Практическое задание №5

Армбристер Никита Владиславович

Группа ПМИ-31

Вариант 1

Цель: сформировать практические навыки оценки числа обусловленности систем линейных алгебраических уравнений.

Задание: сформировать три СЛАУ с размерами N = {5, 10, 20} Ax=f. Реализовать программное решение через SVD разложение, также решить СЛАУ с использованием LU-разложения и QR-разложения на базе отражения Хаусхолдера из «Практического задания №4» (Далее по тексту «ПЗ#4»).

Решение:

Для решения данного практического задания был дополнен проект на языке C++ из ПЗ#4. Были изменены некоторые составляющие и добавлены новые.

Ссылка на репозиторий с работой:

https://github.com/Gribnik24/NumericalMethods-PracticalTasks/tree/main/Task%20%235.%20SLAE_Solver%20(SVD%20addition)

Теперь проект состоит из пяти 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. SVD_Solver.cpp – файл содержит SVD-разложение и решение с ним
6. LinearAlgebra.h – объявление всех используемых функций

Рассмотрим составляющие проекта подробнее:

- **LinearAlgebra.h**
  Назывной файл. Были добавлены функции объявления для метода SVD, новые матричные функции в Matrix operations, где самое основное - computeConditionNumber для подсчета числа обусловленности

```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);
Matrix transpose(const Matrix& A); // Добавлено
Matrix multiply(const Matrix& A, const Matrix& B); // Добавлено
Vector createRightHandSide(const Matrix& A);
double computeError(const Vector& x, const Vector& x_exact);
double computeConditionNumber(const Matrix& A);  // Добавлено

// 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);

// SVD decomposition
void svdDecomposition(const Matrix& A, Matrix& U, Vector& S, Matrix& V);  // Добавлено
Vector solveSVD(const Matrix& U, const Vector& S, const Matrix& V, const Vector& f);  // Добавлено
```

- **main.cpp**
  В файле main.cpp была изменена таблица с размерами матриц под условия ПЗ#5 и косметическая оформления часть для вывода результатов.

```cpp
int main() {
    std::vector<int> sizes = { 5, 10, 20 };
    const int num_measurements = 5;
    std::cout << std::fixed << std::setprecision(6);
    std::cout << "Size | Method |   Median Time  ||   Error   | Condition Number" << std::endl;
    std::cout << "-------------------------------------------------------------" << std::endl;
```

  Далее аналогично ПЗ#4 запускается цикл для каждого размера матрицы A. Для каждого из необходимых размеров матриц создается сама матрица A и вектор-результат f. Вводится новая переменная cond – число обусловленности матрицы. Для каждого из методов выводится погрешность численного вычисления и медиана времени работы. Время работы теперь замеряется в микросекундах (в ПЗ#4 расчет был в милисекундах). Это сделано потому, что на матрицах такого размера LU и QR разложения выдавали бы нулевое время работы.

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

    // 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::microseconds>(stop - start);
            timings.push_back(duration.count());
            final_error = computeError(x, x_exact);
        }
        long long median_time = calculateMedianTime(timings);
        std::cout << std::setw(4) << N << " | LU      | " << std::setw(15) << median_time
            << " | " << std::scientific << std::setprecision(3) << final_error
            << " | " << std::fixed << std::setprecision(2) << cond << std::endl;
    }
```

```cpp
// 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::microseconds>(stop - start);
        timings.push_back(duration.count());
        final_error = computeError(x, x_exact);
    }

    long long median_time = calculateMedianTime(timings);
    std::cout << std::setw(4) << N << " | QR     | " << std::setw(15) << median_time
        << " | " << std::scientific << std::setprecision(3) << final_error
        << " | " << std::fixed << std::setprecision(2) << cond << std::endl;
}
```

```cpp
// SVD 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 U, V;
        Vector S;
        svdDecomposition(A, U, S, V);
        Vector x = solveSVD(U, S, V, f);
        auto stop = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop - start);
        timings.push_back(duration.count());
        final_error = computeError(x, x_exact);
    }
    long long median_time = calculateMedianTime(timings);
    std::cout << std::setw(4) << N << " | SVD    | " << std::setw(15) << median_time
        << " | " << std::scientific << std::setprecision(3) << final_error
        << " | " << std::fixed << std::setprecision(2) << cond << std::endl;
}
```

- **MatrixOperations.cpp**
  Была изменена функция createMatrix, теперь она создает матрицу A для ПЗ#5, по условию варианта 1: $a_{ij} = \frac{1}{1+0.6i+2j}; i, j = \overline{1, N}$

```cpp
Matrix createMatrix(int N) {
    Matrix A(N, Vector(N));
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            A[i][j] = 1.0 / (1.0 + 0.6 * (i + 1) + 2.0 * (j + 1));
        }
    }
    return A;
}
```

Была добавлена функция computeConditionNumber – она считает число обусловленности для подаваемой матрицы. В реализации метода SVD-декомпозиции очень удачно одним из результатов будет вектор S – вектор сингулярных значений при разложении. Воспользуемся им. Его значения расположены по убыванию. Самый максимальный элемент – первый. Для нахождения минимального необходимо запустить цикл с конца, проверяя значения на признак равенства нулю.

```
double computeConditionNumber(const Matrix& A) {
    Matrix U, V;
    Vector S;
    svdDecomposition(A, U, S, V);

    if (S.empty()) {
        return std::numeric_limits<double>::infinity();
    }

    double sigma_max = S.front();

    double sigma_min = 0.0;
    const double epsilon = 1e-12;

    for (auto it = S.rbegin(); it != S.rend(); ++it) {
        if (std::abs(*it) > epsilon) {
            sigma_min = *it;
            break;
        }
    }

    if (sigma_min == 0.0) {
        return std::numeric_limits<double>::infinity();
    }

    return sigma_max / sigma_min;
}
```

Были добавлены функции для транспонирования и умножения матриц

```
Matrix transpose(const Matrix& A) {
    int m = A.size(), n = A[0].size();
    Matrix At(n, Vector(m));
    for (int i = 0; i < m; ++i)
        for (int j = 0; j < n; ++j)
            At[j][i] = A[i][j];
    return At;
}

Matrix multiply(const Matrix& A, const Matrix& B) {
    int m = A.size(), n = B[0].size(), p = B.size();
    Matrix C(m, Vector(n, 0));
    for (int i = 0; i < m; ++i)
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < p; ++k)
                C[i][j] += A[i][k] * B[k][j];
    return C;
}
```

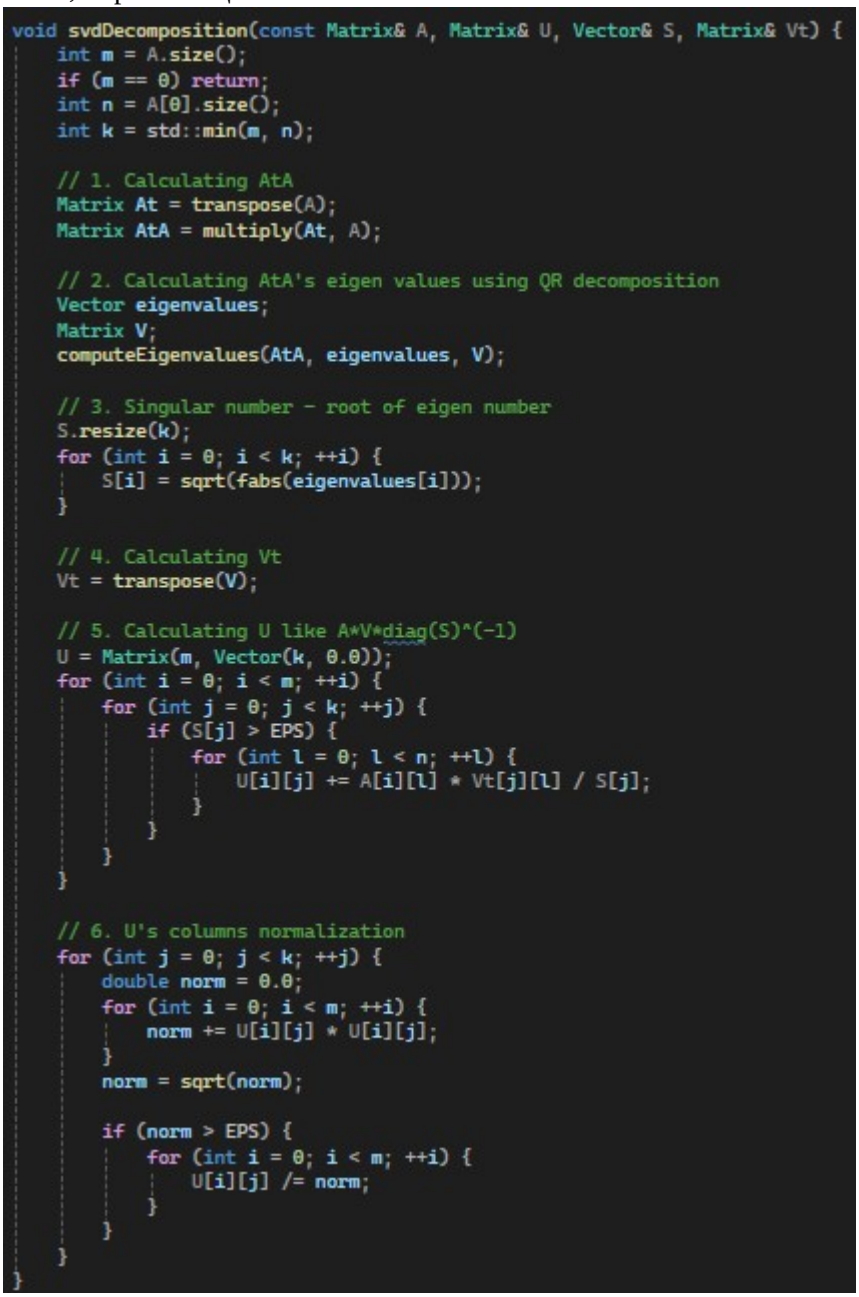
- **LU_Solver.cpp**
  Без изменений с версией ПЗ#4

- **QR_Solver.cpp**
  Без изменений с версией ПЗ#4

- **SVD_Solver.cpp**
  Полностью новая часть проекта. Файл состоит из трех функций. computeEigenvalues используя функцию householderQR (QR-разложение через Хаусхолдера из QR_Solver.cpp) собственными значениями.
  Матрица A постепенно приводится к верхнетреугольному виду, где диагональные элементы как раз и будут собственными значениями. Одновременно с этим находятся векторы V

```cpp
// Calculating eigen values using QR decomposition
void computeEigenvalues(const Matrix& A, Vector& eigenvalues, Matrix& eigenvectors) {
    int n = A.size();
    Matrix Ak = A;
    Matrix Q, R;
    eigenvectors = Matrix(n, Vector(n, 0.0));

    for (int i = 0; i < n; ++i)
        eigenvectors[i][i] = 1.0;

    for (int iter = 0; iter < MAX_ITER; ++iter) {
        householderQR(Ak, Q, R);

        Ak = multiply(R, Q);

        Matrix temp = multiply(eigenvectors, Q);
        eigenvectors = temp;
    }

    eigenvalues.resize(n);
    for (int i = 0; i < n; ++i)
        eigenvalues[i] = Ak[i][i];

    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            if (eigenvalues[i] < eigenvalues[j]) {
                std::swap(eigenvalues[i], eigenvalues[j]);
                for (int k = 0; k < n; ++k)
                    std::swap(eigenvectors[k][i], eigenvectors[k][j]);
            }
        }
    }
}
```

svdDecomposition выполняет функцию как раз нахождения трех матриц для SVD-разложения: произведение A^TA, нахождение собственных значений A^TA через computeEigenvalues и векторов V, нахождение сингулярных чисел (корней результатов computeEigenvalues), транспонирование V, нахождение U по геометрическому свойству SVD-разложения, нормализация U.



```
void svdDecomposition(const Matrix& A, Matrix& U, Vector& S, Matrix& Vt) {
    int m = A.size();
    if (m == 0) return;
    int n = A[0].size();
    int k = std::min(m, n);

    // 1. Calculating AtA
    Matrix At = transpose(A);
    Matrix AtA = multiply(At, A);

    // 2. Calculating AtA's eigen values using QR decomposition
    Vector eigenvalues;
    Matrix V;
    computeEigenvalues(AtA, eigenvalues, V);

    // 3. Singular number - root of eigen number
    S.resize(k);
    for (int i = 0; i < k; ++i) {
        S[i] = sqrt(fabs(eigenvalues[i]));
    }

    // 4. Calculating Vt
    Vt = transpose(V);

    // 5. Calculating U like A*V*diag(S)^(-1)
    U = Matrix(m, Vector(k, 0.0));
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < k; ++j) {
            if (S[j] > EPS) {
                for (int l = 0; l < n; ++l) {
                    U[i][j] += A[i][l] * Vt[j][l] / S[j];
                }
            }
        }
    }

    // 6. U's columns normalization
    for (int j = 0; j < k; ++j) {
        double norm = 0.0;
        for (int i = 0; i < m; ++i) {
            norm += U[i][j] * U[i][j];
        }
        norm = sqrt(norm);

        if (norm > EPS) {
            for (int i = 0; i < m; ++i) {
                U[i][j] /= norm;
            }
        }
    }
}
```

SolveSVD выполняет функцию последовательного перемножения компонент

$$x = U\,\Sigma^{-1}U^{T}f$$

```
Vector solveSVD(const Matrix& U, const Vector& S, const Matrix& Vt, const Vector& f) {
    int m = U.size();
    int n = Vt[0].size();

    // 1. Ut * f
    Vector y(m, 0.0);
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < m; ++j) {
            y[i] += U[j][i] * f[j];
        }
    }

    // 2. diag(S)^(-1) * (Ut * f)
    double max_s = *std::max_element(S.begin(), S.end());
    double threshold = max_s * std::max(m, n) * EPS;

    for (int i = 0; i < S.size(); ++i) {
        if (S[i] > threshold) {
            y[i] /= S[i];
        }
        else {
            y[i] = 0.0;
        }
    }

    // 3. V * (diag(S)^(-1) * Ut * f)
    Vector x(n, 0.0);
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            x[i] += Vt[j][i] * (j < y.size() ? y[j] : 0.0);
        }
    }

    return x;
}
```

Ответ:

Результаты вышли такие. Condition Number – число обусловленности ожидаемо большое, в этом и был смысл ПЗ#5. Время у метода SVD больше, чем у LU и QR. Для матрицы N=5 SVD показал самый худший результат в плане ошибки, но затем на N={10 20} смог опередить своих конкурентов.

Size	Method	Median Time	Error	Condition Number
5	LU	19	9.473e-11	14317095.01
5	QR	43	9.838e-10	14317095.01
5	SVD	4707	2.833e-02	14317095.01
10	LU	27	8.965e-02	48332777.68
10	QR	82	8.144e-01	48332777.68
10	SVD	14912	6.207e-02	48332777.68
20	LU	132	1.133e+02	1724402304.64
20	QR	442	4.042e+01	1724402304.64
20	SVD	85295	3.077e+01	1724402304.64

```
Для продолжения нажмите любую клавишу . . .
```

Практическое задание №5.

После доработки

Был пересмотрен файл SVD_Solver.cpp и внесен ряд изменений:

1. Добавление новых констант и изменений названий существовавших. Теперь SVD_EPS отвечает за точность в самом разложении SVD, а SVD_THRESHOLD – порог для измерений в самом решении СЛАУ для усечений. Уменьшил кол-во итераций

```
const double SVD_EPS = 1e-14;  // increase SVD accuracy
const int MAX_ITER = 10;
const double SVD_THRESHOLD = 1e-10;
```

2. Улучшение в нормализации векторов U. Теперь они проходят две стадии нормализации – это увеличило их точность

```
// 6. New U's columns normalization
for (int j = 0; j < k; ++j) {

    // first normalization
    double norm = 0.0;
    for (int i = 0; i < m; ++i) {
        norm += U[i][j] * U[i][j];
    }
    norm = sqrt(norm);
    if (norm > SVD_EPS) {
        for (int i = 0; i < m; ++i) {
            U[i][j] /= norm;
        }
    }

    // second normalization
    for (int p = 0; p < j; ++p) {
        double dot = 0.0;
        for (int i = 0; i < m; ++i) {
            dot += U[i][p] * U[i][j];
        }
        for (int i = 0; i < m; ++i) {
            U[i][j] -= dot * U[i][p];
        }
    }
```

Новый ответ. Новые результаты выглядят так. Уменьшилось время и показатель ошибки

Size	Method	Median Time	Error	Condition Number
5	LU	9	9.473e-11	14317095.01
5	QR	19	9.838e-10	14317095.01
5	SVD	451	5.765e-06	14317095.01
10	LU	27	8.965e-02	48332777.68
10	QR	57	8.144e-01	48332777.68
10	SVD	1697	4.980e-03	48332777.68
20	LU	121	1.133e+02	1724402304.64
20	QR	225	4.042e+01	1724402304.64
20	SVD	9276	9.549e-03	1724402304.64