МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт кибернетики

Кафедра проблем управления

КУРСОВАЯ РАБОТА

по дисциплине

«Объектно-ориентированное программирование»

Тема курсовой работы

«Моделирование операций тензорной алгебры»

Студент группы КМБО-03-19 *Кретов О.А.*

Руководитель курсовой работы *Петрусевич Д.А.*

Работа представлена к защите «__»_____20___ г. *(подпись студента)*

«Допущен к защите» «__»_____20___ г. *(подпись руководителя)*

МОСКВА — 2020

МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт кибернетики

Кафедра проблем управления

Утверждаю
Заведующий
кафедрой_____*М.П.Романов*

«____» _____20__г.

ЗАДАНИЕ
на выполнение курсовой работы
по дисциплине «Объектно-ориентированное программирование»

Студент *Кретов О.А.* Группа *КМБО-03-19*

1. **Тема: «Моделирование операций тензорной алгебры»**

2. **Исходные данные:**
Реализовать класс для коэффициентов тензора (или «обертку» для массива, хранящего его коэффициенты)
Перегрузить операторы ввода/вывода в поток >>, << так, чтобы можно было вводить и выводить параметры тензора из файла и из консоли
Реализовать операции транспонирования, свертки, умножения тензоров
Реализовать операции симметризации и антисимметризации тензора

3. **Перечень вопросов, подлежащих разработке, и обязательного графического материала:**
Продемонстрировать операции над тензорами

4. **Срок представления к защите курсовой работы: до** « 31» декабря 2020 г.

Задание на курсовую работу выдал	«01» октября 2020 г.	_____	*(_____)*
Задание на курсовую работу получил	«01» октября 2020 г.	_____	*(_____)*

Оглавление

Глава 1. Теоретический обзор

1.1 Основные определения

Цель моей работы является моделирование операций тензорной алгебры, для этого необходима соответствующая теория. Само по себе тензорное исчисление имеет огромный прикладной характер, современную физику невозможно представить без него. Оно упрощает уравнение и делает взаимодействие с ними в разы легче и понятней. Но все же, когда время доходит до поиска конкретных величин, вычисления могут занять много времени и в них легко запутаться, если быть не достаточно внимательным.

Именно для этого, для прикладных вычислений и написан мой код, он создан, чтобы сэкономить время и получить гарантированно правильный результат.

Главным действующим лицом является такой математический объект как тензор, его определение:

Тензором типа (p, q) ранга p+q называется объект, задаваемый набором чисел $T^{i_1 i_2 \dots i_p}_{j_1 j_2 \dots j_q}$ в произвольной системе координат $(x^1, \dots, x^n)$, числовая запись которого зависит от системы координат по следующему закону: если

$x^i = x^i(z^1, \dots, z^n)$, $z^j = z^j(x^1, \dots, x^n)$; $z\big(x(z)\big) = z$, то имеет место формула

$$'T^{i_1 i_2 \dots i_p}_{j_1 j_2 \dots j_q} = \sum_{(r,l)} T^{i'_1 i'_2 \dots i'_p}_{j'_1 j'_2 \dots j'_q} \, c^{i_1}_{i'_1} c^{i_2}_{i'_2} \dots c^{i_p}_{i'_p} c^{j'_1}_{j_1} c^{j'_2}_{j_2} \dots c^{j'_q}_{j_q} \quad (1.1)$$ [1].

Где c – матрица перехода к другому базису.

Компоненты тензора – это его значения на базисных векторах и ковекторах(линейный функционал). Компоненты тензора типа (p, q) имеют p верхних и q нижних индексов. Значениями компонент являются вещественные числа.

При этом, вектор принято записывать в столбец, а ковектор в строку.

Компоненты целесообразно структурировать в виде многомерной таблице $n \times n \dots \times n$, где n размерность пространства, а количество сомножителей равно рангу тензора.

В рамках данного определения тензор – это массив компонент + закон преобразования компонент при замене базиса.

Далее знак суммы писаться практически не будет, он будет подразумеваться согласно правилу Эйнштейна, если в записи встречаются два одинаковых индекса, один верхний, а другой нижний, то по ним происходит суммированию.

Простые примеры:

Скаляр, тип (0, 0)

Вектор, тип (1,0)

Ковектор, тип (0, 1)

Билинейная форма, тип (0, 2)

Линейный оператор, тип (1, 1)

Метрический тензор – это симметричный тензор g_{ij}типа (0,2), посредством которого задается скалярное произведение. С помощью него можно каждому вектору сопоставить ковектор и наоборот:

$x_i = g_{ij}x^j$ (1.2) [1].

1.2 Операции тензорной алгебры

Сложение тензоров одного типа (p, q) в линейном пространстве V, на выходе получается новый тензор того же типа:

$C^{i_1 i_2 \dots i_p}_{j_1 j_2 \dots j_q} = A^{i_1 i_2 \dots i_p}_{j_1 j_2 \dots j_q} + B^{i_1 i_2 \dots i_p}_{j_1 j_2 \dots j_q}$ (1.3) [1].

Тензорное произведение, на выходе получается тензор суммарного типа:

$A^{i_1 i_2 \dots i_p}_{j_1 j_2 \dots j_q} * B^{i\prime_1 i\prime_2 \dots i\prime_k}_{j\prime_1 j\prime_2 \dots j\prime_l} = C^{i_1 i_2 \dots i_p i\prime_1 i\prime_2 \dots i\prime_k}_{j_1 j_2 \dots j_q j\prime_1 j\prime_2 \dots j\prime_l}$ (1.4) [1].

Операция свертки переводит тензор типа (p, q) в тензор типа (p - 1, q - 1), соответственно понижая его ранг на 2:

$\sum_{l_k} T^{i_1 i_2 \dots l_k \dots i_p}_{j_1 j_2 \dots l_k \dots j_q} = T^{i_1 i_2 \dots l_k \dots i_p}_{j_1 j_2 \dots l_k \dots j_q}$ (1.5) [1].

То есть, происходит суммирование по паре индексов.

Поднятие и опускание индексов, осуществляются это с помощью метрического тензора, то есть с помощью g_{ij} и обратного к нему g^{ij}, в результате получается тензор того же ранга, но уже иного типа, если индекс поднимается, то новый тензор имеет тип (p + 1, q - 1), а если опускается, то соответственно (p - 1, q + 1):

$g_{i_k j_{q+1}} * T^{i_1 i_2 \dots i_k \dots i_p}_{j_1 j_2 \dots j_q} = T^{i_1 i_2 \dots i_{p-1}}_{j_1 j_2 \dots j_q j_{q+1}}$ (1.6) [1].

Поднятие индекса происходит аналогично.

Перестановка индексов одного типа:

$T^{i_1 i_2 \dots i_l \dots i_k \dots i_p}_{j_1 j_2 \dots j_q} \rightarrow T^{i_1 i_2 \dots i_k \dots i_l \dots i_p}_{j_1 j_2 \dots j_q}$ (1.7) [1].

Перестановка нижних индексов происходит аналогично.

Любая перестановка индексов, в том числе поднятие и опускание, при котором меняется их порядок – транспонирование.

Симметризация, на выходе получается симметричный тензор, то есть тензор, который остается равным самому себе при перестановке двух индексов одного типа:

$$P_{i_1 i_2 \dots i_q} = T_{(i_1 i_2 \dots i_q)} = \frac{1}{q!} \sum_{(\sigma)} T_{\sigma(i_1 i_2 \dots i_q)} \quad (1.8)\ [1].$$

Сумма по всем возможны перестановкам, соответственно q! слагаемых.

Антисимметризация, на выходе получается антисимметричный тензор, то есть тензор, у которого компоненты меняют знак при перестановке двух индексов одного типа:

$$P_{i_1 i_2 \dots i_q} = T_{[i_1 i_2 \dots i_q]} = \frac{1}{q!} \sum_{(\sigma)} (-1)^{\sigma} T_{\sigma(i_1 i_2 \dots i_q)} \quad (1.9)\ [1].$$

Где σ – четность перестановки.

Глава 2. Практическая реализация операций тензорной алгебры

2.1 Индексация компонент тензора

Из первой главы известно, что каждой компоненте соответствует p верхних и q нижних индексов, вместе они называются координатами(коэфициентами) тензора. Для индексации компонент был создан класс Index.

Зная ранг и размерность пространства можно вычислить количество компонент тензора по формуле:

n^{rang} (2.1)

где n – размерность пространства, а $rang$ - ранг тензора. Соответственно, координаты компоненты можно представить как $rang$- значное число в n-значной системе исчисления, только вместо 0 будет 1, и, соответственно, все остальные числа тоже будут на 1 больше.

Пример, $n = 2, rang = 3$: 111, 112 ,121, 122, 211, 212, 221, 222.

В себе класс хранит ранг соответствующего тензора, количество верхних индексов, размерность пространства, итератор (число, соответствующее порядку индекса, для 111 это 0, для 112 – 1 и так далее), массив **vector<int> ind**, который в определенном порядке хранит координаты компоненты. Правило хранения: первыми иду верхние индексы, слева-направо, за ними нижние, также слева-направо.

$T^{i_1 i_2 \dots i_p}_{j_1 j_2 \dots j_q} \rightarrow \{i_1, i_2, \dots i_p, j_1, j_2, \dots j_q\}$ (2.2)

Пробег по индексам происходит с помощью операции инкрементирования **operator++()**, например: 111 перейдет в 112 и так далее.

Также существуют другие операции и методы:

begin() – возвращает наименьший индекс;

getElem() – возвращает вектор ind, т.е. координаты тензора;

IterIn() – возвращает итератор;

transp(int a, int b) – принимает номера позиций двух индексов и меняет их местами.

operator+(Index a) – принимает индексы одинаковой размерности, и возвращает уже склеенный индекс суммарного ранга, первыми идут верхние индексы первого тензора, за ними верхние индексы второго тензора, далее нижние

индексы первого тензора, и в конце нижние индексы второго тензора, этот оператор понадобиться при перемножении двух тензоров.

operator[](int i) – возвращает i-ый элемент вектора ind.

Тип контейнера – vector, был выбран как оптимальный контейнер, с уже готовым набором полезных инструментов, далее он будет применятся везде, где только можно.

2.2 Реализация класса для компонентов тензора

Поскольку класс Index позволяет естественным образом взаимодействовать с тензором, то пришло время написать сам тензор, класс Tensor.

Важными характеристиками тензора являются: размерность соответствующего пространства dimV(далее, везде размерность пространства будет записана таким образом), ранг, количество верхних upperIndex и нижних индексов lowerIndex, помимо этого, тензор хранит, координаты(коэффициенты) компонентов **vector<Index> indexId** и их значения **vector<long double> tensor**. Также полезным оказалось хранить количество элементов quantiteElems, вычисляемых по формуле (2.1).

При создании Tensor – на вход поступает размерность, количество верхних и нижних индексов, векторы indexId и tensor заполняются автоматически, в indexId элементы расположены в порядке возрастания, ранг и количество элементов также вычисляются на основе входной информации, tensor изначально заполнен нулями. Связаны вектора с помощью индекса массива, т.е. первому элементу из indexId соответствует первый элемент из tensor, и так далее.

Некоторые методы которыми обладает класс Tensor:

addElems(vector<long double> items) – принимает на вход вектор, значения которого принимает tensor.

getElem(vector<long double> cords) – принимает координаты vector, на выходе возвращает соответствующее значение компоненты тензора.

setElem(Index cords, double item) – принимает координаты и вещественное значение, которое вставляется в соответствующею компоненту, также есть перегрузка, вместо Index-а на вход поступает вектор – координаты компоненты, которую нужно заменить.

Operator[](int i) – возвращает **tensor[i]**.

Rang(), **UperIndx()**, **LowIndx()** – возвращают ранг, количество верхних и нижних индексов соответственно.

2.3 Операции произведения и сложения

Операция сложения **operator+(Tensor)** ничего сложного из себя не представляет, нужно всего лишь покомпонентно сложить значения двух тензоров, на выходе получается тензор того же ранг и с тем же набором координат. Само собой, также был написан оператор **operator-(Tensor T)**.

Куда сложнее дело обстоит с тензорным произведением **operator*(Tensor t)**. В результате такой операции получается тензора уже нового ранга, суммарного, и количество верхних и нижних индексов тоже получается из суммы перемножаемых тензоров. Здесь на помощь приходит оператор **operator+(Index a)** склеивания координат из класс Index. Сначала необходимо перемножить значения всех возможных компонент, а их координаты coord склеить вместе, после чего вставить значение в соответствующую компоненту уже нового, результирующего тензора res. Реализовано это было следующим образом:

```
Index coord(dimV,newRang, newUpIndex); // координаты нового тензора

for (int i = 0; i < quantityElems; i++)
{
        for (int j = 0; j < tens.quantityElems; j++)
         {
           elem = tensor[i] * tens.tensor[j]; //значения перемножаются
           coord = indexId[i] + tens.indexId[j]; //их координаты склеиваются
           res.setElem(coord, elem); //вставка значения в соответствующую
        }                                  // компоненту
}
return res; //результат
```

Код 1. Тензорное произведение.

Значения компонент перемножаем, их координаты склеиваем, после чего вставляем значение в соответствующую компоненту.

Также необходима возможность умножить тензор на скаляр, но поскольку скаляр – это тензор типа (0, 0), то такая возможность уже реализована, но из соображений удобства была написана отдельная операция **operator*(long double item)**, которая возвращает тензор, чьи элементы вектора tensor покомпонентно умножены на число item.

2.4 Свертка

В отличии от предыдущих операций, операция свертки **contraction(int m, int n)** уменьшает ранг тензора, а именно тензор типа (p, q) переходи в тензор типа (p-

1, q-1). На вход поступают два целочисленные числа, по которым нужно провести свертку, первое число – номер индекса сверху, второе – номер индекса снизу. Свертка выглядит следующим образом

$$\sum_{l_k} T^{i_1 i_2 \dots l_k \dots i_p}_{j_1 j_2 \dots l_k \dots j_q} = T^{i_1 i_2 \dots l_k \dots i_p}_{j_1 j_2 \dots l_k \dots j_q}, l_k \in [1, \dim V] \quad (2.3)$$

Для начала был создан новый тензор res соответствующего типа, к нему нужно прибавить другие тензоры в количестве dimV штук, строятся эти тензоры следующем образом: берется исходный тензор, из него выбираются только те компоненты, индексы которых в определённых местах сверху и снизу совпадают и равны некоторому числу $\in [1, dimV]$, оно зависит от итерации. Значения этих самых компонент, в порядке возрастания координат, помещаются во временный тензор elem типа (p-1, q-1) с помощью метода **getElem()**, после заполнения который прибавляется к res, после чего операция вновь повторяется, пока l_k не пройдется по всем значения, в коде ниже l_k = i.

```
int newRang = rang - 2; //новый ранг
int newUpper = upperIndex - 1; //новое кол-во верхних индексов
int newLower = lowerIndex - 1; // новое кол-во нижних индексов

Tensor res(dimV, newUpper, newLower); // результирующий тензор
Tensor elem(dimV, newUpper, newLower); // временный тензор
Index newIndx(dimV, newRang, newUpper); // координаты для res

for (int i = 0; i < dimV; i++)
{
        newIndx = newIndx.begin();//Координаты переходят в начало
        for (int j = 0; j < quantityElems; j++)
        {
          if (indexId[j].getElem()[m-1] == i+1 and
          indexId[j].getElem()[upperIndex +n - 1] == i+1)
          {//Если значения верхних и нижних индексов в нужной позиции
            //совпадает, то значение переходит во временный тензор
                elem.setElem(newIndx.getElem(), tensor[indexId[j].IterIn()]);
          }
        }
        res += elem;
}
return res;
```

Код 2. Основная часть метода contraction.

2.5 Поднятие и опускание индексов

Благодаря уже написанным операциям выше, операция опускания/поднятия индекса indexLow/ indexUp реализуется достаточно просто, в итоге получается тензор того же ранга, но уже типа (p-1, q+1)/ (p+1, q-1) Я разберу только операцию опускания индекса, так как поднятие происходит аналогичным способом.

IndexLow(int ind) – на вход поступает целочисленное число – номер верхнего индекса, т.е. тот индекс, который необходимо опустить, счет индексов идет слева-направо, опускаемы индекс займет первое место среди нижних индексов.

Сначала необходимо умножить метрический тензор g_{ij} на исходный тензор, после чего провести свертку contraction(ind, 1) результата, в итоге получается тензор необходимого типа (p-1, p+1).

При поднятии, метрический тензор заменяется обратный к нему g^{ij}, а свертка происходит следующим образом: contraction(1, ind).

2.6 Перестановка индексов одного типа

Перестановка индексов не меняет ни ранга, ни количество верхних/нижних индексов, все что нужно сделать, так это переставить значения в векторе tensor. Если быть точным, то происходит не какая-то перестановка, а транспозиция. Делается эта операция аналогично перестановке двух элементов в массиве, по факту это оно и есть. Здесь на помощь приходит метод из класса Index – **transp(int a, int b)**, он меняет два индекса местами. Для перестанови верхних индексов код выглядит следующим образом.

```
Tensor transposUp(int a, int b)
{
	Tensor res = *this; // результирующий тензор
	Index tempInd(dimV, rang, upperIndex); // временные координаты
	for (int i = 0; i < quantityElems; i++)
	{
	   tempInd = indexId[i].transp(a,b); //индексы меняются местами
	   res.tensor[i] = tensor[tempInd.IterIn()];// закладка нового значения
	}
	return res;
}
```

Код 3. Метод transposUp, переставляет два верхних индекса.

Создается временный индекс tempInd, а далее выполняется пробег по компонентам исходного тензора (от начала до конца), в результирующий тензор кладется значение, равное значению компоненты исходного тензора, чьи координаты отличаются от компоненты результирующего тензора соответствующей транспозицией.

Для перестановки нижних индексов (метод **transposLow()**) реализация аналогична.

2.7 Симметризация

Симметризации подлежат только тензоры типа (0, k), поэтому если тензор не соответствует типу, он будет к нему приводится с помощью опускания верхних индексов.

После того, как тензор будет соответствовать типу – необходимо выполнить все возможные перестановки его нижних индексов, это все сложить и разделить на k!. Но поскольку перестановка может быть результатом множества транспозиций, то необходимо уметь разбивать любую перестановку на эти самые транспозиции, которые приводят к этой самой перестановке.

Например, перестановка (312) (такая перестановка говорит о том, что на первом месте стоит 3 индекс и так далее), чтобы получить ее из (123) необходимо выполнить ряд транспозиций: (31), (23) – в самих транспозиций номера позиций, а не значения индексов. Т.е. индекс стоящий на 3 месте поменять с индексом на 1 месте, а затем индекс на 2 месте поменять с индексом на 3 месте.

Для такой цели была написана новая функция:

vector<vector<int>> transpos(vector<int> index) – на вход поступает некая перестановка в виде вектора, а на выходе получается вектор транспозиций(вектор из векторов размерности 2), из которых получается исходная перестановка.

Чтобы получить вектор всех возможных перестановок transLow(вектор из вектор размерности равной рангу тензора) – необходимо создать вектор из элементов типа Index, где размерность элементов Index равна рангу (так как не интересно, какие значения принимает индекс, а лишь интересно их количество и его местоположение в ряду), при этом необходимо исключить координаты с повторяющимися индексами.

После всех приготовлений сама симметризация реализуется достаточно просто, берется исходный тензор и некая перестановка, эта перестановка раскладывается в транспозиции из n штук, после чего n раз происходит перестановка индексов с

помощью метода **transposLow()**, в результате всех транспозиций – результат прибавляется к результирующему тензору, и так происходит до тех пор, пока не реализуются все перестановки. Результирующий тензор res умножается на 1/k! и отдается.

```
for (int k = 0; k < transLow.size(); k++)
{
        temp2 = temp;  //temp – исходный тензор с опущенными индексами
        trLow = transpos(transLow[k]); //разбитая на транспозиции
        for (int j = 0; j < trLow.size(); j++)                    //перестановка
        {
          temp2 = temp2.transposLow(trLow[j][0], trLow[j][1]); //транспозиция
        }
        res += temp2;
}
double long  e = factorial(rang); //факториал от ранга
res = res * (1/e);
return res;
```

Код 4. Часть метода simetrisation.

2.8 Антисимметризация

Антисимметризация уже практически имеет решение, ведь она аналогична симметризации, но чтобы все-таки реализовать эту операцию необходимо уметь определять четность перестановки, согласно формуле (1.9).

Перестановка, содержащая четное количество инверсий, называется четной, в противном случае нечетной.

Если какая-нибудь пара элементов перестановки расположена в ней так, что элементы с большим номером стоят раньше элемента с меньшим номером, то говорят, что эти элементы образуют **инверсию**. [3]

По итогу, для определения четности была написана функция **int trans_parity(vector<int> trans)**, которая принимает вектор(перестановку), а возвращает 1 или -1, соответственно, если перестановка четная или нечетная.

По итогу новый метод **antisemetr()** мало чем отличается от метода **simetrisation()**, разве что некоторые слагаемы прибавляются с соответствующим их перестановке знаком. В связи с этим, чтобы не дублировать код – был создан приватный метод **simetrAssist()**, который принимает либо 0, если требуется симметризация, либо 1, если требуется антисимметризация, возвращает он уже (анти)симметричный тензор. А в свою очередь, методы **antisemetr()** и

simetrisation() просто используют метод **simetrAssist()** с соответствующим аргументом и возвращают результат.

2.9 Результаты запуска кода

Чтобы провести какие-либо операции над тензором, сначала необходимое его получить – для этого был перегружен оператор ввода **operator<<()**, как для консоли, так и для текстового документа. Сперва приложение просит пользователя выбрать, как будет совершаться ввод, из консоли или из файла, для этого нужно ввести 0 или 1 соответственно, так же есть уже готовый вариант, вшитый в код, демонстрационный, необходимо ввести 2. Если введено что-то иное, приложение вновь повторит запрос. Далее оно запрашивает у пользователя размерность пространства, количество верхних и нижних индексов, после чего просит ввести значения тензора. Если размерность указана неверно, т.е. на вход поступило значение не правильного типа, то программа завершится и попросит пользователя запустить программу снова или проверить текстовый документ. Если же уже после размерности был введен объект неправильного типа, то после него чтение прекращается, а все значения, которые идут следом, включая его, по умолчанию становятся равными 0.

```
*Исходный тензор при выполнении операций остается неизменным.
Выберите, как вы хотите получить тензор:
Ввести самому - 0    Загрузить из input.txt - 1    Демонстрация - 2
0
Введите размерность пространства, кол-во вверхних и нижних индексов
3 1 1
Теперь вводите значения:
1 2 3 4 5 6 7 8 9
```

Рисунок 1. Пример ввода данных тензора в консоль.

```
*Исходный тензор при выполнении операций остается неизменным.
Выберите, как вы хотите получить тензор:
Ввести самому - 0    Загрузить из input.txt - 1    Демонстрация - 2
0
Введите размерность пространства, кол-во вверхних и нижних индексов
3 2 ц
Теперь вводите значения:
Вы ввели:
0  0  0
0  0  0
0  0  0
```

Рисунок 2. Пример не правильного ввода в консоль.

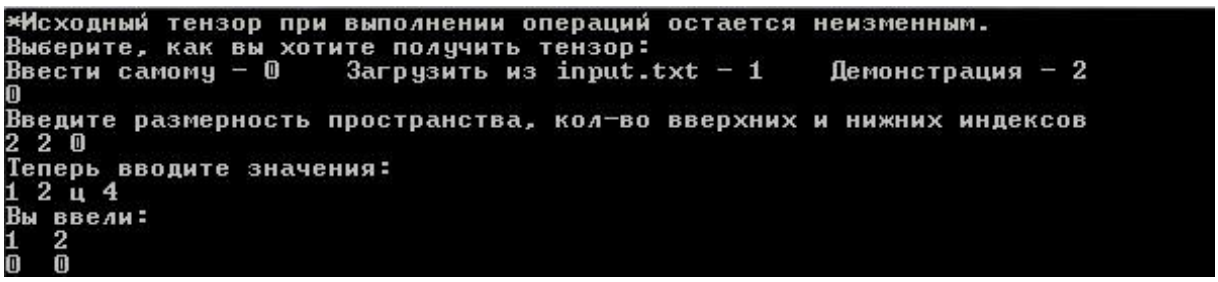

Рисунок 3. Пример неправильного ввода на этапе ввода значений.

В зависимости от характеристик тензора – будут продемонстрированы только те операции, которые можно с ним провести, например, нельзя свернут вектор или ковектор, а также получить из них (анти)симметричный тензор.

С чтением из файла все работает аналогично.

Для вывода были перегружены операторы **operator>>()**, чтобы можно было выводить тензор в консоль и записывать его в текстовый документ.

Помимо всего, также существует защита от неправильного ввода характеристик тензора, если они не будут положительными, а размерность еще и меньше 1, то программа среагирует и на это.

Код написанный в main() предназначен для демонстрации работы классов, методов и функций, а не как аналог калькулятора, где вместо обычных вещественных чисел – тензоры.

Основной задачей было моделирование операций тензорной алгебры, именно их я и собираюсь продемонстрировать.

Пример 1. Вектор, тип (1,0), размерность пространства – 4.

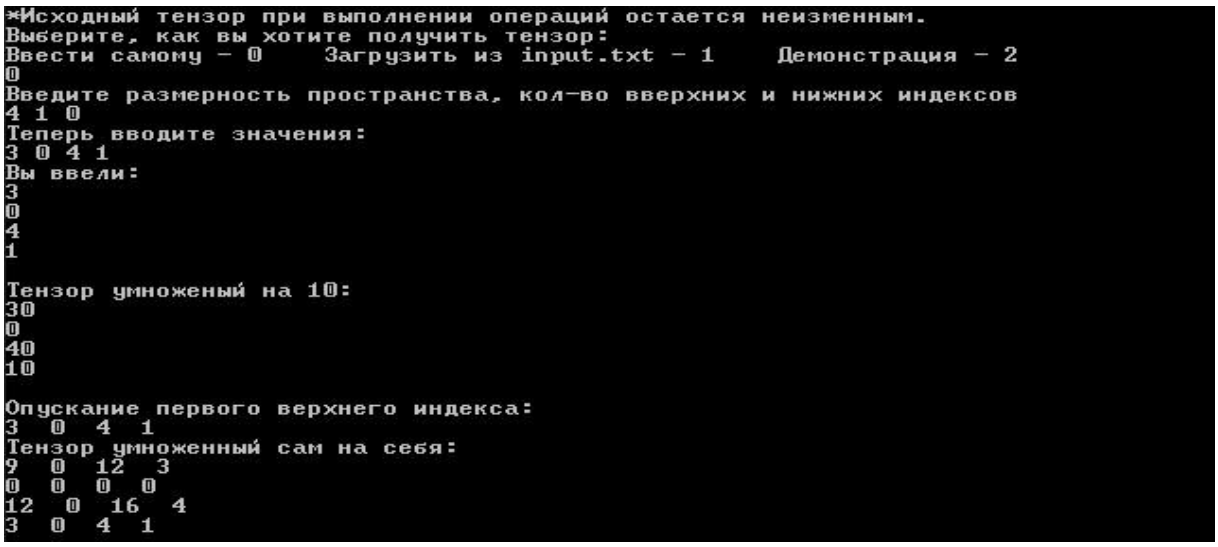

Рисунок 5. Тестовый запуск на примере вектора.

Здесь вектор - вертикальный столбец, именно так принято его записывать.

Пример 2. Билинейная форма, тип (0,2), размерность пространства – 2. В данном примере тензор считывается с текстового файла.

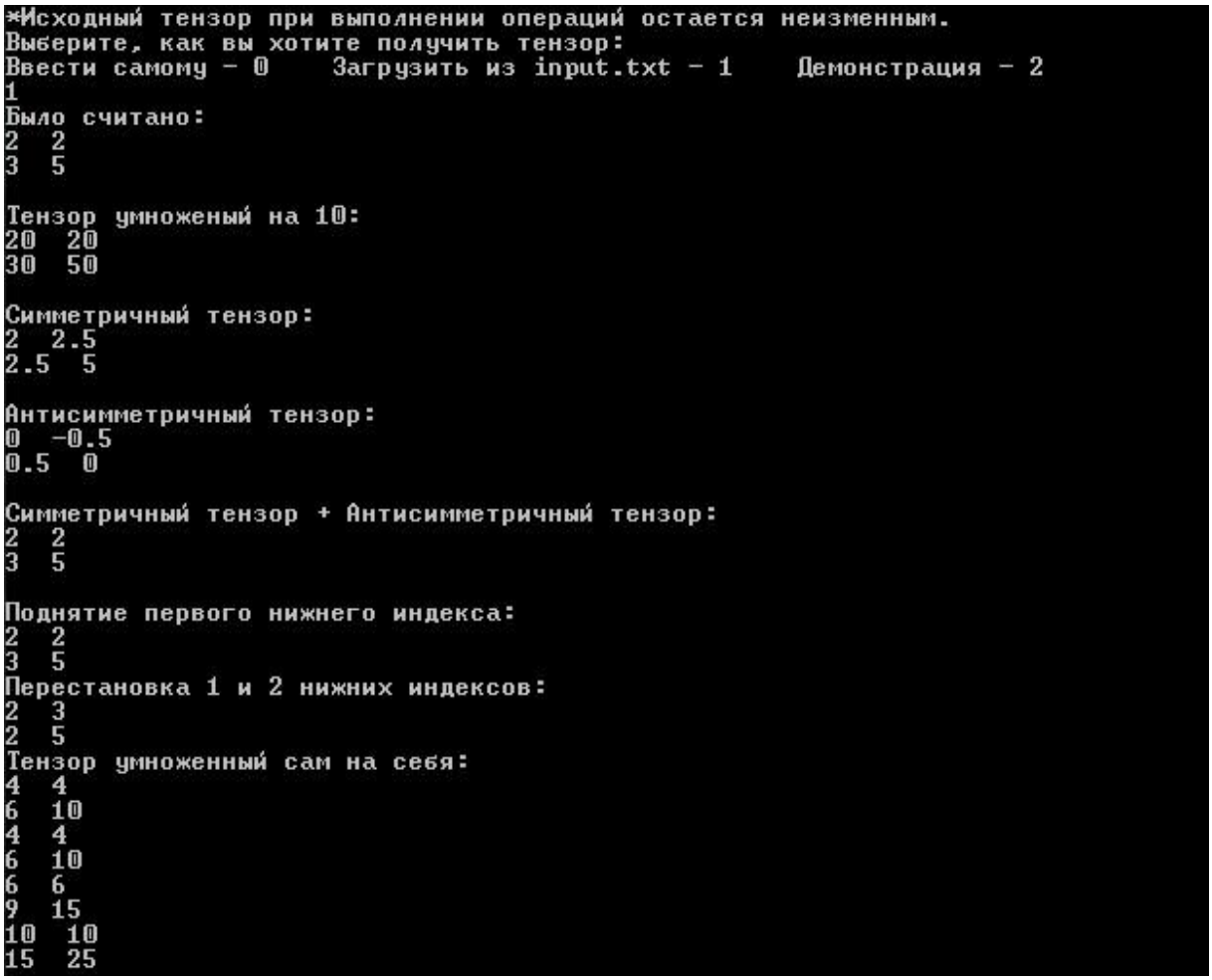

Рисунок 6. Тестовый запуск на примере билинейной формы, чтение из файла.

После каждого запуска программы все результаты записываются, но при этом, перед запуском все содержимое, что было записано ранее – удаляется.

Пример 3. Демонстрационный запуск, линейный оператор, тип (1,1), размерность – 2.

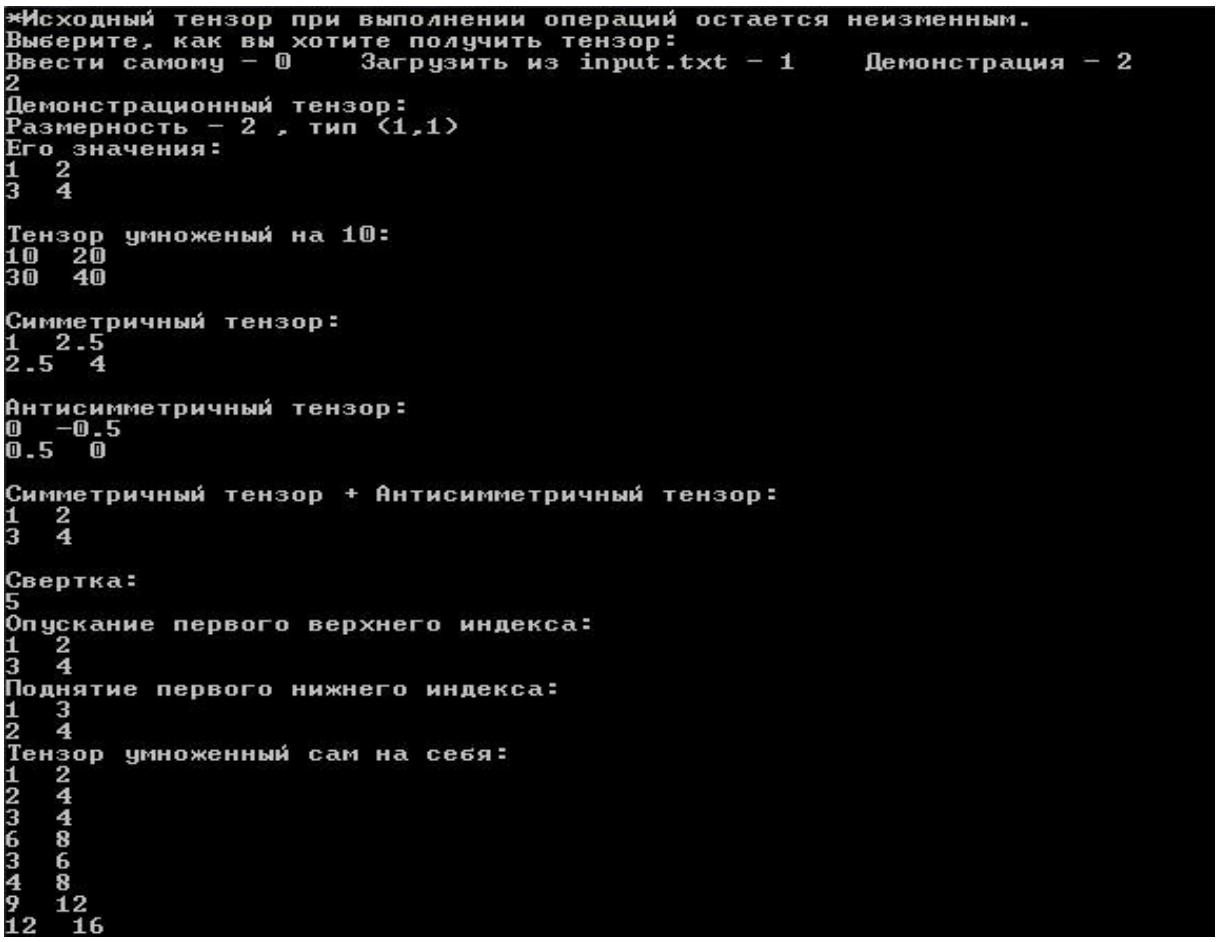

Рисунок 6. Демонстрационный запуск.

На рисунке ниже, результат демонстрационного запуска в текстовом документе.

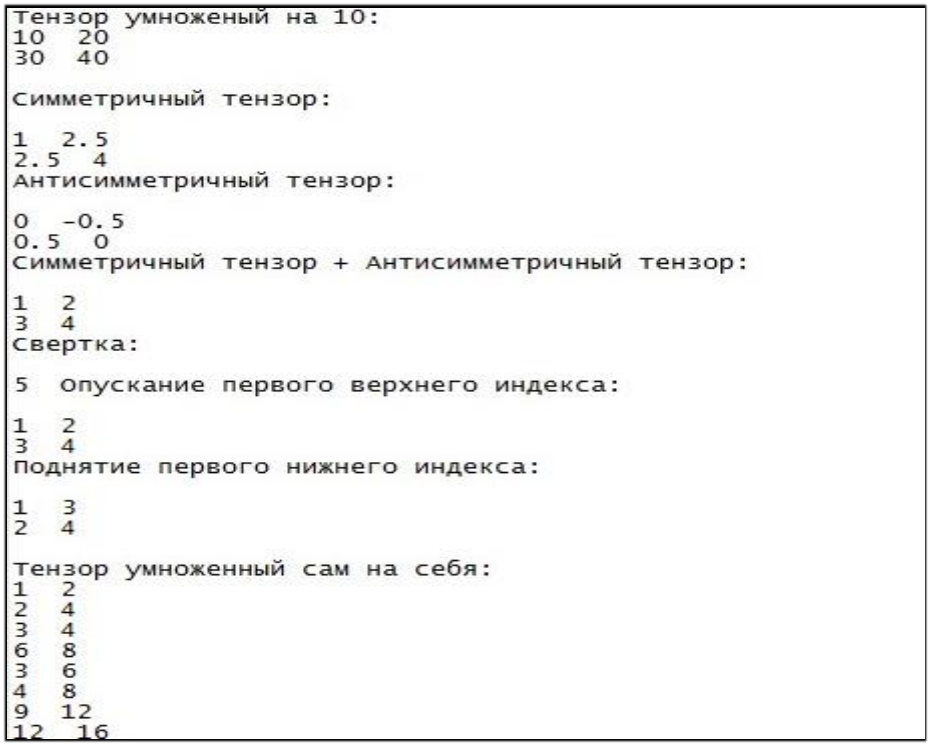

Рисунок 7. Результат демонстрационного запуска в тестовом документе.

Заключение

В данной работе:

1. Был написан класс для хранения компонентов тензора Tensor и класс Index, позволяющий естественным образом взаимодействовать с компонентами самого тензора.
2. Были реализованы следующие операции тензорной алгебры: сложение, произведение, свертка, поднятие и опускание индексов, перестановка индексов одного типа, симметризация и антимсимметризация.
3. Были перегружены операторы ввода и вывода, для удобной работы как в консоли, так и с файлами.

Моделирование операций тензорной алгебры, тема, которая слабо освещена в сети. В связи с чем, мне негде было позаимствовать идей, и реализация проходила основываясь лишь на учебниках по алгебре и тензорному анализу.

В целом, я считаю, мне удалось достичь поставленных задач, но это не значит, что тема исчерпана, здесь есть еще много места для дополнительных исследований.

Список литературы

1. Б.А. Дубровин Современная геометрия. Методы и приложения / Б.А. Дубровин, С.П. Новиков, А.Т. Фоменко. – 2-е изд., перераб. – М.: Наука. Гл. ред. физ.-мат. лит., 1986. – 760 с.
2. Головина Л.И. Линейная алгебра и некоторые ее приложения : учеб. для вузов. – 4-е изд., испр. – М.: Наука, Гл. ред. физ.-мат. лит., 1985. – 392 с.