МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

«Дальневосточный федеральный университет»
(ДВФУ)

ШКОЛА ЕСТЕСТВЕННЫХ НАУК

Кафедра информатики, математического и компьютерного моделирования

ОТЧЕТ

По лабораторной работе 3
по дисциплине «Вычислительная математика»

Направление подготовки
02.03.01 «Математика и компьютерные науки»

Выполнил студент
гр. Б9119-02.03.01сцт
Скурский С.А. ____________
(ФИО) *(подпись)*

«9» мая 20 22 г.

г. Владивосток
2022

Содержание

Введение

Отчёт по лабораторной работе от (03.03) Метод оптимального исключения.

Метод оптимального исключения

Постановка задачи

Изучить, понять и реализовать алгоритм метода оптимального исключения для решения СЛАУ, а также описать работу алгоритма и привести результаты

Метод решения

Метод оптимального исключения.

Алгоритм метода

$$\mathrm{M} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1j} & \dots & a_{1q} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2j} & \dots & a_{2q} & \dots & a_{1n} & b_2 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ a_{p1} & a_{p2} & \dots & a_{pj} & \dots & a_{pq} & \dots & a_{pn} & b_p \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nj} & \dots & a_{nq} & \dots & a_{nn} & b_n \end{bmatrix} (3)$$

Пусть дана система уравнений $Ax = b$. Обозначив b_i через a_{in+1}, преобразуем эту систему к эквивалентной системе более простого вида. Допустим, что $a_{11} \neq 0$. Разделим все коэффициенты первого уравнения системы на a11, который назовем ведущим элементом первого шага, тогда

$$x_1 + a_{12}^{(1)} \cdot x_2 + \dots + a_{1n}^{(1)} \cdot x_n = a_{1n+1}^{(1)}$$

Здесь $a_{1j}^{(1)} = a_{1j}/a_{11}, j = 2, 3, \dots, n+1$

Предположим, что после преобразования первых $k (k \geq 1)$ уравнений система приведена к эквивалентной системе

$$
\begin{cases}
x_1 + \dots\dots\dots\dots + a_{1k+1}^{(k)} x_{k+1} + \dots + a_{1n}^{(k)} x_n & = & a_{1n+1}^{(k)} \\
x_2 + \dots\dots\dots\dots + a_{2k+1}^{(k)} x_{k+1} + \dots + a_{2n}^{(k)} x_n & = & a_{2n+1}^{(k)} \\
\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots & \dots & \\
x_k + a_{kk+1}^{(k)} x_{k+1} + \dots\dots\dots\dots + a_{kn}^{(k)} x_n & = & a_{kn+1}^{(k)} \\
a_{k+11} x_1 + a_{k+12} x_2 + \dots + a_{k+1n+1} x_k + \dots + a_{k+1n} x_n & = & a_{k+1n+1} \\
\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots & \dots & \\
a_{n1} x_1 + a_{n2} x_2 + \dots + a_{nk+1} x_k + \dots + a_{nn} x_n & = & a_{nn+1}
\end{cases}
$$

Исключим неизвестные $x_1, x_2, \dots, x_k$ из $(k+1)$ уравнения посредством вычитания из него первых k уравнений, умноженных соответственно на числа $a_{k+11}, a_{k+12}, \dots, a_{k+1k}$ и разделив вновь полученное уравнение на коэффициенты при x_{k+1}. Теперь $(k+1)$ уравнение примет вид:

$$x_{k+1} + a_{k+1k+2}^{(k+1)} \cdot x_{k+2} + \dots + a_{k+1n}^{(k+1)} \cdot x_n = a_{k+1n+1}^{(k+1)}$$

Исключая с помощью этого уравнения неизвестное x_{k+1} из первых k уравнений (3), получаем опять систему вида (3), но с заменой индекса k на $k+1$, причем

$$a_{i1}^{(1)} = \frac{a_{1i}}{a_{11}}, i = 2, 3, \dots, n+1$$

$$a_{k+1p}^{(k+1)} = \frac{a_{k+1p} - \sum\limits_{r=1}^{k} a_{rp}^{(k)} a_{k+1r}}{a_{k+1k+1} - \sum\limits_{r=1}^{k} a_{rk+1}^{(k)} a_{k+1r}}$$

$$a_{ip}^{(k+1)} = a_{ip}^{(k)} - a_{k+1p}^{(k+1)} a_{ik+1}^{(k)}, i = 1, 2, \dots, k;$$

$$p = k+2, k+3, \dots, n+1; \quad k = 0, 1, \dots, n-1$$

После преобразования всех уравнений находим решение исходной системы $x_i = a_{in+1}^{(n)}, i = 1, 2, \dots, n$.

Спецификации используемых функций и типов данных

Функция, в которой реализован метод оптимального исключения.

Сначала идёт объявление переменной k, которая отвечает за условие цикла while.

Конечное значения данной переменной будет (len(Ax) - 1), так что можно спокойно переработать код изменив цикл while на цикл for и в теории ничего не сломается.

Далее следует самый первый цикл, в котором происходит деление первой строки матрицы на самый первый элемент в этой самой матрице Ax[0][0]. Он вынесен из основного цикла, так как этот шаг является начальным и его участие для цикличной работы алгоритма не требуется, так что будет нелепо вставлять его в основной цикл, добавляя ненужное сравнение.

```
def MethodOptimumExc(Au):
    Ax = np.copy(Au)
    print(Ax)
    k = 0

    for i in range(k, len(A) + 1):
        Ax[k][i] = Au[k][i] / Au[k][k]
```

И следом начинается работа основного цикла. Как я и сказал ранее, конечное значение переменной k будет (len(Ax) - 1) потому что за такое количество итераций выполнится нужное количество вычислений для приведения матрицы к единичной.

Работа цикла while разделяется на три отдельных блока(цикла):

Первый блок: это вычитание строк сверху той, над которой происходит деление на центральный элемент.

Второй блок: это деление на центральный элемент.

Третий блок: это вычитание новой строки, над которой происходило деление на центральный элемент, из строк выше её.

Переменная WhatsDiv хранит в себе элемент, на который будет умножаться вся строка, которую надо будет вычесть Этот элемент единственнен потому что пошагово ноль получается только один, а остальное вычитается "за компанию".

Переменная DIV отвечает за сохранение элемента, на который делится вся строка, ибо если его не сохранять, то он просто обратится в единицу, и тогда деление станет бессмысленным. (Можно использовать цикл в обратную сторону, тогда и переменную выделять будет не нужно, но теряется удобочитаемость кода. Пример такого цикла будет закомменчен и представлен сразу после того, что используется в коде.)

```
while k != (len(Ax) - 1):
    for i in range(k + 1):
        WhatsDiv = Ax[k + 1][i]
        for j in range(len(A) + 1):
            Ax[k + 1][j] = Ax[k + 1][j]
            - Ax[i][j] * WhatsDiv

    DIV = Ax[k + 1][k + 1]
    for i in range(k + 1, len(A) + 1):
        Ax[k + 1][i] = Ax[k + 1][i] / DIV
    /*for i in range(len(A), k, -1):      */
    /*     Ax[k + 1][i] = Ax[k + 1][i] / */
    /*                     Ax[k + 1][k + 1] */

    for i in range(k, -1, -1):
        WhatsDiv = Ax[i][k + 1]
        for j in range(k + 1, len(A) + 1):
            Ax[i][j] = Ax[i][j] - Ax[k + 1][j] *
                                WhatsDiv

    k += 1
print(Ax)
x = X_Finder(Ax)
return x
```

Функция, которая выносит x из матрицы в отдельный массив

```
def X_Finder(Axu):
    x = np.zeros(len(Axu))
    for i in range(len(x)):
        x[i] = Axu[i][len(Axu)]
    return x
```

Функция, которая делает ресайз матрицы и произоводит слияние между матрицами A и b, что требуется для метода оптимального исключения.

```
def MergeMatrix(Aux, bux):
    matrix = np.zeros((len(Aux), len(Aux) + 1))
    for i in range(len(Aux)):
        for j in range(len(Aux)):
            matrix[i][j] = Aux[i][j]
    for i in range(len(Aux)):
        matrix[i][len(Aux)] = bux[i]
    return matrix
```

Функция проверки

```
def CheckResult(Auf, buf, x):
    nx = np.zeros(len(x))
    for i in range(len(Auf)):
        for j in range(len(Auf)):
            nx[i] += Auf[i][j] * x[j]
    for i in range(len(nx)):
        print("x[", i, "] = ", x[i])
    for i in range(len(nx)):
        print("D[", i, "] = ", nx[i] - buf[i])
```

Описание тестов

Первый тест: матрица из примера в методичке для данного метода.

Размер 3x3

```
A = np.array([[5., 2, 3],
              [1, 6, 1],
              [3, -4, -2]])
b = np.array([3., 5., 8.])
```

Второй тест: матрица из примера в методичке для метода квадратного корня.

Размер 5x5

```
A = np.array([[1., 3, -2, 0, -2],
              [3, 4, -5, 1, -3],
              [-2, -5, 3, -2, 2],
              [0, 1, -2, 5, 3],
              [-2, -3, 2, 3, 4]])
b = np.array([0.5, 5.4, 5.0, 7.5, 3.3])
```

Третий тест: матрица из примера, данного на паре.

Размер 7x7

```
A = np.array([
[0.411, 0.421, -0.333, 0.313, -0.141, -0.381, 0.245],
[0.241, 0.705, 0.139, -0.409, 0.321, 0.0625, 0.101],
[0.123, -0.239, 0.502, 0.901, 0.243, 0.819, 0.321],
[0.413, 0.309, 0.801, 0.865, 0.423, 0.118, 0.183],
[0.241, -0.221, -0.243, 0.134, 1.274, 0.712, 0.423],
[0.281, 0.525, 0.719, 0.118, -0.974, 0.808, 0.923],
[0.246, -0.301, 0.231, 0.813, -0.702, 1.223, 1.105]
])
b = np.array([0.096, 1.252, 1.024, 1.023, 1.155, 1.937, 1.673])
```

Для проверки правильности вычисления написанным алгоритмом будем использовать функцию CheckResult(), которая умножает полученные результаты на коэффициенты в матрице соответственно. И следом вычитает это из b массива, чтобы увидеть разность.

Результаты численного эксперимента

Первый тест:

[2. 1. -3.]
x[0] = 1.9999999999999998
x[1] = 1.0
x[2] = -3.0

D[0] = -1.7763568394002505e-15
D[1] = 0.0
D[2] = -8.881784197001252e-16

Второй тест:
[-6.1 -2.2 -6.8 -0.9 0.2]
x[0] = -6.100000000000001
x[1] = -2.2
x[2] = -6.800000000000001
x[3] = -0.8999999999999999
x[4] = 0.1999999999999996

D[0] = -6.106226635438361e-16
D[1] = -4.440892098500626e-15
D[2] = 0.0
D[3] = 8.881784197001252e-16
D[4] = 1.7763568394002505e-15

Третий тест:

[11.09196963 -2.51573632 0.72098648 -2.54467447 -1.60482658 3.62397366 -4.94958981]

x[0] = 11.091969628167796
x[1] = -2.5157363215954667
x[2] = 0.7209864792670801
x[3] = -2.5446744665697354
x[4] = -1.6048265844708471
x[5] = 3.623973659251843
x[6] = -4.9495898138303644

D[0] = 5.856426454897701e-15
D[1] = 1.2212453270876722e-14
D[2] = -5.773159728050814e-15
D[3] = 2.4424906541753444e-15
D[4] = 6.661338147750939e-16
D[5] = 1.1102230246251565e-15
D[6] = 0.0