НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. Ігоря СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

Протокол
до комп'ютерного практикуму №2

РОЗВ'ЯЗАННЯ СИСТЕМ ЛІНІЙНИХ АЛГЕБРАЇЧНИХ РІВНЯНЬ ПРЯМИМИ МЕТОДАМИ

Виконав студент
групи ФІ-81

Шкаліков О.В.

Перевірила:
Стьопочкіна І.В.

Київ — 2021

Теоретичні відомості

У даному комп'ютерному практикумі перед нами постає задача знаходження розв'зку системи лінійних алгебраїчних рівнянь, яку можна представити у вигляді:

$$Ax = b \tag{1}$$

,де A - матриця коефіцієнтів системи розмірності $n \times n$, x - невідомий вектор розмірності n, b - вектор правої частини розмірності n.
Взагалі кажучи, існують декілька ідеї методів, для розв'язання цієї задачі. Ми розглянемо так звані прямі методи, для знаходження коренів. Якщо матриця A системи 1 є симетричною, то можемо спробувати застосувати так званий метод квадратного кореня. Він полягає у тому, щоб застосувати декомпозицію Холецького та представити систему у наступному вигляді:

$$Ax = LL^T x = b \tag{2}$$

де L - нижньо-трикутна матриця. Отже, маємо:

$$\begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & & \\ \vdots & \ddots & \\ a_{n1} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} a_{11} & \dots & a_{n1} \\ & \ddots & \vdots \\ & & a_{nn} \end{pmatrix}$$

За правилами множення матриць, отримаємо:

$$\begin{gathered} l_{11} = \sqrt{a_{11}} \qquad l_{i1} = \frac{a_{i1}}{l_{11}} \; (i > 1) \\ l_{ij} = \frac{a_{ij} - \sum\limits_{k=1}^{i-1} l_{ik} l_{jk}}{l_{ij}} \; (1 < j \leq i) \end{gathered} \tag{3}$$

Маючи декомпозицію Холецького ми можемо розв'зати систему 1 у два кроки:

$$Ly = b \qquad L^T x = y$$

Зазначимо, що матриці L та L^T - нижньо- та верхньо-трикутні відповідно, тому їх можна розв'язати методами підстановки. Наведемо алгоритми цих методів.

Algorithm 1: Пряма підстановка(для нижньо-трикутної матриці)

Input: A, b

$x_1 \leftarrow \frac{b_1}{a_{11}}$

for $i \in \overline{2, n}$ **do**

$\quad x_i \leftarrow \frac{b_i - \sum\limits_{j=1}^{i-1} a_{ij} x_j}{a_{ii}}$

end

Output: x

Algorithm 2: Зворотна підстановка(для верхньо-трикутної матриці)

Input: A, b

$x_n \leftarrow \dfrac{b_n}{a_{nn}}$

for $i \in \{n-1, \ldots, 1\}$ **do**

$\quad x_i \leftarrow \dfrac{b_i - \sum\limits_{j=i+1}^{n} a_{ij} x_j}{a_{ii}}$

end

Output: x

Інколи, застосування операції квадратного кореня значно ускладнене, бо може призвести до значних чисельних похибок, або підкореневий вираз є від'ємним(і потрібно переходити у комплексну площину). Тому можна вдосконалити попередні алгоритми, та представити систему у вигляді:

$$Ax = LDL^T x = b \tag{4}$$

де L - нижньо-трикутна матриця з одиницями на діагоналі, D - діагональна матриця. За правилами множення матриць, отримаємо:

$$\begin{gathered} d_1 = a_{11} \qquad l_{ii} = 1 \qquad l_{i1} = \frac{a_{i1}}{d_i} \; (i > 1) \\ d_i = a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2 d_k \; (i > 1) \qquad l_{ij} = \frac{a_{ij} - \sum\limits_{k=1}^{i-1} l_{ik} l_{jk} d_k}{d_j} \; (1 < j < i) \end{gathered} \tag{5}$$

Практична частина

Розглянемо застосування описаних методів на прикладі наступної системи:

$$A = \begin{pmatrix} 5.5 & 7 & 6 & 5.5 \\ 7 & 10.5 & 8 & 7 \\ 6 & 8 & 10.5 & 9 \\ 5.5 & 7 & 9 & 10.5 \end{pmatrix} \qquad b = \begin{pmatrix} 23 \\ 32 \\ 33 \\ 31 \end{pmatrix}$$

Помітимо, що матриця A цієї системи симметрична, тому ми можемо спробувати застосувати декомпозицію Холецького та LDL.

Декомпозиція Холецького

У навчальних цілях наведемо результат роботи(заповнення матриці L) алгоритму 3 на кожній ітерації зовнішнього циклу. Зазначимо, що "початкова"матриця нульова і кожне число округлене до 6 значущих цифр.

$$L_1 = \begin{pmatrix} 2.34521 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \qquad L_2 = \begin{pmatrix} 2.34521 & 0 & 0 & 0 \\ 2.98481 & 1.26131 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$L_3 = \begin{pmatrix} 2.34521 & 0 & 0 & 0 \\ 2.98481 & 1.26131 & 0 & 0 \\ 2.55841 & 0.2883 & 1.96759 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \qquad L_4 = \begin{pmatrix} 2.34521 & 0 & 0 & 0 \\ 2.98481 & 1.26131 & 0 & 0 \\ 2.55841 & 0.2883 & 1.96759 & 0 \\ 2.34521 & 0 & 1.52471 & 1.63563 \end{pmatrix}$$

Таким чином отримаємо:

$$Ax = LL^T x = b$$
$$Ly = b \qquad L^T x = y$$

Тепер розв'яжемо системи за допомогою алгоритмів 1 та 2 відповідно(округлені до 16 значущої цифри).

$$y = \begin{pmatrix} 9.807232952358079 \\ 2.162249910469345 \\ 3.702853335674256 \\ 1.43935204774342 \end{pmatrix} \qquad x = \begin{pmatrix} 0.1599999999999985 \\ 1.440000000000001 \\ 1.199999999999999 \\ 0.8800000000000007 \end{pmatrix}$$

При обрахунку вектора нев'язки $r = Ax - b$, ми отримали нульовий вектор, що пов'язуємо з тим, що отриманий результат дуже близький до точного кореня $(0.16, 1.44, 1.2, 0.88)$. Та усі розбіжності нівелюються точність операції над типом *double*, який має обжену кількість розвядів у двійковому представленні.

LDL декомпозиція

Як і у попередньому прикладі наведемо результат роботи алгоритму 5 на кожній ітерації зовнішнього циклу.

$$L_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} d_1 = \begin{pmatrix} 5.5 \\ 0 \\ 0 \\ 0 \end{pmatrix} \qquad L_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1.27273 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} d_2 = \begin{pmatrix} 5.5 \\ 1.59091 \\ 0 \\ 0 \end{pmatrix}$$

$$L_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1.27273 & 1 & 0 & 0 \\ 1.09091 & 0.228571 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \qquad d_3 = \begin{pmatrix} 5.5 \\ 1.59091 \\ 3.87143 \\ 0 \end{pmatrix}$$

$$L_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1.27273 & 1 & 0 & 0 \\ 1.09091 & 0.228571 & 1 & 0 \\ 1 & 0 & 0.774908 & 1 \end{pmatrix} d_4 = \begin{pmatrix} 5.5 \\ 1.59091 \\ 3.87143 \\ 2.67528 \end{pmatrix}$$

Таким чином отримаємо:

$$Ax = LDL^T x = b$$

$$Ly = b \quad Dz = y \quad L^T x = z$$

Тепер розв'яжемо системи за допомогою алгоритмів 1 та 2(округлені до 16 значущої цифри).

$$y = \begin{pmatrix} 23 \\ 2.727272727272727 \\ 7.285714285714285 \\ 2.354243542435427 \end{pmatrix} \qquad z = \begin{pmatrix} 4.181818181818182 \\ 1.714285714285713 \\ 1.881918819188191 \\ 0.8800000000000009 \end{pmatrix} \qquad x = \begin{pmatrix} 0.1600000000000028 \\ 1.439999999999998 \\ 1.199999999999999 \\ 0.8800000000000009 \end{pmatrix}$$

При обрахунку вектора нев'язки $r = Ax - b$, ми отримали нульовий вектор, що пов'язуємо з тим, що отриманий результат дуже близький до точного кореня $(0.16, 1.44, 1.2, 0.88)$. Та усі розбіжності нівелюються точність операції над типом *double*, який має обжену кількість розвядів у двійковому представленні.

Додатки

Далі наведено програмний код імплементованих алгоритмів. Вихідний код, який було створено для даного практикума (у тому числі LaTeX), можна знайти за наступним посиланням.

Класс для інкупсуляції роботи з пам'ятю

```cpp
#pragma once

#include <cstddef>
#include <algorithm>

using std::size_t;


namespace LES
{
    template <typename T>
    class Buffer
    {
    public:
        Buffer(size_t size)
        {
            _size = size;
            _data = new T[size]();
        }

        Buffer(const Buffer &lhs)
        {
            _data = new T[lhs._size];
            _size = lhs._size;
            std::copy(lhs._data, lhs._data + lhs._size, _data);
        }

        Buffer(Buffer &&lhs) noexcept
        {
            _data = std::exchange(lhs._data, nullptr);
            _size = std::exchange(lhs._size, 0);
        }

        ~Buffer()
        {
            delete[] _data;
        }

        Buffer& operator=(const Buffer &lhs)
        {
            T *_newData = new T[lhs._size];
```

```
            std::copy(lhs._data, lhs._data + lhs._size, _newData)
              ↪ ;

            _size = lhs._size;
            delete[] _data;
            _data = _newData;

            return *this;
        }

        Buffer& operator=(Buffer &&lhs) noexcept
        {
            std::swap(_data, lhs._data);
            std::swap(_size, lhs._size);
            return *this;
        }

        T& operator[](std::size_t pos)
        {
            return _data[pos];
        }

        const T& operator[](std::size_t pos) const
        {
            return _data[pos];
        }

        inline size_t size() const noexcept { return _size; }

    private:
        T *_data;
        size_t _size;
    };
}
```

Клас векторів

```
#pragma once

#include <cmath>
#include "Buffer.h"

namespace LES
{
    template <typename T>
    class Vector
    {
    public:
        Vector(size_t nval) : _buff(nval) {}

        template<typename U>
```

```
    Vector(const Vector<U>& vector) : _buff(vector.nval())
    {
        for (size_t i = 0; i < vector.nval(); i++)
        {
            _buff[i] = vector(i);
        }
    };

    inline size_t nval() const noexcept { return _buff.size()
       ↪ ; }

    T& operator()(size_t i);
    const T& operator()(size_t i) const;

    auto norm() const noexcept;

private:
    Buffer<T> _buff;
};

template <typename T>
T& Vector<T>::operator()(size_t i)
{
    if (i >= _buff.size())
        throw std::invalid_argument("Position␣is␣out␣of␣range
           ↪ ");
    return _buff[i];
}

template <typename T>
const T& Vector<T>::operator()(size_t i) const
{
    if (i >= _buff.size())
        throw std::invalid_argument("Position␣is␣out␣of␣range
           ↪ ");
    return _buff[i];
}

template <typename T>
auto Vector<T>::norm() const noexcept
{
    T sum = 0;
    for (size_t i = 0; i < _buff.size(); i++)
    {
        sum += _buff[i] * _buff[i];
    }
    return std::sqrt(sum);
}

template <typename T, typename U>
```

```
    auto operator+(const Vector<T> &rhs, const Vector<U> &lhs)
    {
        if (rhs.nval() != lhs.nval())
            throw std::invalid_argument("Different␣size");

        Vector<typename std::common_type<T, U>::type> result(rhs.
           ↪ nval());
        for (size_t i = 0; i < rhs.nval(); i++)
        {
            result(i) = rhs(i) + lhs(i);
        }
        return result;
    }

    template <typename T, typename U>
    auto operator-(const Vector<T> &rhs, const Vector<U> &lhs)
    {
        if (rhs.nval() != lhs.nval())
            throw std::invalid_argument("Different␣size");

        Vector<typename std::common_type<T, U>::type> result(rhs.
           ↪ nval());
        for (size_t i = 0; i < rhs.nval(); i++)
        {
            result(i) = rhs(i) - lhs(i);
        }
        return result;
    }
}
```

Клас матриць

```
#pragma once

#include "Buffer.h"
#include "Vector.h"

namespace LES
{
    template <typename T>
    class IMatrix
    {
    public:
        size_t nrow() const noexcept { return _nrow; }
        size_t ncol() const noexcept { return _ncol; }

        virtual T &operator()(size_t i, size_t j) = 0;
        virtual const T& operator()(size_t i, size_t j) const =
           ↪ 0;

    protected:
```

```
    size_t _nrow;
    size_t _ncol;
};

template <typename T>
class TransposeView : public IMatrix<T>
{
public:
    TransposeView(IMatrix<T> &matrix)
    {
        _matrix = &matrix;
        this->_nrow = matrix.nrow();
        this->_ncol = matrix.ncol();
    }

    T &operator()(size_t i, size_t j) override
    {
        if (j >= this->_nrow || i >= this->_ncol)
            throw std::invalid_argument("Position␣is␣out␣of␣
              ↪ range");
        return _matrix->operator()(j, i);
    }

    const T& operator()(size_t i, size_t j) const override
    {
        if (j >= this->_nrow || i >= this->_ncol)
            throw std::invalid_argument("Position␣is␣out␣of␣
              ↪ range");
        return _matrix->operator()(j, i);
    }


private:
    IMatrix<T>* _matrix;
};

template <typename T>
class Matrix : public IMatrix<T>
{
public:
    Matrix(size_t nrow, size_t ncol) : _buff(nrow * ncol)
    {
        this->_nrow = nrow;
        this->_ncol = ncol;
    }

    template<typename U>
    Matrix(const Matrix<U>& matrix) : _buff(matrix._buff.size
      ↪ ())
    {
```

```cpp
        for (size_t i = 0; i < matrix.nrow(); i++)
        {
            for (size_t j = 0; j < matrix.ncol(); j++)
            {
                _buff[i] = matrix(i, j);
            }
        }
    };

    T &operator()(size_t i, size_t j) override
    {
        if (i >= this->_nrow || j >= this->_ncol)
            throw std::invalid_argument("Position is out of range");
        return _buff[i * this->_ncol + j];
    }

    const T& operator()(size_t i, size_t j) const override
    {
        if (i >= this->_nrow || j >= this->_ncol)
            throw std::invalid_argument("Position is out of range");
        return _buff[i * this->_ncol + j];
    }

    Matrix<T> transpose()
    {
        Matrix<T> result(this->_nrow, this->_ncol);

        for (size_t i = 0; i < this->_nrow; i++)
        {
            for (size_t j = 0; j < this->_ncol; j++)
            {
                result(i, j) = operator()(j, i);
            }
        }

        return result;
    }

    bool isSymmetric() const noexcept
    {
        if (this->_ncol != this->_nrow)
        return false;

        for (size_t i = 0; i < this->_nrow; i++)
        {
            for (size_t j = 0; j < i; j++)
            {
                if (operator()(i, j) != operator()(j, i))
```

```
                    return false;
                }
            }

            return true;
        }

private:
    Buffer<T> _buff;
};

template <typename T, typename U>
auto operator*(const IMatrix<T> &rhs, const IMatrix<U> &lhs)
{
    if (rhs.ncol() != lhs.nrow())
        throw std::invalid_argument("Number␣of␣column␣!=␣
          ↪ number␣of␣row");

    Matrix<typename std::common_type<T, U>::type> result(rhs.
       ↪ nrow(), lhs.ncol());

    for (size_t i = 0; i < rhs.nrow(); i++)
    {
        for (size_t j = 0; j < lhs.ncol(); j++)
        {
            for (size_t k = 0; k < rhs.ncol(); k++)
            {
                result(i, j) += rhs(i, k) * lhs(k, j);
            }
        }
    }

    return result;
}

template <typename T, typename U>
auto operator*(const IMatrix<T> &matrix, const Vector<U> &
   ↪ vector)
{
    if (matrix.ncol() != vector.nval())
        throw std::invalid_argument("Number␣of␣column␣!=␣size
          ↪ ␣of␣vector");

    Vector<typename std::common_type<T, U>::type> result(
       ↪ matrix.nrow());

    for (size_t i = 0; i < matrix.nrow(); i++)
    {
        for (size_t j = 0; j < matrix.ncol(); j++)
        {
```

```
                result(i) += vector(j) * matrix(i, j);
            }
        }

        return result;
    }
}
```

Класи для оптимізації роботи з матрицями та векторами

```
#pragma once

#include <unordered_map>

#include "Vector.h"
#include "Matrix.h"

namespace LES
{
    template<typename T>
    class IMatrixView : public IMatrix<T>
    {
    public:
        void swapRows(size_t i, size_t j)
        {
            auto it = _rowMap.find(j);
            if (it != _rowMap.end())
            {
                _rowMap.insert({j, it->second});
            }
            else
            {
                _rowMap.insert({j, i});
            }
            _rowMap.insert({i, j});
        }

        size_t getRow(size_t i) const noexcept
        {
            auto it = _rowMap.find(i);
            if (it != _rowMap.end())
            {
                i = it->second;
            }
            return i;
        };

    protected:
        std::unordered_map<size_t, size_t> _rowMap;
    };
```

```
template<typename T>
class MatrixView : public IMatrixView<T>
{
public:
    MatrixView(IMatrix<T> &matrix)
    {
        _matrix = &matrix;
        this->_nrow = matrix.nrow();
        this->_ncol = matrix.ncol();
    }

    const T& operator()(size_t i, size_t j) const override
    {
        return _matrix->operator()(this->getRow(i), j);
    }

    T& operator()(size_t i, size_t j) override
    {
        return _matrix->operator()(this->getRow(i), j);
    }

private:
    IMatrix<T>* _matrix;
};

template<typename T>
class MatrixMatrixView : public IMatrixView<T>
{
public:
    MatrixMatrixView(IMatrix<T> &matrix1, IMatrix<T> &matrix2
        ↪ )
    {
        _matrix1 = &matrix1;
        _matrix2 = &matrix2;
        this->_nrow = matrix1.nrow();
        this->_ncol = matrix1.ncol() + matrix2.ncol();
    }

    const T& operator()(size_t i, size_t j) const override
    {
        if (j >= this->_ncol)
            throw std::invalid_argument("Position␣is␣out␣of␣
                ↪ range");

        if (j >= _matrix1->ncol())
        {
            return _matrix2->operator()(this->getRow(i), j -
                ↪ _matrix1->ncol());
        }
```

```
        return _matrix1->operator()(i, j);
    }

    T& operator()(size_t i, size_t j) override
    {
        if (j >= this->_ncol)
            throw std::invalid_argument("Position is out of
              ↪ range");

        if (j >= _matrix1->ncol())
        {
            return _matrix2->operator()(this->getRow(i), j -
              ↪ _matrix1->ncol());
        }

        return _matrix1->operator()(i, j);
    }

private:
    IMatrix<T>* _matrix1;
    IMatrix<T>* _matrix2;
};

template<typename T>
class MatrixVectorView : public IMatrixView<T>
{
public:
    MatrixVectorView(IMatrix<T> &matrix, Vector<T> &vector)
    {
        _matrix = &matrix;
        _vector = &vector;
        this->_nrow = matrix.nrow();
        this->_ncol = matrix.ncol() + 1;
    }

    const T& operator()(size_t i, size_t j) const override
    {
        if (j >= this->_ncol)
            throw std::invalid_argument("Position is out of
              ↪ range");

        i = this->getRow(i);
        if (j >= _matrix->ncol())
        {
            return _vector->operator()(i);
        }

        return _matrix->operator()(i, j);
    }
```

```
        T& operator()(size_t i, size_t j) override
        {
            if (j >= this->_ncol)
                throw std::invalid_argument("Position␣is␣out␣of␣
                    ↪ range");

            i = this->getRow(i);
            if (j >= _matrix->ncol())
            {
                return _vector->operator()(i);
            }

            return _matrix->operator()(i, j);
        }

    private:
        IMatrix<T>* _matrix;
        Vector<T>* _vector;
    };
}
```

Утіліти для представлення матриць та векторів у консолі

```
#pragma once

#include <cmath>

#include "Matrix.h"
#include "Vector.h"

namespace LES
{
#ifndef LATEX
    template <typename T>
    void print(const Matrix<T> &matrix)
    {
        for (size_t i = 0; i < matrix.nrow(); i++)
        {
            for (size_t j = 0; j < matrix.ncol(); j++)
            {
                std::cout << matrix(i, j) << "\t";
            }
            std::cout << std::endl;
        }
    }

    template <typename T>
    void print(const Vector<T> &vector)
    {
        for (size_t i = 0; i < vector.nval(); i++)
        {
```

```cpp
            std::cout << vector(i) << "\t";
        }
        std::cout << std::endl;
    }

    template <typename T>
    void printAbs(const Vector<T> &vector)
    {
        for (size_t i = 0; i < vector.nval(); i++)
        {
            std::cout << std::abs(vector(i)) << "\t";
        }
        std::cout << std::endl;
    }
#else
    template <typename T>
    void print(const Matrix<T> &matrix)
    {
        std::cout << "\\begin{pmatrix}" << std::endl;
        for (size_t i = 0; i < matrix.nrow(); i++)
        {
            std::cout << "\t";
            for (size_t j = 0; j < matrix.ncol() - 1; j++)
            {
                std::cout << matrix(i, j) << " & ";
            }
            std::cout << matrix(i, matrix.ncol() - 1);
            if (i != matrix.nrow() - 1)
                std::cout << " \\\\";
            std::cout << std::endl;
        }
        std::cout << "\\end{pmatrix}" << std::endl;
    }

    template <typename T>
    void print(const Vector<T> &vector)
    {
        std::cout << "\\begin{pmatrix}" << std::endl << "\t";;
        for (size_t i = 0; i < vector.nval() - 1; i++)
        {
            std::cout << vector(i) << " \\\\ ";
        }
        std::cout << vector(vector.nval() - 1) << std::endl << "\\end{pmatrix}";
    }

    template <typename T>
    void printAbs(const Vector<T> &vector)
    {
        std::cout << "\\begin{pmatrix}" << std::endl << "\t";;
```

```
        for (size_t i = 0; i < vector.nval() - 1; i++)
        {
            std::cout << std::abs(vector(i)) << "␣\\\\␣";
        }
        std::cout << vector(vector.nval() - 1) << std::endl << "
           ↪ \\end{pmatrix}";
    }
#endif
}
```

Методи Гаусса(додатково)

```
#pragma once

namespace LES
{
    template <typename T>
    void gaussJordanEliminationStep(IMatrix<T>& matrix, size_t i,
       ↪  size_t j)
    {
        for (size_t k = 0; k < matrix.nrow(); k++)
        {
            auto pivot = matrix(i, j);
            auto mult = matrix(k, j) / pivot;
            for (size_t l = 0; l < matrix.ncol(); l++)
            {
                if (i == k)
                    matrix(k, l) = matrix(k, l) / pivot;
                else if (j == l)
                    matrix(k, l) = 0;
                else
                    matrix(k, l) = matrix(k, l) - mult * matrix(i
                       ↪ , l);
            }
        }
    }

    template <typename T>
    void gaussEliminationStep(IMatrix<T>& matrix, size_t i,
       ↪ size_t j)
    {
        for (size_t k = i + 1; k < matrix.nrow(); k++)
        {
            auto pivot = matrix(i, j);
            auto mult = matrix(k, j) / pivot;
            for (size_t l = j; l < matrix.ncol(); l++)
            {
                if (j == l)
                    matrix(k, l) = 0;
                else
                    matrix(k, l) = matrix(k, l) - mult * matrix(i
```

```
                    ↪ , 1);
            }
        }
    }
}
```

Декомпозиція Схолецького та LDL

```
#pragma once

#include <cmath>
#include <tuple>
#include "Matrix.h"

#ifdef PRINT
    #include "Utils.h"
#endif

namespace LES
{
    template <typename T>
    auto cholesky(const Matrix<T> &matrix)
    {
        if(!matrix.isSymmetric())
            throw std::invalid_argument("Matrix does not
              ↪ symmetric");

        using ret_type = decltype(std::sqrt(std::declval<T>()));
        Matrix<ret_type> result(matrix.nrow(), matrix.ncol());

        for (size_t i = 0; i < matrix.nrow(); i++)
        {
            for (size_t j = 0; j <= i; j++)
            {
                ret_type sum = 0;
                for(size_t k = 0; k < j; k++)
                {
                    sum += result(i, k) * result(j, k);
                }

                if (j == i)
                    result(i,i) = std::sqrt(matrix(i,i) - sum);
                else
                    result(i,j) = (matrix(i,j) - sum) / result(j,
                       ↪ j);
            }

#ifdef PRINT
            std::cout << "Cholesky decomposition. Step: " << i+1
              ↪ << std::endl;
            print(result);
```

```
            std::cout << std::endl;
#endif
        }

        return result;
    }

    template <typename T>
    auto ldl(const Matrix<T> &matrix)
    {
        if(!matrix.isSymmetric())
            throw std::invalid_argument("Matrix does not 
                ↪ symmetric");

        using ret_type = decltype(std::sqrt(std::declval<T>()));
            ↪ //??? problem: int / int -> int
        Matrix<ret_type> l(matrix.nrow(), matrix.ncol());
        Vector<ret_type> d(matrix.nrow());

        for (size_t i = 0; i < matrix.nrow(); i++)
        {
            for (size_t j = 0; j <= i; j++)
            {
                ret_type sum = 0;
                for(size_t k = 0; k < j; k++)
                {
                    sum += l(i, k) * l(j, k) * d(k);
                }

                if (j == i)
                {
                    l(i, i) = 1;
                    d(i) = matrix(i,i) - sum;
                }
                else
                {
                    l(i,j) = (matrix(i,j) - sum) / d(j);
                }
            }

#ifdef PRINT
            std::cout << "LDL decomposition. Step: " << i+1 <<
                ↪ std::endl;
            std::cout << "L:" << std::endl;
            print(l);
            std::cout << std::endl;
            std::cout << "D:" << std::endl;
            print(d);
            std::cout << std::endl;
#endif
```

```
        }

        return std::make_tuple(l, d);
    }
}
```

Процедура вирішення рівнянь за допомогою замін

```
#pragma once

#include "Matrix.h"
#include "Vector.h"

namespace LES
{
    template <typename T, typename U>
    auto forward(const IMatrix<T> &A, const Vector<U> &b)
    {
        if (A.nrow() != b.nval() || A.ncol() != b.nval())
            throw std::invalid_argument("Invalid␣sizes");

        Vector<typename std::common_type<T, U>::type> x(b.nval())
          ↪ ;

        for (size_t i = 0; i < A.nrow(); i++)
        {
            T sum = 0;
            for (size_t j = 0; j < i; j++)
            {
                sum += A(i, j) * x(j);
            }
            x(i) = (b(i) - sum) / A(i, i);
        }

        return x;
    }

    template <typename T, typename U>
    auto backward(const IMatrix<T> &A, const Vector<U> &b)
    {
        // if (A.nrow() != b.nval() || A.ncol() != b.nval())
        //     throw std::invalid_argument("Invalid sizes");

        Vector<typename std::common_type<T, U>::type> x(b.nval())
          ↪ ;

        size_t i = A.nrow();
        do
        {
            i--;
            T sum = 0;
```

```
            for (size_t j = i + 1; j < A.nrow(); j++)
            {
                sum += A(i, j) * x(j);
            }
            x(i) = (b(i) - sum) / A(i, i);
        } while (i != 0);

        return x;
    }
}
```

Процедура обрахунку коренів лінійної системи рівнянь

```
#pragma once
#include <vector>

#include "Matrix.h"
#include "Vector.h"
#include "MatrixView.h"
#include "Cholesky.h"
#include "Gauss.h"
#include "Substitution.h"

#ifdef PRINT
#include "Utils.h"
#endif

namespace LES
{
        template <typename T, typename U>
        auto choleskySolve(const Matrix<T> &A, const Vector<U> &b
           ↪ )
        {
                auto chol = cholesky(A);
                auto y = forward(chol, b);
                auto x = backward(TransposeView(chol), y);

#ifdef PRINT
                std::cout << "Cholesky␣solver␣y" << std::endl;
                print(y);
                std::cout << std::endl;
#endif

                return x;
        }

        template <typename T, typename U>
        auto ldlSolve(const Matrix<T> &A, const Vector<U> &b)
        {
                auto [l, d] = ldl(A);
                auto z = forward(l, b);
```

```
#ifdef PRINT
                std::cout << "LDL␣solver␣z" << std::endl;
                print(z);
                std::cout << std::endl;
#endif

                for (size_t i = 0; i < z.nval(); i++)
                {
                        z(i) = z(i) / d(i);
                }

#ifdef PRINT
                std::cout << "LDL␣solver␣y" << std::endl;
                print(z);
                std::cout << std::endl;
#endif

                auto x = backward(TransposeView(l), z);
                return x;
        }

        template <typename T, typename U>
        auto gaussJordanSolve(const Matrix<T> &A, const Vector<U>
            ↪  &b)
        {
                using ret_type = typename std::common_type<T, U
                    ↪ >::type;
                Matrix<ret_type> m(A);
                Vector<ret_type> v(b);
                MatrixVectorView<ret_type> aug(m, v);

                for (size_t i = 0; i < m.nrow(); i++)
                {
                        LES::gaussJordanEliminationStep(aug, i, i
                            ↪ );

#ifdef PRINT
                        std::cout << "Gauss␣Jordan␣solver.␣Step:␣
                            ↪ " << i + 1 << std::endl;
                        std::cout << "Matrix␣A:␣" << std::endl;
                        print(m);
                        std::cout << "Vector␣b:␣" << std::endl;
                        print(v);
#endif
                }

                return v;
        }
```

```
	template <typename T, typename U>
	auto gaussSolve(const Matrix<T> &A, const Vector<U> &b)
	{
		using ret_type = typename std::common_type<T, U>::type;
		Matrix<ret_type> m(A);
		Vector<ret_type> v(b);
		MatrixVectorView<ret_type> aug(m, v);

		for (size_t i = 0; i < m.nrow(); i++)
		{
			LES::gaussEliminationStep(aug, i, i);

#ifdef PRINT
			std::cout << "Gauss solver. Step: " << i + 1 << std::endl;
			std::cout << "Matrix A: " << std::endl;
			print(m);
			std::cout << "Vector b: " << std::endl;
			print(v);
#endif
		}

		auto x = backward(m, v);

		return x;
	}

	template <typename T, typename U>
	auto gaussPivotSolve(const Matrix<T> &A, const Vector<U> &b)
	{
		using ret_type = typename std::common_type<T, U>::type;
		Matrix<ret_type> m(A);
		Vector<ret_type> v(b);
		MatrixVectorView<ret_type> aug(m, v);

		for (size_t i = 0; i < m.nrow(); i++)
		{
			auto max = i;
			for (size_t l = i + 1; l < m.nrow(); l++)
			{
				if (std::abs(aug(l, i)) > std::abs(aug(max, i)))
					max = l;
			}
			if (max != i)
			{
				aug.swapRows(i, max);
```

```
                        }

                        LES::gaussEliminationStep(aug, i, i);

#ifdef PRINT
                        std::cout << "Gauss␣with␣pivoting␣solver.
                            ↪ ␣Step:␣" << i + 1 << std::endl;
                        std::cout << "Matrix␣A:␣" << std::endl;
                        print(m);
                        std::cout << "Vector␣b:␣" << std::endl;
                        print(v);
#endif
                }

                size_t i = A.nrow();
                Vector<ret_type> res(b.nval());
                do
                {
                    i--;
                    T sum = 0;
                    for (size_t j = i + 1; j < A.nrow(); j++)
                    {
                        sum += aug(i, j) * res(j);
                    }
                    res(i) = (aug(i, A.ncol()) - sum) / aug(i, i)
                        ↪ ;
                } while (i != 0);

                return res;
        }
}
```

Розв'язання рівняння з практичної частини

```
#include <iostream>
#include <iomanip>

// #define PRINT

#include "Matrix.h"
#include "Vector.h"
#include "Solvers.h"
#include "Utils.h"

int main()
{
    LES::Matrix<double> mat(4, 4);
    mat(0, 0) = mat(0, 3) = mat(3, 0) = 5.5;
    mat(0, 1) = mat(1, 0) = mat(3, 1) = mat(1, 3) = 7.0;
    mat(1, 1) = mat(2, 2) = mat(3,3) = 10.5;
    mat(1, 2) = mat(2, 1) = 8.0;
```

```
mat(2, 0) = mat(0,2) = 6.0;
mat(3, 2) = mat(2,3) = 9.0;

std::cout << "Matrix␣A" << std::endl;
print(mat);
std::cout << std::endl;

LES::Vector<int> vec(4);
vec(0) = 23;
vec(1) = 32;
vec(2) = 33;
vec(3) = 31;

std::cout << "Vector␣b" << std::endl;
print(vec);
std::cout << std::endl;

std::cout << std::setprecision(16); //specify precision

auto xch = LES::choleskySolve(mat, vec);
std::cout << "Roots␣cholesky" << std::endl;
print(xch);

auto res = mat * xch - vec;
std::cout << "Cholesky␣res" << std::endl;
printAbs(res);
std::cout << std::endl;

auto xld = LES::ldlSolve(mat, vec);
std::cout << "Roots␣LDL" << std::endl;
print(xld);

res = mat * xld - vec;
std::cout << "LDL␣res" << std::endl;
printAbs(res);
std::cout << std::endl;

auto xgj = LES::gaussJordanSolve(mat, vec);
std::cout << "Gauss-Jordan" << std::endl;
print(xgj);

res = mat * xgj - vec;
std::cout << "Gauss␣Jordan␣res" << std::endl;
printAbs(res);
std::cout << std::endl;

auto xg = LES::gaussSolve(mat, vec);
std::cout << "Gauss" << std::endl;
print(xg);
```

```
    res = mat * xg - vec;
    std::cout << "Gauss␣res" << std::endl;
    printAbs(res);
    std::cout << std::endl;

    auto xgp = LES::gaussPivotSolve(mat, vec);
    std::cout << "Gauss␣with␣pivot" << std::endl;
    print(xgp);

    res = mat * xgp - vec;
    std::cout << "Gauss␣with␣pivot␣res" << std::endl;
    printAbs(res);

    return 0;
}
```