**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ**

**УНИВЕРСИТЕТ им. В. Г. ШУХОВА»**

**(БГТУ им. В.Г. Шухова)**



ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

**Лабораторная работа №3**

по дисциплине: Вычислительная математика

тема: «Решение Систем Нелинейных Уравнений»

Выполнил: ст. группы ВТ-231

Масленников Даниил

Проверили:

Островский Алексей Мичеславович

Белгород 2025 г

Л а б о р а т о р н а я р а б о т а № 3

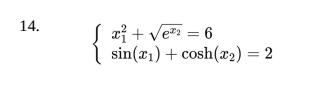
**Цель работы:** Изучить методы решения нелинейных систем уравнений и особенности алгоритмизации в экосистемах языков Rust и Python

**1. Построение графиков и выбор начального приближения (Python):**

* + Построить графики нелинейных функций системы уравнений.
  + Выбрать начальное приближение для нахождения корня, ближайшего к началу координат.
  + Получить контрольное решение с использованием библиотеки scipy.optimize.fsolve.

1. **Реализация метода Ньютона (Rust):**
   * Написать программу на Rust для решения системы нелинейных уравнений методом Ньютона.
   * Использовать якобиан и обратную матрицу для обновления решения.
   * Проверить сходимость и точность.
2. **Реализация метода простой итерации (Rust):**
   * Написать программу на Rust для решения системы нелинейных уравнений методом простой итерации.
   * Преобразовать систему уравнений в итерационную форму.
   * Проверить сходимость и точность.
3. **Реализация метода градиентного спуска (Rust):**
   * Написать программу на Rust для решения системы нелинейных уравнений методом градиентного спуска.
   * Минимизировать сумму квадратов невязок системы.
   * Проверить сходимость и точность.
4. **Сравнение методов:**
   * Сравнить вычислительные схемы и результаты для методов Ньютона, простой итерации и градиентного спуска.
   * Сделать выводы о скорости сходимости, точности и применимости каждого метода.

**Вариант 14**



**Задание 1: Построение графиков нелинейных функций на Python**

**import numpy as np**

**import** **matplotlib.pyplot** **as** **plt**

**from** **scipy.optimize** **import** fsolve

**def** **nonlinear\_equations**(variables):

x1, x2 = variables

equation1 = np.sqrt(np.exp(x2)) + x1\*\***2** - **6**

equation2 = np.sin(x1) + np.cosh(x2) - **2**

**return** [equation1, equation2]

**def** **plot\_solution\_and\_equations**(solution):

x1\_values = np.linspace(-**5**, **5**, **400**)

x2\_values = np.linspace(-**5**, **5**, **400**)

X1, X2 = np.meshgrid(x1\_values, x2\_values)

Z1 = np.sqrt(np.exp(X2)) + X1\*\***2** - **6**

Z2 = np.sin(X1) + np.cosh(X2) - **2**

plt.figure(figsize=(**8**, **6**))

contour1 = plt.contour(X1, X2, Z1, levels=[**0**], colors='r')

plt.clabel(contour1, inline=**True**, fontsize=**10**, fmt='Уравнение 1')

contour2 = plt.contour(X1, X2, Z2, levels=[**0**], colors='b')

plt.clabel(contour2, inline=**True**, fontsize=**10**, fmt='Уравнение 2')

plt.plot(solution[**0**], solution[**1**], 'ko', label='Решение')

plt.text(solution[**0**], solution[**1**], ' Решение', verticalalignment='bottom')

plt.xlabel('x1')

plt.ylabel('x2')

plt.title('Решение системы нелинейных уравнений')

plt.grid(**True**)

plt.legend()

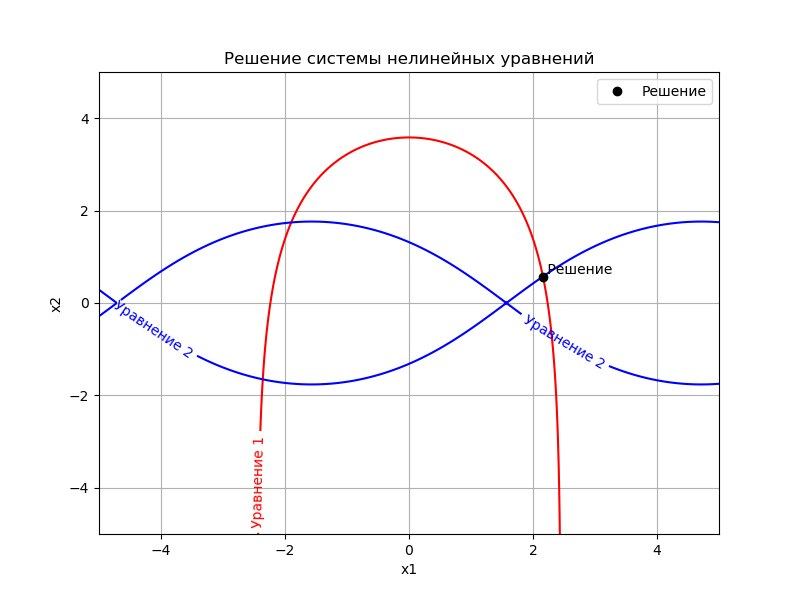
plt.show()

initial\_guess = [**1.0**, **1.0**]

solution = fsolve(nonlinear\_equations, initial\_guess)

print(f"Решение: x1 = {solution[0]}, x2 = {solution[1]}")

plot\_solution\_and\_equations(solution)

  
  
  
  
  
**Задание 2: Реализация следующих алгоритмов на Rust: метод Ньютона, простой итерации и градиентного спуска  
  
use std::f64;**

**fn is\_zero(n: f64, eps: f64) -> bool {**

**n.abs() < eps**

**}**

**fn inverse\_matrix\_2x2(matrix: [[f64; 2]; 2], epsilon: f64) -> Result<[[f64; 2]; 2], &'static str> {**

**let det = matrix[0][0] \* matrix[1][1] - matrix[0][1] \* matrix[1][0];**

**if is\_zero(det, epsilon) {**

**return Err("Матрица вырождена");**

**}**

**Ok([**

**[matrix[1][1] / det, -matrix[0][1] / det],**

**[-matrix[1][0] / det, matrix[0][0] / det],**

**])**

**}**

**fn matrix\_vector\_multiply(matrix: [[f64; 2]; 2], vector: [f64; 2]) -> [f64; 2] {**

**[**

**matrix[0][0] \* vector[0] + matrix[0][1] \* vector[1],**

**matrix[1][0] \* vector[0] + matrix[1][1] \* vector[1],**

**]**

**}**

**fn f(x: [f64; 2]) -> [f64; 2] {**

**[**

**x[0].powi(2) + f64::sqrt(f64::exp(x[1])) - 6.0,**

**f64::sin(x[0]) + f64::cosh(x[1]) - 2.0,**

**]**

**}**

**fn jacobian(x: [f64; 2]) -> [[f64; 2]; 2] {**

**[**

**[2.0 \* x[0], 0.5 \* f64::exp(x[1]) / f64::sqrt(f64::exp(x[1]))],**

**[f64::cos(x[0]), f64::sinh(x[1])],**

**]**

**}**

**fn newton\_method(**

**initial\_guess: [f64; 2],**

**epsilon: f64,**

**max\_iterations: usize,**

**) -> Result<[f64; 2], &'static str> {**

**let mut x = initial\_guess;**

**for iteration in 0..max\_iterations {**

**let j = jacobian(x);**

**let inv\_j = inverse\_matrix\_2x2(j, epsilon)?;**

**let fx = f(x);**

**let delta = matrix\_vector\_multiply(inv\_j, [-fx[0], -fx[1]]);**

**x = [x[0] + delta[0], x[1] + delta[1]];**

**if delta[0].abs() < epsilon && delta[1].abs() < epsilon {**

**println!("Метод Ньютона сошелся за {} итераций", iteration);**

**return Ok(x);**

**}**

**}**

**Err("Метод Ньютона не сошелся")**

**}**

**// простая итерация**

**fn simple\_iteration\_method(**

**initial\_guess: [f64; 2],**

**epsilon: f64,**

**max\_iterations: usize,**

**) -> Result<[f64; 2], &'static str> {**

**let mut x = initial\_guess;**

**for iteration in 0..max\_iterations {**

**let x\_new = [**

**f64::sqrt(6.0 - f64::sqrt(f64::exp(x[1]))),**

**f64::acosh(2.0 - f64::sin(x[0])),**

**];**

**if (x\_new[0] - x[0]).abs() < epsilon && (x\_new[1] - x[1]).abs() < epsilon {**

**println!("Метод простой итерации сошелся за {} итераций", iteration);**

**return Ok(x\_new);**

**}**

**x = x\_new;**

**}**

**Err("Метод простой итерации не сошелся")**

**}**

**// градиентный спуск**

**fn gradient\_descent(**

**initial\_guess: [f64; 2],**

**learning\_rate: f64,**

**epsilon: f64,**

**max\_iterations: usize,**

**) -> Result<[f64; 2], &'static str> {**

**let mut x = initial\_guess;**

**for iteration in 0..max\_iterations {**

**let grad = gradient(x);**

**x = [**

**x[0] - learning\_rate \* grad[0],**

**x[1] - learning\_rate \* grad[1],**

**];**

**if grad[0].abs() < epsilon && grad[1].abs() < epsilon {**

**println!(**

**"Метод градиентного спуска сошелся за {} итераций",**

**iteration**

**);**

**return Ok(x);**

**}**

**}**

**Err("Метод градиентного спуска не сошелся")**

**}**

**fn gradient(x: [f64; 2]) -> [f64; 2] {**

**let h = 1e-6;**

**let mut grad = [0.0; 2];**

**for i in 0..2 {**

**let mut x\_plus\_h = x;**

**let mut x\_minus\_h = x;**

**x\_plus\_h[i] += h;**

**x\_minus\_h[i] -= h;**

**grad[i] = (objective\_function(x\_plus\_h) - objective\_function(x\_minus\_h)) / (2.0 \* h);**

**}**

**grad**

**}**

**fn objective\_function(x: [f64; 2]) -> f64 {**

**let eq1 = x[0].powi(2) + f64::sqrt(f64::exp(x[1])) - 6.0;**

**let eq2 = f64::sin(x[0]) + f64::cosh(x[1]) - 2.0;**

**eq1.powi(2) + eq2.powi(2)**

**}**

**fn main() {**

**let initial\_guess = [1.0, 1.0];**

**let epsilon = 1e-6; // точность**

**let max\_iterations = 1000; // макс итераций**

**let learning\_rate = 0.0501; // шаг**

**match newton\_method(initial\_guess, epsilon, max\_iterations) {**

**Ok(solution) => println!("Метод Ньютона: x1 = {}, x2 = {}", solution[0], solution[1]),**

**Err(e) => println!("Метод Ньютона: {}", e),**

**}**

**match simple\_iteration\_method(initial\_guess, epsilon, max\_iterations) {**

**Ok(solution) => println!(**

**"Метод простой итерации: x1 = {}, x2 = {}",**

**solution[0], solution[1]**

**),**

**Err(e) => println!("Метод простой итерации: {}", e),**

**}**

**match gradient\_descent(initial\_guess, learning\_rate, epsilon, max\_iterations) {**

**Ok(solution) => println!(**

**"Метод градиентного спуска: x1 = {}, x2 = {}",**

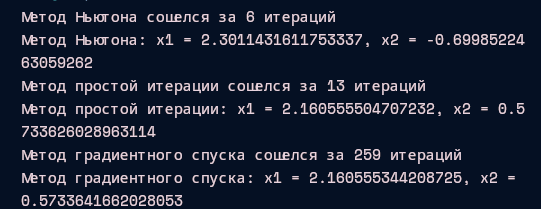
**solution[0], solution[1]**

**),**

**Err(e) => println!("Метод градиентного спуска: {}", e),**

**}**

**}**

 **Вывод: сравнив все методы пришел к выводу, что метод градиентного спуска наиболее универсален, ведь метод Ньютона требует больше ресурсов на каждой итерации, методы Ньютона и простой итерации чувствительны к выбору начального приближения. Градиентный спуск же чувствителен только к шагу, позволяя его применять наиболее часто.**