Алешко Альберт

АС-21-05

Вариант 1

In [58]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import scipy
from math import *
import pandas as pd
from functools import partial
plt.style.use('Solarize_Light2')
```

1. Методы одномерной минимизации

1. Выбрать вариант задачи, соответствующий вашему номеру из списка группы. Вариант 1 делают студенты с номерами 1, 10, 18, 27, вариант 2 – 2, 11, 20, 29, …

1) $f(x)=x^3-3\sin x\to\min,\quad x\in[0,\,1].$

In [59]:
```
def func(x):
    return x**3 - 3 * sin(x)

def deriv1(x):
    return 3*x**2 - 3*cos(x)

def deriv2(x):
    return 6*x + 3 * sin(x)
```

2. Сделать обзор функций матпакета для одномерной минимизации. Найти решение задачи встроенными средствами.

Функция minimize библиотеки scipy

Функция minimize принимает на вход вызываемую функцию, метод, начальную точку, а так же ряд других аргументов, при необходимости...

На выходе мы получаем сообщение о том как завершилась работа функции, № ошибки (при успехе 0), значение функции, значение переменной, кол-во итераций, и число просчитываний функции.

In [60]:
```
f = np.vectorize(func)
minimum = scipy.optimize.minimize(f,method='Powell',bour
minimum
```

Out[60]:
```
 message: Optimization terminated successfully.
 success: True
  status: 0
     fun: -1.6421304129142102
       x: [ 8.241e-01]
     nit: 2
   direc: [[ 3.015e-14]]
    nfev: 54
```

Функция brent

Делает то же что и minimize, только вычисляет каким-то одним методом, поэтому параметров меньше. На вход, помимо функции и интервала, указание на полноту вывода и максимальное кол-во итераций.

В полном выводе возращает переменную, значение функции, кол-во итераций и кол-во вычислений функции.

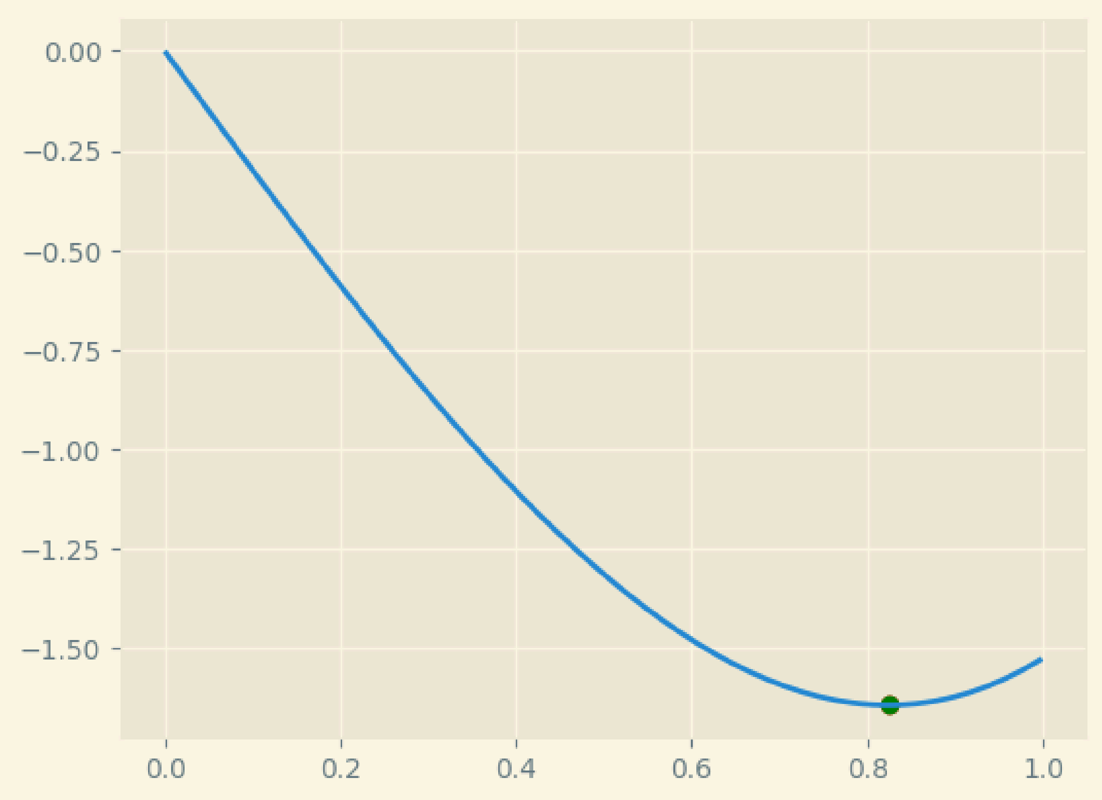
In [61]:
```
minimal_brent = scipy.optimize.brent(func, brack = (0,1)
minimal_brent
```

Out[61]:
```
(0.8241323111380852, -1.6421304129142098, 10, 13)
```

Визуализация функции на заданном интервале с ответом, полученным встроенными методами.

In [62]:
```
x = np.arange(0, 1, 0.001)
plt.plot(x, f(x))
```

```python
plt.scatter(minimum['x'], minimum['fun'], color = 'red')
plt.scatter(minimal_brent[0], minimal_brent[1], color =
plt.grid(True)
```



Функции отвечающие за построение траектории решения.

In [63]:
```python
def plot_convergence_1d(func, x_steps, y_steps, ax, grid
    """
    Функция отрисовки шагов градиентного спуска.
    Не меняйте её код без необходимости!
    :param func: функция, которая минимизируется градиен
    :param x_steps: np.array(float) — шаги алгоритма по
    :param y_steps: np.array(float) — шаги алгоритма по
    :param ax: холст для отрисовки графика
    :param grid: np.array(float) — точки отрисовки функц
    :param title: str — заголовок графика
    """
    ax.set_title(title, fontsize=16, fontweight="bold")

    if grid is None:
        grid = np.linspace(np.min(x_steps), np.max(x_ste

    fgrid = [func(item) for item in grid]
    ax.plot(grid, fgrid)
    yrange = np.max(fgrid) - np.min(fgrid)
```

```python
    arrow_kwargs = dict(linestyle="--", color="grey", al
    for i, _ in enumerate(x_steps):
        if i + 1 < len(x_steps):
            ax.arrow(
                x_steps[i], y_steps[i],
                x_steps[i + 1] - x_steps[i],
                y_steps[i + 1] - y_steps[i],
                **arrow_kwargs
            )

    n = len(x_steps)
    color_list = [(i / n, 0, 0, 1 - i / n) for i in rang
    ax.scatter(x_steps, y_steps, c=color_list)
    ax.scatter(x_steps[-1], y_steps[-1], c="red")
    ax.set_xlabel(r"$x$")
    ax.set_ylabel(r"$y$")
```

In [64]:

```python
class LoggingCallback:
    """
    Класс для логирования шагов градиентного спуска.
    Сохраняет точку (x, f(x)) на каждом шаге.
    Пример использования в коде: callback(x, f(x))
    """
    def __init__(self):
        self.x_steps = []
        self.y_steps = []

    def __call__(self, x, y):
        self.x_steps.append(x)
        self.y_steps.append(y)
```

In [65]:

```python
def plotting(axes, i, answer, title):
    if axes is not None:
        ax = axes[np.unravel_index(i, shape=axes.shape)]
        x_steps = np.array(callback.x_steps)
        y_steps = np.array(callback.y_steps)
        plot_convergence_1d(
            f, x_steps, y_steps,
            ax, grid, title
        )
        ax.axvline(answer, 0, linestyle="--", c="red",
                   label=f"true answer = {answer}")
        ax.axvline(x_steps[-1], 0, linestyle="--", c="xk
```

```
                         label=f"estimate = {np.round(x_steps
        ax.legend(fontsize=16)
```

3. Написать скрипты, реализующие следующие методы (по вариантам):

1 вариант:

3.1.1. Метод поразрядного поиска (два прохода, p – количество интервалов разбиения начального отрезка)

3.1.2. Метод Фибоначчи

3.1.3. Метод средней точки

3.1.4. Метод Ньютона-Рафсона

3.1.5. Метод ломанных

Метод поразрядного поиска

Сначала шаг равен (b-a)/p, потом eps, в начале вычисляется значения функции в 2 точках, потом на каждой итерации только один раз в последующих точках.

In [66]:

```
def bitwise_method(fun,a,b,p = 100,eps = 10**-6, callbac

    if callback is None:
        callback = lambda c, v: 0
    count = 2

    h = (b-a)/p
    x = a
    f0 = fun(x)
    f1 = fun(x+h)

    callback(x,f0)

    x+=h
    while(f1 < f0 and x < b):
#         print(f1,f0, x+h, func(x+h))
        f0 = f1
        callback(x,f0)
        x +=h
        f1 = fun(x)
        count += 1

    h = eps
    f0 = f1
    callback(x,f0)
    x -=h
    f1 = fun(x)
#     print(f0,f1,x)
```

```python
    while(f1 < f0 and x >= a):
        f0 = f1
        callback(x,f0)
        x -= h
        f1 = fun(x)
        #print(f0,f1, x)
        count +=1
    else:
        return x + h/2, count
```

In [67]:

```python
def fibonacci_numbers(max_value):

    num1, num2 = 1, 1

    if num1 > max_value:
        return

    yield num1

    if num2 > max_value:
        return

    yield num2

    while num2 < max_value:
        num1, num2 = num2, num1 + num2
        yield num2
```

In [68]:

```python
def fibonacci_method(func, a, b, interval_length, eps, c
    less = True
    if callback is None:
        callback = lambda c, v: 0

    max_value = (b - a) / interval_length
    fib = [num for num in fibonacci_numbers(max_value)]

    length = len(fib) - 1
    y = a + fib[length - 2] / fib[length] * (b - a)
    z = a + fib[length - 1] / fib[length] * (b - a)
    k = 1
    max_k = length - 3

    func_y, func_z = func(y), func(z)
```

```
    for k in range(max_k + 1):
        if func_y <= func_z:

            less = True
            b, z = z, y
            y = a + fib[length - k - 3] / fib[length - k
            callback(b, func_z)
        else:
            less = False
            a, y = y, z
            z = a + fib[length - k - 2] / fib[length - k
            callback(a, func_y)
        func_y, func_z = (func(y), func_y) if less else

    y = z
    z = y + eps
    x = (a + z) / 2 if func(y) <= func(z) else (y + b) /
    callback(x, func(x))
    return x, max_k + 1
```

3.1. Метод средней точки

Если определение производной $f'(x)$ не представляет затруднений, то в процедуре исключения отрезков методом дихотомии вычисление двух значений $f(x)$ вблизи середи-

ны очередного отрезка можно заменить вычислением одног

значения $f'(x)$ в его средней точке $\bar{x} = \dfrac{a+b}{2}$.

В самом деле, если $f'(\bar{x}) > 0$, то точка $\bar{x}$ лежит на участк монотонного возрастания $f(x)$, поэтому $x^* < \bar{x}$ и точку ми нимума следует искать на отрезке $[a, \bar{x}]$. При $f'(\bar{x}) < 0$ имее противоположную ситуацию и переходим к отрезку $[\bar{x}, b]$ Равенство $f'(\bar{x}) = 0$ означает, что точка минимума найден точно: $x^* = \bar{x}$ (рис. 3.1).

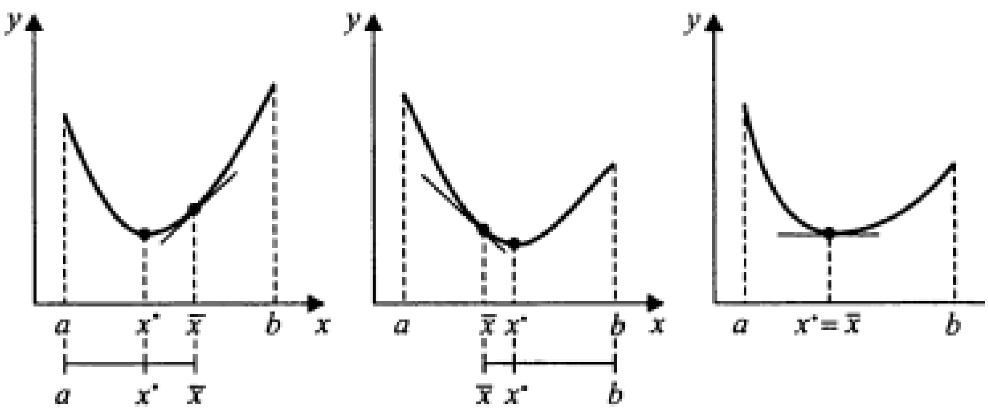


Рис. 3.1. **Иллюстрация исключения отрезков методом средней точки**

In [69]:
```
def midpoint_method(func, deriv, a, b, eps, callback=Nor

    if callback is None:
        callback = lambda c, v: 0

    k = 0

    interval = b - a

    while interval > eps:
        x = (a + b) / 2
        dfunc_x = deriv(x)
        callback(x, func(x))
        if dfunc_x > 0:
```

```
            b = x
        else:
            a = x
        interval /= 2
        k += 1
    return x, k
```

3.3. Метод Ньютона

Если выпуклая на отрезке $[a,b]$ функция $f(x)$ дважды непрерывно дифференцируема на этом отрезке, то точку $x^* \in [a,b]$ минимума этой функции можно найти, решая уравнение $f'(x)=0$ *методом Ньютона* (другое название — метод касательных). Пусть $x_0 \in [a,b]$ — нулевое (начальное) приближение к искомой точке x^*. Линеаризуем функцию $F(x)=f'(x)$ в окрестности начальной точки, приближенно заменив дугу графика этой функции касательной в точке $(x_0, f'(x_0))$

$$F(x) \approx F(x_0)+F'(x_0)(x-x_0). \tag{3.3}$$

Выберем в качестве следующего приближения к x^* точку x_1 пересечения касательной с осью абсцисс. Приравнивая к нулю правую часть в (3.3), получим первый элемент $x_1 = x_0 - \dfrac{F(x_0)}{F'(x_0)}$ итерационной последовательности $\{x_k\}$, $k=1,2,\ldots$

В очередной точке x_k построим линейную аппроксимирующую функцию для $F(x)$ и определим точку, в которой эта функция обращается в нуль, используя в качестве следующего приближения x_{k+1} (рис. 3.3).

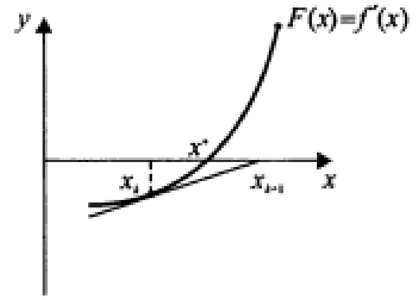


Рис. 3.3. **Иллюстрация метода касательных**

Уравнение касательной к графику $F(x)$ в точке $x=x_k$ имеет вид $y=F(x_k)+F'(x_k)(x-x_k)$, поэтому точка $x=x_{k+1}$, найденная из условия $y=0$, определяется формулой $x_{k+1}=x_k-\dfrac{F(x_k)}{F'(x_k)}$. Поскольку $F(x)\equiv f'(x)$, получим, что для решения уравнения $f'(x)=0$ необходимо построить последовательность

$$x_{k+1}=x_k-\frac{f'(x_k)}{f''(x_k)}, \quad k=1,2,\ldots \tag{3.4}$$

где x_0 — точка, выбранная в качестве начального приближения. Вычисления по формуле (3.4) производятся до тех пор, пока не выполнится неравенство $|f'(x_k)| \le \varepsilon$, после чего полагают $x^* \approx x_k$, $f^* \approx f(x_k)$.

```
In [70]: def newton_method(func, dfunc, d2func, x, eps, max_iter=

             if callback is None:
                 callback = lambda c, v: 0

             for k in range(max_iter):
                 callback(x, func(x))
                 dfunc_x = dfunc(x)
                 d2func_x_inv = 1 / d2func(x)
                 d = -d2func_x_inv * dfunc_x
                 x_prev = x
                 x = x + d

                 if np.all(abs(x - x_prev) < eps):
                     callback(x, func(x))
                     return x, k + 1

             if print_info:
                 print('Max iterations. Stop')
             callback(x, func(x))
             return x, max_iter
```

Метод Ньютона — Рафсона

При переходе к очередной итерации новая точка x_{k+1} рассчитывается по формуле

$$x_{k+1} = x_k - \tau_k \frac{f'(x_k)}{f''(x_k)}, \quad 0 < \tau_k \le 1.$$

В простейшем варианте метода $\tau_k = \tau = \text{const}$ ($\tau = 1$ соответствует исходному методу Ньютона). Оптимальный набор параметров τ_k может быть найден из решения задачи минимизации

$$\varphi(\tau) = f\left(x_k - \tau \frac{f'(x_k)}{f''(x_k)}\right) \to \min.$$

На практике для параметров τ_k обычно используется приближенное решение последней задачи

$$\tau_k = \frac{(f'(x_k))^2}{(f'(x_k))^2 + (f'(\tilde{x}))^2}, \text{ где } \tilde{x} = x_k - \frac{f'(x_k)}{f''(x_k)}.$$

```
In [71]: def newton_raphson_method(func, dfunc, d2func, x, eps, l
                                   n_eps=0.01, max_iter=100_000,
```

```
                           callback=None, print_info=Fals

    if callback is None:
        callback = lambda c, v: 0

    def fi(t, x, d):
        nonlocal func
        return func(x + t * d)

    def dfi(t, x, d):
        nonlocal dfunc
        return dfunc(x + t * d)

    def d2fi(t, x, d):
        nonlocal d2func
        return d2func(x + t * d)
    k = 0

    for k in range(max_iter):
        callback(x, func(x))
        dfunc_x = dfunc(x)

        d2func_x_inv = 1 / d2func(x)
        d = -d2func_x_inv * dfunc_x
        fi_x_d = partial(fi, x=x, d=d)
        dfi_x_d = partial(dfi, x=x, d=d)
        d2fi_x_d = partial(d2fi, x=x, d=d)

        t, _ = newton_method(fi_x_d, dfi_x_d, d2fi_x_d,
        x_prev = x
        func_x_prev = func(x)
        x = x + d
        func_x = func(x)
        if np.abs(x - x_prev) < eps:
            callback(x, func(x))
            return x, k + 1

    if print_info:
        print('Max iterations. Stop')
    callback(x, func(x))
    return x, max_iter
```

Метод ломаных

Этот метод также рассчитан на минимизацию многомодальных функций, удовлетворяющих условию Липшица. В нем используются кусочно-линейные аппроксимации функции $f(x)$, графиками которых являются ломаные, что и объясняет название метода.

Пример 3.5. Пусть функция $f(x)$ удовлетворяет на отрезке $[a,b]$ условию Липшица с константой L. Зафиксируем точку $\bar{x} \in [a,b]$ и введем вспомогательную функцию одной переменной $g(\bar{x}, x) = f(\bar{x}) - L|\bar{x} - x|$, график которой показан на рис. 3.6. Эта функция максимальна в точке $\bar{x}$ и минимальна на концах отрезка $[a,b]$.

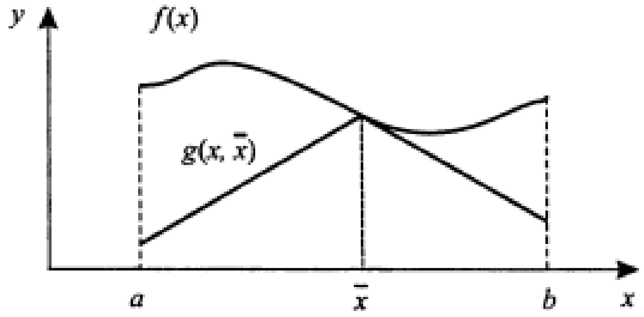


Рис. 3.6. **График функции** $g(\bar{x}, x)$

Аппроксимирующие кусочно-линейные функции $p_k(x)$, $k = 0, 1, \ldots$ строятся следующим образом. Рассмотрим прямые $y = f(a) - L(x-a)$ и $y = f(b) + L(x-b)$. Они пересекаются в точке $(x_0,\ y_0)$ с координатами

$$x_0 = \frac{1}{2L}[f(a) - f(b) + L(a+b)],$$
$$y_0 = \frac{1}{2}[f(a) + f(b) + L(a-b)]. \tag{3.10}$$

Положим

$$p_0(x) = \begin{cases} f(a) - L(x-a), & x \in [a,\ x_0], \\ f(b) + L(x-b), & x \in [x_0,\ b]. \end{cases}$$

График функции $p_0(x)$ показан на рис. 3.7а, ее точка минимума $x_0^* = x_0$, а минимальное значение $p_0^* = y_0$.

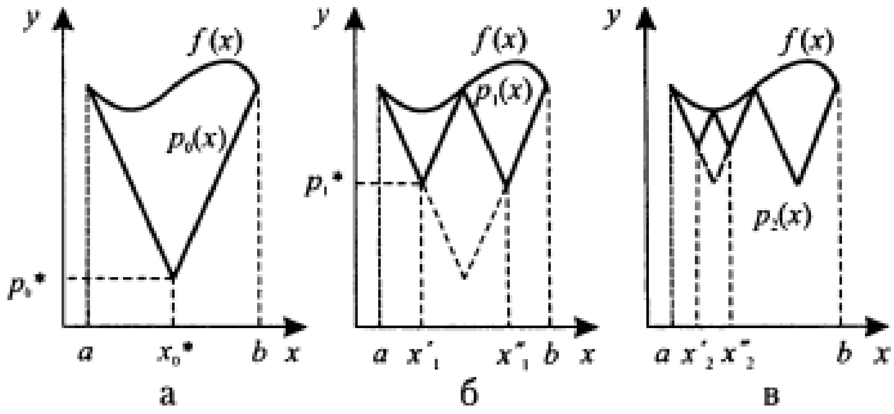


Рис. 3.7. **Построение ломаных** $p_0(x)$, $p_1(x)$, $p_2(x)$

Используя вспомогательную функцию $g(x_0^*, x)$, определим аппроксимирующую функцию

$$p_1(x) = \max\left[p_0(x), g(x_0^*, x)\right],$$

у которой по сравнению с $p_0(x)$ исчезла точка минимума x_0^*, но появились две новые точки локального минимума x_1' и x_1'' (рис. 3.7б):

$$x_1' = x_0^* - \Delta_1, \quad x_1'' = x_0^* + \Delta_1, \text{ где } \Delta_1 = \frac{1}{2L}\left[f(x_0^*) - p_0^*\right], \qquad (3.11)$$

причем $p_1(x_1') = p_1(x_1'') = p_1 = \frac{1}{2}\left[f(x_0^*) + p_0^*\right]$.

Формулы (3.11) получаются исходя из простых геометрических соображений и с учетом того, что тангенсы углов наклона каждого из звеньев ломаной $p_1(x)$ к горизонтальной оси по модулю равны L.

Выберем точку глобального минимума p_1^* функции $p_1(x)$, обозначим ее через x_1^* (в данном случае это x_1' или x_1'', пусть, например, $x_1^* = x_1'$) и положим

$$p_2(x) = \max\left[p_1(x), g(x_1^*, x)\right].$$

У функции $p_2(x)$ по сравнению с $p_1(x)$ вместо x_1^* появились две новые точки локального минимума x_2' и x_2'' (рис. 3.7в), которые находятся аналогично (3.11):

$$x_2' = x_1^* - \Delta_2, \quad x_2'' = x_1^* + \Delta_2, \text{ где } \Delta_2 = \frac{1}{2L}\left[f(x_1^*) - p_1^*\right], \qquad (3.12)$$

причем $p_2(x_2') = p_2(x_2'') = p_2 = \frac{1}{2}\left[f(x_1^*) + p_1^*\right]$.

In [72]:
```
def polyline_method(func, dfunc, a, b, eps, callback=Nor
    if callback is None:
        callback = lambda c, v: 0

    # L = max(np.abs(dfunc(np.linspace(lower_bound, uppe
```

```
    def abs_neg_dfunc(x):
        return -np.abs(dfunc(x))

    x_min, k = fibonacci_method(abs_neg_dfunc, a, b, 0.0
    #print(f'number of eval in fib: {3 + k}')
    L = np.abs(dfunc(x_min))

    k = 0
    func_eval = 0

    func_a = func(a)
    func_b = func(b)
    x = 1 / 2 / L * (func_a - func_b + L * (a + b))
    p = 0.5 * (func_a + func_b + L * (a - b))

    callback(x, func(x))

    d = 1 / 2 / L * (f(x) - p)
    while 2 * L * d > eps:
        callback(x, func(x))
        x1 = x - d
        x2 = x + d

        func_eval += 2
        func_x1 = func(x1)
        func_x2 = func(x2)
        if func_x1 < func_x2:
            x = x1
            func_x = func_x1
        else:
            x = x2
            func_x = func_x2
        p = 0.5 * (func_x + p)
        d = 0.5 / L * (func_x - p)
        k += 1
    callback(x, func(x))
    return x, k
```

4. Изучить зависимость количества итераций от точности ($\varepsilon = 10^{-k}$, k = 1..6) и параметров одного из методов по вариантам (1,…, 5). Сделать визуализацию.

In [73]:
```
iter_number = {'radix_method': [], 'fibonacci_method': [
lower_bound = 0
upper_bound = 1
x0 = 0.2
```

```
for k in range(1, 6 + 1):
    eps = 10**(-k)
    iter_number['radix_method'].append(bitwise_method(fu
    iter_number['fibonacci_method'].append(fibonacci_met
    iter_number['midpoint_method'].append(midpoint_metho
    iter_number['newton_raphson_method'].append(newton_r
    iter_number['polyline_method'].append(polyline_metho
```

In [74]:

```
results = pd.DataFrame(iter_number)
results.index = range(1,7,1)
results

# iter_number
```

Out[74]:

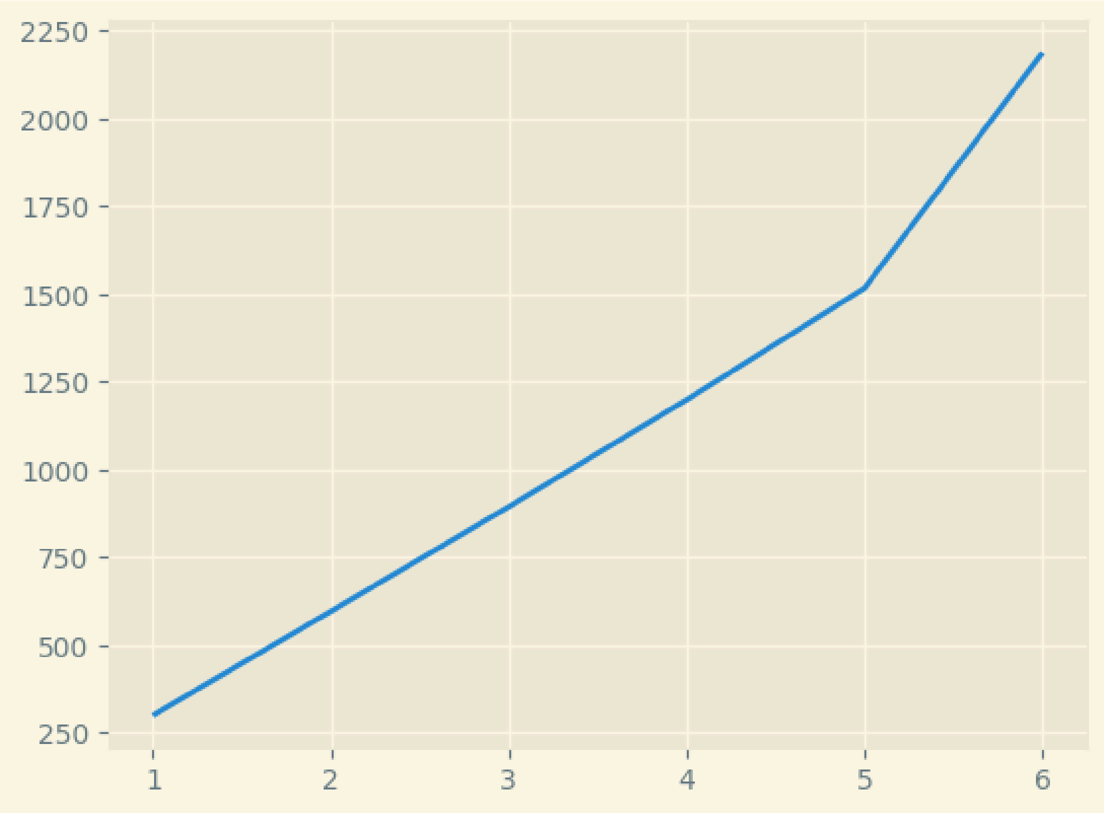
	radix_method	fibonacci_method	midpoint_method	newto
1	299	4	4	
2	595	9	7	
3	893	13	10	
4	1198	18	14	
5	1516	23	17	
6	2187	28	20	

Графики кол-ва итераций от точности

In [75]:

```
results['radix_method'].plot()
```

Out[75]: <AxesSubplot:>

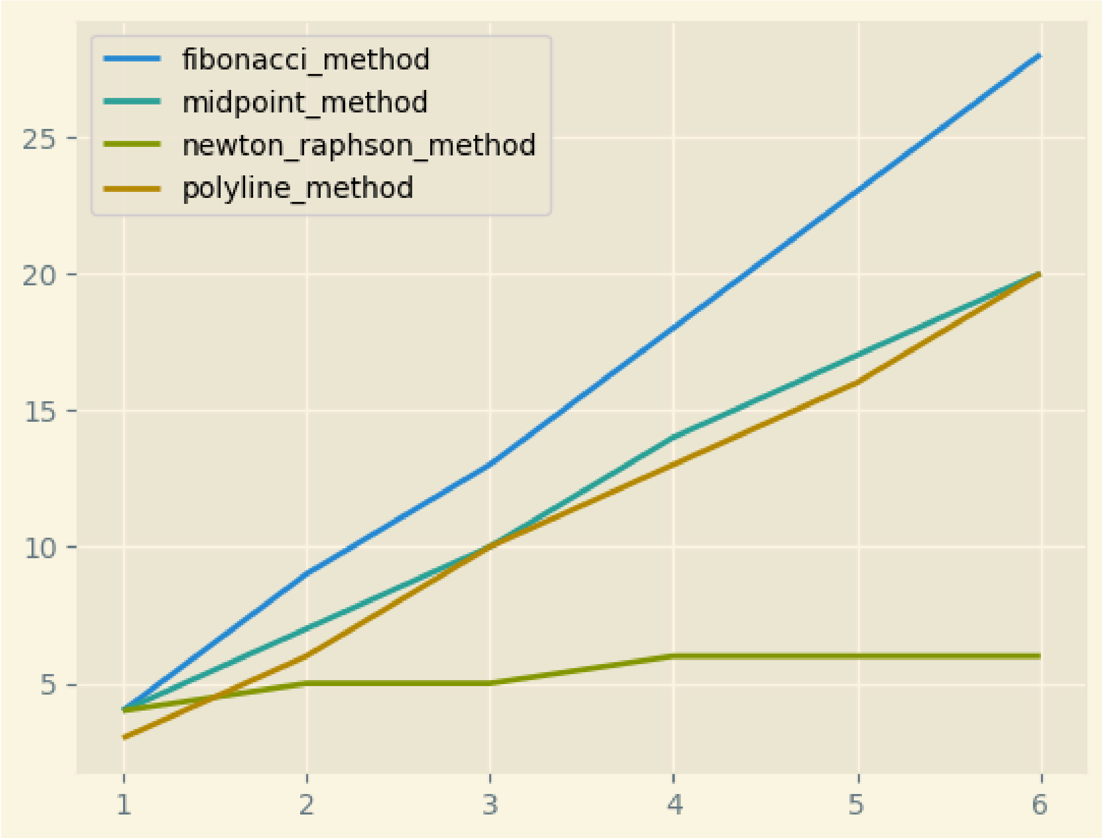


In [76]:
```
list(results)
```

Out[76]:
```
['radix_method',
 'fibonacci_method',
 'midpoint_method',
 'newton_raphson_method',
 'polyline_method']
```

In [77]:
```
results[list(results)[1:]].plot()
```

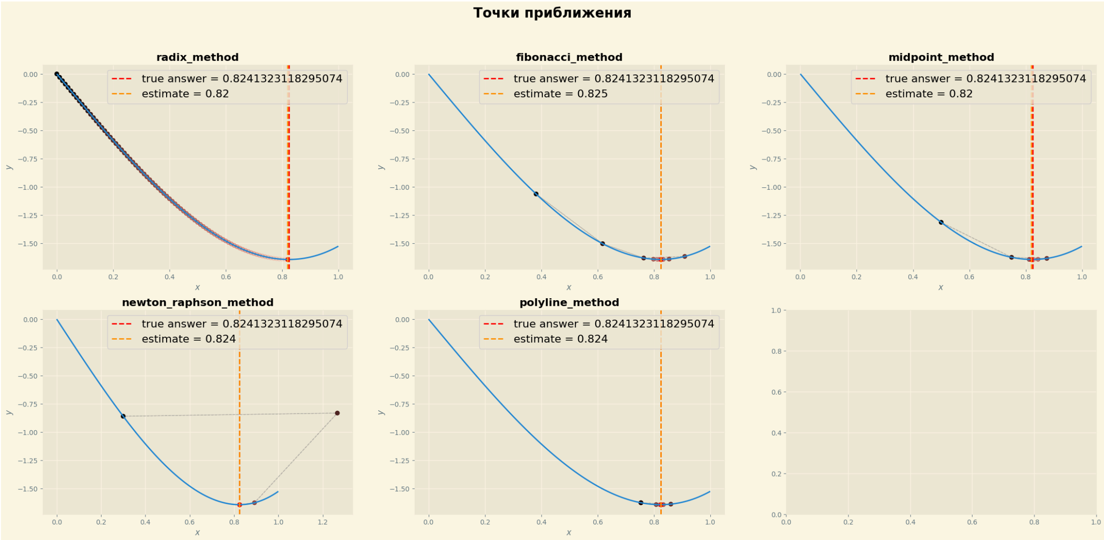
Out[77]:
```
<AxesSubplot:>
```



Визуализация траектории решения

In [78]:
```
eps = 10**-2
fig, axes = plt.subplots(2,3, figsize=(28, 12))
fig.suptitle("Точки приближения", fontweight="bold", for
grid = np.linspace(lower_bound, upper_bound, 100)


callback = LoggingCallback()  # Не забываем про логирова
bitwise_method(func, lower_bound, upper_bound, 100, eps,
plotting(axes, 0, minimum.x[0],'radix_method')

callback = LoggingCallback()
fibonacci_method(func, lower_bound, upper_bound, interva
plotting(axes, 1, minimum.x[0],'fibonacci_method')

callback = LoggingCallback()
midpoint_method(func, deriv1, lower_bound, upper_bound,
plotting(axes, 2, minimum.x[0],'midpoint_method')

callback = LoggingCallback()
newton_raphson_method(func, deriv1, deriv2, 0.3, eps, ca
plotting(axes, 3, minimum.x[0],'newton_raphson_method')
```

```python
callback = LoggingCallback()
polyline_method(func, deriv1, lower_bound, upper_bound,
plotting(axes, 4, minimum.x[0],'polyline_method')
```



5. Для методов и точности $\varepsilon = 10^{-k}$ по вариантам (k = 2..6) привести количество вычислений функции и производных. Результаты разместить в таблице. Построить графики функции и точек приближений к решению.

In [79]:
```python
eps = 10**-2
lower_bound = 0
upper_bound = 1
names = ['func', 'deriv1', 'deriv2']
```

In [80]:
```python
counts_iter = list(results.iloc[1])
counts_iter

eval_number = {'radix_method': [3 + counts_iter[0], 0, 0
               'fibonacci_method': [3 + counts_iter[1],
               'midpoint_method': [0, counts_iter[2], 0]
               'newton_raphson_method': [0, 2 * counts_i
               'polyline_method': [2 + 2 * counts_iter[4

eval_number_df = pd.DataFrame(eval_number)
eval_number_df.index = names
eval_number_df.T
```

Out[80]:

	func	deriv1	deriv2
radix_method	598	0	0
fibonacci_method	12	0	0
midpoint_method	0	7	0
newton_raphson_method	0	10	10
polyline_method	14	17	0

In [81]:
```
[bitwise_method(func, lower_bound, upper_bound, 100, eps
fibonacci_method(func, a=0.1, b=1, interval_length=eps,
midpoint_method(func, deriv1, lower_bound, upper_bound,
newton_raphson_method(func, deriv1, deriv2, x0, eps)[0],
polyline_method(func, deriv1, lower_bound, upper_bound,
minimum.x[0]]
```

Out[81]:
```
[0.8150000000000005,
 0.8243750000000001,
 0.8203125,
 0.8241323352972675,
 0.8242874768836322,
 0.824132318295074]
```

In [82]:
```
[abs(bitwise_method(func, lower_bound, upper_bound, 100,
abs(fibonacci_method(func, a=0.1, b=1, interval_length=e
abs(midpoint_method(func, deriv1, lower_bound, upper_bou
abs(newton_raphson_method(func, deriv1, deriv2, x0, eps)
abs(polyline_method(func, deriv1, lower_bound, upper_bou
```

Out[82]:
```
[False, False, False, False, False]
```

6. Найти минимум функции

$$f(x)=e^x-1-x-\frac{x^2}{2}-\frac{x^3}{6}$$

с точностью 10^{-4} на отрезке [-5, 5] прямыми методами. Объяснить результаты

In [83]:
```
lower_bound = -5
upper_bound = 5
eps = 10**-4
```

In [84]:
```
def f2(x):
    return np.exp(x) - 1 - x - x**2/2 - x**3/6

def deriv1f2(x):
    return np.exp(x) - 1 - x - x**2/2

def deriv2f2(x):
    return np.exp(x) - 1 - x
```

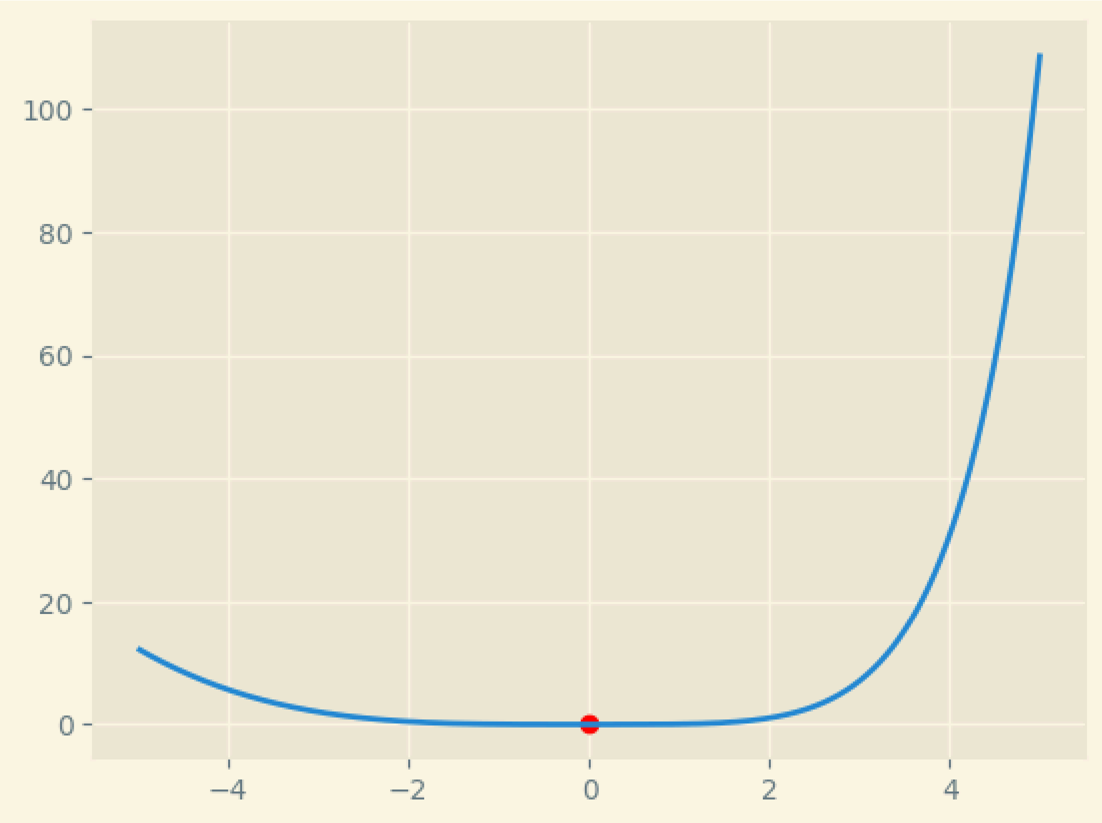
Результат встроенными методами

In [85]:
```
minimum = scipy.optimize.minimize(f2,method = 'Powell',b
minimum
```

Out[85]:
```
 message: Optimization terminated successfully.
 success: True
  status: 0
     fun: -5.020905277629918e-17
       x: [-2.415e-05]
     nit: 4
   direc: [[ 5.418e-05]]
    nfev: 207
```

In [86]:
```
x = np.arange(lower_bound, upper_bound, eps)
plt.plot(x, f2(x))
plt.scatter(minimum['x'], minimum['fun'], color = 'red')
```

Out[86]:
```
<matplotlib.collections.PathCollection at 0x29d10c56800
>
```



Результат написанными методами

In [87]:
```
bitwise_method(f2, lower_bound, upper_bound, 100, eps)
```

Out[87]:
```
(-5.000000000285511e-05, 1052)
```

In [88]:
```
fibonacci_method(f2, a=lower_bound, b=upper_bound, inter
```

Out[88]:
```
(0.00018975414562633557, 23)
```

In [89]:
```
def plotting(axes, i, answer, callback, grid, f, title):
    if axes is not None:
        ax = axes[np.unravel_index(i, shape=axes.shape)]
        x_steps = np.array(callback.x_steps)
        y_steps = np.array(callback.y_steps)
        plot_convergence_1d(
            f, x_steps, y_steps,
            ax, grid, title
        )
        ax.axvline(answer, 0, linestyle="--", c="red",
                    label=f"true answer = {answer}")
        ax.axvline(x_steps[-1], 0, linestyle="--", c="xk
                    label=f"estimate = {np.round(x_steps
        ax.legend(fontsize=16)
```

In [90]:
```
# fig, axes = plt.subplots(1,2, figsize=(20, 6))
# grid = np.linspace(lower_bound, upper_bound, 100)

# callback = LoggingCallback()
# bitwise_method(f2, lower_bound, upper_bound, 180, eps,
# plotting(axes, 0, minimum.x[0], callback, grid, f2, 'r

# callback = LoggingCallback()
# fibonacci_method(f2, lower_bound, upper_bound, interva
# plotting(axes, 1, minimum.x[0], callback, grid, f2, 'f
```

7. Решить задачу минимизации методом Ньютона и его модификациями (по вариантам)

$$f(x) = x \operatorname{arctg} x - \frac{1}{2}\ln(1+x^2)$$

Определить диапазоны допустимых начальных приближений. Объяснить.

In [91]:
```
def f3(x):
    return x * np.arctan(x) - 0.5* np.log(1+x**2)

def deriv1f3(x):
    return np.arctan(x)

def deriv2f3(x):
    return 1 /(1 + x**2)
```

In [92]:
```
lower_bound = -2
upper_bound = 2
x0 = 1.356
eps = 10**-4
```

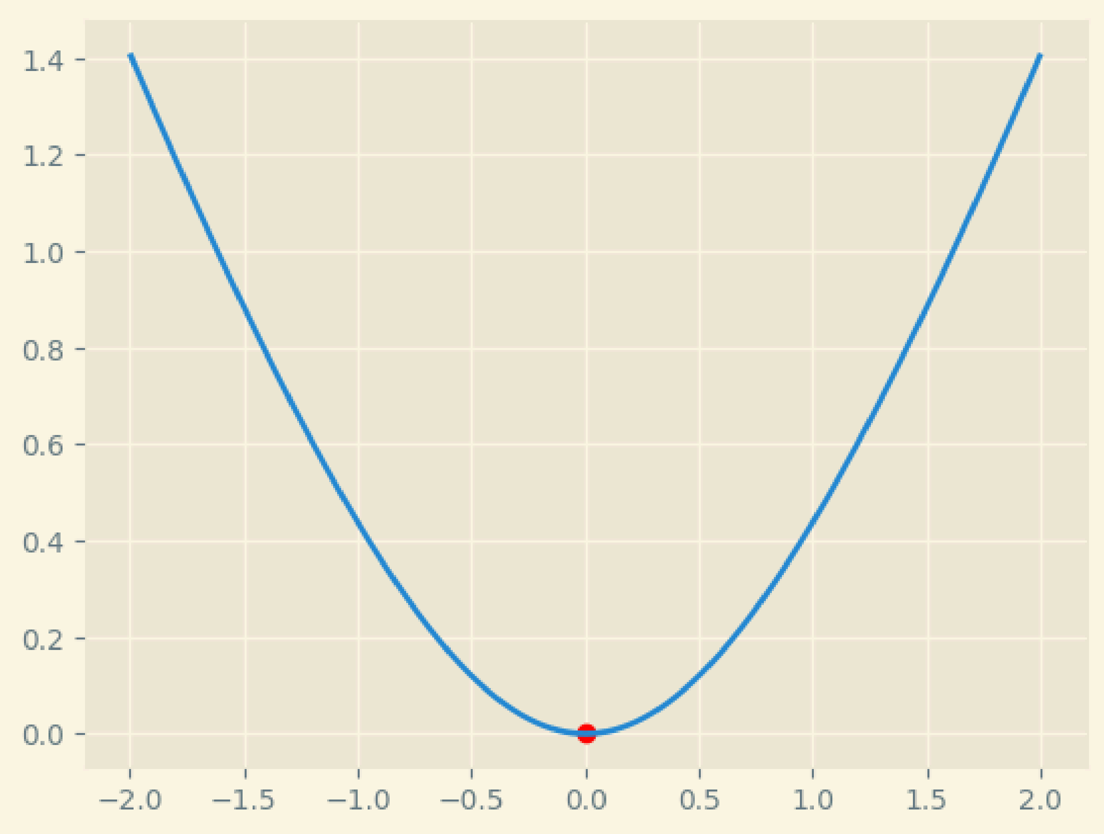
Результат встроенными методами

In [93]:
```
minimum = scipy.optimize.minimize(f3,method = 'Powell',b
minimum
```

Out[93]:
```
 message: Optimization terminated successfully.
 success: True
  status: 0
     fun: 3.0814879110195774e-33
       x: [ 5.551e-17]
     nit: 2
   direc: [[ 1.000e+00]]
    nfev: 14
```

In [94]:
```
x = np.arange(lower_bound, upper_bound, eps)
plt.plot(x, f3(x))
plt.scatter(minimum['x'], minimum['fun'], color = 'red')
```

Out[94]:
```
<matplotlib.collections.PathCollection at 0x29d12dcb7f0
>
```

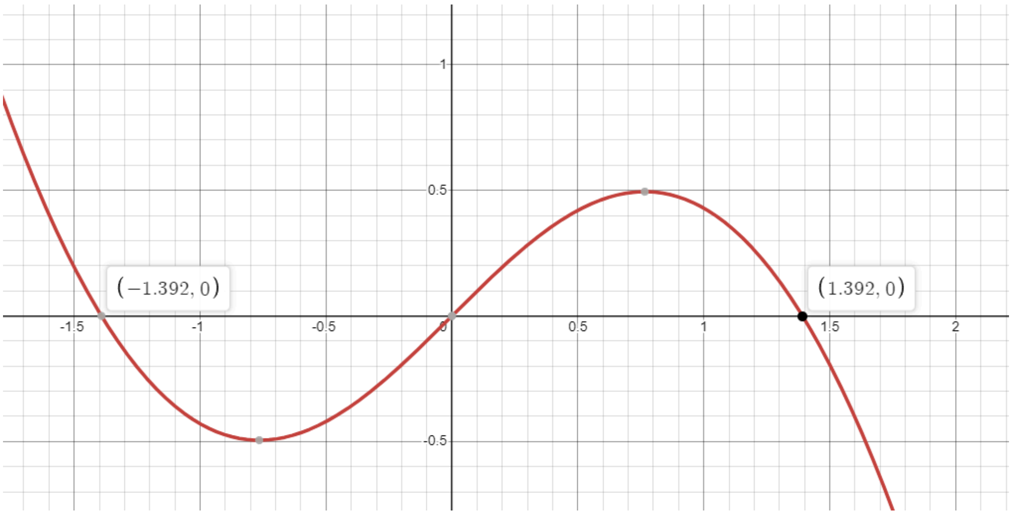


Так как функция симметричка, то решение будет сходиться, если x1 < -x0

Иначе говоря, границами будут являтся нули следующей

$$2x - \arctan(x) \cdot \left(1 + x^2\right)$$

функции:



Результат написанными методами

In [95]:
```
newton_raphson_method(f3, deriv1f3, deriv2f3, x0, eps, e
```

Out[95]:
```
(-8.887252641708726e-15, 7)
```

In [96]:
```
newton_method(f3, deriv1f3, deriv2f3, x0, eps)
```

Out[96]:
```
(-8.887252641708726e-15, 7)
```

In [97]:
```
# fig, axes = plt.subplots(figsize=(10, 6))
# grid = np.linspace(lower_bound, upper_bound, 100)

# callback = LoggingCallback()
# newton_raphson_method(f3, deriv1f3, deriv2f3, x0, eps,
# plotting(np.array(axes), 0, minimum.x[0], callback, gr
```

Вывод:

Исследование методов одномерной минимизации показало, что проще всего и лучше использовать встроенные в матпакеты методы, потому что они уже написаны. Однако, анализируя всё же эти методы, то самым оптимальным получился метод средней точки относительно количества итераций и вычислений, а также простоты реализации. Если особенно важно добиться надежной и устойчивой работы алгоритма, то целесообразно использовать поразрядного поиска как одного из самых стабильных прямых методов. Что касается метода ньютона и его модификаций, то он является самым быстросходящимся. Метод ломанных самый труднореализуемый и имеет дополнительные условия, но его можно использовать для полимодальных функций.