**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ**

**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский Авиационный Институт»**

**(Национальный Исследовательский Университет)**

**Институт: №8 «Компьютерные науки и прикладная математика»   
Кафедра: 806 «Вычислительная математика   
и программирование»**

Лабораторная работа № 2   
по курсу «Численные методы»

Группа: М8О-309Б-22

Студент: Концебалов О. С.

Преподаватель: Шабунина А. А.

Дата: 05.06.2025

Москва, 2025

**ОГЛАВЛЕНИЕ**

[**1**](#_o9boa449fvxm) **Тема** 3

[**2**](#_qiilxww84g5s) **Задание** 3

[**3**](#_igwfqjqtgk4l) **Теория** 4

[**4**](#_u3cpset8ahnh) **Ход лабораторной работы** 6

[**5**](#_gfaw1mnar37s) **Выводы** 15

# **Тема**

Решение нелинейных уравнений и систем нелинейных уравнений

# **Задание**

2.1. Реализовать методы простой итерации и Ньютона решения нелинейных уравнений в виде программ, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения найти положительный корень нелинейного уравнения (начальное приближение определить графически). Проанализировать зависимость погрешности вычислений от количества итераций.

Вариант 9: 

2.2. Реализовать методы простой итерации и Ньютона решения систем нелинейных уравнений в виде программного кода, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения решить систему нелинейных уравнений (при наличии нескольких решений найти то из них, в котором значения неизвестных являются положительными); начальное приближение определить графически. Проанализировать зависимость погрешности вычислений от количества итераций.

Вариант 9:



a = 4

# **Теория**

**Метод Ньютона (метод касательных).** При нахождении корня уравнения методом Ньютона, итерационный процесс определяется формулой Изображение выглядит как Шрифт, текст, рукописный текст, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Для начала вычислений требуется задание начального приближения. Условия сходимости метода определяются следующей теоремой.

Теорема: пусть на отрезке функция имеет первую и вторую производные постоянного знака и пусть . Тогда если точка выбрана на так, что , то начатая с нее последовательность , определяемая методом Ньютона, монотонно сходится к корню уравнения. В качестве условия окончания итераций в практических вычислениях часто используется правило

**Метод простой итерации.** При использовании метода простой итерации уравнение заменяется эквивалентным уравнением с выделенным линейным членом . Решение ищется путем построения последовательности ) начиная с некоторого заданного значения . Если – непрерывная функция, а – сходящаяся последовательность, то значение является решением уравнения. Условия сходимости метода и оценка его погрешности определяются теоремой: пусть функция определена и дифференцируема на отрезке . Тогда если выполняются условия:

1. ,
2. ,

то уравнение имеет и притом единственный на корень ; к этому корню сходится определяемая методом простой итерации последовательность , начиная с любого .

При этом справедливы оценки погрешности :

# **Ход лабораторной работы**

Код был реализован на языке Python.

* 1. Решение нелинейного уравнения:

import numpy as np

def f(x):

return x\*\*3 + x\*\*2 - 2\*x - 1

def df(x):

return 3\*(x\*\*2) + 2\*x - 2

def ddf(x):

return 6\*x + 2

def phi\_2(x):

return (x\*\*3 + x\*\*2 - 1) / 2

def d\_phi\_2(x):

return 1.5\*(x\*\*2) + x

def phi\_4(x):

return np.cbrt(1 + 2\*x - x\*\*2)

def d\_phi\_4(x):

under = 1 + 2\*x - x\*\*2

return (2 - 2\*x) / (3 \* np.cbrt(under\*\*2))

def check\_phi\_first\_derivative(a, b, eps, d\_phi):

max\_q = 0

x\_vals = np.arange(a, b, eps \* 2)

for x in x\_vals:

q = abs(d\_phi(x))

if q - 1 > eps:

print("ERROR: Условие сходимости нарушено: |phi'(x)| >= 1 для некоторого x")

exit(0)

if q > max\_q:

max\_q = q

print(f"Максимальное значение |phi'(x)| на интервале [{a}, {b}] = {max\_q}")

return max\_q

def simple\_iteration\_method(eps, left, right, phi, d\_phi):

x = (left + right) / 2

counter = 0

q = check\_phi\_first\_derivative(left, right, eps, d\_phi)

while True:

# q = abs(d\_phi(x))

# if q >= 1:

# print("ERROR: Условие сходимости не выполнено: |phi'(x)| >= 1")

# exit(0)

prev\_x = x

x = phi(x)

counter += 1

if counter > 1000:

print(f"ERROR: Method is diverged for interval between {left} and {right}")

exit(0)

if (q / (1 - q)) \* abs(x - prev\_x) <= eps:

break

print(f"Простые Итерации на [{left}, {right}] — количество итераций: ", counter)

return x

def newton\_method(eps, left, right, func, first\_d, second\_d):

f\_left = func(left)

f\_double\_left = second\_d(left)

f\_right = func(right)

f\_double\_right = second\_d(right)

if f\_left \* f\_double\_left > 0:

x = left

elif f\_right \* f\_double\_right > 0:

x = right

else:

print("WARN: Метод может не сходиться")

x = (left + right) / 2

counter = 0

while True:

f\_x = func(x)

d\_f\_x = first\_d(x)

if abs(d\_f\_x) < eps:

print("ERROR: Деление на 0!")

exit(0)

prev\_x = x

x = x - f\_x / d\_f\_x

counter += 1

if abs(x - prev\_x) <= eps:

break

print(f"Метод Ньютона на [{left}, {right}] — количество итераций: ", counter)

return x

def check\_result(eps, func, x):

eps = 0.001

return abs(func(x)) < eps

if \_\_name\_\_ == '\_\_main\_\_':

eps = float(input("Введите точность: "))

print("Simple Iterations Method\n")

check\_phi\_first\_derivative(-2, -1.8, eps, d\_phi\_4)

x\_1 = simple\_iteration\_method(eps, -2, -1.8, phi\_4, d\_phi\_4)

print(f"x\_1 = {x\_1}, is correct: {check\_result(eps, f, x\_1)}\n")

check\_phi\_first\_derivative(-0.5, -0.2, eps, d\_phi\_2)

x\_2 = simple\_iteration\_method(eps, -0.5, -0.2, phi\_2, d\_phi\_2)

print(f"x\_2 = {x\_2}, is correct: {check\_result(eps, f, x\_2)}\n")

check\_phi\_first\_derivative(1, 1.5, eps, d\_phi\_4)

x\_3 = simple\_iteration\_method(eps, 1, 1.5, phi\_4, d\_phi\_4)

print(f"x\_3 = {x\_3}, is correct: {check\_result(eps, f, x\_3)}\n")

print("-------------------------------------------------------------------\n")

print("Newton's Method\n")

x\_1 = newton\_method(eps, -2, -1.8, f, df, ddf)

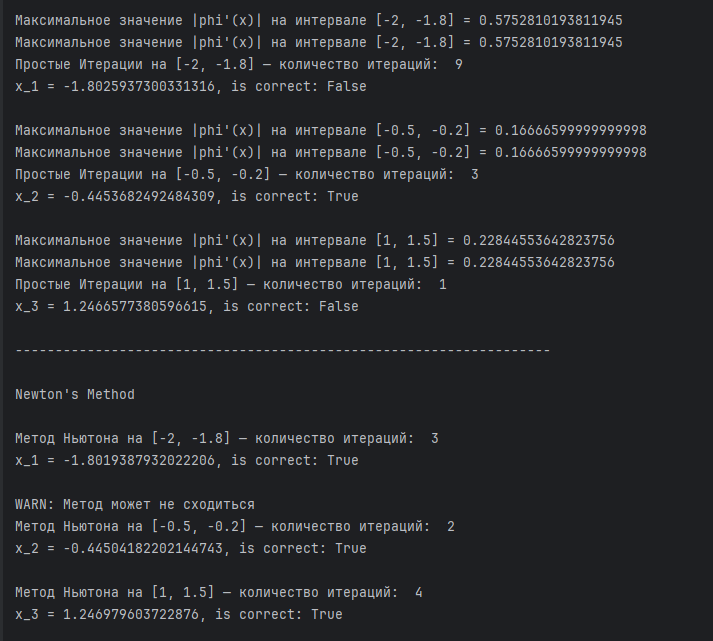
print(f"x\_1 = {x\_1}, is correct: {check\_result(eps, f, x\_1)}\n")

x\_2 = newton\_method(eps, -0.5, -0.2, f, df, ddf)

print(f"x\_2 = {x\_2}, is correct: {check\_result(eps, f, x\_2)}\n")

x\_3 = newton\_method(eps, 1, 1.5, f, df, ddf)

print(f"x\_3 = {x\_3}, is correct: {check\_result(eps, f, x\_3)}\n")



1.2) Решение системы нелинейных уравнений:

import matplotlib.pyplot as plt

import numpy as np

class NonlinearSystemSolver:

def \_\_init\_\_(self, epsilon=1e-6, max\_iter=100, x1\_range=(0.5, 2.0), x2\_range=(0.5, 2.0)):

self.epsilon = epsilon

self.max\_iter = max\_iter

self.x1\_range = x1\_range

self.x2\_range = x2\_range

self.q = self.\_compute\_q()

def \_compute\_q(self):

"""Вычисляет константу Липшица q как максимум нормы матрицы Якоби на заданном отрезке"""

n\_points = 100

x1\_samples = np.linspace(\*self.x1\_range, n\_points)

x2\_samples = np.linspace(\*self.x2\_range, n\_points)

max\_q = 0

for x1 in x1\_samples:

for x2 in x2\_samples:

try:

J = self.J\_iterations([x1, x2])

current\_q = np.max(np.sum(np.abs(J), axis=1))

if current\_q > max\_q:

max\_q = current\_q

except:

continue # Пропускаем точки, где матрица Якоби не определена

if max\_q == 0:

raise ValueError("Не удалось вычислить q на заданном отрезке")

print(f"Вычисленная константа сходимости q = {max\_q:.6f}")

return max\_q

@staticmethod

def F(x):

"""Система уравнений"""

x1, x2 = x[0], x[1]

return np.array([

x1 \*\* 2 - x2 \*\* 2 - 4, # F1(x1,x2)

x1 - np.exp(x2) + 2 # F2(x1,x2)

])

@staticmethod

def g1(x2):

"""Преобразование первого уравнения: x1 = g1(x2)"""

return np.sqrt(x2 \*\* 2 + 4)

@staticmethod

def g2(x1):

"""Преобразование второго уравнения: x2 = g2(x1)"""

return np.log(x1 + 2)

@staticmethod

def J\_iterations(x):

"""Матрица Якоби для метода простых итераций"""

x1, x2 = x[0], x[1]

return np.array([

[0, x2 / np.sqrt(x2 \*\* 2 + 4)],

[1 / (x1 + 2), 0]

])

@staticmethod

def J\_newton(x):

"""Матрица Якоби для метода Ньютона"""

x1, x2 = x[0], x[1]

return np.array([

[2 \* x1, -2 \* x2],

[1, -np.exp(x2)]

])

def check\_convergence(self, x0):

"""Проверка условия сходимости для метода простых итераций"""

J = self.J\_iterations(x0)

norm = np.max(np.sum(np.abs(J), axis=1))

if norm >= 1:

raise ValueError(f"Условие сходимости нарушено: норма Якобиана = {norm:.4f} ≥ 1")

return norm

class NonlinearSystemSolver:

def \_\_init\_\_(self, epsilon=1e-6, max\_iter=100, x1\_range=(0.5, 2.0), x2\_range=(0.5, 2.0)):

self.epsilon = epsilon

self.max\_iter = max\_iter

self.x1\_range = x1\_range

self.x2\_range = x2\_range

self.q = self.\_compute\_q()

def \_compute\_q(self):

"""Вычисляет константу Липшица q как максимум нормы матрицы Якоби на заданном отрезке"""

n\_points = 100

x1\_samples = np.linspace(\*self.x1\_range, n\_points)

x2\_samples = np.linspace(\*self.x2\_range, n\_points)

max\_q = 0

for x1 in x1\_samples:

for x2 in x2\_samples:

try:

J = self.J\_iterations([x1, x2])

current\_q = np.max(np.sum(np.abs(J), axis=1))

if current\_q > max\_q:

max\_q = current\_q

except:

continue # Пропускаем точки, где матрица Якоби не определена

if max\_q == 0:

raise ValueError("Не удалось вычислить q на заданном отрезке")

print(f"Вычисленная константа сходимости q = {max\_q:.6f}")

return max\_q

@staticmethod

def F(x):

"""Система уравнений"""

x1, x2 = x[0], x[1]

return np.array([

x1 \*\* 2 - x2 \*\* 2 - 4, # F1(x1,x2)

x1 - np.exp(x2) + 2 # F2(x1,x2)

])

@staticmethod

def g1(x2):

"""Преобразование первого уравнения: x1 = g1(x2)"""

return np.sqrt(x2 \*\* 2 + 4)

@staticmethod

def g2(x1):

"""Преобразование второго уравнения: x2 = g2(x1)"""

return np.log(x1 + 2)

@staticmethod

def J\_iterations(x):

"""Матрица Якоби для метода простых итераций"""

x1, x2 = x[0], x[1]

return np.array([

[0, x2 / np.sqrt(x2 \*\* 2 + 4)],

[1 / (x1 + 2), 0]

])

@staticmethod

def J\_newton(x):

"""Матрица Якоби для метода Ньютона"""

x1, x2 = x[0], x[1]

return np.array([

[2 \* x1, -2 \* x2],

[1, -np.exp(x2)]

])

def check\_convergence(self, x0):

"""Проверка условия сходимости для метода простых итераций"""

J = self.J\_iterations(x0)

norm = np.max(np.sum(np.abs(J), axis=1))

if norm >= 1:

raise ValueError(f"Условие сходимости нарушено: норма Якобиана = {norm:.4f} ≥ 1")

return norm

def simple\_iteration(self, x0, verbose=True):

try:

if x0[1] \*\* 2 + 4 < 0: # Всегда истинно, оставляем для формальности

raise ValueError("Начальное x2 приводит к отрицательному подкоренному выражению в g1(x2)")

if x0[0] + 2 <= 0:

raise ValueError(f"Начальное x1 = {x0[0]} приводит к неположительному аргументу логарифма в g2(x1)")

# Проверка значения q

if self.q >= 1:

raise ValueError(f"Метод не сойдется: q = {self.q:.4f} ≥ 1")

except ValueError as e:

if verbose:

print(f"Ошибка в начальном приближении: {e}")

return None, 0, [], []

if verbose:

print(f"\nМетод простых итераций с q = {self.q:.6f}")

print(f"Начальное приближение x0 = [{x0[0]}, {x0[1]}] входит в область определения")

print("Теоретическая оценка погрешности после n итераций: |x\_n - x\*| ≤ q^n/(1-q) \* |x1 - x0|")

x = np.array(x0, dtype=float)

history = [x.copy()]

errors = []

for iter\_count in range(1, self.max\_iter + 1):

try:

# Проверка области определения на каждой итерации

if x[1] \*\* 2 + 4 < 0:

raise ValueError("Подкоренное выражение в g1(x2) стало отрицательным")

if x[0] + 2 <= 0:

raise ValueError("Аргумент логарифма в g2(x1) стал неположительным")

x\_new = np.array([self.g1(x[1]), self.g2(x[0])])

error = np.linalg.norm(x\_new - x, np.inf)

errors.append(error)

if verbose:

theoretical\_error = (self.q \*\* iter\_count) / (1 - self.q) \* errors[0]

print(f"Итерация {iter\_count:3d}: x = [{x\_new[0]:.8f}, {x\_new[1]:.8f}]")

print(f"Ошибка: {error:.2e} | Теор. оценка: {theoretical\_error:.2e}")

if error < self.epsilon:

if verbose:

print(f"\nСходимость достигнута за {iter\_count} итераций")

return x\_new, iter\_count, errors, history

x = x\_new

history.append(x.copy())

except ValueError as e:

if verbose:

print(f"\nОшибка на итерации {iter\_count}: {e}")

return x, iter\_count, errors, history

if verbose:

print(f"\nДостигнут максимум итераций ({self.max\_iter})")

return x, self.max\_iter, errors, history

def newton\_method(self, x0, verbose=True):

try:

# Проверка системы уравнений

F\_val = self.F(x0)

if x0[0] + 2 <= 0:

raise ValueError(f"Начальное x1 = {x0[0]} приводит к неположительному аргументу логарифма")

# Проверка матрицы Якоби

J = self.J\_newton(x0)

det = np.linalg.det(J)

if abs(det) < 1e-12:

raise ValueError(f"Матрица Якоби вырождена в начальной точке (det(J) = {det:.2e})")

except ValueError as e:

if verbose:

print(f"Ошибка в начальном приближении: {e}")

return None, 0, [], []

if verbose:

print(f"\nМетод Ньютона")

print(f"Начальное приближение x0 = [{x0[0]}, {x0[1]}] входит в область определения")

print(f"Начальная невязка: F(x0) = [{F\_val[0]:.6f}, {F\_val[1]:.6f}]")

x = np.array(x0, dtype=float)

history = [x.copy()]

errors = []

for iter\_count in range(1, self.max\_iter + 1):

try:

# Проверка области определения перед вычислениями

if x[0] + 2 <= 0:

raise ValueError(f"Аргумент логарифма стал неположительным (x1 = {x[0]})")

F\_val = self.F(x)

J = self.J\_newton(x)

# Проверка определителя матрицы Якоби

det = np.linalg.det(J)

if abs(det) < 1e-12:

raise ValueError(f"Матрица Якоби вырождена (det(J) = {det:.2e})")

dx = np.linalg.solve(J, -F\_val)

x\_new = x + dx

error = np.linalg.norm(dx, np.inf)

errors.append(error)

if verbose:

print(f"Итерация {iter\_count}: x = [{x\_new[0]:.8f}, {x\_new[1]:.8f}]")

print(f"Ошибка: {error:.2e} | Абсолютная погрешность: [{F\_val[0]:.2e}, {F\_val[1]:.2e}]")

if error < self.epsilon:

if verbose:

print(f"\nСходимость достигнута за {iter\_count} итераций")

return x\_new, iter\_count, errors, history

x = x\_new

history.append(x.copy())

except ValueError as e:

if verbose:

print(f"\nОшибка на итерации {iter\_count}: {e}")

return x, iter\_count, errors, history

if verbose:

print(f"\nДостигнут максимум итераций ({self.max\_iter})")

return x, self.max\_iter, errors, history

@staticmethod

def plot\_system():

"""Визуализация системы уравнений"""

x2\_vals = np.linspace(0, 3, 400)

x1\_vals = np.linspace(0.01, 3, 400)

plt.figure(figsize=(10, 6))

plt.plot(np.sqrt(x2\_vals \*\* 2 + 4), x2\_vals, label=r'$x\_1 = \sqrt{x\_2^2 + 4}$')

plt.plot(x1\_vals, np.log(x1\_vals + 2), label=r'$x\_2 = \ln(x\_1 + 2)$')

plt.xlabel("x1")

plt.ylabel("x2")

plt.title("Графическое представление системы уравнений")

plt.legend()

plt.grid(True)

plt.show()

@staticmethod

def plot\_system():

"""Визуализация системы уравнений"""

x2\_vals = np.linspace(0, 3, 400)

x1\_vals = np.linspace(0.01, 3, 400)

plt.figure(figsize=(10, 6))

plt.plot(np.sqrt(x2\_vals \*\* 2 + 4), x2\_vals, label=r'$x\_1 = \sqrt{x\_2^2 + 4}$')

plt.plot(x1\_vals, np.log(x1\_vals + 2), label=r'$x\_2 = \ln(x\_1 + 2)$')

plt.xlabel("x1")

plt.ylabel("x2")

plt.title("Графическое представление системы уравнений")

plt.legend()

plt.grid(True)

plt.show()

def main(): # проверка области определения начального приближения

solver = NonlinearSystemSolver(epsilon=1e-6, max\_iter=100, x1\_range=(0.5, 2.0), x2\_range=(0.5, 2.0))

x0 = [2.25, 2.75]

print("=" \* 50)

print(f"РЕШЕНИЕ СИСТЕМЫ НЕЛИНЕЙНЫХ УРАВНЕНИЙ")

print("=" \* 50)

print(f"Начальное приближение: x1 = {x0[0]}, x2 = {x0[1]}")

print(f"Точность: {solver.epsilon}")

print(f"Максимальное число итераций: {solver.max\_iter}")

print("-" \* 50)

solver.plot\_system()

print("\n" + "=" \* 20 + " МЕТОД ПРОСТЫХ ИТЕРАЦИЙ " + "=" \* 20)

sol\_si, iter\_si, errors\_si, history\_si = solver.simple\_iteration(x0)

print("\n" + "=" \* 20 + " МЕТОД НЬЮТОНА " + "=" \* 20)

sol\_newton, iter\_newton, errors\_newton, history\_newton = solver.newton\_method(x0)

if sol\_si is None or sol\_newton is None:

return

print("\n" + "=" \* 20 + " РЕЗУЛЬТАТЫ " + "=" \* 20)

print(f"{'Метод':<25} | {'x1':<15} | {'x2':<15} | {'Итерации':<10}")

print("-" \* 65)

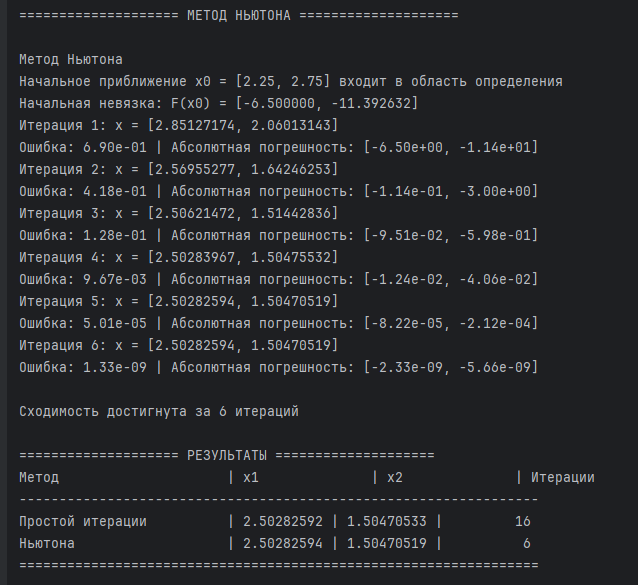
if sol\_si is not None:

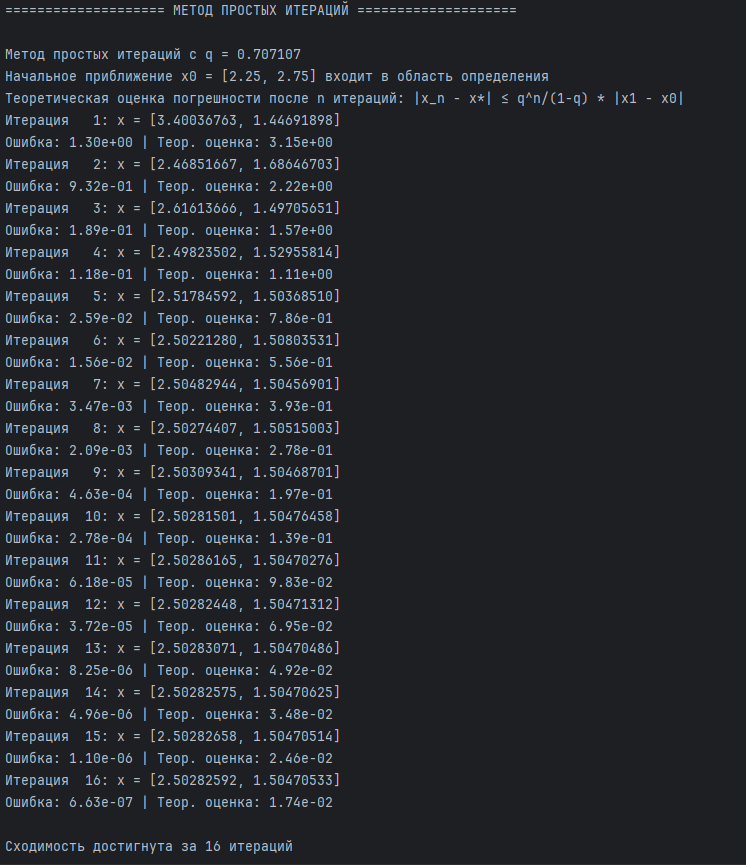
print(f"{'Простой итерации':<25} | {sol\_si[0]:.8f} | {sol\_si[1]:.8f} | {iter\_si:>10}")

print(f"{'Ньютона':<25} | {sol\_newton[0]:.8f} | {sol\_newton[1]:.8f} | {iter\_newton:>10}")

print("=" \* 65)

main()





# **Выводы**

В ходе выполнения лабораторной работы были успешно реализованы методы простой итерации и Ньютона для решения как отдельных нелинейных уравнений, так и систем нелинейных уравнений. Разработанное программное обеспечение позволяет задавать точность вычислений и анализировать сходимость методов. На примере конкретных уравнений и систем было показано, что метод Ньютона демонстрирует более быструю сходимость по сравнению с методом простой итерации, что подтверждается графиками зависимости погрешности от числа итераций. Однако метод Ньютона требует вычисления производных и может быть менее устойчив при неудачном выборе начального приближения. Метод простой итерации, хотя и сходится медленнее, оказывается более надежным для некоторых задач. Полученные решения были верифицированы подстановкой в исходные уравнения, что подтвердило их корректность. Работа также продемонстрировала важность правильного выбора начального приближения, которое можно эффективно определять графическим способом. В целом, лабораторная работа позволила на практике изучить особенности численных методов решения нелинейных уравнений и систем, их преимущества и ограничения.