МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»**

Кафедра теоретических основ компьютерной безопасности и криптографии

**Решение систем нелинейных уравнений**

ОТЧЕТ ПО ДИСЦИПЛИНЕ

«ЧИСЛЕННЫЕ МЕТОДЫ РЕШЕНИЯ ЗАДАЧ»

студента 4 курса 431 группы

специальности 10.05.01 Компьютерная безопасность

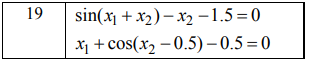
факультета компьютерных наук и информационных технологий

Сенокосова Владислава Владимировича

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
| Преподаватель  Аспирант | \_\_\_\_\_\_\_\_\_\_ | В.M.Шкатов |
|  | подпись, дата |  |

Саратов 2024

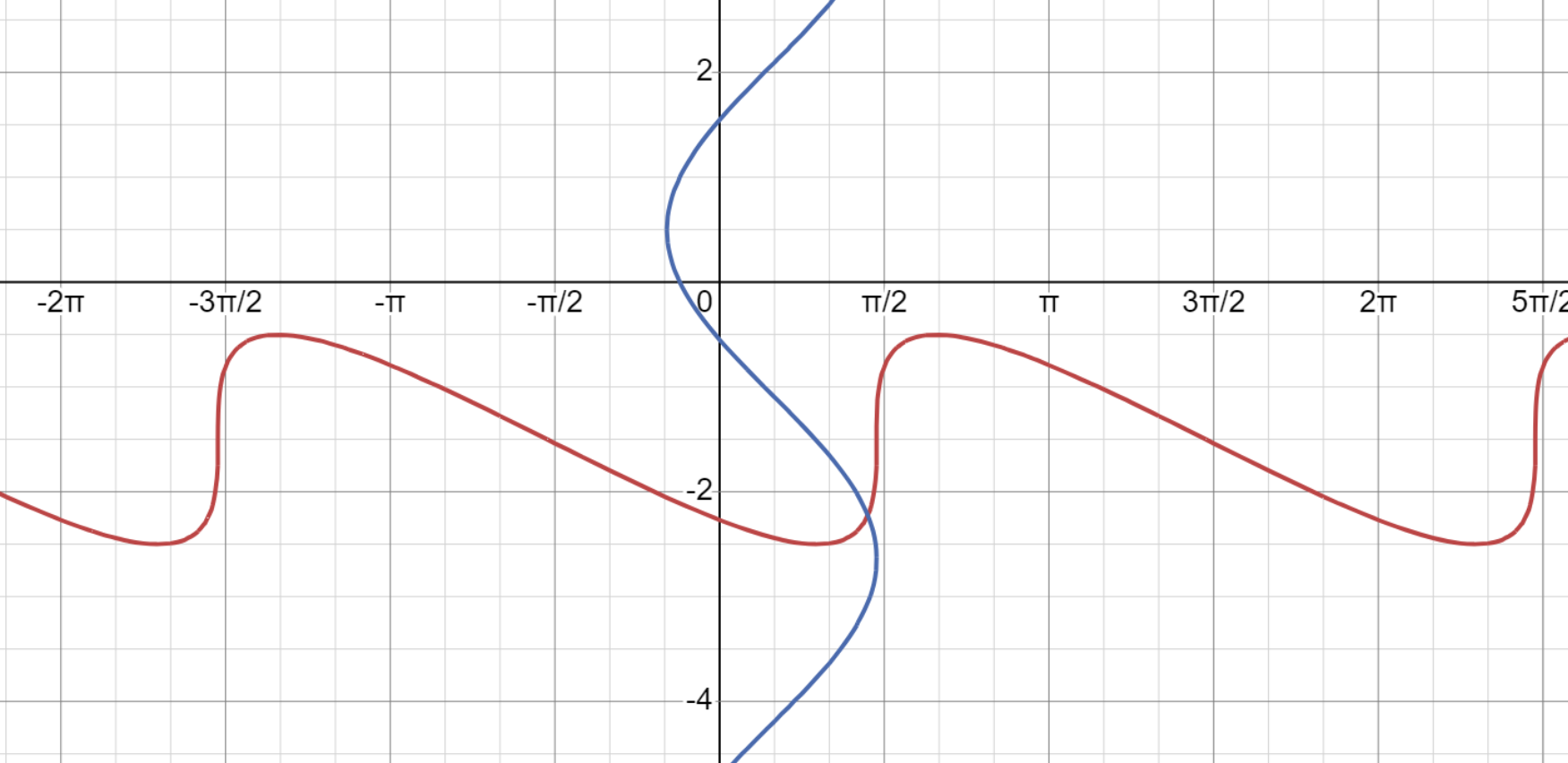
**Вариант 19**

****

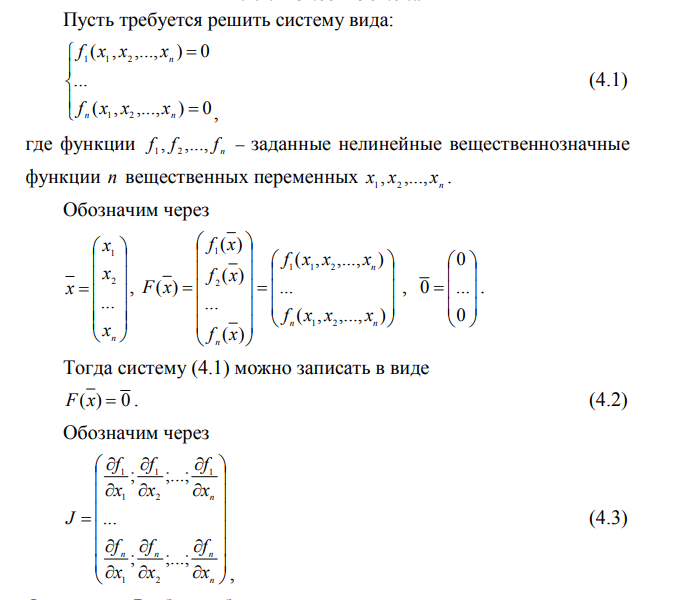
**Задача:**

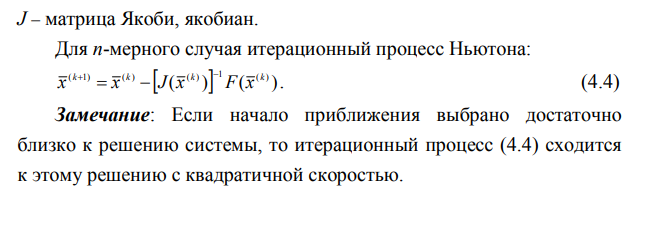
Найти методом Ньютона и наискорейшего спуска все корни системы нелинейных уравнений. При реализации программного кода использовался язык Python с библиотекой sympy, для более удобной работы с линейной алгеброй.

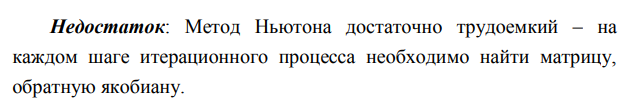
Графическое представление функции имеет следующий вид:



**Задание 1: «Метод Ньютона»**

****

****

****

Функция, реализующая алгоритм поиска корней Ньютона представлена следующим образом:

*# Метод Ньютона нахождения корней*

def method\_newton(params, system, start):

    eps = 0.000001

*# Строим якобиан от параметров x1, x2*

    J = []

    for fun in system:

        buff = []

        for param in params:

            buff.append(sp.diff(fun, param))

        J.append(buff)

*# Получаем необходимые матрицы для расчета*

    J\_x\_inv, F\_x = get\_data\_for\_newton(J, params, system, start)

    approximate\_roots = []

    old\_roots = sp.Matrix(deepcopy(start))

    new\_roots = old\_roots - J\_x\_inv \* F\_x

    approximate\_roots.append((list(old\_roots), None))

    while get\_norm\_vec(new\_roots - old\_roots) > eps:

        J\_x\_inv, F\_x = get\_data\_for\_newton(J, params, system, new\_roots)

        old\_roots = new\_roots

        new\_roots = old\_roots - J\_x\_inv \* F\_x

        approximate\_roots.append((list(old\_roots),

                                 "{:0<20.15f}".format(get\_norm\_vec(new\_roots - old\_roots))))

    return new\_roots, approximate\_roots

Для осуществления итерации с промежуточными значениями использовалась следующая функция:

*# Получаем необходимые данные для текущей итерации методом Ньютона*

def get\_data\_for\_newton(J, params, system, start):

*# Создаем словарь приближенных корней вида {x1: valume1, x2: valume2 ...}*

    lst\_starts = {params[i]: start[i] for i in range(len(params))}

*# Находим значение якбиана в точках x1, x2...*

    J\_x = []

    for index1 in range(len(J)):

        buff = []

        for index2 in range(len(J)):

            val = J[index1][index2].subs(lst\_starts).evalf()

            buff.append(round(val, 7))

        J\_x.append(buff)

*# Обратная матрица*

    J\_x = sp.Matrix(J\_x)

    J\_x\_inv = J\_x.inv()

*# Находим значение системы функций в точках x1, x2 ....*

    F\_x = []

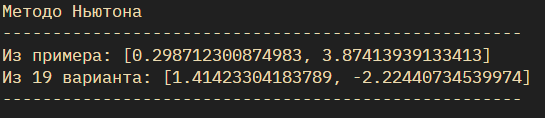
    for i in range(len(system)):

        F\_x.append(system[i].subs(lst\_starts).evalf())

    F\_x = sp.Matrix(F\_x)

    return J\_x\_inv, F\_x

Результаты работы программы для двух примеров (тестового и варианта 19):



Также в качестве проверки найденных корней была написана следующая функция:

*# Функция для проверки найденных корней*

def check\_roots(funcs\_list, lst\_roots, params):

*# Связываем параметры x1, x2 со значениями из params*

    roots\_dict = {params[i]: lst\_roots[i] for i in range(len(params))}

    for i, func in enumerate(funcs\_list):

*# Если значение = 0, то format не сможет преобразовать такое значение*

*# поэтому обрабатываем отдельно*

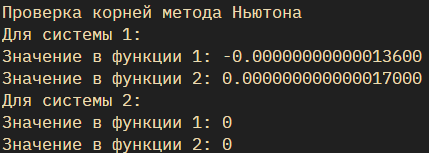
        if func.subs(roots\_dict) != 0:

            print(f"Значение в функции {i + 1}:", "{:0<20.15f}".format(func.subs(roots\_dict)))

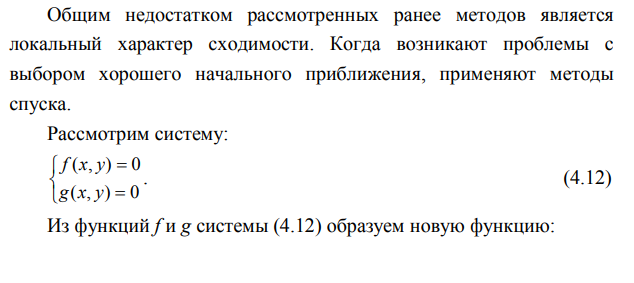
        else:

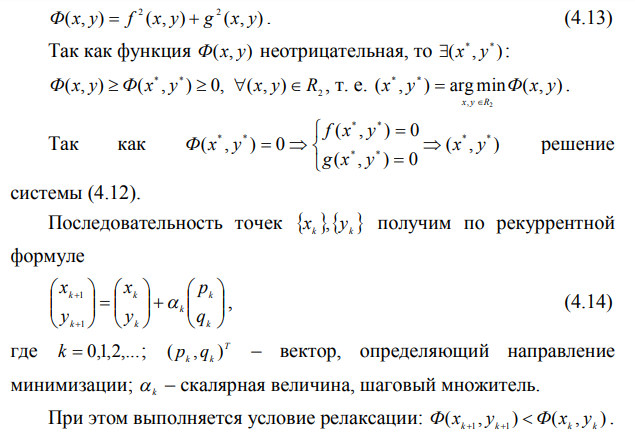
            print(f"Значение в функции {i + 1}:", func.subs(roots\_dict))

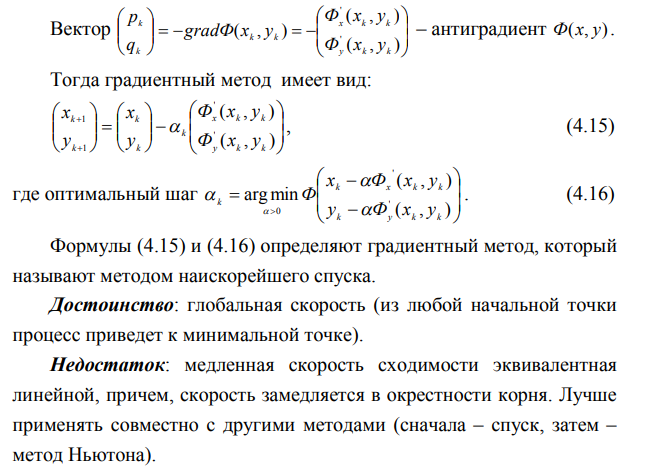
В результате подстановки найденных корней в систему нелинейных уравнений были получены следующие значения:



**Задание 2: «Метод наискорейшего спуска»**

****

****

****

Функция, реализующая алгоритм поиска корней методом наискорейшего спуска представлена следующим образом:

def speed\_down(params, system, start):

    eps = 0.000001

*# Строим новую функцию путем возведения в квадрат исходных функций*

*# и складываем их*

    new\_fun = 0

    for fun in system:

        new\_fun += fun \*\* 2

*# Находим частные производные по x1, x2....*

    diff\_list = []

    for par in params:

        diff\_list.append(sp.diff(new\_fun, par))

*# Берем фиксированный шаг*

    alpha = 0.3

*# Будем собирать приближенные корни на каждом шаге*

    approximate\_roots = []

*# Получаем необходимые данные для итерации*

    val\_func = get\_data\_for\_speed\_down(params, start, diff\_list)

    old\_root = sp.Matrix(start)

    new\_root = old\_root - alpha \* val\_func

    approximate\_roots.append((list(old\_root), None))

    while get\_norm\_vec(new\_root - old\_root) > eps:

        val\_func = get\_data\_for\_speed\_down(params, new\_root, diff\_list)

        old\_root = new\_root

        new\_root = sp.Matrix(old\_root) - alpha \* val\_func

        approximate\_roots.append((list(new\_root), (get\_norm\_vec(new\_root - old\_root))))

    return new\_root, approximate\_roots

Для осуществления итерации с промежуточными значениями использовалась следующая функция:

*# Получаем необходимые данные для текущей итерации методом спуска*

def get\_data\_for\_speed\_down(params, start, diff\_list):

*# Создаем словарь приближенных корней вида {x1: valume1, x2: valume2 ...}*

    lst\_starts = {params[i]: start[i] for i in range(len(params))}

*# Вычисление значения функции от точек x1, x2, ....*

    val\_func = []

    for index in range(len(diff\_list)):

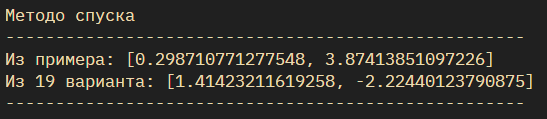
        val = diff\_list[index].subs(lst\_starts).evalf()

        val\_func.append(round(val, 7))

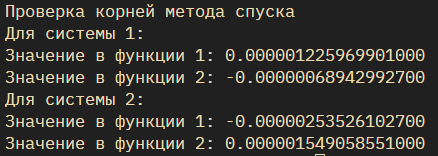
    val\_func = sp.Matrix(val\_func)

    return val\_func

Результаты работы программы для двух примеров (тестового и варианта 19):



В результате подстановки найденных корней в систему нелинейных уравнений были получены следующие значения:



В результате проделанной лабораторной работы были реализованы два способа нахождения корней нелинейных уравнений. Результаты совпадают с заданной точностью. Тестирование производилось по примеру и варианту 19.

**ПРИЛОЖЕНИЕ**

**Листинг программы**

import sympy as sp

from copy import deepcopy

*# Получаем необходимые данные для текущей итерации методом Ньютона*

def get\_data\_for\_newton(J, params, system, start):

*# Создаем словарь приближенных корней вида {x1: valume1, x2: valume2 ...}*

    lst\_starts = {params[i]: start[i] for i in range(len(params))}

*# Находим значение якбиана в точках x1, x2...*

    J\_x = []

    for index1 in range(len(J)):

        buff = []

        for index2 in range(len(J)):

            val = J[index1][index2].subs(lst\_starts).evalf()

            buff.append(round(val, 7))

        J\_x.append(buff)

*# Обратная матрица*

    J\_x = sp.Matrix(J\_x)

    J\_x\_inv = J\_x.inv()

*# Находим значение системы функций в точках x1, x2 ....*

    F\_x = []

    for i in range(len(system)):

        F\_x.append(system[i].subs(lst\_starts).evalf())

    F\_x = sp.Matrix(F\_x)

    return J\_x\_inv, F\_x

*# Метод Ньютона нахождения корней*

def method\_newton(params, system, start):

    eps = 0.000001

*# Строим якобиан от параметров x1, x2*

    J = []

    for fun in system:

        buff = []

        for param in params:

            buff.append(sp.diff(fun, param))

        J.append(buff)

*# Получаем необходимые матрицы для расчета*

    J\_x\_inv, F\_x = get\_data\_for\_newton(J, params, system, start)

    approximate\_roots = []

    old\_roots = sp.Matrix(deepcopy(start))

    new\_roots = old\_roots - J\_x\_inv \* F\_x

    approximate\_roots.append((list(old\_roots), None))

    while get\_norm\_vec(new\_roots - old\_roots) > eps:

        J\_x\_inv, F\_x = get\_data\_for\_newton(J, params, system, new\_roots)

        old\_roots = new\_roots

        new\_roots = old\_roots - J\_x\_inv \* F\_x

        approximate\_roots.append((list(old\_roots),

                                 "{:0<20.15f}".format(get\_norm\_vec(new\_roots - old\_roots))))

    return new\_roots, approximate\_roots

*# Находим норму вектора*

def get\_norm\_vec(vec):

    max\_val = abs(vec[0])

    for val in vec[1:]:

        abs\_val = abs(val)

        if abs\_val > max\_val:

            max\_val = abs\_val

    return max\_val

*# Получаем необходимые данные для текущей итерации методом спуска*

def get\_data\_for\_speed\_down(params, start, diff\_list):

*# Создаем словарь приближенных корней вида {x1: valume1, x2: valume2 ...}*

    lst\_starts = {params[i]: start[i] for i in range(len(params))}

*# Вычисление значения функции от точек x1, x2, ....*

    val\_func = []

    for index in range(len(diff\_list)):

        val = diff\_list[index].subs(lst\_starts).evalf()

        val\_func.append(round(val, 7))

    val\_func = sp.Matrix(val\_func)

    return val\_func

*# Метод спуска*

def speed\_damn(params, system, start):

    eps = 0.000001

*# Строим новую функцию путем возведения в квадрат исходных функций*

*# и складываем их*

    new\_fun = 0

    for fun in system:

        new\_fun += fun \*\* 2

*# Находим частные производные по x1, x2....*

    diff\_list = []

    for par in params:

        diff\_list.append(sp.diff(new\_fun, par))

*# Берем фиксированный шаг*

    alpha = 0.3

*# Будем собирать приближенные корни на каждом шаге*

    approximate\_roots = []

*# Получаем необходимые данные для итерации*

    val\_func = get\_data\_for\_speed\_down(params, start, diff\_list)

*# Стартовые значения*

    old\_root = sp.Matrix(start)

    new\_root = old\_root - alpha \* val\_func

*# Список приближенных корней в процессе итераций*

    approximate\_roots.append((list(old\_root), None))

    while get\_norm\_vec(new\_root - old\_root) > eps:

        val\_func = get\_data\_for\_speed\_down(params, new\_root, diff\_list)

        old\_root = new\_root

        new\_root = sp.Matrix(old\_root) - alpha \* val\_func

        approximate\_roots.append((list(new\_root), (get\_norm\_vec(new\_root - old\_root))))

    return new\_root, approximate\_roots

*# Функция для проверки найденных корней*

def check\_roots(funcs\_list, lst\_roots, params):

*# Связываем параметры x1, x2 со значениями из params*

    roots\_dict = {params[i]: lst\_roots[i] for i in range(len(params))}

    for i, func in enumerate(funcs\_list):

*# Если значение = 0, то format не сможет преобразовать такое значение*

*# поэтому обрабатываем отдельно*

        if func.subs(roots\_dict) != 0:

            print(f"Значение в функции {i + 1}:", "{:0<20.15f}".format(func.subs(roots\_dict)))

        else:

            print(f"Значение в функции {i + 1}:", func.subs(roots\_dict))

if \_\_name\_\_ == "\_\_main\_\_":

*# Тестовые примеры от двух систем с двумя переменными*

    x1 = sp.Symbol("x1")

    x2 = sp.Symbol("x2")

*# Система из примера*

    system\_1 = [

                sp.sin(x1 + 1.5) - x2 + 2.9,

                sp.cos(x2 - 2) + x1

               ]

*# Система варианта 19*

    system\_2 = [

                sp.sin(x1 + x2) - x2 - 1.5,

                x1 + sp.cos(x2 - 0.5) - 0.5

               ]

*#=========================================================*

    print("Методо Ньютона")

    print("----------------------------------------------------")

    root1, data1 = method\_newton([x1, x2], system\_1, [5, 10])

    print("Из примера:", list(root1))

    root2, data2 = method\_newton([x1, x2], system\_2, [5, 10])

    print("Из 19 варианта:", list(root2))

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

*#=========================================================*

*#=========================================================*

    print("Методо спуска")

    print("----------------------------------------------------")

    root3, data3 = speed\_damn([x1, x2], system\_1, [5, 10])

    print("Из примера:", list(root3))

    root4, data4 = speed\_damn([x1, x2], system\_2, [5, 10])

    print("Из 19 варианта:", list(root4))

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

*#==========================================================*

*#==========================================================*

*# Проверка полученных значений*

    print("Проверка корней метода Ньютона")

    print("Для системы 1:")

    check\_roots(system\_1, root1, [x1, x2])

    print("Для системы 2:")

    check\_roots(system\_2, root2, [x1, x2])

    print("Проверка корней метода спуска")

    print("Для системы 1:")

    check\_roots(system\_1, root3, [x1, x2])

    print("Для системы 2:")

    check\_roots(system\_2, root4, [x1, x2])

*#==========================================================*