Documentation Assignment 1

Marco Busato, 852074; Dainese Fabio, 857661

November 14, 2019

1 Tensors and assignment requests

1.1 Tensor: general definition

Tensors are geometric objects that describe linear relations between geometric vectors, scalars, and other tensors.
With respect to our field of interest - the multidimensional arrays - a tensor is just as a vector in an n-dimensional space. It is represented by a one-dimensional array of length n with respect to a given basis; any tensor with respect to a basis is represented by a multidimensional array. Tensors are specified by:

- the type they are holding;
- their rank, i.e., the number of dimensions (or indexes);
- the size of each dimension

1.2 Assignment requirements

Implement a templated library handling tensors, i.e., multidimensional arrays. The tensor class must provide various data-access functions:

1. **direct access**: the user must be able to directly access any entry of the tensor by providing all the indexes. I suggest the use of an operator ();
2. **slicing**: the user can fix one (or more) index producing a lower rank tensor sharing the same data;
3. **iterators**: the class must provide forward iterators to the full content of the tensor or to the content along any one index, keeping the other indexes fixed.

The library must also provide services for flattening two or more indexes into one and to generate sub-windows of any given tensor changing the starting-point and end-point of each index. For efficiency reason, flattening can be restricted to consecutive ranges of indexes.
The library must allow data to be shared in all slicing, flattening and windowing

operations, and also must provide a sharing construction and assignment, but must allow a way to force copying on construction and assignment. It is up to you to decide how each behavior should be invoked. Consider also what move semantics are reasonable for this type semantics.

The tensors, once created, should be unmodifiable in their dimensions (while modifiable in their content), but the library must provide for 2 different levels of compile time knowledge of the tensor's shape:

1. **No Knowledge**: there is no compile time information about rank or dimensions of the tensor, only type;
2. **Rank Only**: the tensor has rank information in its type, but no dimensional information;

The various types of tensors should be able to share data if compatible.

Implementation notes
The data should be maintained in a consecutive area. The template specification must maintain an array of the strides and width of each dimension.

- *stride*: distance in the next element of this index;
- *width*: number of entries for this index.

For example a rank 3 tensor of size (3,4,5) represented in right-major order will have strides (20,5,1) and width (3,4,5). Entry (i,j,k) will be at index (20*i+5*j+k*1) in the flat storage.

2 Tensor base structure

A tensor M is a **multi-dimensional array** composed of elements of the same type **T**. A tensor has a certain number of dimensions called **rank** r = n and each one has a specific size. We decided to adopt a row-major policy, so that the first index is the most significant one.
We also defined how users will read and write data within the tensor. The technique adopted is right measure. We decided to choose this method only for practicality reasons. Indeed, in our opinion, users will better understand how to access data.

2.1 Mathematical definition

The tensor dimensions are defined as follow:

$$d=(d_1,d_2,\ldots,d_n)$$

Where d_i represents the size of i-th dimension.
Although M is a multi-dimensional array, it is stored linearly, as a simple array A. So to access consecutive elements on an index that is not the least significant

it is not necessary to consider consecutive elements on the array, but to skip a certain number of elements observing the so called strides (Ref.1.2).
Strides are calculated in this way:

$$s=(d_2 \cdot d_3 \cdot \ldots \cdot d_n,\ d_3 \cdot \ldots \cdot d_n,\ d_{n-1} \cdot \ldots \cdot d_n,\ d_n,\ 1)$$

In a more synthetic way:

$$s_i = \begin{cases} 1 & \text{if i=n} \\ s_{i+1} \cdot d_{i+1} & \text{otherwise} \end{cases}$$

So given an element x = (x_1 , x_2,. . . , x_n), it is stored on the linear array on the position:

$$P(x) = \sum_{i=1}^{n} x_i \cdot s_i$$

To access its content:

$$M(x) = A(P(x))$$

Then to iterate elements it is necessary to define start and end pointer. The **start pointer**:

$$sp = (0,0,\ldots,0)$$

is obviously the minimum value of each index, instead the **end pointer**:

$$ep = (d_1,0,\ldots,0)$$

is like the first next element if it exists. So that $P(sp)$ = 0 and $P(ep)$= $x_1 \cdot s_1$. Thus, from now consider this way to calculate $P(x)$:

$$P(x) = sp + \sum_{i=1}^{n} x_i \cdot s_i$$

To observe, is that if r=0 the tensor is like a simple variable so that d=∅, s=∅ and sp=ep=0.
Consider also that start and end pointers are calculated in this way, on a new tensor. If users apply some operations on it, as we are going to say, it will be changed. For this reason all following operations will be referred to *sp* and *ep* and not to (0, 0, ..., 0) and (d_1 , 0, ..., 0) because they may be different.

2.2 Iterator

2.2.1 Definition

In computer programming, an iterator is an object that enables a programmer to traverse a container, particularly lists. Various types of iterators are often provided via a container's interface. Though the interface and semantics of a given iterator are fixed, iterators are often implemented in terms of the structures underlying a container implementation and are often tightly coupled to the container to enable the operational semantics of the iterator. Note that an iterator performs traversal and also gives access to data elements in a container, but does not perform iteration (i.e., not without some significant liberty taken with that concept or with trivial use of the terminology). An iterator is behaviorally similar to a database cursor.

2.2.2 Mathematical representation

An iterator permits to scan all content of a tensor. The iterator must scan elements in order, starting from the least significant index: as we chosen a row-major policy it is the last on the n-upla. The iterator we have defined can go up to infinity and does not consider sp and ep of the tensor, but when users try to access the elements, an exception is throwed. Obviously the start iterator of the tensor is:

$$si = sp{=}(si_1,\ldots,si_n)$$

and the end iterator is:

$$ei = ep{=}(ei_1,\ldots,ei_n)$$

Consider a generic iterator, represented as follow:

$$i = (i_1,\ldots,i_n)$$

When users try to move it one step forward it is necessary to increase by one the first least significant index, which must have lower value than its maximum. Starting from the last index i_n , if it has reached its last valid value (d_n - 1) it is moved to 0 and you try to increase the previous index in the same way. Otherwise it is increased by one. If the index to increase is not found, what is done is to impose the first one of its previous values plus one, as it is the next element, as if it exists.

Obviously a tensor with r = 0 **is not iterable**, indeed si = se = 0, so none of the elements are accessible.
We implement a different version of the iterator: **One Dim Iterator**, which essentially is the same of the previous one, but only one dimension iterate and the others are fixed.

2.3 Slicing

The slicing operation consists in set one of the indexes as a constant. We do not want to create a new tensor duplicating the memory. Instead, we want to share the same linear array A of the original tensor. In this way we generate a tensor which has the rank decreased by one. This new tensor has to be read in a different way, starting from a different point and with a different ending point. Consider a tensor M with r = n, d = (d_1 , . . . , d_n), s = (s_1 , . . . , s_n) and as start and end pointers respectively, sp and ep. Suppose the operation of slicing consists on imposing x_i = c, the new tensor M' has:

- r'=n-1
- $d'=(d_1 , \ldots , d_{i-1} , d_{i+1} , \ldots , d_n)$
- $s'=(s_1 , \ldots , s_{i-1} , s_{i+1} , \ldots , s_n)$
- sp'=sp+c $\cdot$ s_i
- $ep'=sp'+s'_1 \cdot d'_1$

Slicing operation cannot be applied on tensors with r = 0.

2.4 Windowing

The windowing operation consists in restricting the indexes between two values. As we did with slicing, we want to share the same memory. Also in this case, we only have to calculate new dimensions, strides, start and end pointers. Consider the same tensor structure of M and suppose the operation consists on imposing min $\leq x_i \leq$ max, the new tensor M' has:

- r = n
- d'=(d_1, ...,d_{i-1} , max-min + 1, d_{i+1}, ..., d_n)
- s'= s
- sp' = sp +s_i · min
- ep' = sp' +d'_1 · s'_1

Windowing operation cannot be applied on tensors with r = 0.

2.5 Flattering

The fattening operation developed is more complicated than windowing and slicing. It consists on taking two consecutive indexes and generate a new one. It is a sort of reduction of the dimensions number, maintaining the same elements. Given the same M, considered in the two previous operations, we generate M' fattening indexes i and i + 1, with i < n, in this way:

- r = n-1
- d'=(d_1, ...,d_{i-1} ,d_i · d_{i+1} , d_{i+2}, ..., d_n)
- s' = (s_1, ..., s_{i-1},s_{i+1},..., s_n)
- sp' = sp
- ep' = sp'+d'_1 · s'_1

Flattening operation can be applied only on tensors with r ¿ 1. Remember that flattening operation can be applied only if dimensions and strides are in a consistent state or better when:

$$s_i = \begin{cases} 1 & \text{if i=n} \\ s_{i+1}\cdot \mathrm{d}_{i+1} & \text{otherwise} \end{cases}$$

After the windowing operation dimensions and strides are not in a consistent state, so flattening operation cannot be applied to the returned tensor.

3 Implementation

3.1 Class No Knowledge tensor

3.1.1 Fields

The No Knowledge tensor class, contains the following fields, which describe the object characteristics:

- **rank_**: it is an int value which stores the tensor's rank;
- **dimensions_**: it is an ints vector, which contains tensor's strides;
- **strides_**: it is a pointers vector of type T, which is used to share tensors memory. Using type T means that the type can be any of those available, to be set by users;
- **start_pointer_**: it is an int value which specify tensor's start pointer;
- **end_pointer_**: it is an int value which specify tensor's end pointer.

3.1.2 Access general methods

NKTensor();
This is the default constructor and it is used to return a new tensor or a tensor with rank equal to 0.

NKTensor(Ints... DIM);
This constructor requires a parameter pack as parameter. The parameter pack received is the tensor's dimensions. The constructor will return a new tensor with dimensions DIM.

NKTensor(const vector<int>& dimensions);
This constructor receives an ints vector, which values correspond to tensor's dimensions. Thus, rank, strides, start pointer and end pointer values are assigned, then a new tensor with dimension dimensions is returned.

NKTensor(const NKTensor& ten);
This is the copy contructor used within the class. It creates a new tensor copied from a tensor received as parameter, sharing the same linear array of ten. It has been set to default.

NKTensor(NKTensor&& ten);
Move constructor for this class. The parameter cannot be constant because it could be changed. The move constructor is set to default.

~NKTensor();
This is the default destructor.

NKTensor& operator=(const NKTensor& ten);
This is the default copy assignment, implicitly is our sharing assignment.

NKTensor& operator=(NKTensor&& ten);
This is the default move constructor.

NKTensor copy();
This is a method creates a copy of the tensor. It creates a new memory and copies only those elements whose are visible by the tensor.

T get(Ints... INDEXES);
T getWithArray(const int indexes[], const int dim);
T get(const vector<int>& indexes);
T operator()(Ints... INDEXES);
These methods only permit to read a specific element represented by a tupla.

void set(const T& value, Ints... INDEXES);
void setWithArray(const T& value, const int indexes[], const int dim);
void set(const T& value, const vector<int>& indexes);
These methods only permit to write a specific element represented by a tupla.

NKIterator<T>begin();
NKIterator<T>end();
These two methods return respectively begin and end iterator of the entire collection of tensor's elements. If you try to create an iterator on tensor with rank 0, an exception is throwed. Iterator are constructable only in this way.

NKIteratorOneDim<T>begin(const int idx, const vector<int>& fixed_values);
NKIteratorOneDim<T>begin(const int idx, Ints... fixed_values);
NKIteratorOneDim<T>end(const int idx, const vector<int>& fixed_values);
NKIteratorOneDim<T>end(const int idx, Ints... fixed_values);
These methods return begin and end iterator of the collection of tensor's elements, with only one dimension variable and the others fixed. If you try to create an iterator on tensor with rank 0, an exception is throwed. Iterator are constructable only in this way.

The provided operations on iterator are:

- Step forward (i.e. a++ and ++a)
- Test for equality (i.e. a==b);
- Test for inequality (i.e. a!=b)

3.1.3 Slicing, Windowing and Flattening

For these three operations, we decided to develop two different versions. The main reason is to allow to users to apply single or multiple times the same (or

different, where it is possible) operation on the same tensor. So, within the class there are two methods for slicing operation - one for single slicing and the other for multiple - and so for windowing operation.
The only exception is the flattening operation. Indeed, we developed only one method which acts in the same way on the case of single and multiple operations on the tensor. The reason is that as parameter, users only need to specify the minimum index and the maximum one, on whose to apply flattening operation(Ref. 2.3, 2.4, 2.5).

NKTensor slicing(const int idx, const int val);
NKTensor operator[](const int val); (slicing on the first index)
NKTensor slicing(const vector<pair <int,int>>& idx val);
NKTensor windowing(const int idx, const int min, const int max);
NKTensor windowing(const vector<tuple<int,int,int>>& t);
NKTensor flattening(const int min, const int max);

3.2 Rank only tensor

Within this section, we present the development of the tensor in which is specified the rank field.
We will not describe the constructors and iterators, since they were developed in the same way of No Knowledge tensor class. The only thing we want to say about iterators is that, rather than check if it is possible to create an iterator on a specific tensor, as it happens in the case of No Knowledge tensors, we simply start to iterate on tensors with rank at least equal to 1.
We want now to analyze where and when this tensor class differs from the first version. Firstly, we create a specialization for tensor with rank equal to 0. For this kind of tensor are only permitted construction, copy, get and set. Instead, slicing, windowing, flattening and iterator are not available.
Regarding multiple slicing and flattening methods, they return a No Knowledge tensor type instead of rank only tensor type, because at compile time we cannot know the tensor's characteristics at the end of all operations applied to the tensor.

Here are the only methods signatures that differ from the No Knowledge tensor.
ROTensor<T, R - 1>slicing(const int idx, const int val);
ROTensor<T, R - 1>operator[](const int val);
NKTensor<T>slicing(const vector<pair<int,int>>& idx val);
ROTensor<T,R>windowing(const int idx, const int min, const int max);
ROTensor<T,R>windowing(const vector<tuple<int,int,int>>& t);
NKTensor<T>fattening(const int min, const int max);