Федеральное государственное автономное образовательное учреждение высшего образования
«Санкт-Петербургский национальный исследовательский Университет ИТМО»

Мегафакультет Трансляционных информационных технологий
Факультет информационных технологий и программирования

Лабораторная работа №2.

По дисциплине «Прикладная математика»

Выполнили:
Студенты M32061
Величко Максим Иванович, 334786
Гусев Андрей Александрович, 336515
Бонет Станислав, 334349

Проверил:
Преподаватель практики
Свинцов Михаил Викторович

____

Санкт-Петербург, 2023

Условие:

Вариант 10

Риски

Предприимчивый бизнесмен затеял очередную авантюру, но решил подойти к этому вопросу системно и учесть все риски. Он исследовал риск от всех возможных предсказуемых факторов. Один из этих факторов, выраженный непрерывной величиной, дал наиболее интересную зависимость риска:

$$y(x) = \sin(0.5 \cdot \ln(x) \cdot x) + 1$$

Определите значение этого фактора, при котором авантюра бизнесмена будет наименее рискованной.

Пусть у нас будет кастомные параметры оптимизации:

```
A, B = 1, 20
```

Исходная функция:

```
def f(x):
  if x <= 0:
      return float('inf')
    return np.sin(0.5 * np.log(x) * x) + 1
```

Напишем функции для определения отношений:

```
threshold = 1


def is_ratio_bellow_threshold(ratio) -> bool:
  return ratio > threshold
```

```
def is_work_correct(arr_ratio) -> bool:
  return len(arr_ratio) < 1 or is_ratio_bellow_threshold(arr_ratio[-1])
```

Метод дихотомии

На каждом шаге алгоритма отрезок делится на две равные части, а затем выбирается та часть, на которой значение функции меньше. Алгоритм продолжается до тех пор, пока длина отрезка не станет меньше заданной точности eps или до тех пор, пока значение отношения длин двух последовательных отрезков не превысит порогового значения threshold.

Входные аргументы: Повторяются во всех методах

eps: требуемая точность решения

delta: параметр метода (расстояние между точками в новом отрезке), только в dichotomy

func: целевая функция

a: левая граница отрезка

b: правая граница отрезка

Выходные значения:

(a + b) / 2: значение минимума функции

count_iter: количество выполненных итераций метода

count_func: количество вызовов функции func

arr_a: массив левых границ отрезков на каждой итерации

arr_b: массив правых границ отрезков на каждой итерации

arr_ratio: массив отношений длин текущего и предыдущего отрезков на каждой итерации.

```
def dichotomy(eps, func, a, b):
    count_iter, count_func, prev_len = 0, 0, b - a
    arr_a, arr_b, arr_ratio = [a], [b], []

    while b - a > eps and settings.is_work_correct(arr_ratio):
        count_iter += 1

        x1, x2 = (a + b) / 2 - eps / 2, (a + b) / 2 + eps / 2
        f_x1, f_x2 = func(x1), func(x2)
        count_func += 2

        if f_x1 < f_x2:
            b = x2
        elif f_x1 > f_x2:
            a = x1
        else:
            a, b = x1, x2

        arr_a.append(a)
        arr_b.append(b)
```

```
        arr_ratio.append(prev_len / (b - a))
        prev_len = b - a

    return (a + b) / 2, count_iter, count_func, arr_a, arr_b, arr_ratio

val = dichotomy(0.001, f, A, B)[0]
print(f"Алгоритм нашел минимум функции sin(0.5*ln(x)*x)+1 в точке {val}")

val = dichotomy(0.005, np.sin, -2.5, 2)[0]
print(f"Алгоритм нашел минимум функции sin(x) в точке {val}")

val = dichotomy(0.005, np.cos, -2.5, 2)[0]
print(f"Алгоритм нашел минимум функции cos(x) в точке {val}")
```

Вывод программы:

Алгоритм нашел минимум функции sin(0.5*ln(x)*x)+1 в точке 5.517980998131538
Алгоритм нашел минимум функции sin(x) в точке -1.5707963267948921
Алгоритм нашел минимум функции cos(x) в точке -2.4974999999999996

Метод золотого сечения

На каждой итерации находятся две новые точки, а затем выбирается та точка, значение функции в которой меньше. Границы отрезка заменяются на точки, образованные разделением отрезка, содержащего эту точку, в золотом соотношении. Повторение происходит до тех пор, пока длина отрезка не станет меньше заданной точности.

```
def golden_ratio(eps, func, a, b):
    golden_const_1 = ((-5 ** 0.5 + 3) / 2)
    golden_const_2 = 1 / ((5 ** 0.5 + 1) / 2)
    count_iter, count_func, prev_len = 0, 0, b - a
    arr_a, arr_b, arr_ratio = [a], [b], []
    saved_part = 0
    prev_func = None
    while b - a > eps and is_work_correct(arr_ratio):
        x1, x2 = a + golden_const_1 * (b - a), a + golden_const_2 * (b - a)
        if saved_part == 0:
            f_x1, f_x2 = func(x1), func(x2)
            count_func += 2
        elif saved_part == -1:
            f_x1, f_x2 = prev_func, func(x2)
            count_func += 1
        else:
            f_x1, f_x2 = func(x1), prev_func
            count_func += 1
        if f_x1 < f_x2:
            b = x2
            saved_part = 1
            prev_func = f_x1
        elif f_x1 > f_x2:
            a = x1
            saved_part = -1
            prev_func = f_x2
```

```
        else:
            a, b = x1, x2
            saved_part = 0
        arr_ratio.append(prev_len / (b - a))
        prev_len = b - a
        count_iter += 1
        arr_a.append(a), arr_b.append(b)

    return (a + b) / 2, count_iter, count_func, arr_a, arr_b, arr_ratio

val = golden_ratio(0.001, f, A, B)[0]
print(f"Алгоритм нашел минимум функции в точке {val}")

val = golden_ratio(0.005, np.sin, -2.5, 2)[0]
print(f"Алгоритм нашел минимум функции sin(x) в точке {val}")

val = golden_ratio(0.005, np.cos, -2.5, 2)[0]
print(f"Алгоритм нашел минимум функции cos(x) в точке {val}")
```

Вывод программы:

Алгоритм нашел минимум функции в точке 13.339198633131074
Алгоритм нашел минимум функции sin(x) в точке -1.5717195080175879
Алгоритм нашел минимум функции cos(x) в точке -2.498350440769321

Метод Фибоначчи

Сначала вычисляется последовательность чисел Фибоначчи, которые больше или равны (b - a) / eps, где eps - требуемая точность. Затем определяются две начальные точки x1 и x2, которые делят отрезок [a, b] в соотношении чисел Фибоначчи. Затем на каждой итерации алгоритма выбирается один из двух подотрезков [a, x2] или [x1, b] на основе значений функции в точках x1 и x2. Затем точки x1 и x2 перемещаются в сторону выбранного подотрезка в соответствии с последовательностью чисел Фибоначчи, а значение функции в одной из этих точек вычисляется заново.

Процесс продолжается до тех пор, пока длина текущего отрезка не станет меньше eps. Все точки a и b, которые определяются на каждой итерации, сохраняются в массивах arr_a и arr_b. Кроме того, на каждой итерации сохраняется отношение длины предыдущего отрезка к длине текущего отрезка в массиве arr_ratio. В конце метода возвращается середина текущего отрезка, количество итераций, количество вычислений функции, массивы arr_a, arr_b и arr_ratio.

```
def gen_fib(min_fib: int) -> list[int]:
    fib_seq = [1, 1]
    while fib_seq[-1] <= min_fib:
        fib_seq.append(fib_seq[-1] + fib_seq[-2])
    return fib_seq

def fibonacci(eps, func, a, b):
    count_iter, count_func, prev_len = 1, 2, b - a
    arr_a, arr_b, arr_ratio = [a], [b], []
    fib_seq = gen_fib(math.ceil((b - a) / eps))
    n = len(fib_seq) - 1
```

```
    if n < 2:
        x1, x2 = a, b
    else:
        x1, x2 = a + fib_seq[n - 2] / fib_seq[n] * (b - a), a + fib_seq[n - 1] / fib_seq[n] * (b - a)
    f1, f2 = func(x1), func(x2)
    k = 1
    while k + 2 < n and settings.is_work_correct(arr_ratio):
        if f1 > f2:
            a = x1
            if n - k < 2:
                x1 = a
            else:
                x1 = a + fib_seq[n - k - 2] / fib_seq[n - k] * (b - a)
            x2 = a + fib_seq[n - k - 1] / fib_seq[n - k] * (b - a)
            f1, f2 = f2, func(x2)
        else:
            b = x2
            x1 = a + fib_seq[n - k - 2] / fib_seq[n - k] * (b - a)
            if n - k < 1:
                x2 = b
            else:
                x2 = a + fib_seq[n - k - 1] / fib_seq[n - k] * (b - a)
            f1, f2 = func(x1), f1
        arr_a.append(a)
        arr_b.append(b)
        count_iter += 1
        count_func += 1
        k += 1
        arr_ratio.append(prev_len / (b - a))
        prev_len = b - a
    return (a + b) / 2, count_iter, count_func, arr_a, arr_b, arr_ratio

val = fibonacci(0.001, f, A, B)[0]
print(f"Алгоритм нашел минимум функции в точке {val}")

val = fibonacci(0.005, np.sin, -2.5, 2)[0]
print(f"Алгоритм нашел минимум функции sin(x) в точке {val}")

val = fibonacci(0.005, np.cos, -2.5, 2)[0]
print(f"Алгоритм нашел минимум функции cos(x) в точке {val}")
```

Вывод программы:

Алгоритм нашел минимум функции в точке 13.339027113794188
Алгоритм нашел минимум функции sin(x) в точке -1.5721884498480243
Алгоритм нашел минимум функции cos(x) в точке -2.493161094224924

Метод парабол

Алгоритм использует приближение функции на каждой итерации параболой, которая проходит через три последних точки, и находит точку минимума параболы.

Начальным интервалом является заданный интервал [a, b]. На первой итерации выбираются три точки на этом интервале: крайние точки a и b, а также середина интервала x2 = (a+b)/2. В дальнейшем на каждой итерации находится точка минимума параболы, проходящей через эти три точки, и затем сравнивается значение функции в точке минимума с значениями функции в точках x1, x2 и x3. Точка, которая соответствует меньшему значению функции, заменяет другую крайнюю точку, и процесс повторяется на следующей итерации. Алгоритм останавливается, когда достигнута заданная точность eps или значение длины интервала между крайними точками становится меньше eps.

В результате работы метода парабол возвращается точка минимума, количество итераций, количество вызовов функции, а также последовательности точек, соответствующих левой и правой границе интервала на каждой итерации и отношение длин интервалов между последовательными итерациями.

```
def parabolas(eps, func, a, b):
    count_iter, count_func, prev_len = 0, 3, b - a
    x1, x2, x3 = a, (b + a) / 2, b
    arr_a, arr_b, arr_ratio = [x1], [x3], []
    f1, f2, f3 = func(x1), func(x2), func(x3)
    while True:
        numerator = ((x2 - x1) ** 2 * (f2 - f3) - (x2 - x3) ** 2 * (f2 - f1))
        denominator = 2 * ((x2 - x1) * (f2 - f3) - (x2 - x3) * (f2 - f1))
        if denominator == 0 or abs(x3 - x1) < eps or not is_work_correct(arr_ratio):
            break
        x_min = x2 - numerator / denominator
        f_min = func(x_min)

        if x_min < x2:
            if f_min > f2:
                x1 = x_min
                f1 = f_min
            else:
                x3, x2 = x2, x_min
                f3, f2 = f2, f_min
        else:
            if f_min > f2:
                x3 = x_min
                f3 = f_min
            else:
                x1, x2 = x2, x_min
                f1, f2 = f2, f_min

        count_iter += 1
        count_func += 1
        arr_a.append(x1)
        arr_b.append(x3)
        arr_ratio.append(prev_len / (x3 - x1))
        prev_len = x3 - x1
    if x2 == x1 or f2 == f1:
        res = arr_a[-1], count_iter, count_func, arr_a, arr_b, arr_ratio
    else:
        res = arr_b[-1], count_iter, count_func, arr_a, arr_b, arr_ratio
    return res

val = parabolas(0.001, f, A, B)[0]
```

```
print(f"Алгоритм нашел минимум функции в точке {val}")

val = parabolas(0.005, np.sin, -2.5, 2)[0]
print(f"Алгоритм нашел минимум функции sin(x) в точке {val}")

val = parabolas(0.005, np.cos, -2.5, 2)[0]
print(f"Алгоритм нашел минимум функции cos(x) в точке {val}")
```

Вывод программы:

Алгоритм нашел минимум функции в точке 56.39254603317852
Алгоритм нашел минимум функции sin(x) в точке -1.5707963142797627
Алгоритм нашел минимум функции cos(x) в точке 0.17336123893531497

Метод Брента

Метод комбинирует в себе три подхода: метод золотого сечения, метод парабол и метод секущих. Он выбирает в каждой итерации один из трех подходов на основе предыдущих итераций и аппроксимации функции параболой.

Метод начинает с использования метода золотого сечения для приближенного нахождения минимума функции на заданном интервале. Затем метод переключается на метод парабол, который позволяет находить точки минимума с высокой точностью. В случае, если метод парабол стагнирует, метод переключается на метод секущих. Метод секущих используется для обхода точек, в которых производная функции равна нулю.

В каждой итерации метода вычисляется значение функции в текущей точке и определяется новая точка для следующей итерации. Метод сохраняет значения границ интервала на каждой итерации, что позволяет отслеживать сходимость метода и оценить скорость сходимости.

```
def brent(eps, f, a, b):
    arr_a, arr_b, arr_ratio = [a], [b], []
    count_iter, count_func = 0, 0

    eps1 = eps / 10
    phi = (3 - np.sqrt(5)) / 2
    x = w = v = a + phi * (b - a)
    fx = fw = fv = f(x)
    d, e = b - a, b - a

    while d > eps:
        g = e
        e = d

        u = None
        if x != w and x != v and w != v and fx != fw and fx != fv and fw != fv:
            p = (x - w) * (fx - fv) - (x - v) * (fx - fw)
            q = 2 * (x - w) * (fx - fv) - 2 * (x - v) * (fx - fw)

            if q != 0:
                u = x - (p / q)
                if a + eps1 <= u <= b - eps1 and np.abs(u - x) < g / 2:
```

```
                d = np.abs(u - x)
            else:
                u = None

        if u is None:
            if x < (b + a) / 2:
                u = x + phi * (b - x)
                d = b - x
            else:
                u = x - phi * (x - a)
                d = x - a

            if np.abs(u - x) < eps1:
                u = x + np.sign(u - x) * eps1

        fu = f(u)
        count_func += 1

        if fu <= fx:
            if u >= x:
                a = x
            else:
                b = x

            v, w, x = w, x, u
            fv, fw, fx = fw, fx, fu
        else:
            if u >= x:
                b = u
            else:
                a = u

            if fu <= fw or w == x:
                v, w = w, u
                fv, fw = fw, fu
            elif fu <= fv or v == x:
                v, fv = u, fu

        arr_a.append(a), arr_b.append(b)
        arr_ratio.append(e / d)
        count_iter += 1

    return x, count_iter, count_func, arr_a, arr_b, arr_ratio

val = brent(0.001, f, A, B)[0]
print(f"Алгоритм нашел минимум функции в точке {val}")

val = brent(0.005, np.sin, -2.5, 2)[0]
print(f"Алгоритм нашел минимум функции sin(x) в точке {val}")

val = brent(0.005, np.cos, -2.5, 2)[0]
print(f"Алгоритм нашел минимум функции cos(x) в точке {val}")
```

Вывод программы:

Алгоритм нашел минимум функции в точке 13.33902697002776
Алгоритм нашел минимум функции в точке 13.33902697002776
Алгоритм нашел минимум функции cos(x) в точке -2.497854069347608

Сравнение методов

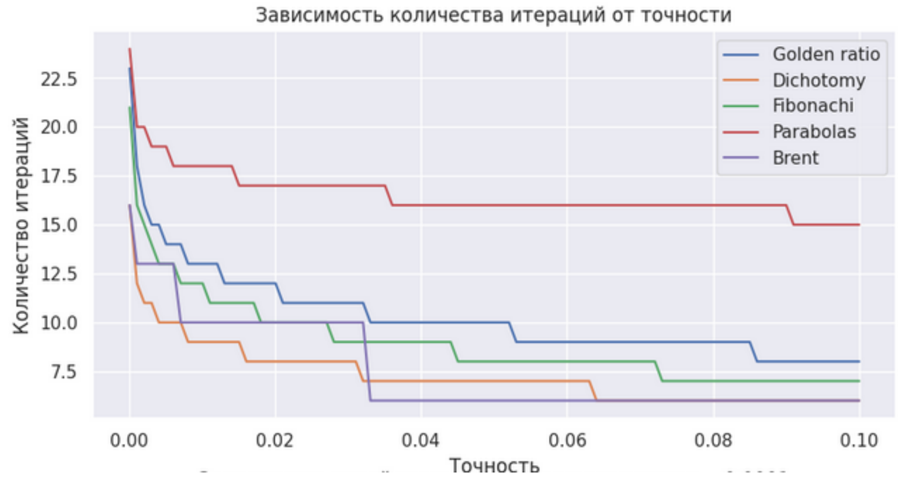
```
def plot_graph(ax, x, y, xlabel, ylabel, title, label):
    sns.lineplot(x=x, y=y, ax=ax, label=label)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.set_title(title)


def plot_scatter(ax, x, y, xlabel, ylabel, title, label):
    sns.scatterplot(x=x, y=y, ax=ax, label=label)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.set_title(title)


def analyze(f, a, b):
    stats = {"golden_ratio": {
        "count_iter": [],
        "count_func": [],
        "arr_a": [],
        "arr_b": [],
        "arr_ratio": [],
        "eps": []
    }, "dichotomy": {
        "count_iter": [],
        "count_func": [],
        "arr_a": [],
        "arr_b": [],
        "arr_ratio": [],
        "eps": []
    }, "fibonacci": {
        "count_iter": [],
        "count_func": [],
        "arr_a": [],
        "arr_b": [],
        "arr_ratio": [],
        "eps": []
    }, "parabolas": {
        "count_iter": [],
        "count_func": [],
        "arr_a": [],
        "arr_b": [],
        "arr_ratio": [],
        "eps": []
    }, "brent": {
        "count_iter": [],
        "count_func": [],
```

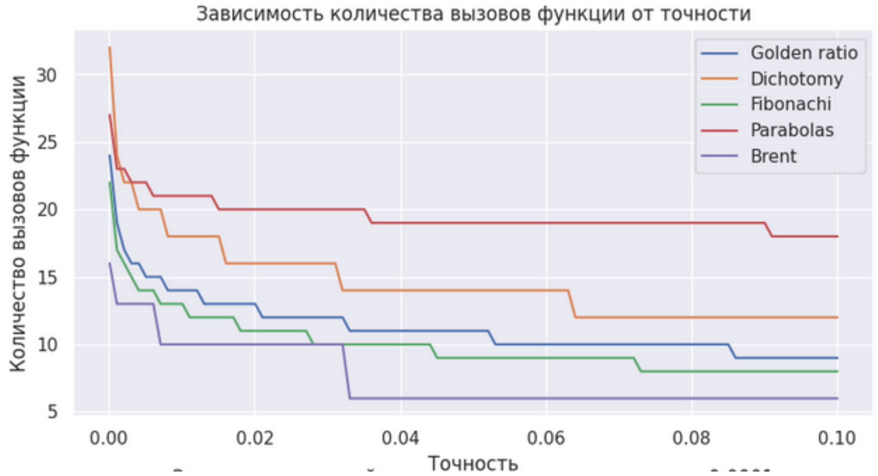
```
    "arr_a": [],
    "arr_b": [],
    "arr_ratio": [],
    "eps": []
}}

for eps in tqdm(np.linspace(0.0001, 0.1, 101)):
    # Dichotomy
    output = dichotomy(eps, f, a, b)
    stats["dichotomy"]["count_iter"].append(output[1])
    stats["dichotomy"]["count_func"].append(output[2])
    stats["dichotomy"]["arr_a"].append(output[3])
    stats["dichotomy"]["arr_b"].append(output[4])
    stats["dichotomy"]["arr_ratio"].append(output[5])
    stats["dichotomy"]["eps"].append(eps)

    # Golden ratio
    output = golden_ratio(eps, f, a, b)
    stats["golden_ratio"]["count_iter"].append(output[1])
    stats["golden_ratio"]["count_func"].append(output[2])
    stats["golden_ratio"]["arr_a"].append(output[3])
    stats["golden_ratio"]["arr_b"].append(output[4])
    stats["golden_ratio"]["arr_ratio"].append(output[5])
    stats["golden_ratio"]["eps"].append(eps)

    # Fibonacci
    output = fibonacci(eps, f, a, b)
    stats["fibonacci"]["count_iter"].append(output[1])
    stats["fibonacci"]["count_func"].append(output[2])
    stats["fibonacci"]["arr_a"].append(output[3])
    stats["fibonacci"]["arr_b"].append(output[4])
    stats["fibonacci"]["arr_ratio"].append(output[5])
    stats["fibonacci"]["eps"].append(eps)

    # Parabolas
    output = parabolas(eps, f, a, b)
    stats["parabolas"]["count_iter"].append(output[1])
    stats["parabolas"]["count_func"].append(output[2])
    stats["parabolas"]["arr_a"].append(output[3])
    stats["parabolas"]["arr_b"].append(output[4])
    stats["parabolas"]["arr_ratio"].append(output[5])
    stats["parabolas"]["eps"].append(eps)

    # Brent's method
    output = brent(eps, f, a, b)
    stats["brent"]["count_iter"].append(output[1])
    stats["brent"]["count_func"].append(output[2])
    stats["brent"]["arr_a"].append(output[3])
    stats["brent"]["arr_b"].append(output[4])
    stats["brent"]["arr_ratio"].append(output[5])
    stats["brent"]["eps"].append(eps)

fig, ax = plt.subplots(3, 2, figsize=(20, 15))

sns.lineplot(x=stats["golden_ratio"]["eps"], y=stats["golden_ratio"]["count_iter"], ax=ax[0][0],
```

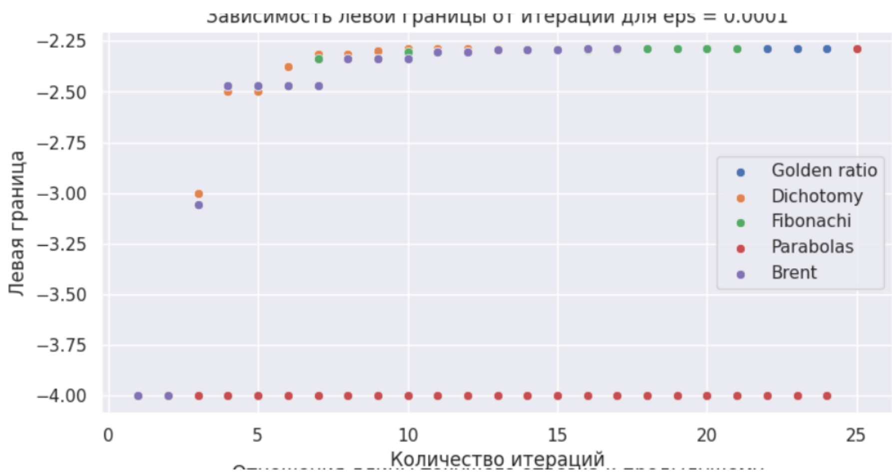
```
              label="Golden ratio")
    sns.lineplot(x=stats["dichotomy"]["eps"], y=stats["dichotomy"]["count_iter"], ax=ax[0][0], label="Dichotomy")
    sns.lineplot(x=stats["fibonacci"]["eps"], y=stats["fibonacci"]["count_iter"], ax=ax[0][0], label="Fibonacci")
    sns.lineplot(x=stats["parabolas"]["eps"], y=stats["parabolas"]["count_iter"], ax=ax[0][0], label="Parabolas")
    sns.lineplot(x=stats["brent"]["eps"], y=stats["brent"]["count_iter"], ax=ax[0][0], label="Brent")
    ax[0][0].set_xlabel("Точность")
    ax[0][0].set_ylabel("Количество итераций")
    ax[0][0].set_title("Зависимость количества итераций от точности")

    sns.lineplot(x=stats["golden_ratio"]["eps"], y=stats["golden_ratio"]["count_func"], ax=ax[0][1],
              label="Golden ratio")
    sns.lineplot(x=stats["dichotomy"]["eps"], y=stats["dichotomy"]["count_func"], ax=ax[0][1], label="Dichotomy")
    sns.lineplot(x=stats["fibonacci"]["eps"], y=stats["fibonacci"]["count_func"], ax=ax[0][1], label="Fibonacci")
    sns.lineplot(x=stats["parabolas"]["eps"], y=stats["parabolas"]["count_func"], ax=ax[0][1], label="Parabolas")
    sns.lineplot(x=stats["brent"]["eps"], y=stats["brent"]["count_func"], ax=ax[0][1], label="Brent")
    ax[0][1].set_xlabel("Точность")
    ax[0][1].set_ylabel("Количество вызовов функции")
    ax[0][1].set_title("Зависимость количества вызовов функции от точности")

    sns.scatterplot(x=np.arange(1, stats["golden_ratio"]["count_iter"][0] + 2), y=stats["golden_ratio"]["arr_a"][0],
                ax=ax[1][0], label="Golden ratio")
    sns.scatterplot(x=np.arange(1, stats["dichotomy"]["count_iter"][0] + 1), y=stats["dichotomy"]["arr_a"][0],
                ax=ax[1][0], label="Dichotomy")
    sns.scatterplot(x=np.arange(1, stats["fibonacci"]["count_iter"][0] + 1), y=stats["fibonacci"]["arr_a"][0],
                ax=ax[1][0], label="Fibonachi")
    sns.scatterplot(x=np.arange(1, stats["parabolas"]["count_iter"][0] + 2), y=stats["parabolas"]["arr_a"][0],
                ax=ax[1][0], label="Parabolas")
    sns.scatterplot(x=np.arange(1, stats["brent"]["count_iter"][0] + 2), y=stats["brent"]["arr_a"][0], ax=ax[1][0],
                label="Brent")
    ax[1][0].set_xlabel("Количество итераций")
    ax[1][0].set_ylabel("Левая граница")
    ax[1][0].set_title("Зависимость левой границы от итерации для eps = 0.0001")

    sns.scatterplot(x=np.arange(1, stats["golden_ratio"]["count_iter"][0] + 2), y=stats["golden_ratio"]["arr_b"][0],
                ax=ax[1][1], label="Golden ratio")
    sns.scatterplot(x=np.arange(1, stats["dichotomy"]["count_iter"][0] + 1), y=stats["dichotomy"]["arr_b"][0],
                ax=ax[1][1], label="Dichotomy")
    sns.scatterplot(x=np.arange(1, stats["fibonacci"]["count_iter"][0] + 1), y=stats["fibonacci"]["arr_b"][0],
                ax=ax[1][1], label="Fibonacci")
    sns.scatterplot(x=np.arange(1, stats["parabolas"]["count_iter"][0] + 2), y=stats["parabolas"]["arr_b"][0],
                ax=ax[1][1], label="Parabolas")
    sns.scatterplot(x=np.arange(1, stats["brent"]["count_iter"][0] + 2), y=stats["brent"]["arr_b"][0], ax=ax[1][1],
                label="Brent")
    ax[1][1].set_xlabel("Количество итераций")
    ax[1][1].set_ylabel("Правая граница")
    ax[1][1].set_title("Зависимость правой границы от итерации для eps = 0.0001")
```

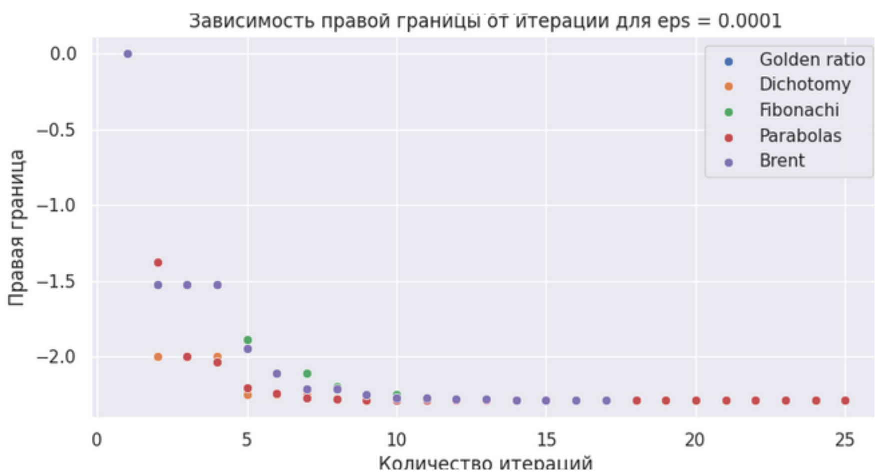
На большом отрезке:

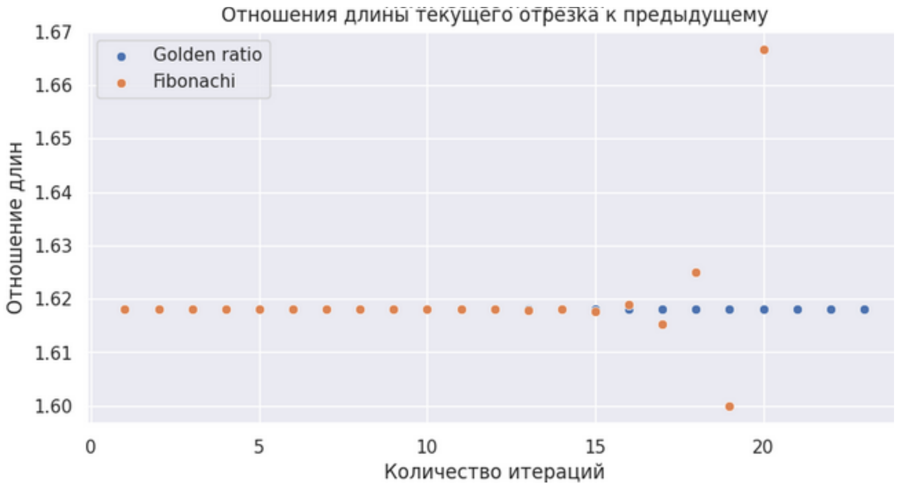
```
analyze(f, A, B)
```

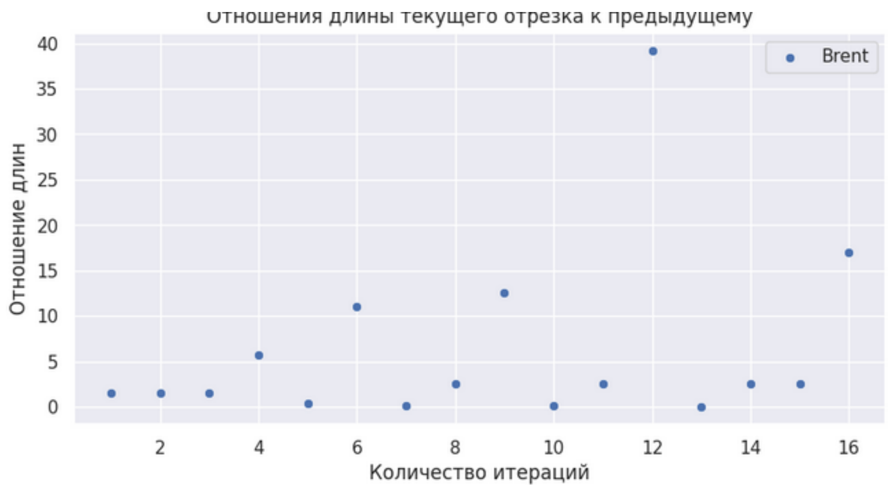

Зависимость количества итераций от точности
Количество итераций
Точность
Golden ratio
Dichotomy
Fibonachi
Parabolas
Brent


Зависимость количества вызовов функции от точности
Количество вызовов функции
Точность
Golden ratio
Dichotomy
Fibonachi
Parabolas
Brent


Зависимость левой границы от итерации для eps = 0.0001
Левая граница
Количество итераций
Golden ratio
Dichotomy
Fibonachi
Parabolas
Brent

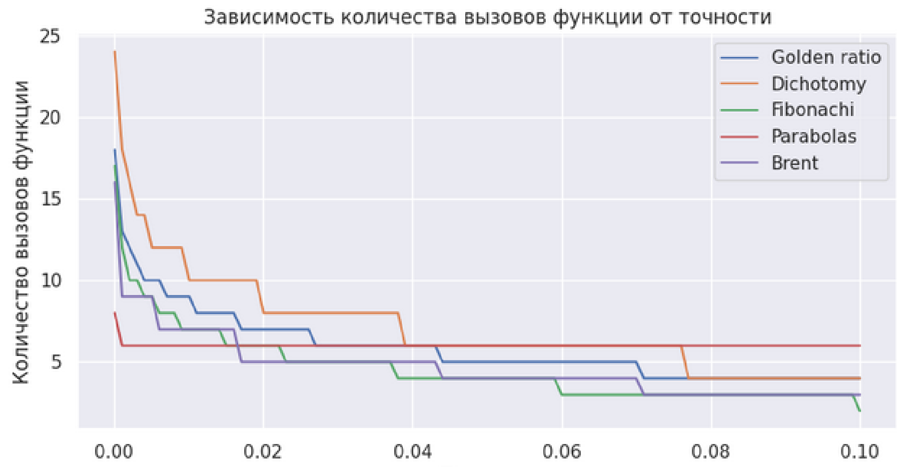

Зависимость правой границы от итерации для eps = 0.0001
Правая граница
Количество итераций
Golden ratio
Dichotomy
Fibonachi
Parabolas
Brent





На маленьком отрезке:

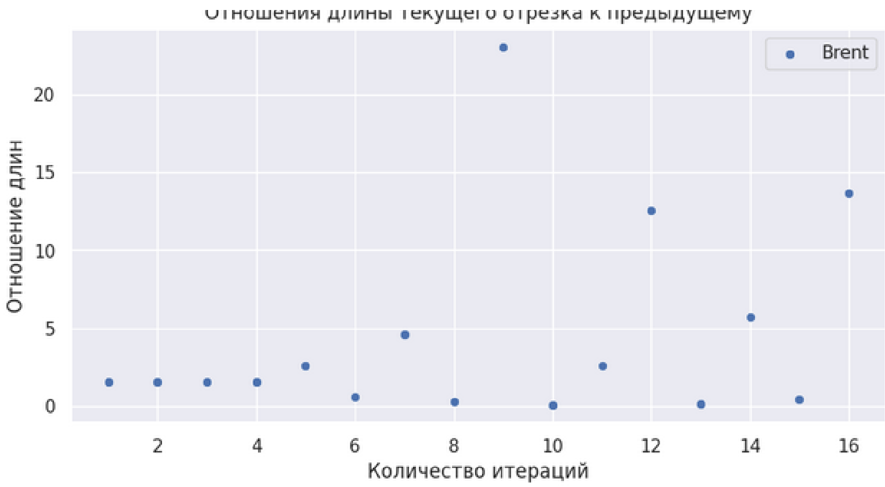
```
analyze(-2.3, -2)
```


Зависимость количества вызовов функции от точности
Количество вызовов функции
Golden ratio
Dichotomy
Fibonachi
Parabolas
Brent


Зависимость левой границы от итерации для eps = 0.0001
Левая граница
Golden ratio
Dichotomy
Fibonachi
Parabolas
Brent


Зависимость правой границы от итерации для eps = 0.0001
Правая граница
Golden ratio
Dichotomy
Fibonachi
Parabolas
Brent


Отношения длины текущего отрезка к предыдущему
Отношение длин
Количество итераций
Golden ratio
Dichotomy
Fibonachi
Parabolas



Вывод по лабораторной работе:

В процессе выполнения лабораторной работы были изучены и реализованы методы одномерной оптимизации.

- Метод дихотомии хорошо работает на гладких функциях с единственным минимумом, но может оказаться неэффективным на функциях с несколькими минимумами или на функциях с быстро изменяющимся градиентом.
- Метод золотого работает хорошо на гладких функциях, но может также потерпеть неудачу на функциях с несколькими минимумами или на функциях, которые изменяют свой градиент быстро.
- Метод Фибоначчи основан на последовательности чисел Фибоначчи и может сходиться быстрее, чем методы дихотомии или золотого сечения.
- Метод парабол имеет преимущество в том, что он может находить минимумы функций с несколькими локальными минимумами.
- Метод Брента обычно сходится быстрее, чем остальные методы и может находить минимумы функций с несколькими локальными минимумами. Он также устойчив к особенностям функций, таким как разрывы второй производной. Однако этот метод более сложен для реализации.

В целом, выбор метода одномерной оптимизации зависит от конкретной задачи и свойств функции.