|  | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |
| --- | --- |

ФАКУЛЬТЕТ Информатики и систем управления

КАФЕДРА Теоретической информатики и компьютерных технологий

**ЛАБОРАТОРНАЯ РАБОТА № 1**

**ПО КУРСУ:**

***«Численные методы»***

Студент *Виленский С.Д.*

Преподаватель *Домрачева А.Б.*

*Москва, 2024 г.*

**ОГЛАВЛЕНИЕ**

[1. Постановка задачи 3](#_heading=h.30j0zll)

[2. Практическая реализация 4](#_heading=h.1fob9te)

# 1. Постановка задачи

Реализовать алгоритм решения СЛАУ с 3-диагональной матрицей коэффициентов методом прогонки.

# 2. Практическая реализация

lib/matrix.cpp

#pragma once

#include <vector>

#include <random>

#include <cassert>

#include <iostream>

template< typename T, std::size\_t HEIGHT, std::size\_t WIDTH >

requires (

std::is\_arithmetic\_v<T> &&

HEIGHT != 0 &&

WIDTH != 0

)

class Matrix {

public:

Matrix() : m\_matrix(HEIGHT, std::vector<T>(WIDTH)) {}

Matrix(const std::vector<std::vector<T>>& matrix)

: m\_matrix(matrix) {

assert(matrix.size() == HEIGHT);

for (const auto& row : matrix) {

assert(row.size() == WIDTH);

}

}

virtual ~Matrix() {};

auto fillRandomValues(const T& minValue, const T& maxValue, const T& stepValue) -> void {

std::size\_t countValues = (maxValue - minValue) / stepValue + 1;

#define getRandValue() static\_cast<T>((rand() % countValues) \* stepValue + minValue);

for (auto& row : m\_matrix) {

for (auto& element : row) {

element = getRandValue();

}

}

}

virtual auto at(std::size\_t i, std::size\_t j) -> T& {

assert(i < HEIGHT && j < WIDTH);

return m\_matrix[i][j];

}

virtual auto at(std::size\_t i, std::size\_t j) const -> const T& {

assert(i < HEIGHT && j < WIDTH);

return m\_matrix[i][j];

}

auto operator\*(const T& coef) const -> Matrix<T, HEIGHT, WIDTH> {

Matrix<T, HEIGHT, WIDTH> resMatrix{};

for (std::size\_t row = 0; row != HEIGHT; ++row) {

for (std::size\_t col = 0; col != WIDTH; ++col) {

resMatrix.at(row, col) = this->at(row, col) \* coef;

}

}

return resMatrix;

}

auto operator+(const Matrix<T, HEIGHT, WIDTH>& rightMatrix) const -> Matrix<T, HEIGHT, WIDTH> {

Matrix<T, HEIGHT, WIDTH> resMatrix{};

for (std::size\_t row = 0; row != HEIGHT; ++row) {

for (std::size\_t col = 0; col != WIDTH; ++col) {

resMatrix.at(row, col) = this->at(row, col) + rightMatrix.at(row, col);

}

}

return resMatrix;

}

auto operator-(const Matrix<T, HEIGHT, WIDTH>& rightMatrix) const -> Matrix<T, HEIGHT, WIDTH> {

Matrix<T, HEIGHT, WIDTH> resMatrix{};

for (std::size\_t row = 0; row != HEIGHT; ++row) {

for (std::size\_t col = 0; col != WIDTH; ++col) {

resMatrix.at(row, col) = this->at(row, col) - rightMatrix.at(row, col);

}

}

return resMatrix;

}

template< std::size\_t WIDTH\_OTHER >

auto operator\*(const Matrix<T, WIDTH, WIDTH\_OTHER>& other) const -> Matrix<T, HEIGHT, WIDTH\_OTHER> {

Matrix<T, HEIGHT, WIDTH\_OTHER> matrixProduct{};

for (std::size\_t i = 0; i != HEIGHT; ++i) {

for (std::size\_t j = 0; j != WIDTH; ++j) {

for (std::size\_t k = 0; k != WIDTH\_OTHER; ++k) {

matrixProduct.at(i, k) += this->at(i, j) \* other.at(j, k);

}

}

}

return matrixProduct;

}

auto averageValue() const -> T {

T summ{};

for (std::size\_t row = 0; row != HEIGHT; ++row) {

for (std::size\_t col = 0; col != WIDTH; ++col) {

summ += this->at(row, col);

}

}

return summ / (HEIGHT \* WIDTH);

}

protected:

std::vector<std::vector<T>> m\_matrix;

};

template< typename T, std::size\_t HEIGHT, std::size\_t WIDTH >

std::ostream& operator<<(std::ostream& os, const Matrix<T, HEIGHT, WIDTH>& matrix) {

for (std::size\_t row = 0; row != HEIGHT; ++row) {

for (std::size\_t col = 0; col != WIDTH; ++col) {

os << matrix.at(row, col);

if (col + 1 != WIDTH) {

os << '\t';

}

}

if (row + 1 != HEIGHT) {

os << '\n';

}

}

return os;

}

lib/vector.cpp

#pragma once

#include "Matrix.cpp"

template< typename T, std::size\_t SIZE >

class Vector : public Matrix<T, SIZE, 1> {

public:

Vector() {}

Vector(const Matrix<T, SIZE, 1>& other) {

for (std::size\_t row = 0; row != SIZE; ++row) {

this->at(row) = other.at(row, 0);

}

}

Vector(const std::vector<T>& vector) {

assert(vector.size() == SIZE);

for (std::size\_t row = 0; row != SIZE; ++row) {

this->at(row) = vector[row];

}

}

auto operator\*(const T& coef) const -> Vector<T, SIZE> {

Vector<T, SIZE> resMatrix{};

for (std::size\_t row = 0; row != SIZE; ++row) {

resMatrix.at(row) = this->at(row) \* coef;

}

return resMatrix;

}

auto at(std::size\_t i) -> T& {

assert(i < SIZE);

return Matrix<T, SIZE, 1>::at(i, 0);

}

auto at(std::size\_t i) const -> const T& {

assert(i < SIZE);

return Matrix<T, SIZE, 1>::at(i, 0);

}

template< std::size\_t WIDTH\_OTHER >

auto operator\*(const Matrix<T, 1, WIDTH\_OTHER>& other) const -> Matrix<T, SIZE, WIDTH\_OTHER> {

Matrix<T, SIZE, WIDTH\_OTHER> matrixProduct{};

for (std::size\_t i = 0; i != SIZE; ++i) {

for (std::size\_t j = 0; j != WIDTH\_OTHER; ++j) {

matrixProduct.at(i, j) = this->at(i) \* other.at(1, j);

}

}

return matrixProduct;

}

};

lib/SquareMatrix.cpp

#pragma once

#include "Matrix.cpp"

template< typename T, std::size\_t SIZE >

requires std::is\_floating\_point\_v<T>

class SquareMatrix : public Matrix<T, SIZE, SIZE> {

public:

SquareMatrix() {}

SquareMatrix(const Matrix<T, SIZE, SIZE>& other) {

for (std::size\_t row = 0; row != SIZE; ++row) {

for (std::size\_t col = 0; col != SIZE; ++col) {

this->at(row, col) = other.at(row, col);

}

}

}

SquareMatrix(const std::vector<std::vector<T>>& matrix)

: Matrix<T, SIZE, SIZE>::m\_matrix(matrix) {

assert(matrix.size() == SIZE);

for (const auto& row : matrix) {

assert(row.size() == SIZE);

}

}

auto operator\*(const Matrix<T, SIZE, SIZE>& other) const -> SquareMatrix<T, SIZE> {

SquareMatrix<T, SIZE> matrixProduct{};

for (std::size\_t i = 0; i != SIZE; ++i) {

for (std::size\_t j = 0; j != SIZE; ++j) {

for (std::size\_t k = 0; k != SIZE; ++k) {

matrixProduct.at(i, k) += this->at(i, j) \* other.at(j, k);

}

}

}

return matrixProduct;

}

template< std::size\_t WIDTH\_OTHER >

auto operator\*(const Matrix<T, SIZE, WIDTH\_OTHER>& other) const -> Matrix<T, SIZE, WIDTH\_OTHER> {

Matrix<T, SIZE, WIDTH\_OTHER> matrixProduct{};

for (std::size\_t i = 0; i != SIZE; ++i) {

for (std::size\_t j = 0; j != SIZE; ++j) {

for (std::size\_t k = 0; k != WIDTH\_OTHER; ++k) {

matrixProduct.at(i, k) += this->at(i, j) \* other.at(j, k);

}

}

}

return matrixProduct;

}

auto getTransposed() const -> SquareMatrix<T, SIZE> {

SquareMatrix<T, SIZE> transposed{};

for (std::size\_t row = 0; row != SIZE; ++row) {

for (std::size\_t col = 0; col != SIZE; ++col) {

transposed.at(row, col) = this->at(col, row);

}

}

return transposed;

}

auto getMinor(std::size\_t minorRow, std::size\_t minorCol) const -> SquareMatrix<T, SIZE - 1> {

assert(minorRow < SIZE && minorCol < SIZE);

SquareMatrix<T, SIZE - 1> minor{};

for (std::size\_t row = 0; row != SIZE - 1; ++row) {

for (std::size\_t col = 0; col != SIZE - 1; ++col) {

minor.at(row, col) = this->at(row + (row >= minorRow), col + (col >= minorCol));

}

}

return minor;

}

template<

std::size\_t S = SIZE,

class = typename std::enable\_if<S == 1>::type

>

auto getDeterminant() const -> T {

return this->at(0, 0);

}

template<

int S = SIZE,

class = typename std::enable\_if<S != 1>::type

>

auto getDeterminant() const -> T {

T determinant{};

for (std::size\_t col = 0; col != SIZE; ++col) {

determinant +=

(col % 2 == 0 ? 1 : -1)

\* this->at(0, col)

\* this->getMinor(0, col).getDeterminant();

}

return determinant;

}

auto getInverse() const -> SquareMatrix<T, SIZE> {

T determinant{this->getDeterminant()};

assert(determinant != 0);

SquareMatrix<T, SIZE> inverseMatrix{};

for (std::size\_t row = 0; row != SIZE; ++row) {

for (std::size\_t col = 0; col != SIZE; ++col) {

inverseMatrix.at(row, col) =

((row + col) % 2 == 0 ? 1 : -1)

\* this->getMinor(col, row).getDeterminant()

/ determinant;

}

}

return inverseMatrix;

}

};

lib/DiagonalMatrix.cpp

#pragma once

#include <vector>

#include <cassert>

#include "SquareMatrix.cpp"

template< typename T, std::size\_t SIZE, std::size\_t DIAGS >

requires (

(DIAGS & 1) == 1 &&

SIZE > (DIAGS >> 1)

)

class DiagonalMatrix : public SquareMatrix<T, SIZE> {

public:

DiagonalMatrix() {

this->m\_matrix = std::vector<std::vector<T>>(SIZE, std::vector<T>(DIAGS));

}

DiagonalMatrix(const std::vector<std::vector<T>>& matrix) {

assert(matrix.size() == SIZE);

for (std::size\_t row = 0; row != SIZE; ++row) {

assert(matrix[row].size() == SIZE);

for (std::size\_t col = 0; col != SIZE; ++col) {

this->at(row, col) = matrix[row][col];

}

}

}

auto operator\*(const T& coef) const -> DiagonalMatrix<T, SIZE, DIAGS> {

DiagonalMatrix<T, SIZE, DIAGS> resMatrix{};

for (std::size\_t row = 0; row != SIZE; ++row) {

for (std::size\_t col = 0; col != SIZE; ++col) {

resMatrix.at(row, col) = this->at(row, col) \* coef;

}

}

return resMatrix;

}

auto at(std::size\_t i, std::size\_t j) -> T& override {

assert(i < SIZE && j < SIZE);

if (std::max(i, j) - std::min(i, j) > (DIAGS >> 1)) {

return m\_zeroValue;

}

return this->m\_matrix[std::min(i, j)][j - i + (DIAGS >> 1)];

}

auto at(std::size\_t i, std::size\_t j) const -> const T& override {

assert(i < SIZE && j < SIZE);

if (std::max(i, j) - std::min(i, j) > (DIAGS >> 1)) {

return m\_zeroValue;

}

return this->m\_matrix[std::min(i, j)][j - i + (DIAGS >> 1)];

}

template< std::size\_t WIDTH\_OTHER >

auto operator\*(const Matrix<T, SIZE, WIDTH\_OTHER>& other) const -> Matrix<T, SIZE, WIDTH\_OTHER> {

Matrix<T, SIZE, WIDTH\_OTHER> matrixProduct{};

for (std::size\_t i = 0; i != SIZE; ++i) {

for (std::size\_t j = 0; j != SIZE; ++j) {

for (std::size\_t k = 0; k != WIDTH\_OTHER; ++k) {

matrixProduct.at(i, k) += this->at(i, j) \* other.at(j, k);

}

}

}

return matrixProduct;

}

private:

T m\_zeroValue{};

};

template< typename T, std::size\_t SIZE >

using ThreeDiagonalMatrix = DiagonalMatrix<T, SIZE, 3>;

lab1.cpp

#include <iostream>

#include "lib/DiagonalMatrix.cpp"

#include "lib/Vector.cpp"

template< int SIZE >

auto isSolveExists(

const ThreeDiagonalMatrix<float, SIZE>& A

) -> bool {

bool wasNotStrict = false;

if (

abs(A.at(0, 0)) < .001 ||

abs(A.at(SIZE - 1, SIZE - 1)) < .001

) {

return false;

}

for (std::size\_t i = 1; i != SIZE - 1; ++i) {

float rightPart = abs(A.at(i, i - 1)) + abs(A.at(i, i + 1));

if (

abs(A.at(i, i)) < .001 ||

A.at(i, i) < rightPart

) {

return false;

}

if (abs(A.at(i, i) - rightPart) < .001) {

wasNotStrict = true;

}

}

return wasNotStrict;

}

template< int SIZE >

auto findSolve(

const ThreeDiagonalMatrix<float, SIZE>& A,

const Vector<float, SIZE>& d

) -> Vector<float, SIZE> {

std::vector<float> alpha(SIZE);

std::vector<float> beta(SIZE);

alpha[0] = - A.at(0, 1) / A.at(0, 0);

beta[0] = d.at(0) / A.at(0, 0);

for (std::size\_t i = 1; i != SIZE - 1; ++i) {

alpha[i] = A.at(i, i + 1)

/ (- A.at(i, i) - A.at(i, i - 1) \* alpha[i - 1]);

beta[i] = (A.at(i, i - 1) \* beta[i - 1] - d.at(i))

/ (- A.at(i, i) - A.at(i, i - 1) \* alpha[i - 1]);

}

alpha[SIZE - 1] = 0;

beta[SIZE - 1] = (A.at(SIZE - 1, SIZE - 2) \* beta[SIZE - 2] - d.at(SIZE - 1))

/ (- A.at(SIZE - 1, SIZE - 1) - A.at(SIZE - 1, SIZE - 2) \* alpha[SIZE - 2]);

Vector<float, SIZE> x{};

x.at(SIZE - 1) = beta[SIZE - 1];

for (std::size\_t i = SIZE - 2; i != -1; --i) {

x.at(i) = alpha[i] \* x.at(i + 1) + beta[i];

}

return x;

}

template< std::size\_t SIZE >

auto checkSolve(

const ThreeDiagonalMatrix<float, SIZE>& A,

const Vector<float, SIZE>& xStar,

const Vector<float, SIZE>& d

) -> Vector<float, SIZE> {

Vector<float, SIZE> dStar = A \* xStar;

Vector<float, SIZE> r = d - dStar;

Vector<float, SIZE> e = A.getInverse() \* r;

return e;

}

template< std::size\_t SIZE >

auto test\_alghorithm() -> void {

ThreeDiagonalMatrix<float, SIZE> A{};

A.fillRandomValues(-10, 10, .1);

if (!isSolveExists<SIZE>(A)) {

test\_alghorithm<SIZE>();

return;

}

Vector<float, SIZE> d{};

d.fillRandomValues(-10, 10, .1);

Vector<float, SIZE> x = findSolve<SIZE>(A, d);

Vector<float, SIZE> error = checkSolve(A, x, d);

std::cout << "Error for this test = " << error.averageValue() << '\n';

}

template< std::size\_t SIZE >

auto test\_check(

const ThreeDiagonalMatrix<float, SIZE>& A,

const Vector<float, SIZE>& x

) -> void {

// *if (!isSolveExists<SIZE>(A)) {*

// *std::cout << "Solve unexists\n\n";*

// *return;*

// *}*

Vector<float, SIZE> xStar = findSolve<SIZE>(A, A \* x);

std::cout << xStar << "\n\n";

}

auto main() -> int {

test\_check<4>(

std::vector<std::vector<float>>({

{4, 1, 0, 0},

{1, 4, 1, 0},

{0, 1, 4, 1},

{0, 0, 1, 4}

}),

std::vector<float>{1, 1, 1, 1}

);

test\_check<4>(

ThreeDiagonalMatrix<float, 4>({

{4, 1, 0, 0},

{1, 4, 1, 0},

{0, 1, 4, 1},

{0, 0, 1, 4}

}) \* (1. / 3),

Vector<float, 4>(

{1./3, 1./3, 1, 2}

)

);

return 0;

}