МИНИСТЕРСТВО НАУКИ И ОБРАЗОВАНИЯ РФ

Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский университет
ИТМО»

ФАКУЛЬТЕТ ПРОГРАММНОЙ ИНЖЕНЕРИИ И КОМПЬЮТЕРНОЙ ТЕХНИКИ

ЛАБОРАТОРНАЯ РАБОТА №3

по дисциплине
ВЫЧИСЛИТЕЛЬНАЯ МАТЕМАТИКА

Вариант – метод Ньютона

Выполнил:
Студент группы P3232
Чмурова Мария
Владиславовна

г. Санкт-Петербург
2024 год

Оглавление

Задание

Дана система нелинейных уравнений. По заданному начальному приближению необходимо найти решение системы с точностью до 5 верного знака после запятой при помощи метода Ньютона.

Формат входных данных:

k

n

x0

y0

...

где k - номер системы, n - количество уравнений и количество неизвестных, а остальные значения - начальные приближения для соответствующих неизвестных.

Формат выходных данных: список такого же типа данных, как списки входных данных, содержащие значения корня для каждой из неизвестных с точностью до 5 верного знака.

Описание метода

Необходимо найти решение СНАУ, используя метод Ньютона.

Для этого нам необходимо взять начальное приближение, которое задает пользователь. Например, (x^0, y^0) для функции двух переменных.

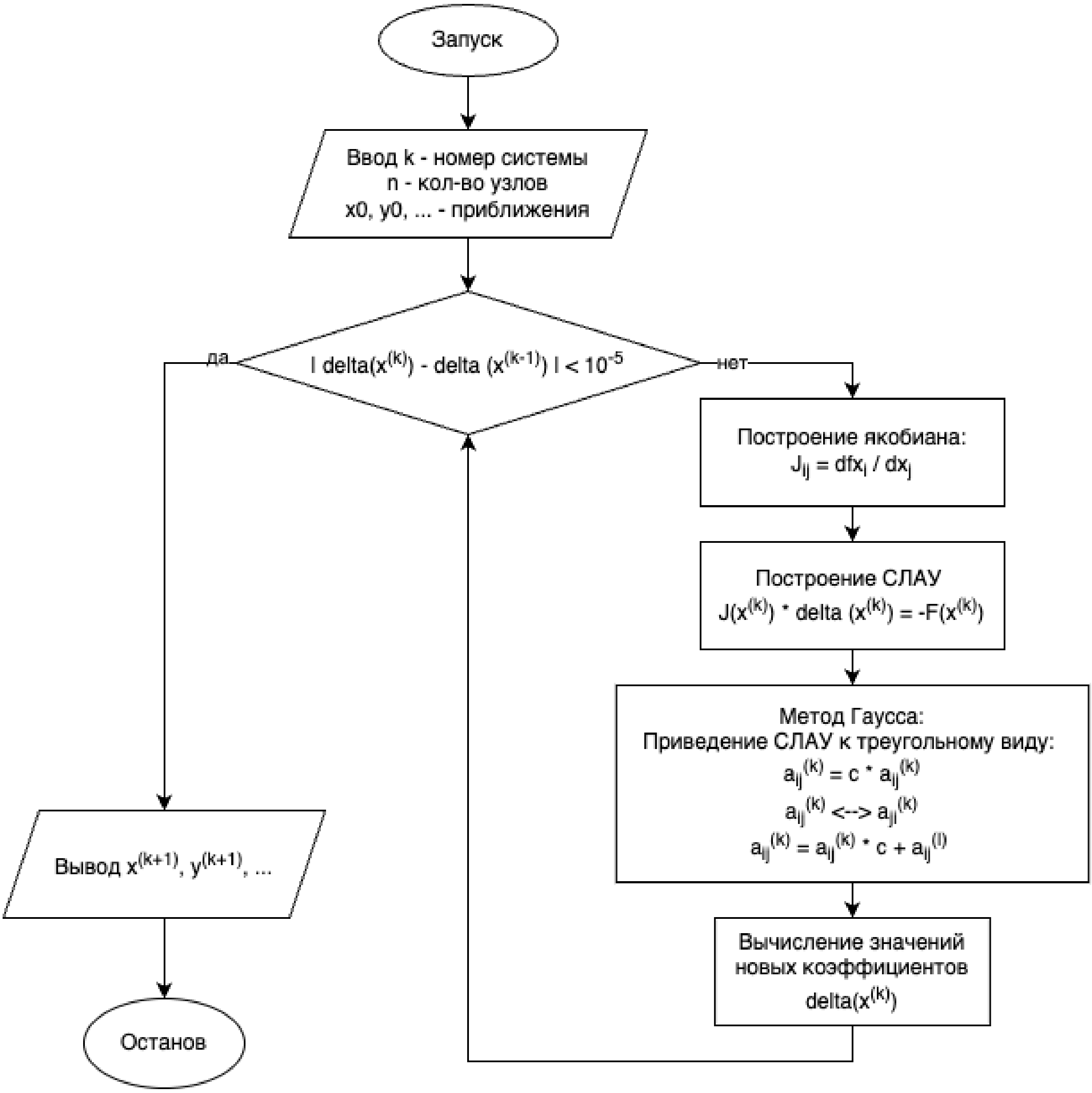
По этим точкам требуется построить матрицу Якоби и вычислить значение всех частных производных в точке приближения.

После этого необходимо решить систему СЛАУ методом Гаусса и найти вектор поправок $\Delta x^{(k)}$ (где k – итерация, на которой взято приближение):

$J\left(x^{(k)}\right) \cdot \Delta x^{(k)} = -F(x^{(k)})$

Если же $|\ \Delta x^{(k)} - \ \Delta x^{(k)}| < 10^{-5}$, то мы достигли необходимой точности и значения вектора поправок являются ответом.

Блок схема


Запуск
Ввод k - номер системы
n - кол-во узлов
x0, y0, ... - приближения
| delta(x^(k)) - delta (x^(k-1)) | < 10^-5
да
нет
Построение якобиана:
J_ij = dfx_i / dx_j
Построение СЛАУ
J(x^(k)) * delta (x^(k)) = -F(x^(k))
Метод Гаусса:
Приведение СЛАУ к треугольному виду:
a_ij^(k) = c * a_ij^(k)
a_ij^(k) <--> a_ji^(k)
a_ij^(k) = a_ij^(k) * c + a_ij^(l)
Вычисление значений
новых коэффициентов
delta(x^(k))
Вывод x^(k+1), y^(k+1), ...
Останов

Код численного метода

```
import math
import os
import random
import re
import sys

k = 0.4
a = 0.9

def first_function(args: []) -> float:
    return math.sin(args[0])

def second_function(args: []) -> float:
    return (args[0] * args[1]) / 2

def third_function(args: []) -> float:
    return math.tan(args[0]*args[1] + k) - pow(args[0], 2)

def fourth_function(args: []) -> float:
    return a * pow(args[0], 2) + 2 * pow(args[1], 2) - 1

def fifth_function(args: []) -> float:
    return pow(args[0], 2) + pow(args[1], 2) + pow(args[2], 2) - 1

def six_function(args: []) -> float:
    return 2 * pow(args[0], 2) + pow(args[1], 2) - 4 * args[2]

def seven_function(args: []) -> float:
    return 3 * pow(args[0], 2) - 4 * args[1] + pow(args[2], 2)

def default_function(args: []) -> float:
    return 0.0

def get_functions(n: int):
    if n == 1:
        return [first_function, second_function]
    elif n == 2:
        k = 0.4
        a = 0.9
        return [third_function, fourth_function]
    elif n == 3:
        k = 0
        a = 0.5
        return [third_function, fourth_function]
    elif n == 4:
        return [fifth_function, six_function, seven_function]
    else:
        return [default_function]

#решение СЛАУ методом Гаусса
class Solution:
```

```
    isSolutionExists = True
    errorMessage = ""

    # Функция для переставления строк при обнаружении нуля на главное
диагонали
    def swap_lines(i, matrix):
        for k in range(i + 1, len(matrix)):
            if matrix[k][i] != 0:
                matrix[i], matrix[k] = matrix[k], matrix[i]
                return matrix
        return matrix

    # Функция для проверки существования решения СЛАУ
    def isSolvable(matrix):
        for row in matrix:
            # Проверка всех элементов кроме последнего (после знака
равно)
            for coefficient in row[:-1]:
                if coefficient != 0:
                    return True
        return False

    #Функция для решения СЛАУ методом Гаусса
    def solveByGauss(n, matrix):

        # Создание копии матрицы
        matrix_copy = [row[:] for row in matrix]

        # Проверка на то, что количество неизвестных равно количеству
уравнений
        for row in matrix:
            if len(row) != n + 1:
                Solution.isSolutionExists = False
                Solution.errorMessage = "The system has no roots of
equations or has an infinite set of them."
                return

        # 1. Приведение матрицы к диагональному (треугольному) виду
        for i in range(n):

            # 1.1 Если элемент на главной диагонали равен нулю, то
необходимо поменять строки местами
            if matrix[i][i] == 0:
                matrix = Solution.swap_lines(i, matrix)
                if matrix[i][i] == 0:
                    Solution.isSolutionExists = False
                    Solution.errorMessage = "The system has no roots
of equations or has an infinite set of them."
                    return

            # 1.2 После переставления высчитываем треугольную матрицу
            for j in range(i + 1, n):
```

```
                # Значение на которое умножаем все элементы следующих
строк
                koef = matrix[j][i] / matrix[i][i]
                for k in range(i, n + 1):
                    matrix[j][k] = matrix[j][k] - koef * matrix[i][k]

        # 2. Проверка на возможность решения (если все коэффициенты в
строке до знака равно обнулились, то решения не существует)
        if (Solution.isSolvable(matrix) == False):
            Solution.isSolutionExists = False
            Solution.errorMessage = "The system has no roots of
equations or has an infinite set of them."
            return

        # 3. Нахождение корней уравнения обратным ходом
        # Массив для хранения решений
        solution = [0] * n
        for i in range(n - 1, -1, -1):
            sum_koef = 0
            for j in range(i + 1, n):
                # Сумма коэффициентов в левой части уравнения
                sum_koef = sum_koef + matrix[i][j] * solution[j]
            solution[i] = round((matrix[i][-1] - sum_koef) /
matrix[i][i], 10)

        # 5. Возвращение полученных значений неизвестных
        return solution

#частная производная
def partial_derivative(func, args: [], index, h=1e-6):
    args_changed = args[:]
    args_changed[index] += h
    return ((func(args_changed) - func(args)) / h)

def solve_by_fixed_point_iterations(system_id, number_of_unknowns,
initial_approximations):

    if ((system_id == 1) or (system_id == 2) or (system_id == 3)) and
(number_of_unknowns != 2):
        return []

    if (system_id == 4) and (number_of_unknowns != 3):
        return []

    if (system_id <= 0) or (system_id > 4):
        return initial_approximations

    #получаем систему уравнений
    system_of_equations = get_functions(system_id)

    flag = True
    for i in range(len(system_of_equations)):
        if (system_of_equations[i](initial_approximations) != 0):
```

```
            flag = False
    if flag == True:
        return initial_approximations


    #найдем якобиан и столбец значений функции
    jacobian = []
    for i in range(len(system_of_equations)):
        matrix = []
        for j in range(len(initial_approximations)):
            matrix.append(partial_derivative(system_of_equations[i],
initial_approximations, j))
        matrix.append(system_of_equations[i](initial_approximations) *
(-1))
        jacobian.append(matrix)

    print(jacobian)

    #решим СЛАУ
    result = Solution.solveByGauss(number_of_unknowns, jacobian)

    result_initial_approximations = []
    if Solution.isSolutionExists:
        for i in range(len(initial_approximations)):

result_initial_approximations.append(initial_approximations[i] +
result[i])
    else:
        return [] #unsolvable by gauss matrix

    #циклим
    while True:
        #проверка точности найденного результата
        flag = True

        for i in range(len(initial_approximations)):
            if abs(result_initial_approximations[i] -
initial_approximations[i]) > 0.00001:
                flag = False

        initial_approximations = result_initial_approximations

        if (flag == True):
            break

        #найдем якобиан и столбец значений функции
        jacobian = []
        for i in range(len(system_of_equations)):
            matrix = []
            for j in range(len(initial_approximations)):

matrix.append(partial_derivative(system_of_equations[i],
initial_approximations, j))
```

```
matrix.append(system_of_equations[i](initial_approximations) * (-1))
            jacobian.append(matrix)

        #решим СЛАУ
        result = Solution.solveByGauss(number_of_unknowns, jacobian)

        result_initial_approximations = []
        if Solution.isSolutionExists:
            for i in range(len(initial_approximations)):

result_initial_approximations.append(initial_approximations[i] +
result[i])
        else:
            return "WM" #unsolvable Gauss Matrix

    for i in range(len(initial_approximations)):
        initial_approximations[i] = round(initial_approximations[i],
5)
    return initial_approximations


if __name__ == '__main__':
    system_id = int(input().strip()) #k - номер системы

    number_of_unknowns = int(input().strip()) #n - кол-во неизвестных

    initial_approximations = []

    for _ in range(number_of_unknowns):
        initial_approximations_item = float(input().strip())
        initial_approximations.append(initial_approximations_item)

    result = solve_by_fixed_point_iterations(system_id,
number_of_unknowns, initial_approximations)

    print('\n'.join(map(str, result)))
    print('\n')
```

Примеры работы программы

№ Теста	Входные данные	Выходные данные
1	1 2 1 1	0.0 1.74417
2	1 4 1 1 1 1	[] Too much variables
3	6 2 0 0	0.0 0.0
4	4 2 1 1	[] Not enough variables
5	4 3 100 150 200	0.7852 0.49661 0.36992

Вывод

В ходе данной лабораторной работы была написана программа, позволяющая найти приближенное значение аргументов нелинейной системы уравнений при помощи метода Ньютона, если решение существует.

Метод Ньютона является наиболее эффективным за счет того, что позволяет достичь минимальной требуемой погрешности за наименьшее количество итераций. Он является особенно эффективным, если выбранное начальное приближение является близким к искомой точке, так как это позволяет сократить количество итераций.

Алгоритмическая сложность данного метода оценивается с помощью Big O Notation как $O(n^2)$ где n – количество неизвестных/уравнений. Данная сложность достигается за счет того, что необходимо вычислять частные производные и выполнять построение матрицы Якоби на каждой итерации.

При использовании данного метода могут возникнуть ошибки во время операций деления или округления, так как ошибки могут накапливаться с каждой итерацией. Особенно такое возможно если начальное приближение далеко от решения. Данная проблема может быть минимизирована правильным выбором начального приближения.

Таким образом, метод Ньютона является эффективным способом нахождения решения СНАУ, обладающий высокой скоростью сходимости при правильном выборе начального приближения.