

Міністерство освіти і науки України
Національний університет “Львівська політехніка”
Інститут комп’ютерних наук та інформаційних технологій
Кафедра програмного забезпечення



Звіт

До лабораторної роботи №3

На теми: “Розв’язування систем лінійних алгебраїчних рівнянь методом Крамера та методом оберненої матриці”

З дисципліни: “Чисельні методи”

Лектор:

доц. каф. ПЗ
Мельник Н.Б.

Виконав:

ст. гр. ПЗ-18
Юшкевич А.І.

Прийняв:

проф. каф. ПЗ
Гавриш В.І.
« ... » ... 2023 р.

Σ = _____

Тема: розв'язування систем лінійних алгебраїчних рівнянь методом Крамера та методом оберненої матриці.

Мета: ознайомлення на практиці з методом Крамера та методом оберненої матриці розв'язування систем лінійних алгебраїчних рівнянь. Код програмної реалізації подано у додатку.

Завдання

Скласти програму розв'язування системи лінійних алгебраїчних рівнянь методом оберненої матриці та методом Крамера

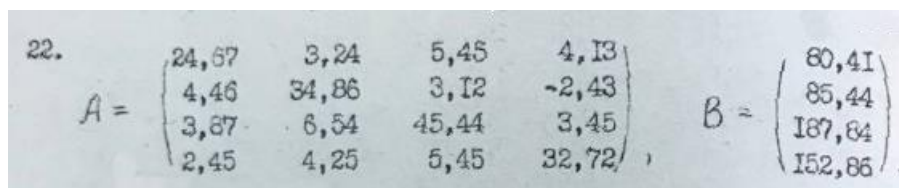

$$22. \quad A = \begin{pmatrix} 24,67 & 3,24 & 5,45 & 4,13 \\ 4,46 & 34,86 & 3,12 & -2,43 \\ 3,67 & 6,54 & 45,44 & 3,45 \\ 2,45 & 4,25 & 5,45 & 32,72 \end{pmatrix}, \quad B = \begin{pmatrix} 80,41 \\ 85,44 \\ 187,64 \\ 152,86 \end{pmatrix}.$$

Рис. 1. Система лінійних алгебраїчних рівнянь

Метод оберненої матриці (матричний метод)

У лінійній алгебрі часто використовують матричний метод розв'язування систем лінійних алгебраїчних рівнянь. Цей метод ґрунтується на обчисленні оберненої матриці A^{-1} , яка існує лише при умові, коли визначник матриці A відмінний від нуля $\det A \neq 0$. Якщо обидві частини матричного рівняння зліва домножити на матрицю A^{-1} , то отримаємо співвідношення $A^{-1}AX = A^{-1}B$. Враховуючи, що добуток оберненої матриці на саму матрицю дає одиничну матрицю, а результатом добутку одиничної матриці E на матрицю-стовпець X є матриця-стовпець X , тобто $EX = X$, одержимо матричний розв'язок системи лінійних алгебраїчних рівнянь у вигляді $x = A^{-1}B$.

Метод Крамера

Розглянемо СЛАР, яка містить n рівнянь та n невідомих, причому визначник її не дорівнює нулеві. Для знаходження невідомих x_i застосовують формули Крамера: $x_i = \frac{\det A_i}{\det A}$, $i = \underline{1, n}$, де $\det A$ - визначник матриці A , $\det A_i$ - визначник матриці A_i , яку отримують з матриці A шляхом заміни її i -го стовпця стовпцем вільних членів. Щоб розв'язати СЛАР з n невідомими потрібно обчислити $(n+1)$ визначників n -го порядку, що призводить до виконання $n \cdot n!$ операцій. Через громіздкість обчислень визначників метод Крамера не застосовують на практиці для великої розмірності матриці коефіцієнтів СЛАР.

Основні етапи обчислювального алгоритму для розв’язування системи лінійних алгебраїчних рівнянь методом оберненої матриці, реалізованого у програмному продукті мовою C++

- 1) внесення даних в конструкторі FunctionHolder() (рис. 2);
- 2) виклик метода MatrixMethod() (рис. 3), що реалізує знаходження коренів системи лінійних рівнянь матричним методом;
- 3) знаходження оберненої матриці реалізоване за допомогою методу FindReversedMatrix() (рис. 4), який в свою чергу використовує метод FindDeterminant() (рис. 5) для знаходження визначника заданої матриці та перевантаження цього методу – FindDeterminant() (рис. 6) для знаходження алгебраїчних доповнень;
- 4) знаходження добутку оберненої матриці та вільних членів у методі FindMultiplicationOfMatrixAndColumn() (рис. 7);
- 5) вивід результату виконання в консоль (рис. 8);
- 6) перевірка точності отриманого розв’язку системи лінійних рівнянь (рис. 11).

```
FunctionHolder(const T matrix, const ldouble* free_term, const int size) {
    this->matrix = AssignSMatrix(matrix, size);

    this->free_term.values = new ldouble[size];
    this->free_term.number_of_values = size;

    for (int i = 0; i < size; i++)
        this->free_term.values[i] = free_term[i];

    result.number_of_values = size;
    result.values = new ldouble[size];
}
```

Рис. 2. Конструктор FunctionHolder()

```
SColumn MatrixMethod() {
    SMatrix reversed = FindReversedMatrix(matrix);

    if (reversed.matrix) {
        result = FindMultiplicationOfMatrixAndColumn(reversed, free_term);

        for (int i = 0; i < result.number_of_values; i++)
            if (fabs(result.values[i]) <= 1e-13)
                result.values[i] = 0;
    }
    else {
        delete[] result.values;
        result.values = nullptr;
        result.number_of_values = 0;
    }

    return result;
}
```

Рис. 3. Метод MatrixMethod()

```

SMatrix FindReversedMatrix(const SMatrix& matrix) const {
    SMatrix local_matrix = { nullptr, 0 };
    ldouble determinant = FindDeterminant(matrix);

    if (fabs(determinant) <= 1e-13) {
        cout << "Reversed matrix does not exist. Try another method" << endl << endl;
    }
    else {
        local_matrix = GetNewSMatrix(matrix.size);

        for (int i = 0; i < matrix.size; i++) {
            for (int j = 0; j < matrix.size; j++) {
                local_matrix.matrix[j][i] = FindDeterminant(matrix, i, j) * (1 / determinant);
            }
        }
        return local_matrix;
    }
}

```

Рис. 4. Метод FindReversedMatrix()

```

ldouble FindDeterminant(const SMatrix matrix) const {
    int index = 0;
    if (matrix.size == 1)
        return matrix.matrix[0][0];

    SMatrix smaller_matrix = GetNewSMatrix(matrix.size - 1);

    ldouble determinant = 0;
    int column = 0;
    bool wrong_k_found = false;

    for (int i = 0; i < matrix.size; i++)
    {
        for (int j = 1; j < matrix.size; j++) {
            for (int k = 0; k < matrix.size; k++) {
                if (k == index) {
                    wrong_k_found = true;
                    continue;
                }

                if (wrong_k_found)
                    column = k - 1;
                else
                    column = k;

                smaller_matrix.matrix[j - 1][column] = matrix.matrix[j][k];
            }
            wrong_k_found = false;
        }

        determinant += pow(-1, i) * matrix.matrix[0][i] * FindDeterminant(smaller_matrix);
        index++;
    }

    return determinant;
}

```

Рис. 5. Метод FindDeterminant()

```

ldouble FindDeterminant(const SMatrix matrix, int row, int column) const {
    if (matrix.size == 1)
        return matrix.matrix[0][0];

    SMatrix smaller_matrix = GetNewSMatrix(matrix.size - 1);

    ldouble determinant = 0;
    int row_index = 0, column_index = 0;
    bool wrong_i_found = false, wrong_j_found = false;

    for (int i = 0; i < matrix.size; i++) {
        if (i == row) {
            wrong_i_found = true;
            continue;
        }
        if (wrong_i_found)
            row_index = i - 1;
        else
            row_index = i;

        for (int j = 0; j < matrix.size; j++) {
            if (j == column) {
                wrong_j_found = true;
                continue;
            }
            if (wrong_j_found)
                column_index = j - 1;
            else
                column_index = j;

            smaller_matrix.matrix[row_index][column_index] = matrix.matrix[i][j];
        }
        wrong_j_found = false;
    }

    determinant = pow(-1, row + column) * FindDeterminant(smaller_matrix);
    if (fabs(determinant) <= 1e-13)
        determinant = 0;
    return determinant;
}

```

Рис. 6. Метод FindDeterminant()

```

SColumn FindMultiplicationOfMatrixAndColumn(const SMatrix& matrix, const SColumn& column) const {
    SColumn local_column = { nullptr, column.number_of_values };
    local_column.values = new ldouble[local_column.number_of_values];
    ldouble sum = 0;
    for (int i = 0; i < matrix.size; i++) {
        for (int j = 0; j < matrix.size; j++) {
            sum += matrix.matrix[i][j] * column.values[j];
        }
        local_column.values[i] = sum;
        sum = 0;
    }

    return local_column;
}

```

Рис. 7. Метод FindMultiplicationOfMatrixAndColumn()

Matrix Method:

Root №1:	1.59999
Root №2:	2.19985
Root №3:	3.40001
Root №4:	3.6999

Рис. 8. Результат виконання програмної реалізації матричного методу

Основні етапи обчислювального алгоритму для розв'язування системи лінійних алгебраїчних рівнянь методом Крамера, реалізованого у програмному продукті мовою C++

- 1) внесення даних в конструкторі FunctionHolder() (рис. 2);
- 2) виклик метода Kramer() (рис. 3), що реалізує знаходження коренів системи лінійних рівнянь методом Крамера;
- 3) знаходження визначника за допомогою методу FindDeterminant() (рис. 5);
- 4) знаходження визначників для матриць, отриманих послідовною заміною кожного стовпця заданої матриці (рис. 1) методом FindDeterminantKramer() (рис. 9);
- 5) вивід результату виконання в консоль (рис. 10);
- 6) перевірка точності отриманого розв'язку системи лінійних рівнянь (рис. 11).

```
ldouble FindDeterminantKramer(const SMatrix& matrix, const SColumn& column, const int position) const {  
    SMatrix current = CopySMatrix(matrix);  
  
    for (int i = 0; i < current.size; i++) {  
        current.matrix[i][position] = column.values[i];  
    }  
  
    return FindDeterminant(current);  
}
```

Рис. 9. Метод FindDeterminantKramer()

Kramer :

Root №1:	1.59999
Root №2:	2.19985
Root №3:	3.40001
Root №4:	3.6999

Рис. 10. Результат виконання програмної реалізації методу Крамера

Accuracy of the solution of the system:

method result	free term
80.41	80.41
85.44	85.44
187.84	187.84
152.86	152.86

Рис. 11. Перевірка точності отриманого розв'язку системи лінійних рівнянь

Висновки

У результаті виконання лабораторної роботи я розробив програму розв'язування системи лінійних алгебраїчних рівнянь методом оберненої матриці та методом Крамера для заданої системи лінійних алгебраїчних рівнянь.

Додаток

MethodsHeader.h:

```
#pragma once
#include <cmath>

using namespace std;

typedef long double ldouble;

struct SMatrix {
    ldouble** matrix;
    int size;
};

struct SColumn {
    ldouble* values;
    int number_of_values;
};

template <class T>
class FunctionHolder {
private:
    SMatrix matrix;
    SColumn free_term;
    SColumn result;

    ldouble FindDeterminant(const SMatrix matrix) const {
        int index = 0;
        if (matrix.size == 1)
            return matrix.matrix[0][0];

        SMatrix smaller_matrix = GetNewSMatrix(matrix.size - 1);

        ldouble determinant = 0;
        int column = 0;
        bool wrong_k_found = false;

        for (int i = 0; i < matrix.size; i++)
        {
            for (int j = 1; j < matrix.size; j++) {
                for (int k = 0; k < matrix.size; k++) {
                    if (k == index) {
                        wrong_k_found = true;
                        continue;
                    }
                }

                if (wrong_k_found)
                    column = k - 1;
            }
        }
    }
};
```

```

        else
            column = k;

            smaller_matrix.matrix[j - 1][column] = matrix.matrix[j][k];
        }
        wrong_k_found = false;
    }

    determinant += pow(-1, i) * matrix.matrix[0][i] *
FindDeterminant(smaller_matrix);
    index++;
}

return determinant;
}

ldouble FindDeterminant(const SMatrix matrix, int row, int column) const {
    if (matrix.size == 1)
        return matrix.matrix[0][0];

    SMatrix smaller_matrix = GetNewSMatrix(matrix.size - 1);

    ldouble determinant = 0;
    int row_index = 0, column_index = 0;
    bool wrong_i_found = false, wrong_j_found = false;

    for (int i = 0; i < matrix.size; i++) {
        if (i == row) {
            wrong_i_found = true;
            continue;
        }
        if (wrong_i_found)
            row_index = i - 1;
        else
            row_index = i;

        for (int j = 0; j < matrix.size; j++) {
            if (j == column) {
                wrong_j_found = true;
                continue;
            }
            if (wrong_j_found)
                column_index = j - 1;
            else
                column_index = j;

            smaller_matrix.matrix[row_index][column_index] =
matrix.matrix[i][j];
        }
        wrong_j_found = false;
    }

    determinant = pow(-1, row + column) * FindDeterminant(smaller_matrix);
    if (fabs(determinant) <= 1e-13)
        determinant = 0;
    return determinant;
}

ldouble FindDeterminantKramer(const SMatrix& matrix, const SColumn& column, const int
position) const {
    SMatrix current = CopySMatrix(matrix);

    for (int i = 0; i < current.size; i++) {
        current.matrix[i][position] = column.values[i];
    }

```



```

        return FindDeterminant(current);
    }
    SMatrix FindReversedMatrix(const SMatrix& matrix) const {
        SMatrix local_matrix = { nullptr, 0 };
        ldouble determinant = FindDeterminant(matrix);

        if (fabs(determinant) <= 1e-13) {
            cout << "Reversed matrix does not exist. Try another method" << endl << endl;
        }
        else {
            local_matrix = GetNewSMatrix(matrix.size);

            for (int i = 0; i < matrix.size; i++) {
                for (int j = 0; j < matrix.size; j++) {
                    local_matrix.matrix[j][i] = FindDeterminant(matrix, i, j) * (1
/determinant);
                }
            }
        }
        return local_matrix;
    }

    SColumn FindMultiplicationOfMatrixAndColumn(const SMatrix& matrix, const SColumn&
column) const {
        SColumn local_column = { nullptr, column.number_of_values };
        local_column.values = new ldouble[local_column.number_of_values];
        ldouble sum = 0;
        for (int i = 0; i < matrix.size; i++) {
            for (int j = 0; j < matrix.size; j++) {
                sum += matrix.matrix[i][j] * column.values[j];
            }
            local_column.values[i] = sum;
            sum = 0;
        }

        return local_column;
    }

    SMatrix AssignSMatrix(const T matrix, int size) {
        SMatrix values = { nullptr, 0 };
        bool is_allocated = true;

        ldouble** local_matrix = new ldouble * [size];
        if (local_matrix) {
            for (int i = 0; i < size; i++) {
                if (local_matrix + i) {
                    local_matrix[i] = new ldouble[size];
                    for (int j = 0; j < size; j++)
                        local_matrix[i][j] = matrix[i][j];
                }
                else {
                    is_allocated = false;
                }
            }
        }
        else {
            is_allocated = false;
        }

        if (is_allocated) {
            values.matrix = local_matrix;
            values.size = size;
        }
    }

```

```

        return values;
    }
    SMatrix GetNewSMatrix(const int size) const {
        SMatrix values = { nullptr, 0 };
        bool is_allocated = true;

        ldoube** matrix = new ldoube * [size];
        if (matrix) {
            for (int i = 0; i < size; i++) {
                if (matrix + i) {
                    matrix[i] = new ldoube[size];
                }
                else {
                    is_allocated = false;
                }
            }
        }
        else {
            is_allocated = false;
        }

        if (is_allocated) {
            values.matrix = matrix;
            values.size = size;
        }
        return values;
    }
    SMatrix CopySMatrix(const SMatrix& arr) const {
        SMatrix values = { nullptr, 0 };
        bool is_allocated = true;

        ldoube** matrix = new ldoube * [arr.size];
        if (matrix) {
            for (int i = 0; i < arr.size; i++) {
                if (matrix + i) {
                    matrix[i] = new ldoube[arr.size];
                    for (int j = 0; j < arr.size; j++)
                        matrix[i][j] = arr.matrix[i][j];
                }
                else {
                    is_allocated = false;
                }
            }
        }
        else {
            is_allocated = false;
        }

        if (is_allocated) {
            values.matrix = matrix;
            values.size = arr.size;
        }
        return values;
    }
}

public:
    FunctionHolder(const T matrix, const ldoube* free_term, const int size) {
        this->matrix = AssignSMatrix(matrix, size);

        this->free_term.values = new ldoube[size];
        this->free_term.number_of_values = size;

        for (int i = 0; i < size; i++)
            this->free_term.values[i] = free_term[i];
    }

```

```

        result.number_of_values = size;
        result.values = new ldouble[size];
    }
    FunctionHolder(const FunctionHolder& other) {
        this->matrix = AssignSMatrix(other.matrix.matrix, other.matrix.size);

        this->free_term.number_of_values = other.free_term.number_of_values;
        this->free_term.values = new ldouble[this->free_term.number_of_values];

        for (int i = 0; i < this->free_term.number_of_values; i++)
            this->free_term.values[i] = other.free_term.values[i];

        result.number_of_values = this->matrix.size;
        result.values = new ldouble[size];
    }
    ~FunctionHolder() {
        delete[] free_term.values;
        delete[] result.values;

        for (int i = 0; i < matrix.size; i++)
            delete[] matrix.matrix[i];
        delete[] matrix.matrix;
    }

    void SetSMatrix(const T matrix, const int size) {
        FunctionHolder(matrix, size);
    }
    SMatrix GetSMatrix() const {
        return matrix;
    }

    SColumn Kramer() {

        ldouble main_determinant = 0;
        ldouble* var_determinants = new ldouble[free_term.number_of_values];

        if (!matrix.matrix || !var_determinants) {
            cout << "Set a matrix!" << endl << endl;
        }
        else {

            main_determinant = FindDeterminant(matrix);

            for (int i = 0; i < free_term.number_of_values; i++) {
                var_determinants[i] = FindDeterminantKramer(matrix, free_term, i);
            }

            if (main_determinant == 0) {
                bool var_are_all_zeros = true;
                for (int i = 0; i < free_term.number_of_values; i++) {
                    if (var_determinants[i] != 0) {
                        var_are_all_zeros = false;
                        break;
                    }
                }
                if (var_are_all_zeros) {
                    cout << "System is undefined. Try to use other method" <<
endl << endl;

                    result.number_of_values = 0;
                    delete[] result.values;

```

```

        result.values = nullptr;
    }
    else {
        cout << "System is incompatible." << endl << endl;
        result.number_of_values = 0;
        delete[] result.values;
        result.values = nullptr;
    }
}
else {
    for (int i = 0; i < free_term.number_of_values; i++)
        result.values[i] = var_determinants[i] / main_determinant;
}

return result;
}

}

SColumn MatrixMethod() {
    SMatrix reversed = FindReversedMatrix(matrix);

    if (reversed.matrix) {
        result = FindMultiplicationOfMatrixAndColumn(reversed, free_term);

        for (int i = 0; i < result.number_of_values; i++)
            if (fabs(result.values[i]) <= 1e-13)
                result.values[i] = 0;
    }
    else {
        delete[] result.values;
        result.values = nullptr;
        result.number_of_values = 0;
    }

    return result;
}

};

```

Lab_03_NM.cpp:

```

#include <iostream>
#include "..\Methods_Lib\MethodsHeader.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 } };

    ldouble free_terms[size] = { 80.41, 85.44, 187.84, 152.86 };

    FunctionHolder<ldouble(*)[size]> Fh(matrix, free_terms, size);
    SColumn kramer = Fh.Kramer();
    SColumn matrix_method = Fh.MatrixMethod();
}

```

```

cout << "Kramer: " << endl << endl;
for (int i = 0; i < kramer.number_of_values; i++) {
    cout << "Root №" << i + 1 << ": " << kramer.values[i] << endl;
}
cout << endl << endl << endl;

cout << "Matrix Method: " << endl << endl;
for (int i = 0; i < matrix_method.number_of_values; i++) {
    cout << "Root №" << i + 1 << ": " << matrix_method.values[i] << endl;
}
cout << endl << endl << endl;

cout << "Accuracy of the solution of the system: " << endl << endl << "method result \t free term" << endl;

double sum{ 0 };
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        sum += matrix[i][j] * kramer.values[j];
    }

    cout << " " << sum << "\t\t " << free_terms[i] << endl;
    sum = 0;
}

return 0;
}

```