Федеральное государственное образовательное бюджетное учреждение
высшего профессионального образования
«Нижегородский Государственный Университет им.
Н.И.Лобачевского» (ННГУ)

Национальный исследовательский Университет
Институт Информационных Технологий Математики и Механики

Отчёт по лабораторной работе
Создание библиотеки с классами векторов и матриц

Выполнил:
студент группы 3821Б1ПМ3

Заботин М.А

Проверил:
заведующий лабораторией
суперкомпьютерных технологий и
высокопроизводительных вычислений

Лебедев И.Г

Нижний Новгород
2022 г.

Оглавление

1. Введение

Программирование - это интересный, полезный и увлекательный процесс, благодаря которому мы можем с помощью специальных команд обучать компьютер, делать для нас разнообразные полезные задачи, от выполнения операций с числами и навигации, до управления самолетами, спутниками и марсоходами.

Чтобы программировать сложные алгоритмы, необходимо постоянно пополнять свои знания о структурах и методах изучаемого языка программирования. Одной из таких сложных и интересных структур в языке C++ являются классы. Это по сути инструмент для создания новых типов переменных, наряду с int, float, bool и т.д. Классы используются, когда нам необходимо описать множество схожих объектов, например, животных в зоопарке, у каждого из которых есть вес, рост, количество особей. Но неэффективно описывать каждого объекта по отдельности, гораздо проще создать структуру, которая будет содержать в себе данные о каждом объекте в целом, а не по отдельности. Также классы облегчают работу с разными математическими объектами, так как могут содержать в себе различные функции (методы), присущие каждому из объектов, которые могут выполнить нужную задачу, а также набор разных значений, существующих у объекта (поля). Для объектов класса также можно выполнить перегрузку разных стандартных операций, по сути указать программе как нужно действовать с объектами класса, что значительно упростит работу с ними. Также, если не понятно, с каким типом данных внутри класса придется работать используют шаблоны, вместо того чтобы кодировать много одинаковых функций, различающихся только типом данных. Шаблоны позволяют в процессе работы использовать нужный тип данных в работе класса.

В данной лабораторной работе для изучения особенностей работы с классами, была поставлена задача: на языке «C++», используя шаблонные классы, а также перегрузки операций, необходимо написать программу, исходные файлы которой вынесены в статистическую библиотеку, для последующей удобной работы с алгебраическими векторами N-мерного пространства и матрицами.

2. Постановка задачи

Используя шаблонные классы и перегрузки операций, написать программу, которая позволяет удобно работать с векторами N-мерного пространства и матрицами, а именно, складывать, вычитать, умножать вектора и матрицы между собой, а также по отдельности на скаляры. Исходные файлы, содержащие описание классов векторов и матриц должны быть вынесены в отдельную статистическую библиотеку, для последующей удобной работы с ними. Реализовать потоковые ввод и вывод для каждого класса, а также доступ к защищённым полям.

Программа должна быть написана на языке «C++». Класс матрицы должен быть наследником класса вектора.

3. Руководство пользователя

Как таковой интерфейс работы с программой не реализован, так как задача была реализовать классы, для удобной работы с векторами и матрицами, но в исходном файле «main.cpp» написаны функции с тестами различных операций с векторами и матрицами. Функции закоментарены, и, чтобы испытать, как работает тот или иной метод класса нужно убрать комментарии с нужной тестовой функции (См. Рис.1)

```
//testmultvectmatrix();
//testinput();
//testmatrixoutput();
//testmult();
//testmultomatrix();
//testmultvector();
//testsum();
//testravno();
testcountinclude();
}
```

Рисунок 1. Список тестовых функций.

Но так как исходные файлы, содержащие описания классов вынесены в статистическую библиотеку «vectorLib.lib», написанный инструмент может быть использован в работе других программ разных людей, если понадобится работать с векторами или матрицами. Единственное что нужно в таком случае, это подключить данную библиотеку в своей программе и удобно работать с векторами и матрицами. В таком случае считаю необходимым описать методы и конструкторы классов векторов и матриц, чтобы дальнейшие пользователи знали, как работает тот или иной метод, а также перегрузка операции.

3.1 Класс векторов

Чтобы подключить библиотеку с вектором, необходимо в начале программы написать «`#include "Vector.h"`», после чего пользоваться инструментарием данного класса.

Создать вектор можно тремя способами, тип данных, который будет использоваться вектором указывается в треугольных скобках:

`TVector(int n, T v)`
Пример:
`TVector<int> V(10,-3)`
	Конструктор, создающий вектор из n элементов, в каждой ячейке которого будет лежать значение «v»

`TVector(int length = 1,T* mas = nullptr)`
Пример:
`int*data[3]={1,3,4}`
`TVector<int> V(3,data)`
	Конструктор, создающий вектор, с помощью массива

`TVector(const TVector<T>& p)`
Пример:
`TVector<int> V(V1)`
	Конструктор копирования, принимающий на вход другой вектор

Таблица 1.

Далее будет описание методов и перегруженных операций. После перегрузки оператор понимает, как поступать с новыми объектами, если для них вызывается данная операция, поэтому всё что остается пользователю, это пользоваться нужными операциями, как и при обычном их использовании, допустим с целочисленными типами данных:

Метод	Описание	Что возвращает
`GetLen()`	Возвращает длину вектора	int
`SetTran(bool t)`	Изменяет значение транспонированности вектора, на вход подается новое значение	void
`GetTran()`	Возвращает значение транспонированности вектора	bool
`CountIncludeVector(T ch)`	Возвращает количество вхождений заданного значения в вектор, на вход подается необходимый для подсчёта элемент	int
`T& operator[](int i)`	Перегрузка оператора доступа к ячейке вектора, возвращает ссылку на шаблонный тип	T&
`TVector<T>& operator = (const TVector<T>& p)`	Перегрузка оператора присваивания, которая переопределяет вектор и присваивает ему нужное значение	Ссылка на Вектор
`TVector<T> operator + (const TVector<T>& p)`	Перегрузка оператора «+», при которой происходит покоординатное сложение (если длины векторов разные то программа выдаст исключение)	Вектор
`TVector<T> operator - (const TVector<T>& p)`	Перегрузка оператора «-», при которой происходит покоординатное вычитание (если длины векторов разные то программа выдаст исключение)	Вектор
`TVector<T> operator / (const TVector<T>& p)`	Перегрузка оператора «/», при которой происходит покоординатное деление (если длины векторов разные то программа выдаст исключение)	Вектор
`TVector<T> operator * (const T n)`	Перегрузка оператора «*», при которой на вход подается шаблонный элемент и на него умножается каждый элемент вектора	Вектор
`TVector<T> operator / (const T n)`	Перегрузка оператора «/», при которой на вход подается шаблонный элемент и на него делится каждый элемент вектора	Вектор
`bool operator == (const TVector<T>& p)`	Оператор сравнивания (если длины векторов разные то возвращается ложь), далее происходит покоординатное сравнивание, и	bool

	если координаты равны, то возвращается истина, если нет, то ложь	
`TVector<T> operator * (const TVector<T>& p)`	Перегрузка оператора умножения вектора на вектор. Если оба вектора транспонированные, то происходит покоординатное умножение векторов, и возвращается соответствующий вектор произведения. Если же только второй транспонирован, то возвращается единичный вектор, в ячейку которого находится результат умножения векторов строка на столбец.	Вектор
`ostream& operator<<(ostream& ostr, TVector<T>& p)`	Перегрузка оператора потокового вывода, красиво выводит вектор на консоль	Ссылку на объект ostream
`istream& operator>>(istream& istr, TVector<T>& p)`	Перегрузка оператора потокового ввода, сначала с клавиатуры надо ввести размер вектора, после чего ввести каждую координату вектора	Ссылку на объект istream

Таблица 2.

3.2 Класс матриц

Чтобы подключить библиотеку с матрицами, необходимо в начале программы написать «`#include "Matrix.h"`», после чего пользоваться инструментарием данного класса.

Создать матрицу можно двумя способами, тип данных, который будет использоваться матрицей, указывается в треугольных скобках:

`TMatrix(int numrow, int numcol,A ch=0)`
Пример:
`TMatrix<int> M(5,4,-3)`
	Конструктор, принимающий на вход количество строк и столбцов в матрице, а также заполняющий её символами или нулями, если символа на вход не пришло

`TMatrix()`
Пример:
`TMatrix<int> M;`
	Конструктор по умолчанию, создающий матрицу размером 1 на 1 и заполняющий ее ячейку нулём

Таблица 3.

Далее будет описание методов и перегруженных операций. После перегрузки оператор понимает, как поступать с новыми объектами, если для них вызывается данная операция, поэтому всё что остается пользователю, это пользоваться нужными операциями, как и при обычном их использовании, допустим с целочисленными типами данных:

Метод	Описание	Что возвращает
`GetLen2()`	Возвращает количество столбцов в матрице	int
`void MultVect(TVector<A>& p1, const TVector<A>& p2)`	Метод, являющийся дополнением для векторного умножения, в результате	void

	которого создаётся матрица и присваивается текущей	
`void MultVectMatrix(TVector<A>& p, TMatrix<A>& m)`	Метод, который выполняет матрично-векторное умножение и в зависимости от транспонированности вектора присваивает текущей матрице новое значение	void
`int CountIncludeMatrix(A ch)`	Возвращает количество вхождений заданного значения в матрицу, на вход подаётся необходимый для подсчёта элемент	int
`bool operator == (const TMatrix<A>& p)`	Оператор сравнивания (если размеры матриц разные то возвращается ложь), далее происходит покоординатное сравнивание, и если координаты равны, то возвращается истина, если нет, то ложь	bool
`TMatrix<A> operator + (const TMatrix<A>& p)`	Перегрузка оператора «+», при которой выполняется поэлементное сложение	Матрица
`TMatrix<A> operator - (const TMatrix<A>& p)`	Перегрузка оператора «-», при которой выполняется поэлементное вычитание	Матрица
`TMatrix<A> operator * (const TMatrix<A>& p)`	Оператор умножения матриц, который вначале сравнивает размеры матриц и если количество столбцов первой матрицы не совпадает с количеством строк второй, то выдаётся исключение. Если же размеры совпадают, то происходит умножение матриц строка на столбец и создаётся новая результирующая матрица	Матрица
`TMatrix<A> operator = (const TMatrix<A>& p)`	Перегрузка оператора присваивания, которая переопределяет матрицу и присваивает ей нужное значение	Матрица
`ostream& operator <<(ostream& ostr, TMatrix<A>& p)`	Перегрузка оператора потокового вывода, красиво выводит матрицу на консоль	Ссылка на объект ostream
`istream& operator >>(istream& istr, TMatrix<A>& p)`	Перегрузка оператора потокового ввода, сначала с клавиатуры надо ввести размер матрицы, то есть сначала количество строк, а потом столбцов, после чего ввести каждую ячейку матрицы, для того чтобы заполнить	Ссылка на объект istream

Таблица 4.

4. Руководство программиста

4.1 Описание структур данных

4.1.1 Классы

Класс векторов:

Этот шаблонный класс реализован в отдельном файле заголовка «Vector.h» и может быть использован в дальнейшем для удобной работы с векторами.

(См. Рис.2)

```
template<class T>
class TVector
{
protected:
  T* data;
  int len;
  bool tran;
public:
  TVector(int n, T v);
  TVector(const TVector<T>& p);
  TVector(int length = 1,T* mas = nullptr);
  ~TVector();

  int GetLen() const;
  void SetTran(bool t);
  bool GetTran() const;
  int CountIncludeVector(T ch);

  T& operator[](int i);
  TVector<T>& operator = (const TVector<T>& p);
  TVector<T> operator + (const TVector<T>& p);
  TVector<T> operator - (const TVector<T>& p);
  TVector<T> operator / (const TVector<T>& p);
  TVector<T> operator * (const T n);
  TVector<T> operator / (const T n);
  bool operator == (const TVector<T>& p);
  TVector<T> operator * (const TVector<T>& p);
};
```

Рисунок 2. Класс векторов.

Поля класса:

Поля класса находятся в режиме доступа «protected», что делает возможным дальнейшее использование их для наследников класса.

1) «data», массив шаблонного типа, который определяет тип данных в процессе работы, для разных ситуаций
2) «len», переменная типа «int», хранящая в себе данные о длине вектора, иными словами, размерность пространства
3) «tran», переменная типа «bool», которая хранит в себе данные о транспонированности вектора.

Методы класса:

Методы класса находятся в режиме доступа «public», что позволяет пользоваться ими при работе с объектами класса.

1) «TVector», это три конструктора, один из которых создаёт вектор заданной длины и заполняет его заданным символом (принимает на вход длину и шаблонный символ), также конструктор по умолчанию (который вызывается без параметров), создающий вектор единичной длины и конструктор копирования другого вектора (принимающий на вход ссылку на копируемый вектор), а также конструктор, создающий вектор с помощью массива и его длины.
2) «~TVector», деструктор, очищающий выделенную для массива память в конце работы программы.
3) «GetLen», функция, возвращающая длину вектора.
4) «SetTran», функция, изменяющая значение транспонированности вектора, принимающая на вход значение транспонированности.
5) «GetTran», функция, возвращающая значение транспонированности вектора.
6) «CountIncludeVector», функция, возвращающая количество вхождений какого-либо значения в данный вектор, принимающая на вход символ.
7) Также дальше в описании класса идут перегрузки операций, а именно ([],*,+,-,/,=,==), принимающие на вход константные ссылки на вектора, либо значение транспонированности типа «bool», либо же шаблонный элемент.
8) Вне описания методов класса реализованы перегрузки операторов потокового ввода и вывода, принимающие на вход ссылку на объект класса векторов и поток, и возвращающие поток. (См. Рис.3)

```cpp
template<class T>
ostream& operator<<(ostream& ostr, TVector<T>& p)
{
  for (int i = 0; i < p.GetLen(); i++)
    ostr << p[i] << "\t";
  return ostr;
}

template<class T>
istream& operator>>(istream& istr, TVector<T>& p)
{
  int length;
  istr >> length;
  TVector<T> res(length);
  for (int i = 0; i < length; i++)
  {
    istr >> res[i];
  }
  p = res;
  return istr;
}
```

Рисунок 3. Реализация перегрузок потокового ввода и вывода векторов.

Класс матриц:

Этот шаблонный класс реализован в отдельном файле заголовка «Matrix.h», он является наследником класса векторов, и является по сути вектором векторов. Он позволяет удобно работать с матрицами, а также дополняет векторное умножение. (См. Рис.4)

```cpp
template<class A>
class TMatrix: public TVector<TVector<A>>
{
protected:
  int len2;
public:
  TMatrix(int numrow, int numcol, A ch=0);
  TMatrix();

  int GetLen2();

  void MultVect(TVector<A>& p1, const TVector<A>& p2);
  void MultVectMatrix(TVector<A>& p, TMatrix<A>& m);
  int CountIncludeMatrix(A ch);

  bool operator == (const TMatrix<A>& p);
  TMatrix<A> operator + (const TMatrix<A>& p);
  TMatrix<A> operator - (const TMatrix<A>& p);
  TMatrix<A> operator * (const TMatrix<A>& p);
  TMatrix<A> operator = (const TMatrix<A>& p);
};
```

Рисунок 4. Класс Матриц.

Поля класса:

Поля класса находятся в режиме доступа «protected», что делает возможным дальнейшее использование их для наследников класса.

1) «len2», хранит в себе целочисленное значение, а именно вторую размерность матрицы, количество столбцов.

Так как матрица является наследником вектора, можно представить её, как вектор, каждой ячейкой которого будет являться ещё один вектор, поэтому единственным полем данного класса будет являться значение количества столбцов, либо говоря по-другому, количество ячеек в каждом векторе вектора.

Методы класса:

Методы класса находятся в режиме доступа «public», что позволяет пользоваться ими при работе с объектами класса.

1) «TMatrix», это 2 конструктора, один по умолчанию, не принимающий на вход никаких параметров, а другой инициализатор, который на вход принимает размерности матрицы и символ, которым необходимо заполнить ячейки матрицы, по умолчанию символ равен нулю. Оба конструктора отвечают за создание матрицы и выделение памяти.
2) «GetLen2», возвращает значение количества столбцов в матрице.
3) «MultVect», функция, принимающая на вход 2 вектора и возвращающая матрицу, полученную в результате умножения транспонированного вектора на не транспонированный.
4) «MultVectMatrix», принимающая на вход вектор и матрицу и возвращающая в зависимости от транспонированности вектора, вектор или матрицу, полученные в результате умножения.
5) «CountIncludeMatrix», принимающая на вход шаблонный элемент и возвращающая количество его вхождений в матрицу.
6) Также дальше в описании класса идут перегрузки операций, а именно (*,+,-,=,==), принимающие на вход константные ссылки на вектора.
7) Вне описания методов класса реализованы перегрузки операторов потокового ввода и вывода, принимающие на вход ссылку на объект класса матриц и поток, и возвращающие поток. (См. Рис.5)

```
template<class A>
ostream& operator <<(ostream& ostr, TMatrix<A>& p)
{
  for (int i = 0; i < p.GetLen(); i++)
  {
    auto v = p[i];
    for (int j = 0; j < p.GetLen2(); j++)
    {
      ostr << p[i][j] << "\t";
    }
    ostr<< endl;
  }
  return ostr;
}

template<class A>
istream& operator >>(istream& istr, TMatrix<A>& p)
{
  int numrow;
  int numcol;
  istr >> numrow >> numcol;
  TMatrix<A> res(numrow, numcol);
  for (int row = 0; row < numrow; row++)
  {
    for (int col = 0; col < numcol; col++)
    {
      istr >> res[row][col];
    }
  }
  p = res;
  return istr;
}
```

Рисунок 5. Реализация перегрузок потокового ввода и вывода матриц.

4.1.2 Библиотеки

В данной программе используется несколько библиотек:

1) «iostream», основная библиотека, включающая в себя элементы работы с потоковым вводом и выводом, а также содержащая базовые элементы.
2) «locale.h», позволяющая поставить русскую локализацию, для удобной работы с русскими буквами.
3) «sstream», расширенная библиотека для работы с потоковыми данными, позволяющая создать переменную потокового типа, в которую записывать данные, чтобы проводить тесты.
4) «string», библиотека, позволяющая работать со строками.
5) Также классы вектора и матрицы реализованы в отдельных заголовочных файлах, «Vector.h» и «Matrix.h», которые подключаются в другие файлы как библиотеки, чтобы работать с элементами созданных классов.

6) «time.h» и «stdlib.h» используются для подсчета времени работы алгоритма для эксперимента

4.1.3 Функции

Про функции, являющиеся методами реализованных классов было рассказано ранее, а в данном абзаце речь пойдёт про тестовые функции, которые соответственно проводят тесты разных методов классов и перегрузок операторов.

1) «testcountinclude», тестовая функция проверяющая правильность работы поиска количества вхождений элемента в массив или вектор
2) «testmultvectmatrix», тестовая функция, проверяющая правильность работы матрично-векторного умножения.
3) «testinput», проверка ввода и вывода на консоль вектора.
4) «testmatrixoutput», проверка вывода на экран матрицы.
5) «testmult», тестовая функция проверяющая правильность работы умножения векторов.
6) «testmultomatrix», тестовая функция, проверяющая правильность умножения матриц.
7) «testmultvector», проверка правильности работы умножения векторов, в результате которого получается матрица.
8) «testsum», проверка правильности работы суммы векторов.
9) «testravno», тестовая функция, проверяющая правильность сравнения векторов.
10) «testtime», функция, позволяющая измерить время работы программы в миллисекундах

4.1.4 Типы данных

В данной программе используются как локальные переменные, существующие только внутри функций, предназначенные для каких-то локальных вычислений и работы, так и глобальные, которые распространяются на область программы. Но все переменные относятся к одному из типов:

1) «TVector<T>», образующая объект, вектор с каким-то шаблоном «T», который будет использоваться в типе данных самого вектора.
2) «TMatrix<A>», образующая объект, матрицу с каким-то шаблоном «A», который будет использоваться в типе данных самой матрицы.
3) «int», создающий переменную целочисленного типа, которая может в дальнейшем понадобиться в расчётах.
4) «bool», логический тип данных, имеющий только 2 значения, истина или ложь.
5) «stringstream», универсальный потоковый тип данных, который можно использовать и для ввода, и для вывода.
6) «double», используемый для создания вещественного типа данных.
7) «T» и «A», шаблонные типы данных, которые подразумевают под собой любой другой тип данных, и используемые для того, чтобы не описывать каждый метод класса сразу для всех типов данных, а описать один раз для шаблона, который сам везде поменяет тип данных.
8) «ostream», тип данных, для работы с потоковым выводом.
9) «istream», тип данных для работы с потоковым вводом.
10) «clock_t», тип, используемый для подсчета времени работы алгоритма.

4.2 Описание структуры программы

4.2.1 Описание структуры заголовочных файлов

Файл «Vector.h»:

Данный заголовочный файл содержит в себе реализацию класса векторов, вместе с реализацией его методов. Реализация методов происходит в сразу в заголовочном файле из-за особенностей компиляции шаблонных классов, при реализации которых в исходном файле могут возникнуть проблемы.

Сначала идет подключение необходимых для работы библиотек и использование стандартного пространства имен.

```
#pragma once
#include <iostream>
#include <string>
using namespace std;
```

Фрагмент кода 1.

После чего идет объявление шаблонного класса векторов и заполнение его защищенных и общедоступных областей полями и методами, конструкторами, перегрузками операций.

```
template<class T>
class TVector
{
protected:
  T* data;
  int len;
  bool tran;
public:
  TVector(int n, T v);
  TVector(const TVector<T>& p);
  TVector(int length = 1,T* mas = nullptr);
  ~TVector();

  int GetLen() const;
  void SetTran(bool t);
  bool GetTran() const;
  int CountIncludeVector(T ch);

  T& operator[](int i);
  TVector<T>& operator = (const TVector<T>& p);
  TVector<T> operator + (const TVector<T>& p);
  TVector<T> operator - (const TVector<T>& p);
  TVector<T> operator / (const TVector<T>& p);
  TVector<T> operator * (const T n);
  TVector<T> operator / (const T n);
  bool operator == (const TVector<T>& p);
  TVector<T> operator * (const TVector<T>& p);

};
```

Фрагмент кода 2.

За этим начинается реализация методов класса, перед каждым методом при помощи комментариев будет указано, за что отвечает этот метод. Перегрузки операторов потокового ввода и вывода не являются методами класса, так как работают с потоком и являются внешними, поэтому для доступа к защищенным полям класса используются соответствующие методы класса.

```
//Конструктор инициализатор, принимающий на вход длину и символ, которым
необходимо заполнить вектор
template<class T>
```

```
inline TVector<T>::TVector(int n, T v)
{
  if (n > 0)
  {
    data = new T[n];
    len = n;
    tran = false;
    for (int i = 0; i < n; i++)
      data[i] = v;
  }
  else
  {
    throw out_of_range("out of range");
  }
}

//Конструктор копирования, принимающий на вход ссылку на вектор
template<class T>
inline TVector<T>::TVector(const TVector<T>& p)
{
  data = new T[p.len];
  len = p.len;
  tran = p.tran;
  for (int i = 0; i < p.len; i++)
    data[i] = p.data[i];
}

//Конструктор, собирающий вектор по длине и переданному массиву шаблонного типа
template<class T>
inline TVector<T>::TVector(int length,T* mas)
{
  if (length <= 0)
  {
    throw out_of_range("out of range");
  }
  len = length;
  tran = false;
  data = new T[len];
  if (mas != nullptr)
  {
    for (int i = 0; i < len; i++)
      data[i] = mas[i];
  }
}

//Деструктор, очищающий выделенную память
template<class T>
inline TVector<T>::~TVector()
{
  if (data != nullptr)
  {
    delete[] data;
    data = nullptr;
  }
}

//Метод, возвращающий длину вектора
template<class T>
inline int TVector<T>::GetLen() const
{
  return len;
}

//Метод, изменяющий значение транспонированности вектора
```

```
template<class T>
inline void TVector<T>::SetTran(bool t)
{
  tran = t;
}

//Метод, возвращающий значение транспонированности вектора
template<class T>
inline bool TVector<T>::GetTran() const
{
  return tran;
}

//Метод для подсчёта количества вхождения элемента в вектор
template<class T>
inline int TVector<T>::CountIncludeVector(T ch)
{
  int count = 0;
  for (int i = 0; i < len; i++)
  {
    if (data[i] == ch)
    {
      count++;
    }
  }
  return count;
}

//Перегрузка оператора [], принимающая на вход номер ячейки вектора
template<class T>
inline T& TVector<T>::operator[](int i)
{
  if (i >= 0 && i < len)
  {
    return data[i];
  }
  else
  {
    stringstream info;
    info << "opreator [] i=" << i << " len=" << len;
    throw out_of_range(info.str());
  }
}

//Перегрузка оператора =, принимающая на вход ссылку на вектор
template<class T>
inline TVector<T>& TVector<T>::operator=(const TVector<T>& p)
{
  if (data != nullptr)
  {
    delete[] data;
    data = nullptr;
  }
  data = new T[p.len];
  len = p.len;
  tran = p.tran;
  for (int i = 0; i < p.len; i++)
    data[i] = p.data[i];
  return *this;
}

//Перегрузка оператора +, принимающая на вход ссылку на вектор
template<class T>
inline TVector<T> TVector<T>::operator+(const TVector<T>& p)
```

```
{
  TVector<T> res(len);
  if (len != p.len)
  {
    throw out_of_range("different sizes");
  }
  for (int i = 0; i < len; i++)
    res[i] = data[i] + p.data[i];
  return res;
}

//Перегрузка оператора -, принимающая на вход ссылку на вектор
template<class T>
inline TVector<T> TVector<T>::operator-(const TVector<T>& p)
{
  TVector<T> res(len);
  if (len != p.len)
  {
    throw out_of_range("different sizes");
  }
  for (int i = 0; i < len; i++)
    res[i] = data[i] - p.data[i];

  return res;
}

//Перегрузка оператора /, принимающая на вход ссылку на вектор
template <class T>
inline TVector<T> TVector<T>::operator /(const TVector<T>& p)
{
  TVector<T> res(len);
  if (len != p.len)
  {
    throw out_of_range("different sizes");
  }
  for (int i = 0; i < len; i++)
    res[i] = data[i] / p.data[i];

  return res;
}

//Перегрузка оператора *, принимающая на вход шаблонный скаляр
template<class T>
inline TVector<T> TVector<T>::operator*(const T n)
{
  TVector<T> res(len);

  for (int i = 0; i < len; i++)
    res[i] = data[i] * n;

  return res;
}

//Перегрузка оператора /, принимающая на вход шаблонный скаляр
template<class T>
inline TVector<T> TVector<T>::operator/(const T n)
{
  TVector<T> res(len);

  for (int i = 0; i < len; i++)
    res[i] = data[i] / n;

  return res;
}
```

```
//Перегрузка оператора ==, принимающая на вход ссылку на вектор
template<class T>
inline bool TVector<T>::operator==(const TVector<T>& p)
{
  if (len != p.len || tran != p.tran)
    return false;
  for (int i = 0; i < len; i++)
    if (!(data[i] == p.data[i]))
      return false;
  return true;
}

//Перегрузка оператора *, принимающая на вход ссылку на вектор
template<class T>
inline TVector<T> TVector<T>::operator*(const TVector<T>& p)
{
  if (len == p.len && tran == p.tran && tran == true)
  {
    TVector<T> res(len);
    for (int i = 0; i < len; i++)
      res[i] = data[i] * p.data[i];
    return res;
  }
  if (len == p.len && p.tran == true && tran == false)
  {
    TVector<T> res(1,0);
    for (int i = 0; i < len; i++)
      res[0] = res[0] + (data[i] * p.data[i]);
    return res;
   }
}

//Перегрузка оператора <<, принимающая на вход поток и ссылку на вектор
template<class T>
ostream& operator<<(ostream& ostr, TVector<T>& p)
{
  for (int i = 0; i < p.GetLen(); i++)
    ostr << p[i] << "\t";
  return ostr;
}
//Перегрузка оператора >>, принимающая на вход поток и ссылку на вектор
template<class T>
istream& operator>>(istream& istr, TVector<T>& p)
{
  int length;
  istr >> length;
  TVector<T> res(length);
  for (int i = 0; i < length; i++)
  {
    istr >> res[i];
  }
  p = res;
  return istr;

}
```

Фрагмент кода 3.

Файл «Matrix.h»:

Данный заголовочный файл содержит в себе реализацию класса матриц, вместе с реализацией его методов. Реализация методов происходит в сразу в заголовочном файле из-за особенностей компиляции шаблонных классов, при реализации которых в исходном файле могут возникнуть проблемы.

Сначала идет подключение необходимых для работы библиотек и использование стандартного пространства имен. Так как в файле вектора уже подключены все необходимые библиотеки и пространство имён, нам всего лишь необходимо подключить файл заголовка вектора.

```
#pragma once
#include "Vector.h"
```

Фрагмент кода 4.

После чего идет объявление шаблонного класса матриц, который является наследником вектора и по сути является вектором, состоящим из векторов, и заполнение его защищенных и общедоступных областей полями и методами, конструкторами, перегрузками операций.

```
template<class A>
class TMatrix: public TVector<TVector<A>>
{
protected:
  int len2;
public:
  TMatrix(int numrow, int numcol, A ch=0);
  TMatrix();

  int GetLen2();

  void MultVect(TVector<A>& p1, const TVector<A>& p2);
  void MultVectMatrix(TVector<A>& p, TMatrix<A>& m);
  int CountIncludeMatrix(A ch);

  bool operator == (const TMatrix<A>& p);
  TMatrix<A> operator + (const TMatrix<A>& p);
  TMatrix<A> operator - (const TMatrix<A>& p);
  TMatrix<A> operator * (const TMatrix<A>& p);
  TMatrix<A> operator = (const TMatrix<A>& p);

};
```

Фрагмент кода 5.

После объявления класса начинается реализация методов класса, перед каждым методом при помощи комментариев будет указано, за что отвечает этот метод. Перегрузки операторов потокового ввода и вывода не являются методами класса, так как работают с потоком и являются внешними, поэтому для доступа к защищенным полям класса используются соответствующие методы класса.

```
//Конструктор инициализатор, принимающий на вход количество строк и столбцов в
матрице
template<class A>
inline TMatrix<A>::TMatrix(int numrow, int numcol, A ch=0) : TVector<TVector<A>>
::TVector(numrow)
{
  len2 = numcol;
  for (int i = 0; i < numrow; i++)
  {
    TVector<A> a(numcol);
    data[i] = a;
    for (int j = 0; j < numcol; j++)
    {
      data[i][j] = 0;
    }
  }
}
```

```
//Конструктор по умолчанию, создающий матрицу 1 на 1 и заполняющий её нулём
template<class A>
inline TMatrix<A>::TMatrix():TMatrix<A> (1,1)
{
}

//Метод, возвращающий количество столбцов в матрице
template<class A>
inline int TMatrix<A>::GetLen2()
{
  return len2;
}

//Метод, выполняющий векторное умножение, в результате которого получается матрица
template<class A>
inline void TMatrix<A>::MultVect(TVector<A>& p1, const TVector<A>& p2)
{
  if ((!p1.GetTran()) || p2.GetTran())
  {
    throw out_of_range("wrong transponation");
  }
  TMatrix<A> M1(p1.GetLen(), 1);
  for (int row = 0; row < p1.GetLen(); row++)
  {
    M1[row][0] = p1[row];
  }

  TMatrix<A> M2(1, p2.GetLen());
  M2[0] = p2;

  *this = M1*M2;
}

//Метод, выполняющий матрично векторные умножения
template<class A>
inline void TMatrix<A>::MultVectMatrix(TVector<A>& p, TMatrix<A>& m)
{
  if (p.GetTran())
  {
    if (p.GetLen() != m.GetLen2())
    {
      throw out_of_range("diferent sizes");
    }
    TMatrix<A> M1(p.GetLen(), 1);
    for (int row = 0; row < p.GetLen(); row++)
    {
      M1[row][0] = p[row];
    }
    *this = m * M1;
  }
  else
  {
    if (p.GetLen() != m.GetLen())
    {
      throw out_of_range("diferent sizes");
    }
    TMatrix<A> M2(1, p.GetLen());
    M2[0] = p;
    *this = M2 * m;
  }
}

//Метод, ищущий количество вхождений заданного элемента в матрицу
template<class A>
```

```
inline int TMatrix<A>::CountIncludeMatrix(A ch)
{
  int count=0;
  for (int i = 0; i < len; i++)
  {
    for (int j = 0; j < len2; j++)
    {
      if (data[i][j] == ch)
      {
        count++;
      }
    }
  }
  return count;
}

//Перегрузка оператора ==, принимающая на вход ссылку на матрицу
template<class A>
inline bool TMatrix<A>::operator==(const TMatrix<A>& p)
{
  return TVector<TVector<A>>::operator==(p);
}

//Перегрузка оператора +, принимающая на вход ссылку на матрицу
template<class A>
inline TMatrix<A> TMatrix<A>::operator+(const TMatrix<A>& p)
{
  if (len == p.len && len2 == p.len2)
  {
    TMatrix<A> res(len, len2);
    for (int i = 0; i < len; i++)
    {
      for (int j = 0; j < len2; j++)
      {
        res[i][j] = data[i][j] + p.data[i][j];
      }
    }
    return res;
  }
  else
  {
    throw out_of_range("different sizes");
  }

}

//Перегрузка оператора -, принимающая на вход ссылку на матрицу
template<class A>
inline TMatrix<A> TMatrix<A>::operator-(const TMatrix<A>& p)
{
  if (len == p.len && len2 == p.len2)
  {
    TMatrix<A> res(len, len2);
    for (int i = 0; i < len; i++)
    {
      for (int j = 0; j < len2; j++)
      {
        res[i][j] = data[i][j] - p.data[i][j];
      }
    }
    return res;
  }
  else
  {
```

```
    throw out_of_range("different sizes");
  }

}

//Перегрузка оператора *, принимающая на вход ссылку на матрицу
template<class A>
inline TMatrix<A> TMatrix<A>::operator*(const TMatrix<A>& p)
{
  if (len2 == p.len)
  {
    TMatrix<A> res(len, p.len2);
    for (int i = 0; i < len; i++)
    {
      for (int j = 0; j < p.len2; j++)
      {
        res[i][j] = 0;
        for (int t = 0; t < len2; t++)
        {
          res[i][j] += data[i][t] * p.data[t][j];
        }
      }
    }
    return res;
  }
  else
  {
    throw out_of_range("different sizes*");
  }
}

//Перегрузка оператора =, принимающая на вход ссылку на матрицу
template<class A>
inline TMatrix<A> TMatrix<A>::operator=(const TMatrix<A>& p)
{
  TVector<TVector<A>>::operator = (p);
  return *this;
}

//перегрузка оператора <<, принимающая на вход поток и ссылку на матрицу
template<class A>
ostream& operator <<(ostream& ostr, TMatrix<A>& p)
{
  for (int i = 0; i < p.GetLen(); i++)
  {
    auto v = p[i];
    for (int j = 0; j < p.GetLen2(); j++)
    {
      ostr << p[i][j] << "\t";
    }
    ostr<< endl;
  }
  return ostr;
}

//перегрузка оператора >>, принимающая на вход поток и ссылку на матрицу
template<class A>
istream& operator >>(istream& istr, TMatrix<A>& p)
{
  int numrow;
  int numcol;
  istr >> numrow >> numcol;
  TMatrix<A> res(numrow, numcol);
  for (int row = 0; row < numrow; row++)
```

```
    {
      for (int col = 0; col < numcol; col++)
      {
        istr >> res[row][col];
      }
    }
    p = res;
    return istr;
}
```

Фрагмент кода 6.

4.2.2 Описание структуры исходных файлов

Файл «Vector.cpp»:

Так как все методы класса вектора реализованы в соответственном заголовочном файле, в исходном файле вектора нету никаких реализаций методов класса и перегрузок операторов. Данный исходный файл используется только для того, чтобы программа скомпилировалась, ведь заголовочные файлы сами по себе не компилируются, но если подключить заголовочный файл в исходном, то такая связка пройдет компиляцию успешно.

```
#include "Vector.h"
#include <iostream>
using namespace std;
```

Фрагмент кода 7.

Файл «Matrix.cpp»:

В этом случае ситуация такая же, как и с исходным файлом векторов, он нужен только для подключения файла заголовка и его компиляции.

```
#include <iostream>
#include "Matrix.h"
using namespace std;
```

Фрагмент кода 8.

Файл «main.cpp»:

Это основной исходный файл, в котором представлены тестовые функции, для проверки работы разных методов как класса векторов, так и матриц. Сначала идёт подключение всех необходимых библиотек, для работы программы, а также заголовочных файлов векторов и матриц, и использование стандартного пространства имён.

```
#include <iostream>
#include <locale.h>
#include <sstream>
#include <stdlib.h>
#include <time.h>
#include "Vector.h"
#include "Matrix.h"
using namespace std;
```

Фрагмент кода 9.

Далее в файле идет реализация тестовых функций, при подключении которых на консоли выводится тест, позволяющий сравнить данные, полученные программой с теоретическими вычислениями, для убеждения в правильности работы.

```
// Проверка суммы векторов
void testsum()
{
```

```
  TVector<double> B(10, 2.0);
  TVector<double> C(10, 6.0);
  C[2] = 4;
  TVector<double> D(10);
  D = C + B;
  cout << "D=" << D;
}

//Тест умножения векторов
void testmult()
{
  double data[3] = { 1,2,3 };
  TVector<double> B(3, data);
  TVector<double> D(3);

  D = B * 2;
  cout << "D=" << D;
}

//Тест сравнения двух векторов
void testravno()
{
  double data1[3] = { 1,2,3 };
  TVector<double> B1(3, data1);

  double data2[3] = { 1,2,3 };
  TVector<double> B2(3, data2);

  double data3[2] = { 1,2};
  TVector<double> B3(2, data3);

  cout << (B1 == B2);
}

//Тест ввода вектора с клавиатуры и вывода его на консоль
void testinput()
{
  TVector<double> B1;
  cin >> B1;
  cout << "B1=" << B1;
}

//Тест вывода матрицы на консоль
void testmatrixoutput()
{
  TMatrix<int> M(2, 3);
  cout << M;
}

//Тест векторного умножения, в результате которого получается матрица
void testmultvector()
{
  double data1[2] = { 1,2 };
  TVector<double> B1(2, data1);

  double data2[3] = { 1,2,3 };
  TVector<double> B2(3, data2);

  B1.SetTran(true);
  TMatrix<double> M1(2,3);

  M1.MultVect(B1,B2);

  cout << M1;
```

```
}

//Тест матричного умножения
void testmultomatrix()
{
  TMatrix<int> M1(2, 2);
  TMatrix<int> M2(2, 3);
  stringstream str;

  str << "2 2 2 3 4 5 ";
  str >> M1;
  cout << M1;

  str << "2 3 1 3 6 5 2 4 ";
  str >> M2;
  cout << M2;

  TMatrix<int> M3(2, 3);
  M3 = M1 * M2;

  TMatrix<int> check(2, 3);
  str << "2 3 17 12 24 29 22 44 ";
  str >> check;

  cout << M3 << endl;

  if (check == M3)
  {
    cout << "ok";
  }
  else
  {
    cout << "ne ok(";
  }
}

//Тест матрично-векторного умножения
void testmultvectmatrix()
{
  TMatrix<int> M1(2,3);
  TVector<int> V1(2);

  stringstream str;
  str << "2 3 1 2 3 4 5 6 ";
  str >> M1;

  str << "3 1 2 3 ";
  str >> V1;
  V1.SetTran(true);

  TMatrix<int> M3(2,1);
  M3.MultVectMatrix(V1, M1);

  cout << M3;
}

//Тест нахождения числа вхождений заданного значения в вектор или матрицу
void testcountinclude()
{
  int count;
  TMatrix<int> V1(2,3);
  stringstream str;
  str << "2 3 2 2 1 3 2 3  ";
  str >> V1;
```

```
  count = V1.CountIncludeMatrix(2);
  cout << count;
}
//Проверка времени работы алгоритма
void testtime()
{
  clock_t t1, t2;
  int n = 125;
  TMatrix<int> A(n, n, 99);
  TMatrix<int> B(n, n, 99);

  t1 = clock();
  A * B;
  t2 = clock();

  cout << "Time:" << (t2 - t1)<< " ms";
}
```

Фрагмент кода 10.

После этого начинается основная функция «main», в которой сначала ставится русская локализация, для корректной работы с русскими символами, а после чего в блоке «try»-«catch», вызываются тестовые функции и если возникает исключение, то оно обрабатывается и выводится на консоль.

```
int main()
{
  setlocale(LC_ALL, "Rus");
  try
  {
    //testmultvectmatrix();
    //testinput();
    //testmatrixoutput();
    //testmult();
    //testmultomatrix();
    //testmultvector();
    //testsum();
    //testravno();
    testcountinclude();
  }
  catch (exception&ex)
  {
    cout << ex.what();
  }
  return 0;
}
```

Фрагмент кода 11.

4.3 Описание алгоритмов

Большинство алгоритмов данной программы работают достаточно просто, взять, например, конструкторы, которые выделяют память при создании переменных и заполняют ячейки вектора или матрицы соответствующими значениями, которые необходимы пользователю. Или деструктор, который очищает выделенную память, чтобы избежать переполнения памяти. Алгоритм доступа к защищенным полям очень примитивен и просто либо возвращает значение защищенного поля, либо изменяет его. Также практически все алгоритмы при работе с векторами являются простыми и интуитивно понятными, например, умножение на скаляр просто умножает каждую ячейку вектора, сложение и вычитание векторов возвращает новый вектор с координатами, являющимися суммой координат, присваивание выделяет для вектора новую память и заполняет значениями, которые необходимо было присвоить. Либо потоковые операции, которые красиво распечатывают вектор на консоли, либо наоборот считывают длину и данные для вектора и заполняют его.

Самым интересным алгоритмом для вектора является векторное умножение, для которого есть несколько исходов:

1) Если оба вектора транспонированные, то новые координаты результирующего вектора будут являться произведением соответствующих координат
2) Если первый вектор транспонирован, а второй нет, то в результате получится матрица, в ячейках которой будут лежать всевозможные произведения элементов векторов
3) Если первый вектор не транспонирован, а второй транспонирован, то результатом будет число, скаляр.
4) Если же оба вектора не транспонированные, то алгоритм работать не будет.

Но больше всего интересны алгоритмы, работающие с матрицами. Конструкторы и деструкторы конечно, также просты в реализации и просто выделяют память для вектора векторов или очищают ее. Доступ к защищенному полю так же примитивен и возвращает количество строк или столбцов в матрице. Оператор сравнения и присваивания вынесены через представление матрицы, как вектора векторов и вызовом функций перегрузки векторов, что является полиморфизмом. Наибольший интерес вызывают алгоритмы суммирования и перемножения матриц между собой. Они основаны на математических правилах сложения и умножения матриц. И если сложение матриц не совсем интересный алгоритм, ведь по сути значения в каждых одинаковых ячейках складываются и переносятся в соответствующую ячейку новой результирующей матрицы (но сначала проверяются размерности, ведь нельзя складывать матрицы, если количества столбцов и строк не совпадают), то умножение матрицы очень интересный алгоритм. Сложность для сложения двух матриц n*n оценивается, как $O(n^2)$.

Умножение матриц на практике выполняется по формуле $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$, и является более трудоемким процессом. Главное при умножении матриц, чтобы оно получилось необходимо, чтобы количество столбцов первой матрицы совпадало с количеством строк второй матрицы, так как матрицы умножаются строка на столбец, каждые ячейки этих столбцов перемножаются и складываются в общую сумму, которую потом записывают в ячейку результирующей матрицы, а сама ячейка определяется какими по счёту строка и столбец умножаются. Если исходные матрицы были размеров m*n и n*p, то результирующая будет размером m*p. Для нахождения всех элементов которой понадобится m*p раз вычислений суммы перемножения строки на столбец. Алгоритм

исполняется при помощи трех вложенных циклов и для квадратной матрицы n*n сложность выполнения оценивается теоретически как $O(n^3)$.

Сейчас будет продемонстрирован самый интересный и трудоемкий из всех алгоритмов в данной лабораторной работе, а именно алгоритм умножения матриц:

1) В начале идет перегрузка операции умножения для матриц, возвращает данная перегрузка матрицу, а на вход принимает ссылку на матрицу.

```
template<class A>
inline TMatrix<A> TMatrix<A>::operator*(const TMatrix<A>& p)
{
```

Фрагмент кода 12.

2) После чего идет проверка на то, что количество столбцов первой матрицы равно количеству строк второй матрицы, если условие выполнено, то алгоритм продолжает работу, если же нет, то прекращает работу, выдавая исключение с указанием на проблему: несовместимые размеры матриц для умножения.

```
if (len2 == p.len)
{
}
else
{
    throw out_of_range("different sizes*");
}
```

Фрагмент кода 13.

3) Если размеры совпали, то сначала создаётся локальная результирующая матрица, которая впоследствии будет возвращена, как результат работы умножения, потом запускаются 2 вложенных цикла, которые переберут все ячейки новой матрицы и на каждом шаге будут сначала обнулять данную ячейку, а потом высчитывать новое ее значение, как сумму произведений координат соответствующей строки и столбца матриц. После чего функция возвращает результирующую матрицу.

```
TMatrix<A> res(len, p.len2);
for (int i = 0; i < len; i++)
{
    for (int j = 0; j < p.len2; j++)
    {
        res[i][j] = 0;
        for (int t = 0; t < len2; t++)
        {
            res[i][j] += data[i][t] * p.data[t][j];
        }
    }
}
return res;
```

Фрагмент кода 14.

В этом и состоит самый интересный и трудоемкий алгоритм, представленный в данной лабораторной работе. Также хочется выделить ещё 2 интересных алгоритма, это векторное умножение, в результате которого получается матрица, а также алгоритм матрично-векторного умножения. В данных алгоритмах вектора представляются в виде матрицы, состоящей из одной строки, или одного столбца, в зависимости от транспонированности вектора, после чего вызывается алгоритм матричного умножения,

который делает оставшуюся работу и сам создает матрицу, необходимого размера и с заполнением ячеек нужными значениями. То есть алгоритм умножения матриц является основой для двух других алгоритмов, что делает его очень важным и интересным.

5. Эксперименты

Так как самыми затратными по времени исполнения алгоритмами являются матричные сложения и умножения, именно их время работы нужно сравнивать с теоретически предполагаемым. Чтобы измерить время работы программы потребуется соответствующая функция «testtime», которую сначала адаптируем под сложение, а потом под умножение. Также нам потребуется конструктор матрицы, который заполняет каждую ячейку фиксированным значением.

```
void testtime()
{
  clock_t t1, t2;
  int n = 1000;
  TMatrix<int> A(n, n, 99);
  TMatrix<int> B(n, n, 99);

  t1 = clock();
  A + B;
  t2 = clock();

  cout << "Time:" << (t2 - t1)<< " ms";

}
```

Фрагмент кода 15.

Сначала запустим этот тест и посмотрим, сколько же времени затратил данный алгоритм. (См. Рис.6)

```
Time:92 ms
```

Рисунок 6. Время работы алгоритма сложения, при матрице 1000*1000.

Сложение матриц оценивается как $O(n^2)$, так же, как и выделение памяти для создания временной матрицы. Теоретически время будет равно $T(n) \approx an^2$. Подставив наши данные, получаем, что $a \approx 9,11 \cdot 10^{-5}$. После этого можно провести несколько экспериментов и составить по ним таблицу с двумя графиками, практических и теоретических значений, после чего сравнить результаты.

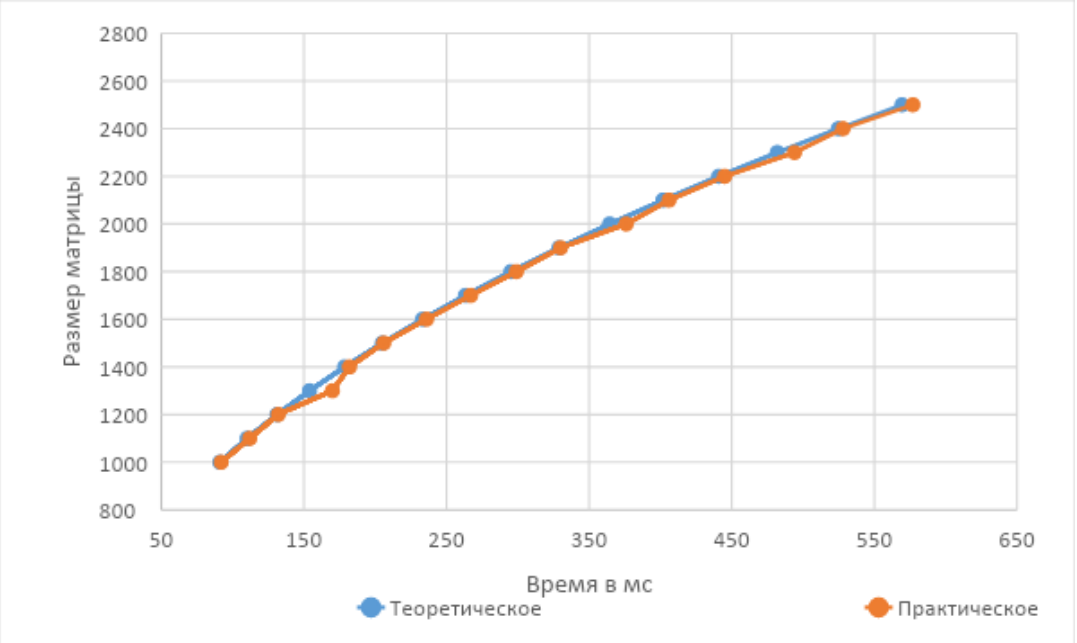

График 1.

Получив практические данные по времени для каждой размерности матриц, мы получили график и теперь можем сравнить. Можно видеть, что практические данные не сильно отличаются от теоретически предполагаемых, есть конечно несколько отклонений, но в целом все данные сходятся.

Далее проверим алгоритм умножения матриц, для этого всего лишь необходимо операцию сложения заменить на умножение. На этот раз диапазон размеров матрицы будет меньше и начинаться от 100. Сам алгоритм умножения работает за an^3 однако, помимо умножения в алгоритме происходит выделение памяти для n^2 элементов, требующее bn^2 времени. Итого получим время работы $T(n) \approx an^3 + bn^2$. Коэффициенты a и b несложно найти из имеющихся данных ($a \approx 7,6 \bullet 10^{-5}$, $b \approx -3,6 \bullet 10^{-4}$) . После этого необходимо провести серию экспериментов, также, как и в случае со сложением и занести данные в таблицу, после чего построить график и анализировать данные.

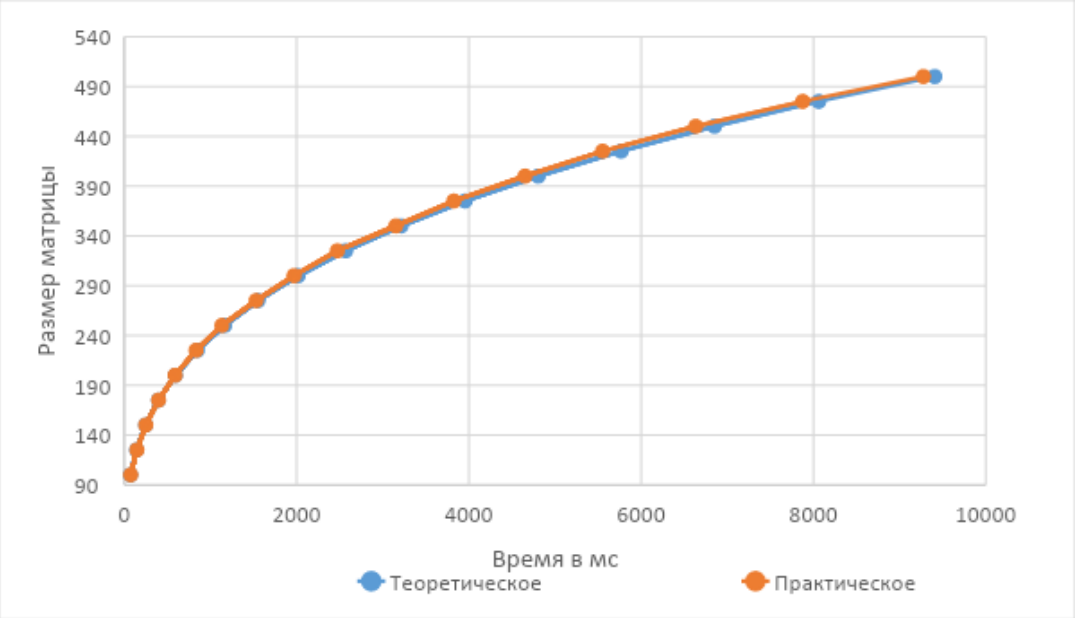

Таблица 2.

Проанализировав данные, можно увидеть, что также, как и в случае со сложением матриц, практические данные не сильно отличаются от теоретически рассчитанных.

6. Заключение

В результате проведенной лабораторной работы, на языке программирования «C++» была написана программа, содержащая в себе описание и реализацию двух шаблонных классов, векторов и матриц, при чём класс матриц является наследником вектора и по сути матрица представляет собой вектор, состоящий из векторов. Эти классы полностью описывают весь необходимый для работы с векторами и матрицами функционал и позволяют упростить работу с ними в дальнейшем. Также, исходные и заголовочные файлы с классами векторов и матриц выведены в статистическую библиотеку, что упрощает дальнейшее использование этого полезного инструментария как мной, так и другими пользователями, всё что остаётся сделать, это подключать в дальнейших программах эту библиотеку при необходимости и пользоваться функциями, описанными для векторов или матриц.

Также было проведено исследование времени работы программы на самых трудоемких алгоритмах, а именно сложение и умножение матриц. Проведено сравнение теоретически предполагаемого времени работы этих алгоритмов, с полученными на практике для разных размеров матриц. В результате исследования было получено, что практическое время выполнения алгоритмов не сильно отличается от теоретически предложенного времени и практически с ним совпадает.

Шаблонные классы являются полезным и интересным объектом языка программирования «C++». Они позволяют создавать классы, то есть по сути новые переменные, со своими данными и функциями, не несколько раз под разные типы данных, а один раз, а дальнейшую работу выполнят шаблоны, которые поставят те типы данных, которые будут необходимы пользователю. Классы очень полезны, когда мы хотим создать некоторую структуру (как например в данной лабораторной работе вектора и матрицы), у которой есть сложные данные, и функции, необходимые для работы с этой структурой. Одними из таких функций являются перегрузки операторов, которые по сути дают понимание компилятору, как поступать при выполнении стандартных операций над элементами данных классов. В ходе работы были полностью изучены методы работы с шаблонными классами, перегрузками операторов и реализованы тесты на практике. К тому же было изучено создание статистической библиотеки с исходными и заголовочными файлами, содержащими реализацию классов векторов и матриц, упрощающее дальнейшее использование этого инструментария в дальнейшем как мной, так и другими людьми. В дальнейшем необходимо и дальше совершенствовать свои знания о языке программирования «C++», изучать более сложные и интересные алгоритмы и структуры, а также писать более новые, совершенные и полезные программы.

7. Литература

1. Т.А. Павловская Учебник по программированию на языках высокого уровня(C/C++) – Режим доступа: http://cph.phys.spbu.ru/documents/First/books/7.pdf

2. Бьерн Страуструп. Язык программирования C++ - Режим доступа: http://8361.ru/6sem/books/Straustrup-Yazyk_programmirovaniya_c.pdf

8. Приложения

8.1 Реализация класса вектора

```
#pragma once
#include <iostream>
#include <string>
using namespace std;

template<class T>
class TVector
{
protected:
  T* data;
  int len;
  bool tran;
public:
  TVector(int n, T v);
  TVector(const TVector<T>& p);
  TVector(int length = 1,T* mas = nullptr);
  ~TVector();

  int GetLen() const;
  void SetTran(bool t);
  bool GetTran() const;
  int CountIncludeVector(T ch);

  T& operator[](int i);
  TVector<T>& operator = (const TVector<T>& p);
  TVector<T> operator + (const TVector<T>& p);
  TVector<T> operator - (const TVector<T>& p);
  TVector<T> operator / (const TVector<T>& p);
  TVector<T> operator * (const T n);
  TVector<T> operator / (const T n);
  bool operator == (const TVector<T>& p);
  TVector<T> operator * (const TVector<T>& p);
};

template<class T>
inline TVector<T>::TVector(int n, T v)
{
  if (n > 0)
  {
    data = new T[n];
    len = n;
    tran = false;
    for (int i = 0; i < n; i++)
      data[i] = v;
  }
  else
  {
    throw out_of_range("out of range");
  }
}

template<class T>
inline TVector<T>::TVector(const TVector<T>& p)
{
  data = new T[p.len];
  len = p.len;
  tran = p.tran;
  for (int i = 0; i < p.len; i++)
    data[i] = p.data[i];
```

```
}

template<class T>
inline TVector<T>::TVector(int length,T* mas)
{
  if (length <= 0)
  {
    throw out_of_range("out of range");
  }
  len = length;
  tran = false;
  data = new T[len];
  if (mas != nullptr)
  {
    for (int i = 0; i < len; i++)
      data[i] = mas[i];
  }
}

template<class T>
inline TVector<T>::~TVector()
{
  if (data != nullptr)
  {
    delete[] data;
    data = nullptr;
  }
}

template<class T>
inline int TVector<T>::GetLen() const
{
  return len;
}

template<class T>
inline void TVector<T>::SetTran(bool t)
{
  tran = t;
}

template<class T>
inline bool TVector<T>::GetTran() const
{
  return tran;
}

template<class T>
inline int TVector<T>::CountIncludeVector(T ch)
{
  int count = 0;
  for (int i = 0; i < len; i++)
  {
    if (data[i] == ch)
    {
      count++;
    }
  }
  return count;
}

template<class T>
inline T& TVector<T>::operator[](int i)
{
```

```
  if (i >= 0 && i < len)
  {
    return data[i];
  }
  else
  {
    stringstream info;
    info << "opreator [] i=" << i << " len=" << len;
    throw out_of_range(info.str());
  }
}

template<class T>
inline TVector<T>& TVector<T>::operator=(const TVector<T>& p)
{
  if (data != nullptr)
  {
    delete[] data;
    data = nullptr;
  }
  data = new T[p.len];
  len = p.len;
  tran = p.tran;
  for (int i = 0; i < p.len; i++)
    data[i] = p.data[i];
  return *this;
}

template<class T>
inline TVector<T> TVector<T>::operator+(const TVector<T>& p)
{
  TVector<T> res(len);
  if (len != p.len)
  {
    throw out_of_range("different sizes");
  }
  for (int i = 0; i < len; i++)
    res[i] = data[i] + p.data[i];
  return res;
}

template<class T>
inline TVector<T> TVector<T>::operator-(const TVector<T>& p)
{
  TVector<T> res(len);
  if (len != p.len)
  {
    throw out_of_range("different sizes");
  }
  for (int i = 0; i < len; i++)
    res[i] = data[i] - p.data[i];

  return res;
}

template <class T>
inline TVector<T> TVector<T>::operator /(const TVector<T>& p)
{
  TVector<T> res(len);
  if (len != p.len)
  {
    throw out_of_range("different sizes");
  }
  for (int i = 0; i < len; i++)
```

```
    res[i] = data[i] / p.data[i];

  return res;
}

template<class T>
inline TVector<T> TVector<T>::operator*(const T n)
{
  TVector<T> res(len);

  for (int i = 0; i < len; i++)
    res[i] = data[i] * n;

  return res;
}

template<class T>
inline TVector<T> TVector<T>::operator/(const T n)
{
  TVector<T> res(len);

  for (int i = 0; i < len; i++)
    res[i] = data[i] / n;

  return res;
}

template<class T>
inline bool TVector<T>::operator==(const TVector<T>& p)
{
  if (len != p.len || tran != p.tran)
    return false;
  for (int i = 0; i < len; i++)
    if (!(data[i] == p.data[i]))
      return false;
  return true;
}

template<class T>
inline TVector<T> TVector<T>::operator*(const TVector<T>& p)
{
  if (len == p.len && tran == p.tran && tran == true)
  {
    TVector<T> res(len);
    for (int i = 0; i < len; i++)
      res[i] = data[i] * p.data[i];
    return res;
  }
  if (len == p.len && p.tran == true && tran == false)
  {
    TVector<T> res(1,0);
    for (int i = 0; i < len; i++)
      res[0] = res[0] + (data[i] * p.data[i]);
    return res;
  }
}

template<class T>
ostream& operator<<(ostream& ostr, TVector<T>& p)
{
  for (int i = 0; i < p.GetLen(); i++)
    ostr << p[i] << "\t";
  return ostr;
}
```

```
template<class T>
istream& operator>>(istream& istr, TVector<T>& p)
{
  int length;
  istr >> length;
  TVector<T> res(length);
  for (int i = 0; i < length; i++)
  {
    istr >> res[i];
  }
  p = res;
  return istr;
}
```

8.2 Реализация класса матриц

```
#pragma once
#include "Vector.h"

template<class A>
class TMatrix: public TVector<TVector<A>>
{
protected:
  int len2;
public:
  TMatrix(int numrow, int numcol, A ch=0);
  TMatrix();

  int GetLen2();

  void MultVect(TVector<A>& p1, const TVector<A>& p2);
  void MultVectMatrix(TVector<A>& p, TMatrix<A>& m);
  int CountIncludeMatrix(A ch);

  bool operator == (const TMatrix<A>& p);
  TMatrix<A> operator + (const TMatrix<A>& p);
  TMatrix<A> operator - (const TMatrix<A>& p);
  TMatrix<A> operator * (const TMatrix<A>& p);
  TMatrix<A> operator = (const TMatrix<A>& p);
};

template<class A>
inline TMatrix<A>::TMatrix(int numrow, int numcol) : TVector<TVector<A>>
::TVector(numrow)
{
  len2 = numcol;
  for (int i = 0; i < numrow; i++)
  {
    TVector<A> a(numcol);
    data[i] = a;
    for (int j = 0; j < numcol; j++)
    {
      data[i][j] = 0;
    }
  }
}

template<class A>
inline TMatrix<A>::TMatrix():TMatrix<A> (1,1)
{
}

template<class A>
inline int TMatrix<A>::GetLen2()
{
  return len2;
}

template<class A>
inline void TMatrix<A>::MultVect(TVector<A>& p1, const TVector<A>& p2)
{
  if ((!p1.GetTran()) || p2.GetTran())
  {
    throw out_of_range("wrong transponation");
  }
  TMatrix<A> M1(p1.GetLen(), 1);
  for (int row = 0; row < p1.GetLen(); row++)
  {
```

```
    M1[row][0] = p1[row];
  }

  TMatrix<A> M2(1, p2.GetLen());
  M2[0] = p2;

  *this = M1*M2;
}

template<class A>
inline void TMatrix<A>::MultVectMatrix(TVector<A>& p, TMatrix<A>& m)
{
  if (p.GetTran())
  {
    if (p.GetLen() != m.GetLen2())
    {
      throw out_of_range("diferent sizes");
    }
    TMatrix<A> M1(p.GetLen(), 1);
    for (int row = 0; row < p.GetLen(); row++)
    {
      M1[row][0] = p[row];
    }
    *this = m * M1;
  }
  else
  {
    if (p.GetLen() != m.GetLen())
    {
      throw out_of_range("diferent sizes");
    }
    TMatrix<A> M2(1, p.GetLen());
    M2[0] = p;
    *this = M2 * m;
  }
}

template<class A>
inline int TMatrix<A>::CountIncludeMatrix(A ch)
{
  int count=0;
  for (int i = 0; i < len; i++)
  {
    for (int j = 0; j < len2; j++)
    {
      if (data[i][j] == ch)
      {
        count++;
      }
    }
  }
  return count;
}

template<class A>
inline bool TMatrix<A>::operator==(const TMatrix<A>& p)
{
  return TVector<TVector<A>>::operator==(p);
}

template<class A>
inline TMatrix<A> TMatrix<A>::operator+(const TMatrix<A>& p)
{
  if (len == p.len && len2 == p.len2)
```

```
  {
    TMatrix<A> res(len, len2);
    for (int i = 0; i < len; i++)
    {
      for (int j = 0; j < len2; j++)
      {
        res[i][j] = data[i][j] + p.data[i][j];
      }
    }
    return res;
  }
  else
  {
    throw out_of_range("different sizes");
  }

}

template<class A>
inline TMatrix<A> TMatrix<A>::operator-(const TMatrix<A>& p)
{
  if (len == p.len && len2 == p.len2)
  {
    TMatrix<A> res(len, len2);
    for (int i = 0; i < len; i++)
    {
      for (int j = 0; j < len2; j++)
      {
        res[i][j] = data[i][j] - p.data[i][j];
      }
    }
    return res;
  }
  else
  {
    throw out_of_range("different sizes");
  }

}

template<class A>
inline TMatrix<A> TMatrix<A>::operator*(const TMatrix<A>& p)
{
  if (len2 == p.len)
  {
    TMatrix<A> res(len, p.len2);
    for (int i = 0; i < len; i++)
    {
      for (int j = 0; j < p.len2; j++)
      {
        res[i][j] = 0;
        for (int t = 0; t < len2; t++)
        {
          res[i][j] += data[i][t] * p.data[t][j];
        }
      }
    }
    return res;
  }
  else
  {
    throw out_of_range("different sizes*");
  }
}
```

```
template<class A>
inline TMatrix<A> TMatrix<A>::operator=(const TMatrix<A>& p)
{
  TVector<TVector<A>>::operator = (p);
  return *this;
}

template<class A>
ostream& operator <<(ostream& ostr, TMatrix<A>& p)
{
  for (int i = 0; i < p.GetLen(); i++)
  {
    auto v = p[i];
    for (int j = 0; j < p.GetLen2(); j++)
    {
      ostr << p[i][j] << "\t";
    }
    ostr<< endl;
  }
  return ostr;
}

template<class A>
istream& operator >>(istream& istr, TMatrix<A>& p)
{
  int numrow;
  int numcol;
  istr >> numrow >> numcol;
  TMatrix<A> res(numrow, numcol);
  for (int row = 0; row < numrow; row++)
  {
    for (int col = 0; col < numcol; col++)
    {
      istr >> res[row][col];
    }
  }
  p = res;
  return istr;
}
```