Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
Московский государственный технический университет имени Н.Э. Баумана

Факультет: Информатика и системы управления
Кафедра: Теоретическая информатика и компьютерные технологии

Лабораторная работа №1
по курсу «Численные методы»
«Решение СЛАУ с трехдиагональной матрицей методом прогонки»

Выполнила:
студентка группы ИУ9-62Б
Самохвалова П. С.

Проверила:
Домрачева А. Б.

Москва, 2023

Цель:

Изучение погрешности решения СЛАУ с трехдиагональной матрицей методом прогонки.

Постановка задачи:

1. Известно точное решение $\overline{x}$, найти приближенное $\overline{x}^*$, определить $\overline{e_1} = \overline{x} - \overline{x}^*$

2. Неизвестно точное решение $\overline{x}$, тогда найти приближенное решение $\overline{x}^*$ и далее $\overline{e_2} = A^{-1}(\overline{d} - \overline{d}^*)$

3. Сравнить $\overline{e_1}$ и $\overline{e_2}$, обосновать разницу

Описание алгоритма:

$$A = \begin{pmatrix} b_1 & c_1 & 0 & ... & ... & 0 \\ a_1 & b_2 & c_2 & ... & ... & 0 \\ 0 & a_2 & b_3 & c_3 & ... & 0 \\ \vdots & ... & \ddots & \ddots & \ddots & \vdots \\ 0 & ... & ... & a_{n-2} & b_{n-1} & c_{n-1} \\ 0 & ... & ... & ... & a_{n-1} & b_n \end{pmatrix}$$

$$\overline{d} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{pmatrix}$$

$$\begin{cases} b_1x_1 + c_1x_2 = d_1 \\ a_1x_1 + b_2x_2 + c_2x_3 = d_2 \\ ... \\ a_{n-1}x_{n-1} + b_nx_n = d_n \end{cases}$$

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

$$b_1 \neq 0$$

$$\alpha_1 = -\frac{c_1}{b_1}, \beta_1 = \frac{d_1}{b_1}$$

$$a_1(\alpha_1x_2 + \beta_1) + b_2x_2 + c_2x_3 = (a_1\alpha_1 + b_2)x_2 + c_2x_3 = d_2 - a_1\beta_1$$

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

$$x_i = \alpha_ix_{i+1} + \beta_i$$

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

$$x_{n-1} = \alpha_{n-1}x_n + \beta_{n-1}$$

$$a_{n-1}(\alpha_{n-1}x_n + \beta_{n-1}) + b_n x_n = d_n$$

$$(a_{n-1}\alpha_{n-1} + b_n)x_n = d_n - a_{n-1}\beta_{n-1}$$

$$x_n = \frac{d_n - a_{n-1}\beta_{n-1}}{a_{n-1}\alpha_{n-1} + b_n}$$

$$x_i = \alpha_i x_{i+1} + \beta_i$$

$$\left|\frac{a_{i-1}}{b_i}\right| \leq 1$$

$$\left|\frac{c_i}{b_i}\right| \leq 1$$

$$|b_i| \geq |a_{i-1}| + |c_i|$$

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

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

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

$$A\overline{e} = \overline{r}$$

$$\overline{e} = A^{-1}\overline{r}$$

$$\overline{x} = \overline{x}^* + \overline{e}$$

Практическая реализация:

Листинг 1. Решение СЛАУ с трехдиагональной матрицей методом прогонки

```
from decimal import *

getcontext().prec = 32


def det(m):
```

```python
    l = len(m)
    if l == 2:
        return m[0][0] * m[1][1] - m[1][0] * m[0][1]
    else:
        d = 0
        for i in range(l):
            d += m[0][i] * alg_add(m, 0, i)
        return d


def alg_add(m, i, j):
    l = len(m)
    m1 = []
    for k in range(l):
        if k != i:
            m1.append([])
            for q in range(l):
                if q != j:
                    m1[-1].append(m[k][q])
    return (-1) ** (i + j) * det(m1)


def module_v(v):
    s = 0
    for i in range(len(v)):
        s += v[i] ** Decimal(2)
    return s ** Decimal(1/2)


def print_v(v):
    for i in range(len(v)):
        print(v[i])


n = 4

m = [[4, 1, 0, 0],
     [1, 4, 1, 0],
     [0, 1, 4, 1],
```

```python
     [0, 0, 1, 4]]

x_accurate = [1, 1, 1, 1]

a = [0]
b = [0]
c = [0]

for i in range(n):
    for j in range(n):
        m[i][j] = Decimal(m[i][j])
        if i == j:
            b.append(m[i][j])
        elif i == j + 1:
            a.append(m[i][j])
        elif i == j - 1:
            c.append(m[i][j])

d = [0, 5, 6, 6, 5]
d = list(map(str, d))
d = list(map(Decimal, d))

alpha = [0] * n
beta = [0] * n

for i in range(1, n):
    alpha[i] = -c[i] / (a[i - 1] * alpha[i - 1] + b[i])
    beta[i] = (d[i] - a[i - 1] * beta[i - 1]) / \
              (a[i - 1] * alpha[i - 1] + b[i])

x_inaccurate = [0] * (n + 1)

x_inaccurate[n] = (d[n] - a[n - 1] * beta[n - 1]) / \
                  (a[n - 1] * alpha[n - 1] + b[n])

for i in range(n - 1, 0, -1):
    x_inaccurate[i] = alpha[i] * x_inaccurate[i + 1] + beta[i]

x_inaccurate = x_inaccurate[1:]
```

```
e1 = [0] * n

for i in range(n):
    e1[i] = x_accurate[i] - x_inaccurate[i]

print("x_inaccurate")
print_v(x_inaccurate)
print()
print("e1")
print_v(e1)
print()
print("|e1|")
print(module_v(e1))
print()

a = a[1:]
b = b[1:]
c = c[1:]
d = d[1:]

d_inaccurate = [0] * n

d_inaccurate[0] = b[0] * x_inaccurate[0] + c[0] * x_inaccurate[1]
if n > 1:
    d_inaccurate[n - 1] = a[n - 2] * x_inaccurate[-2] + b[n - 1] * \
                          x_inaccurate[-1]
for i in range(1, n - 1):
    d_inaccurate[i] = a[i - 1] * x_inaccurate[i - 1] + b[i] * \
                      x_inaccurate[i] + c[i] * x_inaccurate[i + 1]

r = [0] * n

for i in range(n):
    r[i] = d[i] - d_inaccurate[i]

m_inverse = []
for i in range(n):
    m_inverse.append([0] * n)
```

```python
c = Decimal(1) / Decimal(det(m))

for i in range(n):
    for j in range(n):
        m_inverse[j][i] = c * alg_add(m, i, j)

e2 = [0] * n

for i in range(n):
    for j in range(n):
        e2[i] += m_inverse[i][j] * r[j]

x = [0] * n

for i in range(n):
    x[i] = x_inaccurate[i] + e2[i]

print("d_inaccurate")
print_v(d_inaccurate)
print()
print("x")
print_v(x)
print()
print("e2")
print_v(e2)
print()
print("|e2|")
print(module_v(e2))
```

Результаты работы:

$$A = \begin{pmatrix} 4 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 0 & 1 & 4 & 1 \\ 0 & 0 & 1 & 4 \end{pmatrix}$$

$$\overline{d} = \begin{pmatrix} 5 \\ 6 \\ 6 \\ 5 \end{pmatrix}$$

Точное решение:

$$\overline{x_{accurate}} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

В результате работы программы получаем:

$$\overline{x}^* = \begin{pmatrix} 1.0000000000000000000000000000000 \\ 1.0000000000000000000000000000000 \\ 0.9999999999999999999999999999995 \\ 1.0000000000000000000000000000000 \end{pmatrix}$$

$$\overline{e_1} = \overline{x_{accurate}} - \overline{x}^* = \begin{pmatrix} 0E-31 \\ 0E-31 \\ 5E-32 \\ 0E-31 \end{pmatrix}$$

$$|\overline{e_1}| = 5.0000000000000000000000000000000E-32$$

$$\overline{d}^* = A\overline{x}^* = \begin{pmatrix} 5.0000000000000000000000000000000 \\ 6.0000000000000000000000000000000 \\ 5.9999999999999999999999999999998 \\ 5.0000000000000000000000000000000 \end{pmatrix}$$

$$\overline{e_2} = A^{-1}(\overline{d} - \overline{d}^*) = \begin{pmatrix} 3.8277511961722488038277511961722E-33 \\ -1.5311004784688995215311004784689E-32 \\ 5.7416267942583732057416267942584E-32 \\ -1.4354066985645933014354066985646E-32 \end{pmatrix}$$

$$|\overline{e_2}| = 6.1251494759063196724901316333084E-32$$

$$\overline{x} = \overline{x}^* + \overline{e_2} = \begin{pmatrix} 1.0000000000000000000000000000000 \\ 0.99999999999999999999999999999998 \\ 1.0000000000000000000000000000000 \\ 0.99999999999999999999999999999999 \end{pmatrix}$$

Выводы:

В результате выполнения лабораторной работы было рассмотрено решение СЛАУ с трёхдиагональной матрицей методом прогонки, была написана реализация на языке программирования python. Для данного метода можно сделать вывод, что отсутствует методологическая погрешность, но присутствует вычислительная погрешность из-за использования чисел с плавающей запятой. В вычислениях использовались числа, имеющие 32 знака после запятой. Были получны значения:

$$\overline{e_1} = \begin{pmatrix} 0E-31 \\ 0E-31 \\ 5E-32 \\ 0E-31 \end{pmatrix}$$

$$|\overline{e_1}| = 5.0000000000000000000000000000000E-32$$

$$\overline{e_2} = \begin{pmatrix} 3.8277511961722488038277511961722E-33 \\ -1.5311004784688995215311004784689E-32 \\ 5.7416267942583732057416267942584E-32 \\ -1.4354066985645933014354066985646E-32 \end{pmatrix}$$

$$|\overline{e_2}| = 6.1251494759063196724901316333084E-32$$

Значение вектора ошибки $\overline{e_2}$ можно объяснить накоплением вычислительной ошибки при выполнении арифметических операций над числами с плавающей запятой при вычислении обратной матрицы.