**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ,   
МЕХАНИКИ И ОПТИКИ»**

**Факультет программной инженерии и компьютерной техники**

**Дисциплина:**

**«Вычислительная математика»**

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №2**

**Вариант 9.**

**Выполнил:**

Студент гр. P32151

Понамарев Степан Андреевич

**Проверил:**

Машина Екатерина Алексеевна

1. **Цель лабораторной работы**: изучить численные методы решения нелинейных уравнений и их систем, найти корни заданного нелинейного уравнения/системы нелинейных уравнений, вы-полнить программную реализацию методов.
2. **Изображение выглядит как текст

   Автоматически созданное описаниеЗадание лабораторной работы**:
3. **Рабочий метод по варианту и рабочие формулы (вариант 9)**:
4. Крайний левый корень: **Метод простой итерации**.
   * получается приведением к эквивалентному виду
   * определяется начальным приближением
   * Критерий окончания итерационного процесса:
5. Центральный корень: **Метод секущих**.
   * Выбор : если , иначе
   * выбирается рядом с
6. Крайний правый корень: **Метод половинного деления**.
   * и , если , иначе и
7. **Вычислительная реализация задачи**:
8. Отделить корни заданного нелинейного уравнения графически

Изображение выглядит как диаграмма

Автоматически созданное описаниеВид уравнения:

Рис. 1. График нелинейного уравнения

1. Интервалы изоляции корней:
2. Уточнение корней нелинейного уравнения с точностью .
3. Таблицы вычислительной части лабораторной работы:
   1. Метод простой итерации для крайнего левого корня (:
   * , , условие сходимости выполняется.
   * – начальное приближение.

Таблица 1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № итерации |  |  |  |  |  |
| 1 | -3,000 | -2,076 | -1,703 | -12,714 | 0,924 |
| 2 | -2,076 | -1,703 | -1,528 | -11,918 | 0,373 |
| 3 | -1,703 | -1,528 | -1,438 | -10,909 | 0,175 |
| 4 | -1,528 | -1,438 | -1,388 | -10,257 | 0,090 |
| 5 | -1,438 | -1,388 | -1,360 | -9,867 | 0,049 |
| 6 | -1,388 | -1,360 | -1,344 | -9,637 | 0,028 |
| 7 | -1,360 | -1,344 | -1,335 | -9,501 | 0,016 |
| 8 | -1,344 | -1,335 | -1,329 | -9,420 | 0,009 |

* 1. Метод секущих для центрального корня (:
* ,
* – не подходит, значит берём
* – точка рядом с
* ,

Таблица 2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № итерации |  |  |  |  |  |
| 1 | -0,200 | -0,250 | -0,480 | -0,082 | 0,230 |
| 2 | -0,250 | -0,480 | -0,474 | 0,001 | 0,007 |

* 1. Метод половинного деления для крайнего правого корня (:

Таблица 3

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| № шага |  |  |  |  |  |  |  |
| 1 | 1,800 | 2,200 | 2,000 | 4,023 | -5,202 | -0,040 | 0,400 |
| 2 | 1,800 | 2,000 | 1,900 | 4,023 | -0,040 | 2,123 | 0,200 |
| 3 | 1,900 | 2,000 | 1,950 | 2,123 | -0,040 | 1,075 | 0,100 |
| 4 | 1,950 | 2,000 | 1,975 | 1,075 | -0,040 | 0,526 | 0,050 |
| 5 | 1,975 | 2,000 | 1,988 | 0,526 | -0,040 | 0,245 | 0,025 |
| 6 | 1,988 | 2,000 | 1,994 | 0,245 | -0,040 | 0,103 | 0,013 |
| 7 | 1,994 | 2,000 | 1,997 | 0,103 | -0,040 | 0,032 | 0,006 |

1. Листинг программы (коды используемых методов):

**main.py**:

import numpy as np  
  
from AnyEquationSolver import EquationSolver  
from EquationManager import Equation  
from Exceptions import InvalidAlgebraicEquationException  
from InputManager import InputManager  
from PreparedEquationManager import PreparedEquation  
from PreparedSystemManager import PreparedSystem  
  
  
def variants\_function\_solver():  
 result = None  
 variants = ["3x^3 + 1.7x^2 − 15.42x + 6.89",  
 "x • sin(x) - 1",  
 "e^sin(x) • ln(|x|) - 1.4",  
 "arcsin(e^(-x)) - 0.3",  
 "tg(x/2 • sin(x))"]  
 values = [PreparedEquation(lambda x: 3 \* x \*\* 3 + 1.7 \* x \*\* 2 - 15.42 \* x + 6.89,  
 derivative\_func=lambda x: -15.42 + 3.4 \* x + 9 \* x \*\* 2),  
 PreparedEquation(lambda x: x \* np.sin(x) - 1,  
 derivative\_func=lambda x: np.sin(x) + x \* np.cos(x)),  
 PreparedEquation(lambda x: np.exp(np.sin(x)) \* np.log(abs(x)) - 1.4,  
 derivative\_func=lambda x: np.exp(np.sin(x)) \* (  
 abs(x) \* np.log(abs(x)) \* np.cos(x) + np.sign(x)) / abs(x)),  
 PreparedEquation(lambda x: np.arcsin(np.exp(-x)) - 0.3,  
 derivative\_func=lambda x: -np.exp(-x) / np.sqrt(1 - np.exp(-2 \* x))),  
 PreparedEquation(lambda x: np.tan(x / 2 \* np.sin(x)),  
 derivative\_func=lambda x: (np.sin(x) + x \* np.cos(x)) /  
 (2 \* np.cos(x \* np.sin(x) / 2) \*\* 2))]  
  
 chosen\_equation = InputManager.multiple\_choice\_input(variants, values, "Выберите функцию:")  
 try:  
 result = chosen\_equation.solve()  
 except ValueError:  
 print("Возможно, функция не определена в некоторых точках.\n"  
 "Попробуйте ввести другие концы отрезка.")  
 if result is None:  
 print("Не удалось найти корень уравнения.")  
 else:  
 print("------------РЕШЕНИЕ------------")  
 print("Найденный корень уравнения:", round(result, 5))  
 print("Значение функции в точке: ", round(chosen\_equation.calculate(result), 9))  
 print("Количество итераций:", chosen\_equation.iterations)  
 if InputManager.yes\_or\_no\_input("Показать график функции?"):  
 chosen\_equation.draw\_graphic(dot\_x=result, dot\_y=chosen\_equation.calculate(result))  
  
  
def any\_function\_solver():  
 try:  
 line = InputManager.algebraic\_expression\_input("Введите уравнение: ")  
 eq = Equation(line)  
 EquationSolver.solve(eq)  
 except InvalidAlgebraicEquationException:  
 print("Не удалось распарсить выражение")  
 print('Попробуйте не использовать знак "=".')  
  
  
def system\_solver():  
 name1 = "┌ 0.1x^2 + x + 0.2y^2 - 0.3" + "\n" + \  
 "└ 0.2x^2 + y + 0.1xy - 0.7" + "\n" + \  
 "Пример корня: (0.20 0.67)"  
 ps1 = PreparedSystem([lambda x, y: 0.1 \* x \*\* 2 + x + 0.2 \* y \*\* 2 - 0.3,  
 lambda x, y: 0.2 \* x \*\* 2 + y + 0.1 \* x \* y - 0.7],  
 derivative\_funcs=[lambda x, y: np.array([-0.2 \* x, -0.4 \* y]),  
 lambda x, y: np.array([-0.4 \* x - 0.1 \* y, -0.1 \* x])])  
 name2 = "┌ sin(x)" + "\n" + \  
 "└ xy/20 - 0.5" + "\n" + \  
 "Пример корня: (6.28 1.59)"  
 ps2 = PreparedSystem([lambda x, y: np.sin(x),  
 lambda x, y: x \* y / 20 - 0.5],  
 derivative\_funcs=[lambda x, y: np.array([np.cos(x), 0]),  
 lambda x, y: np.array([y / 20, x / 20])])  
 chosen\_system = InputManager.multiple\_choice\_input([name1, name2], [ps1, ps2], "Выберите систему:")  
  
 result = chosen\_system.solve()  
 if result is None:  
 print("Не удалось найти корень системы.")  
 else:  
 print("------------РЕШЕНИЕ------------")  
 print("Найденный корень системы:", result)  
 print("Значение системы в этой точке:", chosen\_system.calculate(\*result))  
 print("Количество итераций:", chosen\_system.iterations)  
 print("Вектор погрешностей:", chosen\_system.get\_error\_rate())  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 variants = ["Выбрать функцию и метод из предложенных",  
 "Ввести произвольную функцию и решить её методом половинного деления",  
 "Выбрать систему уравнений из предложенных и решить её методом простых итераций"]  
 values = [variants\_function\_solver, any\_function\_solver, system\_solver]  
  
 chosen\_variant = InputManager.multiple\_choice\_input(variants, values, "Чем хотите себя побаловать?")  
  
 chosen\_variant()

**InputManager.py**:

import numpy as np  
  
from Exceptions import InvalidAlgebraicEquationException  
  
  
class InputManager:  
 @staticmethod  
 def string\_input(message=""):  
 buf = ""  
 while buf == "":  
 buf = input(message).strip()  
 return buf  
  
 @staticmethod  
 def \_check\_number(buf):  
 try:  
 float(buf.replace(',', '.'))  
 return True  
 except ValueError:  
 return False  
  
 @staticmethod  
 def \_convert\_to\_number(num):  
 try:  
 return float(num.replace(',', '.'))  
 except ValueError:  
 return None  
  
 @staticmethod  
 def float\_input(message=""):  
 number = None  
 while number is None:  
 number = InputManager.\_convert\_to\_number(InputManager.string\_input(message))  
 return number  
  
 @staticmethod  
 def int\_input(message=""):  
 return int(InputManager.float\_input(message))  
  
 @staticmethod  
 def yes\_or\_no\_input(message=""):  
 answer = "0"  
 while answer[0].lower() not in ["y", "n", "д", "н"]:  
 answer = InputManager.string\_input(message + " [y/n]: ")  
 return answer[0].lower() in ["y", "д"]  
  
 @staticmethod  
 def matrix\_input():  
 lines\_number = InputManager.int\_input("Введите количество строк матрицы: ")  
 print("Введите матрицу вида\n"  
 "a\_11, a\_12, ..., a\_1n, b\_1\n"  
 "...\n"  
 "a\_n1, a\_n2, ..., a\_nn, b\_n\n")  
 matrix = [[0] \* (lines\_number + 1) for \_ in range(lines\_number)]  
 for i in range(lines\_number):  
 line = InputManager.string\_input().split()  
 while len(line) != lines\_number + 1 or not all([InputManager.\_check\_number(t) for t in line]):  
 print("Некорректный ввод строки")  
 line = InputManager.string\_input().split()  
 for j in range(lines\_number + 1):  
 matrix[i][j] = InputManager.\_convert\_to\_number(line[j])  
 print()  
 return np.array(matrix)  
  
 """  
 x => x = 0  
 ln(x) => ln(x) = 0  
 log\_2(x)  
 sin(x)  
 x^2 - x  
 cos(x)^2 + sin(x)^2 - 1  
 cos(x)^2+sin(x)^2-1  
 cos(x\_1)^2)+sin(x\_2)^2 = 0  
   
 нижний индекс поддерживается только у:  
 логарифма  
 икса (сначала идёт индекс, потом степень - x\_1^2  
   
   
 """  
  
 @staticmethod  
 def algebraic\_expression\_input(message=""):  
 line = InputManager.string\_input(message)  
 line = line.replace(" ", "")  
 line = line.replace(",", ".")  
 line = line.replace("^", "\*\*")  
 if line.\_\_contains\_\_('='):  
 raise InvalidAlgebraicEquationException  
 return line  
  
 pass  
  
 @staticmethod  
 def vector\_dict\_input(var\_names, message=""):  
 if message != "":  
 print(message)  
 result = dict()  
 for name in var\_names:  
 result[name] = InputManager.float\_input(f"\tВведите {name}: ")  
 return result  
  
 @staticmethod  
 def enum\_input(variants\_list, message):  
 buf = ""  
 while buf not in variants\_list:  
 buf = InputManager.string\_input(message)  
 return buf  
  
 @staticmethod  
 def multiple\_choice\_input(variant\_list, values\_list, message=""):  
 n = len(variant\_list)  
 if n == 0 or len(variant\_list) != len(values\_list):  
 raise ValueError  
  
 if message != "":  
 print(message)  
  
 for i in range(n):  
 s = f"{i + 1}."  
 lines = variant\_list[i].split('\n')  
 print('\t' + s, lines[0])  
 for line in lines[1:]:  
 print("\t" + ' ' \* len(s), line)  
  
 i = int(InputManager.enum\_input([str(i) for i in range(1, n + 1)], f"Введите число от 1 до {n}: ")) - 1  
 return values\_list[i]

**PreparedEquationManager.py**:

import numpy as np  
from matplotlib import pyplot as plt  
  
from InputManager import InputManager  
  
  
class PreparedEquation:  
  
 def \_\_init\_\_(self, func, derivative\_func=None):  
 self.func = func  
 self.derivative\_func = derivative\_func  
 self.iterations = 0  
  
 def calculate(self, arg):  
 try:  
 return self.func(arg)  
 except Exception:  
 raise ValueError  
  
 def solve(self):  
 variants = ["Метод хорд", "Метод Ньютона", "Метод простых итераций"]  
 values = [self.Chord\_Method, self.Newtons\_Method, self.Simple\_Iteration\_Method]  
 chosen\_method = InputManager.multiple\_choice\_input(variants, values, "Выберите метод решения уравнения: ")  
 return chosen\_method()  
  
 def \_enter\_root\_isolation(self):  
 print("Введите интервал изоляции корня:")  
 left = InputManager.float\_input("\tЛевый конец отрезка: ")  
 right = InputManager.float\_input("\tПравый конец отрезка: ")  
  
 while not self.calculate(left) \* self.calculate(right) < 0:  
 print("Значения функции на концах отрезка должны быть разного знака!")  
 if InputManager.yes\_or\_no\_input(f"Показать график функции для корректировки?"):  
 self.draw\_graphic(left, right)  
 print("Введите другие значения концов:")  
 left = InputManager.float\_input("\tЛевый конец отрезка: ")  
 right = InputManager.float\_input("\tПравый конец отрезка: ")  
 if self.calculate(left) > 0:  
 # функция принимает значение > 0 на правом конце, и < 0 на левом  
 # от left до right функция возрастает  
 left, right = right, left  
 self.left, self.right = left, right  
 return left, right  
  
 def draw\_graphic(self, left=None, right=None, dot\_x=None, dot\_y=None):  
 if left is None or right is None:  
 if self.left is None or self.right is None:  
 print("Невозможно нарисовать график. Не определены концы отрезка")  
 else:  
 left = self.left  
 right = self.right  
 grid = abs(left - right) / 30  
 x\_axis = np.linspace(min(left, right) - grid, max(left, right) + grid, 32)  
 plt.plot(x\_axis, self.calculate(x\_axis))  
 plt.grid(True, which='both')  
 if (min(left, right) - grid <= 0 and 0 <= max(left, right) + grid):  
 plt.axvline(x=0, color='k')  
 plt.axhline(y=0, color='k')  
 if dot\_x is not None:  
 if dot\_y is None:  
 dot\_y = 0  
 plt.plot(dot\_x, dot\_y, "ro")  
 plt.show()  
  
 def Chord\_Method(self):  
 left, right = self.\_enter\_root\_isolation()  
  
 # основная часть решения  
 epsilon = 10 \*\* (-8)  
 x = left - self.calculate(left) \* (right - left) / (self.calculate(right) - self.calculate(left))  
 while abs(self.calculate(x)) > epsilon:  
 x = left - self.calculate(left) \* (right - left) / (self.calculate(right) - self.calculate(left))  
 if self.calculate(x) > 0:  
 right = x  
 else:  
 left = x  
 self.iterations += 1  
  
 return x  
  
 def Simple\_Iteration\_Method(self):  
 left, right = self.\_enter\_root\_isolation()  
 halflife = -0.01 / max(abs(self.derivative\_func(left)), abs(self.derivative\_func(right)))  
 phi = lambda x: x + halflife \* self.calculate(x)  
 print(f"Используемая лямбда={halflife}")  
  
 # основная часть решения  
 epsilon = 10 \*\* (-8)  
 x = InputManager.float\_input("Введите начальное приближение: ")  
 while not (min(left, right) < x and x < max(left, right)):  
 x = InputManager.float\_input("Введите начальное приближение внутри интервала изоляции корня: ")  
 try:  
 self.calculate(x)  
 except Exception:  
 print("Невозможно посчитать значение функции в заданной точке.")  
 return None  
  
  
 while abs(self.calculate(x)) > epsilon and self.iterations <= 1000000:  
 x = phi(x)  
 self.iterations += 1  
 if self.iterations == 1000000:  
 return None  
 return x  
  
 def Newtons\_Method(self):  
 x = InputManager.float\_input("Введите начальное приближение: ")  
  
 epsilon = 10 \*\* (-8)  
 while abs(self.calculate(x)) > epsilon:  
 x = x - self.calculate(x) / self.derivative\_func(x)  
 self.iterations += 1  
  
 return x

**PreparedSystemManager.py**:

import numpy as np  
from matplotlib import cm  
from matplotlib import pyplot as plt  
  
from InputManager import InputManager  
  
  
class PreparedSystem:  
 def \_\_init\_\_(self, funcs, derivative\_funcs=None):  
 *"""  
 funcs: вектор функций  
 derivative\_funcs: матрица частных производных  
 """* self.solutions = None  
 self.funcs = funcs  
 self.derivative\_funcs = derivative\_funcs  
 self.before\_x\_vector = []  
 self.iterations = 0  
 self.x\_left, self.x\_right = None, None  
 self.y\_left, self.y\_right = None, None  
  
 def calculate(self, x, y):  
 return np.array([self.funcs[0](x, y), self.funcs[1](x, y)])  
  
 def calculate\_derivatives(self, \*args):  
 return np.array([f(\*args) for f in self.derivative\_funcs])  
  
 def get\_error\_rate(self):  
 return abs(np.array(self.before\_x\_vector) - self.solutions)  
  
 def \_get\_sufficient\_convergence\_condition(self, x, y):  
 return max(sum(abs(i)) for i in self.calculate\_derivatives(x, y))  
  
 def check\_sufficient\_convergence\_condition(self, x, y):  
 return self.\_get\_sufficient\_convergence\_condition(x, y) < 1  
  
 def solve(self):  
 return self.Simple\_Iteration\_Method()  
  
 def \_enter\_root\_isolation(self):  
 print("Введите интервал изоляции корня:")  
 x\_left = InputManager.float\_input("\tЛевый конец отрезка по x: ")  
 x\_right = InputManager.float\_input("\tПравый конец отрезка по x: ")  
 y\_left = InputManager.float\_input("\tЛевый конец отрезка по y: ")  
 y\_right = InputManager.float\_input("\tПравый конец отрезка по y: ")  
 while not (self.calculate(x\_left, y\_left)[0] \* self.calculate(x\_right, y\_right)[0] < 0 and  
 self.calculate(x\_left, y\_left)[1] \* self.calculate(x\_right, y\_right)[1] < 0):  
 if not (x\_left < x\_right and y\_left < y\_right):  
 print("Значения левых концов должны быть меньше правых.")  
 continue  
 print("Значения функции на углах квадрата должны быть разного знака для обеих функций!")  
 if InputManager.yes\_or\_no\_input(f"Показать график системы для корректировки?"):  
 self.draw\_graphic(x\_left, x\_right, y\_left, y\_right)  
 print("Введите другие значения концов:")  
 x\_left = InputManager.float\_input("\tЛевый конец отрезка по x: ")  
 x\_right = InputManager.float\_input("\tПравый конец отрезка по x: ")  
 y\_left = InputManager.float\_input("\tЛевый конец отрезка по y: ")  
 y\_right = InputManager.float\_input("\tПравый конец отрезка по y: ")  
 self.x\_left, self.x\_right = min(x\_left, x\_right), max(x\_left, x\_right)  
 self.y\_left, self.y\_right = min(y\_left, y\_right), max(y\_left, y\_right)  
  
 return  
  
 def draw\_graphic(self, x\_left=None, x\_right=None, y\_left=None, y\_right=None):  
 if None in [x\_left, x\_right, y\_left, y\_right]:  
 if None in [self.x\_left, self.x\_right, self.y\_left, self.y\_right]:  
 print("Невозможно нарисовать график. Не определены концы отрезка")  
 return  
 else:  
 x\_left = self.x\_left  
 x\_right = self.x\_right  
 y\_left = self.y\_left  
 y\_right = self.y\_right  
 x = np.arange(x\_left, x\_right, abs(x\_left - x\_right) / 30)  
 y = np.arange(y\_left, x\_right, abs(y\_left - y\_right) / 30)  
 x, y = np.meshgrid(x, y)  
 Z = self.calculate(x, y)  
 fig = plt.figure()  
 ax = fig.add\_subplot(projection='3d')  
 # ax.axes.set\_zlim3d(-5, 5)  
 ax.set\_xlabel('X')  
 ax.set\_ylabel('Y')  
 ax.set\_zlabel('Z')  
 ax.plot\_surface(x, y, Z[0], cmap=cm.coolwarm)  
 ax.plot\_surface(x, y, Z[1], cmap="plasma")  
 plt.show()  
  
 def Simple\_Iteration\_Method(self):  
 self.\_enter\_root\_isolation()  
 if InputManager.yes\_or\_no\_input("Показать график системы на введённом квадрате?"):  
 self.draw\_graphic()  
  
 x = InputManager.float\_input("Введите начальное приближение по x: ")  
 y = InputManager.float\_input("Введите начальное приближение по y: ")  
 while not (self.x\_left < x < self.x\_right and self.y\_left < y < self.y\_right):  
 print("Начальное приближение должно быть внутри интервала изоляции корня!")  
 x = InputManager.float\_input("Введите начальное приближение по x: ")  
 y = InputManager.float\_input("Введите начальное приближение по y: ")  
  
 if not self.check\_sufficient\_convergence\_condition(x, y):  
 print("Не выполняется достаточное условие сходимости итерационного процесса.")  
 print("Попробуйте ввести другое начальное приближение.")  
 if not InputManager.yes\_or\_no\_input("Продолжить выполнение?"):  
 return  
  
 # lambda\_x = -max(self.\_get\_sufficient\_convergence\_condition(self.x\_left, self.y\_left),  
 # self.\_get\_sufficient\_convergence\_condition(self.x\_right, self.y\_right))  
 lambda\_x = -1  
 phi\_x = lambda x, y: x + self.calculate(x, y)[0] / lambda\_x  
 # lambda\_y = -max(self.\_get\_sufficient\_convergence\_condition(self.x\_left, self.y\_left),  
 # self.\_get\_sufficient\_convergence\_condition(self.x\_right, self.y\_right))  
 lambda\_y = -1  
 phi\_y = lambda x, y: y + self.calculate(x, y)[1] / lambda\_y  
 epsilon = 10 \*\* (-5)  
 self.before\_x\_vector = [x + 100, y + 100]  
 while max(abs(x - self.before\_x\_vector[0]),  
 abs(y - self.before\_x\_vector[1])) > epsilon and self.iterations <= 100000:  
 self.before\_x\_vector[0], self.before\_x\_vector[1] = x, y  
 x, y = phi\_x(x, y), phi\_y(x, y)  
 self.iterations += 1  
 self.solutions = np.array([x, y])  
 return self.solutions

**EquationManager.py**:

import re  
  
import numpy as np  
from math import sin, cos, asin, acos, tan, atan, log2, log10  
from numpy import log as ln, log10 as lg  
from numpy import log10 as log\_10  
from numpy import log2 as log\_2  
from numpy.lib.scimath import logn  
from numpy.lib.scimath import logn as log\_n  
  
from Exceptions import InvalidVectorException  
from InputManager import InputManager  
  
  
class Equation:  
 equation = ""  
 raw\_equation = ""  
 var\_names = []  
 var\_values = dict([("pi", np.pi), ("e", np.e)])  
 \_valid\_vars\_template = re.compile(r"[a-zA-Z\_]+[0-9]\*")  
 \_valid\_functions\_template = re.compile(r"a?sin|lg|a?cos|a?tan|log\_?(?:2|10)?|logn|pi|e")  
  
 # \_valid\_vars\_template = re.compile(r"[a-zA-Z](?:\_[0-9]\*)?") # другой вариант выделений переменных  
  
 def \_\_init\_\_(self, line):  
 self.equation = line  
 self.raw\_equation = line  
 self.var\_names = self.find\_vars()  
 # print(self.var\_names)  
  
 def find\_vars(self):  
 vars = []  
 for i in self.\_valid\_vars\_template.findall(self.equation):  
 if self.\_valid\_functions\_template.fullmatch(i) is None and i not in vars:  
 vars.append(i)  
 return vars  
  
 def set\_valid\_vars\_template(self, template):  
 self.\_valid\_vars\_template = template  
 self.var\_names = self.\_valid\_vars\_template.findall(self.raw\_equation)  
  
 def get\_var\_list(self):  
 return self.var\_names.copy()  
  
 def \_get\_variable\_value\_with\_match(self, match, x\_vector\_dict):  
 name = match.group(0)  
 value = x\_vector\_dict.get(name, None)  
 if value is None:  
 return name  
 return f'({value})'  
  
 def calculate(self, x\_vector\_dict={}):  
 expression = self.equation  
  
 expression = re.sub(self.\_valid\_vars\_template, lambda m: self.\_get\_variable\_value\_with\_match(m, x\_vector\_dict),  
 expression)  
 expression = re.sub(r"\d+(?:\.\d+)?", lambda x: f'({x.group(0)})', expression)  
 expression = expression.replace(")(", ")\*(")  
 # print(expression)  
 return eval(expression)

**AnyEquationSolver.py**:

from InputManager import InputManager  
  
  
class EquationSolver:  
 @staticmethod  
 def solve(equation):  
 result = EquationSolver.Half\_Division\_Method(equation)  
 if result is None:  
 print("Не удалось найти корень уравнения.")  
 else:  
 print("Корень уравнения:", result)  
  
 @staticmethod  
 def Half\_Division\_Method(equation):  
  
 if len(equation.get\_var\_list()) != 1:  
 print("Для решения методом половинного деления необходимо, "  
 "чтобы в уравнении участвовало только одно неизвестное.")  
 print("Неизвестные в данном уравнении:", equation.get\_var\_list())  
 return  
  
 k = equation.get\_var\_list()[0]  
 a = InputManager.float\_input(f"\tВведите левый край начального приближения: {k} = ")  
 b = InputManager.float\_input(f"\tВведите правый край начального приближения: {k} = ")  
  
 while not (equation.calculate({k: a}) \* equation.calculate({k: b}) < 0):  
 print("Необходимо, чтобы значения функции на концах отрезков были разных знаков!")  
 if not InputManager.yes\_or\_no\_input("Ввести концы заново?"):  
 return  
 a = InputManager.float\_input(f"\tВведите левый край начального приближения: {k} = ")  
 b = InputManager.float\_input(f"\tВведите правый край начального приближения: {k} = ")  
  
 epsilon = 0.001  
 print("Хотите ввести точность? По умолчанию epsilon равно 10^-3.")  
 if InputManager.yes\_or\_no\_input("Ввести epsilon?"):  
 epsilon = InputManager.float\_input("epsilon=")  
  
 if not equation.calculate({k: a}) < 0:  
 a, b = b, a  
 while not abs(a - b) < epsilon:  
 mid = (a + b) / 2  
 if equation.calculate({k: mid}) < 0:  
 a = mid  
 else:  
 b = mid  
  
 return (a + b) / 2

**Exceptions.py**:

class InvalidVectorException(ValueError):  
 def \_\_init\_\_(self, message):  
 # Call the base class constructor with the parameters it needs  
 super().\_\_init\_\_(message)  
  
  
class InvalidAlgebraicEquationException(ValueError):  
 def \_\_init\_\_(self, message):  
 # Call the base class constructor with the parameters it needs  
 super().\_\_init\_\_(message)

1. Результаты выполнения программы при различных исходных данных.

Изображение выглядит как диаграмма

Автоматически созданное описаниеИзображение выглядит как текст

Автоматически созданное описание

1. Выводы

В ходе выполнения лабораторной работы я изучил и реализовал несколько методов вычисления корней. Также отработал навыки создания user-friendly консольных приложений.