Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей
Кафедра информатики
Дисциплина: Методы численного анализа

ОТЧЁТ

к лабораторной работе

на тему

Решение систем нелинейных уравнений

Выполнил: студент группы 053506
Слуцкий Никита Сергеевич

Проверил: Анисимов Владимир Яковлевич

Минск 2022

Оглавление

Вариант 7 (Номер в журнале – 21)

Цели выполнения задания:

- Изучить численное решение систем нелинейных уравнений методами простых итераций и Ньютона
- Провести отделение решений, построить и запрограммировать алгоритмы методов, численно решить тестовое задание, сравнить трудоёмкость методов.

Краткие теоретические сведения:

Пусть дана система

$$f_1(x_1, x_2, ..., x_n) = 0$$
$$...$$
$$f_n((x_1, x_2, ..., x_n) = 0$$

Определю начальное приближение к корням системы графическим способом.
Чтобы применить метод простой итерации запишу систему в виде

$$x_1 = \phi_1(x_1, x_2, ..., x_n)$$
$$...$$
$$x_n = \phi_n((x_1, x_2, ..., x_n)$$

То есть выражу из i-го уравнения i-ю переменную. Обозначу вектор значений $(x_1, x_2, ..., x_n)$ как x. Тогда пусть начальные значения (начальный вектор) будет обозначен как x^0. Итерационный процесс будет сходиться по формулам (ну как будет, нужно ещё проверить достаточный признак сходимости метода простой итерации): $x^{i+1} = \phi\ (x^i)$

(по аналогии с вектором для x я взял ф – как столбец с функциями ϕ_i). Это касательно метода простой итерации.

Теперь про метод Ньютона.

Пусть имеется начальное приближение x^0 к решению и исходя из него построены приближения x^i. В ходе замены функций f_i на линейные части их разложения по формуле Тейлора в точках x^n можно прийти к системе линейных алгебраических уравнений, имеющей вид

$$f(x^n) + f'(x^n)*(x^{n+1} - x^n) = 0.$$

Где под f' подразумевается матрица Якоби.

Выразив x^{n+1}, можно получить:

$$x^{n+1} = x^n - f'(x^n)^{-1} * f(x^n)$$

Квадратичная скорость сходимости метода Ньютона позволяет использовать простой практический критерий окончания:

$| x^n - x^{n-1} | < E$ (заданная точность).

Теперь про трудоёмкости. Возникает проблема вычисления на начальном этапе матрицы Якоби из большого количества (n^2) частных производных, где вычисление одной лишь частной производной в сложной функции может быть сложной для ЭВМ задачей. Также обостряется проблема нахождения хорошего начального приближения. Беря точки по графику из одной и той же окрестности можно как успешно заставить метод сходиться, так и заставить его не сойтись.

Задание

Решить систему нелинейных уравнений:

$$\begin{cases} tg(xy + m) = x \\ ax^2 + 2y^2 = 1, \end{cases} \quad \text{где } x>0,\ y>0,$$

с точностью до 0.0001 методами простых итераций и Ньютона, принимая для номера варианта k значения параметров a и m из таблицы:

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14
m	0,0	0,1	0,1	0,2	0,2	0,3	0,3	0,4	0,4	0,1	0,2	0,3	0,2	0,2
a	0,5	0,6	0,7	0,8	0,9	1,0	0,5	0,6	0,7	0,8	0,9	1,0	0,7	0,5

Начальные приближения найти графически. Сравнить скорость сходимости методов.

Программная реализация:

Ниже можно ознакомиться с программной реализацией главных функций по триангуляции, обратному ходу, а также вызов этого всего из главного файла. Вспомогательные функции и перегруженные операторы (реализации) опущены. Реализация на языке программирования Python с использованием библиотеки SymPy.

```python
import sympy
from system_solvers.general_utils import get_norm_of_root_vectors_difference


def get_jacobi_matrix(system: tuple, variables_list: list) -> sympy.Matrix:
    response: sympy.Matrix = sympy.Matrix([[0, 0], [0, 0]])

    for row_index, equation in enumerate(system):
        for col_index, variable in enumerate(variables_list):
            response[row_index, col_index] = sympy.diff(equation, variable)

    return response


def solve_non_linear_system_by_newton_method(system: tuple, variables_list: list, epsilon: float) -> (sympy.Matrix, int):
    matrix: sympy.Matrix = sympy.Matrix(list(system[0]))
    jacobi_matrix: sympy.Matrix = get_jacobi_matrix(system[0], variables_list)

    current_iteration: sympy.Matrix = sympy.Matrix([*system[1]])
    sympy.Matrix([*system[1]])

    difference: float = 1.00
    iterations_count: int = 0

    while difference > epsilon:
        previous_iteration: sympy.Matrix = sympy.Matrix(current_iteration)

        current_iteration = previous_iteration - jacobi_matrix.subs({
            variables_list[0]: previous_iteration[0],
            variables_list[1]: previous_iteration[1]
        }).inv(method='LU') * matrix.subs({
            variables_list[0]: previous_iteration[0],
            variables_list[1]: previous_iteration[1]
        })

        difference = get_norm_of_root_vectors_difference(current_iteration, previous_iteration)
        iterations_count += 1

    return current_iteration, iterations_count
```

```python
import sympy
from system_solvers.general_utils import get_norm_of_root_vectors_difference


def solve_non_linear_system_by_simple_iterations_method(transformed_system: tuple,
                                                        variables_list: list,
                                                        epsilon: float) -> (sympy.Matrix, int):
    system: sympy.Matrix = sympy.Matrix(list(transformed_system[0]))
    print(list(transformed_system[0]))
```

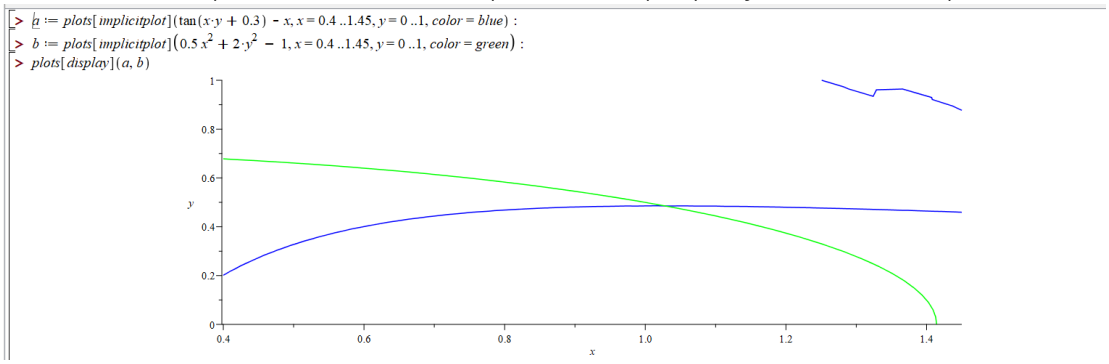
```
    current_iteration: sympy.Matrix = sympy.Matrix(list(transformed_system[1]))

    iterations_count: int = 0

    error: float = 1.00

    while error > epsilon:
        previous_iteration: sympy.Matrix = sympy.Matrix(current_iteration)

        for counter in range(len(current_iteration)):
            current_iteration[counter] = system[counter].subs({
                variables_list[0]: previous_iteration[0],
                variables_list[1]: previous_iteration[1]
            })

        error = get_norm_of_root_vectors_difference(current_iteration, previous_iteration)
        iterations_count += 1

    return current_iteration, iterations_count
```

Полученные результаты:

Для уравнения из варианта 7 метод Ньютона дал следующий результат

```
Solutions by Newton method
Matrix([[1.02798070905978], [0.485606886967462]]) 3
```

Начальное приближение выбрано по графику из СКА Maple.



Тестовые задания

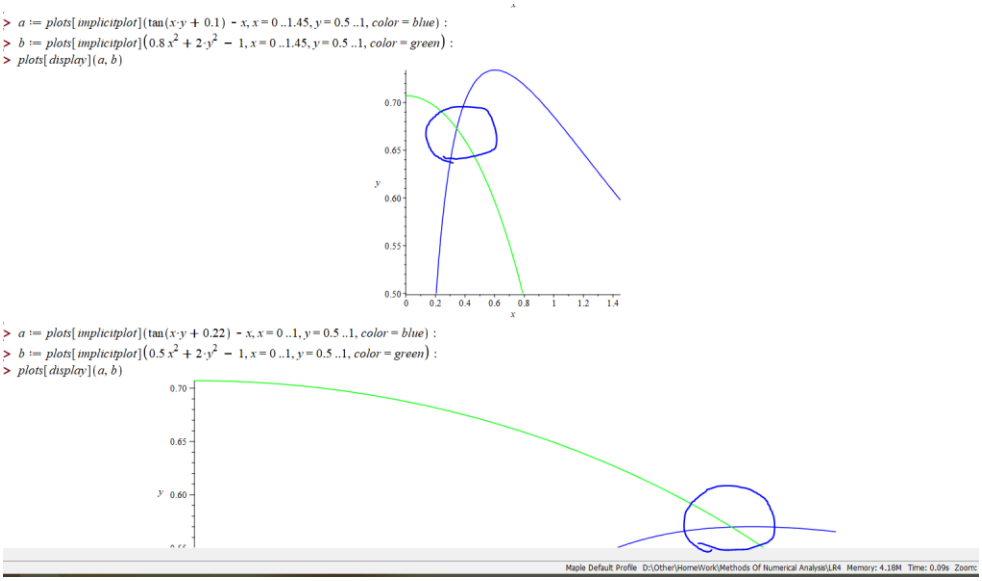
Были решены уравнения с другими коэффициентами. В файле с данными был задан целый массив с примерами и начальными приближениями. Его начало представлено на рисунке:

```
# ((system), (initial approximation by chart)
TEST_SUITE_FOR_NEWTON_METHOD: list[tuple] = [
    ((tan(X * Y + 0.3) - X, 0.5 * (X ** 2) + 2 * (Y ** 2) - 1), (0.8, 0.5)),
    ((0.8 * X ** 2 + 2 * Y ** 2 - 1, tan(X * Y + 0.1) - X), (0.2, 0.5)),
    ((tan(X * Y + 0.22) - X, 0.5 * (X ** 2) + 2 * (Y ** 2) - 1), (0.6, 0.5))
```

Метод Ньютона, например, выдал следующие ответы на данные тестовые задания

```
Solutions by Newton method
Matrix([[1.02798070905978], [0.485606886967462]]) 3
Matrix([[0.344428559812184], [0.672716573863741]]) 4
Matrix([[0.837493545212610], [0.569781673264268]]) 4
```

И также для обоснования выбранных начальных приближений прикрепляю фото первых двух построенных изображений



Выводы

Если оценивать метод Ньютона по числу итераций, то, по идее, можно сделать вывод, что его стоит применять всегда, когда он сойдётся. На практике для достижения разумной точности при выборе достаточно хорошего стартового приближения обычно требуется 3-6 итераций. В тестовых примерах у себя я и наблюдал это “обещанное” в литературе число итераций. В своём программном продукте я НЕ проверял условия сходимости итерационных процессов, а также НЕ использовал какие-либо методы для нахождения начальных приближений. Как минимум, потому что в источниках этот момент опускается. Цели работы можно считать достигнутыми.