Университет ИТМО
МФ КТиУ, Ф ПИиКТ

Лабораторная работа №2
Дисциплина «Вычислительная математика»

Численное решение нелинейных уравнений и систем

Выполнил
Аскаров Эмиль Рамилевич

Преподаватель:
Машина Екатерина
Алексеевна

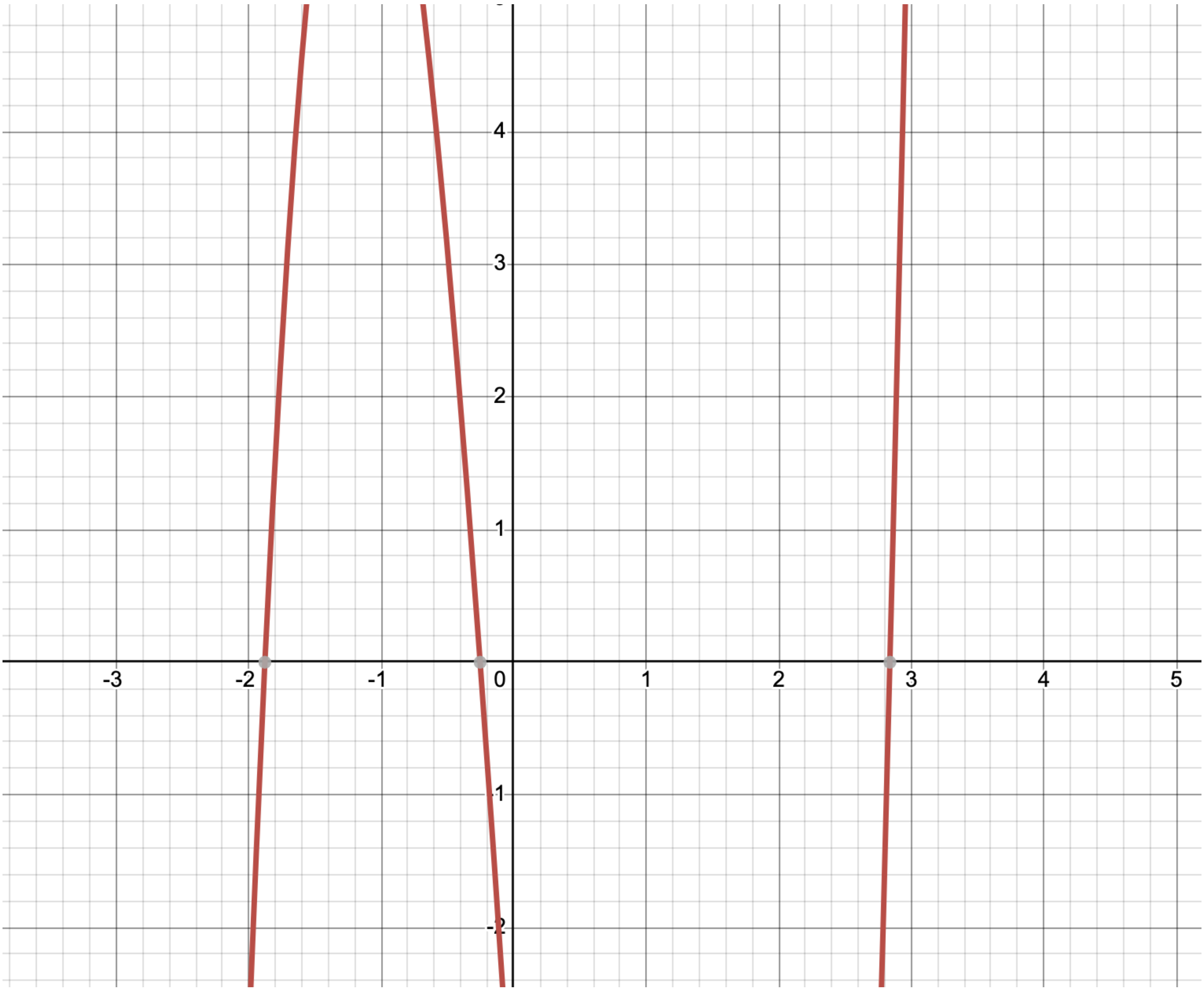
г. Санкт-Петербург
2024 г.

Решение нелинейного уравнения

Вид уравнения:

$$2.74x^3 - 1.93x^2 - 15.28x - 3.72 = 0$$

1. Графическое отделение корней:



2. Интервалы изоляции корней:

Для левого корня: [-2; -1.8]

Для центрального корня: [-0.4; -0.2]

Для правого корня: [2.8; 3]

3. Уточнение корней с точностью eps = 10^{-2}:

Левый корень – 4 (метод секущих)

Центральный корень – 5 (метод простой итерации)

Правый корень – 3 (метод Ньютона)

Уточнение левого корня (метод секущих):

№ итерации	x_{k-1}	x_k	x_{k+1}	$f(x_{k+1})$	$\lvert x_{k+1} - x_k\rvert$
0	-2	-1.98	-1.888	-0.186	0.092>eps
1	-1.98	-1.888	-1.88	-0.014	0.008<eps

Уточнение правого корня (метод Ньютона):

$$f(x) = 2.74x^3 - 1.93x^2 - 15.28x - 3.72$$
$$f'^{(x)} = 2.74 * 3 * x^2 - 1.93 * 2 * x \quad - 15.28$$
$$f''^{(x)} = 2.74 * 3 * 2 * x \quad - 1.93 * 2$$

Производные сохраняют знак на интервале изоляции, поэтому метод Ньютона эффективен.

Начальное приближение: $x_0 = 3$

f (3) = 7.05

f'' (3) = 45.46

Знаки функции и второй производной совпадают, поэтому это подходящее начальное приближение.

№ итерации	x_k	$f(x_k)$	$f'(x_k)$	x_{k+1}	$\lvert x_{k+1} - x_k\rvert$
0	3	7.050	47.120	3 – 7.05 / 47.12 = 2.850	0.150>eps
1	2.85	0.500	40.502	2.85 - 0.500 / 40.502 = 2.838	0.012>eps
2	2.838	0.003	39.973	2.838-0.003 / 39.973 = 2.838	8E-05<eps

Уточнение центрального корня (метод простой итерации):

$$f(x) = 2.74x^3 - 1.93x^2 - 15.28x - 3.72 = 0$$
$$phi(x) = x = \frac{2.74x^3 - 1.93x^2 - 3.72}{15.28}$$
$$phi'^{(x)} = 0.538x^2 - 0.253x$$

На отрезке [-0.4; -0.2] условие сходимости выполняется.

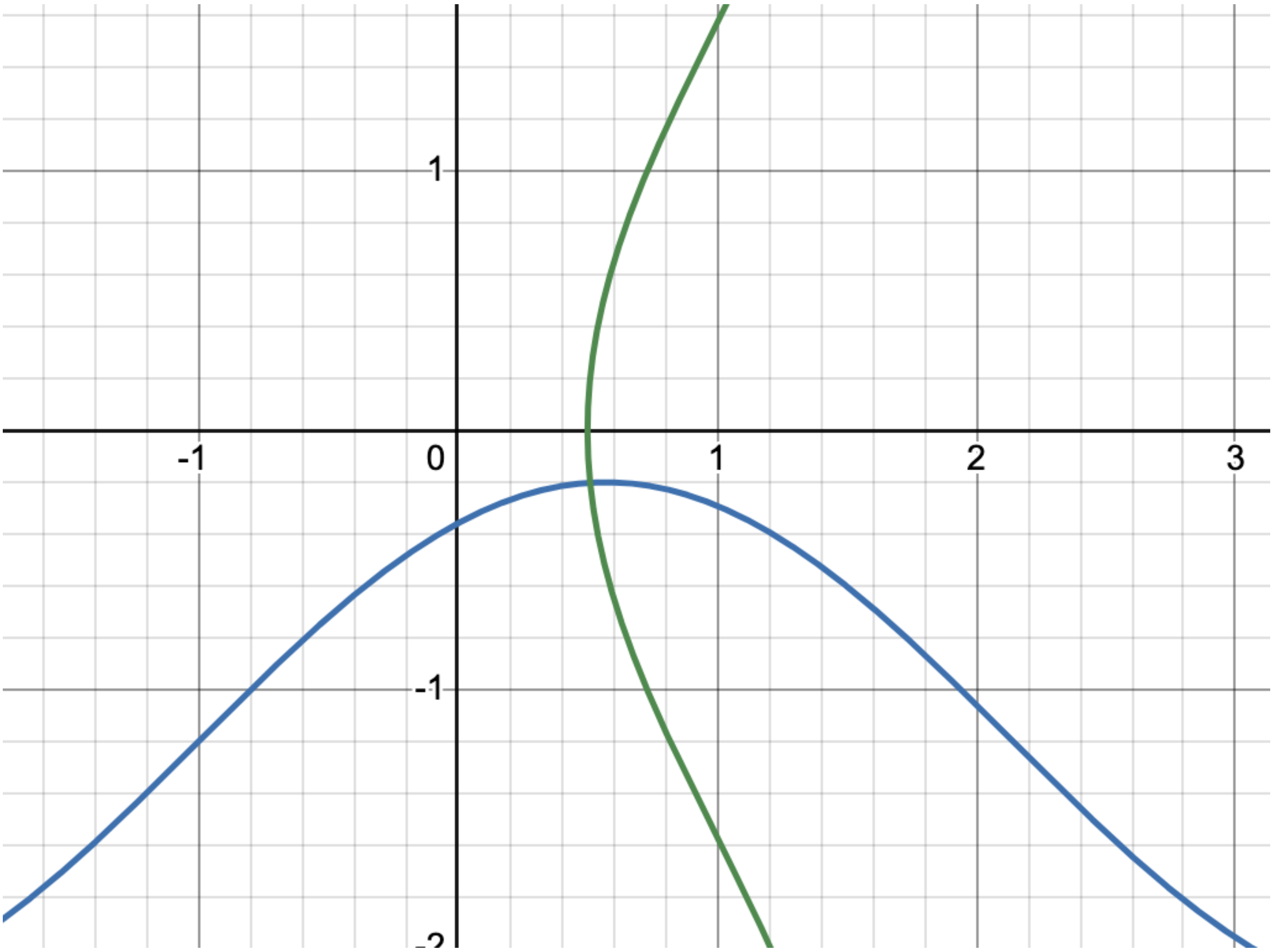
№ итерации	x_k	x_{k+1}	$f(x_{k+1})$	$\lvert x_{k+1} - x_k\rvert$
0	-0.4	-0.275	0.281	0.125>eps
1	-0.275	-0.257	0.03	0.018>eps
2	-0.257	**-0.255**	0.003	0.002<eps

Решение системы нелинейных уравнений

Вид системы:

$$\begin{cases} \sin(x+1) - y = 1.2 \\ 2x + \cos(y) = 2 \end{cases}$$

1. Графическое отделение корней:



Интервал изоляции x: [0.2; 0.6]

Интервал изоляции y: [-0.4; 0]

2. Решение системы с точностью eps = 10^{-2}:

$$\begin{cases} \sin(x+1) - y = 1.2 \\ 2x + \cos(y) = 2 \end{cases}$$

$$\begin{cases} y = \sin(x+1) - 1.2 \\ x = \dfrac{2-\cos(y)}{2} \end{cases}$$

Начальные приближения: x = 0.2, y = 0

$fi_x = \frac{\mathbf{2-\cos(y)}}{\mathbf{2}}$

$fi_y = \mathbf{\sin(x+1) - 1.2}$

$(fi_x)'_x = 0$ $(fi_x)'_y = 0.5\sin(y)$ $|(fi_x)'_x| + |(fi_x)'_y| = |0.5\sin(y)| < 1$

$(fi_y)'_x = \cos(x+1)$ $(fi_y)'_y = 0$ $|(fi_y)'_x| + |(fi_y)'_y| = |\cos(x+1)| < 1$

Следовательно метод сходится

№	X_k	Y_k	X_{k+1}	Y_{k+1}	$\mid x_{k+1} - x_k\mid$	$\mid y_{k+1} - y_k\mid$
0	0.2	0	1-0.5cos(0) = 0.5	sin(0.2+1) - 1.2 = -0.268	0.3>eps	0.268
1	0.5	-0.268	1-0.5cos(-0.268) = 0.517	sin(0.5+1) -1.2= -0.201	0.018	0.067
2	0.517	-0.201	1-0.5cos(-0.201) = 0.510	sin(0.517+1) - 1.2 = -0.202	0.008	0.000

Программная реализация нелинейных уравнений и систем

Методы: 3 (метод Ньютона), 4 (метод секущих), 5 (метод простой итерации).

```
import matplotlib.pyplot as plt
from math import sin, e, cos


def bisection_method(func, a, b, eps):
    while abs(a - b) > eps:
        x = (a + b) / 2
        if func(a) * func(x) > 0:
            a = x
        else:
            b = x
    return (a + b) / 2


# def secant_method(func, x0, eps):
#     x1 = x0 - func(x0) / get_derivative_at_point(func, x0)
#     while abs(x1 - x0) > eps:
#         x2 = x1 - (x1 - x0) * f(x1) / (f(x1) - f(x0))
#         x0, x1 = x1, x2
#     return x1


# def secant_method_manual(func, x0, eps):
#     x1 = x0 + eps * 2
#     # x1 = x0 - func(x0) / get_derivative_at_point(func, x0)
#     x2 = x1 - (x1 - x0) * f(x1) / (f(x1) - f(x0))
#     print(f"x0={x0}, x1={x1}, x2={x2}, f(x2)={f(x2)}, |x2-x1|={abs(x2 - x1)}")
#
#     while abs(x2 - x1) > eps:
#         x0, x1 = x1, x2
#         x2 = x1 - (x1 - x0) * f(x1) / (f(x1) - f(x0))
#         print(f"x0={x0}, x1={x1}, x2={x2}, f(x2)={f(x2)}, |x2-x1|={abs(x2-x1)}")
#     return x2


def newton_method(f, a, b, eps):
    if f(a) * second_derivative(f, a) > 0:
        x0 = a
    else:
        x0 = b
    # print(f"f(a)={f(a)}, f_2(a)={f_2(a)}, f(a)*f\"(a)={f(a) * f_2(a)}")
    # print(f"f(b)={f(b)}, f_2(b)={f_2(b)},  f(b)*f\"(b)={f(b) * f_2(b)}")

    x1 = x0 - f(x0) / derivative(f, x0)
    # print(f"xk={x0}, f(xk)={f(x0)}, f'(xk)={f_1(x0)}, xk+1={x1}, |xk+1-xk|={abs(x1 -
x0)}")
    while abs(x1 - x0) > eps:
        x0 = x1
        x1 = x0 - f(x0) / derivative(f, x0)
        # print(f"xk={x0}, f(xk)={f(x0)}, f'(xk)={f_1(x0)}, xk+1={x1}, |xk+1-xk|={abs(x1
- x0)}")
    return x1


def simple_iteration_method(func, a, b, eps):
    max_func_ = 0
```

```
    x = a
    while x < b:
        max_func_ = max(max_func_, abs(derivative(func, x)))
        x += eps
    if derivative(func, a) > 0:
        h = -1 / max_func_
    else:
        h = 1 / max_func_
    fi = lambda x: x + h * func(x)

    x0 = a
    x1 = fi(x0)
    while abs(x1 - x0) > eps:
        x1, x0 = fi(x1), x1
    return x1
# def simple_iteration_method_2(func, x0, a, b, eps):
#     max_func_ = 0
#     x = a
#     while x < b:
#         max_func_ = max(max_func_, abs(derivative(func, x)))
#         x += eps
#     if derivative(func, a) > 0:
#         h = -1 / max_func_
#     else:
#         h = 1 / max_func_
#     fi = lambda x: x + h * func(x)
#     fi_ = lambda x: 1 + h * derivative(func, x)
#
#     x = fi(x0)
#     while abs(x - x0) > eps:
#         x, x0 = fi(x), x
#     return x


# def simple_iteration_method_manual(f, x0, eps):
#     x1 = (2.74 * x0 ** 3 - 1.93 * x0 ** 2 - 3.72) / 15.28
#     print(f"x0={x0}, x1={x1}, f(x1)={f(x1)}, |x0-x1|={abs(x0 - x1)}")
#     while abs(x1 - x0) > eps:
#         x0 = x1
#         x1 = (2.74 * x0 ** 3 - 1.93 * x0 ** 2 - 3.72) / 15.28
#         print(f"x0={x0}, x1={x1}, f(x1)={f(x1)}, |x0-x1|={abs(x0 - x1)}")
#     return x1


def verify(func, a, b, eps=0.00001):
    # true - if there is only one root, false - else
    if func(a) * func(b) < 0 and derivative(func, a) * derivative(func, b) > 0:
        x = a
        while x < b:
            if derivative(func, a) * derivative(func, x) <= 0:
                return False
            x += eps
        return True
    return False


def derivative(func, x0, dx=0.000001):
    return (func(x0 + dx) - func(x0)) / dx


def second_derivative(func, x0, dx=0.000001):
    return ((func(x0 + 2 * dx) - func(x0 + dx)) / dx - (func(x0 + dx) - func(x0)) / dx) /
dx


def draw_plot(func, a, b, root, eps):
    xs = []
    ys = []
    x = a
    while x < b:
```

```
        xs.append(x)
        ys.append(func(x))
        x += eps
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.plot(xs, ys, 'g')
    #  plt.plot([root], [func(root)], 'r')
    plt.annotate('*', xy=(root, func(root)))
    plt.plot([a, b], [0, 0], 'b')
    plt.show()


# f = lambda x: 2.74 * x ** 3 - 1.93 * x ** 2 - 15.28 * x - 3.72
# f_1 = lambda x: 2.74 * 3 * x ** 2 - 1.93 * 2 * x - 15.28
# f_2 = lambda x: 2.74 * 3 * 2 * x - 1.93 * 2
# print(secant_method_manual(f, -2, 10**-2))
print('Выберите функцию:')
print('1. f(x) = 2.74 * x ** 3 - 1.93 * x ** 2 - 15.28 * x - 3.72')
print('2. f(x) = cos(x)')
print('3. f(x) = x ** 2 - 1')
case_number = input('Введите номер функции: ')
while case_number not in {'1', '2', '3'}:
    case_number = input('Введите номер функции: ')

case_number = int(case_number)
eps = 0.0001
if case_number == 1:
    f = lambda x: 2.74 * x ** 3 - 1.93 * x ** 2 - 15.28 * x - 3.72
    f_1 = lambda x: 2.74 * 3 * x ** 2 - 1.93 * 2 * x - 15.28
    f_2 = lambda x: 2.74 * 3 * 2 * x - 1.93 * 2
    isolation_intervals = [(-2, -1.8), (-0.4, -0.2), (2.8, 3)]
    a, b = isolation_intervals[2]
elif case_number == 2:
    f = lambda x: cos(x)
    a, b = -2, -1
elif case_number == 3:
    f = lambda x: x ** 2 - 1
    a, b = -2, 0
fl = input('Напишите "да", если хотите задать [a; b]: ')
if fl == 'да':
    while True:
        try:
            a, b = map(float, input('Введите a и b через пробел:').split())
            if not verify(f, a, b):
                print('Функция на данном отрезке не имеет корней или имеет множество
корней')
                continue
        except Exception:
            print('Неверный ввод. Повторите')
            continue
        break

x0 = (a + b) / 2
print("Метод деления пополам:", (root := bisection_method(f, a, b, eps)))
print("Метод Ньютона:", newton_method(f, a, b, eps))
print("Метод простой итерации:", simple_iteration_method(f, a, b, eps))
draw_plot(f, a - 2, b + 2, root, 0.0001)
```

Метод простой итерации для систем:

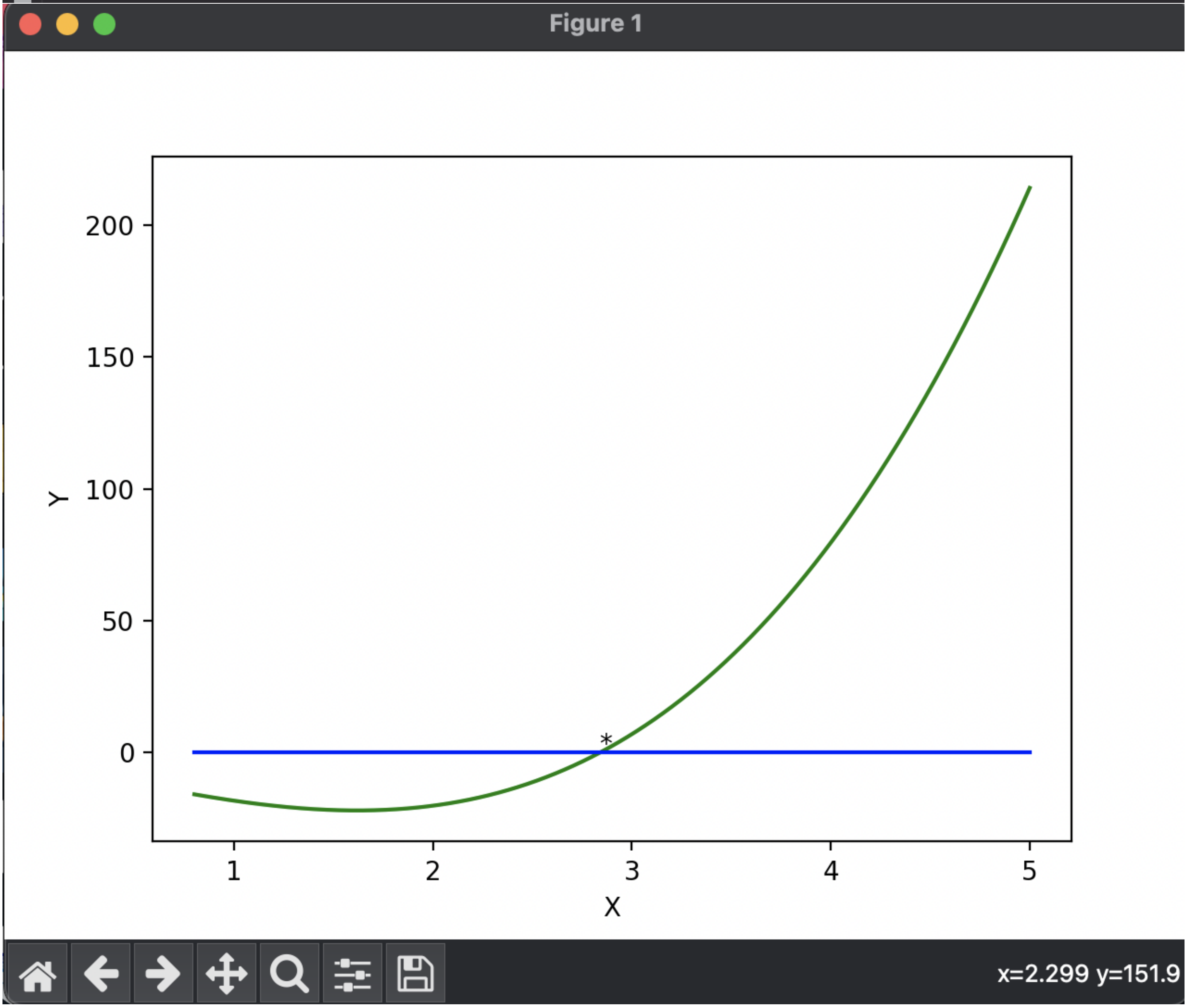
```
import matplotlib.pyplot as plt
from math import cos, sin


def simple_iteration_method_manual(func1, func2, x0, y0, eps):
    x1 = func1(x0, y0)
```

```python
    y1 = func2(x0, y0)
    print(f"x0={x0}, y0={y0}")
    print(f"x1=f1(x0, y0)=1-0.5cos({y0})={x1}, |x1-x0|={abs(x1 - x0)}")
    print(f"y1=f2(x0, y0)=sin({x0}+1)-1.2={y1}, |y1-y0|={abs(y1 - y0)}")
    print("=============")
    while abs(x1 - x0) > eps or abs(y1 - y0) > eps:
        x1, x0 = func1(x1, y1), x1
        y1, y0 = func2(x1, y1), y1
        print(f"x1=f1(x0, y0)=1-0.5cos({y0})={x1}, |x1-x0|={abs(x1 - x0)}")
        print(f"y1=f2(x0, y0)=sin({x0}+1)-1.2={y1}, |y1-y0|={abs(y1 - y0)}")
        print("=============")
    return x1, y1


def check_convergence(func1, func2, x1, x2, y1, y2, dx=0.00001):
    i = 0
    while True:
        break
        dfdx1 = get_derivative_of_x_at_point(func1, x1 + dx * i, y1 + dx * i, dx)
        dfdy1 = get_derivative_of_x_at_point(func1, x1 + dx * i, y1 + dx * i, dx)

        if abs(dfdy1) + abs(dfdx1) > 1:
            return False

        dfdx2 = get_derivative_of_x_at_point(func2, x1 + dx * i, y1 + dx * i, dx)
        dfdy2 = get_derivative_of_x_at_point(func2, x1 + dx * i, y1 + dx * i, dx)

        if abs(dfdy2) + abs(dfdx2) > 1:
            return False

        if x1 >= x2 or y1 >= y2:
            break
        i += 1
    return True


def simple_iteration_method(func1, func2, x01, x02, a1, b1, a2, b2, eps):
    if not check_convergence(func1, func2, a1, b1, a2, b2):
        print("Метод не сходится")
        return
    x1 = func1(x01)
    x2 = func2(x02)
    i = 0
    print(f"x0={x01}, y0={x02}")
    print(f"x1=f1(x0, y0)=1-0.5cos({x02})={x1}, |x1-x0|={abs(x1 - x01)}")
    print(f"y1=f2(x0, y0)=sin({x01}+1)-1.2={x2}, |y1-y0|={abs(x2 - x02)}")
    while abs(x1 - x01) > eps or abs(x2 - x02) > eps:
        i += 1
        x2, x02 = func1(x1), x2
        x1, x01 = func2(x2), x1
        print(f"x{i}={x01}, y{i}={x01}")
        print(f"x{i + 1}=f1(x{i}, y{i})=1-0.5cos({x02})={x1}, |x{i + 1}-x{i}|={abs(x1 -
x01)}")
        print(f"y{i + 1}=f2(x{i}, y{i})=sin({x01}+1)-1.2={x2}, |y{i + 1}-y{i}|={abs(x2 -
x02)}")
    return x1, x2


def get_derivative_of_x_at_point(func, x0, y0, dx=0.001):
    return (func(x0 + dx, y0) - func(x0, y0)) / dx


def get_derivative_of_y_at_point(func, x0, y0, dy=0.001):
    return (func(x0, y0 + dy) - func(x0, y0)) / dy


def draw_plots(func1, func2, a, b, root, eps):
    xs1, xs2 = [], []
    ys1, ys2 = [], []
    x = a
```

```python
    while x < b:
        xs1.append(x)
        ys1.append(func1(x))

        xs2.append(func2(x))
        ys2.append(x)
        x += eps

    plt.xlabel("X")
    plt.ylabel("Y")
    plt.annotate('x', xy=(root, func1(root)))
    plt.plot(xs1, ys1, 'r')
    plt.plot(xs2, ys2, 'g')
    plt.plot([a, b], [0, 0], 'b')

    plt.show()


# simple_iteration_method_manual(lambda x, y: 1 - 0.5 * cos(y), lambda x, y: sin(x + 1) - 
1.2, 0.2, 0, 10 ** -2)
print('Варианты систем:')
print('1. sin(x+1) - y = 1.2 and 2x + cos(y) = 2')
print('2. sin(x) + 2y = 2 and x + cos(y-1) = 0.7')
case_number = input('Введите номер системы: ')
while case_number not in {'1', '2'}:
    case_number = input('Введите номер системы: ')

case_number = int(case_number)
if case_number == 1:
    f1 = lambda x: sin(x + 1) - 1.2
    f2 = lambda y: 1 - 0.5 * cos(y)
    a1, b1 = 0.2, 0.6
    a2, b2 = -0.4, 0
else:
    f1 = lambda x: 1 - 0.5 * sin(x)   # x(y)
    f2 = lambda y: 0.7 - cos(y - 1)   # y(x)
    a1, b1 = -0.2, 0
    a2, b2 = 0.4, 0.6

x01 = (a1 + b1) / 2
x02 = (a2 + b2) / 2
while True:
    try:
        s = input("Введите нач приближения через пробел: ")
        if s.strip():
            x01, x02 = map(float, s.split())
            break
        else:
            break
    except Exception:
        print("Неверный ввод")
        continue

if not a1 <= x01 <= b1:
    x01 = a1
if not a2 <= x02 <= b2:
    x02 = a2

print(x01, x02)
eps = 0.00000001
ans = simple_iteration_method(f1, f2, x01, x02, a1, b1, a2, b2, eps)
print(ans)
root = ans[0]
draw_plots(f1, f2, -10, 10, root, 0.01)
```

Тестирование

```
(0.5101501574504936, -0.20183841535411562)
➜  lab3 python3 single.py
Выберите функцию:
1. f(x) = 2.74 * x ** 3 - 1.93 * x ** 2 - 15.28 * x - 3.72
2. f(x) = cos(x)
3. f(x) = x ** 2 - 1
Введите номер функции: 1
Напишите "да", если хотите задать [a; b]: ljk
Метод деления пополам: 2.8379394531249993
Метод Ньютона: 2.837964053717913
Метод простой итерации: 2.837960581776567
```



Вывод

В ходе выполнения лабораторной работы я познакомился с приближёнными методами для решения нелинейных уравнений и систем нелинейных уравнений.

Методы для уравнений:
Метод половинного деления – простой и хороший метод, напоминает всем известный бинарный поиск. Из минусов – это медленный метод, и при содержании нескольких корней на интервале неизвестно к какому из них приведёт метод.
Метод хорд – простой в реализации, нужно выбирать начальное приближение. Есть вариации с фиксированным концом.
Метод Ньютона – быстрый, минусы: нужно выбирать начальное приближение, нужна дифференцируемость функций, необходимость вычислять производные.
Метод секущих – упрощение метода Ньютона, не требуется вычислять производную, но новое приближение вычисляется на основе двух предыдущих.
Метод простой итерации – простой в реализации, интересный. Из минусов – сходимость в малой окрестности корня, нужно грамотно выбирать начальное приближение.

Методы для систем:
Метод Ньютона – сложный в понимании, важно удачно выбрать начальное приближение.
Метод простой итерации – несложный в понимании, несложно реализовать, мне понравился.