Министерство науки и образования РФ

Федеральное государственное бюджетное учреждение

высшего образования

**«Тверской государственный технический университет»**

(ТвГТУ)

Кафедра программного обеспечения

**Отчет по лабораторной работе №3**

По дисциплине: «Компьютерные системы моделирования»

Тема: «Аналитическое моделирование. Применение численных методов для решения задачи оптимизации»

|  |
| --- |
| Выполнил:  студент группы  Б.ПИН.РИС - 22.06  Иванов А.М. |
| Проверила:  старший преподаватель  кафедры ПО  Корнеева Е.И. |

Тверь 2025

**Оглавление**

[**Цель и постановка задачи для варианта.** 3](#_Toc189679068)

[**Описание алгоритмов.** 4](#_Toc189679069)

[**Отчет по реализации.** 10](#_Toc189679070)

[**Скриншоты** 30](#_Toc189679071)

[**Таблица зависимости количества вычислений от точности** 31](#_Toc189679072)

[**Ссылка на репозиторий** 32](#_Toc189679073)

[**Выводы по результатам решения задачи.** 33](#_Toc189679074)

# **Цель и постановка задачи для варианта.**

**Цель работы**: Научиться реализовывать на языке python методы поиска безусловного минимума или максимума функции и применять их к решению трансцендентных уравнений.

**Постановка задачи:**

Найти безусловный минимум функции одной переменной , представленной трансцендентным уравнением. Минимумом функции является такая точка , что . Значение минимальной точки вычисляется приближенно с заданной точностью . При вычислении применить метод дихотомии и метод «золотого сечения», а также метод, использующий значение производных целевой функции – метод Ньютона (касательных).

**Функция:**

# **Описание алгоритмов.**

Алгоритмы основываются на возможность определения значений в заданных точках.

***Метод бисекции (разделение отрезка пополам)***

В методе пробная точка располагается в середине рассматриваемого отрезка , т.е.

При вычислении использовать производную, Результатом метода является значение середины последнего из найденных отрезков при том, что неравенство должно быть достигнуто. Если точность не достигнута, то меняем значение интервала и рассматриваем новую точку .

Алгоритм

0) Задать .

1) Вычислить значение

2)

Иначе

3)

Завершение (критерий останова)

.

***Метод дихотомии (разделение пополам с максимальным уменьшением)***

В этом методе пробные точки и располагаются близко к середине очередного отрезка , т.е.

где – малое число ( ). Название метода объясняется тем, что отношение длин нового и исходного отрезков

близко по значению к . Для любых точек , величина , поэтому указанный выбор пробных точек объясняется стремлением обеспечить максимально возможное относительное уменьшение отрезка на каждой итерации поиска .

Результатом метода является приближенное значение середины последнего из найденных отрезков при том, что неравенство должно быть достигнуто.

*Алгоритм*

Шаг 1. Определить и по формуле (1) и вычислить и . Перейти к шагу 2.

Шаг 2. Сравнить и . Если , то перейти к отрезку , положив , иначе – к отрезку , положив . Перейти к шагу 3.

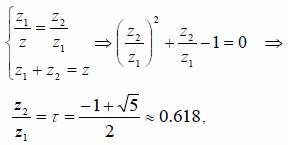
Шаг 3. Найти достигнутую точность , где .

Если , то перейти к следующей итерации, вернувшись к шагу 1. Если , то завершить поиск , перейдя к шагу 4.

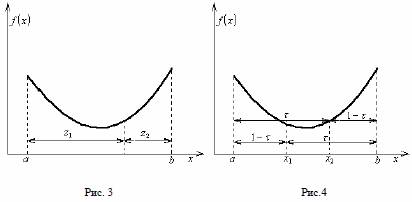
Шаг 4. Положить .

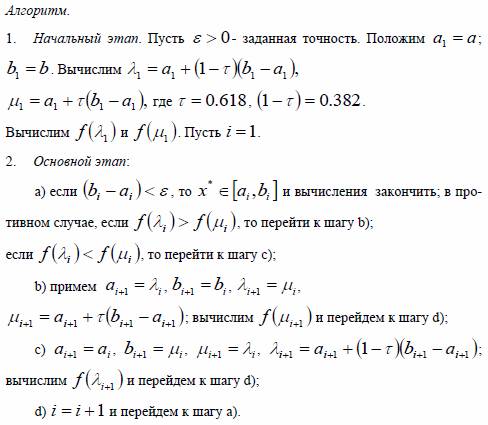
***Метод «золотого сечения»***

Метод золотого сечения – метод поиска экстремума действительной функции одной переменной на заданном отрезке. В основе метода лежит принцип деления отрезка в пропорциях золотого сечения.



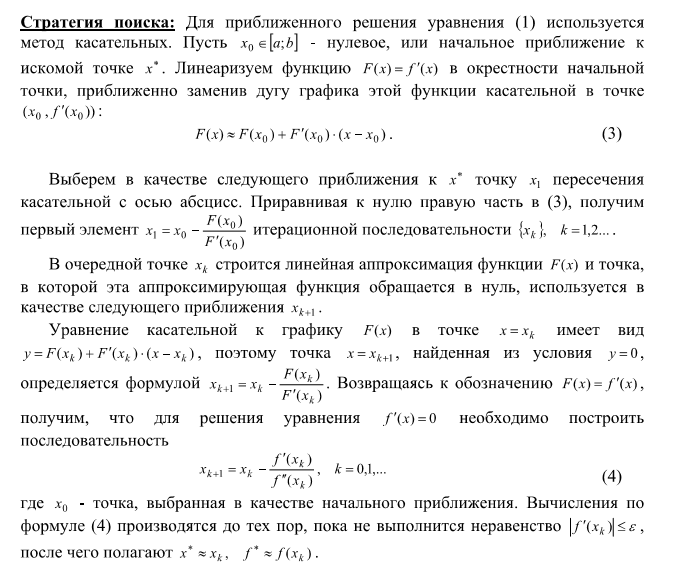
На рисунке выше – это значение золотого сечения.





***Метод, использующий информацию о производных целевой функции***

***Метод Ньютона (касательных)***



# **Отчет по реализации.**

**Программная реализация**

**Структура библиотеки**

Библиотека организована вокруг абстрактного базового класса Function, который определяет интерфейс для всех математических функций. Классы, унаследованные от Function, реализуют конкретные виды функций и их поведение.

**Абстрактный базовый класс Function**

class Function(ABC, Callable[[float], float]):

@abstractmethod

def value\_at(self, x: float) -> float:

pass

def \_\_call\_\_(self, x: float) -> float:

return self.value\_at(x)

def \_\_str\_\_(self) -> str:

return "f(x)"

def derivative(self) -> 'Function':

return AnotherNumericalDerivative(self)

**Метод value\_at** : Абстрактный метод, который должен быть реализован в каждом подклассе. Он вычисляет значение функции в точке x.

**Метод \_\_call\_\_** : Позволяет вызывать объект класса как функцию, используя синтаксис f(x).

**Метод \_\_str\_\_** : Возвращает строковое представление функции.

**Метод derivative** : Возвращает численную производную функции, используя класс AnotherNumericalDerivative.

**Конкретные реализации функций**

**Класс LaboratoryFunction**

class LaboratoryFunction(Function):

def value\_at(self, x: float) -> float:

return x \* math.exp(x) \* (math.sin(x))\*\*2

def \_\_str\_\_(self) -> str:

return "f(x) = x \* e^x \* (sin x)^2"

def derivative(self) -> 'Function':

return CustomStringAndLambdaFunction(

"f'(x) = e^x \* (sin x)^2 + 2x \* sin x \* cos x + x \* (sin x)^2",

lambda x: math.exp(x) \* (math.sin(x)\*\*2 + 2\*x\*math.sin(x)\*math.cos(x) + x\*math.sin(x)\*\*2)

)

Реализует исследуемую в работе функцию f(x)=x⋅e(x)⋅(sinx) ^ 2

Переопределяет метод derivative, возвращая аналитическую производную в виде объекта CustomStringAndLambdaFunction.

**Класс CustomLambdaFunction**

class CustomLambdaFunction(Function):

def \_\_init\_\_(self, func: Callable[[float], float]) -> None:

self.func = func

def value\_at(self, x: float) -> float:

return self.func(x)

Представляет функцию, заданную через лямбда-выражение.

Метод value\_at вызывает переданную лямбда-функцию.

**Класс CustomStringFunction**

class CustomStringFunction(Function):

def \_\_init\_\_(self, expression: str) -> None:

self.expression = expression

def value\_at(self, x: float) -> float:

return eval(self.expression, {"x": x, "math": math})

def \_\_str\_\_(self) -> str:

return f"f(x) = {self.expression}"

Представляет функцию, заданную строковым выражением.

Метод value\_at вычисляет значение выражения с использованием eval.

**Класс CustomStringAndLambdaFunction**

class CustomStringAndLambdaFunction(Function):

def \_\_init\_\_(self, expression: str, lambda\_func: Callable[[float], float]) -> None:

self.expression = expression

self.lambda\_func = lambda\_func

def value\_at(self, x: float) -> float:

return self.lambda\_func(x)

def \_\_str\_\_(self) -> str:

return f"f(x) = {self.expression}"

Комбинирует строковое представление функции и лямбда-функцию для вычисления значения.

**Численное дифференцирование**

**Класс NumericalDerivative**

class NumericalDerivative(Function):

def \_\_init\_\_(self, function: Function, h: float = 1e-5) -> None:

self.function = function

self.h = h

def value\_at(self, x: float) -> float:

h = max(self.h, abs(x) \* 1e-5)

return (self.function.value\_at(x + h) - self.function.value\_at(x - h)) / (2 \* h)

def \_\_str\_\_(self) -> str:

return f"d({self.function})/dx"

Реализует численное дифференцирование по формуле центральной разности.

Параметр h определяет шаг для вычисления производной.

**Класс AnotherNumericalDerivative**

class AnotherNumericalDerivative(NumericalDerivative):

def \_\_init\_\_(self, function: Function, h: float = 1e-7) -> None:

self.function = function

self.h = h

def value\_at(self, x: float) -> float:

return (self.function.value\_at(x) - self.function.value\_at(x - self.h)) / self.h

Реализует альтернативный метод численного дифференцирования с использованием разностной формулы первого порядка.

**Фабричные методы**

Класс Functions предоставляет статические методы для создания объектов различных типов функций:

class Functions:

@staticmethod

def lab\_function() -> Function:

return LaboratoryFunction()

@staticmethod

def new\_from\_lambda(lambda\_func: Callable[[float], float]) -> Function:

return CustomLambdaFunction(lambda\_func)

@staticmethod

def new\_from\_string(expression: str) -> Function:

return CustomStringFunction(expression)

**lab\_function**: Создает экземпляр LaboratoryFunction.

**new\_from\_lambda**: Создает функцию на основе лямбда-выражения.

**new\_from\_string**: Создает функцию из строкового выражения.

**Вспомогательные классы**

**Класс Interval**

Класс Interval представляет числовой интервал и предоставляет методы для работы с ним.

class Interval:

def \_\_init\_\_(self, start: float, end: float) -> None:

self.\_start = start

self.\_end = end

def from\_(self) -> float:

return self.\_start

def to(self) -> float:

return self.\_end

def traverse(self, mapper: Callable[[float], float] = None, step: float = 1e-7) -> List[float]:

x = self.\_start

results = []

while x <= self.\_end:

if mapper is None:

results.append(x)

else:

results.append(mapper(x))

x += step

return results

Методы from\_ и to : Возвращают начало и конец интервала соответственно.

Метод traverse :

* + Проходит по всем точкам интервала с заданным шагом (step).
  + Если передан параметр mapper, применяет его к каждой точке интервала.
  + Возвращает список значений, полученных в результате обхода.

**Класс GraphBuilder**

class GraphBuilder:

def \_\_init\_\_(self, function: Function, interval: Interval, step: float = 0.01) -> None:

self.function = function

self.interval = interval

self.step = step

def build(self) -> None:

x = self.interval.traverse(mapper=None, step=self.step)

y = self.interval.traverse(self.function, self.step)

plt.plot(x, y)

plt.xlabel('x')

plt.ylabel('f(x)')

plt.title(f'График функции {self.function}')

plt.grid()

plt.show()

Параметры конструктора :

* + function: Математическая функция, график которой нужно построить.
  + interval: Интервал, на котором строится график.
  + step: Шаг для дискретизации интервала.

Метод build :

* + Использует метод traverse класса Interval для получения координат x и значений функции y.
  + Строит график с помощью библиотеки matplotlib.

**Класс Optional**

Класс Optional предоставляет механизм для работы с необязательными значениями, аналогичный Optional в Java.

class Optional[T]:

def \_\_init\_\_(self, value: T = None):

self.\_value = value

def is\_present(self) -> bool:

return self.\_value is not None

def get(self) -> T:

if self.\_value is None:

raise ValueError("No value present")

return self.\_value

def or\_else(self, other: T) -> T:

return self.\_value if self.\_value is not None else other

def \_\_str\_\_(self):

return f"Optional({self.\_value})"

Метод is\_present : Проверяет, содержит ли объект значение.

Метод get : Возвращает значение, если оно присутствует, иначе выбрасывает исключение.

Метод or\_else : Возвращает значение, если оно присутствует, иначе возвращает альтернативное значение.

Метод \_\_str\_\_ : Возвращает строковое представление объекта.

**Поиск экстремумов функций**

**Класс Extremum и его подклассы**

Класс Extremum представляет собой абстракцию для хранения информации об экстремуме функции: точке, в которой он достигается, и значении функции в этой точке.

class Extremum:

def \_\_init\_\_(self, point: float, value: float) -> None:

self.\_point = point

self.\_value = value

@property

def point(self) -> float:

return self.\_point

@property

def value(self) -> float:

return self.\_value

def \_\_str\_\_(self) -> str:

return f"Extremum at x = {self.point}, value = {self.value}"

**Подкласс Minimum :**

Представляет минимум функции.

Переопределяет метод \_\_str\_\_ для более информативного вывода.

class Minimum(Extremum):

def \_\_str\_\_(self) -> str:

return f"Minimum at x = {self.point}, value = {self.value}"

**Подкласс Maximum :**

Представляет максимум функции.

Аналогично переопределяет метод \_\_str\_\_.

class Maximum(Extremum):

def \_\_str\_\_(self) -> str:

return f"Maximum at x = {self.point}, value = {self.value}"

**Интерфейс ExtremumFinder**

Интерфейс ExtremumFinder определяет метод для поиска экстремума функции на заданном интервале.

class ExtremumFinder(ABC):

@abstractmethod

def find\_extremum(self, function: Function, interval: Interval, epsilon: float = 1e-7) -> Optional[Extremum]:

pass

Метод find\_extremum должен быть реализован в каждом подклассе.

Параметры:

function: Функция, для которой ищется экстремум.

interval: Интервал, на котором производится поиск.

epsilon: Точность поиска.

**Реализация численных методов поиска экстремумов**

**Метод дихотомии (DichotomyMethod)**

Метод дихотомии — это численный метод, который последовательно делит интервал пополам, чтобы найти экстремум.

class DichotomyMethod(IterationalExtremumFinder):

def \_\_init\_\_(self, context: IterationalContext):

super().\_\_init\_\_(context)

def find\_extremum(self, function: Function, interval: Interval, epsilon: float = 1e-7) -> Optional[Extremum]:

a, b = interval.from\_(), interval.to()

derivative = function.derivative()

if derivative.value\_at(a) \* derivative.value\_at(b) > 0:

raise Exception("No extremum in the interval")

is\_minimum = True

while (b - a) / 2 > epsilon:

self.count\_iteration()

x1 = (b + a - epsilon) / 2

x2 = (b + a + epsilon) / 2

if function.value\_at(x1) <= function.value\_at(x2):

b = x2

is\_minimum = True

else:

a = x1

is\_minimum = False

point = (a + b) / 2

if is\_minimum:

return Minimum(point, function.value\_at(point))

else:

return Maximum(point, function.value\_at(point))

**Логика работы :**

Делит интервал на две части.

Сравнивает значения функции в двух точках, чтобы определить направление движения.

Продолжает процесс до достижения заданной точности (epsilon).

**Метод золотого сечения (GoldenSectionMethod)**

Метод золотого сечения — это улучшенный вариант метода дихотомии, который использует пропорцию золотого сечения для выбора точек.

class GoldenSectionMethod(IterationalExtremumFinder):

golden\_section\_tau: float = (math.sqrt(5) - 1) / 2

def \_\_init\_\_(self, context: IterationalContext):

super().\_\_init\_\_(context)

def find\_extremum(self, function: Function, interval: Interval, epsilon: float = 1e-7) -> Optional[Extremum]:

a, b = interval.from\_(), interval.to()

derivative = function.derivative()

if derivative.value\_at(a) \* derivative.value\_at(b) > 0:

raise Exception("No extremum in the interval")

lambda1 = a + (1 - self.golden\_section\_tau) \* (b - a)

mu1 = a + self.golden\_section\_tau \* (b - a)

is\_minimum = True

while abs(b - a) > epsilon:

self.count\_iteration()

if function.value\_at(lambda1) < function.value\_at(mu1):

b = mu1

mu1 = lambda1

lambda1 = a + (1 - self.golden\_section\_tau) \* (b - a)

is\_minimum = True

else:

a = lambda1

lambda1 = mu1

mu1 = a + self.golden\_section\_tau \* (b - a)

is\_minimum = False

point = (a + b) / 2

if is\_minimum:

return Minimum(point, function.value\_at(point))

else:

return Maximum(point, function.value\_at(point))

**Особенности** :

Использует константу золотого сечения для выбора точек.

Более эффективен, чем метод дихотомии, так как требует меньшего количества вычислений.

**Контекст итераций**

Классы IterationalContext, UnlimitedIterationalContext и LimitedIterationalContext обеспечивают контроль над количеством итераций.

class IterationalContext(ABC):

def \_\_init\_\_(self):

self.iteration\_count = 0

@abstractmethod

def count\_iteration(self):

pass

def drop\_iterations(self):

self.iteration\_count = 0

def get\_iteration\_count(self) -> int:

return self.iteration\_count

**UnlimitedIterationalContext** : Не ограничивает количество итераций.

**LimitedIterationalContext** : Ограничивает количество итераций, выбрасывая исключение при превышении лимита.

**Поиск интервалов с экстремумами**

Класс ExtremumIntervalDetector и его подкласс NewtonExtremumIntervalDetector позволяют находить интервалы, содержащие экстремумы. Класс NewtonExtremumIntervalDetector определяет точки, где производная функции равна нулю, методом Ньютона, и возвращает список интервалов с окрестностями этой точки с точностью до шага, передаваемого в конструкторе класса.

*class* ExtremumIntervalDetector(ABC):

*def* \_\_init\_\_(*self*, *function*: Function, *step*: float = None, *search\_interval*: Interval = Interval(-1000, 1000)):

*self*.function = *function*

*self*.search\_interval = *search\_interval*

        if *step* is None:

            if *search\_interval* is not None:

*self*.step = (*search\_interval*.to() - *search\_interval*.from\_()) / 10

            else:

*self*.step = 0.1

        else:

*self*.step = *step*

*def* find\_extremum\_intervals(*self*) -> List[Interval]:

        derivative = *self*.function.derivative()

        x\_values = np.arange(*self*.search\_interval.from\_(), *self*.search\_interval.to(), *self*.step)

        potential\_intervals = []

        for x in x\_values:

            root = *self*.find\_root(derivative, x)

            if root is not None:

                potential\_intervals.append(Interval(root - *self*.step, root + *self*.step))

        return potential\_intervals

    @abstractmethod

*def* find\_root(*self*, *derivative*: Function, *initial\_guess*: float) -> Optional[float]:

        pass

*class* NewtonExtremumIntervalDetector(ExtremumIntervalDetector):

*def* find\_root(*self*, *derivative*: Function, *initial\_guess*: float, *tolerance*: float = 1e-5, *max\_iterations*: int = 1000) -> Optional[float]:

        x = *initial\_guess*

        second\_derivative = *derivative*.derivative()

        for \_ in range(*max\_iterations*):

            f\_prime\_x = *derivative*.value\_at(x)

            if abs(f\_prime\_x) < *tolerance*:

                return x

            f\_double\_prime\_x = second\_derivative.value\_at(x)

            if f\_double\_prime\_x == 0:

                return None

            x -= f\_prime\_x / f\_double\_prime\_x

        return None

**Основной блок программы**

Основной блок программы (main) выполняет комплексную работу по анализу функции, поиску экстремумов и сравнению эффективности численных методов. Рассмотрим его структуру и логику работы.

def main():

function = Functions.lab\_function()

search\_interval = Interval(0, 20)

finder = NewtonExtremumIntervalDetector(function, search\_interval=search\_interval)

intervals = finder.find\_extremum\_intervals()

graph\_function = GraphBuilder(function, search\_interval, 0.01)

graph\_function.build()

epsilons = [1e-2, 1e-3, 1e-4, 1e-5, 1e-6, 1e-7, 1e-8, 1e-9, 1e-10, 1e-11, 1e-12, 1e-13, 1e-14, 1e-15]

golden\_context = LimitedIterationalContext(max\_iterations=200\_000)

dichotomy\_context = LimitedIterationalContext(max\_iterations=200\_000)

golden\_method = GoldenSectionMethod(golden\_context)

dichotomy\_method = DichotomyMethod(dichotomy\_context)

print(f"{'Точность':<10} {'Золотое сечение':<20} {'Дихотомия':<10}")

for epsilon in epsilons:

golden\_context.drop\_iterations()

dichotomy\_context.drop\_iterations()

for interval in intervals:

try:

golden\_method.find\_extremum(function, interval, epsilon)

except Exception as e:

pass

try:

dichotomy\_method.find\_extremum(function, interval, epsilon)

except Exception as e:

pass

print(f"{epsilon:<10} {golden\_context.get\_iteration\_count():<20} {dichotomy\_context.get\_iteration\_count():<10}")

**Пошаговый разбор**

**Выбор функции и интервала поиска**

search\_interval = Interval(0, 20)

Интервал поиска : Устанавливается диапазон от 0 до 20.

**Поиск интервалов с экстремумами**

finder = NewtonExtremumIntervalDetector(function, search\_interval=search\_interval)

intervals = finder.find\_extremum\_intervals()

Класс NewtonExtremumIntervalDetector находит интервалы, содержащие экстремумы, путем анализа производной функции.

Результат сохраняется в списке intervals.

**Построение графика функции**

graph\_function = GraphBuilder(function, search\_interval, 0.01)

graph\_function.build()

График функции строится на интервале [0, 20] с шагом дискретизации 0.01.

Визуализация помогает понять поведение функции и расположение экстремумов.

**Сравнение методов поиска экстремумов**

epsilons = [1e-2, 1e-3, 1e-4, 1e-5, 1e-6, 1e-7, 1e-8, 1e-9, 1e-10, 1e-11, 1e-12, 1e-13, 1e-14, 1e-15]

golden\_context = LimitedIterationalContext(max\_iterations=200\_000)

dichotomy\_context = LimitedIterationalContext(max\_iterations=200\_000)

golden\_method = GoldenSectionMethod(golden\_context)

dichotomy\_method = DichotomyMethod(dichotomy\_context)

Метод золотого сечения и метод дихотомии сравниваются по количеству итераций для достижения заданной точности (epsilon).

Для каждого метода создается контекст с ограничением на количество итераций (200,000).

**Цикл по значениям точности**

for epsilon in epsilons:

golden\_context.drop\_iterations()

dichotomy\_context.drop\_iterations()

for interval in intervals:

try:

golden\_method.find\_extremum(function, interval, epsilon)

except Exception as e:

pass

try:

dichotomy\_method.find\_extremum(function, interval, epsilon)

except Exception as e:

pass

print(f"{epsilon:<10} {golden\_context.get\_iteration\_count():<20} {dichotomy\_context.get\_iteration\_count():<10}")

Для каждого значения epsilon:

Сбрасывается счетчик итераций для обоих методов.

Производится поиск экстремумов на каждом интервале из списка intervals.

Если поиск завершается успешно, увеличивается счетчик итераций.

Результаты выводятся в виде таблицы, где указаны:

* Точность (epsilon).
* Количество итераций для метода золотого сечения.
* Количество итераций для метода дихотомии.

**Пример вывода программы**

Точность Золотое сечение Дихотомия

0.01 13 9

0.001 18 12

0.0001 23 16

1e-05 27 19

1e-06 32 22

1e-07 37 26

1e-08 42 29

1e-09 46 32

1e-10 51 36

1e-11 56 39

1e-12 61 42

1e-13 66 46

1e-14 70 49

1e-15 75 52

Для метода дихотомии

Minimum at x = -1.4231988133828126, value = -0.33549226559445366 26, где 26 – количество итераций

Для метода «Золотого сечения»

Minimum at x = -1.42319883552639, value = -0.3354922655944534 37, где 37 – количество итераций

Метод золотого сечения требует больше итераций, чем метод дихотомии, для достижения той же точности.

Разница становится более заметной при уменьшении epsilon.

**Анализ результатов**

В ходе выполнения программы и анализа результатов было обнаружено, что метод дихотомии в некоторых случаях показывает лучшую производительность по сравнению с методом золотого сечения. Рассмотрим возможные причины этого явления и проведем более детальный анализ.

**Эффективность методов**

В нашем случае метод дихотомии оказался немного эффективнее, что может быть связано с особенностями функции.

Метод дихотомии делит интервал пополам на каждом шаге, что обеспечивает равномерное сужение области поиска. Это может быть особенно полезно для функций с резкими изменениями или сложной структурой экстремумов.

Теоретически метод золотого сечения должен быть более эффективным, так как он использует оптимальное соотношение точек для минимизации количества вычислений.

Однако на практике его эффективность может снижаться из-за особенностей реализации или поведения функции. Например, если функция имеет много локальных экстремумов или неравномерное изменение производной, метод золотого сечения может требовать дополнительных итераций для достижения той же точности.

**Ограничения**

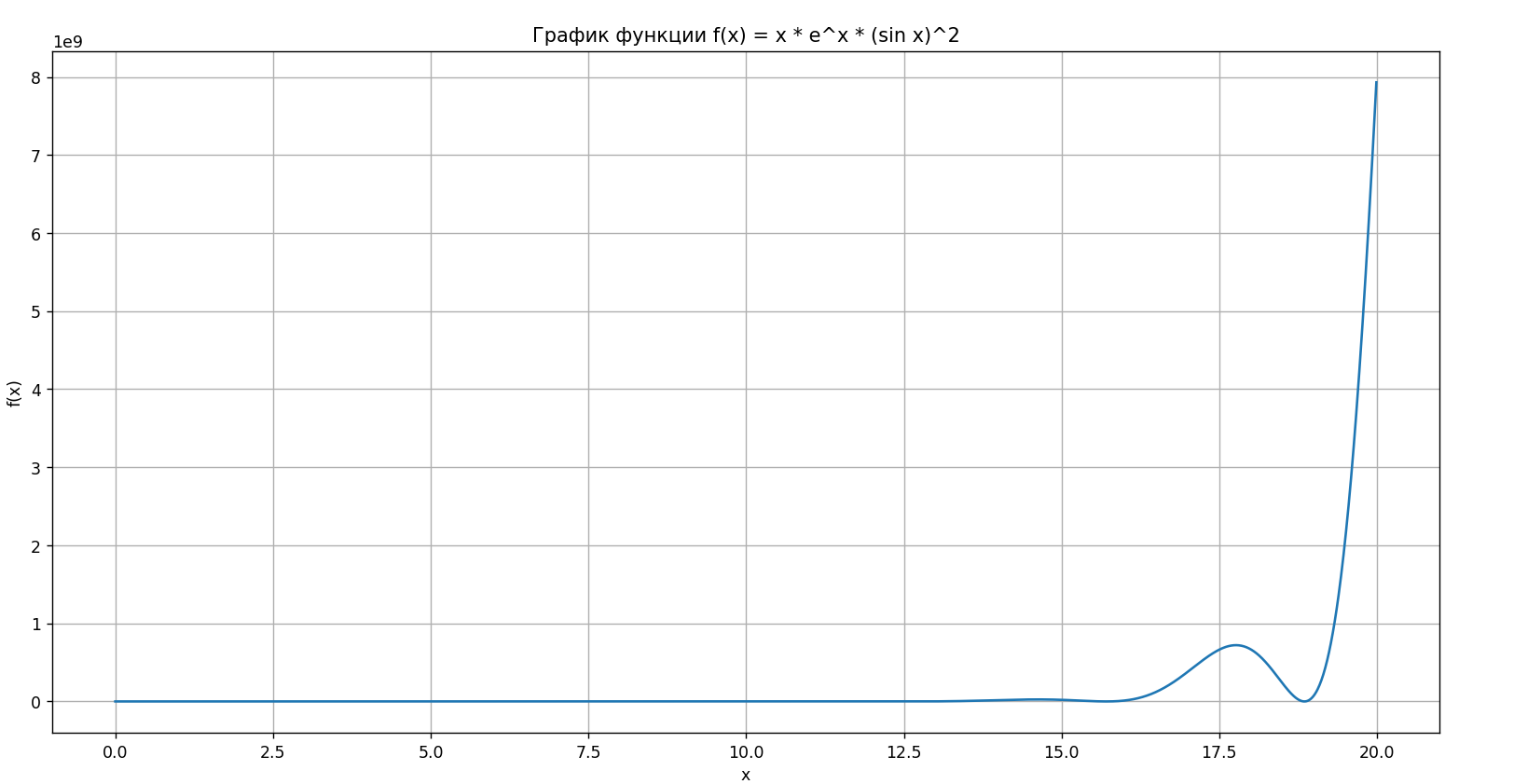
Оба метода ограничены максимальным количеством итераций (200,000), что предотвращает бесконечные циклы.

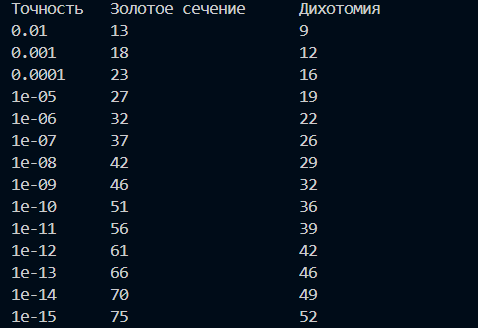
При очень малых значениях epsilon может возникать проблема численной неустойчивости.

**Графическое представление**

Построение графика функции помогает визуально подтвердить корректность найденных экстремумов.

## **Скриншоты**





## **Таблица зависимости количества вычислений от точности**

Зависимость количества вычислений функции от заданной точности ε

|  |  |  |
| --- | --- | --- |
| Точность (ε) | Количество итераций | |
| Метод золотого сечения | Метод дихотомии |
| 10^(-2) | 13 | 9 |
| 10^(-3) | 18 | 12 |
| 10^(-4) | 23 | 16 |
| 10^(-5) | 27 | 19 |
| 10^(-6) | 32 | 22 |
| 10^(-7) | 37 | 26 |

# **Ссылка на репозиторий**

https://github.com/ghimik/Computer-Simulating-Systems-Labs

# **Выводы по результатам решения задачи.**

В рамках данной лабораторной работы была разработана библиотека для анализа математических функций, включая:

* Реализацию классов для представления функций, их производных и экстремумов.
* Разработку численных методов (метод дихотомии и метод золотого сечения) для поиска экстремумов функций на заданном интервале.
* Создание инструментов для визуализации функций и анализа их поведения.

*Зависимость скорости работы от точности ε*

При анализе результатов работы программы можно сделать следующие выводы о зависимости скорости работы (количества вычислений функции n) от заданного значения точности ε.

Общие тенденции:

* Чем меньше значение ε, тем больше итераций требуется для достижения заданной точности.
* Зависимость между n и ε является обратной: при уменьшении ε количество итераций n возрастает.

*Сравнение методов.*

Хотя метод золотого сечения теоретически более эффективен (он минимизирует количество вычислений функции за счет оптимального распределения точек), на практике его производительность может снижаться из-за ошибок округления.

При очень малых значениях ε ошибки округления могут накапливаться, что влияет на точность вычислений.

Метод дихотомии менее подвержен этому эффекту, так как он использует простое деление интервала пополам.

Он менее чувствителен к особенностям функции, таким как резкие изменения или осцилляции.

*Количественная зависимость*

Для ε=10 ^ (−4)

Метод дихотомии: примерно n=16 итераций.

Метод золотого сечения: примерно n=23 итераций.