Cytnx

Kai-Hsin Wu

Jun 28, 2020

CONTENTS

Cytnx is a library design for Quantum physics simulation using GPUs and CPUs.

CHAPTER

ONE

INTRODOCTION

Cytnx is a library design for Quantum/classical Physics simulation.

The library is build from bottum-up, with both C++ and python in mind right at the beginning of develoopement. That's why nearly 95% of the APIs are exactly the same in both C++ and python ends.

Most of Cytnx APIs share very similar interfaces as the most common and popular libraries: numpy/scipy/pytorch. This is spcifically designed so as to reduce the learning curve for users. Further more, we implement these easy-to-use python libraries interface to C++ side in hope to benefit users who want to bring their python programming experience to C++ side and speed up their programs.

Cytnx also support multi-devices (CPUs/GPUs) directly in the base container level. Especially, not only the container but also our linear algebra fucntions share the same APIs regadless of the devices where the input Tensors are, just like pytorch. This provides users ability to accelerate the code without worrying too much details about multi-devices programming.

From the physics side, cytnx_extension namespace/submodule provides powerful tools such as CyTensor, Network, Bond, Symmetry etc. These objects are built on top of Tensor objects, spcifically aiming for reduce the developing work of Tensor network algorithm by simplify the user interfaces.

In this user guide, both python and C++ APIs will be discussed, provided side-by-side for users to better understand how to use Cytnx, and understand the conversion in between Python API and C++ API.

1.1 Features

- C++ and python are co-exists, there is no one first.
- 95% of API are the same in C++ and Python. This means one can do a fast prototype in python, and directly convert to C++ with extremely minimal re-writing of codebase.
- GPUs/CPUs multi-devices support.
- Easy to use user-interface similar to numpy/scipy/pytorch.
- Enhance tools specifically designs for Quantum/classical Physics simulation.

CHAPTER

TWO

USER GUIDE

To use the library, simply include/import cytnx.

In Python, using import

```python
import cytnx
```

In C++, using include header

```cpp
#include "cytnx.hpp";
```

Note: In C++, there is a namespace **cytnx**.

There are equivalence of alias between python module and c++ namespace, for example if we want to alias cytnx as cy,

In python :

```python
import cytnx as cy
```

This is equivalent in C++ as:

```cpp
#include "cytnx.hpp";
namespace cy=cytnx;
```

Now we are ready to start using cytnx!

Continue reading:

2.1 Basic objects

2.1.1 Tensor

Tensor is the basic building block of Cytnx. In fact, the API of Tensor in cytnx is very similar to torch.tensor (so as numpy.array, since they are also similar to each other)

Let's take a look on how to use it:

1. Create a Tensor

Just like numpy.array / torch.tensor, Tensor is generally created using generator such as **zero()**, **arange()**, **ones()**.

For example, suppose we want to define a rank-3 tensor with shape (3,4,5), and initialize all elements with zero:

- In python:

```
A = cytnx.zeros([3,4,5]);
```

- In c++:

```
cytnx::Tensor A = cytnx::zeros({3,4,5});
```

Note:

1. In cytnx, the conversion of python list is equivalent to C++ *vector*; or in some case like here, it is a *initializer list*.
2. The conversion in between is pretty straight forward, one simply replace [] in python with {}, and you are all set!

Other options such as **arange()** (similar as np.arange), and **ones** (similar as np.ones) can also be done.

- In python :

```
A = cytnx.arange(10);      #rank-1 Tensor from [0,10) with step 1
B = cytnx.arange(0,10,2); #rank-1 Tensor from [0,10) with step 2
C = cytnx.ones([3,4,5]);  #Tensor of shape (3,4,5) with all elements set to one.
```

- In c++:

```
auto A = cytnx::arange(10);      //rank-1 Tensor from [0,10) with step 1
auto B = cytnx::arange(0,10,2); //rank-1 Tensor from [0,10) with step 2
auto C = cytnx::ones({3,4,5});  //Tensor of shape (3,4,5) with all elements set to␣
↪one.
```

Tips In C++, you could make use of *auto* to simplify your code!

1.1. Tensor with different dtype and device

By default, the Tensor will be created with *double* type (or *float* in python) on CPU if there is no additional arguments provided upon creating the Tensor.

You can create a Tensor with different data type, and/or on different devices simply by specify the **dtype** and the **device** arguments upon initialization. For example, the following codes create a Tensor with 64bit integer on cuda-enabled GPU.

- In python:

```
A = cytnx.zeros([3,4,5],dtype=cytnx.Type.Int64,device=cytnx.Device.cuda)
```

- In c++:

```
auto A = cytnx.zeros({3,4,5},cytnx::Type::Int64,cytnx::Device::cuda);
```

Note:

1. Remember the difference of . in python and :: in C++ when you use Type and Device classes.
2. If you have multiple GPUs, you can specify which GPU you want to init Tensor by adding gpu-id to cytnx::Device::cuda.

   For example:

   device=cytnx.Device.cuda+2 #will create Tensor on GPU id=2

   device=cytnx.Device.cuda+4 #will create Tensor on GPU id=4
3. In C++, there is no keyword argument as python, so make sure you put the argument in the correct order. Check API documentation for function signatures!

Currently, there are several data types supported by cytnx:

cytnx type	c++ type	Type object
cytnx_double	double	Type.Double
cytnx_float	float	Type.Float
cytnx_uint64	uint64_t	Type.Uint64
cytnx_uint32	uint32_t	Type.Uint32
cytnx_uint16	uint16_t	Type.Uint16
cytnx_int64	int64_t	Type.Int64
cytnx_int32	int32_t	Type.Int32
cytnx_int16	int16_t	Type.Int16
cytnx_complex128	std::complex<double>	Type.ComplexDouble
cytnx_complex64	std::complex<float>	Type.ComplexFloat
cytnx_bool	bool	Type.Bool

For devices, Cytnx currently supports

cytnx type	Device object
CPU	Device.cpu
CUDA-enabled GPU	Device.cuda+x

1.2 Type conversion

It is possible to convert a Tensor to a different data type. To convert the data type, simply use **Tensor.astype()**.

For example, consider a Tensor *A* with **dtype=Type.Int64**, and we want to convert it to **Type.Double**

- In python:

```
A = cytnx.ones([3,4],dtype=cytnx.Type.Int64)
B = A.astype(cytnx.Type.Double)
print(A.dtype_str())
print(B.dtype_str())
```

- In c++:

```
auto A = cytnx::ones({3,4},cytnx::Type::Int64);
auto B = A.astype(cytnx::Type::Double);
cout << A.dtype_str() << endl;
cout << B.dtype_str() << endl;
```

>> Output:

```
Int64
Double (Float64)
```

Note:

1. Use Tensor.dtype() will return a type-id, where Tensor.dtype_str() will return the type name.
2. Complex data type cannot directly convert to real data type. Use Tensor.real()/Tensor.imag() if you want to get the real/imag part.

1.3 Transfer btwn devices

To move a Tensor between different devices is very easy. We can use **Tensor.to()** to move the Tensor to a different device.

For example, let's create a Tensor on cpu and transfer to GPU with gpu-id=0.

- In python:

```
A = cytnx.ones([2,2]) #on CPU
print(A)
A.to(cytnx.Device.cuda+0)
print(A)
```

- In c++:

```
auto A = cytnx::ones([2,2]); //on CPU
cout << A << endl;
A.to(cytnx.Device.cuda+0);
cout << A << endl;
```

>> Output:

```
Total elem: 4
type  : Double (Float64)
cytnx device: CPU
Shape : (2,2)
[[1.00000e+00 1.00000e+00 ]
 [1.00000e+00 1.00000e+00 ]]

Total elem: 4
type  : Double (Float64)
cytnx device: CUDA/GPU-id:0
Shape : (2,2)
[[1.00000e+00 1.00000e+00 ]
 [1.00000e+00 1.00000e+00 ]]
```

Note: You can use **Tensor.device()** to get the current device-id (cpu = -1), where as **Tensor.device_str()** returns the device name.

2. Manipulate Tensor

Next, let's look at the operations that are commonly used to manipulate Tensor object.

2.1 reshape

Suppose we want to create a rank-3 Tensor with shape=(2,3,4), starting with a rank-1 Tensor with shape=(24) initialized using **arange()**.

This operation is called *reshape*

We can use **Tensor.reshape** function to do this.

- In python:

```
A = cytnx.arange(24)
B = A.reshape(2,3,4)
print(A)
print(B)
```

- In C++:

```
auto A = cytnx::arange(24);
auto B = A.reshape(2,3,4);
cout << A << endl;
cout << B << endl;
```

>> Output:

```
Total elem: 24
type  : Double (Float64)
cytnx device: CPU
Shape : (24)
[0.00000e+00 1.00000e+00 2.00000e+00 3.00000e+00 4.00000e+00 5.00000e+00 6.00000e+00
↪7.00000e+00 8.00000e+00 9.00000e+00 1.00000e+01 1.10000e+01 1.20000e+01 1.30000e+01
↪1.40000e+01 1.50000e+01 1.60000e+01 1.70000e+01 1.80000e+01 1.90000e+01 2.00000e+01
↪2.10000e+01 2.20000e+01 2.30000e+01 ]

Total elem: 24
type  : Double (Float64)
cytnx device: CPU
Shape : (2,3,4)
[[[0.00000e+00 1.00000e+00 2.00000e+00 3.00000e+00 ]
  [4.00000e+00 5.00000e+00 6.00000e+00 7.00000e+00 ]
  [8.00000e+00 9.00000e+00 1.00000e+01 1.10000e+01 ]]
 [[1.20000e+01 1.30000e+01 1.40000e+01 1.50000e+01 ]
  [1.60000e+01 1.70000e+01 1.80000e+01 1.90000e+01 ]
  [2.00000e+01 2.10000e+01 2.20000e+01 2.30000e+01 ]]]
```

Notice that calling **reshape()** returns a new object *B*, so the original object *A* is not changed after calls reshape.

There is the other function **Tensor.reshape_** (with a underscore) that also performs reshape, but instead of return a new reshaped object, it performs inplace reshape to the instance that calls the function. For example:

- In python:

```python
A = cytnx.arange(24)
print(A)
A.reshape_(2,3,4)
print(A)
```

- In C++:

```cpp
auto A = cytnx::arange(24);
cout << A << endl;
A.reshape_(2,3,4);
cout << A << endl;
```

>> Output:

```
Total elem: 24
type  : Double (Float64)
cytnx device: CPU
Shape : (24)
[0.00000e+00 1.00000e+00 2.00000e+00 3.00000e+00 4.00000e+00 5.00000e+00 6.00000e+00␣
↪7.00000e+00 8.00000e+00 9.00000e+00 1.00000e+01 1.10000e+01 1.20000e+01 1.30000e+01␣
↪1.40000e+01 1.50000e+01 1.60000e+01 1.70000e+01 1.80000e+01 1.90000e+01 2.00000e+01␣
↪2.10000e+01 2.20000e+01 2.30000e+01 ]

Total elem: 24
type  : Double (Float64)
cytnx device: CPU
Shape : (2,3,4)
[[[0.00000e+00 1.00000e+00 2.00000e+00 3.00000e+00 ]
  [4.00000e+00 5.00000e+00 6.00000e+00 7.00000e+00 ]
  [8.00000e+00 9.00000e+00 1.00000e+01 1.10000e+01 ]]
 [[1.20000e+01 1.30000e+01 1.40000e+01 1.50000e+01 ]
  [1.60000e+01 1.70000e+01 1.80000e+01 1.90000e+01 ]
  [2.00000e+01 2.10000e+01 2.20000e+01 2.30000e+01 ]]]
```

Thus we see that using underscore version modify the instance itself.

Note: In general, all the funcions in Cytnx that end with a underscore _ is either a inplace function that modify the instance that calls it, or return the reference of some class member.

Hint: You can use **Tensor.shape()** to get the shape of Tensor.

2.1 permute

Now, let's again use the same rank-3 with shape=(2,3,4) as example. This time we want to do permute on the Tensor to exchange axes from indices (0,1,2)->(1,2,0)

This can be achieved with **Tensor.permute**

- In python:

```python
A = cytnx.arange(24).reshape(2,3,4)
B = A.permute(1,2,0)
print(A)
print(B)
```

- In c++:

```cpp
auto A = cytnx::arange(24).reshape(2,3,4);
auto B = A.permute(1,2,0);
cout << A << endl;
cout << B << endl;
```

>> Output:

```
Total elem: 24
type  : Double (Float64)
cytnx device: CPU
Shape : (2,3,4)
[[[0.00000e+00 1.00000e+00 2.00000e+00 3.00000e+00 ]
  [4.00000e+00 5.00000e+00 6.00000e+00 7.00000e+00 ]
  [8.00000e+00 9.00000e+00 1.00000e+01 1.10000e+01 ]]
 [[1.20000e+01 1.30000e+01 1.40000e+01 1.50000e+01 ]
  [1.60000e+01 1.70000e+01 1.80000e+01 1.90000e+01 ]
  [2.00000e+01 2.10000e+01 2.20000e+01 2.30000e+01 ]]]

Total elem: 24
type  : Double (Float64)
cytnx device: CPU
Shape : (3,4,2)
[[[0.00000e+00 1.20000e+01 ]
  [1.00000e+00 1.30000e+01 ]
  [2.00000e+00 1.40000e+01 ]
  [3.00000e+00 1.50000e+01 ]]
 [[4.00000e+00 1.60000e+01 ]
  [5.00000e+00 1.70000e+01 ]
  [6.00000e+00 1.80000e+01 ]
  [7.00000e+00 1.90000e+01 ]]
 [[8.00000e+00 2.00000e+01 ]
  [9.00000e+00 2.10000e+01 ]
  [1.00000e+01 2.20000e+01 ]
  [1.10000e+01 2.30000e+01 ]]]
```

Note: Just like before, there is an equivalent **Tensor.permute_** end with underscore that performs inplace permute on the instance that calls it.

Hint: In some situation where we don't want to create a copy of object, using inplace version of functions can reduce

the memory usage.

In Cytnx, the permute operation does not moving the elements in the memory immediately. Only the meta-data that is seen by user are changed. This can avoid the redudant moving of elements. Note that this approach is also taken in numpy.array and torch.tensor .

If the meta-data is distached from the real memery layout, we call the Tensor in this status *non-contiguous*. We can use **Tensor.is_contiguous()** to check if the current Tensor is in contiguous status.

You can force the Tensor to return to it's contiguous status by calling **Tensor.contiguous()/Tensor.contiguous_()**, although generally you don't have to worry about contiguous, as cytnx automatically handles it for you.

- In python:

```python
A = cytnx.arange(24).reshape(2,3,4)
print(A.is_contiguous())
print(A)

A.permute_(1,0,2)
print(A.is_contiguous())
print(A)

A.contiguous_()
print(A.is_contiguous())
```

- In C++:

```cpp
auto A = cytnx::arange(24).reshape(2,3,4);
cout << A.is_contiguous() << endl;
cout << A << endl;

A.permute_(1,0,2);
cout << A.is_contiguous() << endl;
cout << A << endl;

A.contiguous_();
cout << A.is_contiguous() << endl;
```

Output>>

```
True

Total elem: 24
type  : Double (Float64)
cytnx device: CPU
Shape : (2,3,4)
[[[0.00000e+00 1.00000e+00 2.00000e+00 3.00000e+00 ]
  [4.00000e+00 5.00000e+00 6.00000e+00 7.00000e+00 ]
  [8.00000e+00 9.00000e+00 1.00000e+01 1.10000e+01 ]]
 [[1.20000e+01 1.30000e+01 1.40000e+01 1.50000e+01 ]
  [1.60000e+01 1.70000e+01 1.80000e+01 1.90000e+01 ]
  [2.00000e+01 2.10000e+01 2.20000e+01 2.30000e+01 ]]]

False

Total elem: 24
type  : Double (Float64)
cytnx device: CPU
```

(continues on next page)

(continued from previous page)

```
Shape : (3,2,4)
[[[0.00000e+00 1.00000e+00 2.00000e+00 3.00000e+00 ]
  [1.20000e+01 1.30000e+01 1.40000e+01 1.50000e+01 ]]
 [[4.00000e+00 5.00000e+00 6.00000e+00 7.00000e+00 ]
  [1.60000e+01 1.70000e+01 1.80000e+01 1.90000e+01 ]]
 [[8.00000e+00 9.00000e+00 1.00000e+01 1.10000e+01 ]
  [2.00000e+01 2.10000e+01 2.20000e+01 2.30000e+01 ]]]

True
```

Tip:

1. Generally, you don't have to worry about contiguous issue. you can access the elements and call linalg just like this contiguous/non-contiguous thing doesn't exist.
2. In the case where the function does require user to manually make the Tensor contiguous, a warning will be prompt, and you can simply add a **Tensor.contiguous()/.contiguous_()** before the function call.

Note: As metioned before, **Tensor.contiguous_()** (with underscore) make the current instance contiguous, while **Tensor.contiguous()** return a new object with contiguous status. In the case where the current instance is already in it's contiguous status, calling contiguous will return itself, and no new object will be created.

3. Access elements

Next, let's take a look on how we can access elements inside a Tensor.

3.1 Get elements

Just like python list/numpy.array/torch.tensor, on the python side, we can simply use *slice* to get the elements. See This page . In c++, cytnx take this approach from python and bring it to our C++ API. You can simply use the **slice string** to access elements.

For example:

- In python:

```python
A = cytnx.arange(24).reshape(2,3,4)
print(A)

B = A[0,:,1:4:2]
print(B)

C = A[:,1]
print(C)
```

- In C++:

```cpp
auto A = cytnx::arange(24).reshape(2,3,4);
cout << A << endl;

auto B = A(0,":","1:4:2");
```

(continues on next page)

(continued from previous page)

```
cout << B << endl;

auto C = A(":",1);
cout << C << endl;
```

Output>>

```
Total elem: 24
type  : Double (Float64)
cytnx device: CPU
Shape : (2,3,4)
[[[0.00000e+00 1.00000e+00 2.00000e+00 3.00000e+00 ]
  [4.00000e+00 5.00000e+00 6.00000e+00 7.00000e+00 ]
  [8.00000e+00 9.00000e+00 1.00000e+01 1.10000e+01 ]]
 [[1.20000e+01 1.30000e+01 1.40000e+01 1.50000e+01 ]
  [1.60000e+01 1.70000e+01 1.80000e+01 1.90000e+01 ]
  [2.00000e+01 2.10000e+01 2.20000e+01 2.30000e+01 ]]]


Total elem: 6
type  : Double (Float64)
cytnx device: CPU
Shape : (3,2)
[[1.00000e+00 3.00000e+00 ]
 [5.00000e+00 7.00000e+00 ]
 [9.00000e+00 1.10000e+01 ]]


Total elem: 8
type  : Double (Float64)
cytnx device: CPU
Shape : (2,4)
[[4.00000e+00 5.00000e+00 6.00000e+00 7.00000e+00 ]
 [1.60000e+01 1.70000e+01 1.80000e+01 1.90000e+01 ]]
```

Note:

1. To convert in between python and C++ APIs, notice that in C++, we use operator() instead of operator[] if you are using slice string to acess elements.
2. The return will always be Tensor object, even it is only one elements in the Tensor.

In the case where you have only one element in a Tensor, we can use **item()** to get the element in the standard python type/c++ type.

- In python:

```
A = cytnx.arange(24).reshape(2,3,4)
B = A[0,0,1]
C = B.item()
print(B)
print(C)
```

- In C++:

```cpp
auto A = cytnx::arange(24).reshape(2,3,4);
auto B = A(0,0,1);
double C = B.item<double>();

cout << B << endl;
cout << C << endl;
```

Output>>

```
Total elem: 1
type  : Double (Float64)
cytnx device: CPU
Shape : (1)
[1.00000e+00 ]


1.0
```

Note: In C++, using **item<>()** to get the element require explicitly specify the type that match the dtype of the Tensor. If the type specify does not match, an error will be prompt.

3.2 Set elememts

Setting elements is pretty much the same as numpy.array/torch.tensor. You can assign a Tensor to a specific slice, our set all the elements in that slice to be the same value.

For example:

- In python:

```python
A = cytnx.arange(24).reshape(2,3,4)
B = cytnx.zeros([3,2])
print(A)
print(B)

A[1,:,::2] = B
print(A)

A[0,::2,2] = 4
print(A)
```

- In c++:

```cpp
auto A = cytnx::arange(24).reshape(2,3,4);
auto B = cytnx::zeros({3,2});
cout << A << endl;
cout << B << endl;

A(1,":","::2") = B;
cout << A << endl;

A(0,"::2",2) = 4;
cout << A << endl;
```

Output>>

```
Total elem: 24
type  : Double (Float64)
cytnx device: CPU
Shape : (2,3,4)
[[[0.00000e+00 1.00000e+00 2.00000e+00 3.00000e+00 ]
  [4.00000e+00 5.00000e+00 6.00000e+00 7.00000e+00 ]
  [8.00000e+00 9.00000e+00 1.00000e+01 1.10000e+01 ]]
 [[1.20000e+01 1.30000e+01 1.40000e+01 1.50000e+01 ]
  [1.60000e+01 1.70000e+01 1.80000e+01 1.90000e+01 ]
  [2.00000e+01 2.10000e+01 2.20000e+01 2.30000e+01 ]]]


Total elem: 6
type  : Double (Float64)
cytnx device: CPU
Shape : (3,2)
[[0.00000e+00 0.00000e+00 ]
 [0.00000e+00 0.00000e+00 ]
 [0.00000e+00 0.00000e+00 ]]


Total elem: 24
type  : Double (Float64)
cytnx device: CPU
Shape : (2,3,4)
[[[0.00000e+00 1.00000e+00 2.00000e+00 3.00000e+00 ]
  [4.00000e+00 5.00000e+00 6.00000e+00 7.00000e+00 ]
  [8.00000e+00 9.00000e+00 1.00000e+01 1.10000e+01 ]]
 [[0.00000e+00 1.30000e+01 0.00000e+00 1.50000e+01 ]
  [0.00000e+00 1.70000e+01 0.00000e+00 1.90000e+01 ]
  [0.00000e+00 2.10000e+01 0.00000e+00 2.30000e+01 ]]]


Total elem: 24
type  : Double (Float64)
cytnx device: CPU
Shape : (2,3,4)
[[[0.00000e+00 1.00000e+00 4.00000e+00 3.00000e+00 ]
  [4.00000e+00 5.00000e+00 6.00000e+00 7.00000e+00 ]
  [8.00000e+00 9.00000e+00 4.00000e+00 1.10000e+01 ]]
 [[0.00000e+00 1.30000e+01 0.00000e+00 1.50000e+01 ]
  [0.00000e+00 1.70000e+01 0.00000e+00 1.90000e+01 ]
  [0.00000e+00 2.10000e+01 0.00000e+00 2.30000e+01 ]]]
```

3.3 Low-level API (C++ only)

On C++ side, cytnx provide lower-level APIs with slightly smaller overhead for getting elements. These low-level APIs require using with **Accessor** object.

- **Accessor:** **Accessor** object is equivalent to python *slice*. It is sometimes convenient to use alias to simplify the expression when using it.

```
1   typedef ac=cytnx::Accessor;
2
3   ac(4);     // this equal to index '4' in python
```

(continues on next page)

(continued from previous page)

```
    ac::all(); // this equal to ':' in python
    ac::range(0,4,2); // this equal to '0:4:2' in python
```

In the following, let's see how it can be used to get/set the elements from/to Tensor.

1. operator[] (middle level API) :

```
typedef ac=cytnx::Accessor;
auto A = cytnx::arange(24).reshape(2,3,4);
auto B = cytnx::zeros({3,2});

// [get] this is equal to A[0,:,1:4:2] in python:
auto C = A[{ac(0},ac::all(),ac::range(1,4,2)}];

// [set] this is equal to A[1,:,0:4:2] = B in python:
A[{ac(1),ac::all(),ac::range(0,4,2)}] = B;
```

Note: Remember to put a braket{}. This because C++ operator[] can only accept one argument.

2. **get/set (lowest level API) :** get() and set() is the lowest-level API. Operator() and Operator[] are all build base on these.

```
typedef ac=cytnx::Accessor;
auto A = cytnx::arange(24).reshape(2,3,4);
auto B = cytnx::zeros({3,2});

// [get] this is equal to A[0,:,1:4:2] in python:
auto C = A.get({ac(0},ac::all(),ac::range(1,4,2)});

// [set] this is equal to A[1,:,0:4:2] = B in python:
A.set({ac(1),ac::all(),ac::range(0,4,2)}, B);
```

Hint:

1. Similarly, you can also pass a c++ *vector<cytnx_int64>* as argument.

Tip: If your code requires frequently get/set elements, using low-level API can reduce the overhead.

4. Tensor arithmetic

In cytnx, Tensor can performs arithmetic operation such as **+, -, x, /, +=, -=, *=, /=** with another Tensor or scalalr, just like the standard way you do in python.

4.1 Type promotion

Arithmetic operation in Cytnx follows the similar pattern of type promotion as standard C++/python. When Tensor performs arithmetic operation with another Tensor or scalar, the output Tensor will have the dtype as the one that has stronger type.

The Types order from strong to weak as:

- Type.ComplexDouble
- Type.ComplexFloat
- Type.Double
- Type.Float
- Type.Int64
- Type.Uint64
- Type.Int32
- Type.Uint32
- Type.Int16
- Type.Uint16
- Type.Bool

4.2 Tensor-Tensor arithmetic

Tensor can performs arithmetic operation with another Tensor with the same shape.

4.3 Tensor-scalar arithmetic

Tensor can also performs arithmetic operation with scalar.

4.4 Equivalent APIs

Following are some equivalent APIs that are also provided in Cytnx for users who are familiar and coming from pytorch and other librariy communities.

Note: 1. All the arithmetic operation function such as **Add,Sub,Mul,Div...**, as well as linear algebra functions all start with capital characters. While in pytorch, they are all lower-case. 2. All the arithmetic operations with a underscore (such as **Add_, Sub_, Mul_, Div_**)are the inplace version that modify the current instance.

Hint:

1. ComplexDouble/ComplexFloat/Double/Float, these 4 types internally calls BLAS/cuBLAS/MKL ?axpy when the inputs are in the same types.
2. Arithmetic between other types (Including different types) are accelerated with OpenMP on CPU. For GPU, custom kernels are used to perform operation.

2.2 Linear algebra

2.3 Iterative solver

2.4 Cytnx extensions

2.5 linalg extension

CHAPTER

THREE

EXAMPLES

Here, we provide some examples of commonly seen Quantum simualtion methods.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search