Практическое занятие №4

Армбристер Никита Владиславович

Группа ПМИ-31

Вариант 1

Цель: сформировать практические навыки применения прямых методов решения систем линейных алгебраических уравнений с квадратными невырожденными матрицами.

Задание: сформировать три СЛАУ с размерами N = {250, 500, 1000} Ax=f. Реализовать программное решение и использованием LU-разложения и QR-разложения на базе отражения Хаусхолдера.

Решение:

Для решения данного практического задания был написан проект на языке C++.

Ссылка на репозиторий с работой:

https://github.com/Gribnik24/NumericalMethods-PracticalTasks/tree/main/Task%20%234.%20SLAE_Solver

Проект состоит из четырех файлов реализации .cpp и одного назывного файла .h:

1. main.cpp – основной файл, который содержит создание матриц, запуск алгоритмов и замеры времени
2. MatrixOperations.cpp – файл содержит вспомогательные функции для работы с матрицами, необходимые для решения
3. LU_Solver.cpp – файл содержит LU-разложение и решение с ним
4. QR_Solver.cpp – файл содержит QR-разложение и решение с ним
5. LinearAlgebra.h – объявление всех используемых функций

Рассмотрим составляющие проекта подробнее:

- **LinearAlgebra.h**
  Назывной файл. Помимо объявления функций также содержит объявление псевдонимов – вектор и матрицу

```cpp
#pragma once
#include <vector>
#include <cmath>
#include <stdexcept>
#include <chrono>

using Matrix = std::vector<std::vector<double>>;
using Vector = std::vector<double>;

// Matrix operations
Matrix createMatrix(int N);
Vector createRightHandSide(const Matrix& A);
double computeError(const Vector& x, const Vector& x_exact);

// LU decomposition
void luDecomposition(Matrix& A, std::vector<int>& pivot);
Vector solveLU(const Matrix& LU, const std::vector<int>& pivot, const Vector& f);

// QR decomposition
void householderQR(const Matrix& A, Matrix& Q, Matrix& R);
Vector solveQR(const Matrix& Q, const Matrix& R, const Vector& f);
```

- **main.cpp**

В файле main.cpp создается вектор размеров матриц sizes и переменная num_measurments – количество измерений времени, по которым в конце цикла каждого из решений будет браться медиана, для определения среднего времени решения.

```
int main() {
    const int num_measurements = 5; // Количество замеров времени
    std::vector<int> sizes = { 250, 500, 1000 };
    std::cout << std::fixed << std::setprecision(6);
    std::cout << "Size  | Method   | Median Time (ms) | Error" << std::endl;
    std::cout << "----------------------------------------" << std::endl;
```

Далее запускается цикл для каждого размера матрицы A. Для каждого из необходимых размеров матриц создается сама матрица A и вектор-результат f. Для метода LU заводится вектор timings – хранилище временных результатов выполнения алгоритма. Выполняется решение, и время кладется в «копилку». Далее по результатам времени находится медиана

```
for (int N : sizes) {
    Matrix A = createMatrix(N);
    Vector f = createRightHandSide(A);
    Vector x_exact(N, 1.0);

    // LU decomposition
    {
        std::vector<long long> timings;
        double final_error = 0.0;

        for (int i = 0; i < num_measurements; ++i) {
            auto start = std::chrono::high_resolution_clock::now();
            Matrix LU = A;
            std::vector<int> pivot;
            luDecomposition(LU, pivot);
            Vector x = solveLU(LU, pivot, f);
            auto stop = std::chrono::high_resolution_clock::now();
            auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(stop - start);
            timings.push_back(duration.count());
            final_error = computeError(x, x_exact);
        }

        long long median_time = calculateMedianTime(timings);
        std::cout << std::setw(5) << N << " | LU       | " << std::setw(15) << median_time
            << " | " << std::scientific << std::setprecision(3) << final_error << std::endl;
    }
```

Аналогично для QR

```
// QR decomposition
{
    std::vector<long long> timings;
    double final_error = 0.0;

    for (int i = 0; i < num_measurements; ++i) {
        auto start = std::chrono::high_resolution_clock::now();
        Matrix Q, R;
        householderQR(A, Q, R);
        Vector x = solveQR(Q, R, f);
        auto stop = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(stop - start);
        timings.push_back(duration.count());
        final_error = computeError(x, x_exact);
    }

    long long median_time = calculateMedianTime(timings);
    std::cout << std::setw(5) << N << " | QR       | " << std::setw(15) << median_time
        << " | " << std::scientific << std::setprecision(3) << final_error << std::endl;
}
std::cout << "-----------------------------------------" << std::endl;
```

Функция расчета медианы сортирует вектор времен и затем возвращает средний элемент в случае нечетного количества замеров или среднее арифметическое двух средних элементов в случае четного.

```
// Median calculation
long long calculateMedianTime(std::vector<long long>& timings) {
    std::sort(timings.begin(), timings.end());

    if (timings.size() % 2 == 1) {
        // for odd num of measurments
        return timings[timings.size() / 2];
    }
    else {
        // for even num of measurments
        return (timings[timings.size() / 2 - 1] + timings[timings.size() / 2]) / 2;
    }
}
```

- **MatrixOperations.cpp**
  Функция **createMatrix** создает матрицу A, по условию варианта 1: если элемент диагональный (i = j), то он равен 100, иначе его значение вычисляется как 1 + i + j

```
Matrix createMatrix(int N) {
    Matrix A(N, Vector(N));
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            if (i == j) {
                A[i][j] = 100.0;
            }
            else {
                A[i][j] = 1.0 + (i + 1) + (j + 1);
            }
        }
    }
    return A;
}
```

Функция createRightHandSide создаёт вектор правой части f для системы линейных уравнений Ax = f, используя заданную матрицу A и предполагая, что точное решение x* — это вектор, состоящий из единиц. Сначала он инициализирует нулевой вектор, а затем заполняет его суммами из матрицы A

```
Vector createRightHandSide(const Matrix& A) {
    int N = A.size();
    Vector f(N, 0.0);
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            f[i] += A[i][j];
        }
    }
    return f;
}
```

Функция computeError вычисляет относительную погрешность численного решения x по сравнению с точным решением x*

```
double computeError(const Vector& x, const Vector& x_exact) {
    double diff_norm = 0.0;
    double exact_norm = 0.0;
    for (size_t i = 0; i < x.size(); ++i) {
        diff_norm += (x[i] - x_exact[i]) * (x[i] - x_exact[i]);
        exact_norm += x_exact[i] * x_exact[i];
    }
    return std::sqrt(diff_norm) / std::sqrt(exact_norm);
}
```

- **LU_Solver.cpp**
  Состоит из двух функций.
  luDecomposition – видоизменяет матрицу A, выполняет разложение в произведение нижней (L) и верхней (U) треугольных матриц.
  Сначала мы инициализируем переменную N – размер матрицы и заполняем объявленный в main вектор pivot – вектор перестановок. Далее выполняем цикл по столбцам матрицы, выбираем ведущий элемент (номер строки) по максимальному значению для каждого столбика. Это необходимо для устойчивости алгоритма, так больше шанса избежать деления на малые числа. Если ведущий элемент отличен от выбранного изначально, то выполняем перестановку строк в матрице A и векторе pivot. Следом идет цикл на вычисление LU. По итогу матрица A представляет комбинированную матрицу L и U. Этого удалось достичь, т.к. некоторые элементы этих матриц известны заранее (диагональ L равна 1, некоторые элементы равны 0)

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ l_{21} & u_{22} & u_{23} \\ l_{31} & l_{32} & u_{33} \end{pmatrix}$$

```cpp
void luDecomposition(Matrix& A, std::vector<int>& pivot) {
    int N = A.size();
    pivot.resize(N);
    for (int i = 0; i < N; ++i) pivot[i] = i;

    for (int k = 0; k < N; ++k) {
        // Partial pivoting
        int max_row = k;
        for (int i = k + 1; i < N; ++i) {
            if (std::abs(A[i][k]) > std::abs(A[max_row][k])) {
                max_row = i;
            }
        }

        if (max_row != k) {
            std::swap(A[k], A[max_row]);
            std::swap(pivot[k], pivot[max_row]);
        }

        // LU decomposition
        for (int i = k + 1; i < N; ++i) {
            A[i][k] /= A[k][k];
            for (int j = k + 1; j < N; ++j) {
                A[i][j] -= A[i][k] * A[k][j];
            }
        }
    }
}
```

Функция solveLU решает систему Ax=f. Инициализируются N – Размер матрицы A и векторы x, b, y. Вектор b – это вектор f с учетом перестановок pivot (ведущий элемент); y – решение Ly=b; x – решение Ux=y и одновременно Ax=f. Функция возвращает x – основное решение

```cpp
Vector solveLU(const Matrix& LU, const std::vector<int>& pivot, const Vector& f) {
    int N = LU.size();
    Vector x(N), b(N), y(N);

    // Apply permutation
    for (int i = 0; i < N; ++i) {
        b[i] = f[pivot[i]];
    }

    // Forward substitution (Ly = b)
    for (int i = 0; i < N; ++i) {
        y[i] = b[i];
        for (int j = 0; j < i; ++j) {
            y[i] -= LU[i][j] * y[j];
        }
    }

    // Backward substitution (Ux = y)
    for (int i = N - 1; i >= 0; --i) {
        x[i] = y[i];
        for (int j = i + 1; j < N; ++j) {
            x[i] -= LU[i][j] * x[j];
        }
        x[i] /= LU[i][i];
    }

    return x;
}
```

- **QR_Solver.cpp**
  QR-разложение – частный случай LU-разложения. Аналогично состоит из двух функций.Функция householderQR принимает матрицу A, Q и R. Пока Q – единичная матрица, R – копия A. Далее в цикле по столбцам вычисляется вектор отражения Хаусхолдера: сначала вычисляется норма столбца; alpha принимает норму со знаком интересующего элемента; строится сам вектор отражения и наконец вычисляется коэффициент beta. По итогу преобразования Хаусхолдера имеет вид $H = I - \beta vv^T$.
  Оно применяется к матрице R. Идет обработка правой части (от столбца k и до конца). Далее обновляется и матрица Q.

```cpp
void householderQR(const Matrix& A, Matrix& Q, Matrix& R) {
    int n = A.size();
    Q = Matrix(n, Vector(n, 0.0));
    for (int i = 0; i < n; ++i) Q[i][i] = 1.0;
    R = A;

    for (int k = 0; k < n - 1; ++k) {
        // reflection vector calculation
        double norm = 0.0;
        for (int i = k; i < n; ++i)
            norm += R[i][k] * R[i][k];
        norm = sqrt(norm);

        if (fabs(norm) < 1e-12) continue;

        double alpha = -copysign(norm, R[k][k]);
        Vector v(n, 0.0);
        for (int i = k; i < n; ++i)
            if (i == k) {
                v[i] = R[i][k] - alpha;
            }
            else {
                v[i] = R[i][k];
            }

        double beta = 0.0;
        for (int i = k; i < n; ++i)
            beta += v[i] * v[i];
        beta = 2.0 / beta;

        // R update
        for (int j = k; j < n; ++j) {
            double dot = 0.0;
            for (int i = k; i < n; ++i)
                dot += v[i] * R[i][j];
            for (int i = k; i < n; ++i)
                R[i][j] -= beta * v[i] * dot;
        }

        // Q update
        for (int j = 0; j < n; ++j) {
            double dot = 0.0;
            for (int i = k; i < n; ++i)
                dot += Q[j][i] * v[i];
            for (int i = k; i < n; ++i)
                Q[j][i] -= beta * v[i] * dot;
        }
    }
}
```

  Функция solveQR выполняет решение. На вход подаются матрицы Q и R, а также вектор-результат f. Инициализируется вектор y – произведение Q^Tf. Далее инициализируется вектор x и из Rx=y находится решение Ax=f. Функция возвращает x.

```cpp
Vector solveQR(const Matrix& Q, const Matrix& R, const Vector& f) {
    int N = Q.size();

    // Compute Q^T * f  (Q is orthogonal, so Q^T = Q^H = Q.transpose())
    Vector y(N, 0.0);
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            y[i] += Q[j][i] * f[j];
        }
    }

    // Back substitution for Rx = y
    Vector x(N, 0.0);
    for (int i = N - 1; i >= 0; --i) {
        x[i] = y[i];
        for (int j = i + 1; j < N; ++j) {
            x[i] -= R[i][j] * x[j];
        }
        x[i] /= R[i][i];
    }

    return x;
}
```

Ответ:

По итогу результат для пяти вычислений на каждый метод имеет такой вид. Метод QR выполняется дольше метода LU примерно в три раза. LU также опережает конкурента и по точности, я предполагаю, что решающим параметром такого результата здесь является выбор ведущего элемента в методе LU.

Size	Method	Median Time (ms)	Error
250	LU	180	1.457e-12
250	QR	416	5.508e-12
500	LU	1168	2.687e-11
500	QR	3263	4.309e-10
1000	LU	8951	1.882e-11
1000	QR	28889	2.972e-10