МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Факультет прикладной математики и инворматики
Кафедра вычислительной математики

Отчет по лабораторной работе 1
"Нелинейные уравнения"
Вариант 5

Выполнил:
Карпович Артём Дмитриевич
студент 3 курса 7 группы

Преподаватель:
Репников Василий Иванович

Минск, 2024

Задача 1

На примере уравнения

$$\sqrt[3]{x} * \tan x = x^2 + 1$$

провести сравнительный анализ следующих методов решения нелинейных уравнений: 1) Метод простой итерации; 2) Метод Ньютона; 3) Метод Стеффенсена; 4) Метод Чебышева третьего порядка

Постановка задачи

Пусть задана функция $f(x)$ действительного переменного xR. Требуется найти корни уравнения

$$f(x) = 0,$$

или, что то же самое, нули функции $f(x)$. Выясним, является ли задача корректно поставленной. Для ответа на вопрос существования и единственности решения введем теорему из математического анализа.

Теорема 1 *Если функция $f(x)$ непрерывна на отрезке $[a, b]$ и принимает на его концах значения разных знаков, то на этом отрезке существует по крайней мере один корень уравнения $f(x) = 0$. Если при этом функция $f(x)$ будет монотонной на отрезке $[a, b]$, то она может иметь только один корень.*

В данном случае вопрос непрерывной зависимости от входных данных отпадает.

Отделение корней

Данный этап будет общим для всех рассматриваемых методов, так как в независимости от используемого метода сами корни уравнения не меняют своего расположения на числовой прямой. Этот этап необходим для того, чтобы в процессе приближения корня мы случайным образом не пришли к другому корню, не тому, для которого мы считали приближение.

Суть этого этапа заключается в том, что мы выбираем для каждого корня отрезок, в котором он находится, при этом мы гарантируем, что других корней в этом отрезке нет.

Отделять корни будем, используя графический метод. Для начала определим исследуемую функцию. Пусть

$$f(x) = \sqrt[3]{x} * \tan x - x^2 - 1 = 0,$$

то есть корни этого уравнения мы и будем искать. Сразу заметим, что эта функция непрерывная, так как является результатом сложения непрерывных функций.

[1]:
```
import math
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

pd.options.display.float_format ='{:,.8f}'.format
```

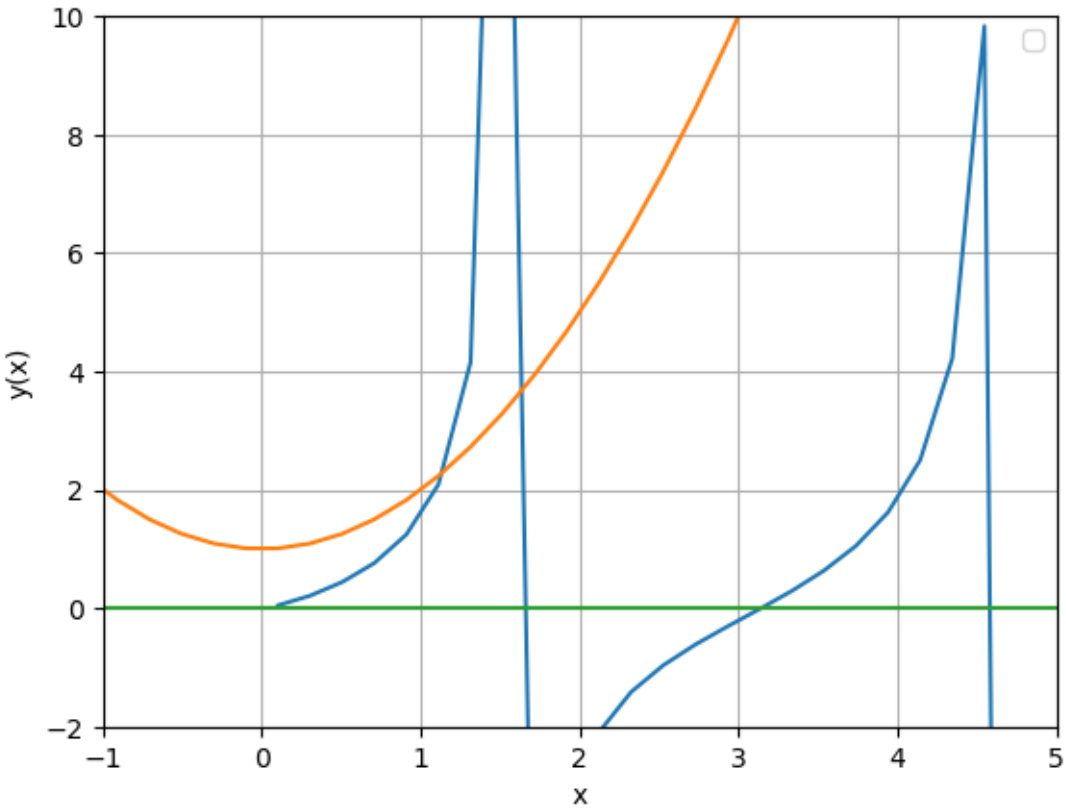
[2]:
```
def f(x):
    return x**(1/3) * math.tan(x) - x**2 - 1
```

Сперва изобразим графически две следующие функции:

$$y_1(x) = \sqrt[3]{x}\tan x, \quad y_2(x) = x^2 + 1,$$

Причем предположим, что отрезка $[10; 10]$ для начала нам будет достаточно (выбор этого отрезка основан на интуитивных предположениях).

[3]:
```
def y_1(x):
    return x**(1/3) * np.tan(x)

def y_2(x):
    return x**2 + 1

x = np.linspace(-10, 10, 100)
y = np.linspace(-2, 10, 100)

fig, ax = plt.subplots()

ax.plot(x, y_1(x))
ax.plot(x, y_2(x))
ax.plot(y, 0*x)

ax.set_xlim(-1, 5)
ax.set_ylim(-2, 10)

ax.set_xlabel('x')
ax.set_ylabel('y(x)')

plt.legend()
plt.grid()
plt.show()
```



Зная свойства параболы и тангенсоиды точно можно сказаать, что в данной плоскости два этих графика больше не пересекутся нигде, кроме двух точек, показанных на графике выше.

Выясним, на каких отрезках находятся данные точки. Для этого найдем производную от нашей функции $f(x)$:

$$f'(x) = \frac{\sin x}{3\sqrt[3]{x^2}\cos x} + \frac{\sqrt[3]{x}}{\cos^2 x} - 2x$$

[4]:
```
def f_derivative(x):
    return (np.sin(x) / (3 * np.cbrt(x**2) * np.cos(x))) + (np.cbrt(x) / np.
↪cos(x)**2) - 2*x
```

Рассмотрим отрезок $[1, 1.5]$

Проверим, меняет ли функция знак на концах отрезка.

[5]:
```
f(1)
```

[5]:
```
-0.4425922753450977
```

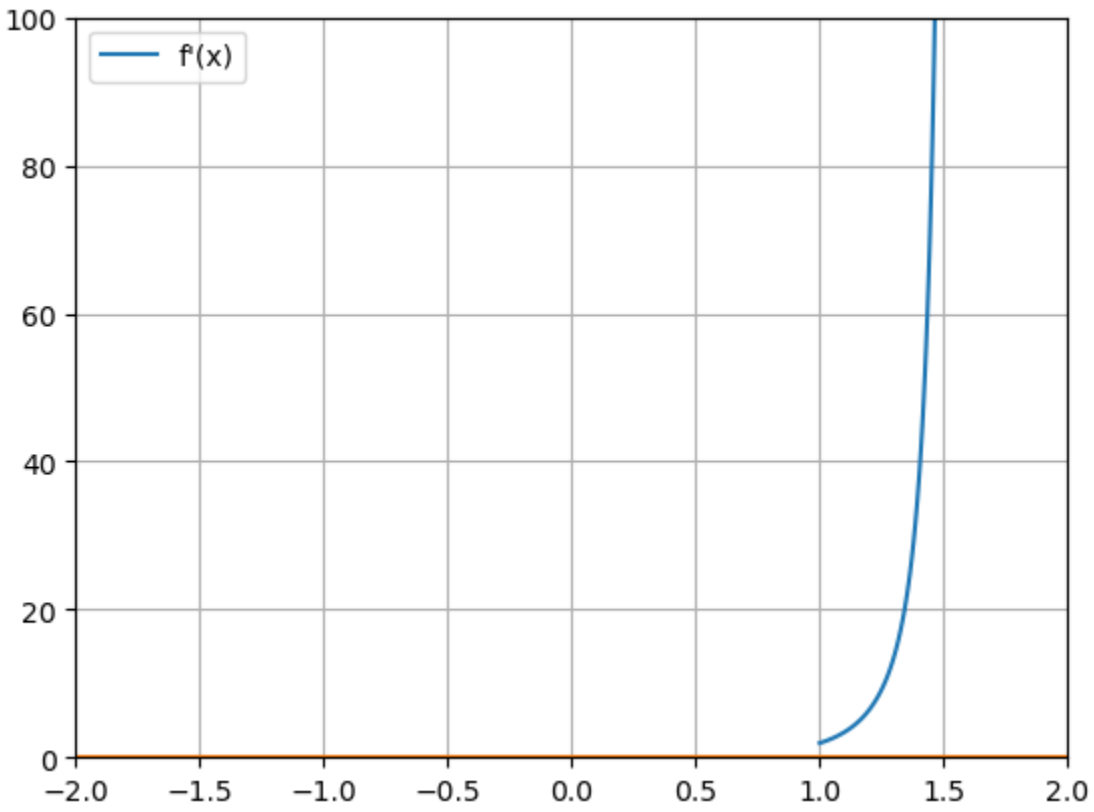
[6]:
```
f(1.5)
```

[6]: 12.89209625375312

Функция меняет знак, соответственно, на отрезке есть как минимум один корень.

Проверим функцию на монотонность. Для этого рассмотрим график производной на нашем отрезке.

[7]:
```
x = np.linspace(1, 1.5, 100)
y = np.linspace(-2, 2, 100)

fig, ax = plt.subplots()

ax.plot(x, f_derivative(x))
ax.plot(y, 0*x)

ax.set_xlim(-2, 2)
ax.set_ylim(-0, 100)

plt.legend()
plt.grid()
plt.show()
```



На графике видно, что первая производная не меняет знак, для закрепления этого факта, проверим значения производной на концах отрезка.

[8]:
```
f_derivative(1)
```

[8]:
```
1.9446547290330596
```

[9]:
```
f_derivative(1.5)
```

[9]:
```
229.35832484522751
```

Производная принимает одинаковые по знаку значения на концах, причем эти значения положительны. Рассмотрим максимум и минимум производной на отрезке.

[10]:
```
m = np.min(np.absolute(f_derivative(x)))
M = np.max(np.absolute(f_derivative(x)))

print(f'Минимум производной на отрезке: {m}')
print(f'Максимум производной на отрезке: {M}')
```

```
Минимум производной на отрезке: 1.9446547290330596
Максимум производной на отрезке: 229.35832484522751
```

И минимум, и максимум первой производной имеют положительные знаки, что говорит о том, что она не меняет знак, соответственно, функция монотонно возрастает, и, из условий теоремы, на отрезке у нас только один корень.

Рассмотрим следующий подозрительный отрезок $[1.5, 1.75]$**.**

[11]:
```
f(1.5)
```

[11]:
```
12.89209625375312
```

[12]:
```
f(1.75)
```

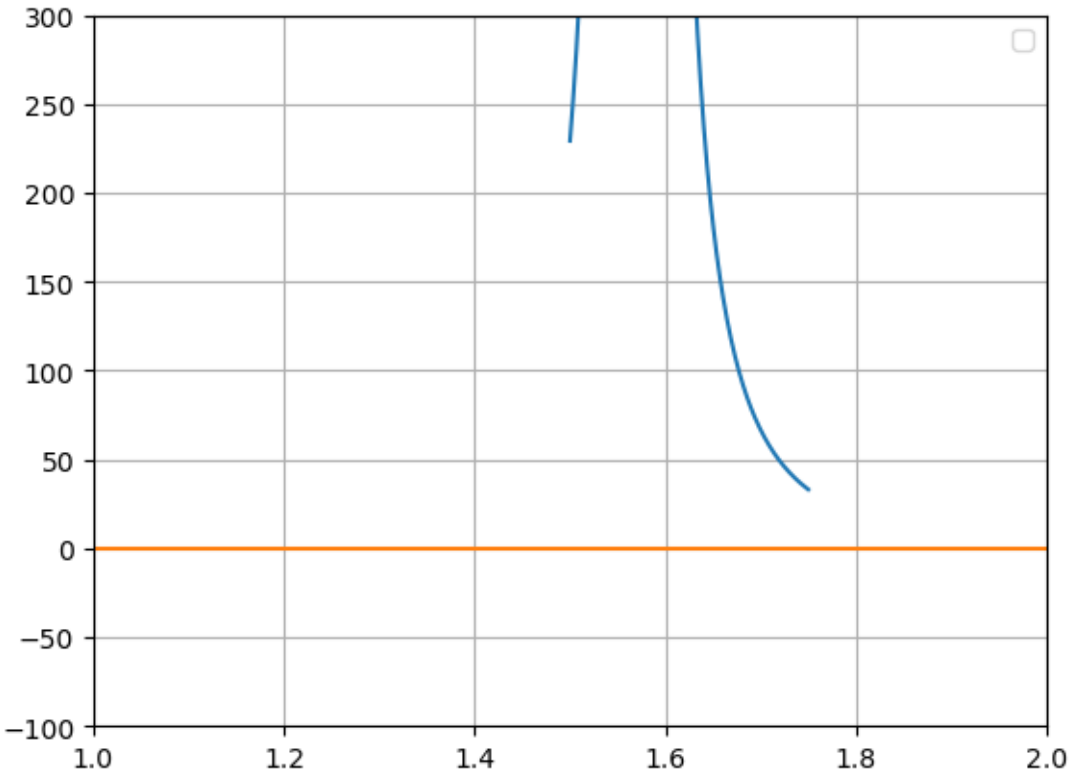
[12]:
```
-10.71495048277206
```

На левом конце отрезка функция положительна, на правом - отрицательна, что говорит о том, что на отрезке есть по крайней мере один корень.

Посмотрим на поведение первой производной на отрезке.

[13]:
```
x = np.linspace(1.5, 1.75, 300)
y = np.linspace(1, 2, 300)

fig, ax = plt.subplots()

ax.plot(x, f_derivative(x))
ax.plot(y, 0*x)
```

```
ax.set_xlim(1, 2)
ax.set_ylim(-100, 300)

plt.legend()
plt.grid()
plt.show()
```



На графике видно, что первая производная не меняет знак, для закрепления этого факта, проверим значения производной на концах отрезка.

[14]:
```
f_derivative(1.5)
```

[14]:
```
229.35832484522751
```

[15]:
```
f_derivative(1.75)
```

[15]:
```
33.161991787380074
```

Производная принимает одинаковые по знаку значения на концах, причем эти значения положительны. Рассмотрим максимум и минимум производной на отрезке.

[16]:
```
m = np.min(np.absolute(f_derivative(x)))
M = np.max(np.absolute(f_derivative(x)))
```

```
print(f'Минимум производной на отрезке: {m}')
print(f'Максимум производной на отрезке: {M}')
```

```
Минимум производной на отрезке: 33.161991787380074
Максимум производной на отрезке: 15494085.316958774
```

И минимум, и максимум первой производной имеют положительные знаки, что говорит о том, что она не меняет знак, соответственно, функция монотонно возрастает, и, из условий теоремы, на отрезке у нас только один корень.

Метод Ньютона

Рассмотрим уравнение

$$f(x) = 0,$$

где $f(x)$ достаточно гладкая функция вещественного переменного. Предположим, что для точного решения x^* каким-либо образом задано начальное приближение x^0. Для построения метода рассмотрим погрешность $\epsilon_0 = x^* - x^0$. В предположении, что ϵ_0 достаточно малая по модулю величина, подставим в уравнение (1) x^* вместо x, тогда

$$f(x^0 + \epsilon_0) = 0.$$

Разложим это выражение в ряд Тейлора в окрестности точки x^0:

$$f(x^0 + \epsilon_0) = f(x^0) + \epsilon_0 f'(x^0) + O(\epsilon_0^2) = 0.$$

Теперь отбросим слагаемое $O(\epsilon_0^2)$ и получим в рамках отброшенной величины получим приближенное уравнение

$$f(x^0) + \epsilon_0 f'(x^0) \approx 0.$$

Решая это уравнение относительно ϵ_0, получим

$$\epsilon_0 \approx -\frac{f(x^0)}{f'(x_0)}.$$

Тогда выразим из $x^* = x^0 + \epsilon_0$ и учитывая, что равенство приближенное, получим

$$x^* \approx x^0 - \frac{f(x^0)}{f'(x_0)}.$$

В итоге, повторяя описанную процедуру, мы можем построить итерационную формулу, которая носит название **метода Ньютона**

$$x^{k+1} = x^k - \frac{f(x^k)}{f'(x^k)}, \quad k = 0, 1, \ldots; \quad x_0$$

Теорема 2 (о сходимости метода Ньютона) *Пусть выполняются следующие условия:*

1. *Функция $f(x)$ определена и дважды непрерывно дифференцируема на отрезке*

$$s_0 = [x^0; x^0 + 2h_0], \quad h_0 = -\frac{f(x^0)}{f'(x^0)}.$$

При этом на концах отрезка $f(x)f'(x) \neq 0$.

2. *Для начального приближения x^0 выполняется неравенство*

$$2|h_0|M \le |f'(x_0)|, \quad M = \max_{x \in s_0} |f''(x)|.$$

Тогда справедливы следующие утверждения:

1. *Внутри отрезка s_0 уравнение $f(x) = 0$ имеет корень x^* и при этом этот корень единственный.*
2. *Последовательность приближений x^k, $k = 1, 2, \ldots$ может быть построена по формуле (2) с заданным приближением x^0.*
3. *Последовательность x^k сходится к корню x^*, то есть $x^k \xrightarrow[k \to \infty]{} x^*$.*
4. *Скорость сходимости характеризуется неравенством*

$$|x^* - x^{k+1}| \le |x^{k+1} - x^k| \le \frac{M}{2|f'(x^*)|} \cdot (x^k - x^{k-1})^2, \quad k = 0, 1, 2, \ldots$$

Рассмотрим теорему о сходимости метода Ньютона для нашего случая.

[13]:
```
x0 = 1.13
print(f'Начальное приближение: {x0}')

h0 = -f(x0) / f_derivative(x0)
print(f'h_0: {h0}')

s0 = np.linspace(x0, x0 + 2 * h0, 1000)
print('s_0 = [', s0[0], ';', s0[-1], ']')
```

```
Начальное приближение: 1.13
h_0: 0.01677822950564221
s_0 = [ 1.13 ; 1.1635564590112843 ]
```

[14]:
```
f(s0[0]) * f_derivative(s0[0])
```

[14]:
```
-0.28384050619426837
```

[15]:
```
f(s0[-1]) * f_derivative(s0[-1])
```

[15]:
```
0.4288433351315808
```

Функция на концах отрезка не обращается в ноль.

Вычислим вторую производную для функции $f(x)$:

$$f''(x) = \frac{3\,x\,\cos(x)\,\sin^2(x) + \left(18\,x^2 - 2\,\cos^2(x)\right)\,\sin(x) + 3\,x\,\cos^3(x) + 3\,x\,\cos(x)}{9\,x^{\frac{5}{3}}\,\cos^3(x)} - 2$$

[16]:
```
def f_second_derivative(x):
```

```
    numerator = 3*x*np.cos(x)*np.sin(x)**2 + (18*x**2 - 2*np.cos(x)**2)*np.
↪sin(x) + 3*x*np.cos(x)**3 + 3*x*np.cos(x)
    denominator = 9*x**(5/3) * np.cos(x)**3
    return numerator/denominator - 2
```

[17]:
```
M = np.max(np.absolute(f_second_derivative(s0)))
print(f'M: {M}')

2 * np.absolute(h0) * M <= np.absolute(f_derivative(x0))
```

```
M: 32.52740213380584
```

[17]:
```
True
```

Оба условия сходимости выполняются при начальном приближении $x_0 = 1.2$.

[18]:
```
def newton_method(x0, epsilon, max_iterations):
    x_prev = x0
    x_next = x_prev - f(x_prev) / f_derivative(x_prev)
    iterations = 1

    x_k = []

    while abs(x_next - x_prev) > epsilon and iterations < max_iterations:
        x_k.append([x_next, abs(x_next - x_prev)])

        x_prev = x_next
        x_next = x_prev - f(x_prev) / f_derivative(x_prev)
        iterations += 1

    if iterations == max_iterations:
        print("Максимальное количество итераций достигнуто!")

    x_k.append([x_next, abs(x_next - x_prev)])
    return x_k

x0 = 1.2  # Начальное приближение
epsilon = 1e-6  # Точность
max_iterations = 100  # Максимальное количество итераций

root = newton_method(x0, epsilon, max_iterations)
print("Корень:", root[len(root) - 1][0])

df = pd.DataFrame(root, columns=[('Метод Ньютона', 'Решение'), ('Метод Ньютона',␣
↪'Погрешность')])
```

```
Корень: 1.1459643670646564
```

Метод простой итерации

Применение метода требует предварительного приведения нашего уравнения

$$f(x) = 0,$$

к каноническому виду

$$x = \varphi(x).$$

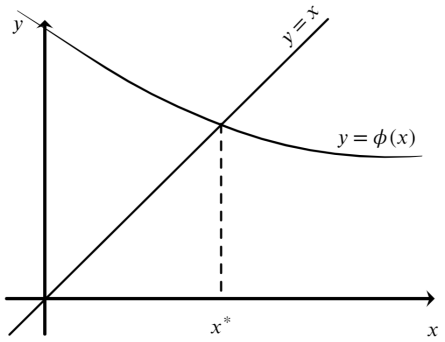
где $\varphi(x)$ — это заданная функция. Метод простой итерации будет иметь следующий вид:

$$x^{k+1} = \varphi(x^k),\ k = 0, 1, 2, \ldots,$$

где x^k — последовательность, начинающаяся с x^0, которая должна сходится к точному решению. Область изменения аргумента x на числовой оси обозначим через X, а через Y обозначим область значений функции $y = \varphi(x)$. Тогда функцию $\varphi(x)$ можно рассматривать как оператор, преобразующий X в Y:

$$\varphi : X \to Y.$$

Таким образом, нам нужно найти такие точки области X, которые при преобразовании оператором φ переходят сами в себя, то есть точки остающиеся неподвижными при преобразовании X в Y. Значит решения уравнения (1) — это точки, остающиеся неподвижными при преобразовании X в Y. Геометрически это можно изобразить следующим образом:



Таким образом, после процедуры отделения корней мы находим начальное приближение x^0 в окрестности корня x^*. И по найденному начальному приближению строится итерационная последовательность, которая и называется **методом простой итерации.** Мы должны обеспечить сходимость этого итерационного процесса. Сформулируем для этого теорему.

Теорема 3 (о сходимости метода простой итерации) *Пусть выполняются следующие условия:*

1. *функция $\varphi(x)$ определена на отрезке*

$$|x - x^0| \le \delta,$$

непрерывна на нем и удовлетворяет условию Липшица с постоянным коэффициентом меньше единицы, то есть $\forall x, \widetilde{x}$

$$|\varphi(x) - \varphi(\widetilde{x})| \le q|x - \widetilde{x}|, \quad 0 \le q < 1;$$

2. *для начального приближения x^0 верно неравенство*

$$|x^0 - \varphi(x^0)| \le m;$$

3. *числа* δ, q, m *удовлетворяют условию*

$$\frac{m}{1-q} \leq \delta.$$

Тогда

1. *уравнение* (1) *в области* (3) *имеет решение;*
2. *последовательность* x^k *построенная по правилу* (2) *принадлежит отрезку* $[x^0-\delta, x^0+\delta]$, *является сходящейся и ее предел удовлетворяет уравнению* (1)*:*

$$x^k \xrightarrow[k\to\infty]{} x^*;$$

3. *скорость сходимости* x^k *к* x^* *оценивается неравенством*

$$|x^* - x^k| \leq \frac{m}{1-q} q^k, \ k = 1, 2, \ldots$$

Также эта теорема может называется ***методом сжимающих отображений****.*

Замечание.

1. Так как сходимость метода простых итераций возможна при сжимающем отображении, то условие $|\varphi'(x)| < 1$ является определяющим при приведении исходного уравнения к каноническому виду. \\ Наиболее универсальными способом приведения к каноническому виду является преобразование

$$x = \underbrace{x + f(x)}_{\varphi(x)},$$

но нам необходимо выполнение условия $|\varphi'(x)| < 1$. Поэтому мы вводим параметр $\psi(x)$, выбираемый таким образом, чтобы обеспечить сходимость:

$$x = \underbrace{x + \psi(x) f(x)}_{\varphi(x)},$$

Параметр $\psi(x)$ должен быть непрерывным и $\psi(x^*) \neq 0$. Самый простой вариант — взять постоянную функцию $\psi(x) = \text{const}$ и подобрать эту константу из условия $|\varphi'(x)| < 1$.

Рассмотрим условие

$$|\varphi'(x)| < 1.$$

Приведение к каноническому виду выполним следующим образом:

$$x = x + \psi(x) f(x) = \varphi(x),$$

где $\psi(x)$ проще всего принять за const, которую подберем из неравенства

$$|\varphi'(x)| < 1,$$

откуда

$$-1 < \varphi'(x) < 1,$$
$$-1 < 1 + \psi(x) f'(x) < 1,$$

$$-\frac{2}{f(x)} < \psi(x) < 0.$$

Пусть $M = max(f'(x))$, тогда

$$\frac{2}{M} < \psi(x) < 0.$$

Найдем M на отрезке [0, 1.5]. Так как раньше мы выяснили, что производная возрастает на этом отрезке, то нам достаточно найти значение производной на правом конце отрезка.

[19]:
```
f_derivative(1.5)

M
```

[19]:
```
43.708114183396475
```

[20]:
```
-2/f_derivative(1.5)
```

[20]:
```
-0.008719979976090307
```

Получаем, что $\psi(x) \in [-0.008719979976090307, 0]$. Пусть $\psi(x) = -0.008$, тогда

$$\varphi(x) = x - 0.008f(x).$$

Возьмем $x_0 = 0.75$, тогда $\delta = 0.75$.

Найдем x_1.

[21]:
```
def phi(x):
    return x - 0.008*f(x)

x_0 = 0.75

x_1 = phi(x_0)

x_1
```

[21]:
```
0.7557287075537045
```

Теперь найдем m из неравенства

$$|x_0 - x_1| \leq m.$$

[22]:
```
m = abs(x_0 - x_1)

m
```

[22]:
```
0.005728707553704471
```

Найдем коэффициент Липшица из неравенства.

[23]:
```
q = 1 - f_derivative(1)

q
```

[23]: -0.9446547290330596

Проверим выполнение последнего условия

$$\frac{m}{1-q} < \delta.$$

[24]:
```
delta = 0.75

m/(1-q) < delta
```

[24]: True

Так как условия теоремы выполняются, то можно сделать вывод, что метод простой итерации на отрезке $[1, 1.5]$ сходится. Реализуем метод простой итерации программно.

[25]:
```
def simple_iteration_method(x0, epsilon, max_iterations):
    x_prev = x0
    x_next = phi(x_prev)
    iterations = 1

    x_k = []

    while abs(x_next - x_prev) > epsilon and iterations < max_iterations:
        x_k.append([x_next, abs(x_next - x_prev)])

        x_prev = x_next
        x_next = phi(x_prev)
        iterations += 1

    if iterations == max_iterations:
        print("Максимальное количество итераций достигнуто!")

    x_k.append([x_next, abs(x_next - x_prev)])
    return x_k

epsilon = 1e-6
max_iterations = 1000

root = simple_iteration_method(x_0, epsilon, max_iterations)
print("Корень:", root[len(root) - 1][0])
```

```
df = pd.concat([df, pd.DataFrame(root, columns=[('Метод простой итерации',␣
↪'Решение'), ('Метод простой итерации', 'Погрешность')])], axis=1,␣
↪verify_integrity=True).fillna('')
```

```
Корень: 1.1459385397465647
```

Метод Стеффенсена

Метод Стеффенсена основывается на том, что мы укажем способ вычисления x^{k+1} через x^k таким образом, чтобы обеспечить квадратичную скорость сходимости. Для увеличения скорости сходимости в данном методе используется преобразование Эйткена.

Пусть мы имеем x^0. Берем приближения

$$x^1 = \varphi(x^0), \quad x^2 = \varphi(x^1) = \varphi(\varphi(x^0)).$$

Тогда, используя формулу (7), мы можем при $n = 1$ получить

$$\sigma_1 = \frac{x^0 x^2 - (x^1)^2}{x^2 - 2x^1 + x^0} = \frac{x^0\varphi(\varphi(x^0)) - (\varphi(x^0))^2}{\varphi(\varphi(x^0)) - 2\varphi(x^0) + x^0}.$$

Заменим в этой формуле соответствующим образом индексы. В итоге получается итерационная формула, которая получила название **метод Стеффенсена**

$$x^{k+1} = \frac{x^k\varphi(\varphi(x^k)) - (\varphi(x^k))^2}{\varphi(\varphi(x^k)) - 2\varphi(x^k) + x^k}, \ k = 0, 1, \ldots; \ x^0.$$

Метод Стеффенсена можно трактовать как метод простой итерации примененный к уравнению вида

$$x = \Phi(x),$$

где

$$\Phi(x) = \frac{x\varphi(\varphi(x)) - \varphi^2(x)}{\varphi(\varphi(x)) - 2\varphi(x) + x}.$$

Возникает вопрос сходимости метода. Можно доказать, что функция $\Phi(x)$ вместе со своей производной будет непрерывна в окрестности точки x^*, причем

$$\lim_{x \to x^*} \Phi(x) = x^*.$$

Если предположить, что функция $\Phi(x^*) = x^*$ (то есть мы доопределяем ее), то $\Phi(x)$ будет непрерывна в точке x^*. Кроме того можно утверждать, что

$$\Phi'(x^*) = \lim_{x \to x^*} \frac{\Phi(x) - \Phi(x^*)}{x - x^*} = 0.$$

Таким образом, можно утверждать, что сходимость метода Стеффенсена будет квадратичной. То есть мы построили метод, обладающий повышенной скоростью сходимости. Но в 2 раза увеличивается объем вычислений, из-за того, что нужно вычислять функцию $\varphi(\varphi(x))$.

Так как метод Стеффенсена является модификацией метода простой итерации, то сходимость этого метода идентична сходимости метода простой итерации.

Построим функцию $\Phi(x)$.

[26]:
```
def Phi(x):
    return (x * phi(phi(x)) - (phi(x))**2) / (phi(phi(x)) - 2 * phi(x) + x)
```

Исходный итерационный процесс сходился, поэтому и новый построенный итерационный процесс будет также сходящимся. Поэтому перейдем сразу к программной реализации итерационного процесса.

[27]:
```
def steffensen_method(x0, epsilon, max_iterations):
    x_prev = x0
    x_next = Phi(x0)
    iterations = 1

    x_k = []

    while iterations < max_iterations:
        x_prev = x_next
        x_next = Phi(x_prev)

        if abs(x_next - x_prev) < epsilon:
            break

        x_k.append([x_next, abs(x_next - x_prev)])
        iterations += 1

    if iterations == max_iterations:
        print("Максимальное количество итераций достигнуто!")

    x_k.append([x_next, abs(x_next - x_prev)])

    return x_k

epsilon = 1e-6
max_iterations = 100

root = steffensen_method(x0, epsilon, max_iterations)
print("Корень:", root[len(root) - 1][0])

df = pd.concat([df, pd.DataFrame(root, columns=[('Метод Стеффенсена',␣
↪'Решение'), ('Метод Стеффенсена', 'Погрешность')])], axis=1,␣
↪verify_integrity=True).fillna('')
```

```
Корень: 1.145964910905985
```

Метод Чебышева третьего порядка

Идея метода базируется на способе построения итерационного процесса таким образом, чтобы обеспечить обращение в ноль производных от функции $\varphi(x)$ в точке x^*, то есть мы берем уравнение

$$x = \varphi(x)$$

и стараемся построить метод, у которого максимальное количество производных обращается в ноль в точке x^*. Для этого функцию $\varphi(x)$ запишем в виде

$$\varphi(x) = x + \psi_1(x)f(x) + \psi_2(x)f^2(x) + \ldots + \psi_{n-1}(x)f^{n-1}(x),$$

где $f(x)$ — это исходная функция, для которой мы ищем корни. Требуется выбрать функции $\psi_1(x), \ldots, \psi_{n-1}(x)$ так, чтобы

$$\varphi^{(j)}(x)\Big|_{f(x)=0} = 0, \quad j = 1, 2, \ldots, n-1$$

Рассмотрим условие на первую производную

$$\varphi'(x)\Big|_{f(x)=0} = 1 + \psi_1'(x)f(x) + \psi_1(x)f'(x) + \psi_2'(x)f^2(x) + 2\psi_2(x)f(x)f'(x) + \ldots\Big|_{f(x)=0} = \\ = 1 + \psi_1(x)f'(x)\Big|_{f(x)=0} = 0.$$

Аналогичным образом мы можем записать вторую производную

$$\varphi''(x)\Big|_{f(x)=0} = 2\psi_1'f'(x) + \psi_1(x)f''(x) + 2\psi_2(x)(f'(x))^2\Big|_{f(x)=0} = 0.$$

Из условия $\varphi'(x)\Big|_{f(x)=0} = 0$ следует, что функция

$$\psi_1(x) = -\frac{1}{f'(x)}.$$

Отсюда

$$\varphi(x) = x + \Big(-\frac{1}{f'(x)}\Big)f(x),$$

то есть мы пришли к методу Ньютона, итерационному процессу второго порядка. Из условия, что $\varphi''(x)\Big|_{f(x)=0} = 0$, применяя простые арифметические действия, мы можем получить

$$\psi_2(x) = -\frac{f''(x)}{2(f'(x))^3}.$$

Учитывая выражения для ψ_1 и ψ_2 мы модем построить итерационный процесс третьего порядка с кубической скоростью сходимости

$$x^{k+1} = x^k - \frac{f(x^k)}{f'(x^k)} - \frac{f^2(x^k)f''(x^k)}{2(f'(x^k))^3}$$

и будем называть эту формулу **методом Чебышева**. В этом методе мы также увеличиваем количество операций, так как необходимо вычислять значения $f(x), f'(x), f''(x)$.

Поскольку метод Чебышева является модификацией метода простой итерации, то его сходимость на отрезке $[1, 1.5]$ вытекает их сходимости метода простой итерации, начальное приближение мы также уже выбрали, поэтому сразу перейдем к программной реализации.

[28]:
```
def phi(x):
```

```
    return x - f(x) / f_derivative(x) - f(x)**2 * f_second_derivative(x) / (2 *␣
 ↪f_derivative(x)**3)

def chebyshev_method(x0, epsilon, max_iterations):
    x_prev = x0
    x_next = phi(x_prev)
    iterations = 1

    x_k = []

    while iterations < max_iterations:
        x_prev = x_next
        x_next = phi(x_prev)

        if abs(x_next - x_prev) < epsilon:
            break

        x_k.append([x_next, abs(x_next - x_prev)])
        iterations += 1

    if iterations == max_iterations:
        print("Максимальное количество итераций достигнуто!")

    x_k.append([x_next, abs(x_next - x_prev)])

    return x_k

def f(x):
    return x**(1/3)*math.tan(x) - x**2 - 1

epsilon = 1e-6
max_iterations = 100

root = chebyshev_method(x0, epsilon, max_iterations)
print("Корень:", root[len(root) - 1][0])

df = pd.concat([df, pd.DataFrame(root, columns=[('Метод Чебышева', 'Решение'),␣
 ↪('Метод Чебышева', 'Погрешность')])], axis=1, verify_integrity=True).fillna('')
```

```
Корень: 1.1459643670645736
```

Сравнительная характеристика методов

На протежении выполнения лабораторной работы заполнялся DataFrame для наглядности происходящего. Рассмотрим его.

[29]:
```
df
```

[29]:

	(Метод Ньютона, Решение)	(Метод Ньютона, Погрешность)	\
0	1.15454128	0.04545872	
1	1.14619221	0.00834908	
2	1.14596453	0.00022768	
3	1.14596437	0.00000016	
4			
..	...	...	
294			
295			
296			
297			
298			

	(Метод простой итерации, Решение)	(Метод простой итерации, Погрешность)	\
0	0.75572871	0.00572871	
1	0.76143083	0.00570212	
2	0.76710587	0.00567503	
3	0.77275329	0.00564742	
4	0.77837256	0.00561928	
..	...	...	
294	1.14593442	0.00000113	
295	1.14593551	0.00000109	
296	1.14593656	0.00000105	
297	1.14593757	0.00000101	
298	1.14593854	0.00000097	

	(Метод Стеффенсена, Решение)	(Метод Стеффенсена, Погрешность)	\
0	1.14616572	0.00801378	
1	1.14596449	0.00020123	
2	1.14596343	0.00000106	
3	1.14596445	0.00000102	
4	1.14596491	0.00000046	
..	...	...	
294			
295			
296			
297			
298			

	(Метод Чебышева, Решение)	(Метод Чебышева, Погрешность)
0	1.14596441	0.00157775
1	1.14596437	0.00000005
2		
3		
4		
..	...	...
294		

```
295
296
297
298

[299 rows x 8 columns]
```

Видно, что каждый метод получил решение с нужной нам точностью до шести знаков. Однако методу простой итерации понадобилось 299 итераций, что гораздо медленней, чем у остальных методов, что логично, так как он является наиболее простым из итерационных и обладает линейно скоростью сходимости. Наиболее быстрым оказался метод Чебышева, что было очевидно, так как он имеет кубическую скорость сходимости, который справился всего за 2 итерации, он же оказался и самым точным. На втором месте по точности стал метод Ньютона, имеющий квадратичную скорость сходимости. На третьем месте стал метод Стеффенсена, который также обладает квадратичной скоростью сходимости. То, что метод Стеффенсена оказался немного медленней можно объяснить особенностями нашей задачи.

Задача 2

Найти с точностью $\epsilon = 10^{-6}$ наибольший по модулю корень уравнения

$$\sum_{i=0}^{n} a_i x^i = 0,$$

где векто коэффицирента a есть решение системы линейных алгебраических уравнений $Aa = f$ с

$$A = \begin{bmatrix} 15 & -2 & 3.5 & 1 & -0.1 \\ 1 & -8 & -3.1 & 1 & 2.3 \\ -1 & 3 & 30 & 2 & 4.6 \\ 0 & 0.1 & 2 & 15 & 2 \\ 1 & -2 & 0.4 & 3.2 & -17 \end{bmatrix}, \quad f = \begin{bmatrix} 1 \\ 0 \\ 20 \\ -3 \\ 5 \end{bmatrix}$$

Первым делом найдем решение системы $Aa = f$. Сделаем это одним из реализованных в прошлом методом, а именно методом релаксации.

[30]:
```
A = np.array([[15, -2, 3.5, 1, -0.1],
              [1, -8, -3.1, 1, 2.3],
              [-1, 3, 30, 2, 4.6],
              [0, 0.1, 2, 15, 2],
              [1, -2, 0.4, 3.2, -17]])

n = len(A[0])

b = np.array([1, 0, 20, -3, 5])

def sum1(A, x, i, n):
    s = 0

    for j in range(i):
```

```
        s += A[i, j] * x[j]

    return s

def sum2(A, x, i, n):
    s = 0

    for j in range(i + 1, n):
        s += A[i, j] * x[j]

    return s

def relaxation_method(A, b, n, epsilon = 1e-4, max_iterations = 300, w = 1.5):
    x = np.zeros((n, 1))

    for _ in range(max_iterations):
        x_new = np.zeros_like(x)

        for i in range(n):
            x_new[i] = (1 - w) * x[i] + (w / A[i, i]) * (b[i] - sum1(A, x_new,␣
 ↪i, n) - sum2(A, x, i, n))

        if np.max(np.abs(x_new - x)) < epsilon:
            return x_new

        x = x_new

    return x

a = relaxation_method(A, b, n)

print(a)
```

```
[[-0.15378659]
 [-0.4301327 ]
 [ 0.76546924]
 [-0.26136514]
 [-0.28375325]]
```

Получили коэффициенты для нашего многочлена. Определим его.

[31]:
```
def P(x):
    p = 0
    n = len(a)

    for i in range(n):
        p += x**i * float(a[i])
```

```
    return p
```

Задачу отыскания наибольшего по модулю корня уравнения можно поделить по тем же этапам, что и в первой задаче. То есть сначала нам необходимо решить задачу отделения корней, а затем поиск его приближения с заданной точностью.

Для того, чтобы понять, есть ли у нашего уравнения действительные корни, рассмотрим график функции $P(x)$.

[32]:
```
x = np.linspace(-10, 10, 1000)

fig, ax = plt.subplots()

ax.plot(x, P(x))
ax.plot(x, 0*x)

ax.set_xlim(-3, 3)
ax.set_ylim(-3, 3)

ax.set_xlabel('x')
ax.set_ylabel('y(x)')

plt.legend()
plt.grid()
plt.show()
```

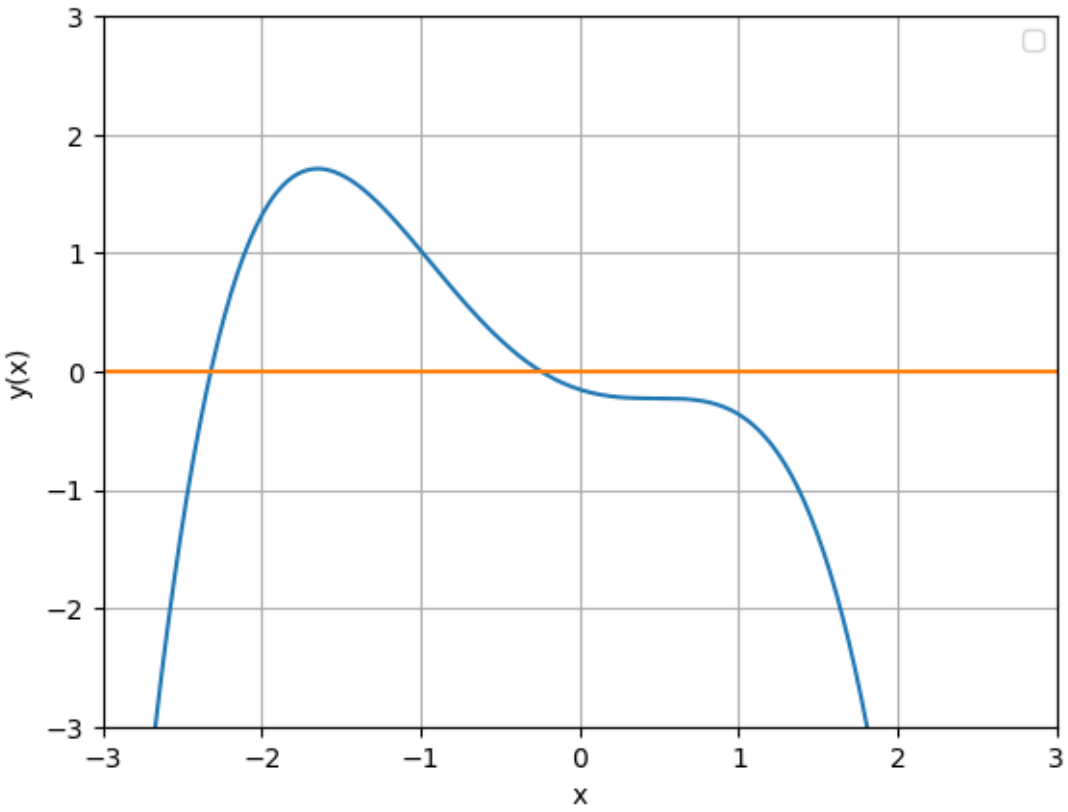


График нашей функции пересекает ось Ox в двух местах, что говорит о наличии как минимум двух корней. Наибольший по модулю из них находится на отрезке $[-2.5, -2]$. Докажем то, что на этом отрезке есть корень, и он является единственным.

Рассмотрим значение функции $P(x)$ на концах отрезка.

[33]:
```
P(-2.5)
```

[33]:
```
-1.294552946934635
```

[34]:
```
P(-2)
```

[34]:
```
1.3192249598960295
```

Функция меняет знак, что говорит о том, что как минимум один корень на этом отрезке существует. Исследуем функцию на монотонность, для этого найдем ее производную

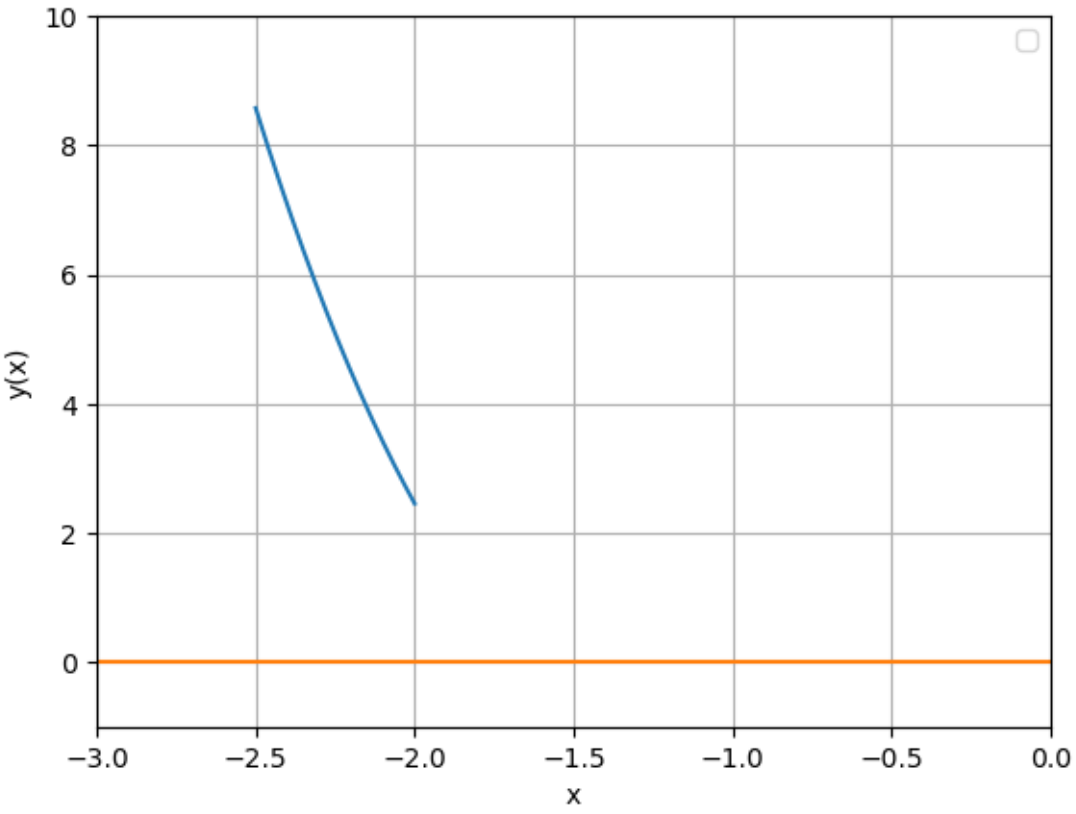
$$P'(x) = 4a_4x^3 + 3a_3x^3 + 2a_2x + a_1$$

[35]:
```
def P_derivative(x):
    p = 0
    n = len(a)
```

```
    for i in range(1, n):
        p += i * x**(i - 1) * float(a[i])

    return p
```

Рассмотрим график производной.

[36]:
```
x = np.linspace(-2.5, -2, 1000)
y = np.linspace(-3, 3, 1000)

fig, ax = plt.subplots()

ax.plot(x, P_derivative(x))
ax.plot(y, 0*x)

ax.set_xlim(-3, 0)
ax.set_ylim(-1, 10)

ax.set_xlabel('x')
ax.set_ylabel('y(x)')

plt.legend()
plt.grid()
plt.show()
```



На графике видно, что производная является отрицательной, что говорит о том, что сама функция монотонна на отрезке, что свидетельствует о наличии только одного корня на исследуемом отрезке.

Для отыскания приближения с заданной точностью. Воспользуемся стандартным методом Ньютона

$$x^{k+1} = x^2 - \frac{f(x^k)}{f'(x^k)}$$

Аналогично, рассмотрим теорему о сходимости.

[57]:
```
x0 = -2.3
print(f'Начальное приближение: {x0}')

h0 = -P(x0) / P_derivative(x0)
print(f'h_0: {h0}')

s0 = np.linspace(x0, x0 + 2 * h0, 1000)
print('s_0 = [', s0[0], ';', s0[-1], ']')
```

```
Начальное приближение: -2.3
h_0: -0.021766977923114692
s_0 = [ -2.3 ; -2.343533955846229 ]
```

[58]:
```
P(s0[0]) * P_derivative(s0[0])
```

[58]:
```
0.7098287060118967
```

[59]:
```
P(s0[-1]) * P_derivative(s0[-1])
```

[59]:
```
-0.8590677328935202
```

На концах отрезка функция не обращается в ноль.

Вычислим вторую производную для функции $P(x)$:

$$P''(x) = 12a_4x^2 + 6a_3x + 2a_2$$

[60]:
```
def P_second_derivative(x):
    p = 0
    n = len(a)

    for i in range(2, n):
        p += i * (i - 1) * x**(i - 2) * float(a[i])

    return p
```

[61]:
```
M = np.max(np.absolute(P_second_derivative(s0)))
M
```

[61]: 13.494942499874902

[62]:
```
2 * np.absolute(h0) * M <= np.absolute(P_derivative(x0))
```

[62]: True

Оба условия сходимости выполняются при начальном приближении. $x_0 = -2.3$.

[63]:
```
def newton_method(x0, epsilon, max_iterations):
    x_prev = x0
    x_next = x_prev - P(x_prev) / P_derivative(x_prev)
    iterations = 1

    x_k = []

    while abs(x_next - x_prev) > epsilon and iterations < max_iterations:
        x_k.append([x_next, abs(x_next - x_prev)])

        x_prev = x_next
        x_next = x_prev - P(x_prev) / P_derivative(x_prev)
        iterations += 1

    if iterations == max_iterations:
        print("Максимальное количество итераций достигнуто!")

    x_k.append([x_next, abs(x_next - x_prev)])
    return x_k

x0 = -2.3  # Начальное приближение
epsilon = 1e-6  # Точность
max_iterations = 100  # Максимальное количество итераций

root = newton_method(x0, epsilon, max_iterations)
print("Корень:", root[len(root) - 1][0])

df = pd.DataFrame(root, columns=[('Метод Ньютона', 'Решение'), ('Метод Ньютона',␣
↪'Погрешность')])
df
```

Корень: -2.3212537947949174

[63]:

	(Метод Ньютона, Решение)	(Метод Ньютона, Погрешность)
0	-2.32176698	0.02176698
1	-2.32125408	0.00051289
2	-2.32125379	0.00000029

[64]:
```
P(root[len(root) - 1][0])
```

[64]: -5.524469770534779e-13

Для того, чтобы точно убедиться, что получившийся корень является наибольшим по модулю, рассмотрим и второй корень. Он находится на отрезке $[-0.5, 0]$. Проведем те же действия для его нахождения. Рассмотрим значения функции $P(x)$ на концах отрезка.

[45]:
```
P(-0.5)
```

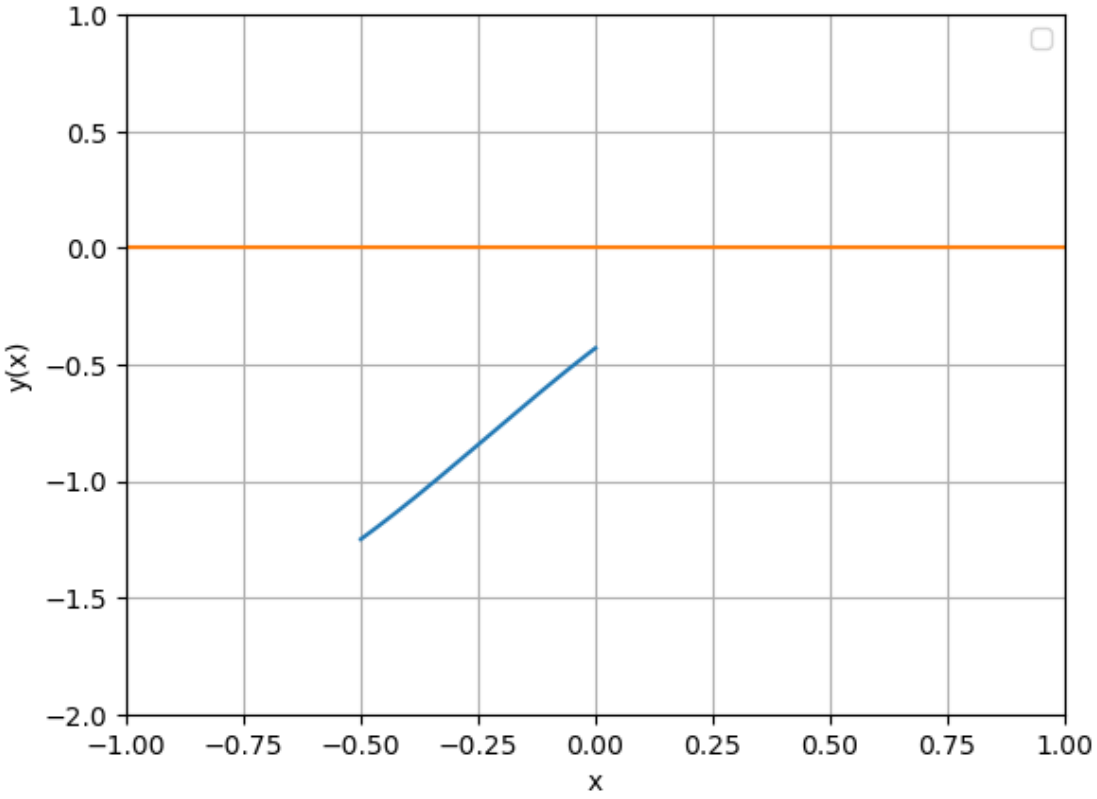
[45]: 0.2675831407406165

[46]:
```
P(0)
```

[46]: -0.153786586002759

Функция меняет знак, следовательно, как минимум один корень на отрезке есть. Рассмотрим, является ли он единственным. Для этого изобразим график производной.

[47]:
```
x = np.linspace(-0.5, 0, 1000)
y = np.linspace(-1, 1, 1000)

fig, ax = plt.subplots()

ax.plot(x, P_derivative(x))
ax.plot(y, 0*x)

ax.set_xlim(-1, 1)
ax.set_ylim(-2, 1)

ax.set_xlabel('x')
ax.set_ylabel('y(x)')

plt.legend()
plt.grid()
plt.show()
```



Видно, что производная функции не меняет знак, следовательно, функция монотонна, и на отрезке только один корень. Найдем его приближение методом Ньютона. Рассмотрим сначала сходимость.

[48]:
```
x0 = -0.3
print(f'Начальное приближение: {x0}')

h0 = -P(x0) / P_derivative(x0)
print(f'h_0: {h0}')

s0 = np.linspace(x0, x0 + 2 * h0, 1000)
print('s_0 = [', s0[0], ';', s0[-1], ']')
```

```
Начальное приближение: -0.3
h_0: 0.052622341386282814
s_0 = [ -0.3 ; -0.19475531722743436 ]
```

[49]:
```
P(s0[0]) * P_derivative(s0[0])
```

[49]: -0.04544824085396322

[50]:
```
P(s0[-1]) * P_derivative(s0[-1])
```

[50]: 0.029580709852124895

[51]:
```
M = np.max(np.absolute(P_second_derivative(s0)))
M
```

[51]:
```
1.711496067244416
```

[52]:
```
2 * np.absolute(h0) * M <= np.absolute(P_derivative(x0))
```

[52]:
```
True
```

Оба условия сходимости выполняются при начальном приближении. $x_0 = -0.3$.

[67]:
```
def newton_method(x0, epsilon, max_iterations):
    x_prev = x0
    x_next = x_prev - P(x_prev) / P_derivative(x_prev)
    iterations = 1

    x_k = []

    while abs(x_next - x_prev) > epsilon and iterations < max_iterations:
        x_k.append([x_next, abs(x_next - x_prev)])

        x_prev = x_next
        x_next = x_prev - P(x_prev) / P_derivative(x_prev)
        iterations += 1

    if iterations == max_iterations:
        print("Максимальное количество итераций достигнуто!")

    x_k.append([x_next, abs(x_next - x_prev)])
    return x_k

x0 = -0.3  # Начальное приближение
epsilon = 1e-6  # Точность
max_iterations = 100  # Максимальное количество итераций

root = newton_method(x0, epsilon, max_iterations)
print("Корень:", root[len(root) - 1][0])

df = pd.DataFrame(root, columns=[('Метод Ньютона', 'Решение'), ('Метод Ньютона',␣
↪'Погрешность')])
df
```

```
Корень: -0.24456355680256817
```

[67]:

	(Метод Ньютона, Решение)	(Метод Ньютона, Погрешность)
0	-0.24737766	0.05262234
1	-0.24457162	0.00280604
2	-0.24456356	0.00000807

```
3               -0.24456356                    0.00000000
```

[66]:
```
P(root[len(root) - 1][0])
```

[66]:
```
1.6696713456276768e-17
```

Получили значение $1.6696713456276768e-17$, что явно меньше по модулю, чем $-5.524469770534779e-13$, то есть изначательно предположение о нахождении максимального по модулю корня на отрезке $[-2.5, -2]$ было верным. И искомое приближение корня имеет вид

$$x = -2.3212537947949174$$

И этот корень был найден за 3 итерации.