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

Вычисление собственных значений и векторов

Выполнил: студент группы 053506

Слуцкий Никита Сергеевич

Проверил: Анисимов Владимир Яковлевич

Минск 2022

**Вариант 8 (Номер в журнале – 21)**

**Цели выполнения задания:**

Освоить методы вычисления собственных значений и векторов.

**Краткие теоретические сведения:**

Пусть дана квадратная матрица **A**. Собственным вектором данной матрицы называется вектор **b**, для которого: **A \* b = c \* b**, где **c** — это некоторая константа. То есть вектор является собственным для матрицы, если при умножении матрицы на этот вектор получается сам же этот вектор и всплывает какая-то константа. Это число **с** называется собственным значением данной матрицы.

Часто требуется получение всех собственных значений и соответствующих им собственных векторов. Методы решения проблемы собственных значений делятся на прямые и итерационные.

Ниже будет рассмотрен итерационный метод Якоби, так как именно его необходимо было реализовать в программном продукте.

Дана симметричная относительно главной диагонали матрица **A**.Необходимо получить её собственные значения и векторы с точностью до какого-то **Е**. Существует алгоритм. Исходя из него, создаётся итерационная последовательность, которая принимает вид: **Ak+1 = (Vk)-1 \* Ak \* Vk**.Матрица **Vk** — это матрица плоского поворота.   
Как же построить такой процесс ?

1. Ищу максимальный по модулю недиагональный элемент в матрице Ak. А точнее – номера его строки и столбца. Обозначу как q, w.
2. Вычисляю синус и косинус угла поворота по формулам:

Если разность между диагональными элементами aqq – aww равна нулю, то вместо угла **u** беру значение π/4. Следовательно, синус и косинус уходят в значение **0.5 \* корень(2).**

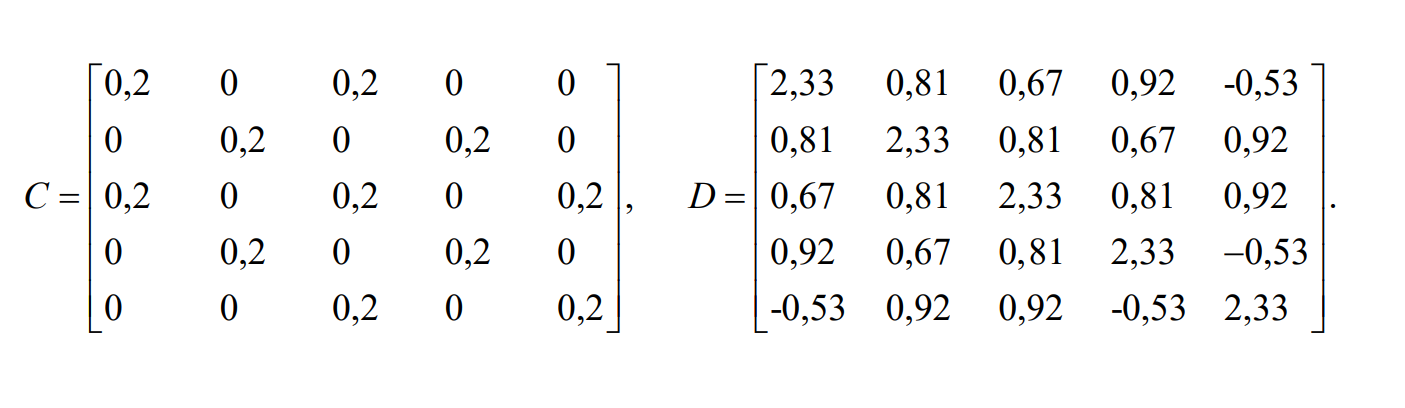
1. Строю единичную матрицу такой же размерности. Помещаю на позиции [w, w] и [q, q] значения вычисленного на прошлом этапе косинуса. На место [w, q] значение -sin(u), на место [q, w] - значение sin(u). **ПРЕДПОЛАГАЕТСЯ, ЧТО максимальный по модулю элемент был найден НАД главной диагональю и w < q.** Получившаяся матрица и есть обозначенное выше **V**.
2. Нахожу обратную к **V** матрицу. Некоторые источники пишут, что обратная в данном случае будет равна транспонированной. Так и есть.
3. Нахожу следующую **Ak+1** перемножая: **Ak+1 = (Vk)-1 \* Ak \* Vk**.

До каких пор повторять процесс ? Для ответа нужно ввести некоторую функцию от матрицы **T(A)**. Она вычисляет сумму квадратов элементов НЕ с главной диагонали (всех, кроме диагональных). Так вот процесс повторяется до тех пор, пока **T(A) >=** нашего **E**.

В конце в качестве собственных значений возьму диагональные элементы матрицы **An**. А чтобы получить собственные вектора, необходимо перемножить все получившиеся **Vk** и разбить на вектор-столбцы. Получим пары: вектор столбец из этого произведения + соответствующий диагональный элемент из матрицы **An**.

Что же вообще произошло ? С каждой итерацией сумма квадратов элементов становится меньше. При этом при каждой итерации происходит “исключение” большего по модулю элемента. Умножение на матрицу поворота построенную по этому элементу обращает в ноль его, а также, возможно, числа на одной вертикали и горизонтали.

**Задание**

С точностью до 0.0001 вычислить собственные значения и собственные векторы матрицы **A**, где **A = k \* C + D**, где **k** – номер варианта, а **C и D**:   


**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 NumPy.

**from utils import get\_matrix\_according\_to\_my\_variant  
import data  
from eigenvalues\_by\_jacobi import get\_eigen\_values\_and\_vectors\_by\_jacobi\_method  
  
  
def main() -> None:  
 for variant in data.VARIANT:  
 matrix = get\_matrix\_according\_to\_my\_variant(variant)  
 values, vectors, iterations = get\_eigen\_values\_and\_vectors\_by\_jacobi\_method(matrix, data.ERROR)  
   
 print(f'Variant {variant}')  
 print('Values:\n', values)  
 print('Vectors:')  
 for vector in vectors:  
 print(vector)  
 print(f'Number of iterations: {iterations}\n')**

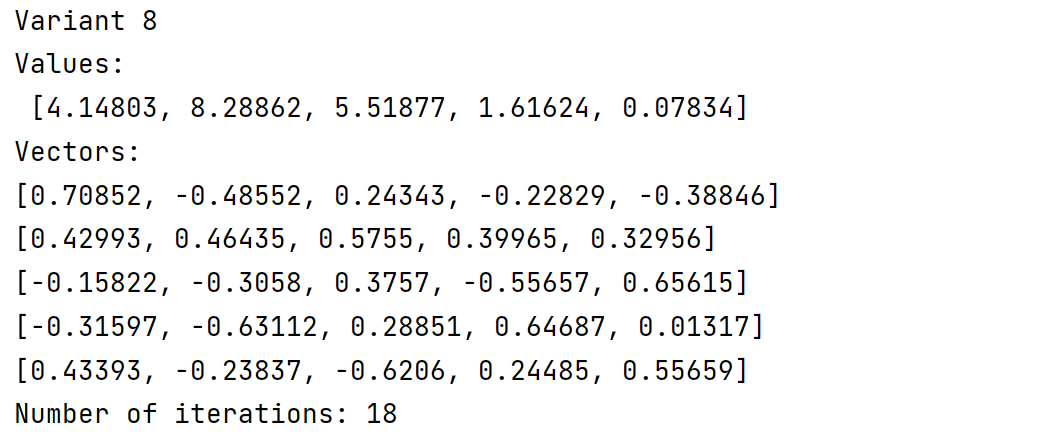
**if \_\_name\_\_ == '\_\_main\_\_':  
 main()**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

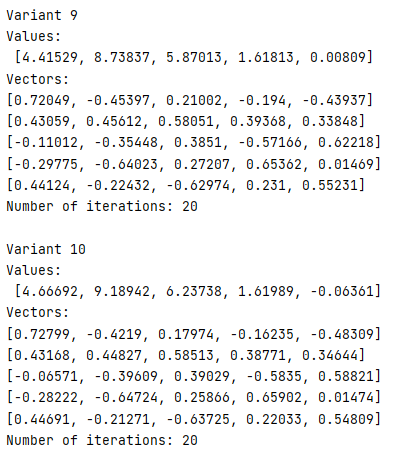
**import math  
import numpy as np  
  
import data  
  
  
def find\_max\_absolute\_non\_diagonal\_element(matrix: np.matrix) -> (int, int):  
 max\_element: float = matrix[0, 1]  
 response\_row, response\_col = 0, 1  
 size: int = len(matrix)  
  
 for row in range(size):  
 for col in range(size):  
 if row != col:  
 if abs(matrix[row, col]) > max\_element:  
 max\_element = abs(matrix[row, col])  
 response\_row, response\_col = row, col  
  
 if response\_row > response\_col:  
 response\_row, response\_col = response\_col, response\_row  
  
 return response\_row, response\_col  
  
  
def get\_matrix\_of\_rotation(matrix: np.matrix, row\_max: int, col\_max: int) -> np.matrix:  
 response: np.matrix = np.matrix(np.eye(len(matrix)))  
  
 a\_ii: float = matrix[row\_max, row\_max]  
 a\_jj: float = matrix[col\_max, col\_max]  
 a\_ij: float = matrix[row\_max, col\_max]  
  
 if a\_ii != a\_jj:  
 p\_k: float = (2 \* a\_ij) / (a\_ii - a\_jj)  
 p\_k\_2: float = p\_k \*\* 2  
  
 cosine = math.sqrt(0.5 \* (1 + (1 + p\_k\_2) \*\* -0.5))  
 sinus = np.sign(p\_k) \* math.sqrt(0.5 \* (1 - (1 + p\_k\_2) \*\* -0.5))  
 else:  
 *# if a[i][i] == a[j][j] we take sin(Pi / 4) and cos(Pi / 4)* cosine = math.sqrt(2) / 2  
 sinus = math.sqrt(2) / 2  
  
 response[row\_max, row\_max] = cosine  
 response[col\_max, col\_max] = cosine  
  
 response[row\_max, col\_max] = -1 \* sinus  
 response[col\_max, row\_max] = sinus  
  
 return response  
  
  
def get\_sum\_of\_squares\_of\_non\_diagonal\_elements(matrix: np.matrix) -> float:  
 size: int = len(matrix)  
 response: float = 0.00  
  
 for row in range(size):  
 for col in range(size):  
 if row != col:  
 response += matrix[row, col] \*\* 2  
  
 return response  
  
  
*# in result matrix we have eigenvalues on diagonal*def get\_total\_eigenvalues\_of\_totally\_rotated\_matrix(result\_matrix: np.matrix) -> list[float]:  
 response: list[float] = list()  
  
 for row in range(len(result\_matrix)):  
 response.append(result\_matrix[row, row])  
  
 return response  
  
  
*# multiplies all matrices of rotation and extracts columns to get eigenvectors*def get\_total\_eigenvectors(rotation\_matrices: list[np.matrix]) -> list[list[float]]:  
 multiplication\_result: np.matrix = np.matrix(rotation\_matrices[0])  
  
 for counter in range(1, len(rotation\_matrices)):  
 multiplication\_result = np.matrix(np.matmul(multiplication\_result, rotation\_matrices[counter]))  
  
 response: list[list[float]] = list()  
  
 for col in range(len(multiplication\_result)):  
 column: list[float] = list()  
  
 for row in range(len(multiplication\_result)):  
 column.append(multiplication\_result[row, col])  
  
 response.append(column)  
  
 return response  
  
  
*# for printing necessary number of digits after dot*def prettify\_response(eigenvalues: list[float], eigenvectors: list[list[float]]) -> None:  
 for counter in range(len(eigenvalues)):  
 eigenvalues[counter] = round(eigenvalues[counter], data.ACCURACY)  
  
 for vector in eigenvectors:  
 for counter in range(len(vector)):  
 vector[counter] = round(vector[counter], data.ACCURACY)  
  
  
*# returns (list with eigen values + list with eigen vectors + number of iterations)*def get\_eigen\_values\_and\_vectors\_by\_jacobi\_method(source: np.matrix, error: float) -> (list[float], list, int):  
 matrix: np.matrix = np.matrix(source)  
 for\_eigenvectors: list[np.matrix] = list()  
 iterations\_count: int = 0  
  
 while get\_sum\_of\_squares\_of\_non\_diagonal\_elements(matrix) > error:  
 row\_max, col\_max = find\_max\_absolute\_non\_diagonal\_element(matrix)  
  
 rotation\_matrix: np.matrix = get\_matrix\_of\_rotation(matrix, row\_max, col\_max)  
 reversed\_rotation\_matrix: np.matrix = np.linalg.inv(np.matrix(rotation\_matrix))  
  
 matrix = np.matrix(np.matmul(np.matmul(reversed\_rotation\_matrix, matrix), rotation\_matrix))  
  
 for\_eigenvectors.append(rotation\_matrix)  
  
 iterations\_count += 1  
  
 eigenvalues: list[float] = get\_total\_eigenvalues\_of\_totally\_rotated\_matrix(matrix)  
 eigenvectors: list[list[float]] = get\_total\_eigenvectors(for\_eigenvectors)  
  
 prettify\_response(eigenvalues, eigenvectors)  
  
 return eigenvalues, eigenvectors, iterations\_count**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Полученные результаты:**

Для уравнения из варианта 8 были найдены собственные векторы и значения  


# **Тестовые задания**

Были решены уравнения с другими коэффициентами из всех вариантов. Например  


**Выводы**

Создаётся ощущение, что единственное трудоёмкое действие, что использует метод Якоби, это перемножение большого количества матриц. Количество операций для перемножения описывается порядком N3. Во время одной итерации таких перемножений 2. Итераций может быть 20. Плюс необходимо дополнительно перемножать матрицы поворота (чтобы потом легко найти собственные векторы). Остаётся надеяться, что в библиотеке NumPy, использованной при выполнении данной ЛР, реализовано быстрое перемножение матриц по Штрассену (или ещё какое-нибудь). Достоинство метода, что при выполнении каждый раз плоского поворота уменьшается сумма квадратов недиагональных элементов. То есть они стремятся к нулю. А у диагональной матрицы хорошо находятся собственные значения, поэтому мы и занимаемся этим аннулированием недиагональных элементов. В этом, по-видимому, и есть суть метода. Цель работы можно считать достигнутой.