Федеральное агентство по образованию Российской Федерации

Государственное образовательное учреждение

высшего профессионального образования

Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий математики и механики

Направление подготовки: «Фундаментальная информатика и информационные технологии»

Отчёт по лабораторной работе

**Реализация классов для работы с матрицами и векторами.**

Выполнил:

студент ИИТММ гр. 381906-2

Кулемин П. А.

Проверил:

ассистент каф. математического обеспечения

и суперкомпьютерных технологий

Лебедев И.Г.

Нижний Новгород

2019 г.

Содержание

I. [Введение 3](#_Toc24457830)

II. [Постановка задачи](#_Toc24457831) 4

III. [Руководство пользователя](#_Toc24457832) 5

IV. [Руководство программиста](#_Toc24457833) 6

1. [Описание структуры программы](#_Toc24457834) 6

2. [Описание структур данных](#_Toc24457835) 7

3. [Описание алгоритмов](#_Toc24457836) 7

V. [Эксперименты](#_Toc24457837) 10

VI. [Заключение](#_Toc24457838) 11

VII. [Литература](#_Toc24457839) 12

VIII. [Приложение 1](#_Toc24457840)3

# 1. Введение

Матрицы и векторы, это важная часть векторной алгебры, и поэтому информатизация и автоматизация в этой области математики крайне важна. Именно для этого мной были созданы классы для работы с векторами и матрицами, а также произведена теоретическая и практическая оценка времени работы различных арифметических операций над матрицами, а также времени различных сортировок, которые можно произвести над векторами. Также было произведено сравнение этих оценок.

# 2. Постановка задачи

Реализовать классы для работы с векторами и матрицами использовать шаблоны. Продемонстрировать их работу на примере (написать в main пример).

Должны быть:

* конструкторы (по умолчанию, инициализатор, копирования).
* деструктор.
* доступ к защищенным полям.
* перегруженные операции: +, -, \*, /, =, ==, [].
* потоковый ввод и вывод.
* перегруженные операции +, -, \*, / должны быть реализованы для векторов (вектор +, -, \*, / вектор), матриц (матрица +, -, \* матрица), матрично-векторные (матрица \* вектор и наоборот).
* в классе вектор должна быть возможность отсортировать его тремя способами.

Сравнить время работы, и сделать выводы.

# 3. Руководство пользователя

Шаги, совершаемые пользователем:

1. Создать объект типа Matrix или Vector, в зависимости от необходимости.
2. Расширить матрицу или вектор на нужное количество элементов, используя конструкторы либо, методы доступа к защищенным полям.
3. Произвести необходимые математические операции.

# 4. Руководство программиста

## 4.1. Описание структуры программы

Программа состоит из одного решения.

В решении Vector определен 5 модулей main.cpp, Matrix.h, Matrix\_doing.h, Vector.h, Vector\_doing.h.

* В модуле main.cpp определена стандартная функция int main(), внутри которой содержится набор действий с объектами типа Matrix и Vector.
* В модуле Matrix.h определен класс Matrix, а также объявлены все его методы.
* В модуль Matrix\_doing.h вынесены все определения методов класса Matrix
* В модуле Vector.h определен класс Vector – наследник класса Matrix.
* В модуль Vector\_doing.h вынесены все определения методов класса Vector.

## Описание структур данных

В программе определены два следующих шаблонных класса:

Class Matrix

Class Vector

Внутри класса Matrix определены следующие поля:

* Type\*\* matrix – шаблонный двойной указатель.
* int width – целочисленный размер ширины матрицы
* int length – целочисленный размер длины матрицы

Внутри класса Matrix определен следующий набор public-методов

(плюс конструкторы и деструктор):

* Matrix() – конструктор по умолчанию, не принимает никаких параметров, инициализирует все три поля 0, через списки инициализации.
* Matrix(int, int) – конструктор инциализатор, принимает на вход два параметра типа int, создает динамический массив заданных размеров, заполняет его 0, поля width и length, становятся значениями переданными в конструктор соответственно.
* Matrix(const Matrix& lhs) – конструктор копирования, принимает на вход объект типа Matrix, создает объект с теми же характеристиками что и переданный, и матрицей того же содержимого.
* Virtual ~Matrix() – виртуальный деструктор, очищает выделенную методами и конструкторами память.
* Int get\_width() – метод возвращающий ширину матрицы
* Int get\_length() – метод возвращающий длину матрицы
* Type\*\* get\_array() – метод возвращающий массив содержащий в себе все элементы матрицы.
* Void set\_width\_length(int,int) – метод принимающий новые размеры матрицы, приводящий матрицу к этим размерам. В случае если новая матрица больше старой, новые элементы равняются нулю. В противном случае, все элементы на которые хватило места, останутся в матрице. В случае если матрица имеет любую размерность равную 0, метод ведет себя аналогично конструктору инциализатору
* Type\*& operator [ ](int) - перегрузка оператора индексации, позволяет возвращать элемент матрицы по индексу.
* Friend Matrix operator +(Matrix & lhs, Matrix & rhs) – перегрузка оператора суммы, позволяет возвращать матрицу, являющуюся суммой двух матриц.
* Friend Matrix operator –(Matrix& lhs, Matrix& rhs) – перегрузка оператора –, аналогична в своей сути перегрузке оператора +
* Matrix& operator =(const Matrix & lhs) - перегрузка оператора присвоить, позволяет присваивать один объект типа Matrix другому
* friend bool operator ==(Matrix& lhs, Matrix& rhs) – перегрузка оператора сравнения, позволяет сравнивать объекты типа Matrix
* friend Matrix operator \*(const Matrix& lhs, const Matrix& rhs) - перегрузка оператора умножить, позволяет перемножать обьекты типа Matrix
* friend ostream& operator <<(ostream& out, const Matrix& rhs) – перегрузка оператора вывода, позволяет выводить матрицы на экран
* friend istream& operator >>(istream& in, Matrix& rhs) - перегрузка оператора ввода, позволяет вводить матрицы уже заданного размера, либо задавать размер и вводить самостоятельно.

Единственным «Private» методом, является метод

* void MemorySize(int, int) –выделяет память до размеров матрицы переданных в функцию

Шаблонный класс Vector, является public, наследником класса Matrix, поэтому никаких полей в этом классе не определено. Также почти 70% методов, он наследует от класса-родителя, поэтому внутри него, определены только специфичные для него методы:

* T1\* get\_vector() - метод возвращающий массив содержащий в себе все элементы объекта Vector.
* T1 operator [](const int) - перегрузка оператора индексации, позволяет возвращать элемент вектора по индексу.
* void set\_length(int) - метод принимающий новые размеры вектора, приводящий вектор к этим размерам. В случае если новый вектор больше старого, новые элементы равняются нулю. В противном случае, все элементы на которые хватило места, останутся в векторе. В случае если вектор имеет размерность равную 0, метод ведет себя аналогично конструктору инциализатору
* friend Vector operator /(Vector& lhs, Vector& rhs) – перегрузка оператора / возвращает вектор, координаты которого являются частным от деления двух соответствующих координат операндов. В случае, если делитель равен нулю, значение частного считается равным делимому.
* friend Type operator \*(Vector<Type>& lhs, Vector & rhs) – перегрузка оператора умножить, возвращает скалярное произведение операндов.
* friend istream& operator >>(istream& in, Vector& rhs) - перегрузка оператора ввода, позволяет вводить матрицы уже заданного размера, либо задавать размер и вводить самостоятельно.
* clock\_t BubbleSort() - сортировка методом «Пузырек». Возвращает разницу между тактами процессора при входе в сортировку и при выходе из нее.
* clock\_t InsertionSort() - сортировка методом «Вставка». Возвращает разницу между тактами процессора при входе в сортировку и при выходе из нее.

## Описание алгоритмов

* Алгоритм суммирования или вычитания матриц:

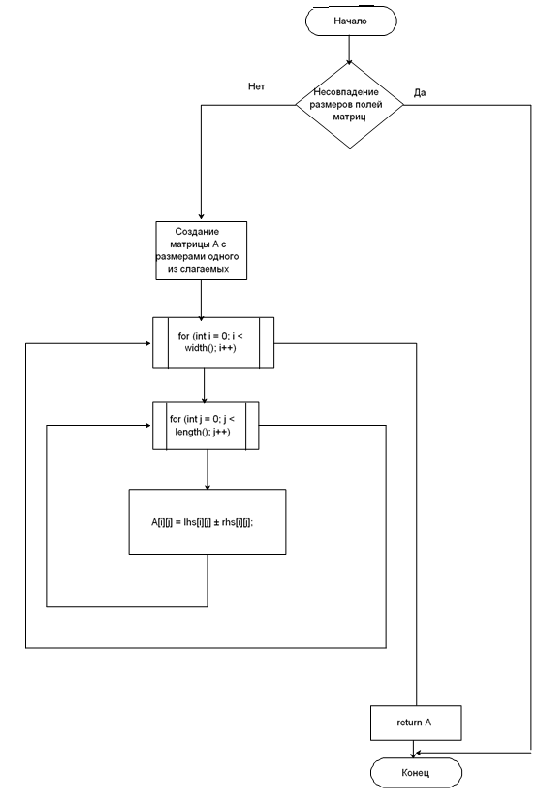


Рисунок 1: Блок-схема алгоритма сложения/вычитания матриц

# Алгоритм умножения матрицы на матрицу:

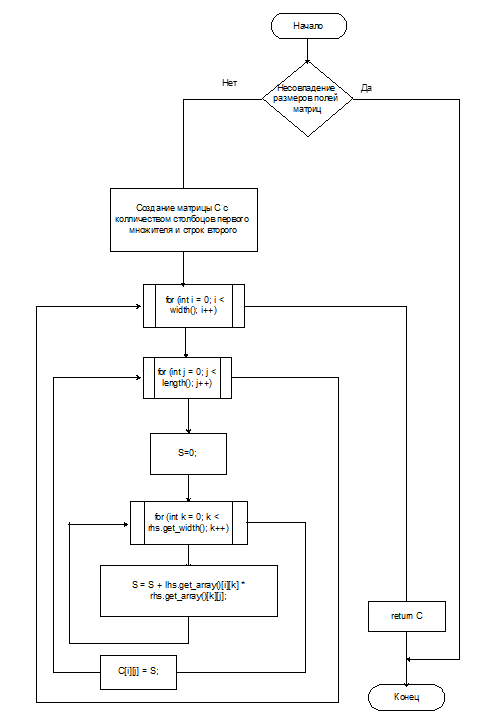


Рисунок 2: Блок-схема алгоритма умножения матриц.

Поскольку класс Vector это публичный наследник класса Matrix, то для него нет смысла описывать каким-либо образом арифметическую операцию сложения, так как эта операция в числе других, наследуется от класса Matrix.

* Алгоритм деления двух векторов:

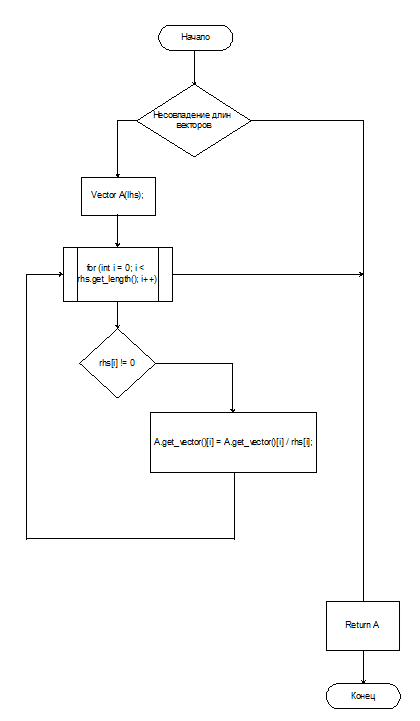


Рисунок 3: Блок-схема умножения одной матрицы на другою.

* Алгоритм умножения двух векторов:

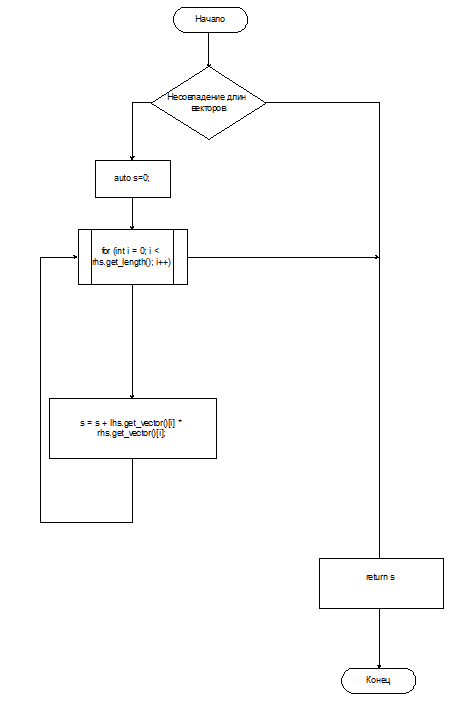


Рисунок 4: Блок-схема умножения двух векторов.

# Эксперименты

Оценим время, которое занимают матричные и векторно-матричные операции, с помощью асимптотической сложности. Рассмотрим код отвечающий за суммирование матриц:

Matrix<Type> A(lhs.get\_width(), rhs.get\_length());

for (int i = 0; i < rhs.get\_width(); i++)

{

for (int j = 0; j < rhs.get\_length(); j++)

{

A[i][j] = lhs[i][j] + rhs[i][j];

}

}

Асимптотическая сложность этого кода:

Теперь мы произведем замеры времени сложения квадратных матриц (чтобы точнее соотнести результаты замеров с асимптотической сложностью):



Рисунок 5: Замеры времени, полученные при выполнении сложения матриц (сек.).

На основе полученных данных, построим таблицу соотношения размеров матриц и времени их суммирования.

|  |  |
| --- | --- |
| Количество элементов в матрице | Время выполнения (сек.) |
| 1000 | 0.265 |
| 2000 | 0.785 |
| 4000 | 3.89 |

Таблица 1: Результаты замеров времени суммирования матриц.

Как мы можем видеть действительно, при увеличении количества элементов в 2 раза относительно предыдущего, время увеличивается примерно 4 раза.

Теперь аналогично оценим асимптотическую сложность матричного умножения:

Matrix<F> C(lhs.get\_width(), rhs.get\_length());

F S;

for (int i = 0; i < lhs.get\_width(); i++)

{

for (int j = 0; j < rhs.get\_length(); j++)

{

S = 0;

for (int k = 0; k < rhs.get\_width(); k++)

{

S = S + lhs.get\_array()[i][k] \* rhs.get\_array()[k][j];

}

C[i][j] = S;

}

}

Асимптотическая сложность умножения:

Теперь мы произведем замеры времени умножения квадратных матриц (чтобы точнее соотнести результаты замеров с асимптотической сложностью):



Рисунок 6: Замеры времени, полученные при выполнении умножения матриц (сек.).

На основе полученных данных, построим таблицу соотношения размеров матриц и времени умножения.

|  |  |
| --- | --- |
| Количество элементов в матрице | Время выполнения (сек.) |
| 1000 | 7.559 |
| 2000 | 124.24 |
| 4000 | 1224.47 |

Таблица 2: Результаты замеров времени умножения матриц.

Как мы можем заметить соотношение времени чуть выше кубического, но я связываю это с тем, что во время замеров, часть возможностей процесса была затрачена на другую компиляцию. Однако в любом случае результаты близки к ожидаемым.

Теперь аналогично оценим асимптотическую сложность векторно-матричного умножения:

Matrix<F> C(lhs.get\_width(), rhs.get\_length());

F S;

for (int i = 0; i < lhs.get\_width(); i++)

{

for (int j = 0; j < rhs.get\_length(); j++)

{

S = 0;

for (int k = 0; k < rhs.get\_width(); k++)

{

S = S + lhs.get\_array()[i][k] \* rhs.get\_array()[k][j];

}

C[i][j] = S;

}

}

Асимптотическая сложность умножения:

Стоит отметить, что важной для асимптотической сложности является только длина вектора (равная ширине матрицы).

Теперь мы произведем замеры времени умножения вектора на матрицу:

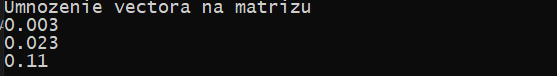


Рисунок 7: Замеры времени, полученные при выполнении умножения вектора на матрицу(сек.).

На основе полученных данных, построим таблицу соотношения размеров матриц и времени умножения:

|  |  |
| --- | --- |
| Количество элементов в матрице | Время выполнения (сек.) |
| 1000 | 0.003 |
| 2000 | 0.023 |
| 4000 | 0.11 |

Таблица 3: Результаты замеров времени умножения матриц.

С сортировками все значительнее проще, поскольку, для стандартных алгоритмов сортировки (BubbleSort, IsertionSort и QuickSort) Асимптотическая сложность уже определена. Поэтому просто произведём замеры и составим таблицу:



Рисунок 8: Замеры времени, полученные при выполнении сортировки вектора (сек.)



Рисунок 9: Замеры времени, полученные при выполнении сортировки вектора(сек.)



Рисунок 10: Замеры времени, полученные при выполнении сортировки вектора(сек.)

Внесем полученные данные в таблицу:

|  |  |  |  |
| --- | --- | --- | --- |
| Вид сортировки | Асимптотическая сложность | Количество элементов | Время (сек.) |
| BubbleSort |  | 100000 | 12.608 |
| 200000 | 50.296 |
| 400000 | 200.226 |
| InsertionSort |  | 100000 | 1.786 |
| 200000 | 6.981 |
| 400000 | 28.339 |
| QuickSort |  | 100000 | 0.012 |
| 200000 | 0.022 |
| 400000 | 0.042 |

Таблица 4: Результаты сортировок по времени для массивов

Как видно из таблицы, действительно, затрачиваемое на реализацию операции время растет эквивалентно асимптотической функции.

# 6. Заключение

По итогу проведения данной работы, нам удалось реализовать удобные объекты для работы с матрицами и векторами. Также по проведенным экспериментам можно сделать вывод, что с некоторыми погрешностями, затраченное на реализацию матричных, матрично-векторных операций и сортировок время, совпадает с ожидаемым относительно асимптотической сложности.

# 7. Литература

1. Павловская Т.А. C/C++, Программирование на языке высокого уровня, 2003.
2. <https://ru.wikipedia.org>
3. <https://ru.stackoverflow.com>
4. <https://learnc.info/c/memory_allocation.html>

# 8. Приложение