Федеральное государственное образовательное бюджетное учреждение

высшего профессионального образования

«Нижегородский Государственный Университет им.

Н.И.Лобачевского» (ННГУ)

Институт Информационных Технологий Математики и Механики

Отчёт по лабораторной работе

Библиотека для работы с векторами и матрицами

Выполнил:

студент группы 3821Б1ПМ3

Сучков В.Н.

Проверил:

заведующий лабораторией суперкомпьютерных технологий и высокопроизводительных вычислений

Лебедев И.Г

Нижний Новгород

2022г.

**Оглавление**

[1 Введение 4](#_Toc104113736)

[2 Постановка задачи 5](#_Toc104113737)

[3 Руководство пользователя 6](#_Toc104113738)

[3.1 Работа с векторами при помощи библиотеки Vector 6](#_Toc104113739)

[3.1.1 Подключаем библиотеку Vector 6](#_Toc104113740)

[3.1.2 Создаем объекты-векторы 6](#_Toc104113741)

[4 Базовые методы класса TVector 6](#_Toc104113742)

[4.1.1 Операции над векторами 7](#_Toc104113743)

[4.2 Работа с матрицами при помощи библиотеки Matrix 9](#_Toc104113744)

[4.2.1 Подключаем библиотеку Matrix 9](#_Toc104113745)

[4.2.2 Создаем объекты-матрицы 9](#_Toc104113746)

[4.2.3 Базовые методы класса TMatrix 9](#_Toc104113747)

[4.2.4 Операции над матрицами 10](#_Toc104113748)

[5 Руководство программиста 12](#_Toc104113749)

[5.1 Класс TVector 12](#_Toc104113750)

[5.1.1 Объявление шаблонного класса 12](#_Toc104113751)

[5.1.2 Реализация методов и операций 13](#_Toc104113752)

[5.2 Класс TMatrix 19](#_Toc104113753)

[5.2.1 Объявление шаблонного класса 19](#_Toc104113754)

[5.2.2 Реализуем методы и операции 20](#_Toc104113755)

[6 Алгоритмы и данные 27](#_Toc104113756)

[6.1 О данных 27](#_Toc104113757)

[6.2 Об алгоритмах 27](#_Toc104113758)

[7 Эксперименты 29](#_Toc104113759)

[7.1 Умножение матриц 29](#_Toc104113760)

[7.2 Сложение матриц 30](#_Toc104113761)

[8 Заключение 32](#_Toc104113762)

[9 Список литературы 33](#_Toc104113763)

[10 Приложение 34](#_Toc104113765)

[10.1 Приложение 1 34](#_Toc104113766)

[10.2 Приложение 2 42](#_Toc104113767)

# Введение

Цель настоящей работы описать написанную в ходе лабораторной работы библиотеку, предназначенную для работы с алгебраическими векторами и матрицами. Отдельной темой работы является вопрос об эффективности используемых методов работы с векторами и матрицами.

В прикладных задачах часто требуется работать с данными, собираемыми в коллекции. Векторы и матрицы позволяют удобно обрабатывать и хранить большое количество информации. Помимо того, векторы и матрицы необходимы при решении различных задач физики.

Потому реализация библиотеки, позволяющей работать с этими объектами, является важным практическим опытом, укрепляющим умения и навыки программирования. Отдельной темой работы является вопрос об эффективности используемых методов работы с векторами и матрицами.

# Постановка задачи

Написать классы для работы с векторами и матрицами, использовать шаблоны. Вектора в математическом понимании: имеется набор значений из N мерного пространства, размерность задается как параметр. Классы вектора и матрицы должны быть вынесены в статическую библиотеку.

# Руководство пользователя

## Работа с векторами при помощи библиотеки Vector

### Подключаем библиотеку Vector

|  |
| --- |
| *#include* "Vector.h" |

Листинг ‑

### Создаем объекты-векторы

Векторы реализуются в классе TVector библиотеки Vector. Задать вектор при создании можно разыми способами. Ниже с комментариями продемонстрированы все возможные варианты. Тип данных, с которым предстоит работать указывается в угловых скобках при объявлении новой переменной.

|  |
| --- |
| int main() {  TVector<int> a; *//При таком объявлении вектор изначально будет иметь нулевую длинну и не будет содержать никаких данных*  TVector<int> b(2); *//В этом случае мы получим вектор длинной 2, каждая координата которого изначально равна 0 - b = (0, 0)*  TVector<int> c(3, -1); *//Вектор с будет иметь 3 координаты изначально ранвые -1. с = (-1, -1, -1)*  TVector<int> d = c; *//Создаем вектор равный другому. Их длины будут равны, также как и их соответствующие координаты*  int t[4] = {0, -1, 2, 4};  TVector<int> e(4, t); *//Вектор e будет иметь длину 4 и будет заполнен соответсвующим числами из массива t*  } |

Листинг ‑

# Базовые методы класса TVector

В классе TVector реализованы методы необходимые для дальнейшей работы с векторами. В их числе (для некоторого типа T элементов вектора):

|  |  |  |
| --- | --- | --- |
| Метод | Описание | Тип возвращаемого значения |
| GetLen() | Данный метод возвращает длину[[1]](#footnote-1) вектора, для которого он вызван | int |
| GetCoord(const int index) | Возвращает значение координаты под номером index данного вектора (отсчет ведется с 0) | T |
| SetVector(const int len, const T\* tuple) | Позволяет изменить существующий вектор на вектор с новыми характеристиками: длиной len и координатами, указанными в массиве T\* tuple | void |
| ElementQuantity(const T x) | Возвращает количество вхождений элемента x в набор координат данного вектора. (0 если элемента нет среди координат) | int |
| AllEntries(const T x, int\*& p) | Ищет все вхождения элемента x в набор координат и записывает соответствующие индексы в массив, на который после выполнения будет указывать передаваемый в качестве аргумента указатель p | void |
| BubbleSort()  InsertSort()  QuickSort() | При вызове любого их этим методов набор координат вектора будет упорядочен по возрастанию. В зависимости от вызванного метода будет применен один из трех методов сортировки: пузырьковая, вставками или быстрая | void |

Таблица ‑

### Операции над векторами

Для объектов класса TVector реализованы базовые арифметические операции, соответствующие операциям над математическими векторами (дальнейшие операции справедливы для векторов с одинаковым типом элементов T):

|  |  |  |
| --- | --- | --- |
| Операция | Описание | Тип возвращаемого значения |
| «=»  (a = b) | Операция присваивания аналогична той, что используется для встроенных типов. | Ссылка на l-value объект TVector данной операции |
| «+»  (a + b) | Операция арифметического сложения векторов (покоординатное сложение) | Объект TVector |
| «–»  (a – b) | Операция арифметического вычитания векторов (покоординатная разность) | Объект TVector |
| «\*»  (a\* scalar)  (a \* b) | 1. Если операндами являются вектор и число типа T, то результатом будет вектор умноженный на скаляр 2. Если операндами являются два вектора одинаковой длины, то результатом будет их скалярное произведения типа Т | 1)Объект TVector  2)число типа T |
| «==»  (a == b) | Сравнивает два вектора между собой. Если длины векторов равны, а также равны их соответствующие координаты, то будет возвращено true иначе false | bool |
| «[const int index]»  (a[i]) | Позволяет получить доступ к координате с номером index. Может быть как r-value так и l-value | Ссылка на координату типа T |
| «<<»  (cout << a) | Позволяет вывести координаты вектора в консоль с использованием библиотеки iostream. Координаты выводятся через запятую в одну строку | Ссылка на объект ostream |
| «>>»  (cin >> a) | Позволяет ввести (изменить) координаты вектора пользователем через консоль. Элементы по очереди отправляются пользователем в консоль пока не будут заполнены все поля | Ссылка на объект istream |

Таблица ‑

Выше были описаны методы и операции, реализованные для векторов. Далее речь пойдет о матрицах.

## Работа с матрицами при помощи библиотеки Matrix

### Подключаем библиотеку Matrix

|  |
| --- |
| *#include* "Matrix.h" |

Листинг ‑

### Создаем объекты-матрицы

Как и в случае с векторами, мы можем объявить матрицу разными способами, что продемонстрированно в листинге 3-4.

|  |
| --- |
| int main() {  TMatrix<int> A; //Матрица нулевого размера, не содержащая ни одного элемента  TMatrix<int> B(2, 3, 0); //Матрица размером 2х3(2 строки 3 столбца) заполненная нулями  TVector<double> a(3, 1);  TMatrix<double> C(3, a); //Матрица, полученная из трех векторов a, расположенных как столбцы  TVector<int> b(2,2), c(2, 0);  TVector<int> t1[2] = {b, c};  TMatrix<int> D(2, t1); // Матрица будет собрана из двух столбцов: вектора b и вектора c одинаковой(!) длины  TMatrix<double> E = C; //Матрица-копия матрицы C  } |

Листинг ‑

### Базовые методы класса TMatrix

Для матриц реализованы следующие методы (для некоторого типа данных Т):

|  |  |  |
| --- | --- | --- |
| Метод | Описание | Тип возвращаемого значения |
| GetColumnsCount() | Возвращает число столбцов матрицы. | int |
| GetRowsCount() | Возвращает число строк матрицы | int |
| GetElem(const int row, const int column) | Возвращает элемент, стоящий на пересечении строки под номером row и столбца column | T |
| GetVector(int column) | Возвращает вектор-столбец под номером column | Объект TVector |
| SetElem(const int row, const int column, T x) | Позволяет заменить элемент, стоящий на пересечении row строки и column столбца на элемент x | void |
| SetMatrix(const int columnsCount, const TVector<T> \*columns) | Позволяет изменить текущую матрицу на матрицу с columnsCount столбцами, представляющими собой векторы(одинаковой длины) массива columns | void |
| Transp() | Транспонирует матрицу | void |

Таблица ‑

### Операции над матрицами

Теперь рассмотрим операции, реализованные для матриц (матрицы с типом элементов T):

|  |  |  |
| --- | --- | --- |
| Операция | Описание | Тип возвращаемого значения |
| «=»  (A = B) | Операция присвоения, аналогичная присвоению встроенных типов | Ссылка на объект TMatrix |
| «+»  (A + B) | Возвращает матрицу, представляющую из себя сумму матриц в математическом смысле (матрицы должны быть одинаковых размеров) | Объект TMatrix |
| «–»  (A – B) | Возвращает матрицу, представляющую из себя разность матриц в математическом смысле (матрицы должны быть одинаковых размеров) | Объект TMatrix |
| «\*»  (A \* B)  (A \* scalar) (A \* vector) | 1. Результатам операции является произведение матриц в математическом смысле (количество столбцов первой матрицы должно быть равно количеству строк второй) 2. В результате умножения матрицы на скаляр, получим матрицу, состоящую из элементов исходной умноженных на скаляр 3. Произведение матрицы на вектор справа вернет нам вектор-столбец, заполненный в соответствии с математическими правилами | Объект TMatrix  Объект TMatrix  Объект TVector |
| «[const int index]»  (A[i]) | Предоставляет доступ к вектору-столбцу, стоящему на месте index. Используется в сочетании с оператором «[]» для векторов, чтобы получить доступ к конкретному элементу матрицы (A[j][i]) | Ссылка на объект TVector |
| «<<» (cout << A) | Выводит матрицу построчно в консоль (используя библиотеку iostream, как и для векторов) | Ссылка на объект ostream |
| «>>» (cin >> A) | Позволяет поэлементно задать матрицу пользователем через консоль | Ссылка на объект istream |

Таблица ‑

Таким образом, в таблицах 3-1 – 3-4 были рассмотрены все основные возможности, предоставляемые для взаимодействия между объектами матриц и векторов.

# Руководство программиста

## Класс TVector

### Объявление шаблонного класса

Для начала требуется объявить класс, который будет описывать векторы. Поле «len» отвечает за длину вектора, поле «tuple» за хранение значений координат вектора в виде массива с типом T.

|  |
| --- |
| template <class T>  class TVector {  protected:  int len;  T \*tuple;  public:  TVector();  TVector(const int \_len);  TVector(const int \_len, const T \*\_tuple);  TVector(const int \_len, const T a);  TVector(const TVector<T>& obj);  ~TVector();  int GetLen() const;  T GetCoord(const int indx) const;  void SetVector(const int \_len, const T \*\_tuple);  int ElementQuantity(const T a);  void AllEntries(const T a, int\*& p);  void BubbleSort();  void InsertSort();  void QuickSort();  TVector<T>& operator=(const TVector<T>& v);  TVector<T> operator+(const TVector<T>& v);  TVector<T> operator-(const TVector<T>& v);  TVector<T> operator\*(const T a);  TVector<T> operator/(const TVector<T>& v);  T operator\*(const TVector<T>& v);  bool operator==(const TVector<T>& v);  T& operator[](const int indx);    friend ostream& operator<<(ostream& t, const TVector<T>& v) {  if(v.GetLen() == 0) {  t << 0 << endl;  return t;  }  for(int i = 0; i < v.GetLen() - 1; i++) {  t << v.GetCoord(i) << ", ";  }  t << v.GetCoord(v.GetLen() - 1) << endl;  return t;  }  friend istream& operator>>(istream& t, TVector<T>& v) {  if(v.GetLen() == 0)  throw "Error: 0-lenght vector >>";  for(int i = 0; i < v.GetLen(); i++) {  cout << "Enter vector " << i << " coordinate" << endl;  t >> v[i];  }  return t;  }  }; |

Листинг ‑

Внутри класса объявлены прототипы методов, операций. Дружественные функции ввода-вывода были реализованы внутри класса.

### Реализация методов и операций

Реализуем конструкторы и деструктор.

|  |
| --- |
| //Конструктор по умолчанию  template <class T>  TVector<T>::TVector() {  len = 0;  tuple = 0;  }  //Деструктор  template <class T>  TVector<T>::~TVector() {  if(tuple != 0) {  delete [] tuple;  tuple = 0;  }  }  //Конструктор инициализации, заполняющий вектор известной длины нулями  template <class T>  TVector<T>::TVector(const int \_len) {  len = \_len;  tuple = new T[len];  for(int i = 0; i < len ; i++) {  tuple[i] = 0;  }  }  //Конструктор инициализации, заполняющий вектор скаляром a  template <class T>  TVector<T>::TVector(const int \_len, const T a) {  if(len == 0) {  TVector();  return;  }  len = \_len;  tuple = new T[len];  for(int i = 0; i < len; i++) {  tuple[i] = a;  }  }  //Конструктор инициализации массивом значений известной длины  template <class T>  TVector<T>::TVector(const int \_len, const T\* \_tuple) {  if((\_len == 0) || (\_tuple == 0)) {  TVector();  return;  }  len = \_len;  tuple = new T[len];  for(int i = 0; i < len; i++) {  tuple[i] = \_tuple[i];  }  }  //Конструктор копирования  template <class T>  TVector<T>::TVector(const TVector<T>& obj) {  if(obj.len == 0) {  TVector();  return;  }  len = obj.len;  tuple = new T[len];  for(int i = 0; i < len; i++) {  tuple[i] = obj.GetCoord(i);  }  } |

Листинг ‑

Реализуем основные методы.

|  |
| --- |
| template <class T>  int TVector<T>::GetLen() const {  return len;  }  //Получение координаты  template <class T>  T TVector<T>::GetCoord(const int indx) const {  if(indx < 0 || indx >= len)  throw "Error: out of tuple range";  return tuple[indx];  }  //Задание вектора  template <class T>  void TVector<T>::SetVector(const int \_len, const T \*\_tuple) {  if( \_len == 0) {  len = \_len;  if(tuple != 0) {  delete [] tuple;  tuple = 0;  }  return;  }    len = \_len;  if(tuple != 0) {  delete [] tuple;  tuple = 0;  }  tuple = new T[len];  for(int i = 0; i < len; i ++) {  tuple[i] = \_tuple[i];  }  }  //Считаем количество вхождений элемента а  template <class T>  int TVector<T>::ElementQuantity(const T a) {  if(len == 0)  return 0;    int result = 0;  for(int i = 0; i < len; i++){  if(tuple[i] == a)  result++;  }  return result;  }  //Находим все вхождения элемента а  template <class T>  void TVector<T>::AllEntries(const T a, int\*& p) {  int quan = this->ElementQuantity(a);  if(quan = 0) {  p = 0;  return;  }  p = new int[quan];  int count = 0;  for(int i = 0; i < len; i++) {  if(tuple[i] == a) {  p[count] = i;  count++;  }  }  return;  }  //Сортировки  template <class T>  void TVector<T>::BubbleSort() {  T temp;  for(int i = 0; i < len; i++) {  for(int j = len - 1; j > i; j--) {  if(tuple[j-1] > tuple[j]) {  temp = tuple[j-1];  tuple[j-1] = tuple[j];  tuple[j] = temp;  }  }  }  }  template <class T>  void TVector<T>::InsertSort() {  T temp;  for(int i = 1; i < len; i++) {  for(int j = i-1; j >= 0 && tuple[j] > tuple[j+1]; j--) {  temp = tuple[j+1];  tuple[j+1] = tuple[j];  tuple[j] = temp;  }  }  }  template <class T>  void Quick(T \*a, int N) {  int i = 0, j = N-1;  T temp, p;  p = a[N>>1];  while(i < j) {  while(a[i] < p) i++;  while(a[j] > p) j--;  if(i < j) {  temp = a[i];  a[i] = a[j];  a[j] = temp;  i++;  j--;  }  }  if(j > 1) Quick(a, j);  if(N > i+1) Quick(a+i, N-i);  }  template <class T>  void TVector<T>::QuickSort() {  Quick(tuple, len);  return;  } |

Реализуем операторы

|  |
| --- |
| template <class T>  T& TVector<T>::operator[](const int indx) {  if((indx < 0) || (indx >= len)) {  throw "Error: out of tuple range";  }  return tuple[indx];  }  template <class T>  TVector<T>& TVector<T>::operator=(const TVector<T>& v) {  if(v.len == 0) {  throw "Error: vector=null error";  }  len = v.len;  if(tuple != 0) {  delete [] tuple;  tuple = 0;  }  tuple = new T[len];  for(int i = 0; i < len; i++) {  tuple[i] = v.tuple[i];  }  return \*this;  }  template <class T>  bool TVector<T>::operator==(const TVector<T>& v) {  if(len != v.len)  return false;  for(int i = 0; i < len; i++) {  if(tuple[i] != v.tuple[i])  return false;  }  return true;  }  template <class T>  TVector<T> TVector<T>::operator+(const TVector<T>& v) {  if(len != v.GetLen())  throw "Error: vector+vector(len1 != len2) error";  TVector<T> temp(len);  for(int i = 0; i < len; i++) {  temp[i] = (\*this)[i] + v.GetCoord(i);  }  return temp;  }  template <class T>  TVector<T> TVector<T>::operator\*(const T a) {  if(len == 0)  return \*this;  TVector<T> temp(len);  for(int i = 0; i < len; i++) {  temp[i] = a\*tuple[i];  }  return temp;  }  template<class T>  T TVector<T>::operator\*(const TVector<T>& v) {  if(len != v.GetLen())  throw "Error: multiplying diffetent-lenght vectors";  if(len == 0)  throw "Error: multiplying 0 lenght vectors";  T result = 0;  for(int i = 0; i < len; i++) {  result += (\*this)[i]\*v.GetCoord(i);  }  return result;  }  template <class T>  TVector<T> TVector<T>::operator-(const TVector<T>& v) {  return ((\*this) + v\*(-1));  } |

## Класс TMatrix

### Объявление шаблонного класса

Класс, описывающий матрицы, основан на агрегационном подходе: матрицы представлены как массив векторов. При таком представлении матрица характеризуется полями: количество столбцов, количество строк, указатель на массив векторов.

Итак, объявим класс.

|  |
| --- |
| template <class T>  class TMatrix : public TVector<T> {  protected:  int rowsCount; //Количество строк  int columnsCount; //Количество столбцов  TVector<T> \*columns; //Указатель на массив векторов  public:  TMatrix();  TMatrix(const int \_rowsCount, const int \_columnsCount, const T a);  TMatrix(const int \_columnsCount, const TVector<T> \*\_columns);  TMatrix(const int \_columnsCount, const TVector<T>& \_column);  TMatrix(const TMatrix<T>& obj);  TMatrix(const TVector<T>& vect);  ~TMatrix();  T GetElem(const int row, const int column) const;  int GetRowsCount() const;  int GetColumnsCount() const;  TVector<T> GetVector(int column) const;  void SetElem(const int row, const int column, T a);  void SetMatrix(const int \_columnsCount, const TVector<T> \*\_columns);  void Transp();  bool operator==(const TMatrix<T>& m);  TMatrix<T>& operator=(const TMatrix<T>& m);  TVector<T>& operator[](const int indx);  TMatrix<T> operator+(const TMatrix<T>& m);  TMatrix<T> operator\*(const TMatrix<T>& m);  TMatrix<T> operator\*(const T a);  TVector<T> operator\*(const TVector<T>& v);    TMatrix<T> operator-(const TMatrix<T>& m);    friend ostream& operator<<(ostream& t, const TMatrix<T>& m) {  if(m.GetColumnsCount() == 0)  throw "Error: cannot print 0-matrix";  for(int i = 0; i < m.GetRowsCount(); i++) {  for(int j = 0; j < m.GetColumnsCount(); j++) {  t << m.GetElem(i, j) << ' ';  }  t << '\n';  }  return t;  }  friend istream& operator>>(istream& t, const TMatrix<T>& m) {  if(m.GetColumnsCount() == 0)  throw "Error: cannot write in 0-matrix";  for(int i = 0; i < m.GetRowsCount(); i++) {  for(int j = 0; j < m.GetColumnsCount(); j++) {  cout << "Enter (" << j << ", " << i << ") element:" << endl;  t >> m[j][i];  }  }  return t;  }  }; |

Как и в случае с векторами операторы ввода-вывода объявлены на месте.

### Реализуем методы и операции

Далее перейдем к реализации конструкторов и деструкторов.

|  |
| --- |
| //Конструктор по умолчанию  template <class T>  TMatrix<T>::TMatrix() {  rowsCount = 0;  columnsCount = 0;  columns = 0;  }  //Деструктор  template <class T>  TMatrix<T>::~TMatrix() {  if(columns != 0) {  delete [] columns;  columns = 0;  }  }  //Конструктор инициализации, заполняющий матрицу известных размеров скаляром а  template <class T>  TMatrix<T>::TMatrix(const int \_rowsCount, const int \_columnsCount, const T a) {  if(\_rowsCount == 0 || \_columnsCount == 0) {  TMatrix();  return;  }  rowsCount = \_rowsCount;  columnsCount = \_columnsCount;  columns = new TVector<T>[columnsCount];  for(int i = 0; i < columnsCount; i++) {  columns[i] = TVector<T>(rowsCount, a);  }  }  //Конструктор инициализации, заполняющий матрицу векторами-столбцами  template <class T>  TMatrix<T>::TMatrix(const int \_columnsCount, const TVector<T> \*\_columns) {  if(\_columnsCount == 0 || \_columns == 0) {  TMatrix();  return;  }  columnsCount = \_columnsCount;  rowsCount = \_columns[0].GetLen();  columns = new TVector<T>[columnsCount];  for(int i = 0; i < columnsCount; i++) {  columns[i] = \_columns[i];  }  }  //Конструктор инициализации, заполняющий матрицу одним и тем же столбцом-вектором  template <class T>  TMatrix<T>::TMatrix(const int \_columnsCount, const TVector<T>& \_column) {  if(\_columnsCount == 0 || \_column.GetLen() == 0) {  TMatrix();  return;  }  columnsCount = \_columnsCount;  rowsCount = \_column.GetLen();  columns = new TVector<T>[columnsCount];  for(int i = 0; i < columnsCount; i++) {  columns[i] = \_column;  }  }  //Конструктор копирования  template <class T>  TMatrix<T>::TMatrix(const TMatrix<T>& obj) {  if(obj.columnsCount == 0) {  TMatrix();  return;  }  columnsCount = obj.columnsCount;  rowsCount = obj.rowsCount;  columns = new TVector<T>[columnsCount];  for(int i = 0; i < columnsCount; i++) {  columns[i] = obj.columns[i];  }  }  //Конструктор, преобразующий вектор в матрицу из одного столбца  template <class T>  TMatrix<T>::TMatrix(const TVector<T>& vect) {  if(vect.GetLen() == 0) {  TMatrix();  return;  }  columnsCount = 1;  rowsCount = vect.GetLen();  columns = new TVector<T>[1];  columns[0] = vect;  } |

Затем реализуем методы.

|  |
| --- |
| //Получаем число строк  template <class T>  int TMatrix<T>::GetRowsCount() const {  return rowsCount;  }  //Получаемчисло столбцов  template <class T>  int TMatrix<T>::GetColumnsCount() const {  return columnsCount;  }  //Получаем элемент матрицы  template <class T>  T TMatrix<T>::GetElem(const int row, const int column) const {  if(row < 0 || row >= rowsCount || column < 0 || column >= columnsCount)  throw "Error: run out of matrix range";    return columns[column][row];  }  //Получаем вектор-столбец матрицы  template <class T>  TVector<T> TMatrix<T>::GetVector(int column) const {  if(column < 0 || column >= columnsCount)  throw "Error: run out of matrix range";  return columns[column];  }  //Задаем элемент матрицы  template <class T>  void TMatrix<T>::SetElem(const int row, const int column, T a) {  if(row < 0 || row >= rowsCount || column < 0 || column >= columnsCount)  throw "Error: out of matrix range";  columns[column][row] = a;  return;  }  //Задаем матрицу  template <class T>  void TMatrix<T>::SetMatrix(const int \_columnsCount, const TVector<T> \*\_columns) {  if(\_columnsCount == 0 || \_columns == 0) {  columnsCount = 0;  rowsCount = 0;  if(columns != 0) {  delete [] columns;  columns = 0;  }  }  columnsCount = \_columnsCount;  rowsCount = \_columns[0].GetLen();  if(columns != 0) {  delete [] columns;  columns = 0;  }  columns = new TVector<T>[columnsCount];    for(int i = 0; i < columnsCount; i++) {  columns[i] = \_columns[i];  }  }  //Транспонирование  template <class T>  void TMatrix<T>::Transp() {  if(columns == 0) {  return;  }  int tm;  tm = columnsCount;  columnsCount = rowsCount;  rowsCount = tm;  TVector<T> \*temp = new TVector<T>[columnsCount];  for(int i = 0; i < rowsCount; i++) {  for(int j = 0; j < columnsCount; j++) {  temp[i][j] = columns[j][i];  }  }  delete [] columns;  columns = new TVector<T>[columnsCount];  for(int i = 0; i < columnsCount; i++) {  columns[i] = temp[i];  }  return;  } |

Реализуем операции.

|  |
| --- |
| template <class T>  bool TMatrix<T>::operator==(const TMatrix<T>& m) {  if(columnsCount == m.GetColumnsCount() && rowsCount == m.GetRowsCount()) {  for(int i = 0; i < columnsCount; i++) {  if(columns[i] != m.GetVector(i))  return false;  }  return true;  }  return false;  }  template <class T>  TMatrix<T>& TMatrix<T>::operator=(const TMatrix<T>& m) {  if(m.GetColumnsCount == 0 || m.GetRowsCount() == 0)  throw "Error: matrix=0-matrix";    columnsCount = m.GetColumnsCount();  rowsCount = m.GetRowsCount();  if(columns != 0) {  delete [] columns;  columns = 0;  }  columns = new TVector<T>[columnsCount];  for(int i = 0; i < columnsCount; i++) {  columns[i] = m.GetVector(i);  }  return \*this;  }  template <class T>  TVector<T>& TMatrix<T>::operator[](const int indx) {  if(indx < 0 || indx >= columnsCount)  throw "Error: run out of matrix range";  return columns[indx];  }  template <class T>  TMatrix<T> TMatrix<T>::operator+(const TMatrix<T>& m) {  if(m.GetColumnsCount() != columnsCount || m.GetRowsCount() != rowsCount)  throw "Error: different matrix size in +";    TMatrix<T> temp(columnsCount, rowsCount, 0);  for(int i = 0; i < rowsCount; i++) {  for(int j = 0; j < columnsCount; j++) {  temp[j][i] = this->GetElem(i, j) + m.GetElem(i, j);  }  }  return temp;  }  template <class T>  TMatrix<T> TMatrix<T>::operator\*(const T a) {  TMatrix<T> temp = this;  for(int i = 0; i < columnsCount; i++) {  temp[i] = temp[i]\*a;  }  return temp;  }  template <class T>  TMatrix<T> TMatrix<T>::operator-(const TMatrix<T>& m) {  if(m.GetColumnsCount() != columnsCount || m.GetRowsCount() != rowsCount)  throw "Error: different matrix size in +";  TMatrix<T> temp(columnsCount, rowsCount, 0);  for(int i = 0; i < columnsCount; i++) {  temp[i] = this->GetVector(i) - m.GetVector();  }  return temp;  }  template <class T>  TMatrix<T> TMatrix<T>::operator\*(const TMatrix<T>& m) {  if(columnsCount != m.GetRowsCount())  throw "Error: cannot multiply matrix (invalid size)";    TMatrix<T> temp(m.GetColumnsCount(), rowsCount, 0);  for(int k = 0; k < rowsCount; k++) {  for(int j = 0; j < m.GetColumnsCount(); j++) {  for(int i = 0; i < columnsCount; i++) {  temp[j][k] += this->GetElem(k, i)\*m.GetElem(i, j);  }  }  }  return temp;  }  template <class T>  TVector<T> TMatrix<T>::operator\*(const TVector<T>& v) {  if(v.GetLen() != columnsCount)  throw "Error: cannot muiltiply matrix\*vector (invalid size)";    TVector<T> temp(rowsCount);  for(int i = 0; i < rowsCount; i++) {  for(int j = 0; j < v.GetLen(); j++) {  temp[i] += this->GetElem(i, j) \* v.GetCoord(j);  }  }  return temp;  } |

# Алгоритмы и данные

## О данных

Что класс векторов, что матриц – оба шаблонные классы. Таким образом, пользователь сам выбирает, какого типа будут данные, которые будут содержать векторы и матрицы, в зависимости от решаемой задачи это могут быть и целые числа, и числа с плавающей запятой. Однако, для векторов и матриц были реализованы именно арифметические операции, то есть, предполагается использование именно скалярного типа, или, по крайней мере, типа для которого реализованы все необходимые операции, соответствующие операциям над встроенными скалярными типами.

В случае с векторами все просто: при объявлении вектора вызывается один из конструкторов, который инициализирует поля создаваемого объекта. В случае, если создается не пустой вектор (вектор нулевой длины), то происходит динамическое выделение памяти – создается динамический массив, которые заполняется какими-либо значениями.

С матрицами все обстоит схоже: при объявлении матрицы не нулевого размера, будет создан динамический массив, каждым элементом которого, является вектор. Каждый из этих векторов так же будет проинициализирован и под него будет выделена память в куче.

## Об алгоритмах

Наибольший интерес представляют алгоритмы, используемые при реализации основных арифметических операций над матрицами. Они основаны на математических правилах сложения и умножения матриц, а быть точнее на их определениях.

Сложение матриц довольно примитивное действие: соответствующие элементы матриц складываются и записываются на свое место в матрице-результате. В данном случае реализация следующая: берутся соответствующие векторы-столбцы матриц и складываются, как векторы. Результат записываться в вектор-столбец результата и так для всех столбцов. Сложность для матрицы оценивается как .

Умножение матриц операция куда более трудоемкая. В рамках этой работы был реализован самый простой вариант умножения – по определению. Формула для нахождения элемента матрицы-результата умножения матриц размером и выглядит следующим образом:

В результате умножения получим матрицу , для нахождения всех элементов которой, потребуется вычислений сумм формулы (1).

Из формулы (1) не сложно получить, что алгоритм умножения, реализованный с использованием трех вложенных циклов, для случая двух квадратных матриц размера по сложности можно оценить как .

# Эксперименты

## Умножение матриц

В первом эксперименте попробуем сравнить время выполнения операций сложения и умножения в теории и на практике. Для этого воспользуемся библиотекой «time.h».

Напишем соответствующий код.

|  |
| --- |
| int main() {  try {  int n = 125;    TMatrix<int> A(n, n, 99);  TMatrix<int> B(n, n, 99);  TMatrix<int> C(n, n, 0);    unsigned start = clock(), finish = 0;    C = A\*B;  finish = clock() - start;    cout << "Executed for " << finish << " ms"<< endl;  }  catch(const char \*s) {  cout << s << endl;  } |

Листинг ‑

По результатам выполнения программы (см. рисунок 6-1), две квадратные матрицы 125-ого порядка умножаются за .



Рисунок ‑

Листинг ‑

После проведения серии опытов с разными значениями n получим следующие результаты, представленные в виде графика:

График ‑. Зависимость времени работы от n

Непосредственно сам алгоритм умножения отрабатывает за однако, помимо умножения в алгоритме происходит выделение памяти для элементов, требующее времени. Итого получим время работы Коэффициенты несложно найти из имеющихся данных ( . Полученные результаты добавлены на график 6-1 для сравнения.

## Сложение матриц

Во втором эксперименте сравним время работы сложения матриц в теории и на практике. Соответствующий код для эксперимента приведен ниже.

|  |
| --- |
| int main() {  try {  int n = 1000;  for(n; n < 3000; n += 100) {  TMatrix<int> A(n, n, 9999999);  TMatrix<int> B(n, n, 9999999);  TMatrix<int> C(n, n, 0);  unsigned start = clock(), finish = 0;    C = A+B;  finish = clock() - start;    cout << "Executed for " << finish << " ms"<< endl;  }  }  catch(const char \*s) {  cout << s << endl;  }  } |

Листинг ‑

По полученным после срабатывания программы данным построим график:

График ‑. Зависимость времени работы от n

Сложение матриц оценивается как , так же, как и выделение памяти для создания временной матрицы. Отсюда получим (). Вычисленные по формуле значения отмечены на графике для сравнения.

# Заключение

В ходе лабораторной работы мной была написана библиотека для работы с математическими векторами и матрицами, реализованная с использованием шаблонных классов и агрегации.

Помимо того, было проведено сравнение времени работы алгоритмов, реализующих арифметические операции для матриц на практике с ожидаемым временем работы, по результатам которого удалось выяснить, что алгоритмы отрабатывают с минимальным отклонением от ожидаемого времени.

# Список литературы

### С++. Объектно-ориентированное программирование: Учебное пособие.–СПб.: Питер, 2008. – 464с.: ил. – (Серия «Учебное пособие»)

# Приложение

## Приложение 1

|  |
| --- |
| #pragma once  #include <iostream>  using namespace std;  template <class T>  class TVector {  protected:  int len;  T \*tuple;  public:  TVector();  TVector(const int \_len);  TVector(const int \_len, const T \*\_tuple);  TVector(const int \_len, const T a);  TVector(const TVector<T>& obj);  ~TVector();  int GetLen() const;  T GetCoord(const int indx) const;  void SetVector(const int \_len, const T \*\_tuple);  int ElementQuantity(const T a);  void AllEntries(const T a, int\*& p);  void BubbleSort();  void InsertSort();  void QuickSort();  TVector<T>& operator=(const TVector<T>& v);  TVector<T> operator+(const TVector<T>& v);  TVector<T> operator-(const TVector<T>& v);  TVector<T> operator\*(const T a);  TVector<T> operator/(const TVector<T>& v);  T operator\*(const TVector<T>& v);  bool operator==(const TVector<T>& v);  T& operator[](const int indx);    friend ostream& operator<<(ostream& t, const TVector<T>& v) {  if(v.GetLen() == 0) {  t << 0 << endl;  return t;  }  for(int i = 0; i < v.GetLen() - 1; i++) {  t << v.GetCoord(i) << ", ";  }  t << v.GetCoord(v.GetLen() - 1) << endl;  return t;  }  friend istream& operator>>(istream& t, TVector<T>& v) {  if(v.GetLen() == 0)  throw "Error: 0-lenght vector >>";  for(int i = 0; i < v.GetLen(); i++) {  cout << "Enter vector " << i << " coordinate" << endl;  t >> v[i];  }  return t;  }  };  //##################### Конструкторы #######################  template <class T>  TVector<T>::TVector() {  len = 0;  tuple = 0;  }  template <class T>  TVector<T>::~TVector() {  if(tuple != 0) {  delete [] tuple;  tuple = 0;  }  }  template <class T>  TVector<T>::TVector(const int \_len) {  len = \_len;  tuple = new T[len];  for(int i = 0; i < len ; i++) {  tuple[i] = 0;  }  }  template <class T>  TVector<T>::TVector(const int \_len, const T a) {  if(len == 0) {  TVector();  return;  }  len = \_len;  tuple = new T[len];  for(int i = 0; i < len; i++) {  tuple[i] = a;  }  }  template <class T>  TVector<T>::TVector(const int \_len, const T\* \_tuple) {  if((\_len == 0) || (\_tuple == 0)) {  TVector();  return;  }  len = \_len;  tuple = new T[len];  for(int i = 0; i < len; i++) {  tuple[i] = \_tuple[i];  }  }  template <class T>  TVector<T>::TVector(const TVector<T>& obj) {  if(obj.len == 0) {  TVector();  return;  }  len = obj.len;  tuple = new T[len];  for(int i = 0; i < len; i++) {  tuple[i] = obj.GetCoord(i);  }  }  //####################### Методы ########################  template <class T>  int TVector<T>::GetLen() const {  return len;  }  template <class T>  T TVector<T>::GetCoord(const int indx) const {  if(indx < 0 || indx >= len)  throw "Error: out of tuple range";  return tuple[indx];  }  template <class T>  void TVector<T>::SetVector(const int \_len, const T \*\_tuple) {  if( \_len == 0) {  len = \_len;  if(tuple != 0) {  delete [] tuple;  tuple = 0;  }  return;  }    len = \_len;  if(tuple != 0) {  delete [] tuple;  tuple = 0;  }  tuple = new T[len];  for(int i = 0; i < len; i ++) {  tuple[i] = \_tuple[i];  }  }  //околодопы  template <class T>  int TVector<T>::ElementQuantity(const T a) {  if(len == 0)  return 0;    int result = 0;  for(int i = 0; i < len; i++){  if(tuple[i] == a)  result++;  }  return result;  }  template <class T>  void TVector<T>::AllEntries(const T a, int\*& p) {  int quan = this->ElementQuantity(a);  if(quan = 0) {  p = 0;  return;  }  p = new int[quan];  int count = 0;  for(int i = 0; i < len; i++) {  if(tuple[i] == a) {  p[count] = i;  count++;  }  }  return;  }  template <class T>  void TVector<T>::BubbleSort() {  T temp;  for(int i = 0; i < len; i++) {  for(int j = len - 1; j > i; j--) {  if(tuple[j-1] > tuple[j]) {  temp = tuple[j-1];  tuple[j-1] = tuple[j];  tuple[j] = temp;  }  }  }  }  template <class T>  void TVector<T>::InsertSort() {  T temp;  for(int i = 1; i < len; i++) {  for(int j = i-1; j >= 0 && tuple[j] > tuple[j+1]; j--) {  temp = tuple[j+1];  tuple[j+1] = tuple[j];  tuple[j] = temp;  }  }  }  template <class T>  void Quick(T \*a, int N) {  int i = 0, j = N-1;  T temp, p;  p = a[N>>1];  while(i < j) {  while(a[i] < p) i++;  while(a[j] > p) j--;  if(i < j) {  temp = a[i];  a[i] = a[j];  a[j] = temp;  i++;  j--;  }  }  if(j > 1) Quick(a, j);  if(N > i+1) Quick(a+i, N-i);  }  template <class T>  void TVector<T>::QuickSort() {  Quick(tuple, len);  return;  }  //######################### Операторы #########################  template <class T>  T& TVector<T>::operator[](const int indx) {  if((indx < 0) || (indx >= len)) {  throw "Error: out of tuple range";  }  return tuple[indx];  }  template <class T>  TVector<T>& TVector<T>::operator=(const TVector<T>& v) {  if(v.len == 0) {  throw "Error: vector=null error";  }  len = v.len;  if(tuple != 0) {  delete [] tuple;  tuple = 0;  }  tuple = new T[len];  for(int i = 0; i < len; i++) {  tuple[i] = v.tuple[i];  }  return \*this;  }  template <class T>  bool TVector<T>::operator==(const TVector<T>& v) {  if(len != v.len)  return false;  for(int i = 0; i < len; i++) {  if(tuple[i] != v.tuple[i])  return false;  }  return true;  }  template <class T>  TVector<T> TVector<T>::operator+(const TVector<T>& v) {  if(len != v.GetLen())  throw "Error: vector+vector(len1 != len2) error";  TVector<T> temp(len);  for(int i = 0; i < len; i++) {  temp[i] = (\*this)[i] + v.GetCoord(i);  }  return temp;  }  template <class T>  TVector<T> TVector<T>::operator\*(const T a) {  if(len == 0)  return \*this;  TVector<T> temp(len);  for(int i = 0; i < len; i++) {  temp[i] = a\*tuple[i];  }  return temp;  }  template<class T>  T TVector<T>::operator\*(const TVector<T>& v) {  if(len != v.GetLen())  throw "Error: multiplying diffetent-lenght vectors";  if(len == 0)  throw "Error: multiplying 0 lenght vectors";  T result = 0;  for(int i = 0; i < len; i++) {  result += (\*this)[i]\*v.GetCoord(i);  }  return result;  }  template <class T>  TVector<T> TVector<T>::operator/(const TVector<T>& v) {  if(len != v.GetLen())  throw "Error: multiplying diffetent-lenght vectors";  if(len == 0)  throw "Error: multiplying 0 lenght vectors";  TVector<T> temp(len);  for(int i = 0; i < len; i++) {  if(v.GetCoord != 0)  temp[i] = tuple[i] / v.GetCoord(i);  else temp[i] = 0;  }  return temp;  }  template <class T>  TVector<T> TVector<T>::operator-(const TVector<T>& v) {  return ((\*this) + v\*(-1));  } |

## Приложение 2

|  |
| --- |
| #pragma once  #include "Vector.h"  template <class T>  class TMatrix : public TVector<T> {  protected:  int rowsCount;  int columnsCount;  TVector<T> \*columns;  public:  TMatrix();  TMatrix(const int \_rowsCount, const int \_columnsCount, const T a);  TMatrix(const int \_columnsCount, const TVector<T> \*\_columns);  TMatrix(const int \_columnsCount, const TVector<T>& \_column);  TMatrix(const TMatrix<T>& obj);  TMatrix(const TVector<T>& vect);  ~TMatrix();  T GetElem(const int row, const int column) const;  int GetRowsCount() const;  int GetColumnsCount() const;  TVector<T> GetVector(int column) const;  void SetElem(const int row, const int column, T a);  void SetMatrix(const int \_columnsCount, const TVector<T> \*\_columns);  void Transp();  bool operator==(const TMatrix<T>& m);  TMatrix<T>& operator=(const TMatrix<T>& m);  TVector<T>& operator[](const int indx);  TMatrix<T> operator+(const TMatrix<T>& m);  TMatrix<T> operator\*(const TMatrix<T>& m);  TMatrix<T> operator\*(const T a);  TVector<T> operator\*(const TVector<T>& v);    TMatrix<T> operator-(const TMatrix<T>& m);    friend ostream& operator<<(ostream& t, const TMatrix<T>& m) {  if(m.GetColumnsCount() == 0)  throw "Error: cannot print 0-matrix";  for(int i = 0; i < m.GetRowsCount(); i++) {  for(int j = 0; j < m.GetColumnsCount(); j++) {  t << m.GetElem(i, j) << ' ';  }  t << '\n';  }  return t;  }  friend istream& operator>>(istream& t, const TMatrix<T>& m) {  if(m.GetColumnsCount() == 0)  throw "Error: cannot write in 0-matrix";  for(int i = 0; i < m.GetRowsCount(); i++) {  for(int j = 0; j < m.GetColumnsCount(); j++) {  cout << "Enter (" << j << ", " << i << ") element:" << endl;  t >> m[j][i];  }  }  return t;  }  };  //####################### Конструкторы #######################  template <class T>  TMatrix<T>::TMatrix() {  rowsCount = 0;  columnsCount = 0;  columns = 0;  }  template <class T>  TMatrix<T>::~TMatrix() {  if(columns != 0) {  delete [] columns;  columns = 0;  }  }  template <class T>  TMatrix<T>::TMatrix(const int \_rowsCount, const int \_columnsCount, const T a) {  if(\_rowsCount == 0 || \_columnsCount == 0) {  TMatrix();  return;  }  rowsCount = \_rowsCount;  columnsCount = \_columnsCount;  columns = new TVector<T>[columnsCount];  for(int i = 0; i < columnsCount; i++) {  columns[i] = TVector<T>(rowsCount, a);  }  }  template <class T>  TMatrix<T>::TMatrix(const int \_columnsCount, const TVector<T> \*\_columns) {  if(\_columnsCount == 0 || \_columns == 0) {  TMatrix();  return;  }  columnsCount = \_columnsCount;  rowsCount = \_columns[0].GetLen();  columns = new TVector<T>[columnsCount];  for(int i = 0; i < columnsCount; i++) {  columns[i] = \_columns[i];  }  }  template <class T>  TMatrix<T>::TMatrix(const int \_columnsCount, const TVector<T>& \_column) {  if(\_columnsCount == 0 || \_column.GetLen() == 0) {  TMatrix();  return;  }  columnsCount = \_columnsCount;  rowsCount = \_column.GetLen();  columns = new TVector<T>[columnsCount];  for(int i = 0; i < columnsCount; i++) {  columns[i] = \_column;  }  }  template <class T>  TMatrix<T>::TMatrix(const TMatrix<T>& obj) {  if(obj.columnsCount == 0) {  TMatrix();  return;  }  columnsCount = obj.columnsCount;  rowsCount = obj.rowsCount;  columns = new TVector<T>[columnsCount];  for(int i = 0; i < columnsCount; i++) {  columns[i] = obj.columns[i];  }  }  template <class T>  TMatrix<T>::TMatrix(const TVector<T>& vect) {  if(vect.GetLen() == 0) {  TMatrix();  return;  }  columnsCount = 1;  rowsCount = vect.GetLen();  columns = new TVector<T>[1];  columns[0] = vect;  }  //####################### Методы #######################  template <class T>  int TMatrix<T>::GetRowsCount() const {  return rowsCount;  }  template <class T>  int TMatrix<T>::GetColumnsCount() const {  return columnsCount;  }  template <class T>  T TMatrix<T>::GetElem(const int row, const int column) const {  if(row < 0 || row >= rowsCount || column < 0 || column >= columnsCount)  throw "Error: run out of matrix range";    return columns[column][row];  }  template <class T>  TVector<T> TMatrix<T>::GetVector(int column) const {  if(column < 0 || column >= columnsCount)  throw "Error: run out of matrix range";  return columns[column];  }  template <class T>  void TMatrix<T>::SetElem(const int row, const int column, T a) {  if(row < 0 || row >= rowsCount || column < 0 || column >= columnsCount)  throw "Error: out of matrix range";  columns[column][row] = a;  return;  }  template <class T>  void TMatrix<T>::SetMatrix(const int \_columnsCount, const TVector<T> \*\_columns) {  if(\_columnsCount == 0 || \_columns == 0) {  columnsCount = 0;  rowsCount = 0;  if(columns != 0) {  delete [] columns;  columns = 0;  }  }  columnsCount = \_columnsCount;  rowsCount = \_columns[0].GetLen();  if(columns != 0) {  delete [] columns;  columns = 0;  }  columns = new TVector<T>[columnsCount];    for(int i = 0; i < columnsCount; i++) {  columns[i] = \_columns[i];  }  }  template <class T>  void TMatrix<T>::Transp() {  if(columns == 0) {  return;  }  int tm;  tm = columnsCount;  columnsCount = rowsCount;  rowsCount = tm;  TVector<T> \*temp = new TVector<T>[columnsCount];  for(int i = 0; i < rowsCount; i++) {  for(int j = 0; j < columnsCount; j++) {  temp[i][j] = columns[j][i];  }  }  delete [] columns;  columns = new TVector<T>[columnsCount];  for(int i = 0; i < columnsCount; i++) {  columns[i] = temp[i];  }  return;  }  //####################### Операторы #######################  template <class T>  bool TMatrix<T>::operator==(const TMatrix<T>& m) {  if(columnsCount == m.GetColumnsCount() && rowsCount == m.GetRowsCount()) {  for(int i = 0; i < columnsCount; i++) {  if(columns[i] != m.GetVector(i))  return false;  }  return true;  }  return false;  }  template <class T>  TMatrix<T>& TMatrix<T>::operator=(const TMatrix<T>& m) {  if(m.GetColumnsCount() == 0 || m.GetRowsCount() == 0)  throw "Error: matrix=0-matrix";    columnsCount = m.GetColumnsCount();  rowsCount = m.GetRowsCount();  if(columns != 0) {  delete [] columns;  columns = 0;  }  columns = new TVector<T>[columnsCount];  for(int i = 0; i < columnsCount; i++) {  columns[i] = m.GetVector(i);  }  return \*this;  }  template <class T>  TVector<T>& TMatrix<T>::operator[](const int indx) {  if(indx < 0 || indx >= columnsCount)  throw "Error: run out of matrix range";  return columns[indx];  }  template <class T>  TMatrix<T> TMatrix<T>::operator+(const TMatrix<T>& m) {  if(m.GetColumnsCount() != columnsCount || m.GetRowsCount() != rowsCount)  throw "Error: different matrix size in +";    TMatrix<T> temp(columnsCount, rowsCount, 0);  for(int i = 0; i < rowsCount; i++) {  for(int j = 0; j < columnsCount; j++) {  temp[j][i] = this->GetElem(i, j) + m.GetElem(i, j);  }  }  return temp;  }  template <class T>  TMatrix<T> TMatrix<T>::operator\*(const T a) {  TMatrix<T> temp = this;  for(int i = 0; i < columnsCount; i++) {  temp[i] = temp[i]\*a;  }  return temp;  }  template <class T>  TMatrix<T> TMatrix<T>::operator-(const TMatrix<T>& m) {  if(m.GetColumnsCount() != columnsCount || m.GetRowsCount() != rowsCount)  throw "Error: different matrix size in +";  TMatrix<T> temp(columnsCount, rowsCount, 0);  for(int i = 0; i < columnsCount; i++) {  temp[i] = this->GetVector(i) - m.GetVector();  }  return temp;  }  template <class T>  TMatrix<T> TMatrix<T>::operator\*(const TMatrix<T>& m) {  if(columnsCount != m.GetRowsCount())  throw "Error: cannot multiply matrix (invalid size)";    TMatrix<T> temp(m.GetColumnsCount(), rowsCount, 0);  for(int k = 0; k < rowsCount; k++) {  for(int j = 0; j < m.GetColumnsCount(); j++) {  for(int i = 0; i < columnsCount; i++) {  temp[j][k] += this->GetElem(k, i)\*m.GetElem(i, j);  }  }  }  return temp;  }  template <class T>  TVector<T> TMatrix<T>::operator\*(const TVector<T>& v) {  if(v.GetLen() != columnsCount)  throw "Error: cannot muiltiply matrix\*vector (invalid size)";    TVector<T> temp(rowsCount);  for(int i = 0; i < rowsCount; i++) {  for(int j = 0; j < v.GetLen(); j++) {  temp[i] += this->GetElem(i, j) \* v.GetCoord(j);  }  }  return temp;  } |

1. В настоящей работе под «длиной» вектора подразумевается длина набора его координат (их количество). [↑](#footnote-ref-1)