Advanced algorithms and programming methods - 2 [CM0470]

Assignment report: Tensor library

Group members:

- Bernardi Riccardo 864018
- Cecchini Davide 862701

Table of contents

1 Introduction and general structure

The task was to implement a templated library that provides a class to manipulate a Tensor object, defining either statically or dynamically the rank. Library also provide two types of iterators, one that iterate the full content of the tensor and one that iterate along one dimension. It is possible to read and write the data saved into the tensor but is not possible to change any sort of information about the dimensions and the rank(i.e. metadata), furthermore the library provides different types of funtions:

- slicing: fix a dimension on an index and reduce the rank of 1
- flattening: merge two dimensions and reduce the rank of 1
- multi flattening: merge more than two dimension and reduce the rank
- windowing: reduce the width of a dimension cutting out some indexes

Library includes three templated classes for the tensor, one with a static positive rank (different from 1), one with a static rank equal to 1 and one with a static rank equal to 0 that represents the tensor with dynamic rank; Both the iterator classes are templated on rank of the relative tensor.

All the classe templates contain the type parameter relative to the objects saved in tensor.

We decided to not represent a tensor zero template because It would be a datapoint that is not of interest and can be overkill to represent it with a structure that at the end costs more than the datum itself. Furthermore since we cannot go from a lower templated type to an higher templated one(because every operation reduces rank or leave it as before) the zero template will remain simple variable with a get and a set.

2 Tensor

Static rank tensor

The static rank tensor has this signature:

```
template<class T = size_t, size_t rank=0>
class Tensor{}
```

This class is part of a framework composed by specialized templates and this means that if the user specifies the rank then the methods that will be called will be ones in this class. Constructors for this class are many because of the fact that in this way it is possible to insert data in different manners. Fact of having many constructors is important also for the methods inside the class.

```
Tensor<T,rank>(std::initializer_list<size_t> a){}
Tensor<T,rank>(const std::vector<size_t> a){}
Tensor<T, rank>(const Tensor<T, rank>& a) : widths(a.widths),
strides(a.strides), data(a.data), offset(a.offset){}
Tensor<T, rank>(const std::initializer_list<size_t>& a, std::vector<T>&
new_data){}
```

For all the constructors holds the property below:

We iterate on it for checking rightness of values because we accept only positive values for dimensions. We check also right dimension for the vector that is coherent with the rank declared statically. If the dimensions are right then they are assigned to the new tensor, are calculated the strides and it is set-up a vector that is linear and is large enough to fit the dimensions declared. This last one is a vector that is also shared because in case of copy-construction of another tensor from this one we want to copy reference to my data instead of passing the entire linear vector, this is due to performances.

The first constructor takes an initializer list that is a list in the form that a user can feel comformtable because it is faster than preparing a vector.

The second constructor takes a constant vector passed entirely, this is worst for performances because obviously if the vector is huge then the cost is high but it is useful to pass data from internal methods that cannot pass a reference because the reference will be dangling after method resolution.

The third method takes a reference to a tensor and since the creator belongs to tensor class then it can access to methods of the new one setting values as the current instance. The result will be a new tensor created starting from an older one and they are perfectly identical. They also refer to the same data.

The fourth method exists because of the fact that methods before create an empty tensor that has to be filled in a second moment instead this one permits at the same time to create and fill.

After constructors is important to remember my friends:

```
friend class TensorIterator<T, rank>;
friend class TensorIteratorFixed<T, rank>;
friend class Tensor<T, rank + 1>;
```

The class declares as friends its tensorIterator of the same rank and its tensorIteratorFixed. We needed to do this because of the fact that the Iterators need to have a copy of strides and widths that in the tensors are private and they are not available to user.

The class is also friend with same class with rank + 1 because of the fact that upper tensor needs to create a lower ranked tensor when it applies flatten or slicing functions.

Class Iterators:

```
TensorIterator<T, rank> begin(){};
TensorIterator<T, rank> end(){};
TensorIteratorFixed<T, rank> begin(const std::vector<int>& starting_indexes,
const size_t& sliding_index){};
TensorIteratorFixed<T, rank> end(const std::vector<int>& starting_indexes,
const size_t& sliding_index){};
```

It would be important to be compliant with the smart fors so the tensors have begin and end, in this way you can use iterators as members of a class that is more convenient than using them as static methods passing the tensor as parameter. Obviously we have two overloaded begins and ends because the simpler is for the random access and the more complex is because it is requested to fix rank-1 axis to iterate on the remaining one.

Enforcing copy:

To enforce the copy of a tensor and its own data there is a dedicated method, here below the signature:

```
Tensor<T, rank> copy() const{}
```

This is needed in which cases the user thinks that data have to be preserved in the original array and modified in a copy.

Getter and setter for values:

```
T& operator()(const initializer_list<size_t>& indices){};
T& operator()(vector<int> indices_v){};
```

Through this method we can get a reference to a memory cell of the tensor that we can read or write, the method checks for bounds to ensure that user doesn't goes out of bounds. To receive the value we search for the cell in the position that is calculated in this way:

$(\sum_i^n strides[i] \cdot widths[i]) + offset$

The offset is needed in case we do a slicing operation, we will see how it is calculated in the slicing section.

The second method that uses a vector is useful when the queries does not come from the user but from other methods of the library. Using vector in practice permits a programmatically buildable query.

Slicing Method:

```
Tensor<T, rank - 1> slice(const size_t&  index, const size_t& value)
```

It receives as input the dimension index that the user wants to cut out and the value of that precise dimension that user wants to filter. There are some initial check for coherence of input data and actual configuration of the tensor and the reliability of an actual storage inside the tensor(i.e. we check if the linear array exists). The result of a slice is a Tensor of rank - 1 so we create such vector and we instantiate its internal state with:

- widths of the old tensor without the one in the position of index
- strides of the old tensor without the one in the position of index

The offset is calculated as:

$offset = strides[index] \cdot value$

And the data is the same data of the old vector.

The flatten method:

```
Tensor<T, rank - 1> flatten(const size_t& start){};
```

flatten takes a starting dimension index and it performs a flattening operation on the starting index and the dimension at its right. Also checks for coherence are performed before every operation both on input data and on a consistent state of the tensor. The operation is performed by calculating a newer vector of widths and the relative strides. The offset is the same of the older tensor.

The multi-flatten method:

```
Tensor<T> multiFlatten(const size_t& start, const size_t& stop){};
```

This function does the same of the regular flatten except for the fact that starting and ending point are at an arbitrary distance(i.e. not bound to one) but checks on coherence of input are performed to avoid overflow. Furthermore the result of this method is not a tensor of rank-1 but a dynamic tensor since we cannot know at compile time the ariety of the flattening.

The window method:

```
Tensor<T, rank> window(const size_t& index, const size_t& start, const size_t&
stop){};
```

Window method shrinks the allowed range of values in a given dimension index. This is done recalculating the widths of the index dimension such as:

```
widths[index] = stop - start + 1
```

The offset as:

```
offset += strides[index] * start
```

Strides and data are maintained. At the end the resulting vector is a vector with the same rank as initial one.

Static rank 1 tensor

The static rank tensor 1 has this signature:

```
template<class T = size_t, 1>
class Tensor{}
```

This class is part of a framework composed by specialized templates and this means that if the user specifies the rank then the methods that will be called will be ones in this class. Constructors for this class are many because of the fact that in this way it is possible to insert data in different manners. Fact of having many constructors is important also for the methods inside the class.

```
Tensor<T, 1>(std::initializer_list<size_t> a){}
Tensor<T, 1>(const std::vector<size_t> a){}
Tensor<T, 1>(const Tensor<T, rank>& a) : widths(a.widths), strides(a.strides),
data(a.data), offset(a.offset){}
Tensor<T, 1>(const std::initializer_list<size_t>& a, std::vector<T>& new_data)
{}
```

For all the constructors holds the property below:

We iterate on it for checking rightness of values because we accept only positive values for dimensions. We check also right dimension for the vector that is coherent with the rank declared statically. If the dimensions are right then they are assigned to the new tensor, are calculated the strides and it is set-up a vector that is linear and is large enough to fit the dimensions declared. This last one is a vector that is also shared because in case of copy-construction of another tensor from this one we want to copy reference to my data instead of passing the entire linear vector, this is due to performances.

The first constructor takes an initializer list that is a list in the form that a user can feel comformtable because it is faster than preparing a vector.

The second constructor takes a constant vector passed entirely, this is worst for performances because obviously if the vector is huge then the cost is high but it is useful to pass data from internal methods that cannot pass a reference because the reference will be dangling after method resolution.

The third method takes a reference to a tensor and since the creator belongs to tensor class then it can access to methods of the new one setting values as the current instance. The result will be a new tensor created starting from an older one and they are perfectly identical. They also refer to the same data.

The fourth method exists because of the fact that methods before create an empty tensor that has to be filled in a second moment instead this one permits at the same time to create and fill.

After constructors is important to remember my friends:

```
friend class TensorIterator<T, rank>;
friend class TensorIteratorFixed<T, rank>;
friend class Tensor<T, rank + 1>;
```

The class declares as friends its tensorIterator of the same rank and its tensorIteratorFixed. We needed to do this because of the fact that the Iterators need to have a copy of strides and widths that in the tensors are private and they are not available to user.

The class is also friend with same class with rank + 1 because of the fact that upper tensor needs to create a lower ranked tensor when it applies flatten or slicing functions.

Class Iterators:

```
TensorIterator<T, rank> begin(){};
TensorIterator<T, rank> end(){};
TensorIteratorFixed<T, rank> begin(const std::vector<int>& starting_indexes,
const size_t& sliding_index){};
TensorIteratorFixed<T, rank> end(const std::vector<int>& starting_indexes,
const size_t& sliding_index){};
```

It would be important to be compliant with the smart fors so the tensors have begin and end, in this way you can use iterators as members of a class that is more convenient than using them as static methods passing the tensor as parameter. Obviously we have two overloaded begins and ends because the simpler is for the random access and the more complex is because it is requested to fix rank-1 axis to iterate on the remaining one.

Enforcing copy:

To enforce the copy of a tensor and its own data there is a dedicated method, here below the signature:

```
Tensor<T, rank> copy() const{}
```

This is needed in which cases the user thinks that data have to be preserved in the original array and modified in a copy.

Getter and setter for values:

```
T& operator()(const initializer_list<size_t>& indices){};
T& operator()(vector<int> indices_v){};
```

Through this method we can get a reference to a memory cell of the tensor that we can read or write, the method checks for bounds to ensure that user doesn't goes out of bounds. To receive the value we search for the cell in the position that is calculated in this way:

$(\sum_i^n strides[i] \cdot widths[i]) + offset$

The offset is needed in case we do a slicing operation, we will see how it is calculated in the slicing section.

The second method that uses a vector is useful when the queries does not come from the user but from other methods of the library. Using vector in practice permits a programmatically buildable query.

Slicing and Flattening operator are not permitted on a single dimension tensor because we decided to not represent a tensor zero template.

The window method:

```
Tensor<T, rank> window(const size_t& index, const size_t& start, const size_t&
stop){};
```

Window method shrinks the allowed range of values in a given dimension index. This is done recalculating the widths of the index dimension such as:

```
widths[index] = stop - start + 1
```

The offset as:

```
offset += strides[index] * start
```

Strides and data are maintained. At the end the resulting vector is a vector with the same rank as initial one.

Dynamic rank tensor

The static rank tensor has this signature:

```
template<class T = size_t, 0>
class Tensor{}
```

This class is part of a framework composed by specialized templates and this means that if the user specifies the rank then the methods that will be called will be ones in this class. Constructors for this class are many because of the fact that in this way it is possible to insert data in different manners. Fact of having many constructors is important also for the methods inside the class.

```
Tensor<T,rank>(std::initializer_list<size_t> a){}
Tensor<T,rank>(const std::vector<size_t> a){}
Tensor<T, rank>(const Tensor<T, rank>& a) : widths(a.widths),
strides(a.strides), data(a.data), offset(a.offset){}
Tensor<T, rank>(const std::initializer_list<size_t>& a, std::vector<T>&
new_data){}
```

For all the constructors holds the property below:

We iterate on it for checking rightness of values because we accept only positive values for dimensions. We check also right dimension for the vector that is coherent with the rank declared statically. If the dimensions are right then they are assigned to the new tensor, are calculated the strides and it is set-up a vector that is linear and is large enough to fit the dimensions declared. This last one is a vector that is also shared because in case of copy-construction of another tensor from this one we want to copy reference to my data instead of passing the entire linear vector, this is due to performances.

The first constructor takes an initializer list that is a list in the form that a user can feel comformtable because it is faster than preparing a vector.

The second constructor takes a constant vector passed entirely, this is worst for performances because obviously if the vector is huge then the cost is high but it is useful to pass data from internal methods that cannot pass a reference because the reference will be dangling after method resolution.

The third method takes a reference to a tensor and since the creator belongs to tensor class then it can access to methods of the new one setting values as the current instance. The result will be a new tensor created starting from an older one and they are perfectly identical. They also refer to the same data.

The fourth method exists because of the fact that methods before create an empty tensor that has to be filled in a second moment instead this one permits at the same time to create and fill.

After constructors is important to remember my friends:

```
friend class TensorIterator<T, 0>;
friend class TensorIteratorFixed<T, 0>;
template<typename S, size_t rank> friend class Tensor;
```

The class declares as friends its tensorIterator of the same rank and its tensorIteratorFixed. We needed to do this because of the fact that the Iterators need to have a copy of strides and widths that in the tensors are private and they are not available to user.

The class is also friend with same class of all ranks because of the fact that a static tensor can be everytime converted to a dynamic one.

Class Iterators:

```
TensorIterator<T> begin(){};
TensorIterator<T> end(){};
TensorIteratorFixed<T> begin(const std::vector<int>& starting_indexes, const
size_t& sliding_index){};
TensorIteratorFixed<T> end(const std::vector<int>& starting_indexes, const
size_t& sliding_index){};
```

It would be important to be compliant with the smart fors so the tensors have begin and end, in this way you can use iterators as members of a class that is more convenient than using them as static methods passing the tensor as parameter. Obviously we have two overloaded begins and ends because the simpler is for the random access and the more complex is because it is requested to fix rank-1 axis to iterate on the remaining one.

Enforcing copy:

To enforce the copy of a tensor and its own data there is a dedicated method, here below the signature:

```
Tensor<T> copy() const{}
```

This is needed in which cases the user thinks that data have to be preserved in the original array and modified in a copy.

Getter and setter for values:

```
T& operator()(const initializer_list<size_t>& indices){};
T& operator()(vector<int> indices_v){};
```

Through this method we can get a reference to a memory cell of the tensor that we can read or write, the method checks for bounds to ensure that user doesn't goes out of bounds. To receive the value we search for the cell in the position that is calculated in this way:

$(\sum_i^n strides[i] \cdot widths[i]) + offset$

The offset is needed in case we do a slicing operation, we will see how it is calculated in the slicing section.

The second method that uses a vector is useful when the queries does not come from the user but from other methods of the library. Using vector in practice permits a programmatically buildable query.

Slicing Method:

```
Tensor<T> slice(const size_t&  index, const size_t& value)
```

It receives as input the dimension index that the user wants to cut out and the value of that precise dimension that user wants to filter. There are some initial check for coherence of input data and actual configuration of the tensor and the reliability of an actual storage inside the tensor(i.e. we check if the linear array exists). The result of a slice is a Tensor of rank - 1 so we

create such vector and we instantiate its internal state with:

- widths of the old tensor without the one in the position of index
- strides of the old tensor without the one in the position of index

The offset is calculated as:

$offset = strides[index] \cdot value$

And the data is the same data of the old vector.

The flatten method:

```
Tensor<T> flatten(const size_t& start){};
```

flatten takes a starting dimension index and it performs a flattening operation on the starting index and the dimension at its right. Also checks for coherence are performed before every operation both on input data and on a consistent state of the tensor. The operation is performed by calculating a newer vector of widths and the relative strides. The offset is the same of the older tensor.

The multi-flatten method:

```
Tensor<T> multiFlatten(const size_t& start, const size_t& stop){};
```

This function does the same of the regular flatten except for the fact that starting and ending point are at an arbitrary distance(i.e. not bound to one) but checks on coherence of input are performed to avoid overflow. Furthermore the result of this method is not a tensor of rank-1 but a dynamic tensor since we cannot know at compile time the ariety of the flattening.

The window method:

```
Tensor<T> window(const size_t& index, const size_t& start, const size_t& stop)
{};
```

Window method shrinks the allowed range of values in a given dimension index. This is done recalculating the widths of the index dimension such as:

```
widths[index] = stop - start + 1
```

The offset as:

```
offset += strides[index] * start
```

Strides and data are maintained. At the end the resulting vector is a vector with the same rank as initial one.

3 Tensor Iterators

Tensor Iterator

The tensorIterator acts as a random-access iterator and its signatures are:

```
TensorIterator<T, rank>(Tensor<T, rank>& tensor) : ttensor(tensor) {};
TensorIterator<T, rank>(const TensorIterator<T, rank>& old_iterator) :
ttensor(old_iterator.ttensor), indexes(old_iterator.indexes) {};
TensorIterator<T, rank>(Tensor<T, rank>& tensor, const std::vector<int>&
starting_indexes) : indexes(starting_indexes), ttensor(tensor){};
```

The first one takes reference of the tensor and binds it to its inner ttensor. It also sets up the indexes of the iterator based on dimensions of the tensor and initially they are zeros, this permits to start from the first indexable value in the tensor.

The second is a copy constructor to permit the duplication of an iterator.

The third one takes a reference to a tensor and a vector of starting indexes so the user can decide to skip some initial cells. There is no check about coherence of starting indexes with the internal state of the iterator or of the tensor.

A tensor has also operations between tensors so here there is a list of them:

```
T& operator*() const {};
T* operator->() const {};
```

The first operator gives a pointer to write/read the value pointed and the pointer is constant.

The second one returns a pointer to a pointer that is again constant.

```
TensorIterator<T, rank> operator++(int) {};
TensorIterator<T, rank>& operator++() {};
TensorIterator<T, rank> operator--(int) {};
TensorIterator<T, rank>& operator--() {};
```

These operators in case they are prefix(parenthesis with null inside) increment pointer and return it otherwise if they are postfix(parentheses with "int" inside) then they return pointer and after they increment.

```
bool operator==(const TensorIterator<T, rank>& other_iterator) const {};
bool operator!=(const TensorIterator<T, rank>& other_iterator) const {};
```

Equality operators are useful in the smart fors and they returns a boolean value checking that the tensor is the same for both the iterators and the indexes are equal.

```
TensorIterator<T, rank>& operator+=(int inc) {};
TensorIterator<T, rank>& operator-=(int dec) {};
TensorIterator<T, rank> operator+(int inc) const {};
TensorIterator<T, rank> operator-(int dec) const {};
```

They uses a private function called "increment" to modify in a proper way the indexes on which It is pointing the iterator. This auxiliary function is needed because of the fact that inside every tensor is multi-dimensional so we need to transform for example a "+1" into an operation that respects the widths of various dimensions.

```
T& operator[](int access_index) const {};
```

This operation simulates a movement of the iterator to the point requested by the user but at the end of the operation the iterator returns to the point before of this operation.

```
bool operator<(const TensorIterator<T, rank>& other_iterator) const {};
bool operator>(const TensorIterator<T, rank>& other_iterator) const {};
bool operator<=(const TensorIterator<T, rank>& other_iterator) const {};
bool operator>=(const TensorIterator<T, rank>& other_iterator) const {};
```

These operators are useful to cycle on a tensor, they checks if the current iterator respects inequality described returning a boolean value. Since we need only to check that two tensors and indexes of the two iterators are right then we need only the const reference to the other vector.

```
int operator-(const TensorIterator<T, rank>& other_iterator) const {}:
```

This operator returns a value that is the distance between two references of two iterators that belongs to the same tensor.

Tensor Iterator Fixed

```
TensorIteratorFixed<T, rank>(Tensor<T>& tensor, const std::vector<int>&
starting_indexes, const size_t& sliding_index) :ttensor(tensor) {};
TensorIteratorFixed<T, rank>(const TensorIteratorFixed<T, rank>& old_iterator)
: ttensor(old_iterator.ttensor), indexes(old_iterator.indexes),
sliding_index(old_iterator.sliding_index) {};
```

The first constructor takes starting indexes as an initializer list of the same size of number of dimensions of the tensor and takes also a sliding index that is the index of the dimension that can vary so the dimnsion on which the user wants to iterate. Other dimensions remains fixed. The second constructor is a copy constructor to dupliucate the current iterator.

The operations below works in the same way of the ones of the "Tensor Iterator"

```
T& operator*() const {};
T* operator->() const {};
```

```
TensorIteratorFixed<T, rank> operator++(int) {};
TensorIteratorFixed<T, rank>& operator++() {};
TensorIteratorFixed<T, rank> operator--(int) {};
TensorIteratorFixed<T, rank>& operator--() {};
```

```
bool operator==(const TensorIteratorFixed<T, rank>& other_iterator) const {};
bool operator!=(const TensorIteratorFixed<T, rank>& other_iterator) const {};
```

```
TensorIteratorFixed<T, rank>& operator+=(int inc) {};
TensorIteratorFixed<T, rank>& operator-=(int dec) {};
TensorIteratorFixed<T, rank> operator+(int inc) const {};
TensorIteratorFixed<T, rank> operator-(int dec) const {};
```

```
T& operator[](int access_index) const {};
```

```
bool operator<(const TensorIteratorFixed<T, rank>& other_iterator) const {};
bool operator>(const TensorIteratorFixed<T, rank>& other_iterator) const {};
bool operator<=(const TensorIteratorFixed<T, rank>& other_iterator) const {};
bool operator>=(const TensorIteratorFixed<T, rank>& other_iterator) const {};
```

```
int operator-(const TensorIteratorFixed<T, rank>& other_iterator) const {};
```