МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное государственное учреждение высшего образования

Санкт-петербургский национальный исследовательский институт информационных технологий, механики и оптики

Мегафакультет трансляционных информационных технологий

Факультет информационных технологий и программирования

**Лабораторная работа №1**

**Вариант 5**

**По дисциплине «Прикладная математика»**

**Выполнил студент группы №M32061:**

*Золотарева Анастасия Кирилловна*

**Проверил:**

*Москаленко Мария Александровна*

**САНКТ-ПЕТЕРБУРГ**

**2021**

задание на лабораторную работу

Задания

Изображение выглядит как текст

Автоматически созданное описание

Постановка задачи в соответствии с вариантом

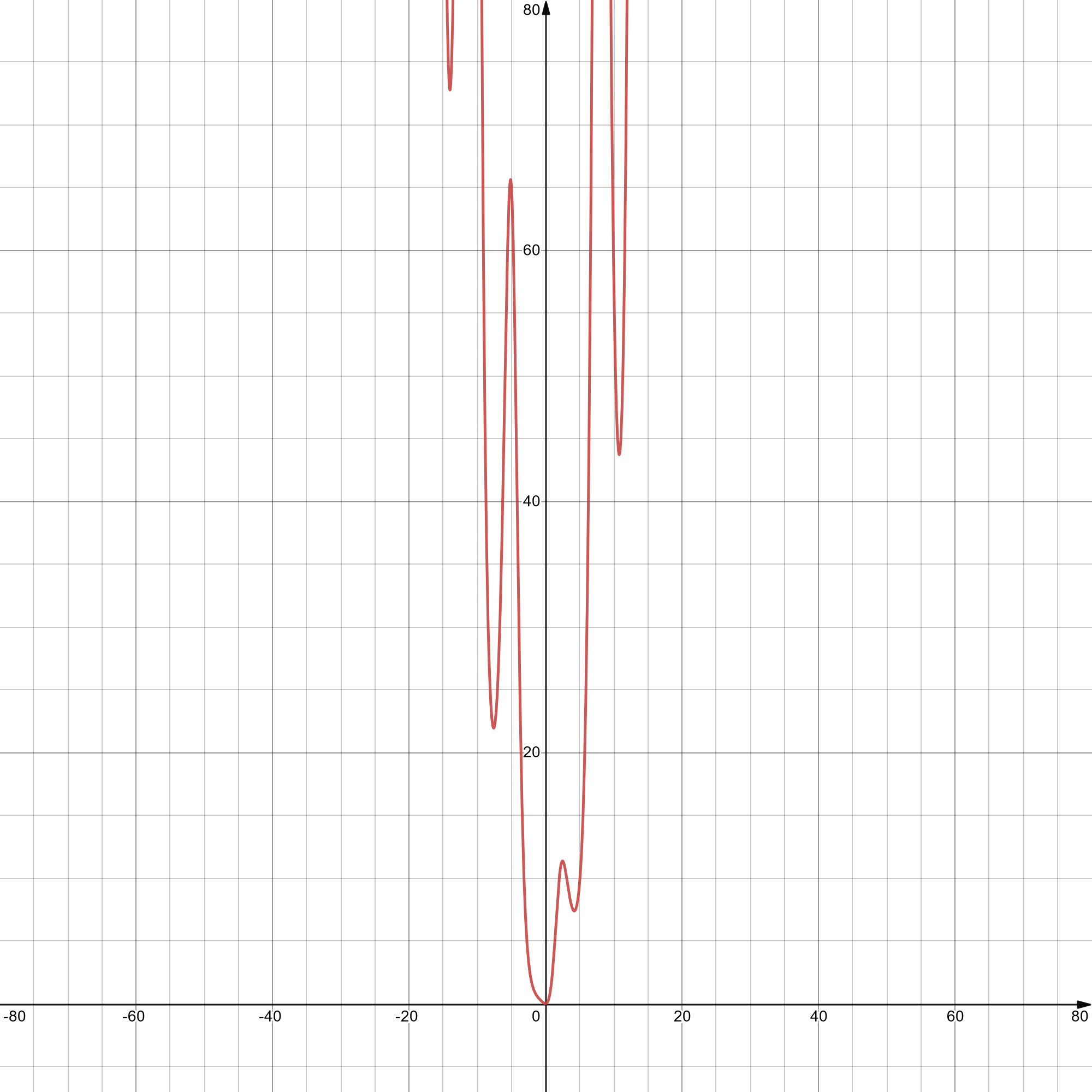
Изображение выглядит как текст

Автоматически созданное описание

краткий анализ заданной функции

График заданной функции

Данная функция имеет вид:



Область определения для данной функции , а область допустимых значений . Минимум данная функция, как видно по графику и как можно понять аналитическим путем, имеет в точке (0; 0).

реализация алгоритмов минимизации функции

Метод дихотомии

Методом дихотомии называют метод прямого поиска, основанный на делении пополам отрезка, на котором находится точка x\* (точка отрезка [a, b], в которой унимодальная функция достигает наименьшего значения).

Опишем алгоритм:

Пусть известно, что на k-ом шаге последовательного поиска x\* ∈ [ak, bk] ⸦ [a, b] (на первом шаге k=1, a1=a и b1=b).

1) На отрезке [ak, bk] длинной lk выберем две точки xk1=(ak+bk)/2-δ и xk2=(ak+bk)/2+δ, где δ>0 – некоторое достаточно малое число.

2) Вычислим значение f(xk1) и f(xk2) в этих точках и выполним процедуру исключения отрезка. И в результате получим новый отрезок [ak+1, bk+1] ⸦ [ak, bk]:

- Если длина lk+1 нового отрезка больше заданной длины наибольшей допустимой длины ε\* интервала неопределенности, то алгоритм метода дихотомии переходит к шагу (k+1), повторяя все описанные для k-го шага действия.

- Если же lk+1 <= ε\*, то вычисления прекращаются и полагают x\*=(ak+1+bk+1)/2.

Длина отрезка на k-м шаге метода: ; и ограничения для δ: Δ\*<2δ<ε\*.

Реализованный алгоритм:

def Dichotomy(a, b, epsilon):

    ans = [(a, b)]

    delta = epsilon / 3

    k = 0 # iterations

    while (abs(b - a) >= epsilon):

        k = k + 1

        x1 = (a + b) / 2 - delta

        x2 = (a + b) / 2 - delta

        f1 = Func(x1)

        f2 = Func(x2)

        if f1 == f2:

            a = x1

            b = x2

        elif f1 < f2:

            b = x2

        else:

            a = x1

        ans.append([(a, b)])

    OutputNumberOfIterations("Dychotomy", k)

    return ans

Метод золотого сечения

Золотое сечение отрезка – это такое деление его на две неравные части, при котором отношение длины всего отрезка к длине его большей части равно отношению длины большей части к длине меньшей.

Чтобы выполнить процедуру исключения отрезка на k-м шаге, отрезок [ak, bk] необходимо разделить на три части двумя внутренними точками xk1, xk2, xk1<xk2. Эти точки выбираются симметрично относительно середины отрезка [ak, bk] и так чтобы каждая из них производила золотое сечение данного отрезка. Тогда отрезок [ak+1, bk+1] внутри будет содержать одну из точек xk1, xk2 (другая будет одним из концов отрезка), причем эта точка будет производить золотое сечение отрезка [ak+1, bk+1]. Таким образом на (k+1)-м шаге в одной из точек xk+1,1, xk+1,2 значение функции вычислять не нужно. При этом отношение lk/lk+1 длин отрезков сохраняется от шага к шагу:

Описание алгоритма:

На первом шаге последовательного поиска k=1 на отрезке [a, b] выбираем две точки x11 = a1 + (1 – 1/τ)/b1 = 1 – 1/τ и x12 = a1 + τ/b1 = 1/τ, осуществляющие золотое сечение отрезка [a, b].

1) Вычисляем значения минимизируемой функции в этих точках и выполняем процедуру исключения отрезка.

- Если f(x11)<f(x12), то выбираем отрезок [a1, x12], т.е. полагаем a2=a1, b2=x12.

- В противном случае выбираем отрезок [x11, b1], т.е. полагаем a2=x11, b2=b1.

Также в первом случае принимаем x’2=x11, а во втором - x’2=x12, где точка x’2 – одна из точек, осуществляющих золотое сечение отрезка [a2, b2], меньшая в первом случае и большая во втором.

- Если длина вновь полученного отрезка больше заданной допустимой длины ε\* интервала неопределенности, то следует перейти ко второму шагу алгоритма, на котором одна из точек x21, x22 есть точка x’2, а вторую можно найти по формуле. На втором шаге вычисляем только одно значение функции в точке, симметричной x’2 относительно середины отрезка [a2, b2].

- Если длина l2 отрезка [a2, b2], полученного после первого шага алгоритма, меньше ε\*, то поиск прекращают и полагают x\*≈(a2+b2)/2.

На k-м шаге, k>=2, последовательного поиска по методу золотого сечения выбран отрезок [ak, bk] и в нем точка x’k, осуществляющая золотое сечение этого отреза. Значение f(x’k) функции в этой точке было вычислено на предыдущем шаге. Находим вторую точку x’’k золотого сечения по формуле и вычисляем в ней значение функции. Если x’’k<x’k, то xk1=x’’k и xk2=x’k, иначе то xk1=x’k и xk2=x’’k.

Пусть для определенности Если x’’k<x’k, то xk1=x’’k и xk2=x’k.

* Если f(xk1)<f(xk2), то выбираем отрезок [ak, xk2], т.е. полагаем ak+1=ak, bk+1=xk2, x’k+1=xk1.
* Иначе выбираем отрезок [xk1, bk], т.е. полагаем ak+1=xk1, bk+1=bk, x’k+1=xk2.

Длину lk+1 нового отрезка [ak+1, bk+1] сравниваем с ε\* и принимаем решение продолжать поиск при lk+1 >= ε\* или нет – при lk+1 < ε\*. В случае прекращения поиска полагаем x\*≈(a2+b2)/2.

Реализованный алгоритм:

# golden ratio method

def GoldenRatio(a, b, epsilon):

    k = 1 # iteration number

    fn = 0 # func counting number

    ans = [(a, b)]

    tau = (np.sqrt(5) - 1) / 2

    x1 = a + (b - a) \* (1 - tau)

    x2 = a + (b - a) \* tau

    f1 = Func(x1)

    f2 = Func(x2)

    fn = fn + 2

    OutputL("GoldenRatio", abs(b - a))

    while (abs(b - a) >= epsilon):

        k = k + 1

        if f1 <= f2:

            b = x2

            x2 = x1

            x1 = a + (b - a) \* (1 - tau)

            f2 = f1

            f1 = Func(x1)

            fn = fn + 1

        else:

            a = x1

            x1 = x2

            x2 = a + (b - a) \* tau

            f1 = f2

            f2 = Func(x2)

            fn = fn + 1

        ans.append([(a, b)])

        OutputL("GoldenRatio", abs(b - a))

    OutputNumberOfIterations("GoldenRatio", k)

    OutputNumberOfFunc("GoldenRatio", fn)

    return ans

Метод Фибоначчи

Это алгоритм метода прямого поиска, удовлетворяющий соотношению для длины интервала неопределенности:

а коэффициенты Fm определены соотношением

Fm – множество натуральных чисел и называются числами Фибоначчи.

Так метод, использующий числа Фибоначчи для выбора длин отрезков lk, а значит и точек xk ∈ [a, b], k=1, N, в которых вычисляют значения минимизируемой функции, называют методом Фибоначчи.

Описание алгоритма:

Пренебрегая малой δ из формулы для l2 – длины нового отрезка:

При l1=length([a, b]) находим l2 = l1\*FN/FN+1. Таким образом выбираем точки:

.

После выполнения на этом шаге процедуры исключения отрезка одна из точек x1, x2 будет граничной точкой нового отрезка [a2, a1], а другая – его внутренней точкой, которую обозначим x’2. Вторая внутренняя точка должна быть выбрана симметрично точке x’2 относительно его середины. Аналогично на всех последующих шагах.

На k-м шаге в соответствии с равенством в котором следует положить K=N-k, и равенством для длины lNf длина отрезка [ak, bk] равна lk=FN+2-k/FN+1 и происходит ее уменьшение в lk/lk+1=FN+2-k/FN+1-k раз.

- Если внутренние точки на этом отрезке обозначить αk и βk то приведенные рассуждения позволяют написать:

*Метод Фибоначчи не осуществим без задания требуемого количества N вычисляемых значений функции (или количества шагов поиска).*

Реализованный алгоритм:

# Get Fibonach number

def FibonachiNumber(n):

    n = n + 1

    f = 1 / np.sqrt(5) \* (((1 + np.sqrt(5)) / 2) \*\* n - ((1 - np.sqrt(5)) /2) \*\* n)

    return f

# Fibonachi method

def Fibonachi(a, b, epsilon):

    k = 1 # iteration number

    fn = 0 # func counting number

    ans = [(a, b)]

    n = 1

    while (FibonachiNumber(n) <= (b - a) / epsilon):

        n = n + 1

    x1 = a + (b - a) \* FibonachiNumber(n - 2) / FibonachiNumber(n)

    x2 = a + (b - a) \* FibonachiNumber(n - 1) / FibonachiNumber(n)

    f1 = Func(x1)

    f2 = Func(x2)

    fn = fn + 2

    OutputL("Fibonachi", abs(b - a))

    while(n > 2):

        k = k + 1

        if (f1 < f2):

            b = x2

            x2 = x1

            f2 = f1

            n = n - 1

            x1 = a + (b - a) \* FibonachiNumber(n - 2) / FibonachiNumber(n)

            f1 = Func(x1)

        else:

            a = x1

            x1 = x2

            f1 = f2

            n = n - 1

            x2 = a + (b - a) \* FibonachiNumber(n - 1) / FibonachiNumber(n)

            f2 = Func(x2)

        fn = fn + 1

        OutputL("Fibonachi", abs(b - a))

        ans.append([(a, b)])

    OutputNumberOfIterations("Fibonachi", k)

    OutputNumberOfFunc("Fibonachi", fn)

    return ans

Метод парабол

В данном методе аппроксимация происходит при помощи квадратичной функции:

.

Пусть имеются три точки x1<x2<x3 такие, что интервал [x1, x3] содержит точку минимума функции f(x). Тогда коэффициенты аппроксимирующей параболы a, b, c могут быть найдены путем решения системы линейных уравнений:

Минимум такой параболы равен:

Если f2< f1 и f2<f3, то точка u гарантированно попадает в интервал [x1, x3]. Таким образом, внутри интервала [x1, x3] определены две точки x2 и u, с помощью сравнения значений функции f(x) в которых можно сократить интервал поиска.

Также следует отметить, что на первой итерации метод парабол требует измерения значений функции f(x) в крайних точках интервала оптимизации.

Реализованный алгоритм:

# Parabol method

def Parabol(a, b, epsilon):

    k = 1 # iteration number

    fn = 0 # func counting number

    ans = [(a, b)]

    x1 = a

    f1 = Func(x1)

    x3 = b

    f3 = Func(x3)

    fn = fn + 2

    OutputL("Parabol", abs(x1 - x3))

    while (abs(x3 - x1) >= epsilon):

        k = k + 1

        x2 = (x1 + x3) / 2

        f2 = Func(x2)

        u = x2 - (((x2 - x1) \*\* 2)\*(f2 - f3)-((x2 - x3) \*\* 2)\*(f2 - f1))/(2\*((x2 - x1)\*(f2 - f3)-(x2 - x3)\*(f2 - f1)))

        fu = Func(u)

        fn = fn + 2

        if (x1 < u and u < x2):

            x3 = x2

            x2 = u

            f3 = f2

            f2 = fu

        else:

            x1 = x2

            x2 = u

            f1 = f2

            f2 = fu

        OutputL("Parabol", abs(x1 - x3))

        ans.append([(x1, x3)])

    OutputNumberOfIterations("Parabol", k)

    OutputNumberOfFunc("Parabol", fn)

    return ans

Метод Брента

В данном методе на каждой итерации отслеживаются значения в шести точках (не обязательно различных): a, c, x, w, v, u. Точки a, c задают текущий интервал поиска решения, x – точка, соответствующая наименьшему значению функции, w – точка, соответствующая второму снизу значению функции, v – предыдущее значение w. В отличие от метода парабол, в методе Брента аппроксимирующая парабола строится с помощью трех наилучших точек x, w, v (в случае, если эти три точки различны и значения в них также различны). При этом минимум аппроксимирующей параболы u (см. Метод Парабол) принимается в качестве следующей точки оптимизационного процесса, если:

- u попадает внутрь интервала [a, c];

- u отстоит от точки x не более, чем на половину от длины предпредыдущего шага.

Описание алгоритма:

Реализованный алгоритм:

# Brent method

def Brent(a, c, epsilon):

    k\_iter = 1 # iteration number

    f\_iter = 0 # func counting number

    epsilon = epsilon / 2

    x = w = v = (a + c) / 2

    K = (3 - np.sqrt(5))/2

    fx = fw = fv = Func(x)

    f\_iter = f\_iter + 1

    d = e = c - a

    u = 0

    ans = [(a, c)]

    OutputL("Brent", abs(a - c))

    while (abs(c - a) / 2 > epsilon):

        k\_iter = k\_iter + 1

        g = e

        e = d

        flag = 0

        # if points x, v, w and func fx, fw, fv are different

        if (x != w and x != v and w != v and fx != fw and fx != fv and fw != fv):

            # parabol approximation

            x1 = min(x, min(w, v))

            x2 = x3 = f1 = f2 = f3 = 0

            if (x1 == x):

                f1 = fx

            elif (x1 == v):

                f1 = fv

            else :

                f1 = fw

            x3 = max(x, max(w, v))

            if (x3 == x):

                f3 = fx

            elif (x3 == v):

                f3 = fv

            else :

                f3 = fw

            if (x != x1 and x != x3):

                x2 = x

                f2 = fx

            elif (w != x1 and w != x3):

                x2 = w

                f2 = fw

            elif (v != x1 and v != x3):

                x2 = v

                f2 = fv

            if (x1 < x2 and x2 < x3 and f1 > f2 and f2 < f3):

                # take u

                flag = 1

                u = x2 - (((x2 - x1) \*\* 2)\*(f2 - f3)-((x2 - x3) \*\* 2)\*(f2 - f1))/(2\*((x2 - x1)\*(f2 - f3)-(x2 - x3)\*(f2 - f1)))

        if (flag == 1 and a + epsilon < u and u < c - epsilon and abs(u - x) < g / 2):

            e = abs(u - x)

        else: # if u wasn't taken priviously

            if (x < (c + a) / 2):

                # GoldenRation [x, c]

                u = x + K \* (c - x)

                e = c - x

            else:

                # GoldenRation [a, x]

                u = x - K \* (x - a)

                e = x - a

        if abs(u - x) < epsilon:

            # set min u between u and x

            u = x + np.sign(u - x) \* epsilon

        # count func(u)

        fu = Func(u)

        f\_iter = f\_iter + 1

        if (fu <= fx):

            if (u >= x):

                a = x

            else:

                c = x

            v = w

            w = x

            x = u

            fv = fw

            fw = fx

            fx = fu

        else:

            if (u >= x):

                c = u

            else:

                a = u

            if (fu <= fw or w == x):

                v = w

                w = u

                fv = fw

                fw = fu

            elif fu <= fv or v == x or v == w:

                v = u

                fv = fu

        OutputL("Brent", abs(a - c))

        ans.append([(a, c)])

    OutputNumberOfIterations("Brent", k\_iter)

    OutputNumberOfFunc("Brent", f\_iter)

    return ans

Полный код работы представлен здесь: <https://github.com/anastasia-zolotareva/primat-lab1>

сравнение алгоритмов минимизации функции и тестирование

Тесты для проверки работы и сравнения

Тестирование проходило для следующих промежутков [a, b]:

[-1, 1], [-2, 1], [-5, 2], [-5, 0], [-2, 0], [0, 1], [0, 2], [2.5, 7.5]

И каждый для значения ε=

0,1; 0,01; 0,001; 0,0001; 0,00001; 0,000001; 0,0000001; 0,00000001; 0,000000001

Код тестирования:

test\_data = [[-1, 1], [-2, 1], [-5, 2], [-5, 0], [-2, 0], [0, 1], [0, 2], [2.5, 7.5]]

for i in range(len(test\_data)):

    file = open("output/output\_tech.txt", "a")

    file.write("Test" + str(i + 1) + " ")

    file.close()

    a = test\_data[i][0]

    b = test\_data[i][1]

    epsilon = 1

    for j in range(9):

        epsilon = epsilon / 10

        file = open("output/output\_tech.txt", "a")

        file.write("with epsilon = " + str(epsilon) + "\n")

        file.close()

        ans1\_dych = Dichotomy(a, b, epsilon)

        ans1\_gr = GoldenRatio(a, b, epsilon)

        ans1\_fib = Fibonachi(a, b, epsilon)

        ans1\_par = Parabol(a, b, epsilon)

        ans1\_brent = Brent(a, b, epsilon)

Также параллельно производился вывод результатов в файлы для каждого из методов: количество итераций и вызова функции для каждого набора значений и значение длины lk для каждой итерации.

Таблица сравнения

На основании полученных данных были составлены сравнительные таблицы:

1) Количество итераций:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Test | | | iteration number | | | | |
| № | [a,b] | epsilon | Dychotomy | GoldenRation | Fibonachi | Parabol | Brent |
| 1 | [-1, 1] | 0,1 | 6 | 8 | 6 | 6 | 8 |
| 0,01 | 10 | 13 | 11 | 9 | 10 |
| 0,001 | 13 | 17 | 16 | 12 | 13 |
| 0,0001 | 16 | 22 | 21 | 16 | 17 |
| 0,00001 | 20 | 27 | 26 | 19 | 19 |
| 0,000001 | 23 | 32 | 30 | 22 | 22 |
| 0,0000001 | 26 | 36 | 35 | 26 | 25 |
| 0,00000001 | 30 | 41 | 40 | 29 | 27 |
| 0,000000001 | 33 | 46 | 45 | 32 | 30 |
| 2 | [-2, 1] | 0,1 | 7 | 9 | 7 | 6 | 9 |
| 0,01 | 10 | 13 | 12 | 10 | 11 |
| 0,001 | 14 | 18 | 17 | 13 | 15 |
| 0,0001 | 17 | 23 | 22 | 16 | 16 |
| 0,00001 | 20 | 28 | 26 | 20 | 20 |
| 0,000001 | 24 | 32 | 31 | 23 | 22 |
| 0,0000001 | 27 | 37 | 36 | 26 | 24 |
| 0,00000001 | 30 | 42 | 41 | 30 | 26 |
| 0,000000001 | 34 | 47 | 46 | 33 | 33 |
| 3 | [-5, 2] | 0,1 | 8 | 10 | 9 | 8 | 11 |
| 0,01 | 12 | 15 | 14 | 11 | 14 |
| 0,001 | 15 | 20 | 19 | 14 | 17 |
| 0,0001 | 18 | 25 | 23 | 18 | 19 |
| 0,00001 | 22 | 29 | 28 | 21 | 20 |
| 0,000001 | 25 | 34 | 33 | 24 | 25 |
| 0,0000001 | 28 | 39 | 38 | 28 | 27 |
| 0,00000001 | 31 | 44 | 42 | 31 | 32 |
| 0,000000001 | 35 | 49 | 47 | 34 | 30 |
| 4 | [-5, 0] | 0,1 | 8 | 10 | 8 | 7 | 10 |
| 0,01 | 11 | 14 | 13 | 10 | 15 |
| 0,001 | 14 | 19 | 18 | 14 | 20 |
| 0,0001 | 18 | 24 | 23 | 17 | 25 |
| 0,00001 | 21 | 29 | 27 | 20 | 29 |
| 0,000001 | 24 | 34 | 32 | 24 | 34 |
| 0,0000001 | 28 | 38 | 37 | 27 | 39 |
| 0,00000001 | 31 | 43 | 42 | 30 | 44 |
| 0,000000001 | 34 | 48 | 47 | 34 | 48 |
| 5 | [-2, 0] | 0,1 | 6 | 8 | 6 | 6 | 8 |
| 0,01 | 10 | 13 | 11 | 9 | 13 |
| 0,001 | 13 | 17 | 16 | 12 | 18 |
| 0,0001 | 16 | 22 | 21 | 16 | 23 |
| 0,00001 | 20 | 27 | 26 | 19 | 27 |
| 0,000001 | 23 | 32 | 30 | 22 | 32 |
| 0,0000001 | 26 | 36 | 35 | 26 | 37 |
| 0,00000001 | 30 | 41 | 40 | 29 | 42 |
| 0,000000001 | 33 | 46 | 45 | 32 | 47 |
| 6 | [0, 1] | 0,1 | 5 | 6 | 5 | 5 | 6 |
| 0,01 | 9 | 11 | 10 | 8 | 11 |
| 0,001 | 12 | 16 | 15 | 11 | 15 |
| 0,0001 | 15 | 21 | 19 | 15 | 20 |
| 0,00001 | 19 | 25 | 24 | 18 | 25 |
| 0,000001 | 22 | 30 | 29 | 21 | 30 |
| 0,0000001 | 25 | 35 | 34 | 25 | 35 |
| 0,00000001 | 29 | 40 | 38 | 28 | 39 |
| 0,000000001 | 32 | 45 | 43 | 31 | 44 |
| 7 | [0, 2] | 0,1 | 6 | 8 | 6 | 6 | 7 |
| 0,01 | 10 | 13 | 11 | 9 | 12 |
| 0,001 | 13 | 17 | 16 | 12 | 17 |
| 0,0001 | 16 | 22 | 21 | 16 | 22 |
| 0,00001 | 20 | 27 | 26 | 19 | 26 |
| 0,000001 | 23 | 32 | 30 | 22 | 31 |
| 0,0000001 | 26 | 36 | 35 | 26 | 36 |
| 0,00000001 | 30 | 41 | 40 | 29 | 41 |
| 0,000000001 | 33 | 46 | 45 | 32 | 46 |
| 8 | [2,5, 7,5] | 0,1 | 8 | 10 | 8 | 7 | 9 |
| 0,01 | 11 | 14 | 13 | 10 | 10 |
| 0,001 | 14 | 19 | 18 | 14 | 14 |
| 0,0001 | 18 | 24 | 23 | 17 | 18 |
| 0,00001 | 21 | 29 | 27 | 20 | 18 |
| 0,000001 | 24 | 34 | 32 | 24 | 22 |
| 0,0000001 | 28 | 38 | 37 | 27 | 24 |
| 0,00000001 | 27 | 43 | 42 | 30 | 30 |
| 0,000000001 | 23 | 48 | 47 | 34 | 35 |

Далее для удобства были построены графики по которым видно, что в среднем это число равномерно увеличивается прямопропорционально точности, но уровень наклона условной линии тренда таков, что чем большее число итераций необходимо вычислить, тем более выгодно использовать комбинированный алгоритм Брента. У него выше скорость сокращения длины искомого отрезка за счет совмещения нескольких стратегий для поиска: использование параболы и метода Золотого сечения. Первый работает быстро в малой окрестности оптимального решения (что видно на графике), но долго и неустойчиво на начальных стадиях итерационного процесса. А метод Золотого сечения стабильно находит минимум за фиксированное число итераций. Поэтому при комбинировании их в один, метод Брента работает намного стабильнее метода Парабол, и быстрее методов Золотого сечения, Дихотомии и Фибоначчи. Однако он может работать долго на отрезках, имеющих минимум на граничном значении функции – тогда его работа приближается к скорости метода Золотого Сечения.

При малый значениях точности, наоборот меньшее ресурсов тратится на более простые алгоритмы, как метод Дихотомии.

2) Количество вызова функции:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Test | | | function iteration number | | | | |
| № | [a,b] | epsilon | Dychotomy | GoldenRation | Fibonachi | Parabol | Brent |
| 1 | [-1, 1] | 0,1 | 12 | 9 | 7 | 12 | 8 |
| 0,01 | 20 | 14 | 12 | 18 | 10 |
| 0,001 | 26 | 18 | 17 | 24 | 13 |
| 0,0001 | 32 | 23 | 22 | 32 | 17 |
| 0,00001 | 40 | 28 | 27 | 38 | 19 |
| 0,000001 | 46 | 33 | 31 | 44 | 22 |
| 0,0000001 | 52 | 37 | 36 | 52 | 25 |
| 0,00000001 | 60 | 42 | 41 | 58 | 27 |
| 0,000000001 | 66 | 47 | 46 | 64 | 30 |
| 2 | [-2, 1] | 0,1 | 14 | 10 | 8 | 12 | 9 |
| 0,01 | 20 | 14 | 13 | 20 | 11 |
| 0,001 | 28 | 19 | 18 | 26 | 15 |
| 0,0001 | 34 | 24 | 23 | 32 | 16 |
| 0,00001 | 40 | 29 | 27 | 40 | 20 |
| 0,000001 | 48 | 33 | 32 | 46 | 22 |
| 0,0000001 | 54 | 38 | 37 | 52 | 24 |
| 0,00000001 | 60 | 43 | 42 | 60 | 26 |
| 0,000000001 | 68 | 48 | 47 | 66 | 33 |
| 3 | [-5, 2] | 0,1 | 16 | 11 | 10 | 16 | 11 |
| 0,01 | 24 | 16 | 15 | 22 | 14 |
| 0,001 | 30 | 21 | 20 | 28 | 17 |
| 0,0001 | 36 | 26 | 24 | 36 | 19 |
| 0,00001 | 44 | 30 | 29 | 42 | 20 |
| 0,000001 | 50 | 35 | 34 | 48 | 25 |
| 0,0000001 | 56 | 40 | 39 | 56 | 27 |
| 0,00000001 | 62 | 45 | 43 | 62 | 32 |
| 0,000000001 | 70 | 50 | 48 | 68 | 30 |
| 4 | [-5, 0] | 0,1 | 16 | 11 | 9 | 14 | 10 |
| 0,01 | 22 | 15 | 14 | 20 | 15 |
| 0,001 | 28 | 20 | 19 | 28 | 20 |
| 0,0001 | 36 | 25 | 24 | 34 | 25 |
| 0,00001 | 42 | 30 | 28 | 40 | 29 |
| 0,000001 | 48 | 35 | 33 | 48 | 34 |
| 0,0000001 | 56 | 39 | 38 | 54 | 39 |
| 0,00000001 | 62 | 44 | 43 | 60 | 44 |
| 0,000000001 | 68 | 49 | 48 | 68 | 48 |
| 5 | [-2, 0] | 0,1 | 12 | 9 | 7 | 12 | 8 |
| 0,01 | 20 | 14 | 12 | 18 | 13 |
| 0,001 | 26 | 18 | 17 | 24 | 18 |
| 0,0001 | 32 | 23 | 22 | 32 | 23 |
| 0,00001 | 40 | 28 | 27 | 38 | 27 |
| 0,000001 | 46 | 33 | 31 | 44 | 32 |
| 0,0000001 | 52 | 37 | 36 | 52 | 37 |
| 0,00000001 | 60 | 42 | 41 | 58 | 42 |
| 0,000000001 | 66 | 47 | 46 | 64 | 47 |
| 6 | [0, 1] | 0,1 | 10 | 7 | 6 | 10 | 6 |
| 0,01 | 18 | 12 | 11 | 16 | 11 |
| 0,001 | 24 | 17 | 16 | 22 | 15 |
| 0,0001 | 30 | 22 | 20 | 30 | 20 |
| 0,00001 | 38 | 26 | 25 | 36 | 25 |
| 0,000001 | 44 | 31 | 30 | 42 | 30 |
| 0,0000001 | 50 | 36 | 35 | 50 | 35 |
| 0,00000001 | 58 | 41 | 39 | 56 | 39 |
| 0,000000001 | 64 | 46 | 44 | 62 | 44 |
| 7 | [0, 2] | 0,1 | 12 | 9 | 7 | 12 | 7 |
| 0,01 | 20 | 14 | 12 | 18 | 12 |
| 0,001 | 26 | 18 | 17 | 24 | 17 |
| 0,0001 | 32 | 23 | 22 | 32 | 22 |
| 0,00001 | 40 | 28 | 27 | 38 | 26 |
| 0,000001 | 46 | 33 | 31 | 44 | 31 |
| 0,0000001 | 52 | 37 | 36 | 52 | 36 |
| 0,00000001 | 60 | 42 | 41 | 58 | 41 |
| 0,000000001 | 66 | 47 | 46 | 64 | 46 |
| 8 | [2,5, 7,5] | 0,1 | 16 | 11 | 9 | 14 | 9 |
| 0,01 | 22 | 15 | 14 | 20 | 10 |
| 0,001 | 28 | 20 | 19 | 28 | 14 |
| 0,0001 | 36 | 25 | 24 | 34 | 18 |
| 0,00001 | 42 | 30 | 28 | 40 | 18 |
| 0,000001 | 48 | 35 | 33 | 48 | 22 |
| 0,0000001 | 56 | 39 | 38 | 54 | 24 |
| 0,00000001 | 54 | 44 | 43 | 60 | 30 |
| 0,000000001 | 46 | 49 | 48 | 68 | 35 |

На основании этих данный получаем следующую гистограмму, где видно, как в зависимости от алгоритма изменяется это число:

Меньше всего вызовов происходит при алгоритме Брента, поскольку равняется числу итераций), в методе Фибоначчи и Золотого сечения значение на один больше, так как требуют дополнительное вычисление в начале. Больше всего вызовов в методах Дихотомии и Парабол – в два раза больше числа итераций.

3) Таблицы для анализа изменения длины отрезка lk и основании их получены значения среднего уменьшения длины для каждого метода:

|  |  |
| --- | --- |
| Метод | Среднее значение |
| Дихотомии | 1,893710114 (в 2 раза, но используется поскольку добавляется ±δ, то данное значение получилось меньше) |
| Золотого Сечения | 1,618033989 = const |
| Фибоначчи | 1,614774705 |
| Парабол | 2 = const |
| Брента | 1,994723931 (приближено к 4-м, поскольку работает за счет метода парабол, но зачастую < 1,2-1,3) |

Более подробно изменение отрезков можно посмотреть в приложенной таблице.

вывод

В зависимости от выбираемого промежутка функции реализованные методы показывают разную скорость поиска минимума. Также они имеют разную степень надежности – так для некоторых функций метод комбинированный метод Брента может не гарантировать правильность нахождения локального минимума, поскольку использует допущение в шаге (ограничение скорости уменьшения интервала за счет использования препредыдущего значения).

По скорости более высокие показатели по скорости у алгоритмов Брента и Парабол, но они выигрывают только при большом значении числа итераций (большего значения точности и расстояния между изначальными границами отрезка), но их эффективность падает при малых значениях или при определенном виде функции (когда, например, для параболы функция имеет экспоненциальную форму).

Также следует отметить, что число вычисления функции также может иметь критическое значение при вычислениях, и занимать ресурсы.

Таким образом при выборе алгоритма поиска минимума стоит учитывать все факторы: вид функции, скорость сокращения, риски ошибок, требуемая точность, сложность вычисления функции.