Министерство науки и высшего образования Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)» (МГТУ им. Н.Э. Баумана)

Факультет «Информатика и системы управления»

Кафедра «Теоретическая информатика и компьютерные технологии»

Отчёт о лабораторной работе № 1

по курсу «Численные методы»

РЕШЕНИЕ 3Х-ДИАГОНАЛЬНОЙ СИСТЕМЫ ЛИНЕЙНЫХ АЛГЕБРАИЧЕСКИХ УРАВНЕНИЙ МЕТОДОМ ПРОГОНКИ

Студент: К. Лозовска

Группа: ИУ9И-64Б

Преподаватель: А.Б.Домрачева

Москва, 2023

СОДЕРЖАНИЕ

ЦЕЛЬ И ПОСТАНОВКА ЗАДАЧИ

Цель данной лабораторной работы: изучить накопление погрешности в решении СЛАУ с 3х-диагональной матрицей.

Для достижения цели были поставлены следующие задачи:

1. ознакомиться с теорией метода прогонки и его применением для решения трехдиагональных систем линейных уравнений;
2. реализовать алгоритм метода прогонки на языке программирования Python;
3. проверить реализацию на нескольких трехдиагональных системах линейных уравнений с различными коэффициентами и правыми частями, а также проверить, дает ли метод точные решения;
4. подготовить отчет, обобщающий полученные результаты, выводы, включая описание метода развертки, подробности реализации.

ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Метод прогонки — популярный алгоритм, используемый в программировании для решения линейных алгебраических уравнений. Одним из применений метода прогонки является решение систем линейных алгебраических уравнений с 3х-диагональной матрицей. Эти системы имеют три диагонали – главная диагональ и две примыкающие к ней диагонали.

Метод прогонки, также известный как метод прогонки по линии, представляет собой алгоритм, используемый для решения линейных алгебраических уравнений. Он работает, перебирая систему и решая каждую неизвестную переменную по очереди. На каждой итерации алгоритм обновляет систему решаемой переменной и использует обновленную систему для поиска следующей неизвестной переменной. Этот процесс повторяется до тех пор, пока не будут решены все неизвестные переменные.

Описание алгоритма:

Пусть a – массив элементов под главной диагональю, b – массив элементов главной диагонали, c – массив элементов над главной диагональю.

$$\begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 \\ a_1 & b_2 & c_2 & \dots & \dots & 0 \\ 0 & a_2 & b_3 & c_3 & \dots & 0 \\ \vdots & \dots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & a_{n-2} & b_{n-1} & c_{n-1} \\ 0 & \dots & \dots & \dots & a_{n-1} & b_n \end{pmatrix} \qquad (*)$$

Матрица (*) задает следующую СЛАУ:

$$\begin{cases} b_1x_1 + c_1x_2 = d_1 & (1) \\ a_1x_1 + b_2x_2 + c_2x_3 = d_2 & (2) \\ \dots & (\dots) \\ a_{n-1}x_{n-1} + b_nx_n = d_n & (n) \end{cases}$$

Из уравнения (1) получаем:

$$x_1 = \frac{d_1 - c_1x_2}{b_1} = \frac{d_1}{b_1} - \frac{c_1}{b_1}x_2$$

Вводим замену переменных: $\alpha_1 = -\frac{c_1}{b_1}, \beta_1 = \frac{d_1}{b_1}$: $\quad x_1 = \alpha_1x_2 + \beta_1$.

Подставляем в уравнение (2):

$$a_1\frac{d_1 - c_1x_2}{b_1} + b_2x_2 + c_2x_3 = d_2$$

$$a_1(\alpha_1x_2 + \beta_1) + b_2x_2 + c_2x_3 = d_2$$

$$x_2 = -\frac{c_2}{a_1\alpha_1 + b_2}x_3 + \frac{d_2 - a_1\beta_1}{a_1\alpha_1 + b_2}$$

Вводим замену: $\alpha_2 = -\frac{c_2}{a_1\alpha_1+b_2}, \beta_2 = \frac{d_2-a_1\beta_1}{a_1\alpha_1+b_2}$: $\quad x_2 = \alpha_2x_3 + \beta_2$.

Аналогично продолжаем для всех уравнений. Для x_i, где $i = \overline{2,n}$, получаем:

$$x_i = -\frac{c_i}{a_{i-1}\alpha_{i-1} + b_i}x_{i+1} + \frac{d_i - a_{i-1}\beta_{i-1}}{a_{i-1}\alpha_{i-1} + b_i} = \alpha_ix_{i+1} + \beta_i$$

$$\alpha_i = -\frac{c_i}{a_{i-1}\alpha_{i-1} + b_i}, \beta_i = \frac{d_i - a_{i-1}\beta_{i-1}}{a_{i-1}\alpha_{i-1} + b_i}.$$

Получили систему:

$$\begin{cases} x_1 = \alpha_1 x_2 + \beta_1, \ \ \alpha_1 = -\dfrac{c_1}{b_1}, \beta_1 = \dfrac{d_1}{b_1}, b_1 \neq 0 \\ x_i = \alpha_i x_{i+1} + \beta_i, \ \ \alpha_i = -\dfrac{c_i}{a_{i-1}\alpha_{i-1} + b_i}, \beta_i = \dfrac{d_i - a_{i-1}\beta_{i-1}}{a_{i-1}\alpha_{i-1} + b_i} \ , i = \overline{2,n} \\ x_n = \beta_n, \ \ \beta_n = \dfrac{d_n - a_{n-1}\beta_{n-1}}{a_{n-1}\alpha_{n-1} + b_n} \end{cases}$$

Вычисление α_i , $\beta_i, i = \overline{2,n}$ называется **прямым ходом** метода прогонки.

Система: $\begin{cases} x_n = \beta_n \\ x_{n-1} = \alpha_{n-1} x_n + \beta_{n-1} \\ \dots \\ x_1 = \alpha_1 x_2 + \beta_1 \end{cases}$ называется **обратным ходом** метода прогонки.

В качестве начального приближения x_n выбирается значение β_n такое что $\alpha_{n-1} a_{n-1} = 0$.

Условия диагонального приближения.

Необходимое условие: $b_1 \neq 0$.

Достаточные условия:

1. $|b_i| \geq |a_{i-1}| + |c_i|, \ i = \overline{2,n}$
2. $\left|\frac{a_{i-1}}{b_i}\right| \leq 1, \ \left|\frac{c_i}{b_i}\right| \leq 1$

Для оценки погрешности вычисления вычисляем:

$$A\overline{x^*} = \overline{d^*}$$

$$A(\bar{x} - \overline{x^*}) = (\bar{d} - \overline{d^*})$$

$$\bar{r} = \ (\bar{d} - \overline{d^*})$$

$$\bar{e} = (\bar{x} - \overline{x^*})$$

$$A\bar{e} = \bar{r},$$

где $\bar{e}$ – искомый вектор ошибок.

Тогда $\bar{e} = A^{-1}\bar{r}$. Точное решение $\bar{x} = \overline{x^*} - \ \bar{e}$.

ПРАКТИЧЕСКАЯ РЕАЛИЗАЦИЯ

Импортируется библиотека ***NumPy*** для научных вычислений как ***np***. Для глобальной переменной N (размер матрицы) устанавливается значение None. Определяется функция ***parseArrs()***, которая принимает строку ***line*** и целое число ***N*** в качестве входных данных и возвращает массив ***NumPy*** чисел с плавающей запятой. Эта функция разбивает строку входной строки на отдельные строки, преобразует каждую строку в число с плавающей запятой и добавляет ее в список.

```
import numpy as np

N = None

def parseArrs(line: str, N: int) -> np.ndarray:
    arr = []
    strs = line.split(" ")

    for s in strs:
        num = float(s)
        arr.append(num)

    return np.array(arr)
```

Определяется функция ***solution()***, которая принимает четыре массива NumPy ***a***, ***b***, ***c*** и ***d*** в качестве входных данных и возвращает массив NumPy ***x***. Эта функция реализует метод прогонки для решения трехдиагональной системы линейных уравнений. Он начинается с шага прямой подстановки и вычисляет значения α и β для каждой строки трехдиагональной матрицы. Затем выполняется шаг обратной подстановки для вычисления вектора решения x.

```
def solution(a: np.ndarray, b: np.ndarray, c: np.ndarray, d: np.ndarray) -> np.ndarray:
    global N
    N = len(b)
    x = np.zeros(N)
    #forward
    alpha, beta = [- c[0] / b[0]], [d[0] / b[0]]
    for i in range(1, N):
        if i != N-1:
            y = a[i-1] * alpha[i-1] + b[i]
            alpha.append(-c[i] / y)
            beta.append((d[i]-a[i-1] * beta[i-1]) / y)
        else:
            y = a[N-2] * alpha[N-2] + b[N-1]
            beta.append((d[N-1] - a[N-2] * beta[N-2]) / y)
```

```python
    #backwards
    for i in reversed(range(N)):
        if i == N-1:
            x[N-1] = beta[N-1]
        else:
            x[i] = alpha[i] * x[i+1] + beta[i]
    return x
```

Определяется функция ***makeMatrix()***, которая принимает три массива NumPy ***c***, ***b*** и ***a*** в качестве входных данных и возвращает трехдиагональную матрицу в виде массива NumPy.

```python
def makeMatrix(c: np.ndarray, b: np.ndarray, a: np.ndarray) -> np.ndarray:
    m = np.zeros((N,N))
    for i in range(N):
        for j in range(N):
            if i == j:
                m[i][j] = b[i]
                if i != N-1:
                    m[i][i+1] = c[i]
                    m[i+1][i] = a[i]
    return m
```

Определяется функция ***mulMatVec()***, которая принимает две матрицы массивов в качестве входных данных и возвращает массив NumPy ***d***. Эта функция умножает матричную матрицу на вектор x для получения вектора d.

```python
def mulMatVec(matrix: np.ndarray, x: np.ndarray) -> np.ndarray:
    d = np.zeros(N)
    for i in range(N):
        s = 0
        for j in range(N):
            s += matrix[i][j] * x[j]
        d[i] = s
    return d
```

Определяется функция ***main()***, читающая текстовый файл и выполняющая следующие шаги. Считывает первую строку файла, содержащую размерность матрицы. Читает следующие четыре строки файла, которые содержат главную диагональ, элементы над диагональю, элементы под диагональю и вектор D. Вызывает функцию ***solution()*** для решения системы линейных уравнений.

```python
def main():
    global N
    # tets<i>.txt = dimension; matrix: main diagonal, above diagonal , under diagonal; vector D
    with open("test1.txt") as file:
        N = int(file.readline().strip())
        arrs = [file.readline().strip() for _ in range(4)]

    b = parseArrs(arrs[0], N)
    c = parseArrs(arrs[1], N-1)
    a = parseArrs(arrs[2], N-1)
    d = parseArrs(arrs[3], N)

    x = solution(a, b, c, d)
```

Вызывает функцию ***makeMatrix()*** для создания трехдиагональной матрицы. Вызывает функцию ***mulMatVec()*** для вычисления нового вектора ***d***.

```python
    m = makeMatrix(c, b, a)
    # np.set_printoptions(precision=16)
    f = np.array2string(m, prefix="    ", suppress_small=True, formatter={'float_kind':lambda x: "%.8f" % x})

    #print(f"A = {f}\n")

    print("X: ", end="")
    print(" ".join([f"{n:.16f}" for n in x]))

    d_new = mulMatVec(m, x)
    print("new vector d: ", end="")
    print(" ".join([f"{n:.16f}" for n in d_new]))
```

Вычисляет вектор ошибки ***r***. Вызывает функцию ***mulMatVec()*** с обратной трехдиагональной матрицей и вектором ошибки ***r*** для вычисления вектора ошибки ***e***. Выводит на консоль вектор решения ***x***, новый вектор ***d***, вектор ошибки ***r*** и вектор ошибки ***e***.

```python
    r = [np.abs(d[i] - d_new[i]) for i in range(N)]
    print("vector r: ", r)
    #print("A^(-1) = ", np.linalg.inv(m))
    e = mulMatVec(np.linalg.inv(m), r)
    f = np.array2string(e, prefix="    ", suppress_small=True, formatter={'float_kind': lambda x: "%.16f" % x
    print(f"Погрешность : vector e = {f}")

    ans = [1, 1, 1, 1]
    #print(f"|x-x*| = : {abs(ans - x)}")

main()
```

ТЕСТИРОВАНИЕ

Тестирование было проведено на следующих входных данных:

– в качестве трехдиагональной матрицы A была взята матрица $\begin{pmatrix} 4 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 0 & 1 & 4 & 1 \\ 0 & 0 & 1 & 4 \end{pmatrix}$;

– в качестве вектора $\bar{d}$ – вектор $\begin{pmatrix} 5 \\ 6 \\ 6 \\ 5 \end{pmatrix}$.

В результате работы программы получаем следующие значения:

```
X: 0.9952153110047848 1.0191387559808611 0.9282296650717703 1.2679425837320575
new vector d: 5.0000000000000000 5.9999999999999991 6.0000000000000000 6.0000000000000000
vector r:  [0.0, 8.881784197001252e-16, 0.0, 0.0]
Погрешность : vector e = [-0.0000000000000001 0.0000000000000003 -0.0000000000000001
     0.0000000000000000]
```

$x = (0.995,\ 1.019,\ 0.928,\ 1.2679);$

$e = (-0.0000000000000001,\ 0.0000000000000003, -0.0000000000000001,$
$0.0000000000000000).$

Вектор погрешности в данном тесте не является нулевым, что связано с использованием типа данных float с точностью в 16 знаков после запятой.

ВЫВОДЫ

В ходе выполнения данной лабораторной работы был рассмотрен метод решения СЛАУ с трехдиагональной матрицей – метод прогонки. Этот метод был реализован на языке программирования Python.

Лабораторная работа показала, что метод прогонки может сходиться к решению системы за относительно небольшое число итераций. Кроме того, этот метод эффективен и требует меньше памяти по сравнению с другими методами, такими как метод исключения Гаусса. Однако в данной реализации точность решения недостаточно высока – полученное решение существенно отличается от единичного вектора (точного решения данной системы).

Для метода прогонки можно сделать вывод о том, что в нем отсутствует методологическая, т.е. логическая погрешность, но при этом присутствует вычислительная погрешность. Это можно объяснить использованием чисел с плавающей запятой, что ведет к высокому накоплению вычислительной ошибки.