Using pichi

Christian Walther Andersen*

May 2, 2018

*cwandersen@imada.sdu.dk

CHAPTER 1

THE BASICS

This chapter describes how to use pichito do simple tensor contractions. There are two central classes used for this purpose. The Tensor class (`Tensor.h` and `Tensor.cc`) and the contraction functions (`Contraction.h` and `Contraction.cc`). The Tensor class essentially manages the storage of the data for each individual tensor, while the contraction functions contract the indices.

There are four different contraction functions, which we will go through now:

Loading data to a tensor

This section describes how to create a tensor and fill it with data. The example below creates a rank 4 tensor of size 64 (all dimensions have this size) and fills it with data from some source, for example a data file. We use `cdouble` as an alias for `std::complex<double>`.

```
// Create a 64^4 tensor with all elements set to 0
Tensor t( 4 , 64 );

cdouble data[64*64];
// Load data with numbers
// ...

t.setSlice({-1,3,48,-1} , data);
// After this call:
// t(0,3,48,0)    = data[0]
// t(1,3,48,0)    = data[1]
// ...
// t(63,3,48,0)   = data[63]
// t(0,3,48,1)    = data[64]
// t(1,3,48,1)    = data[65]
// ...
// t(63,3,48,63) = data[64*64-1]
```

In the example we see that we modify the tensor in 2-dimensional slices of the tensor. The slice in question is the slice with the second index equal to 3 and the third index equal to 48. Note that we assume that the leading dimension of the array is the first running index. This is always the case.

To fill the entire tensor, this process would have to be repeated for slices with all possible values of the second and third index:

```
Tensor t( 4 , 64 );
```

```
cdouble data[64*64];
// Load data with numbers
// ...
t.setSlice({-1,3,48,-1} , data);
// ...
t.setSlice({-1,4,48,-1} , data);
// ...
```

Contractions

This section discusses how to contract the tensors. There are four different ways of doing this:

One fully contracted tensor

```
Tensor a( 4 , 64 );
// Load data into tensor ...

cdouble r = contract(a , {{0,3},{1,2}});
```

Here we create a four dimensional tensor and contract it into a number. The list of integer pairs indicate that we contract index 0 with index 3 and index 1 with index 2, or in mathematical notation:

$$r = A_{abba} \tag{1.1}$$

Two fully contracted tensors

```
Tensor a( 3 , 64 );
Tensor b( 3 , 64 );
// Load data into tensors ...

cdouble r = contract(a,b,{{0,2},{1,0},{2,1}});
```

Here we create two rank-3 tensors and contract their indices. The list indicates that index 0 of the first tensor is contracted with index 2 on the second tensor, etc. In mathematical notation, the contraction above would be

$$r = A_{abc}B_{bca} \tag{1.2}$$

One partially contracted tensor

```
Tensor a( 4 , 64 );
// Load data into tensor ...

Tensor b( 2 , 64 );
contract(a , {{0,3}}, b);
```

Here we create a four dimensional tensor and contract two indices (index 0 and index 3). The resulting rank 2 tensor is copied into the last input argument, which must have the correct rank and size beforehand. Again, in mathematical notation

$$B_{bc} = A_{abca} \tag{1.3}$$

Two partially contracted tensors

```
Tensor a( 3 , 64 );
Tensor b( 3 , 64 );
// Load data into tensors ...

Tensor c( 2 , 64 );
contract(a,b,{{0,2},{1,0}},c);
```

Here we create two rank-3 tensors and contract two indices to form a rank 2 tensor, C. As before, the result is copied into the last input argument, which again must have the correct dimensions. In mathematical notation, the contraction above would be

$$C_{cd} = A_{abc}B_{bda} \tag{1.4}$$

CHAPTER 2

GOING FURTHER

This chapter describes how to use pichias efficiently as possible. When doing tensor contractions there are huge gains to be made by doing things the right way.

Tensor data storage

Inside the tensor class, the data is stored in one consecutive array. This imposes a choice of how to map a tensor index to a 1-dimensional array index. By default, tensors store their data with the first index as the leading dimension, the second indexas the next to leading dimension, etc. This means that, when inserting data into the tensor, it is beneficial to use the first two indices as the running indices:

```
Tensor t1( 4 , 64 );
Tensor t1( 4 , 64 );

cdouble data[64*64];
// Load data with numbers
// ...

// Fast
t.setSlice({-1,-1,16,11} , data);

// Slow
t.setSlice({-1,3,48,-1} , data);
```

The storage of the tensors can be changed at any point, and will be changed when contracting tensors. See the documentation in the code.

Contractions of more than two tensors

To contract more than two tensors, a combination of contraction routines will have to be used. For example, the code excerpt below shows how to make the contraction

$$A_{abc}B_{abd}C_{cd} \tag{2.1}$$

```
Tensor a( 3 , 64 );
Tensor b( 3 , 64 );
Tensor c( 2 , 64 );
// Load data into tensors ...
```

```
Tensor d( 2 , 64 );
contract(a,b,{{0,0},{1,1}},d);

cdouble r = contract(d,c,{{0,0},{1,1}});
```

Here again we are faced with a choice. We choose to start by contracting A and B, but we could have also chosen to start with A and C, or B and C. It turns out that the example above is the optimal way with regards to runtime. For a full overview of the contraction strategies needed for hadron spectroscopy, see the benchmark document.