МИНОБРНАУКИ РОССИИ
РГУ НЕФТИ И ГАЗА (НИУ) ИМЕНИ И.М. ГУБКИНА
ФАКУЛЬТЕТ АВТОМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ
КАФЕДРА АВТОМАТИЗИРОВАННЫХ СИСТЕМ УПРАВЛЕНИЯ

ДИСЦИПЛИНА

«Методы и модели оптимизции в автоматизированном управлении НГО»

ОТЧЁТ

по лабораторной работе №1

«Методы одномерной минимизации»

Выполнил:
студент группы АС-21-05, Негробов В.А.
Вариант 43.
Проверила:
старший преодаватель, Степанкина О.А.

МОСКВА 2024

Предварительная подготовка

Подключение используемых библиотек и модулей:

In [1]:
```
import numpy as np
import pandas as pd
from scipy.optimize import minimize
from matplotlib import pyplot as plt
from sympy import lambdify, symbols, diff
import sympy
from functools import partial
from utils import plot_convergence_1d, LoggingCallback,
import scipy
from scipy.optimize import fsolve
plt.style.use('ggplot')
```

Создание символьной переменной, необходимой для работы с библиотекой символьного исчисления SymPy:

In [2]:
```
x_symbol = symbols('x_symbol')
x_symbol
```

Out[2]: x_{symbol}

1. Выбрать вариант задачи, соответствующий вашему номеру из списка группы.

Нижняя и верхняя граница интервала, начальное положение, точность решения задачи библиотекой SciPy:

In [3]:
```
lower_bound = 0.1
upper_bound = 1
```

```
start_x = (lower_bound + upper_bound) / 2
eps = 1e-8
```

Создадим символьную функцию, соответствующую заданной:

In [4]:
```
f_symbol = 10 * x_symbol * sympy.ln(x_symbol) - x_symbol
f_symbol
```

Out[4]:
$$-\frac{x_{symbol}^2}{2} + 10x_{symbol}\log\left(x_{symbol}\right)$$

Первая производная заданной функции:

In [5]:
```
df_symbol = diff(f_symbol, x_symbol)
df_symbol
```

Out[5]:
$$-x_{symbol} + 10\log\left(x_{symbol}\right) + 10$$

Вторая производная заданной функции:

In [6]:
```
d2f_symbol = diff(df_symbol, x_symbol)
d2f_symbol
```

Out[6]:
$$-1 + \frac{10}{x_{symbol}}$$

Преобразование символьных функций в лямбда функции:

In [7]:
```
f = lambdify(x_symbol, f_symbol, 'numpy')
df = lambdify(x_symbol, df_symbol, 'numpy')
d2f = lambdify(x_symbol, d2f_symbol, 'numpy')
```

Визуализация функции:

In [8]:
```
plt.plot((x:=np.linspace(lower_bound, upper_bound, 100))
plt.xticks(np.linspace(lower_bound, upper_bound, 10));
```

В библиотеке SciPy представленно множество методов, предназначенных для минимизации функций. Рассмотрим основные из них.

Минимизция методом из библиотеки SciPy: Функция Brent: В численном анализе метод Брента представляет собой гибридный алгоритм поиска, сочетающий метод деления пополам, метод секущей и обратную квадратичную интерполяцию.
https://docs.scipy.org/doc/scipy/reference/generated/scipy.optin

Функция minimize может использовать следующие методы: ‘Nelder-Mead’, ‘Powell’, ‘CG’, ‘BFGS’, ‘Newton-CG’, ‘L-BFGS-B’, ‘TNC’, ‘COBYLA’, ‘SLSQP’, ‘trust-constr’, ‘dogleg’, ‘trust-ncg’, ‘trust-exact’, ‘trust-krylov’, custom.
https://docs.scipy.org/doc/scipy/reference/generated/scipy.optin

Функция golden реализует метод золотого сечения.
https://docs.scipy.org/doc/scipy/reference/generated/scipy.optin

Получить решение удалось получить только следующим методом:

In [9]:
```
x0 = np.array([start_x])
res = minimize(f, x0, method='nelder-mead',
               options={'xatol': eps, 'disp': True})
print(f'min x: {res.x}, f(x) = {f(res.x)}')
```

```
Optimization terminated successfully.
         Current function value: -3.749081
         Iterations: 27
         Function evaluations: 54
min x: [0.38221242], f(x) = [-3.74908101]
```

3. Написать скрипты, реализующие следующие методы

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

Рассмотрим усовершенствованный метод перебора с целью уменьшения количества значений $f(x)$, которые необходимо находить в процессе минимизации.

Во-первых, если $f(x_i) \le f(x_{i+1})$, то отпадает необходимость вычислять $f(x)$ в точках x_{i+2}, x_{i+3} и т.д., так как из унимодальности функции $f(x)$ следует, что $x^* \le x_{i+1}$.

Во-вторых, разумно было бы сначала определить отрезок, содержащий точку x^* с небольшой точностью, а затем искать ее на этом отрезке с меньшим шагом дискретизации, повышая точность.

Такая возможность улучшения метода перебора реализована в методе поразрядного поиска, в котором перебор точек отрезка происходит сначала с шагом $\Delta = x_{i+1} - x_i > \varepsilon$ до тех пор, пока не выполнится условие $f(x_i) \le f(x_{i+1})$ или пока очередная из этих точек не совпадет с концом отрезка. После этого шаг уменьшается (обычно в четыре раза), и перебор точек с новым шагом производится в противоположном направлении до тех пор, пока значения $f(x)$ не перестанут уменьшаться или очередная точка не совпадет с концом отрезка и т.д. Процесс завершается, когда перебор в данном направлении закончен, а использованный при этом шаг дискретизации не превосходит ε.

In [10]:
```
def radix_method(func, a, b, p, eps, callback=None):

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

    k = 0
    h = (b - a) / p
    x = a

    func_x_prev = func(x)
    callback(x, func_x_prev)
    func_x = func(x+h)
    x += h
    callback(x, func_x)

    while(func_x < func_x_prev and x < b):
        func_x_prev = func_x
        x +=h
        func_x = func(x)
        callback(x, func_x)
        k += 1

    h = eps
    func_x_prev = func_x
    x -= h
    func_x = func(x)
    callback(x, func_x)

    while(func_x <= func_x_prev and x > a):
```

```
        func_x_prev = func_x
        x -= h
        func_x = func(x)
        callback(x, func_x)
        k += 1
    callback(x + h, f(x + h))

    return x + h, k
```

Метод Фибоначчи

Для построения эффективного метода одномерной минимизации, работающего по принципу последовательного сокращения интервала неопределенности, следует задать правило выбора на каждом шаге двух внутренних точек. Конечно, желательно, чтобы одна из них всегда использовалась в качестве внутренней и для следующего интервала. Тогда количество вычислений функции сократится вдвое и одна итерация потребует расчета только одного нового значения функции. В методе Фибоначчи реализована стратегия, обеспечивающая максимальное гарантированное сокращение интервала неопределенности при заданном количестве вычислений функции и претендующая на оптимальность. Эта стратегия опирается на числа Фибоначчи.

Реализация функции, вычисляющей числа Фибоначчи, которые не превышают заданное значение и ещё одного:

In [11]:
```
def fibonacci_numbers(max_value):

    num1, num2 = 1, 1

    yield num1

    if num1 > max_value and  num2 > max_value:
        return

    if num1 < max_value and num2 > max_value:
        yield num2

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

Метод Фибонначи:

In [12]:
```
def fibonacci_method(func, a, b, interval_length, eps, c
```

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

    max_value = (b - a) / interval_length
    fib_nums = [num for num in fibonacci_numbers(max_val
    length = len(fib_nums) - 1
    y = a + fib_nums[length - 2] / fib_nums[length] * (b
    z = a + fib_nums[length - 1] / fib_nums[length] * (b
    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_nums[length - k - 3] / fib_nums[
            callback(b, func_z)
        else:
            less = False
            a, y = y, z
            z = a + fib_nums[length - k - 2] / fib_nums[
            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
```

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

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

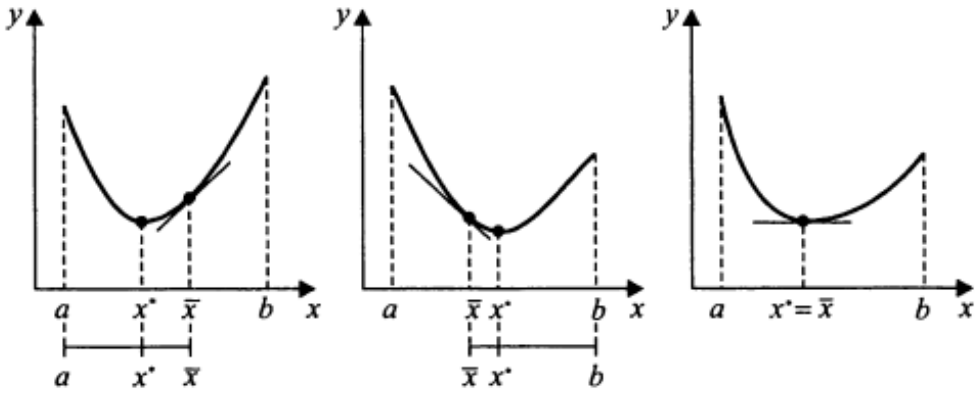


Рис. 3.1. **Иллюстрация исключения отрезков методом средней точки**

In [13]:
```
def midpoint_method(func, dfunc, 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 = dfunc(x)
        callback(x, func(x))
        if dfunc_x > 0:
            b = x
        else:
            a = x
        interval /= 2
        k += 1
    return x, k
```

Метод Ньютона-Рафсона

Стратегия метода Ньютона–Рафсона [Newton–Raphson] состоит в построении последовательности точек $\{x^k\}, k = 0,1,\ldots$, таких, что $f(x^{k+1}) < f(x^k)$, $k = 0,1,\ldots$. Точки последовательности вычисляются по правилу

$$x^{k+1} = x^k - t_k H^{-1}(x^k)\nabla f(x^k), \quad k = 0,1,\ldots , \tag{7.4}$$

где x^0 задается пользователем, а величина шага t_k определяется из условия

$$\varphi(t_k) = f\left(x^k - t_k H^{-1}(x^k)\nabla f(x^k)\right) \to \min_{t_k}. \tag{7.5}$$

Задача (7.5) может решаться либо аналитически с использованием необходимого условия минимума $\frac{d\varphi}{dt_k} = 0$ с последующей проверкой достаточного условия $\frac{d^2\varphi}{dt_k^2} > 0$, либо численно как задача

$$\varphi(t_k) \to \min_{t_k \in [a,b]}, \tag{7.6}$$

где интервал $[a,b]$ задается пользователем.

Если функция $\varphi(t_k)$ достаточно сложна, то возможна ее замена полиномом $P(t_k)$ второй или третьей степени и тогда шаг t_k может быть определен из условия $\frac{dP}{dt_k} = 0$ при выполнении условия $\frac{d^2P}{dt_k^2} > 0$.

При численном решении задачи определения величины шага степень близости найденного значения t_k к оптимальному значению t_k^*, удовлетворяющему условиям $\frac{d\varphi}{dt_k} = 0$, $\frac{d^2\varphi}{dt_k^2} > 0$, зависит от задания интервала $[a,b]$ и точности методов одномерной минимизации [28].

Реализация метода Ньютона, используемого в методе Ньютона-Рафсона:

In [14]:
```
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:

In [15]:
```
def newton_raphson_method(func, dfunc, d2func, x, eps, l
                          n_eps=0.01, max_iter=10_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
        x = x + d
```

```
        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 ломанных

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 $[a, b]$.

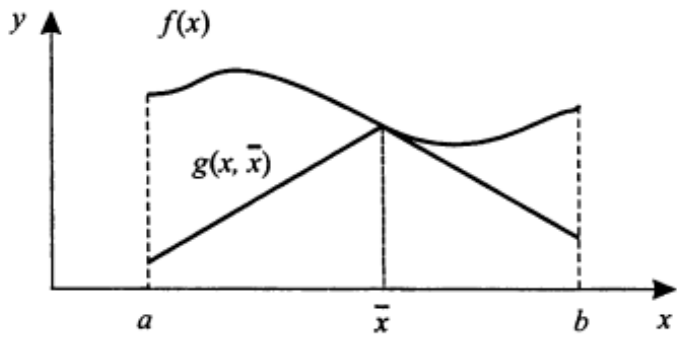


Рис. 3.6. **График функции** $g(\bar{x}, x)$

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

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

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 $p_0^* = y_0$.

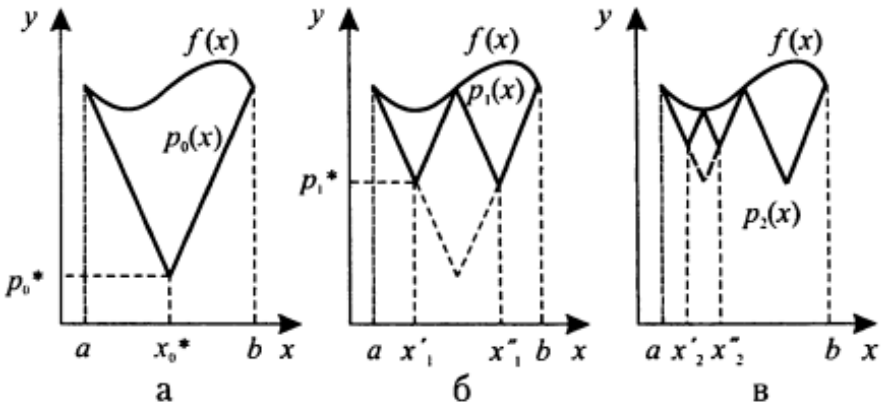


Рис. 3.7. **Построение ломаных** $p_0(x)$, $p_1(x)$, $p_2(x)$

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):

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

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):

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

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

Пусть теперь функция $p_{k-1}(x)$ построена. Выбрав точку глобального минимума p^*_{k-1} функции $p_{k-1}(x)$ и обозначив ее через x^*_{k-1}, определим функцию

$$p_k(x) = \max\left[p_{k-1}(x), g(x^*_{k-1}, x)\right].$$

Новые точки локального минимума x'_k и x''_k функции $p_k(x)$, появившиеся взамен x^*_{k-1}, а также значения $p_k(x)$ в этих точках находятся по формулам, аналогичным (3.11) и (3.12):

$$x'_k = x^*_{k-1} - \Delta_k, \quad x''_k = x^*_{k-1} + \Delta_k, \text{ где } \Delta_k = \frac{1}{2L}\left[f(x^*_{k-1}) - p^*_{k-1}\right],$$

$$p_k(x'_k) = p_k(x''_k) = p_k = \frac{1}{2}\left[f(x^*_{k-1}) + p^*_{k-1})\right].$$

In [16]:

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

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

        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 ($\mathcal{E} = 10^{-k}$, $k = 1..6$). Сделать визуализацию.

Вычисление количества итераций, выполняемых методами для заданных значений точности:

In [17]:
```
iter_number = {'epsilon': [], 'radix_method': [], 'fibor
               'midpoint_method': [], 'newton_raphson_me

for k in range(1, 6 + 1):
    eps = 10**(-k)
    iter_number['epsilon'].append(f'1e-{k}')
    iter_number['radix_method'].append(radix_method(f, 1
    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
df_iter_number = pd.DataFrame(iter_number)
df_iter_number = df_iter_number.set_index('epsilon')
df_iter_number
```

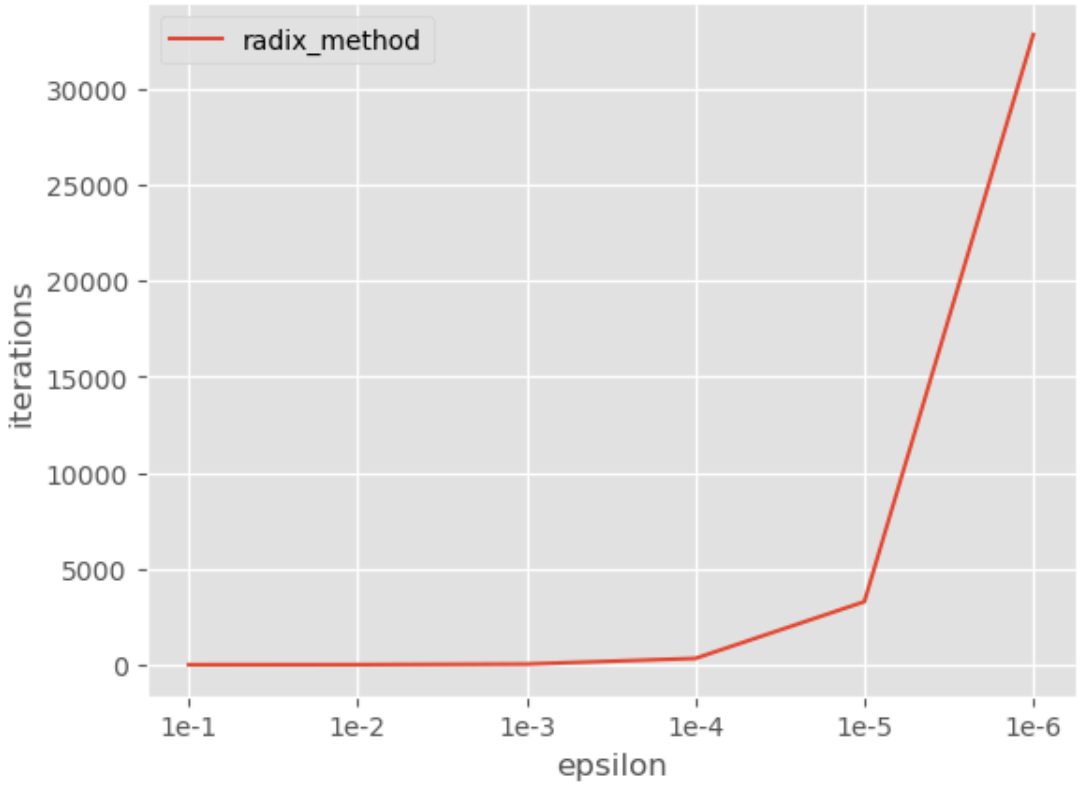
```
number of eval in fib: 15
number of eval in fib: 15
number of eval in fib: 15
number of eval in fib: 15
number of eval in fib: 15
number of eval in fib: 15
```

Out[17]:

epsilon	radix_method	fibonacci_method	midpoint_method
1e-1	6	3	4
1e-2	9	8	7
1e-3	39	12	10
1e-4	334	17	14
1e-5	3285	22	17
1e-6	32794	27	20

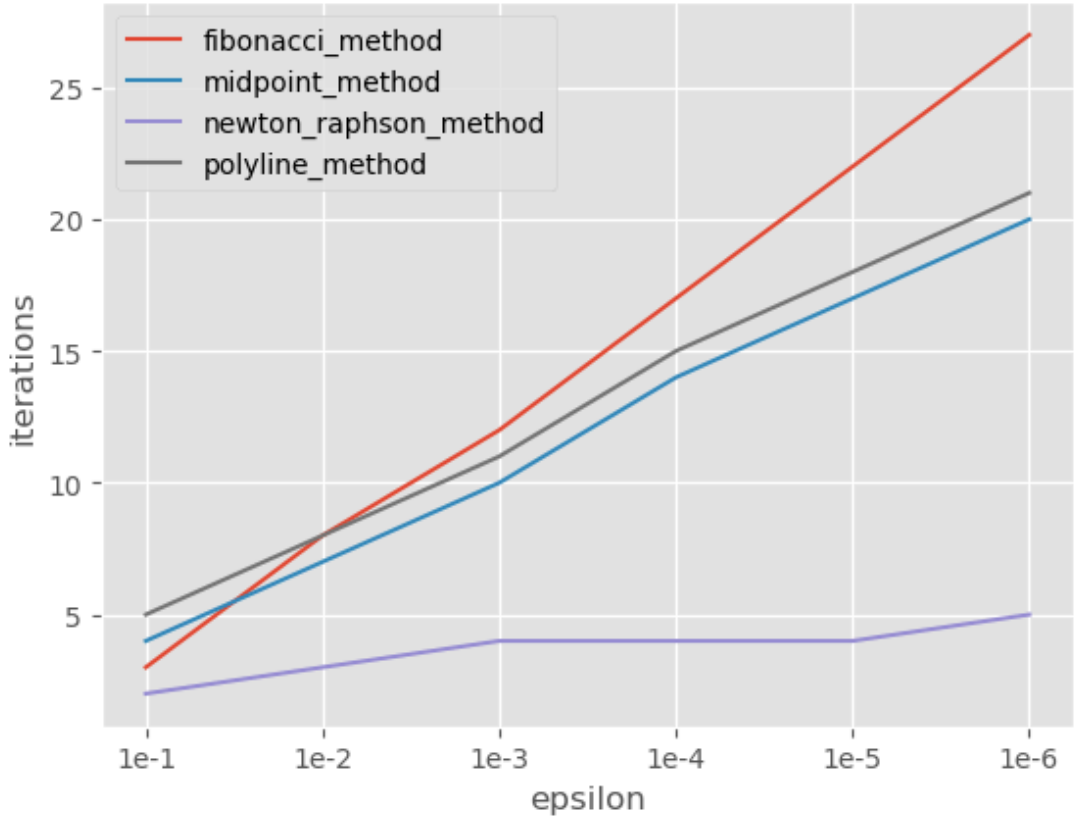
Визуализация количества итераций, необходимых методу поразрядного поиска:

In [18]:
```
plt.plot(range(6), iter_number['radix_method'], label='r
plt.legend()
plt.xlabel('epsilon')
plt.ylabel('iterations')
plt.xticks(range(6), [f'1e-{k}' for k in range(1, 7)]);
```



Визуализация количества итераций, необходимых остальным методам:

In [19]:
```
try:
    iter_number.pop('radix_method')
    iter_number.pop('epsilon')
except KeyError:
    print('elements was deleted')

for method, iters in iter_number.items():
    plt.plot(range(6), iters, label=method)
plt.legend()
plt.xlabel('epsilon')
plt.ylabel('iterations')
plt.xticks(range(6), [f'1e-{k}' for k in range(1, 7)]);
```



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

Заданная точность:

In [20]:
```
eps = 1e-4
```

Подсчёт количества итераций:

In [21]:
```
xr, kr = radix_method(f, lower_bound, upper_bound, 20, e
xf, kf = fibonacci_method(f, a=0.1, b=1, interval_length
xm, km = midpoint_method(f, df, lower_bound, upper_bound
xn, kn = newton_raphson_method(f, df, d2f, x0, eps)
xp, kp = polyline_method(f, df, lower_bound, upper_bound
```

```
number of eval in fib: 15
```

In [22]:
```
abs(xr - res.x) < eps, abs(xf - res.x) < eps, abs(xm - r
```

Out[22]:
```
(array([ True]),
 array([ True]),
 array([ True]),
 array([ True]),
 array([ True]))
```

Количество вычислений функции и её производных:

In [23]:
```
eval_number = {'method': ['f evals', 'df evals', 'd2f ev
               'radix_method': [3 + kr, 0, 0],
               'fibonacci_method': [3 + kf, 0, 0],
               'midpoint_method': [0, km, 0],
               'newton_raphson_method': [0, 2 * kn, 2 *
               'polyline_method': [2 + 2 * kp, 15, 0]}

df_eval_number = pd.DataFrame(eval_number)
df_eval_number = df_eval_number.set_index('method')
df_eval_number.T
```

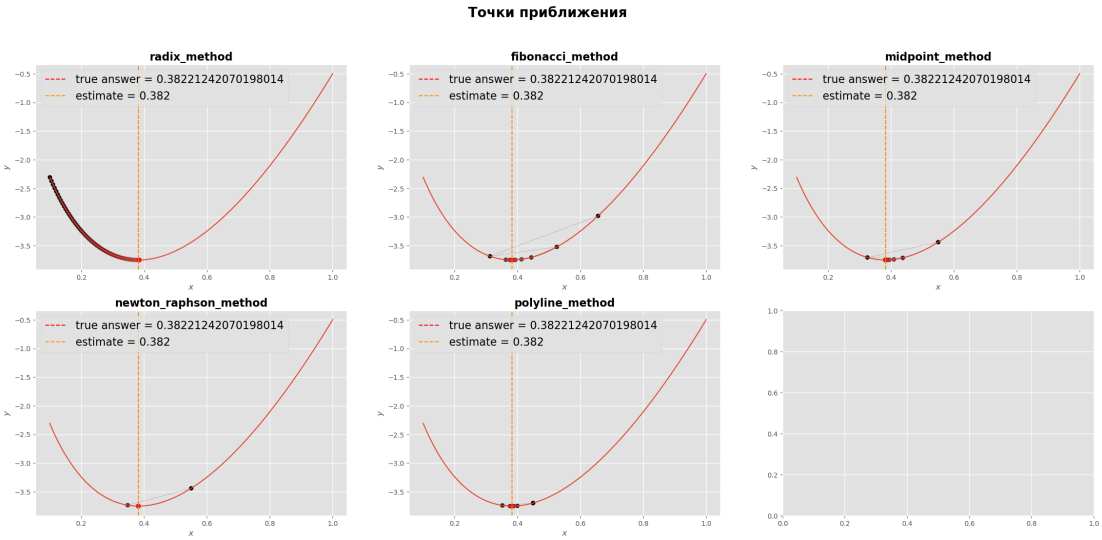
Out[23]:

method	f evals	df evals	d2f evals
radix_method	337	0	0
fibonacci_method	20	0	0
midpoint_method	0	14	0
newton_raphson_method	0	8	8
polyline_method	32	15	0

Визуализация приближений:

In [24]:
```
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
radix_method(f, lower_bound, upper_bound, 180, eps, call
plotting(axes, 0, res.x[0], callback, grid, f, 'radix_me

callback = LoggingCallback()
fibonacci_method(f, lower_bound, upper_bound, interval_l
plotting(axes, 1, res.x[0], callback, grid, f, 'fibonacc

callback = LoggingCallback()
midpoint_method(f, df, lower_bound, upper_bound, eps, ca
plotting(axes, 2, res.x[0], callback, grid, f, 'midpoint

callback = LoggingCallback()
newton_raphson_method(f, df, d2f, start_x, eps, eps, cal
plotting(axes, 3, res.x[0], callback, grid, f, 'newton_r

callback = LoggingCallback()
polyline_method(f, df, lower_bound, upper_bound, eps, ca
plotting(axes, 4, res.x[0], callback, grid, f, 'polyline
```

```
number of eval in fib: 15
```



6. Найти минимум функции

Создание символьное выражение

In [25]:
```
f_symbol = sympy.exp(x_symbol) - 1 - x_symbol - x_symbol
f_symbol
```

Out[25]:
$$-\frac{x_{symbol}^3}{6} - \frac{x_{symbol}^2}{2} - x_{symbol} + e^{x_{symbol}} - 1$$

Преобразуем символьное выражение к лямбда функции:

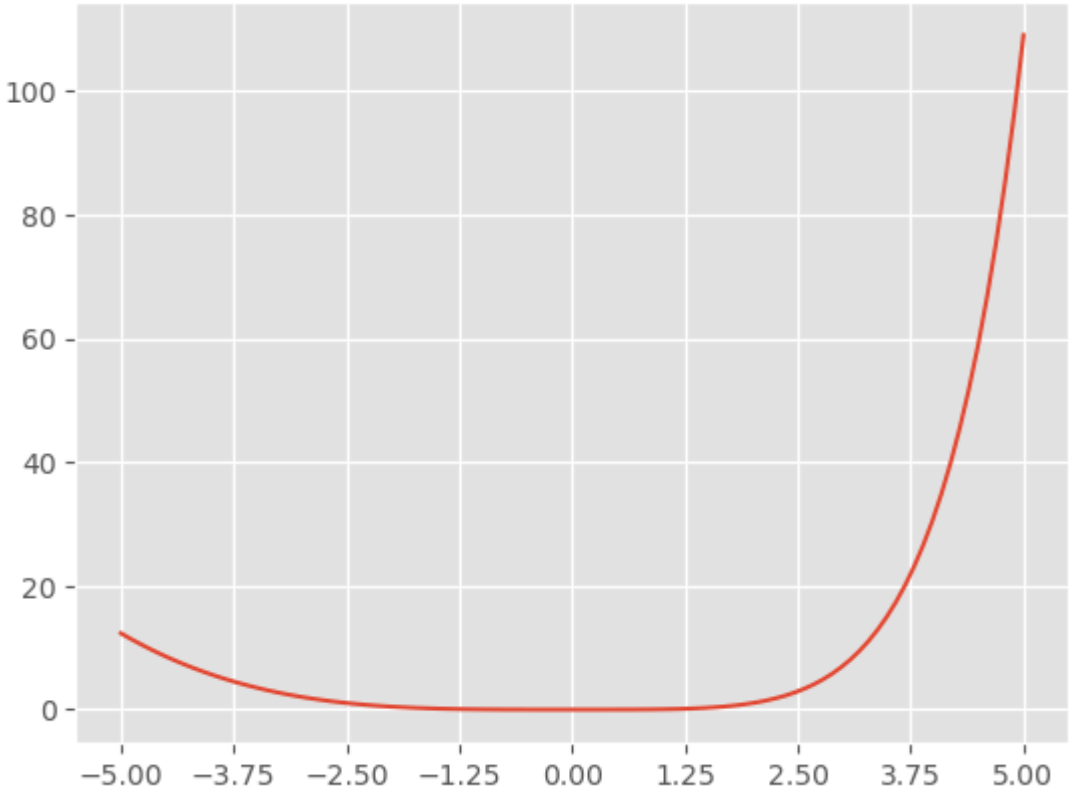
In [26]:
```
f = lambdify(x_symbol, f_symbol, 'numpy')
```

Зададим границы поиска мимимума функции:

In [27]:
```
lower_bound = -5
upper_bound = 5
eps = 1e-4
```

Визуализация функции:

In [28]:
```
plt.plot((x:=np.linspace(lower_bound, upper_bound, 100))
plt.xticks(np.linspace(lower_bound, upper_bound, 9));
```



Минимизация функции методом библиотеки SciPy:

In [29]:
```
x0 = np.array([1])
res = minimize(f, x0, method='nelder-mead',
               options={'xatol': eps, 'disp': True})
print(f'min x: {res.x}, f(x) = {f(res.x)}')
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 17
         Function evaluations: 35
min x: [-8.8817842e-16], f(x) = [0.]
```

Нахождение минимума функции и необходимого для минимизации с заданной точностью количества итераций методом поразрядного поиска и методом Фибонначи:

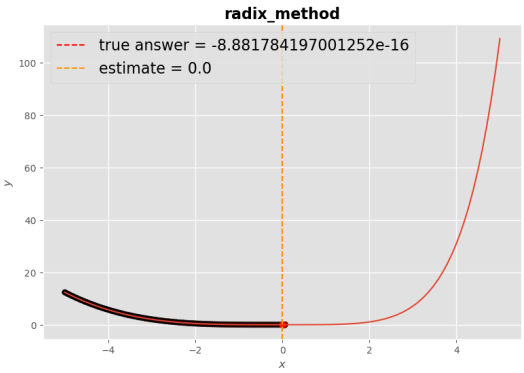
In [30]:
```
x, k = radix_method(f, lower_bound, upper_bound, 20, eps
k, round(x, 4), f(x), abs(round(x, 4) - round(res.x[0],
```

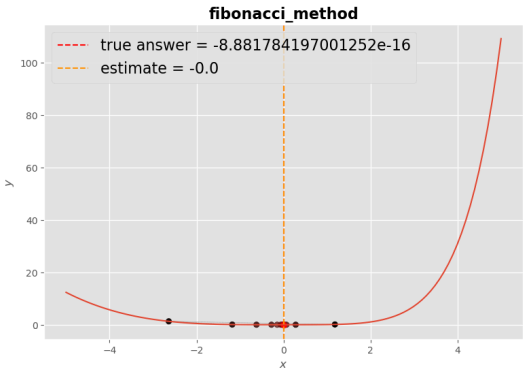
Out[30]:
```
(5009, 0.0001, -1.1102230246251565e-16, True)
```

In [31]:
```
x, k = fibonacci_method(f, a=lower_bound, b=upper_bound,
k, round(x, 4), f(x), abs(round(x, 4) - res.x[0]) < eps
```

Out[31]:
```
(22, -0.0001, 0.0, True)
```

In [32]:
```
fig, axes = plt.subplots(1,2, figsize=(20, 6))
grid = np.linspace(lower_bound, upper_bound, 100)

callback = LoggingCallback()
radix_method(f, lower_bound, upper_bound, 180, eps, call
plotting(axes, 0, res.x[0], callback, grid, f, 'radix_me

callback = LoggingCallback()
fibonacci_method(f, lower_bound, upper_bound, interval_l
plotting(axes, 1, res.x[0], callback, grid, f, 'fibonacc
```





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

Заданная функция, записанная в виде символьного выражения:

In [33]:
```
f_symbol = x_symbol * sympy.atan(x_symbol) - 0.5 * sympy
f_symbol
```

Out[33]: $x_{symbol} \operatorname{atan}\left(x_{symbol}\right) - 0.5 \log\left(x_{symbol}^{2} + 1\right)$

Первая производная заданной функции:

In [34]:
```
df_symbol = diff(f_symbol, x_symbol)
df_symbol
```

Out[34]: $\operatorname{atan}\left(x_{symbol}\right)$

Вторая производная заданной функции:

In [35]:
```
d2f_symbol = diff(df_symbol, x_symbol)
d2f_symbol
```

Out[35]:
$$\frac{1}{x_{symbol}^2 + 1}$$

Преобразование символьных функций к лямбда-функциям:

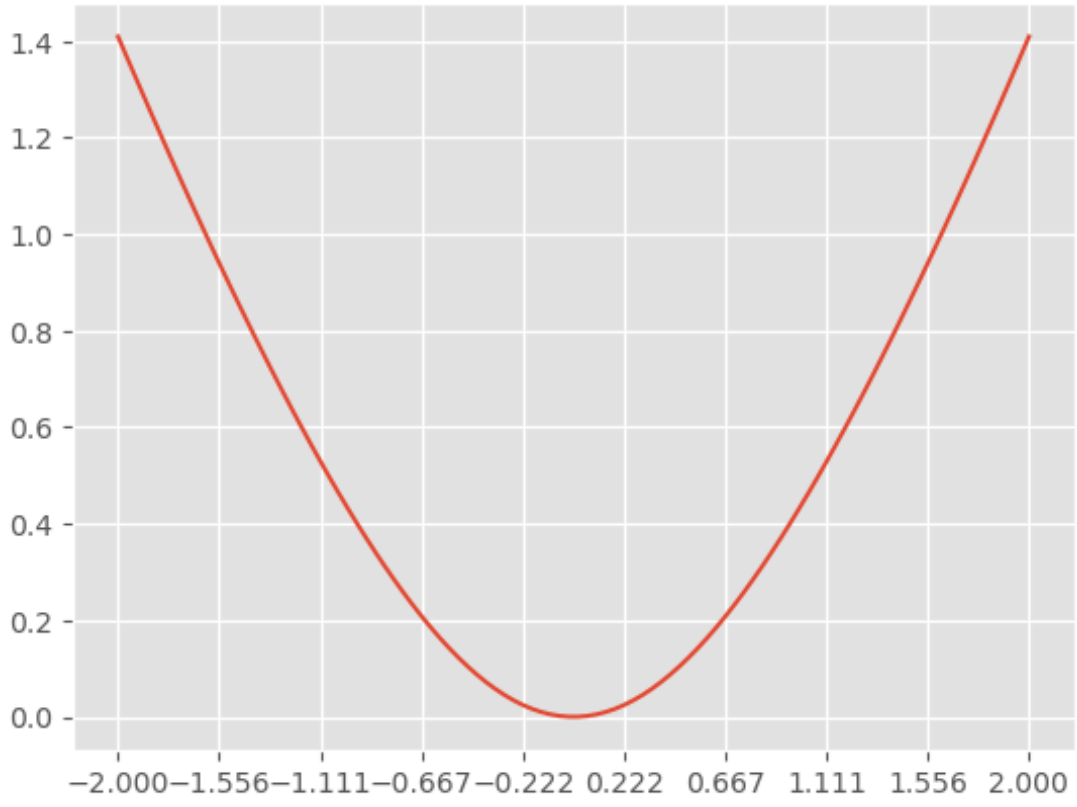
In [36]:
```
f = lambdify(x_symbol, f_symbol, 'numpy')
df = lambdify(x_symbol, df_symbol, 'numpy')
d2f = lambdify(x_symbol, d2f_symbol, 'numpy')
```

Задание начальных значений, начального приближения, точности решения:

In [37]:
```
lower_bound = -2
upper_bound = 2
x_start = 1.35
eps = 1e-4
```

Визуализация заданной функции:

In [38]:
```
plt.plot((x:=np.linspace(lower_bound, upper_bound, 100))
plt.xticks(np.linspace(lower_bound, upper_bound, 10));
```



Решение задачи минимизации методом библиотеки SciPy:

In [39]:
```
x0 = np.array([x_start])
res = minimize(f, x0, method='nelder-mead',
               options={'xatol': eps, 'disp': True})
print(f'min x: {res.x}, f(x) = {f(res.x)}')
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 18
         Function evaluations: 36
min x: [2.22044605e-15], f(x) = [4.93038066e-30]
```

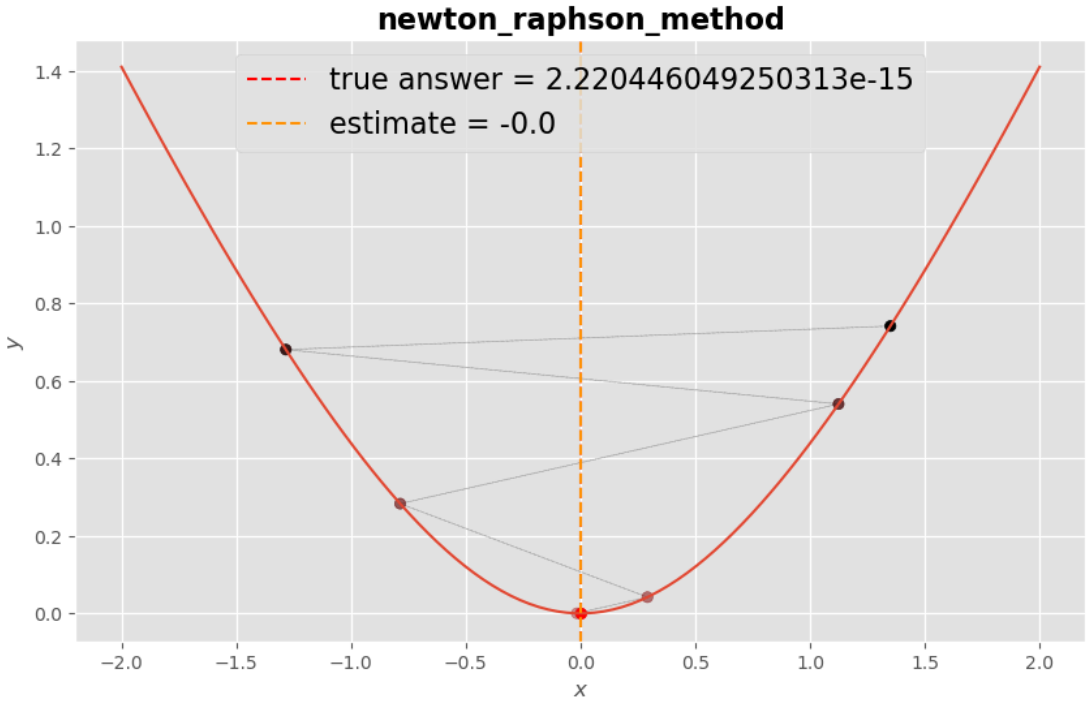
Решение задачи методом Ньютона-Рафсона, необходимое для решения количество итераций:

In [40]:
```
x, k = newton_raphson_method(f, df, d2f, x_start, eps)
k, x, f(x), abs(x - res.x) < eps
```

Out[40]:
```
(7, -1.561293480026489e-17, 2.437637330773225e-34, arra
y([ True]))
```

Визуализаций приближений, полученных в ходе решения:

In [41]:
```
fig, axes = plt.subplots(figsize=(10, 6))
grid = np.linspace(lower_bound, upper_bound, 100)
callback = LoggingCallback()
newton_raphson_method(f, df, d2f, 1.35, eps, 0.01, callb
plotting(np.array(axes), 0, res.x[0], callback, grid, f,
```



Решение будет сходиться, если выполняется неравенство, которое является верным, если функция симметрична:

$$x_0 - \frac{f'(x)}{f''(x)} < -x_0$$

Следовательно, необходимо найти решение уравнения:

$$2x - arctan(x) * (x^2 + 1) = 0$$

Конечно, используемый метод Ньютона-Рафсона должен бороться с данной проблемой. Однако, при увеличении x из-за того, что первая производная стремится к константе, а вторая к 0, их частное стремится к бесконечности. В итоге это приводит к тому, что переменная d принимает большое значение и происходит переполнение переменной.

In [42]:
```
def func(x):
    return 2*x - np.arctan(x) * (x**2 + 1)
```

Найшём решение данного уравнения:

In [43]:
```
root = fsolve(func, [1])
```

```
root
```

Out[43]:
```
array([1.3917452])
```

8. Сформулировать выводы об эффективности методов и рекомендации по их применению.

Метод поразрядного поиска выполняет большое количество итераций, при этом не требует дифференцируемости функции. Его преимуществами является возможность поиска минимума на любых участках функции и на любом интервале и устойчивость.

Метод Фибоначчи является методом нулевого порядка (не требует информации о производной), который способен относительно быстро найти минимум функции.

Метод средней точки использует информацию только о производной функции, которая вычисляется 1 раз за итерацию. Также он достаточно быстро сходится, так как каждый раз отрезок, на котором находится точка минимума, уменьшается в 2 раза.

Метод Ньютона-Рафсона является самым быстро сходящимся методом, но при этом он требует знание функции, её первой и второй производной, получение которых может быть затруднительным.

Метод ломаных выполняет по 2 вычисления функции на итерации, требует знание функции и постоянной Липшица этой функции. Он способен выполнять минимизацию многомодальных функций, поэтому мало

смысла применять его для минимизации унимодальных функций.

Методы нулевого порядка могут испытывать проблемы нахождения минимума функции, если он находится на плато или участке с малой производной. По сравнению с методом поразрядного поиска, метод Фибоначчи выигрывает в решении таких задач.

Метод Ньютона-Рафсона является самым быстро сходящимся, но при этом его стоит использовать только вблизи минимума функции, так как он имеет свойство притягиваться ко всем экстремумам, максимумам в том числе. Также, в случае, если вторая производная на порядок меньше первой, то метод может не сойтись в целом или может произойти переполнение переменных.