Київський національний університет імені Тараса Шевченка

Факультет комп’ютерних наук та кібернетики

Чисельні методи в інформатиці

Лабораторна робота №2

“Розв’язок систем нелінійних рівнянь

прямими та ітераційними методами”

Варіант №7

Виконала студентка групи ІПС-31

Сенечко Дана Володимирівна

Київ - 2025

Постановка задачі

Знайти розв'язок систем рівнянь:

Методом Гауса, знайти визначник та обернену матрицю:

1	2	3	0		X1		22
4	3	1	2	x	X2	=	30
2	1	2	1		X3		21
0	3	0	-5		X4		-21

Методом прогонки:

3	2	0		X1		9
2	4	1	x	X2	=	19
0	1	5		X3		28

Методом Якобі:

6	0	2	3		X1		24
0	4	2	1	x	X2	=	18
2	2	5	0		X3		21
1	1	0	3		X4		15

Теоретичні відомості

Метод Гауса.

Для зменшення обчислювальної похибки в методі Гауса використовують вибір головного елементу: а) по стовпцях; б) по рядках; в) за всією матрицею. Розглядаємо алгоритм на прикладі методу Гауса з вибором головного елементу по стовпцях.

Покладемо $A_0 = A$. Ведучим елементом обирається максимальний по модулю елемент стовпця $a_{lk} = max_i |a_{ik}^{(k-1)}|,\ i = \overline{k,\ n}$. Для того, щоб ведучий елемент зайняв відповідне місце, переставляються рядки k та l в матриці A_{k-1} за допомогою матриці перестановок: $\overline{A_k} = P_k A_{k-1}$, де P_k -

матриця перестановок, отримана з одиничної матриці перестановкою k та l рядків.

Прямий хід Гауса в матричній формі: $A_k = M_k \overline{A_k}$, де M_k – матриця розмірності $n \times n$:

$$M_k = \begin{pmatrix} 1 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & m_{kk} & 0 & \dots & 0 \\ 0 & 0 & \dots & m_{(k+1)k} & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & m_{nk} & 0 & \dots & 1 \end{pmatrix},$$

$$m_{kk} = \frac{1}{\tilde{a}_{kk}^{(k)}}, \quad m_{ik} = \frac{-\tilde{a}_{ik}^{(k)}}{\tilde{a}_{kk}^{(k)}}, \quad i = \overline{k+1, n}.$$

За допомогою прямого ходу методу Гауса в матричній формі: $M_n P_n \dots M_2 P_2 M_1 P_1 Ax = M_n P_n \dots M_2 P_2 M_1 P_1 b$, зводимо систему до вигляду:

$$\begin{cases} x_1 + a_{12}^{(1)} x_2 + \dots + a_{1n}^{(1)} x_n = & a_{1(n+1)}^{(1)}; \\ \qquad\qquad x_2 + \dots + a_{2n}^{(2)} x_n = & a_{2(n+1)}^{(2)}; \\ \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots & \\ \qquad\qquad\qquad\qquad\qquad x_n = & a_{n(n+1)}^{(n)}. \end{cases}$$

Розв’язок знаходимо за допомогою зворотнього ходу Гауса: $x_n = a_{n(n+1)}^{n}$,

$x_i = a_{i(n+1)}^{(i)} - \sum_{j=i+1}^{n} a_{ij}^{(i)}$, $i = \overline{n-1, 1}$.

Складність алгоритму: $Q(n) = \frac{2}{3}n^3 + O(n^2)$.

Методом Гауса з вибором головного можна знайти визначник:

$$detA = (-1)^p\ \tilde{a}_{11}^{(1)}\ \tilde{a}_{22}^{(2)}\ \dots\ a_{nn}^{(n)} = (-1)^p\ a_{11}^{(0)}\ a_{22}^{(1)}\ \dots\ a_{nn}^{(n-1)},$$

де p – кількість перестановок.

Метод прогонки (Томаса).

Нехай маємо систему вигляду:

$$\begin{cases} -c_0y_0 + b_0y_1 = -f_0; \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_iy_{i-1} - c_iy_i + b_iy_{i+1} = -f_i, \quad i = \overline{1, n-1}; \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \qquad a_ny_{n-1} - c_ny_n = -f_n; \end{cases}$$

Достатня умова стійкості. Нехай коефіцієнти $a_0, b_0 = 0;\ c_0, c_n \neq 0;$ $a_i, b_i, c_i \neq 0;\ i = \overline{1, n-1}$. Якщо виконуються умови:

1) $|c_i| \geq |a_i| + |b_i|,\ i = \overline{0, n};$

2) $\exists\, i : |c_i| > |a_i| + |b_i|,$

то метод є стійким: $|\alpha_i| \leq 1;\ |z_i| > 1,\ i = \overline{1, n}.$

Прямий хід метода Гауса в методі прогонки відповідає знаходженню прогонкових коефіцієнтів: $\alpha_1 = \frac{b_0}{c_0};$ $\beta_1 = \frac{f_0}{c_0};$ $\alpha_{i+1} = \frac{b_i}{z_i};$ $\beta_{i+1} = \frac{f_i + a_i\beta_i}{z_i};$ $z_i = c_i - \alpha_i a_i;$ $i = \overline{1, n-1}.$

Зворотній хід:

$y_n = \frac{f_n + a_n\beta_n}{z_n};$ $y_i = \alpha_{i+1}y_{i+1} + \beta_{i+1},\ i = \overline{n-1, 0}.$

Складність методу прогонки: $Q(n) = 8n - 2.$

Метод Якобі.

Ітераційний процес має вигляд: $x_i^{k+1} = -\sum\limits_{j=1}^{i-1} \frac{a_{ij}}{a_{ii}} x_j^k - \sum\limits_{j=i+1}^{n} \frac{a_{ij}}{a_{ii}} x_j^k + \frac{b_i}{a_{ii}}.$

Достатня умова збіжності. Якщо $\forall i : i = \overline{1, n}$ виконується нерівність: $|a_{ii}| \geq \sum\limits_{j=1, j\neq i}^{n} |a_{ij}|$, то ітераційний процес методу Якобі збігається, при чому швидкість збіжності лінійна.

Умова припинення: $||x^n - x^{n-1}|| \leq \varepsilon.$

В якості норми зазвичай обирають неперервну (кубічну) норму вектору: $||x||_\infty = max_{1\leq i\leq n}|x_i|.$

Необхідні та достатні умови збіжності. Для $\forall x^0$ ітераційний процес методу збігається тоді і тільки тоді, коли $|\lambda| < 1$, де λ – це корені нелінійного рівняння:

$$\begin{vmatrix} \lambda a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & \lambda a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & \lambda a_{nn} \end{vmatrix} = 0.$$

При розв’язанні системи лінійних алгебраїчних рівнянь перевіряють достатні умови збіжності. Якщо в задачі необхідно щось довести, знайти область збіжності, то використовують необхідні і достатні умови збіжності.

Хід роботи

Мова програмування – Python. Починаємо з ініціалізації матриць та векторів в функції main, де також і викликаємо методи для кожної з них:

```
if __name__ == "__main__":
    # Gaussian elimination
    A1 = [
        [1.0, 2.0, 3.0, 0.0],
        [4.0, 3.0, 1.0, 2.0],
        [2.0, 1.0, 2.0, 1.0],
        [0.0, 3.0, 0.0, -5.0]
    ]
    b1 = [22.0, 30.0, 21.0, -21.0]
    x1, swaps1, inv_A1 = gaussian_elimination(A1, b1)

    # Tridiagonal (Thomas) method
    A2 = [
        [3.0, 2.0, 0.0],
        [2.0, 4.0, 1.0],
        [0.0, 1.0, 5.0]
    ]
    b2 = [9.0, 19.0, 28.0]
    x2 = tridiagonal_matrix_algorithm(A2, b2)
```

```
    # Yacobi method
    A3 = [
        [6.0, 0.0, 2.0, 3.0],
        [0.0, 4.0, 2.0, 1.0],
        [2.0, 2.0, 5.0, 0.0],
        [1.0, 1.0, 0.0, 3.0]
    ]
    b3 = [24.0, 18.0, 21.0, 15.0]
    x3 = jacobi_method(A3, b3, eps = 1e-3, max_iter = 50)
```

Функція для форматованого виводу матриць:

```
def print_matrix(A, b, title = ""):
    print(f"\n{title}")
    n = len(A)
    for i in range(n):
        row = "".join(f"{A[i][j]:8.3f}" for j in range(n))
        print(f"[{row} | {b[i]:8.3f}]")
    print("-" * (10 * n + 6))
```

Метод Гауса.

Копіюємо матрицю та вектор, щоб не змінювати вихідні дані при обчисленнях. Також ставимо лічильник свопів рядків та ініціалізуємо матрицю для обчислення оберненої. Виводимо початкову матрицю та починаємо прямий хід:

```
def gaussian_elimination(A, b):
    n = len(A)
    A = [row[:] for row in A]  # make a copy of A
    b = b[:]  # make a copy of b
    swaps = 0 # count of row swaps

    # store original matrix for inverse calculation
    A_orig = [row[:] for row in A]

    # add identity matrix for inverse calculation
    inv_A = [[1.0 if i == j else 0.0 for j in range(n)] for i in range(n)]

    print_matrix(A, b, " Initial Matrix:")
```

```python
# forward elimination
for k in range(n):
    # pivoting
    print(f"Step {k + 1}: Pivoting for column {k + 1}")
    max_row = max(range(k, n), key = lambda j: abs(A[j][k]))
    pivot = A[max_row][k]
    print(f"Pivot element: A[{max_row + 1}, {k + 1}] = {pivot:.3f}")
    if (abs(A[max_row][k]) < 1e-12):
        raise ValueError("Matrix is singular")

    # rows swap
    if max_row != k:
        print(f"Swapping rows {k + 1} and {max_row + 1}")
        A[k], A[max_row] = A[max_row], A[k]
        b[k], b[max_row] = b[max_row], b[k]
        inv_A[k], inv_A[max_row] = inv_A[max_row], inv_A[k]
        swaps += 1
        print_matrix(A, b, f" After swapping rows:")

    # elimination
    for i in range(k + 1, n):
        factor = A[i][k] / A[k][k]
        print(f"Eliminating A[{i + 1}, {k + 1}], factor = {factor:.3f}")
        for j in range(k, n):
            A[i][j] -= factor * A[k][j]
        b[i] -= factor * b[k]
        # inverse matrix update
        for j in range(n):
            inv_A[i][j] -= factor * inv_A[k][j]
    print_matrix(A, b, f" After elimination in column {k + 1}:")
```

Pivoting: вибір найбільшого елемента в стовпці для уникнення ділення на маленьке число. Якщо максимальний елемент не на діагоналі – міняємо рядки місцями (swap). Обчислюємо factor і приводимо елементи нижче діагоналі до нуля. Оновлюємо обернену матриці одночасно з прямим ходом.

```
dunnaya@MacBook-Air-Dana lab2 % python3 lab2v7.py

 Initial Matrix:
[   1.000   2.000   3.000   0.000 |   22.000]
[   4.000   3.000   1.000   2.000 |   30.000]
[   2.000   1.000   2.000   1.000 |   21.000]
[   0.000   3.000   0.000  -5.000 |  -21.000]
---------------------------------------------
Step 1: Pivoting for column 1
Pivot element: A[2, 1] = 4.000
Swapping rows 1 and 2
```

```
 After swapping rows:
[   4.000   3.000   1.000   2.000 |   30.000]
[   1.000   2.000   3.000   0.000 |   22.000]
[   2.000   1.000   2.000   1.000 |   21.000]
[   0.000   3.000   0.000  -5.000 |  -21.000]
----------------------------------------------
Eliminating A[2, 1], factor = 0.250
Eliminating A[3, 1], factor = 0.500
Eliminating A[4, 1], factor = 0.000
```

```
 After elimination in column 1:
[   4.000   3.000   1.000   2.000 |   30.000]
[   0.000   1.250   2.750  -0.500 |   14.500]
[   0.000  -0.500   1.500   0.000 |    6.000]
[   0.000   3.000   0.000  -5.000 |  -21.000]
----------------------------------------------
Step 2: Pivoting for column 2
Pivot element: A[4, 2] = 3.000
Swapping rows 2 and 4

 After swapping rows:
[   4.000   3.000   1.000   2.000 |   30.000]
[   0.000   3.000   0.000  -5.000 |  -21.000]
[   0.000  -0.500   1.500   0.000 |    6.000]
[   0.000   1.250   2.750  -0.500 |   14.500]
----------------------------------------------
Eliminating A[3, 2], factor = -0.167
Eliminating A[4, 2], factor = 0.417

 After elimination in column 2:
[   4.000   3.000   1.000   2.000 |   30.000]
[   0.000   3.000   0.000  -5.000 |  -21.000]
[   0.000   0.000   1.500  -0.833 |    2.500]
[   0.000   0.000   2.750   1.583 |   23.250]
----------------------------------------------
Step 3: Pivoting for column 3
Pivot element: A[4, 3] = 2.750
Swapping rows 3 and 4
```

```
 After swapping rows:
[   4.000   3.000   1.000   2.000 |   30.000]
[   0.000   3.000   0.000  -5.000 |  -21.000]
[   0.000   0.000   2.750   1.583 |   23.250]
[   0.000   0.000   1.500  -0.833 |    2.500]
----------------------------------------------
Eliminating A[4, 3], factor = 0.545

 After elimination in column 3:
[   4.000   3.000   1.000   2.000 |   30.000]
[   0.000   3.000   0.000  -5.000 |  -21.000]
[   0.000   0.000   2.750   1.583 |   23.250]
[   0.000   0.000   0.000  -1.697 |  -10.182]
----------------------------------------------
Step 4: Pivoting for column 4
Pivot element: A[4, 4] = -1.697

 After elimination in column 4:
[   4.000   3.000   1.000   2.000 |   30.000]
[   0.000   3.000   0.000  -5.000 |  -21.000]
[   0.000   0.000   2.750   1.583 |   23.250]
[   0.000   0.000   0.000  -1.697 |  -10.182]
----------------------------------------------
```

Далі починаємо зворотній хід:

```
# back substitution
print("\nBack substitution:")
x = [0.0] * n
for i in range(n - 1, -1, -1):
    s = sum(A[i][j] * x[j] for j in range(i + 1, n))
    x[i] = (b[i] - s) / A[i][i]
    print(f"x{i + 1} = ({b[i]:.4f} - {s:.4f}) / {A[i][i]:.4f} = {x[i]:.6f}")
```

Розв’язання системи від останнього до першого рівня. Вивід обчислень:

```
Back substitution:
x4 = (-10.1818 - 0.0000) / -1.6970 = 6.000000
x3 = (23.2500 - 9.5000) / 2.7500 = 5.000000
x2 = (-21.0000 - -30.0000) / 3.0000 = 3.000000
x1 = (30.0000 - 26.0000) / 4.0000 = 1.000000
```

Виводимо отримані корені та кількість свопів рядків:

```
print("\nFinal solution:")
for i, val in enumerate(x, start = 1):
    print(f"x{i} = {val:.6f}")
print(f"Total row swaps: {swaps}")
```

```
Final solution:
x1 = 1.000000
x2 = 3.000000
x3 = 5.000000
x4 = 6.000000
Total row swaps: 3
```

Перевіримо результат – підставимо отримані корені в початкову систему:

1*1	+2*3	+3*5	+0		22
4*1	+3*3	+1*5	+2*6	=	30
2*1	+1*3	+2*5	+1*6		21
0	+3*3	+0	-5*6		-21

Отриманий результат є правильним.

Обчислюємо визначник:

```
# determinant calculation
det = 1.0
for i in range(n):
    det *= A[i][i]
det *= (-1)**swaps
print(f"\nDeterminant of the matrix: {det:.6f}")
```

```
Determinant of the matrix: 56.000000
```

Шукаємо та виводимо обернену матрицю, додатково перевіряємо правильність рішення помноживши початкову матрицю на обернену:

```
# inverse matrix calculation
for k in range(n - 1, -1, -1):
    pivot = A[k][k]
    for j in range(n):
        inv_A[k][j] /= pivot
    A[k][k] = 1.0
    for i in range(k):
        factor = A[i][k]
        for j in range(n):
            inv_A[i][j] -= factor * inv_A[k][j]
        A[i][k] = 0.0
print_matrix(inv_A, b = [0] * n, title = " Inverse Matrix:")

# checking the result: inv(A) * A_orig should be identity
result = [[0.0 for _ in range(n)] for _ in range(n)]
for i in range(n):
    for j in range(n):
        result[i][j] = sum(inv_A[i][k] * A_orig[k][j] for k in range(n))
print_matrix(result, b = [0] * n, title = " Inverse * initial:")

return x, swaps, inv_A
```

```
 Inverse Matrix:
[  -0.607  -0.071   0.946   0.161 |     0.000]
[   0.536   0.357  -0.982  -0.054 |     0.000]
[   0.179  -0.214   0.339  -0.018 |     0.000]
[   0.321   0.214  -0.589  -0.232 |     0.000]
-------------------------------------------------

 Inverse * initial:
[   1.000  -0.000  -0.000  -0.000 |     0.000]
[   0.000   1.000   0.000   0.000 |     0.000]
[  -0.000  -0.000   1.000  -0.000 |     0.000]
[   0.000   0.000   0.000   1.000 |     0.000]
-------------------------------------------------
```

Отримали одиничну матрицю, а отже знайдена обернена матриця правильна.

Метод прогонки (Томаса).

Розбиваємо матрицю: a – піддіагональ, b – наддіагональ, c – головна діагональ, виводимо початкову матрицю та перевіряємо умову стійкості $|c_i| \geq |a_i| + |b_i|,\ i = \overline{0, n}$:

```python
def tridiagonal_matrix_algorithm(A, f):
    n = len(A)
    a = [0.0] + [A[i][i - 1] for i in range(1, n)] # sub-diagonal
    c = [A[i][i] for i in range(n)] # main diagonal
    b = [A[i][i + 1] for i in range(n - 1)] + [0.0] # super-diagonal (upper)

    print("\nTridiagonal Matrix Algorithm (Thomas Method):")
    print_matrix(A, f, " Initial Tridiagonal Matrix:")

    # stability check
    print(" Checking stability condition:")
    stable = all(abs(c[i]) >= abs(a[i]) + abs(b[i]) for i in range(n))
    if not stable:
        print("The stability condition of the Thomas method IS NOT met.")
    else:
        print("The stability condition of the Thomas method IS met.")
```

```
Tridiagonal Matrix Algorithm (Thomas Method):

 Initial Tridiagonal Matrix:
[   3.000   2.000   0.000 |    9.000]
[   2.000   4.000   1.000 |   19.000]
[   0.000   1.000   5.000 |   28.000]
------------------------------------
 Checking stability condition:
The stability condition of the Thomas method IS met.
```

Прямий хід – обчислюємо α та β для кожного рівняння:

```python
# forward sweep
print("\n Forward Sweep:")
alpha = [0.0] * n
beta = [0.0] * n
z = [0.0] * n

alpha[0] = -b[0] / c[0]
beta[0] = f[0] / c[0]
z[0] = c[0]
print(f"i = 1: α1 = {-b[0]:.3f} / {c[0]:.3f} = {alpha[0]:.3f}, "
      f"β1 = {f[0]:.3f} / {c[0]:.3f} = {beta[0]:.3f}")

for i in range(1, n):
    z[i] = c[i] + a[i] * alpha[i - 1]
    alpha[i] = -b[i] / z[i]
    beta[i] = (f[i] - a[i] * beta[i - 1]) / z[i]
    print(f"i = {i+1}: z{i+1} = {c[i]:.3f} + ({a[i]:.3f}) * ({alpha[i-1]:.3f}) = {z[i]:.3f}")
    print(f"      α{i+1} = {-b[i]:.3f} / {z[i]:.3f} = {alpha[i]:.3f}")
    print(f"      β{i+1} = ({f[i]:.3f} - {a[i]:.3f} * {beta[i-1]:.3f}) / {z[i]:.3f} = {beta[i]:.3f}")
```

```
 Forward Sweep:
i = 1: α1 = -2.000 / 3.000 = -0.667, β1 = 9.000 / 3.000 = 3.000
i = 2: z2 = 4.000 + (2.000) * (-0.667) = 2.667
      α2 = -1.000 / 2.667 = -0.375
      β2 = (19.000 - 2.000 * 3.000) / 2.667 = 4.875
i = 3: z3 = 5.000 + (1.000) * (-0.375) = 4.625
      α3 = -0.000 / 4.625 = -0.000
      β3 = (28.000 - 1.000 * 4.875) / 4.625 = 5.000
```

Знайшовши коефіцієнти, починаємо зворотній хід:

```
# back substitution
print("\n Back Substitution:")
x = [0.0] * n
x[- 1] = beta[- 1]
print(f"x{n} = β{n} = {x[-1]:.6f}")
for i in range(n - 2, -1, -1):
    x[i] = alpha[i] * x[i + 1] + beta[i]
    print(f"x{i+1} = α{i+1} * x{i+2} + β{i+1} = "
          f"{alpha[i]:.3f} * {x[i+1]:.3f} + {beta[i]:.3f} = {x[i]:.6f}")
```

```
 Back Substitution:
x3 = β3 = 5.000000
x2 = α2 * x3 + β2 = -0.375 * 5.000 + 4.875 = 3.000000
x1 = α1 * x2 + β1 = -0.667 * 3.000 + 3.000 = 1.000000
```

Виводимо результат:

```
# final solution
print("\n Final solution:")
for i in range(n):
    print(f"x{i + 1} = {x[i]:.6f}")

return x
```

```
 Final solution:
x1 = 1.000000
x2 = 3.000000
x3 = 5.000000
```

Перевіримо результат підставивши отримані корені в початкову систему:

3*1	+2*3	+0		9
2*1	+4*3	+1*5	=	19
0	+1*3	+5*5		28

Отриманий результат є правильним.

Метод Якобі.

Виводимо початкову матрицю та перевіряємо умову збіжності – метод збігається, якщо матриця діагонально домінантна:

```python
def jacobi_method(A, b, eps = 1e-3, max_iter = 100):
    n = len(A)
    x = [0.0] * n
    x_new = [0.0] * n

    print("\nJacobi Method:")
    print_matrix(A, b, " Initial Matrix:")

    # checking convergence condition
    print(" Checking convergence condition:")
    diag_dom = True
    for i in range(n):
        sum_row = sum(abs(A[i][j]) for j in range(n) if j != i)
        if abs(A[i][i]) < sum_row:
            diag_dom = False
            break
    if not diag_dom:
        print("The convergence condition for the Jacobi method IS NOT met.")
    else:
        print("The convergence condition for the Jacobi method IS met.")
```

```
Jacobi Method:

 Initial Matrix:
[   6.000   0.000   2.000   3.000 |    24.000]
[   0.000   4.000   2.000   1.000 |    18.000]
[   2.000   2.000   5.000   0.000 |    21.000]
[   1.000   1.000   0.000   3.000 |    15.000]
----------------------------------------------
 Checking convergence condition:
The convergence condition for the Jacobi method IS met.
```

Умова збіжності виконується, починаємо ітераційний процес:

```python
# iterations
print("\n Iteration process:")
print(f"Initial guess: x(0) = {[f'{v:.3f}' for v in x]}")

for k in range(max_iter + 1):
    print(f"\n Iteration {k}:")
    for i in range(n):
        s = sum(A[i][j] * x[j] for j in range(n) if j != i)
        x_new[i] = (b[i] - s) / A[i][i]
        terms = " + ".join([f"{A[i][j]:.3f} * {x[j]:.3f}" for j in range(n) if j != i])
        print(f"x{i+1}({k}) = (b{i+1} - ({terms})) / {A[i][i]:.3f} "
              f"= ({b[i]:.3f} - {s:.3f}) / {A[i][i]:.3f} = {x_new[i]:.6f}")
```

Для кожного рядка обчислюємо нове значення $x\_new[i]$ та виводимо проміжні обчислення кожної ітерації.

```python
    # compute infinity norm of the difference
    diff = max(abs(x_new[i] - x[i]) for i in range(n))
    print(f"‖Δx‖∞ = {diff:.3f}")

    # termination condition check
    if diff < eps:
        print(f"\nConvergence achieved after {k} iterations (ε = {eps}).")
        break
else:
    print("\nConvergence NOT achieved.")

    x = x_new.copy()
```

Перевіряємо умову зупинки: $||x^n - x^{n-1}|| \leq$ ε в кінці кожної ітерації:

```
 Iteration process:
Initial guess: x(0) = ['0.000', '0.000', '0.000', '0.000']

 Iteration 0:
x1(0) = (b1 - (0.000 * 0.000 + 2.000 * 0.000 + 3.000 * 0.000)) / 6.000 = (24.000 - 0.000) / 6.000 = 4.000000
x2(0) = (b2 - (0.000 * 0.000 + 2.000 * 0.000 + 1.000 * 0.000)) / 4.000 = (18.000 - 0.000) / 4.000 = 4.500000
x3(0) = (b3 - (2.000 * 0.000 + 2.000 * 0.000 + 0.000 * 0.000)) / 5.000 = (21.000 - 0.000) / 5.000 = 4.200000
x4(0) = (b4 - (1.000 * 0.000 + 1.000 * 0.000 + 0.000 * 0.000)) / 3.000 = (15.000 - 0.000) / 3.000 = 5.000000
‖Δx‖∞ = 5.000

Convergence NOT achieved.

 Iteration 1:
x1(1) = (b1 - (0.000 * 4.500 + 2.000 * 4.200 + 3.000 * 5.000)) / 6.000 = (24.000 - 23.400) / 6.000 = 0.100000
x2(1) = (b2 - (0.000 * 4.000 + 2.000 * 4.200 + 1.000 * 5.000)) / 4.000 = (18.000 - 13.400) / 4.000 = 1.150000
x3(1) = (b3 - (2.000 * 4.000 + 2.000 * 4.500 + 0.000 * 5.000)) / 5.000 = (21.000 - 17.000) / 5.000 = 0.800000
x4(1) = (b4 - (1.000 * 4.000 + 1.000 * 4.500 + 0.000 * 4.200)) / 3.000 = (15.000 - 8.500) / 3.000 = 2.166667
‖Δx‖∞ = 3.900

Convergence NOT achieved.

 Iteration 2:
x1(2) = (b1 - (0.000 * 1.150 + 2.000 * 0.800 + 3.000 * 2.167)) / 6.000 = (24.000 - 8.100) / 6.000 = 2.650000
x2(2) = (b2 - (0.000 * 0.100 + 2.000 * 0.800 + 1.000 * 2.167)) / 4.000 = (18.000 - 3.767) / 4.000 = 3.558333
x3(2) = (b3 - (2.000 * 0.100 + 2.000 * 1.150 + 0.000 * 2.167)) / 5.000 = (21.000 - 2.500) / 5.000 = 3.700000
x4(2) = (b4 - (1.000 * 0.100 + 1.000 * 1.150 + 0.000 * 0.800)) / 3.000 = (15.000 - 1.250) / 3.000 = 4.583333
‖Δx‖∞ = 2.900

Convergence NOT achieved.

 Iteration 3:
x1(3) = (b1 - (0.000 * 3.558 + 2.000 * 3.700 + 3.000 * 4.583)) / 6.000 = (24.000 - 21.150) / 6.000 = 0.475000
x2(3) = (b2 - (0.000 * 2.650 + 2.000 * 3.700 + 1.000 * 4.583)) / 4.000 = (18.000 - 11.983) / 4.000 = 1.504167
x3(3) = (b3 - (2.000 * 2.650 + 2.000 * 3.558 + 0.000 * 4.583)) / 5.000 = (21.000 - 12.417) / 5.000 = 1.716667
x4(3) = (b4 - (1.000 * 2.650 + 1.000 * 3.558 + 0.000 * 3.700)) / 3.000 = (15.000 - 6.208) / 3.000 = 2.930556
‖Δx‖∞ = 2.175

Convergence NOT achieved.
```

…

```
 Iteration 31:
x1(31) = (b1 - (0.000 * 2.001 + 2.000 * 3.000 + 3.000 * 4.000)) / 6.000 = (24.000 - 18.002) / 6.000 = 0.999723
x2(31) = (b2 - (0.000 * 1.001 + 2.000 * 3.000 + 1.000 * 4.000)) / 4.000 = (18.000 - 10.001) / 4.000 = 1.999738
x3(31) = (b3 - (2.000 * 1.001 + 2.000 * 2.001 + 0.000 * 4.000)) / 5.000 = (21.000 - 6.003) / 5.000 = 2.999322
x4(31) = (b4 - (1.000 * 1.001 + 1.000 * 2.001 + 0.000 * 3.000)) / 3.000 = (15.000 - 3.002) / 3.000 = 3.999435
‖Δx‖∞ = 0.001

Convergence NOT achieved.

 Iteration 32:
x1(32) = (b1 - (0.000 * 2.000 + 2.000 * 2.999 + 3.000 * 3.999)) / 6.000 = (24.000 - 17.997) / 6.000 = 1.000508
x2(32) = (b2 - (0.000 * 1.000 + 2.000 * 2.999 + 1.000 * 3.999)) / 4.000 = (18.000 - 9.998) / 4.000 = 2.000480
x3(32) = (b3 - (2.000 * 1.000 + 2.000 * 2.000 + 0.000 * 3.999)) / 5.000 = (21.000 - 5.999) / 5.000 = 3.000216
x4(32) = (b4 - (1.000 * 1.000 + 1.000 * 2.000 + 0.000 * 2.999)) / 3.000 = (15.000 - 2.999) / 3.000 = 4.000180
‖Δx‖∞ = 0.001

Convergence achieved after 32 iterations (ε = 0.001).
```

З обраною точністю $\varepsilon = 10^{-3}$ зупиняємось після 32 ітерації. Виводимо фінальний результат:

```
# final result
print("\n Final solution:")
for i in range(n):
    print(f"x{i + 1} = {x_new[i]:.6f}")

return x_new
```

```
 Final solution:
x1 = 1.000508
x2 = 2.000480
x3 = 3.000216
x4 = 4.000180
```

Перевіримо його, підставивши корені в початкову систему:

6*1	+0	+2*3	+3*4		24
0	+4*2	+2*3	+1*4	=	18
2*1	+2*2	+5*3	+0		21
1*1	+1*2	+0	+3*4		15

Отриманий результат є правильним.