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 c 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 E. Существует алгоритм. Исходя из него, создаётся итерационная последовательность, которая принимает вид: $A^{k+1} = (V^k)^{-1} * A^k * V^k$.
Матрица V^k — это матрица плоского поворота.
Как же построить такой процесс ?

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

$$\cos(u) = \sqrt{\frac{1}{2}(1 + (1 + p_k^2)^{-\frac{1}{2}})} \quad \sin(u) = sgn(p_k)\sqrt{\frac{1}{2}(1 - (1 + p^2)^{-\frac{1}{2}})}$$

$$p = \frac{2 * a_{qw}^{(k)}}{a_{qq}^{(k)} - a_{ww}^{(k)}}$$

Если разность между диагональными элементами $a_{qq} – a_{ww}$ равна нулю, то вместо угла u беру значение π/4. Следовательно, синус и косинус уходят в значение 0.5 * корень(2).

3. 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.
4. Нахожу обратную к V матрицу. Некоторые источники пишут, что обратная в данном случае будет равна транспонированной. Так и есть.
5. Нахожу следующую A^{k+1} перемножая: $A^{k+1} = (V^k)^{-1} * A^k * V^k$.

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 A^n. А чтобы получить собственные вектора, необходимо перемножить все получившиеся V^k и разбить на вектор-столбцы. Получим пары: вектор столбец из этого произведения + соответствующий диагональный элемент из матрицы A^n.

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:

$$C = \begin{bmatrix} 0{,}2 & 0 & 0{,}2 & 0 & 0 \\ 0 & 0{,}2 & 0 & 0{,}2 & 0 \\ 0{,}2 & 0 & 0{,}2 & 0 & 0{,}2 \\ 0 & 0{,}2 & 0 & 0{,}2 & 0 \\ 0 & 0 & 0{,}2 & 0 & 0{,}2 \end{bmatrix}, \quad D = \begin{bmatrix} 2{,}33 & 0{,}81 & 0{,}67 & 0{,}92 & -0{,}53 \\ 0{,}81 & 2{,}33 & 0{,}81 & 0{,}67 & 0{,}92 \\ 0{,}67 & 0{,}81 & 2{,}33 & 0{,}81 & 0{,}92 \\ 0{,}92 & 0{,}67 & 0{,}81 & 2{,}33 & -0{,}53 \\ -0{,}53 & 0{,}92 & 0{,}92 & -0{,}53 & 2{,}33 \end{bmatrix}.$$

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.

```python
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()
```

```python
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
```

```python
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

```
Variant 8
Values:
 [4.14803, 8.28862, 5.51877, 1.61624, 0.07834]
Vectors:
[0.70852, -0.48552, 0.24343, -0.22829, -0.38846]
[0.42993, 0.46435, 0.5755, 0.39965, 0.32956]
[-0.15822, -0.3058, 0.3757, -0.55657, 0.65615]
[-0.31597, -0.63112, 0.28851, 0.64687, 0.01317]
[0.43393, -0.23837, -0.6206, 0.24485, 0.55659]
Number of iterations: 18
```

Тестовые задания

Были решены уравнения с другими коэффициентами из всех вариантов. Например

```
Variant 9
Values:
 [4.41529, 8.73837, 5.87013, 1.61813, 0.00809]
Vectors:
[0.72049, -0.45397, 0.21002, -0.194, -0.43937]
[0.43059, 0.45612, 0.58051, 0.39368, 0.33848]
[-0.11012, -0.35448, 0.3851, -0.57166, 0.62218]
[-0.29775, -0.64023, 0.27207, 0.65362, 0.01469]
[0.44124, -0.22432, -0.62974, 0.231, 0.55231]
Number of iterations: 20

Variant 10
Values:
 [4.66692, 9.18942, 6.23738, 1.61989, -0.06361]
Vectors:
[0.72799, -0.4219, 0.17974, -0.16235, -0.48309]
[0.43168, 0.44827, 0.58513, 0.38771, 0.34644]
[-0.06571, -0.39609, 0.39029, -0.5835, 0.58821]
[-0.28222, -0.64724, 0.25866, 0.65902, 0.01474]
[0.44691, -0.21271, -0.63725, 0.22033, 0.54809]
Number of iterations: 20
```

Выводы

Создаётся ощущение, что единственное трудоёмкое действие, что использует метод Якоби, это перемножение большого количества матриц. Количество операций для перемножения описывается порядком N^3. 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.