**Міністерство Освіти І НАУКИ України**

**Національний університет "Львівська політехніка"**

Інститут **ІКНІ**

Кафедра **ПЗ**

**ЗВІТ**

До лабораторної роботи № 5

**На тему:** *“НАБЛИЖЕНІ МЕТОДИ РОЗВ’ЯЗУВАННЯ СИСТЕМ ЛІНІЙНИХ*

*АЛГЕБРАЇЧНИХ РІВНЯНЬ.”*

**З дисципліни:** *“Чисельні методи”*

**Лектор:**

доцент кафедри ПЗ

Мельник Н.Б.

**Виконав:**

студент групи ПЗ-18

Юшкевич А.І.

**Прийняв:**

професор кафедри ПЗ

Гавриш В.І.

Львів – 2023

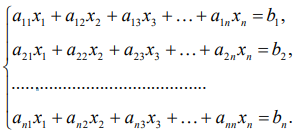
**Тема:** наближені методи розв’язування систем лінійних алгебраїчних рівнянь

**Мета роботи**: ознайомлення на практиці з методами Якобі та Зейделя розв’язування систем лінійних алгебраїчних рівнянь.

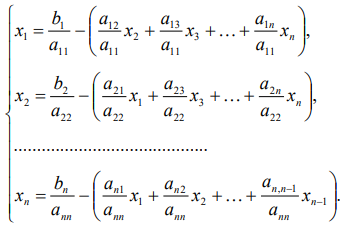
**ТЕОРЕТИЧНІ ВІДОМОСТІ**

**Метод простої ітерації (Якобі)**

Розглянемо систему лінійних алгебраїчних рівнянь виду:



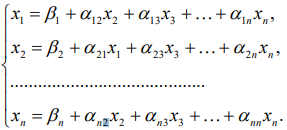
Припустивши, що коефіцієнти aii ≠ 0 (i 1,n) , розв’яжемо i-те рівняння системи відносно xі . У результаті отримаємо таку систему:



Введемо позначення



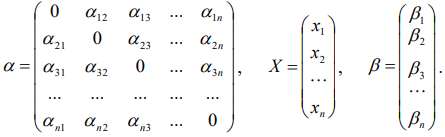
і перепишемо систему у вигляді



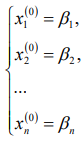
Таку систему називають зведеною. У матричному вигляді запишемо її так:



де



За нульове наближення розв’язку системи виберемо стовпець вільних членів, тобто



або



Перше наближення розв’язку системи знаходимо у вигляді

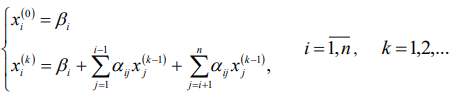


Аналогічно довільне наближення розв’язку системи визначимо співвідношенням



Якщо послідовність X1, X2,… Xn,… збігається, тобто , то граничний вектор X є розв’язком системи рівнянь, а отже і системи.

Формули перепишемо у розгорнутому вигляді



**Збіжність ітераційного процесу**

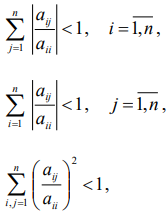
Збіжність ітераційного процесу залежить від величини коефіцієнтів матриці . Тому не обов’язково за нульове наближення розв’язку вибирати стовпець вільних членів. За початкове наближення X(0) можна також вибирати

 вектор, усі координати якого xi(0)  0 (i 1,n) ;

 вектор, усі координати якого xi(0)  1 (i 1,n) ;

 вектор X(0) , отриманий у результаті аналізу особливостей об’єкту дослідження та задачі, яку розв’язують.

Теорема (про збіжність ітераційного процесу). Якщо елементи матриці  системи рівнянь задовольняють одну з умов:



то система рівнянь має єдиний розв'язок X\*, який не залежить від початкового наближення X(0).

**Критерії припинення ітераційного процесу**

Якщо задана похибка  наближеного розв’язку, то критерієм припинення ітераційного процесу вважають виконання однієї з умов:

* модуль різниці між наступним та попереднім наближенням розв’язку повинен бути меншим за 



* максимальне значення модуля різниць між відповідними компонентами наступного та попереднього наближення розв’язку повинно бути меншим за 



* максимальне значення модуля відносних різниць між відповідними компонентами наступного та попереднього наближення розв’язку повинно бути меншим за 



Для попередження необґрунтованих витрат машинного часу для випадку, коли ітераційний процес не є збіжним для конкретної СЛАР, використовують лічильник кількості ітерацій, і при досягненні нею деякого заданого числа N обчислення припиняють.

**Метод Зейделя**

Даний метод є модифікацією методу простої ітерації. Основна його ідея полягає в тому, що при обчисленні чергового k-го наближення розв’язку хi(k) (i 1,n)використовують вже знайдені значення k-го наближеного розв’язку х1(k), х2(k), …, хi-1(k). Ітераційні формули методу Зейделя мають вигляд



Умова збіжності і критерій припинення ітераційного процесу за методом Зейделя є такими ж, як і для методу простої ітерації.

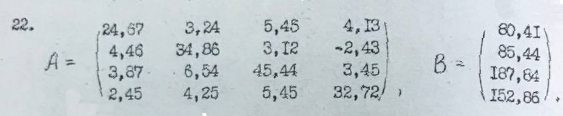
За методом Зейделя отримують кращу збіжність, ніж за методом Якобі, але доводиться проводити більш громіздкі обчислення. Крім того, ітераційний процес, проведений за методом Зейделя іноді буває збіжним у тому випадку, коли він є розбіжним за методом простої ітерації і навпаки.

**Індивідуальне завдання**

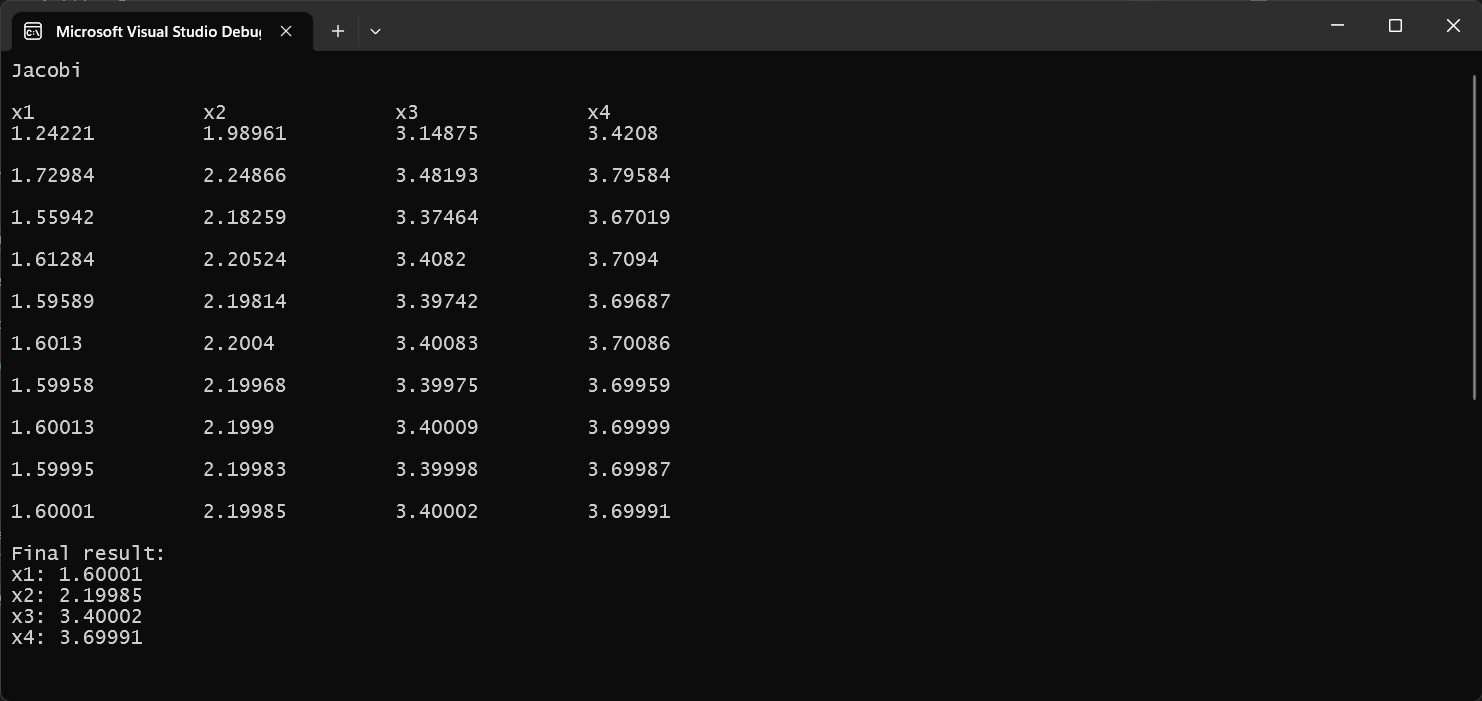
1. Ознайомитись з теоретичними відомостями.

2. Розв'язати систему нелінійних рівнянь з точністю eps = 10-3 методом ітерацій та методом Ньютона.

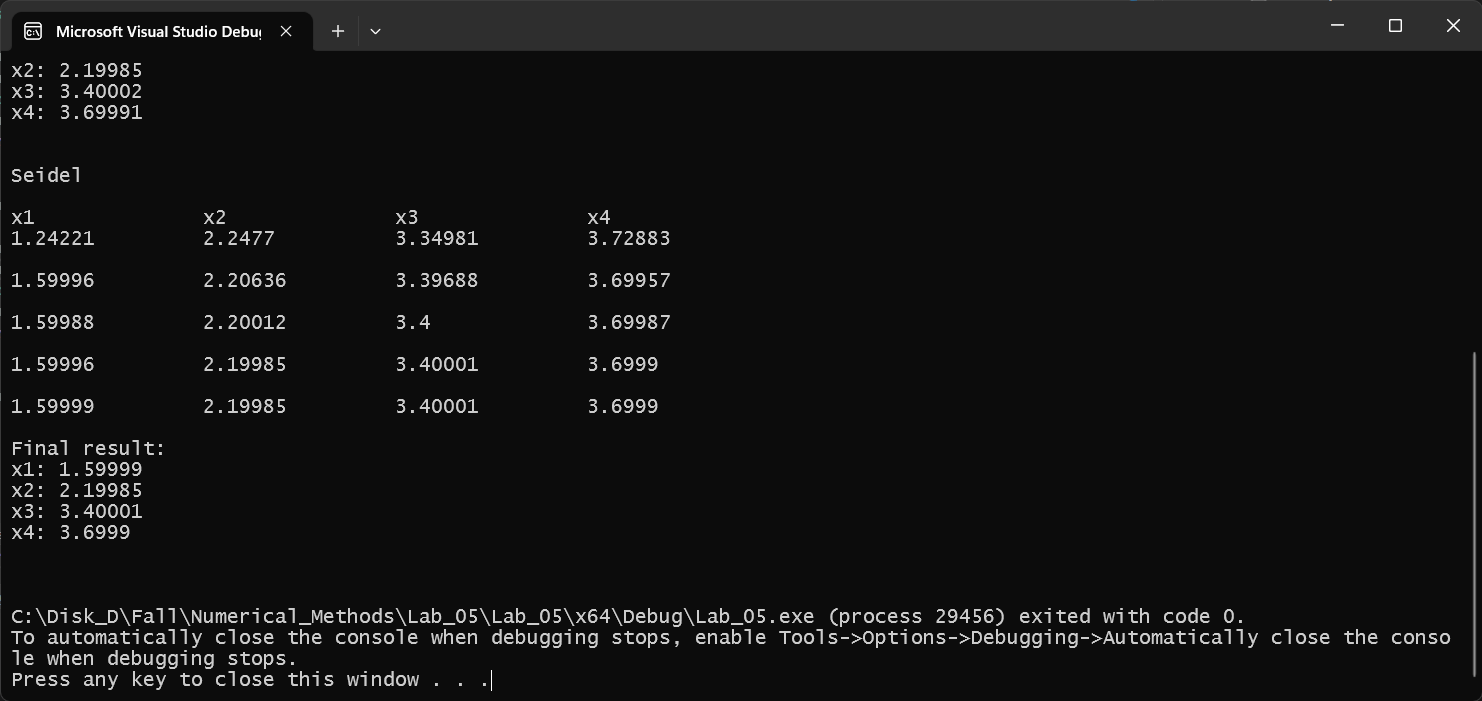
**Варіант 22**



**Результат виконання програми**

**

*Рис. 1. Зображення ітераційного процесу методу Якобі*

**

*Рис. 2. Зображення ітераційного процесу методу Зeйделя*

**Висновки**

У результаті виконання лабораторної роботи визначено дійсні корені заданої СЛАР з заданою точністю 10-3 методом Якобі та методом Зейделя. Розв’язок отриманий із заданою точністю, методом Якобі за 10 кроків ітерації, а методом Зейделя за 5.

**Код програми**

SystemSolver\_2\_Header.h:

#pragma once

#include <iostream>

#include <vector>

#include <string>

#include <cmath>

using namespace std;

typedef long double ldouble;

class SystemSolver\_2 {

public:

SystemSolver\_2() = delete;

template <class T>

SystemSolver\_2(const T matrix, const size\_t matrix\_size, const vector<ldouble> free\_terms);

template <class T>

SystemSolver\_2(const T matrix, const size\_t matrix\_size, const vector<ldouble> free\_terms, const ldouble epsilon);

vector<ldouble> Jacobi();

vector<ldouble> Seidel();

private:

vector<vector<ldouble>> m\_matrix;

vector<ldouble> m\_free\_terms;

size\_t m\_matrix\_size;

bool m\_is\_roots\_found;

ldouble m\_epsilon;

template <typename T>

vector<vector<ldouble>> CopyMatrix(const T matrix, const size\_t size);

bool IsMatrixConvergent() const;

void DivideByMaxElement();

void SetHollowMatrix(vector<vector<ldouble>>& hollow\_matrix, vector<ldouble>& new\_free\_terms) const;

void CheckIfRootsFound(const vector<ldouble> roots, const vector<ldouble> previous\_roots);

vector<ldouble> FindCloserJacobi(const vector<vector<ldouble>> hollow\_matrix, const vector<ldouble> new\_free\_terms, vector<ldouble> previous\_roots);

vector<ldouble> FindCloserSeidel(const vector<vector<ldouble>> hollow\_matrix, const vector<ldouble> new\_free\_terms, vector<ldouble> previous\_roots);

};

template <class T>

SystemSolver\_2::SystemSolver\_2(const T matrix, const size\_t matrix\_size, const vector<ldouble> free\_terms) {

m\_matrix = CopyMatrix(matrix, matrix\_size);

m\_free\_terms = free\_terms;

m\_matrix\_size = matrix\_size;

m\_is\_roots\_found = false;

m\_epsilon = 0.001;

}

template <class T>

SystemSolver\_2::SystemSolver\_2(const T matrix, const size\_t matrix\_size, const vector<ldouble> free\_terms, const ldouble epsilon) {

m\_matrix = CopyMatrix(matrix, matrix\_size);

m\_free\_terms = free\_terms;

m\_matrix\_size = matrix\_size;

m\_is\_roots\_found = false;

m\_epsilon = epsilon;

}

template <typename T>

vector<vector<ldouble>> SystemSolver\_2::CopyMatrix(const T matrix, const size\_t size) {

vector<vector<ldouble>> new\_vector(size, vector<ldouble>(size));

do {

if (new\_vector.empty())

break;

for (int i = 0; i < size; i++) {

for (int j = 0; j < size; j++) {

new\_vector[i][j] = matrix[i][j];

}

}

} while (false);

return new\_vector;

}

Functions.cpp:

#include "SystemSolver\_2\_Header.h"

vector<ldouble> SystemSolver\_2::Jacobi() {

if (!IsMatrixConvergent())

DivideByMaxElement();

vector<vector<ldouble>> hollow\_matrix(m\_matrix\_size, vector<ldouble>(m\_matrix\_size));

vector<ldouble> new\_free\_terms(m\_matrix\_size);

SetHollowMatrix(hollow\_matrix, new\_free\_terms);

vector<ldouble> roots(new\_free\_terms);

do {

roots = FindCloserJacobi(hollow\_matrix, new\_free\_terms, roots);

} while (!m\_is\_roots\_found);

m\_is\_roots\_found = false;

return roots;

}

vector<ldouble> SystemSolver\_2::Seidel() {

if (!IsMatrixConvergent())

DivideByMaxElement();

vector<vector<ldouble>> hollow\_matrix(m\_matrix\_size, vector<ldouble>(m\_matrix\_size));

vector<ldouble> new\_free\_terms(m\_matrix\_size);

SetHollowMatrix(hollow\_matrix, new\_free\_terms);

vector<ldouble> roots(new\_free\_terms);

do {

roots = FindCloserSeidel(hollow\_matrix, new\_free\_terms, roots);

} while (!m\_is\_roots\_found);

m\_is\_roots\_found = false;

return roots;

}

bool SystemSolver\_2::IsMatrixConvergent() const {

//Sum of elements of each row less than 1:

ldouble sum\_of\_row{ 0 };

bool is\_all\_rows\_less\_than\_1{true};

for (int i = 0; i < m\_matrix\_size; i++) {

for (int j = 0; j < m\_matrix\_size; j++) {

sum\_of\_row += m\_matrix[i][j];

}

if (sum\_of\_row > 1) {

is\_all\_rows\_less\_than\_1 = false;

break;

}

sum\_of\_row = 0;

}

if (is\_all\_rows\_less\_than\_1)

return true;

//Sum of elements of each column less than 1:

ldouble sum\_of\_column{ 0 };

bool is\_all\_columns\_less\_than\_1{true};

for (int i = 0; i < m\_matrix\_size; i++) {

for (int j = 0; j < m\_matrix\_size; j++) {

sum\_of\_column += m\_matrix[j][i];

}

if (sum\_of\_column > 1) {

is\_all\_columns\_less\_than\_1 = false;

break;

}

sum\_of\_column = 0;

}

if (is\_all\_columns\_less\_than\_1)

return true;

//Sum of elements of each row less than diagonal element:

ldouble sum\_of\_incompete\_row{ 0 };

bool is\_all\_rows\_less\_than\_diagonal{ true };

for (int i = 0; i < m\_matrix\_size; i++) {

for (int j = 0; j < m\_matrix\_size; j++) {

if (i != j)

sum\_of\_incompete\_row += m\_matrix[i][j];

}

if (sum\_of\_incompete\_row > m\_matrix[i][i]) {

is\_all\_rows\_less\_than\_diagonal = false;

break;

}

sum\_of\_incompete\_row = 0;

}

if (is\_all\_rows\_less\_than\_diagonal)

return true;

//Sum of elements of each column less than diagonal element:

ldouble sum\_of\_incompete\_column{ 0 };

bool is\_all\_columns\_less\_than\_diagonal{ true };

for (int i = 0; i < m\_matrix\_size; i++) {

for (int j = 0; j < m\_matrix\_size; j++) {

if (i != j)

sum\_of\_incompete\_column += m\_matrix[i][j];

}

if (sum\_of\_incompete\_column > m\_matrix[i][i]) {

is\_all\_columns\_less\_than\_diagonal = false;

break;

}

sum\_of\_incompete\_column = 0;

}

if (is\_all\_columns\_less\_than\_diagonal)

return true;

return false;

}

void SystemSolver\_2::DivideByMaxElement() {

ldouble max\_element{ 0 };

for (int i = 0; i < m\_matrix\_size; i++) {

for (int j = 0; j < m\_matrix\_size; j++) {

if (m\_matrix[i][j] > max\_element)

max\_element = m\_matrix[i][j];

}

}

max\_element \*= m\_matrix\_size + 1;

for (int i = 0; i < m\_matrix\_size; i++) {

for (int j = 0; j < m\_matrix\_size; j++) {

m\_matrix[i][j] /= max\_element;

}

m\_free\_terms[i] /= max\_element;

}

}

void SystemSolver\_2::SetHollowMatrix(vector<vector<ldouble>>& hollow\_matrix, vector<ldouble>& new\_free\_terms) const {

for (int i = 0; i < m\_matrix\_size; i++) {

new\_free\_terms[i] = m\_free\_terms[i] / m\_matrix[i][i];

hollow\_matrix[i][i] = 0;

for (int j = 0; j < m\_matrix\_size; j++) {

if (i == j)

continue;

hollow\_matrix[i][j] = -(m\_matrix[i][j] / m\_matrix[i][i]);

}

}

}

vector<ldouble> SystemSolver\_2::FindCloserJacobi(

const vector<vector<ldouble>> hollow\_matrix,

const vector<ldouble> new\_free\_terms,

vector<ldouble> previous\_roots) {

vector<ldouble> roots(m\_matrix\_size);

for (int i = 0; i < m\_matrix\_size; i++) {

roots[i] = 0;

for (int j = 0; j < m\_matrix\_size; j++) {

roots[i] += hollow\_matrix[i][j] \* previous\_roots[j];

}

roots[i] += new\_free\_terms[i];

}

for (int i = 0; i < roots.size(); i++) {

cout << roots[i] << "\t\t";

}

cout << endl << endl;

CheckIfRootsFound(roots, previous\_roots);

return roots;

}

vector<ldouble> SystemSolver\_2::FindCloserSeidel(

const vector<vector<ldouble>> hollow\_matrix,

const vector<ldouble> new\_free\_terms,

vector<ldouble> previous\_roots) {

vector<ldouble> roots(m\_matrix\_size);

vector<ldouble> previous\_roots\_copy(previous\_roots);

for (int i = 0; i < m\_matrix\_size; i++) {

roots[i] = 0;

for (int j = 0; j < m\_matrix\_size; j++) {

roots[i] += hollow\_matrix[i][j] \* previous\_roots\_copy[j];

}

roots[i] += new\_free\_terms[i];

previous\_roots\_copy[i] = roots[i];

}

for (int i = 0; i < roots.size(); i++) {

cout << roots[i] << "\t\t";

}

cout << endl << endl;

CheckIfRootsFound(roots, previous\_roots);

return roots;

}

void SystemSolver\_2::CheckIfRootsFound(const vector<ldouble> roots, const vector<ldouble> previous\_roots) {

// first condition

ldouble random\_bullshit{ 0 };

for (int i = 0; i < m\_matrix\_size; i++) {

random\_bullshit += pow(roots[i] - previous\_roots[i], 2);

}

random\_bullshit = pow(random\_bullshit, 0.5);

if (random\_bullshit < m\_epsilon)

m\_is\_roots\_found = true;

// second condition

random\_bullshit = 0;

for (int i = 0; i < m\_matrix\_size; i++) {

if (fabs(roots[i] - previous\_roots[i]) > random\_bullshit)

random\_bullshit = fabs(roots[i] - previous\_roots[i]);

}

if (random\_bullshit < m\_epsilon)

m\_is\_roots\_found = true;

}

Lab\_05.cpp:

#include <iostream>

#include "SystemSolver\_2\_Header.h"

int main()

{

const size\_t size{ 4 };

ldouble matrix[size][size] = { { 24.67, 3.24, 5.45, 4.13},

{ 4.46, 34.86, 3.12, -2.43},

{ 3.87, 6.54, 45.44, 3.45},

{ 2.45, 4.25, 5.45, 32.72} };

vector<ldouble> B{ 80.41, 85.44, 187.84, 152.86 };

const ldouble epsilon = 0.0001;

SystemSolver\_2 solver(matrix, size, B, epsilon);

cout << "Jacobi" << endl << endl;

cout << "x1\t\tx2\t\tx3\t\tx4\n";

vector<ldouble> result\_jacobi = solver.Jacobi();

cout << "Final result:\n";

for (int i = 0; i < result\_jacobi.size(); i++) {

cout << "x" << i + 1 << ": " << result\_jacobi[i] << endl;

}

cout << endl << endl;

cout << "Seidel" << endl << endl;

cout << "x1\t\tx2\t\tx3\t\tx4\n";

vector<ldouble> result\_seidel = solver.Seidel();

cout << "Final result:\n";

for (int i = 0; i < result\_seidel.size(); i++) {

cout << "x" << i + 1 << ": " << result\_seidel[i] << endl;

}

cout << endl << endl;

}