Advanced algorithms and programming methods - 2 [CM0470] - Prof. A. Torsello

Group:

Baccega Sandro (865024)

Santoro Arnaldo (822274)

BUILDING AND RUNNING

```
mkdir build
cd build
cmake ../
make
./tensor-library
```

USAGE

Examples

With this library you can represent a tensor in the Einstein notation with the following syntax:

Tensor a_{ij} becomes `a["ij"]`

Where:

- a is the tensor
- `"ij"` are the selected indexes

This will return a `tensor_expr` type, that can be evaluated into a Tensor or a value using the `evaluate()` method in the following way:

```
tensor::tensor_expr exp = a["i"] + b["i"];

tensor::tensor<int> c = exp.evaluate();
```

We encourage the use of the **auto** keyword to specify the type of the expression,

since the type **tensor::tensor_expr** can be unconfortable to use.

Trace

You can calculate the tensor's trace in the traditional Einstein notation way:

```
auto exp = a["ii"];

int c = exp.evaluate();
```

Addition, Subtraction and Multiplication

There are 3 supported operations between tensors:

- Addition

  `a["ij"] + b["ik"]`

- Subtraction

  `a["ij"] - b["ik"]`

- Multiplication

  `a["ij"] * b["ik"]`

To use one of these operations you can use the following syntax:

```
auto exp = a["ij"] * b["ik"];

tensor::tensor<int> c = exp.evaluate();
```

This expression fails an assert (unintentionally) at runtime:

```
auto exp = a["ii"] * b["ij"];

tensor::tensor<int> c = exp.evaluate();
```

This expression works but does not behave as intended:

```
auto exp = a["iij"] * b["ij"];

tensor::tensor<int> c = exp.evaluate();
```

Using more (or less) indexes than needed by a tensor fails (intentionally) an assert at runtime. This could have been implemented at static time.

Also all dummy indexes have to be used in ranks of the same dimensions or they fail a runtime assert. This cannot be implemented at static time without static information on the dimension of the ranks.

Generalized Kronecker Product

The operation OP between two tensors A and B or rank r and s with no common indexes returns a tensor of rank r*s whose ranks' elements are the result of OP between each element of A and each element of the rank of B.

This results similar to a Kronecker product which stores the results in different ranks of a tensor.

If there wasn't a bug in the flatten operation one could easily implement a Kronecker product exploiting this functionality.

DESIGN

Composite Pattern

We thought of the Composite Pattern to implement tensor operations in a uniform way.

This is obtained through different implementation of the function `evaluate()`, which behaves differently for each class implementing the operations.

Strategy Pattern

A definition of the *strategy pattern* is a reusable solution that lets an algorithm vary independently from clients that use it; in our code we use this solution to implement the different operations in the implicit summation of Einstein's formalism; this application is combined with the template system to obtain a *static strategy pattern* implementation.

The object `tensor_op` is forward-declared and consists of the *functor* applying the algorithm.

The following structs are **"functors"** (in a broader sense) implementing the operations between two elements of type T, which are then placed in the specialized template of the definition for the `operator+`, `operator-` and `operator*` respectively.

New operations by design easy to add.

Bugs And Missing Features

A simple check that can be added is an assert that checks ranks in different tensors that should have same dimensions and shouldn't be able to compute. E.g. given a rank 2 tensor A of dimension (2,2) and a rank 1 tensor B of dimension (3), the following operation shouldn't be possible $A_{ij} + B_i$

Combined operations

A combination of contraction and other operations with shared free indexes break the program. This happes because no addition and multiplication operations were implemented outside the einstein summation, and because the evaluation of the trace happens before the other operation, effecively ereasing what was once a dummy index from computation.

To solvethe problem one should make an extensive use of the composite pattern to materialize the whole tree of the expression and handle the evaluation checking every index in the operation when the `evaluate` function is called; this delayed evaluation however is computationally expensive for the computer (or the compiler if implemented statically).

Static checks

In our version no significant static checks were made, due to a lack of time caused by jobs and other courses projects.

However we designed two possible solutions:

- check that the number of indexes and the ranks match each other;
- check that the type of indexes is compatible with each other.

  The solution we designed intended to implement static asserts only to check that the number of indexes of `tensor_expr` concides with the rank of the tensor; this choice won over the second one due to two factors:

- simplicity
- efficiency
- a case of "it's not a bug, it's a feature": no common indexes lead to the interesting effect of a "stacking tensor" operation, similar to Kronecker's operation.