Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИТМО

Лабораторная работа №1

по дисциплине
«Линейная алгебра и анализ данных»

Выполнил:
студент группы J3112
Бессонов Александр Русланович
ИСУ 465220

Санкт-Петербург
2024

Содержание

Цели и задачи лабораторной работы

Цель

Изучить и реализовать алгоритмы работы с матрицами в разреженно-строчном формате, включая основные операции, такие как подсчет следа, доступ к элементам, арифметические операции, а также расчет определителя и проверку существования обратной матрицы. Это позволит закрепить знания в области линейной алгебры и освоить практическое применение алгоритмов работы с матрицами на языке программирования C++.

Задачи

Задача 1: Хранение матриц в разреженно-строчном виде

1. Разработать класс для представления матрицы в разреженно-строчном формате.
2. Реализовать метод подсчета следа матрицы.
3. Реализовать метод доступа к элементу матрицы по индексу строки и столбца (нумерация с единицы).

Задача 2: Операции над матрицами

1. Реализовать функцию сложения двух матриц, заданных экземплярами разработанного класса.
2. Реализовать функцию умножения матрицы на скаляр.
3. Реализовать функцию умножения двух матриц.
4. Выполнить проверки на корректность входных данных (совместимость размеров матриц).

Задача 3: Вычисление определителя и проверка существования обратной матрицы

1. Разработать функцию для вычисления определителя квадратной матрицы.
2. Определить, существует ли обратная матрица (определитель не равен нулю).
3. Реализовать вывод результата: значение определителя и наличие обратной матрицы (ответ «да»/«нет»).

Структура проекта

Структура проекта

```
/research-linear-algebra
|-- include/
|   |-- SparseMatrix.h
|   |-- MatrixAddition.h
|   |-- MatrixScalarMultiplication.h
|   |-- MatrixMultiplication.h
|   |-- Determinant.h
|
|-- src/
|   |-- SparseMatrix.cpp
|   |-- MatrixAddition.cpp
|   |-- MatrixScalarMultiplication.cpp
|   |-- MatrixMultiplication.cpp
|   |-- Determinant.cpp
|
|-- CMakeLists.txt
|-- main.cpp
```

Основания выбора структуры:

Проект разделён на две основные директории: `include/` и `src/`. Данная структура разработана на основе принципов модульности и разделения обязанностей, что делает проект более организованным, читаемым и поддерживаемым.

- **Директория `include/`:** Содержит заголовочные файлы, где описаны интерфейсы классов и функций. Это позволяет отделить декларации от реализаций, что способствует удобству работы с проектом, упрощает сопровождение кода и тестирование. Основные файлы:
  - `SparseMatrix.h` — заголовочный файл класса разреженной матрицы. Содержит объявления методов подсчёта следа, доступа к элементам и базовых операций.
  - `MatrixAddition.h` — заголовочный файл функции сложения матриц.
  - `MatrixScalarMultiplication.h` — заголовочный файл функции умножения матрицы на скаляр.
  - `MatrixMultiplication.h` — заголовочный файл функции умножения двух матриц.
  - `Determinant.h` — заголовочный файл функции вычисления определителя матрицы.

- **Директория `src/`:** Содержит файлы с реализациями функций и классов, объявленных в заголовочных файлах. Использование отдельной директории для исходных файлов упрощает компиляцию проекта и улучшает читаемость структуры.
  - `SparseMatrix.cpp` — реализация методов класса разреженной матрицы.
  - `MatrixAddition.cpp` — реализация функции сложения матриц.
  - `MatrixScalarMultiplication.cpp` — реализация функции умножения матрицы на скаляр.
  - `MatrixMultiplication.cpp` — реализация функции умножения двух матриц.
  - `Determinant.cpp` — реализация функции вычисления определителя матрицы.
- **`main.cpp`:** Главный файл программы, содержащий тестовые примеры для проверки корректности реализации всех функций. Его структура включает создание объектов, вызовы методов, тестовые примеры и вывод результатов.

Данная структура проекта обеспечивает следующие преимущества:

1. **Разделение интерфейса и реализации:** Декларации классов и функций отделены от их реализаций, что облегчает понимание архитектуры проекта.
2. **Масштабируемость:** Добавление новых функций и модулей происходит легко благодаря чёткой организации директорий.
3. **Удобство поддержки:** Разделение на модули упрощает отладку, тестирование и сопровождение проекта в дальнейшем.
4. **Повышенная читаемость:** Логическая структура проекта делает его код более понятным для разработчиков.

Решение задач

Задача №1: Хранение матрицы в разреженно-строчном формате

Теоретическая справка

Разреженная матрица — это матрица, в которой большинство элементов равны нулю. Для хранения таких матриц неэффективно использовать традиционные двумерные массивы, так как это приводит к увеличению потребляемой памяти. Вместо этого применяется структура данных, которая хранит только ненулевые элементы.

В данной реализации разреженная матрица представлена с использованием вложенных хэш-таблиц `unordered_map`, где первый уровень хранит номера строк, а второй — номера столбцов и соответствующие значения элементов. Такой подход обеспечивает доступ к элементам за амортизированное время $O(1)$.

Логика разработки:

1. Для представления матрицы был выбран класс `SparseMatrix`.
2. Хранение данных реализовано через `unordered_map<int, unordered_map<int, double»`, что позволяет экономно использовать память.
3. Проверка на корректность индексов помогает предотвратить выход за границы матрицы.
4. Для работы с матрицей реализованы следующие методы:
   - `addValue` — добавление значения в матрицу.
   - `getValue` — получение значения элемента.
   - `trace` — подсчёт следа матрицы.
   - `print` — вывод матрицы на экран.

Код реализации

```cpp
#ifndef SPARSE_MATRIX_H
#define SPARSE_MATRIX_H

#include <iostream>
#include <unordered_map>
using namespace std;

class SparseMatrix {

public:
    SparseMatrix(int r, int c);

    void addValue(int row, int col, double value);
    double getValue(int row, int col) const;
    double trace() const;
    void print() const;

private:
    int cols;
    int rows;
    unordered_map<int, unordered_map<int, double>> data;
};

#endif // SPARSE_MATRIX_H
```

Листинг 1: SparseMatrix.h

```cpp
#include "SparseMatrix.h"
#include <cmath>

SparseMatrix::SparseMatrix(int r, int c) : rows(r), cols(c) {}

void SparseMatrix::addValue(int row, int col, double value) {
    if (row >= 1 && row <= rows && col >= 1 && col <= cols && fabs(
        value) > 1e-9) {
        data[row][col] = value;
    }
}

double SparseMatrix::getValue(int row, int col) const {
    if (data.count(row) && data.at(row).count(col)) {
        return data.at(row).at(col);
    }
    return 0.0;
}

double SparseMatrix::trace() const {
    double tr = 0;
    for (int i = 1; i <= min(rows, cols); ++i) {
        tr += getValue(i, i);
    }
    return tr;
}

void SparseMatrix::print() const {
    for (int i = 1; i <= rows; ++i) {
        for (int j = 1; j <= cols; ++j) {
            cout << getValue(i, j) << " ";
        }
        cout << endl;
    }
}
```

Листинг 2: SparseMatrix.cpp

Микровывод:

Использование вложенных хэш-таблиц позволило эффективно хранить разреженные матрицы, обеспечив компактность и быстрый доступ к элементам. Методы класса обеспечивают корректную работу с матрицей и предоставляют базовые операции, необходимые для последующих задач.

Задача №2: Операции с матрицами

Теоретическая справка

Операции с матрицами, такие как сложение, умножение на скаляр и умножение матриц, являются основными элементами линейной алгебры. Для разреженных матриц выполнение таких операций требует обработки только ненулевых элементов, что позволяет существенно снизить вычислительные затраты.

В данной реализации для хранения элементов используются вложенные хэш-таблицы. Сложение и умножение матриц реализованы с учетом минимизации операций с нулевыми значениями. Проверки на совместимость размеров матриц гарантируют корректность выполнения операций.

Обоснование выбора подхода

1. **Сложение матриц:** Сложение выполняется путём поэлементного суммирования соответствующих элементов двух матриц с одинаковыми размерами. Если сумма элементов не равна нулю, результат добавляется в выходную матрицу.

2. **Умножение матрицы на скаляр:** Каждый ненулевой элемент матрицы умножается на заданный скаляр, а результат сохраняется в новой матрице.

3. **Умножение матриц:** Выполняется с проверкой совместимости размеров матриц. Для каждого ненулевого элемента первой матрицы проверяются соответствующие элементы второй матрицы, и вычисляются их произведения.

Код реализации

```cpp
#ifndef MATRIX_ADDITION_H
#define MATRIX_ADDITION_H

#include "SparseMatrix.h"

SparseMatrix addMatrices(const SparseMatrix& A, const SparseMatrix& B);

#endif // MATRIX_ADDITION_H
```

Листинг 3: MatrixAddition.h

```cpp

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

SparseMatrix addMatrices(const SparseMatrix& A, const SparseMatrix& B) {
    if (A.rows != B.rows || A.cols != B.cols) {
        throw invalid_argument("␣␣␣!");
    }

    SparseMatrix result(A.rows, A.cols);
    for (int i = 1; i <= A.rows; ++i) {
        for (int j = 1; j <= A.cols; ++j) {
            double value = A.getValue(i, j) + B.getValue(i, j);
            if (fabs(value) > 1e-9) {
                result.addValue(i, j, value);
            }
        }
    }
    return result;
}
```

Листинг 4: MatrixAddition.cpp

```cpp
#ifndef MATRIX_SCALAR_MULTIPLICATION_H
#define MATRIX_SCALAR_MULTIPLICATION_H

#include "SparseMatrix.h"

SparseMatrix multiplyMatrixScalar(const SparseMatrix& A, double scalar);

#endif // MATRIX_SCALAR_MULTIPLICATION_H
```

Листинг 5: MatrixScalarMultiplication.h

```cpp
#include "MatrixScalarMultiplication.h"

SparseMatrix multiplyMatrixScalar(const SparseMatrix& A, double scalar) {
    SparseMatrix result(A.rows, A.cols);
    for (const auto& row : A.data) {
        for (const auto& element : row.second) {
            result.addValue(row.first, element.first, element.second * scalar);
        }
    }
    return result;
}
```

Листинг 6: MatrixScalarMultiplication.cpp

```cpp
#ifndef MATRIX_MULTIPLICATION_H
#define MATRIX_MULTIPLICATION_H

#include "SparseMatrix.h"

SparseMatrix multiplyMatrices(const SparseMatrix& A, const SparseMatrix& B);

#endif // MATRIX_MULTIPLICATION_H
```

Листинг 7: MatrixMultiplication.h

```cpp
#include "MatrixMultiplication.h"

SparseMatrix multiplyMatrices(const SparseMatrix& A, const SparseMatrix& B) {
    if (A.cols != B.rows) {
        throw invalid_argument("␣␣␣␣!");
    }

    SparseMatrix result(A.rows, B.cols);
    for (const auto& row : A.data) {
        for (const auto& element : row.second) {
            int i = row.first;
            int k = element.first;
            double value = element.second;

            for (int j = 1; j <= B.cols; ++j) {
                result.addValue(i, j, result.getValue(i, j) + value * B.getValue(k, j));
            }
        }
    }
    return result;
}
```

Листинг 8: MatrixMultiplication.cpp

Микровывод:

Реализация основных операций с матрицами, таких как сложение, умножение на скаляр и умножение матриц, обеспечила корректную обработку разреженных матриц с минимальными вычислительными затратами. Применение структуры данных на основе вложенных хэш-таблиц позволило обрабатывать только ненулевые элементы, что значительно сократило объём используемой памяти и время выполнения операций.

Задача №3: Определитель и обратимость

Теоретическая справка

Определитель квадратной матрицы — это скалярная величина, которая характеризует множество свойств матрицы, включая её обратимость, ранг и поведение при линейных преобразованиях. Если определитель равен нулю, матрица считается вырожденной, а её строки или столбцы линейно зависимы. Такая матрица не имеет обратной. Если определитель отличен от нуля, матрица обратима, а её строки и столбцы линейно независимы.

Определитель можно вычислить различными методами, такими как разложение по строке, метод Гаусса, метод миноров и использование LU-разложения. В данной реализации использован метод Гаусса, заключающийся в приведении матрицы к верхнетреугольному виду путём элементарных строковых операций, после чего определитель находится как произведение элементов главной диагонали. Перестановка строк меняет знак определителя на противоположный, что учитывается в вычислениях.

Проверка обратимости матрицы сводится к вычислению её определителя: если он равен нулю, матрица не имеет обратной, иначе обратная матрица существует.

Обоснование выбора подхода

1. **Метод Гаусса:** Метод Гаусса является одним из наиболее эффективных алгоритмов для вычисления определителя. Его вычислительная сложность составляет $O(n^3)$, что приемлемо для большинства задач.

2. **Перестановка строк:** Если ведущий элемент строки равен нулю, строки матрицы меняются местами для предотвращения деления на ноль. Такая перестановка инвертирует знак определителя.

3. **Проверка обратимости:** Проверка обратимости матрицы реализована путём сравнения определителя с числом, близким к нулю (в пределах машинной точности). Это позволяет учитывать погрешности при работе с вещественными числами.

Код реализации

```cpp
#ifndef DETERMINANT_H
#define DETERMINANT_H

#include <vector>
using namespace std;

double determinant(vector<vector<double>> matrix, int n);
bool isInvertible(const vector<vector<double>>& matrix, int n);

#endif // DETERMINANT_H
```

Листинг 9: Determinant.h

```cpp
#include "Determinant.h"
#include <cmath>

double determinant(vector<vector<double>> matrix, int n) {
    double det = 1.0;

    for (int i = 0; i < n; ++i) {
        if (fabs(matrix[i][i]) < 1e-9) {
            bool swapped = false;

            for (int j = i + 1; j < n; ++j) {
                if (fabs(matrix[j][i]) > 1e-9) {
                    swap(matrix[i], matrix[j]);
                    det *= -1;
                    swapped = true;
                    break;
                }
            }
            if (!swapped) return 0.0;
        }

        det *= matrix[i][i];

        for (int j = i + 1; j < n; ++j) {
            double ratio = matrix[j][i] / matrix[i][i];
            for (int k = i; k < n; ++k) {
                matrix[j][k] -= ratio * matrix[i][k];
            }
        }
    }
    return det;
}

bool isInvertible(const vector<vector<double>>& matrix, int n) {
    return fabs(determinant(matrix, n)) > 1e-9;
}
```

Листинг 10: Determinant.cpp

Микровывод:

Использование метода Гаусса позволило вычислять определитель матрицы за $O(n^3)$ с учётом перестановок строк. Такой подход обеспечил корректное определение обратимости матрицы. Проверка на малые значения в главной диагонали позволила избежать ошибок, связанных с делением на ноль. Благодаря учёту машинной точности алгоритм остаётся устойчивым даже при работе с вещественными числами, минимизируя возможные вычислительные ошибки.

Тестирование

Реализация:

1. **Выбор тестовых случаев:** Для тестирования выбраны квадратные матрицы с разными значениями.
2. **Тестирование арифметических операций:** Проведена проверка сложения, умножения на скаляр и умножения матриц.
3. **Проверка обратимости:** Осуществлена проверка обратимости матриц путём вычисления их определителя и сравнения его с нулём.

Код тестирования

```
#include <iostream>
#include "SparseMatrix.h"
#include "MatrixAddition.h"
#include "MatrixScalarMultiplication.h"
#include "MatrixMultiplication.h"
#include "Determinant.h"

using namespace std;

int main() {
    //           1:
    SparseMatrix A(3, 3), B(3, 3);
    A.addValue(1, 1, 1.0); A.addValue(2, 2, 2.0); A.addValue(3, 3, 3.0)
        ;
    B.addValue(1, 1, 4.0); B.addValue(2, 2, 5.0); B.addValue(3, 3, 6.0)
        ;

    cout << "                  ␣                ␣            ␣A␣  ␣B:"
       << endl;
    SparseMatrix C = addMatrices(A, B);
    C.print();

    //           2:
    vector<vector<double>> mat = {{2, -1, 0}, {1, 6, -1}, {1, 3, 5}};
    cout << "                        ␣              :␣" << determinant(
       mat, 3) << endl;
    cout << (isInvertible(mat, 3) ? "               ␣                " :
       "              ␣                    ") << endl;

    //           3:
    cout << "\n                 ␣                  ␣              ␣A␣
        ␣            ␣2.0:" << endl;
    SparseMatrix D = multiplyMatrixScalar(A, 2.0);
    D.print();

    return 0;
}
```

Листинг 11: main.cpp

output:

```
Результат сложения матриц A и B:
5 0 0
0 7 0
0 0 9

Определитель матрицы: 49
Матрица обратима

Результат умножения матрицы A на скаляр 2.0:
2 0 0
0 4 0
0 0 6
```

Вывод

В ходе выполнения лабораторной работы была разработана система работы с матрицами в разреженно-строчном формате, включающая хранение, основные арифметические операции, вычисление определителя и проверку обратимости. Для реализации использовались эффективные структуры данных на основе вложенных хэш-таблиц, что позволило сократить объём используемой памяти за счёт хранения только ненулевых элементов. Такой подход обеспечил компактность и высокую производительность алгоритмов.

Основные математические операции, такие как сложение матриц, умножение на скаляр и умножение матриц, были реализованы с учётом минимизации вычислений с нулевыми элементами. Проверки совместимости размеров матриц гарантировали корректность выполнения операций, предотвращая логические ошибки. Особое внимание уделялось вычислению определителя матрицы с использованием метода Гаусса, который обеспечил точность и надёжность вычислений за приемлемое время $O(n^3)$.

Кроме того, была реализована проверка обратимости матриц, основанная на вычислении определителя с учётом машинной точности, что позволило избежать ошибок, связанных с округлением при работе с вещественными числами. Перестановка строк в случае нулевого элемента главной диагонали обеспечила устойчивость алгоритма.

Работа над проектом способствовала углублённому изучению алгоритмов линейной алгебры и структур данных. Также был освоен модульный подход к разработке программного обеспечения с разделением кода на заголовочные и исходные файлы. Использование классов и функций обеспечило логическую организацию программы, упростило её поддержку и тестирование.

Таким образом, выполнение лабораторной работы дало ценный опыт разработки сложных алгоритмов, применения принципов объектно-ориентированного программирования, тестирования кода и его структурирования. Полученные знания могут быть применены в дальнейшем при разработке научных и инженерных приложений, требующих работы с большими разреженными матрицами.