Если разница a и b меньше, чем заданный eps, то ответ найден, это середина между a и b.
5. На промежутке [a, b] находим оставшуюся внутреннюю точку (формула $x_2 = b – (x_1 – a)$ или $x_1 = a – (x_2 – b)$) и считаем её значение.
6. В зависимости от того, какое значение больше (значение в новой точке или старой), сохраняем одну внутреннюю точку с её значением в памяти и приравниваем одну из границ отрезка другой внутренней точке и переходим к пункту 4.

Особенности метода Фибоначчи:
- При заранее известном числе итераций получается определённый выигрыш относительно метода золотого сечения по скорости выполнения за счёт менее затратного поиска внутренних точек, однако количество итераций и количество обращений к оракулу останется таким же, как и у метода золотого сечения.
- При неизвестном числе итераций нужно потратить время на поиск нужного числа Фибоначчи.
- Большую часть времени коэффициент сужения отрезка – 0.62 (Значение золотого сечения), но при приближении к концу коэффициент начинает немного изменяться в разные стороны, так как отношение чисел Фибоначчи, при приближении к первым числам ряда, начинает заметно отличаться от 0.62, но не критично для данного метода. Хоть в алгоритме формула в цикле не использует числа Фибоначчи, но можно доказать, что эта формула идентична формуле с использованием соотношения чисел Фибоначчи.
- На каждой итерации, кроме начальной, выполняется только 1 вызов к оракулу, что может выиграть много времени, если процесс высчитывания значения функции трудоёмок.

Метод парабол

Этот метод основывается на том, что можно взять 3 точки, построить по ним параболу и найти её минимум, этот минимум используется как внутренняя точка для последующих сравнений. Предполагается, что данная унимодальная функция будет похожа на параболическую функцию, тогда найденный минимум будет недалеко от настоящего минимума функции и промежуток поиска сойдётся очень быстро по сравнению с предыдущими методами. Однако, если функция будет иметь не параболический вид, а, скажем, логарифмический, то этот метод может очень долго, опять же по сравнению с предыдущими методами, сужать промежуток поиска.



Сам алгоритм делится на два логических этапа:
1. Поиск удачной тройки, в которой выполняется условие $f(x_1) > f(x_2) < f(x_3)$.
2. Минимизация функции путём поиска минимума параболы, проходящей через точки: $(x_1, f(x_1))$, $(x_2, f(x_2))$, $(x_3, f(x_3))$.

Примерный алгоритм:
1. Берём x_2, как середину между a и b и считаем во всех трёх точках значения функции.
2. Если условие $f(x_1) > f(x_2) < f(x_3)$ не выполняется, то меняем одну из границ на середину, и возвращаемся к пункту 1.
3. Считаем значения функции для x_1, x_2, x_3.
4. Находим уравнение параболы проходящей через точки: $(x_1, f(x_1))$, $(x_2, f(x_2))$, $(x_3, f(x_3))$ и находим её минимум.
5. Берём новую тройку, которая удовлетворяет условию $f(x_1) > f(x_2) < f(x_3)$ из старых границ, текущего x_2 и посчитанного минимума параболы.
6. Если разница между x_2 и одной из границ больше, чем eps, то вернуться к пункту 3.
7. Иначе ответ – это точно x_2.

Особенности метода парабол:
- Нестабильный коэффициент сужения отрезка, но крайне быстрый на параболических функциях. Однако, с функциями другого вида, метод парабол может работать сильно дольше, чем предыдущие методы.
- На каждой итерации происходит только один вызов оракула.

Комбинированный метод Брента

Среди вышеупомянутых методов самым эффективным можно назвать метод парабол, однако с некоторыми функциями он может работать сильно дольше, чем тот же метод золотого сечения, также метод парабол нестабильно сокращает отрезок поиска на первых итерациях. Идея метода Брента заключается в том, чтобы использовать метод золотого сечения на начальных этапах, а когда отрезок удобен использовать метод парабол.

Метод парабол выбирается, если:
- Найденная внутренняя точка попадает в отрезок поиска и отстоит от границ больше, чем на eps.
- Найденная внутренняя точка отстоит от середины не более, чем на половину длины пред предыдущего шага, для определения этого, мы должны хранить в памяти длины предыдущих шагов.

Логически алгоритм разбивается на две части:
1. Поиск нового минимума с помощью метода парабол, если он подходит под условия, берём его.
2. Иначе ищем новую внутреннюю точку с помощью метода золотого сечения.

Примерный алгоритм:
0. Пусть изначально левая и правая внутренние точки равны середине.
1. Запоминаем предыдущий и пред предыдущий шаги.
2. Если середина не совпадает с границей отрезка, находим минимум параболы из точек: левая граница, середина, правая граница.
3. Если найденный минимум имеет шаг меньшей, чем пред предыдущий делённый на 2, то берём его как внутреннюю точку.
4. Иначе выбираем новую внутреннюю точку согласно золотому сечению.
5. Изменяем промежуток поиска согласно найденным внутренним точкам.
6. Если длина отрезка больше, чем eps, то вернуться к пункту 1.

Реализация:

https://github.com/JabaJabila/ITMO_AppliedMath/tree/main/Lab1

Дихотомия:

```
def find_min(function, left_bound, right_bound, eps):
    segments = []
    calls = 0

    while right_bound - left_bound > eps:
        segments.append((left_bound, right_bound))

        middle = (left_bound + right_bound) / 2

        delta = eps / 2 * 0.9
        left_result = function(middle - delta)
        right_result = function(middle + delta)
        calls += 2

        if left_result < right_result:
            right_bound = middle + delta
        else:
            left_bound = middle - delta

    segments.append((left_bound, right_bound))
    return (left_bound + right_bound) / 2, calls, segments
```

Золотое сечение:

```
from math import sqrt


def find_min(function, left_bound, right_bound, eps):
    golden_ratio = (sqrt(5) + 1) / 2
    segments = []
    calls = 0

    left_point = right_bound - (right_bound - left_bound) / golden_ratio
    right_point = left_bound + (right_bound - left_bound) / golden_ratio
    calls += 2

    left_result = function(left_point)
    right_result = function(right_point)

    while right_bound - left_bound > eps:
        segments.append((left_bound, right_bound))

        if left_result < right_result:
            right_bound = right_point
            right_result = left_result

            right_point = left_point
            left_point = right_bound - (right_bound - left_bound) /
golden_ratio

            left_result = function(left_point)
            calls += 1
```

```
        else:
            left_bound = left_point
            left_result = right_result

            left_point = right_point
            right_point = left_bound + (right_bound - left_bound) /
golden_ratio

            right_result = function(right_point)
            calls += 1


    segments.append((left_bound, right_bound))
    return (right_bound + left_bound) / 2, calls, segments
```

Фибоначчи:

```
from math import sqrt, pow


def get_fibonacci_number(n):
    return -1 / sqrt(5) * pow((1 - sqrt(5)) / 2, n) + 1 / sqrt(5) * pow((1 +
sqrt(5)) / 2, n)


def find_min(function, left_bound, right_bound, eps):
    segments = []
    calls = 0

    iteration_number = 1
    fibonacci_number_n = 1
    while fibonacci_number_n <= (right_bound - left_bound) / eps:
        iteration_number += 1
        fibonacci_number_n = get_fibonacci_number(iteration_number)

    prev_1 = get_fibonacci_number(iteration_number - 1)
    prev_2 = fibonacci_number_n - prev_1

    left_point = left_bound + (right_bound - left_bound) * prev_2 /
fibonacci_number_n
    right_point = left_bound + (right_bound - left_bound) * prev_1 /
fibonacci_number_n

    left_result = function(left_point)
    right_result = function(right_point)
    calls += 2

    for _ in range(iteration_number, 1, -1):
        segments.append((min(left_bound, right_bound), max(left_bound,
right_bound)))

        if abs(right_bound - left_bound) < eps:
            return (segments[-1][0] + segments[-1][1]) / 2, calls, segments

        if left_result < right_result:
            right_bound = right_point
```

```
            right_point = left_point
            right_result = left_result

            left_point = left_bound + (right_bound - right_point)
            left_result = function(left_point)
            calls += 1
        else:
            left_bound = left_point

            left_point = right_point
            left_result = right_result

            right_point = right_bound + (left_bound - left_point)
            right_result = function(right_point)
            calls += 1

    segments.append((left_bound, right_bound))
    return (segments[-1][0] + segments[-1][1]) / 2, calls, segments
```

Параболы:

```
def find_min(function, left_bound, right_bound, eps):
    segments = []
    calls = 0
    middle = (right_bound + left_bound) / 2

    left_result = function(left_bound)
    middle_result = function(middle)
    right_result = function(right_bound)
    calls += 3

    while abs(right_bound - left_bound) > eps:
        segments.append((left_bound, right_bound))

        if left_result < middle_result:
            right_bound = middle
            middle = (left_bound - right_bound) / 2

            right_result = middle_result
            middle_result = function(middle)
            calls += 1
        elif right_result < middle_result:
            left_bound = middle
            middle = (left_bound - right_bound) / 2

            left_result = middle_result
            middle_result = function(middle)
            calls += 1
        else:
            break

    left_point = left_bound
    inner_point = middle
    right_point = right_bound

    inner_result = middle_result

    while inner_point - left_point > eps and right_point - inner_point > eps:
```

```
        a1 = (inner_result - left_result) / (inner_point - left_point)
        a2 = 1 / (right_point - inner_point) * ((right_result - left_result)
/ (right_point - left_point) -
                                                (inner_result - left_result)
/ (inner_point - left_point))

        min_point = 1.0 / 2.0 * (left_point + inner_point - a1 / a2)

        if left_point < min_point < inner_point:
            right_point = inner_point
            inner_point = min_point

            right_result = inner_result
            inner_result = function(inner_point)
            calls += 1

            segments.append((inner_point, right_point))

        elif inner_point < min_point < right_point:
            left_point = inner_point
            inner_point = min_point

            left_result = inner_result
            inner_result = function(inner_point)
            calls += 1

            segments.append((left_point, inner_point))

        else:
            return inner_point, calls, segments

    return inner_point, calls, segments
```

Метод Брента:

```
from math import fabs, sqrt

const = (3 - sqrt(5)) / 2


def find_min(function, left_bound, right_bound, eps):

    def find_parabola_min(l, m, r, left_res, middle_res, right_res):
        if l == m or m == r or l == r:
            return None

        a1 = (middle_res - left_res) / (m - l)
        a2 = 1 / (r - m) * ((right_res - left_res) / (r - l) - (middle_res -
left_res) / (m - l))

        if a2 == 0:
            return None

        return 1.0 / 2.0 * (l + m - a1 / a2)

    middle = (left_bound + right_bound) / 2.0
    left_point = right_point = middle
```

```
    left_bound_result = function(left_bound)
    right_bound_result = function(right_bound)
    middle_result = function(middle)
    calls = 3

    left_result = right_result = middle_result
    prev_len = current_len = right_bound - left_bound
    segments = [(left_bound, right_bound)]
    inner_point = middle
    eps /= 10

    while fabs(right_bound - left_bound) >= 10 * eps:
        prev_prev_len = prev_len
        prev_len = current_len
        parabola_min = None

        if fabs(left_point - middle) > eps or fabs(middle - right_point) >
eps:
            parabola_min = find_parabola_min(left_bound, middle, right_bound,
                                             left_bound_result,
middle_result, right_bound_result)

        if (parabola_min is not None and fabs(parabola_min - left_bound) >
eps
                and fabs(right_bound - parabola_min) > eps
                and fabs(parabola_min - middle) < prev_prev_len / 2):

            current_len = fabs(inner_point - middle)
            if left_bound < parabola_min < middle:
                right_bound = middle
                inner_point = parabola_min

                right_bound_result = middle_result

            elif middle < parabola_min < right_bound:
                left_bound = middle
                inner_point = parabola_min

                left_bound_result = middle_result
        else:
            if middle <= (right_bound + left_bound) / 2:
                inner_point = middle + const * (right_bound - middle)
                current_len = right_bound - middle

            else:
                inner_point = middle + const * (left_bound - middle)
                current_len = middle - left_bound

        if fabs(inner_point - middle) < eps:
            inner_point = middle + eps if inner_point - middle > 0 else
middle - eps

        inner_result = function(inner_point)
        calls += 1

        if inner_result <= middle_result:
            if inner_point >= middle:
                left_bound = middle
```

```
                left_bound_result = middle_result
            else:
                right_bound = middle
                right_bound_result = middle_result

            right_point, left_point, middle = left_point, middle, inner_point
            right_result, left_result, middle_result = left_result,
middle_result, inner_result
            segments.append((min(left_bound, right_bound), max(left_bound,
right_bound)))

        else:
            if inner_point >= middle:
                right_bound = inner_point
                right_bound_result = inner_result
            else:
                left_bound = inner_point
                left_bound_result = inner_result

            if inner_result <= left_result or left_point == middle:
                right_point, left_point = left_point, inner_point
                right_result, left_result = left_result, inner_result
                segments.append((min(left_bound, right_bound),
max(left_bound, right_bound)))

            elif inner_result <= right_result or right_point == middle:
                right_point, right_result = inner_point, middle_result
                segments.append((min(left_bound, right_bound),
max(left_bound, right_bound)))

    return (segments[-1][0] + segments[-1][1]) / 2, calls, segments
```

Тесты:

Возьмём пару унимодальных промежутков на заданной во варианте функции и посчитаем точку минимума с точностью 0.0001

Промежуток [-1, 2] – ожидаемый результат = 0

Dichotomy algorithm:

-1.52587890625e-05

GoldenRatio algorithm:

-1.858616715820573e-05

Fibonacci algorithm:

-3.234989422440382e-05

Parabola algorithm:

5.852661747301489e-07

Brent algorithm:

2.5222630203143654e-05

Промежуток [3, 6] – ожидаемый результат = 5.23294 (вычислено через Wolfram alpha)

Dichotomy algorithm:

5.2329254150390625

GoldenRatio algorithm:

5.232940087302798

Fibonacci algorithm:

5.232951604552608

Parabola algorithm:

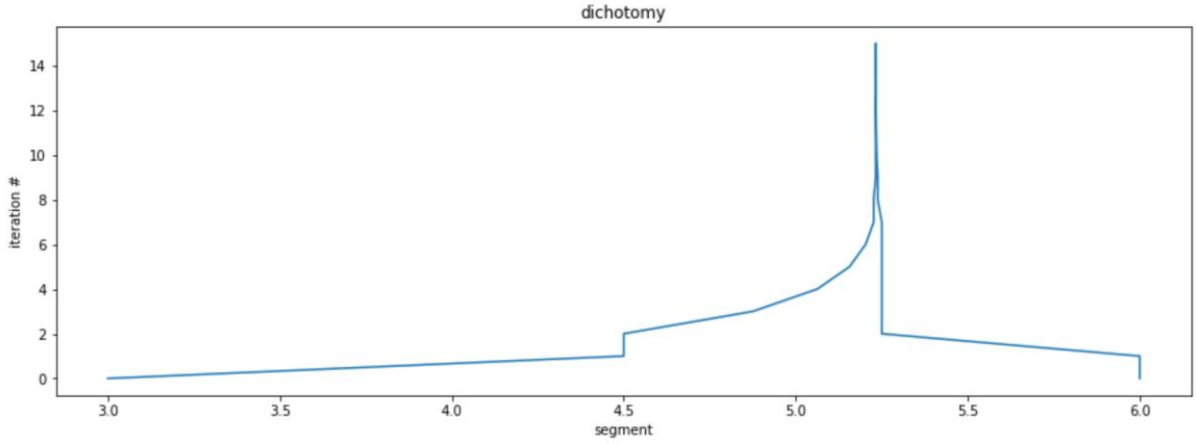
5.232914149621308

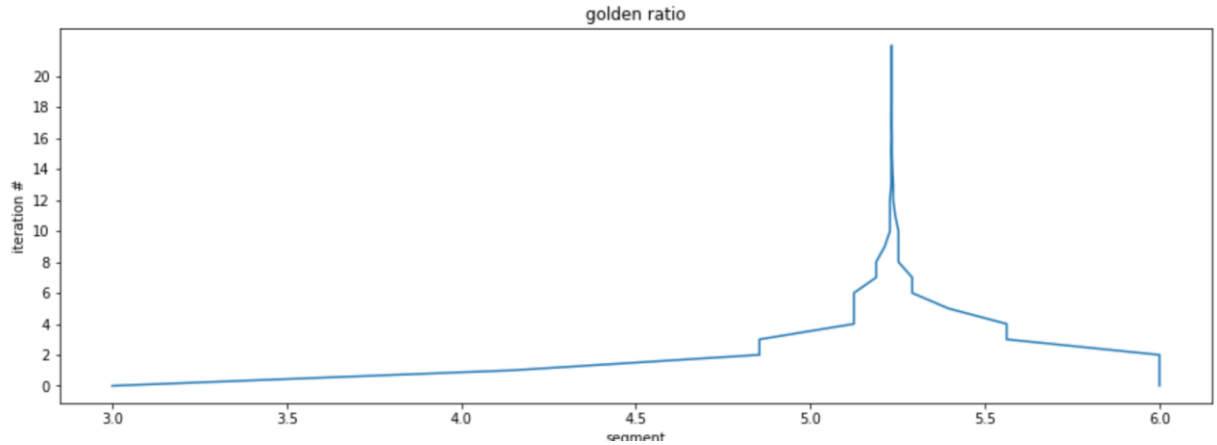
Brent algorithm:

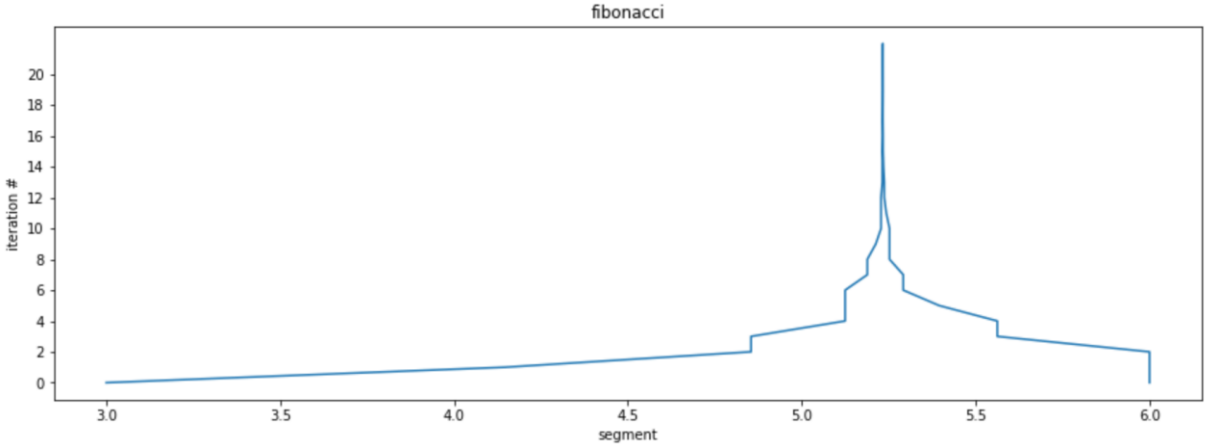
5.232942528972387

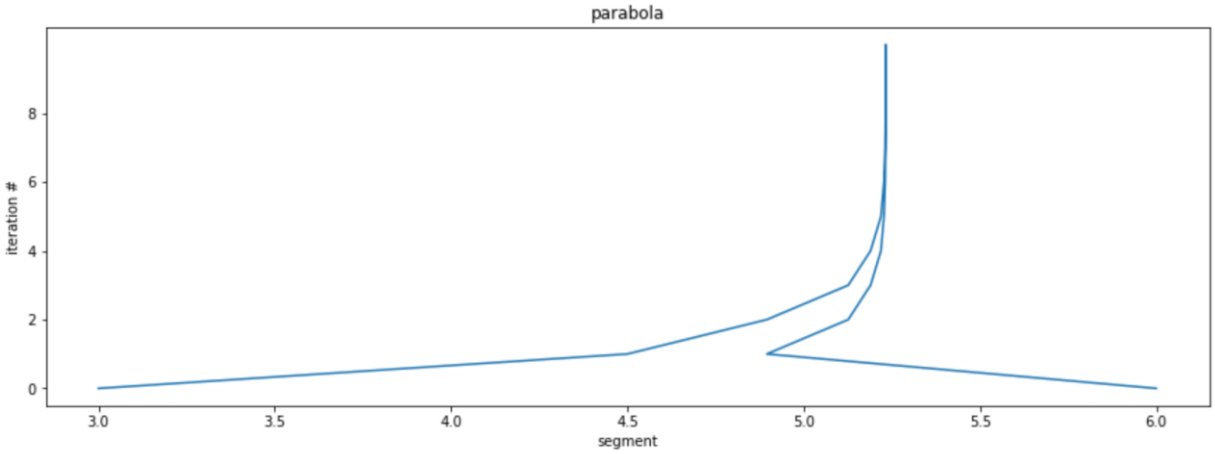
На примере последнего промежутка проанализируем и сравним алгоритмы:

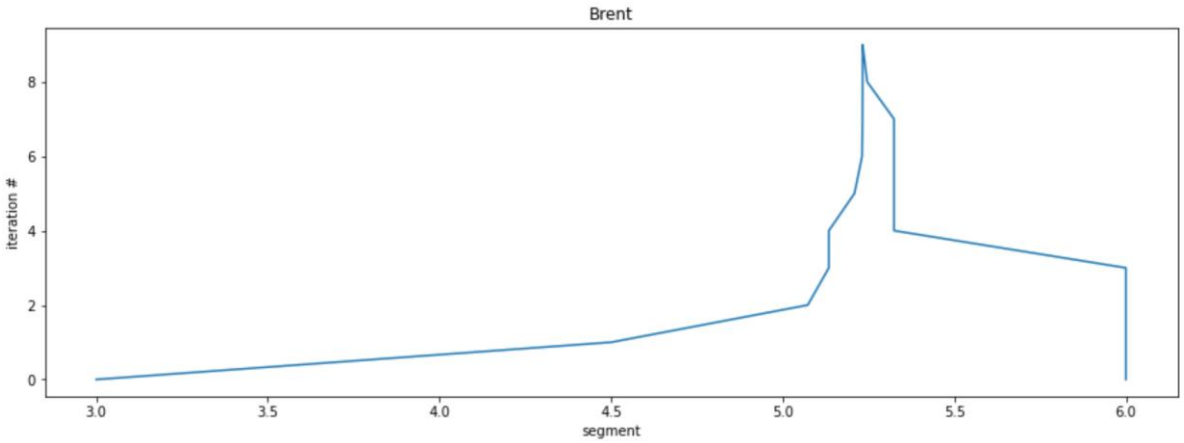
Поведение отрезков поиска:



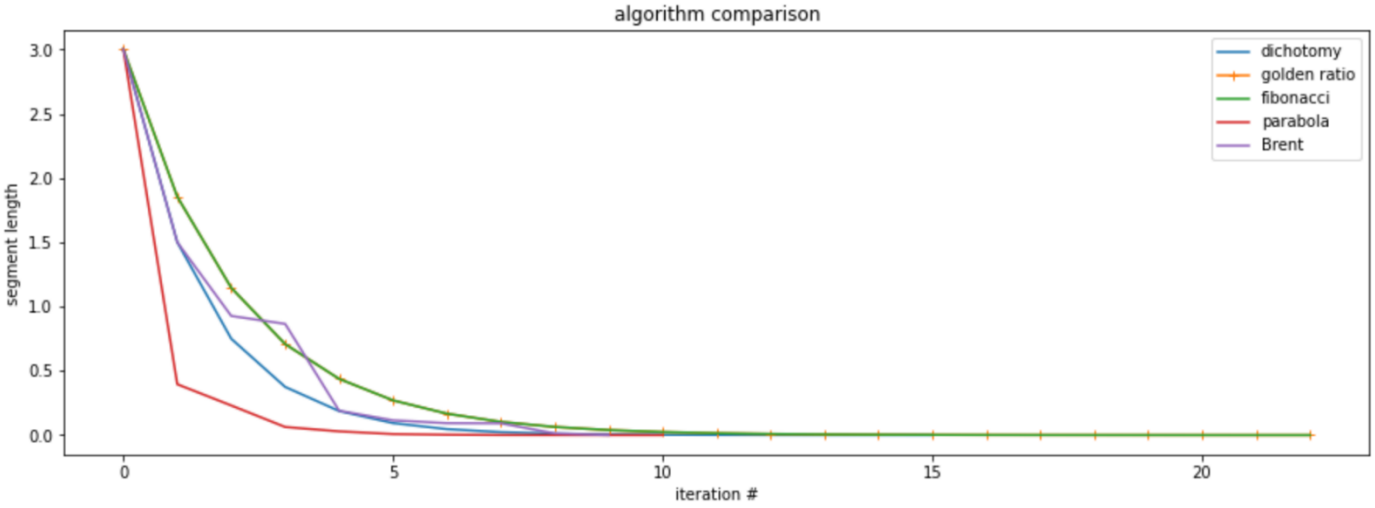






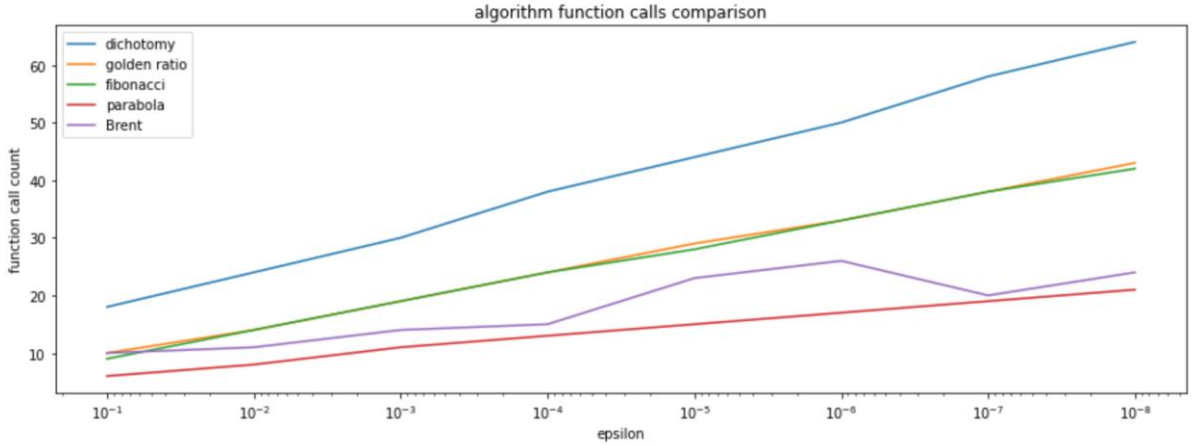


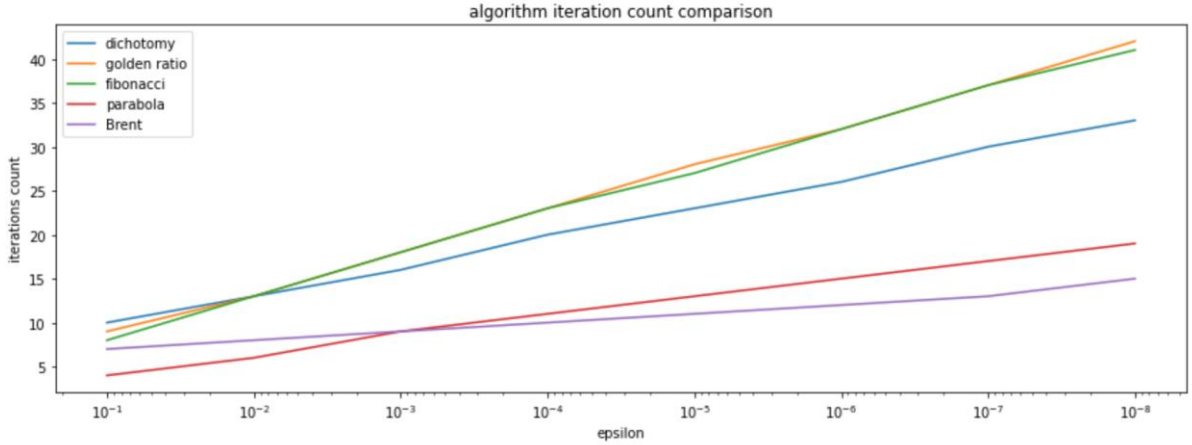
Сравнение всех алгоритмов по скорости сходимости отрезка поиска:



Как видно, для нашего случая скорость сходимости у метода парабол – самая высокая, на втором месте – метод Брента и дихотомии, а алгоритмы Фибоначчи и золотого сечения сходятся примерно одинаково и медленнее всего

Сравнение всех алгоритмов по количеству обращений к оракулу и итераций для разных точностей:





При увеличении точности алгоритм Брента начинает выигрывать по количеству итераций, а также хорошо себя показывает по количеству обращений к оракулу. Алгоритм дихотомии требует больше всего обращений к оракулу, наихудшую асимптотику по итерациям показывают методы золотого сечения и Фибоначчи.

Сводные таблицы по работе каждого алгоритма по итерациям:

Dichotomy algorithm

iteration	segment length	segment ratio
0	3.000000	1.000000
1	1.500045	0.500015
2	0.750068	0.500030
3	0.375079	0.500060
4	0.187584	0.500120
5	0.093837	0.500240
6	0.046964	0.500480
7	0.023527	0.500958
8	0.011808	0.501913
9	0.005949	0.503811
10	0.003020	0.507564
11	0.001555	0.514903
12	0.000822	0.528943
13	0.000456	0.554718
14	0.000273	0.598641
15	0.000182	0.664775
16	0.000136	0.747866
17	0.000113	0.831431
18	0.000101	0.898627
19	0.000096	0.943596

Golden ratio algorithm

iteration	segment length	segment ratio
0	3.000000	1.000000
1	1.854102	0.618034
2	1.145898	0.618034
3	0.708204	0.618034
4	0.437694	0.618034
5	0.270510	0.618034
6	0.167184	0.618034
7	0.103326	0.618034
8	0.063859	0.618034
9	0.039467	0.618034
10	0.024392	0.618034
11	0.015075	0.618034
12	0.009317	0.618034
13	0.005758	0.618034
14	0.003559	0.618034
15	0.002199	0.618034
16	0.001359	0.618034
17	0.000840	0.618034
18	0.000519	0.618034
19	0.000321	0.618034
20	0.000198	0.618034
21	0.000123	0.618034
22	0.000076	0.618034

Fibonacci algorithm

iteration	segment length	segment ratio
0	3.000000	1.000000
1	1.854102	0.618034
2	1.145898	0.618034
3	0.708204	0.618034
4	0.437694	0.618034
5	0.270510	0.618034
6	0.167184	0.618034
7	0.103326	0.618034
8	0.063859	0.618034
9	0.039467	0.618034
10	0.024392	0.618033
11	0.015075	0.618037
12	0.009317	0.618026
13	0.005758	0.618056
14	0.003558	0.617978
15	0.002200	0.618182
16	0.001359	0.617647
17	0.000841	0.619048
18	0.000518	0.615385
19	0.000323	0.625000
20	0.000194	0.600000
21	0.000129	0.666667
22	0.000065	0.500000

Parabola algorithm

iteration	segment length	segment ratio
0	3.000000	1.000000
1	0.395711	0.131904
2	0.230776	0.583194
3	0.063374	0.274614
4	0.029431	0.464394
5	0.008543	0.290278
6	0.003431	0.401602
7	0.001069	0.311574
8	0.000401	0.374668
9	0.000131	0.326111
10	0.000047	0.361290

Brent algorithm

iteration	segment length	segment ratio
0	3.000000	1.000000
1	1.500000	0.500000
2	0.927051	0.618034
3	0.865447	0.933549
4	0.190068	0.219618
5	0.115369	0.606985
6	0.093221	0.808025
7	0.092060	0.987554
8	0.013380	0.145345
9	0.000010	0.000747

А что там с многомодальностью?

Возьмём нашу функцию на промежутке [-15, -3] где находятся 2 минимума. Посмотрим как сработают наши алгоритмы.

Dichotomy algorithm:

-11.256088256835938

GoldenRatio algorithm:

-11.256043664059536

Fibonacci algorithm:

-11.256110319396061

Parabola algorithm:

-3.0

Brent algorithm:

-5.232942927872527

Вывод:

Все алгоритмы, кроме метода парабол нашли локальные минимумы, а в случае с первыми тремя алгоритмами нам повезло и найденный минимум совпал с глобальным минимумом на отрезке. При тестах на некоторых других промежутках метод парабол иногда уходит в бесконечный цикл.