ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«Национальный исследовательский университет ИТМО»

Факультет программной инженерии и компьютерной техники

Дисциплина:

«Вычислительная математика»

Лабораторная работа №3

«Решение нелинейных уравнений»

Выполнил:

Студент группы P3231

Колмаков Дмитрий Владимирович

Преподаватель:

Перл О.В.

Санкт-Петербург

2024 г.

Оглавление

Задание

Название: **Метод Ньютона**

Описание: 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.

Теория

Метод Ньютона решения СНАУ основан на методе Ньютона решения нелинейных уравнений. В нем поиск нуля функции происходит итерационно, с помощью последовательных приближений. Для начала задается некоторое приближение вблизи корня. Затем в этой точке строится касательная к графику функции, находится ее пересечение с осью абсцисс. Эта точка пересечения становится следующим приближением. Итерация продолжается, пока не будет достигнута заданная точность. Иначе говоря, получаем итеративный метод:

$$x^{(k+1)} = x^{(k)} - F'(x^{(k)})^{-1}F(x^{(k)})$$

Для решения уже системы нелинейных уравнений, мы можем аппроксимировать $F = (F_1, F_2, \dots, F_n)$ с помощью линейной функции. Для этого используем первые два члена из разложения в ряд Тейлора:

$$F(x^{(k+1)}) \approx F(x^{(k)}) + J(x^{(k)})(x^{(k+1)} - x^{(k)}),$$ где J - якобиан F.

Так как мы хотим приблизить $F(x^{(k+1)})$ к нулю, скажем, что:

$$F(x^{(k)}) + J(x^{(k)})(x^{(k+1)} - x^{(k)}) = 0$$

$$F(x^{(k)}) = J(x^{(k)})(x^{(k)} - x^{(k+1)})$$

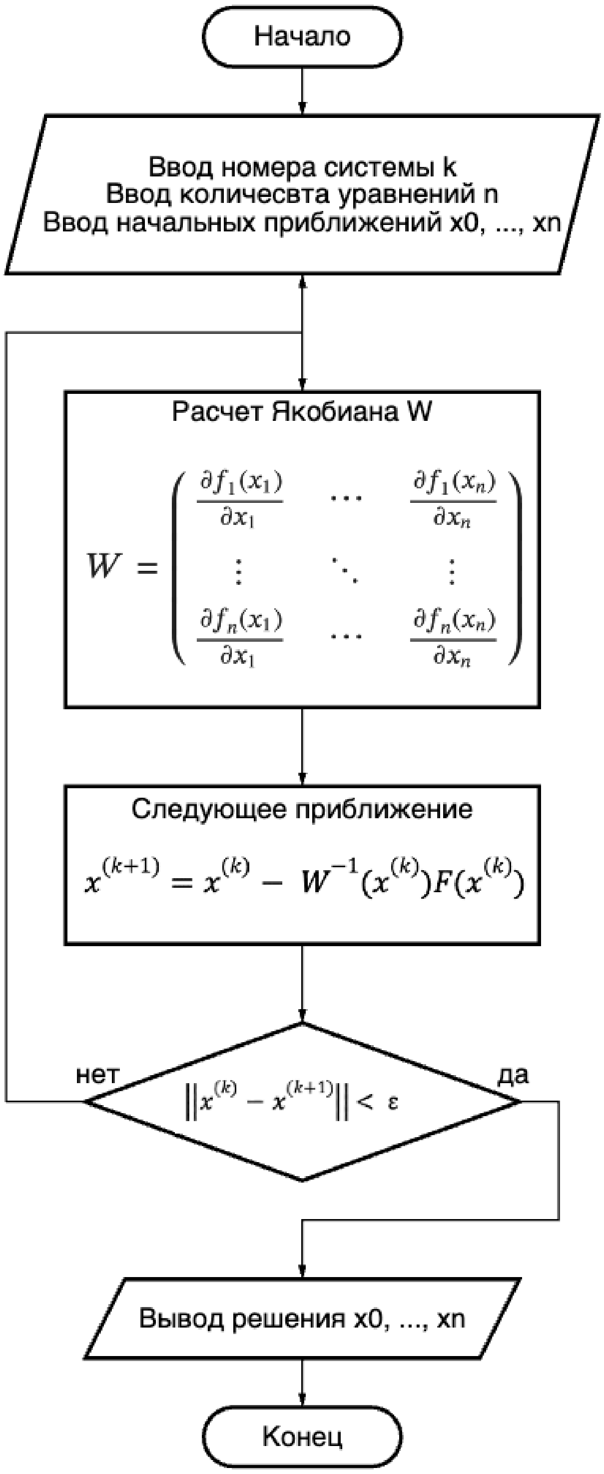
$$J^{-1}(x^{(k)})F(x^{(k)}) = x^{(k)} - x^{(k+1)}$$

$$x^{(k+1)} = x^{(k)} - J^{-1}(x^{(k)})F(x^{(k)})$$

Это и будет рабочей формулой метода. Начальное приближение задается во входных данных. Итерационный процесс останавливается при условии:

$$\left\|x^{(k)} - x^{(k+1)}\right\| < \varepsilon$$

Блок-схема


Начало
Ввод номера системы k
Ввод количесвта уравнений n
Ввод начальных приближений x0, ..., xn
Расчет Якобиана W
$W = \begin{pmatrix} \frac{\partial f_1(x_1)}{\partial x_1} & \cdots & \frac{\partial f_1(x_n)}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n(x_1)}{\partial x_1} & \cdots & \frac{\partial f_n(x_n)}{\partial x_n} \end{pmatrix}$
Следующее приближение
$x^{(k+1)} = x^{(k)} - W^{-1}(x^{(k)})F(x^{(k)})$
нет
$\left\| x^{(k)} - x^{(k+1)} \right\| < \varepsilon$
да
Вывод решения x0, ..., xn
Конец

Реализация программы

```
import math

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]
```

```python
EPSILON = 1e-10


def derivative(f, num: int, args: []):
    delta = EPSILON
    return (f([*args[:num], args[num] + delta, *args[num + 1:]]) -
            f([*args[:num], args[num] - delta, *args[num + 1:]])) / (2 * delta)


def get_triangle_form(matrix, epsilon):
    triangle_matrix = [row[:] for row in matrix]
    count_permutations = 0
    n = len(matrix)
    for i in range(n - 1):
        for j in range(n - 1, i, -1):
            if triangle_matrix[j][i] == 0:
                continue
            else:
                try:
                    ratio = triangle_matrix[j][i] / triangle_matrix[j - 1][i]
                except ZeroDivisionError:
                    triangle_matrix[j], triangle_matrix[j - 1] = (
                        triangle_matrix[j - 1], triangle_matrix[j])
                    count_permutations += 1
                    continue
                for k in range(len(triangle_matrix[j])):
                    triangle_matrix[j][k] = (triangle_matrix[j][k] -
                                             ratio * triangle_matrix[j - 1][k])
                    if abs(triangle_matrix[j][k]) < epsilon ** 2:
                        triangle_matrix[j][k] = 0
    return triangle_matrix, count_permutations


def count_determinant(matrix):
    determinant = 1
    triangle_matrix, count_permutations = get_triangle_form(matrix, EPSILON)
    for i in range(len(matrix)):
        determinant *= triangle_matrix[i][i]
    return determinant * (-1) ** (count_permutations != 0)


def get_transposed_matrix(matrix):
    return list(map(lambda x: list(x), zip(*matrix)))


def get_matrix_minor(matrix, i, j):
    return [row[:j] + row[j + 1:] for row in (matrix[:i] + matrix[i + 1:])]


def get_regularized_matrix(matrix, regularization_param):
    modified_matrix = [[matrix[i][j] + regularization_param if i == j else
matrix[i][j] for j in range(len(matrix[0]))]
                       for i in range(len(matrix))]
    return modified_matrix


def get_inverse_matrix(matrix):
    determinant = count_determinant(matrix)
    if determinant == 0:
```

```
        return get_inverse_matrix(get_regularized_matrix(matrix, 1))
    if len(matrix) == 2:
        return [[matrix[1][1] / determinant, -matrix[0][1] / determinant],
                [-matrix[1][0] / determinant, matrix[0][0] / determinant]]

    cofactors = []
    for r in range(len(matrix)):
        cofactor_row = []
        for c in range(len(matrix)):
            minor = get_matrix_minor(matrix, r, c)
            cofactor_row.append(((-1) ** (r + c)) * count_determinant(minor))
        cofactors.append(cofactor_row)
    cofactors = get_transposed_matrix(cofactors)
    for r in range(len(cofactors)):
        for c in range(len(cofactors)):
            cofactors[r][c] = cofactors[r][c] / determinant
    return cofactors


def multiply_matrix_by_matrix(matrix_a, matrix_b):
    if len(matrix_a[0]) != len(matrix_b):
        return None
    transposed_b = get_transposed_matrix(matrix_b)
    return [[sum(el_a * el_b for el_a, el_b in zip(row_a, col_b))
             for col_b in transposed_b] for row_a in matrix_a]


def multiply_matrix_by_vector(matrix, vector):
    if len(matrix[0]) != len(vector):
        return None
    result_vector = []
    for i in range(len(matrix[0])):
        summ = 0
        for j in range(len(vector)):
            summ += vector[j] * matrix[i][j]
        result_vector.append(summ)
    return result_vector


def distance(a, b):
    return math.sqrt(sum([(x1 - x2) ** 2 for x1, x2 in zip(a, b)]))


def          solve_by_fixed_point_iterations(system_id,          number_of_unknowns,
initial_approximations):
    system = get_functions(system_id)
    X = initial_approximations
    while True:
        X_old = X.copy()
        F = [f(X_old) for f in system]
         W = [[derivative(system[i], j, X) for j in range(number_of_unknowns)] for i
in range(len(system))]
        W_inverse = get_inverse_matrix(W)
        P = multiply_matrix_by_vector(W_inverse, F)
        X = [X_old[i] - P[i] for i in range(len(X))]
        if distance(X, X_old) < EPSILON:
            break

    return X
```

```python
if __name__ == '__main__':
   system_id = int(input().strip())

   number_of_unknowns = int(input().strip())

   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')
```

Тесты

Тест 1

Первая система

Входные данные:
1
2
3
5

Выходные данные:
3.141592653589793
0.0

Тест 2

Вторая система

Входные данные:
2
2
0
0

Выходные данные:
-0.3845561836821676
0.6583710532187097

Тест 3

Третья система

Входные данные:
3
2
10000
-10000

Выходные данные:
0.9281399820811074
0.33518693015706447

Тест 4

Четвертая система

Входные данные:
4

3
-10
10
10

Выходные данные:
-0.7851969330623553
0.4966113929446564
0.3699228307458724

Тест 5

Четвертая система, плохое начальное приближение

Входные данные:
4
3
100000
100000
100000

Выходные данные:
0.7851969330623553
0.4966113929446564
0.36992283074587234

Вывод

В ходе проделанной работы я повторил общие сведения о нелинейных уравнениях, СНАУ и их решении. Изучил различные методы решения нелинейных уравнений - методы половинного деления, хорд, Ньютона, простой итерации. Также изучил методы решения СНАУ - методы Ньютона, итерации, градиентный спуск. Подробно изучил метод Ньютона. Работая над ним, выяснил, что:

- основная сложность алгоритма заключается в нахождении обратной матрицы Якоби, вычислительная сложность которого составляет $O(n^3)$;
- если алгоритм сойдется за L итераций, то итоговая сложность будет $O(L \cdot n^3)$;
- в алгоритме необходимо находить обратную матрицу Якоби, однако ее определитель может быть равен нулю. В таком случае придется либо использовать псевдообратную матрицу, либо пользоваться другим методом решения СНАУ;
- вследствие большой вычислительной сложности данный метод будет работать медленно с матрицами большого размера;
- у градиентного спуска, в отличие от метода Ньютона, лучше алгоритмическая сложность: $O(L \cdot n^2)$, но сходимость не гарантируется.

Также мне удалось реализовать следующие численные методы:

- для нахождения производной;
- для приведения матрицы к верхнетреугольному виду;
- для вычисления определителя матрицы;
- для получения транспонированной матрицы;
- для получения минора матрицы;
- для получения обратной матрицы;
- для умножения матрицы на матрицу;
- для умножения матрицы на вектор;
- для нахождения нормы вектора.

А также реализовал численный метод, позволяющий решать СНАУ с заданной точностью методом Ньютона. На тестовых данных моя реализация работает корректно, находит правильное решение с заданной точностью.