Class uni10::Qnum

Quantum number of states

A Qnum is comprised of one or more eigenvalues of following three symmetry operators:

U1: If the system has some U1 symmetry, for example, conservation of total Sz or total particle number, we can write down the state with the U1 symmetry eigenstates. U1 eigenvalues here is the U1 in Qnum of the state.

Z2(parity of Bosonic system): it corresponds to the conservation of the parity of the total particle number, or the parity of total spin up(or down) sites in spin system.

Z2(parity of Fermionic system): This symmetry is reserved for the parity of the total particle number in fermionic system. In fermionic system, this symmetry is used to take care of the fermionic signs in operations.

relavant datatype

uni10::**parityType**

```
enum parityType{ PRT_EVEN = 0, PRT_ODD = 1};
```

Type of quantum number parity, belong to Z_2 symmetry group.

uni10::**parityFType**

```
enum parityType{ PRT_EVEN = 0, PRT_ODD = 1};
```

Type of quantum number parity for **fermionic system**, belong to Z_2 symmetry group.

member functions:

uni10::Qnum::**Qnum**

```
(1) Qnum(int U1 = 0, parityType prt = PRT_EVEN);
(2) Qnum(parityFType prtF, int U1=0, parityType prt=PRT_EVEN);
(3) Qnum(const Qnum& qnum);
```

Construct Qnum

(1) Constructs a quantum number for bosonic system.
(2) Constructs a quantum number for fermionic system.
(3) Copy constructor.

see example: ***egQ1***

Parameters

U1: int, optional(0)

Initial U1 value for quantum number. the value must bounded by (uni10::Qnum::U1_UPB, uni10::Qnum::U1_LOB).

prt: parityType, optional(PRT_EVEN)

Initial parity value for quantum number.

prtF: parityFType

Initial fermionic parity value for quantum number.

uni10::Qnum::**~Qnum**

```
~Qnum();
```

Destruct Qnum

Destroys Qnum.

uni10::Qnum::**assign**

```
(1) void assign(int U1 = 0, parityType prt = PRT_EVEN);
(2) void assign(parityFType prtF, int U1 = 0, parityType prt = PRT_EVEN);
```

Assign Qnum content

Assigns new content to Qnum, replacing its current content.
(2) assigns a fermionic quantum number.

Parameters

U1: int, optional(0)

Initial U1 value for quantum number. the value must bounded by (uni10::Qnum::U1_UPB, uni10::Qnum::U1_LOB).

prt: parityType, optional(PRT_EVEN)

Initial parity value for quantum number.

prtF: parityFType

Initial fermionic parity value for quantum number.

uni10::Qnum::**U1**

```
int U1()const;
```

Return U1

Returns the value of U1 quantum number.

Return Value

The value of U1 quantum number, bounded by (uni10::Qnum::U1_UPB, uni10::Qnum::U1_LOB).

uni10::Qnum::**prt**

parityType prt()const;

Return parity

Returns the value of parity quantum number.

Return Value

The value of parity quantum number.

uni10::Qnum::**prtF**

parityFType prtF()const;

Return parity

Returns the value of fermionic parity quantum number.

Return Value

The value of fermionic parity quantum number.

static member functions:

uni10::Qnum::**isFermionic**

static bool isFermionic();

Test whether the system is fermionic

Tests whether fermionic parity *PRTF_ODD* exists in the system,

Return Value

true if the fermionic odd parity exists, *false* otherwise.

Example: **egQ1**

source: http://uni10.org/examples/egQ1.cpp

```
#include <iostream>
#include <uni10.hpp>

int main(){
  // U1 = 1, parity even.
  uni10::Qnum q10(1, uni10::PRT_EVEN);
  // U1 = -1, parity odd.
  uni10::Qnum q_11(-1, uni10::PRT_ODD);

  std::cout<<"q10: "<<q10<<std::endl;
  std::cout<<"q_11: "<<q_11<<std::endl;
  std::cout<<"q_11: U1 = "<<q_11.U1()<<", parity = "<<q_11.prt()<<std::endl;
  q_11.assign(-2, uni10::PRT_EVEN);
  std::cout<<"q_11(after assign): "<<q_11<<std::endl;
  // check the for fermionic
  std::cout<<"isFermioinc: "<<uni10::Qnum::isFermionic()<<std::endl ;

  // Fermionic system
  std::cout<<"----- Fermionic -----\n";
  // fermionic parity even, U1 = 1, parity even.
  uni10::Qnum f0_q10(uni10::PRTF_EVEN, 1, uni10::PRT_EVEN);
  // fermionic parity odd, U1 = 1, parity even.
  uni10::Qnum f1_q10(uni10::PRTF_ODD, 1, uni10::PRT_EVEN);

  std::cout<<"f0_q10: "<<f0_q10<<std::endl;
  std::cout<<"f1_q10: "<<f1_q10<<std::endl;
  std::cout<<"f1_q10: fermionic parity = " <<f1_q10.prtF()<<std::endl;
  std::cout<<"isFermioinc: "<<uni10::Qnum::isFermionic()<<std::endl;

  return 0;
}
```

Output:

```
q10: (U1 = 1, P = 0, 0)
q_11: (U1 = -1, P = 1, 0)
q_11: U1 = -1, parity = 1
q_11(after assign): (U1 = -2, P = 0, 0)
isFermioinc: 0
----- Fermionic -----
f0_q10: (U1 = 1, P = 0, 0)
f1_q10: (U1 = 1, P = 0, 1)
f1_q10: fermionic parity = 1
isFermioinc: 1
```

non-member overloads:

operator<

friend bool operator< (const Qnum& q1, const Qnum& q2);

friend bool operator<= (const Qnum& q1, const Qnum& q2);

Define less than operator

Defines q1 < q2

Parameters

q1, q2: Qnum

Two *Qnums* being compared.

Return Value

true if q1 < q2, *false* otherwise.

operator==

```
friend bool operator== (const Qnum& q1, const Qnum& q2);
```

Define equal operator

Defines q1 == q2

Parameters

q1, q2: Qnum

Two *Qnums* being compared.

Return Value

true if q1 == q2, *false* otherwise.

operator-

```
friend Qnum operator- (const Qnum& q1);
```

Define minus operator

Defines the minus sign behavior. Minus sign is defined by the change be when quantum number on in-coming bond *permute* to out-coming bond, or vice versa.

For U1 symmetry in Qnum, minus sign corresponds to minus U1 value. As for parity, minus sign have no effect.

Parameters

q1: Qnum

Original *Qnum*

Return Value

Returns the resulting Qnum q = -q1.

operator*

```
friend Qnum operator* (const Qnum& q1, const Qnum& q2);
```

Define multiplication operator

Defines the fusion rules for quantum numbers. For U1 symmetry in *Qnum*, *q1 * q2* corresponds to *q1.U1() + q2.U1()*. For parity, *q1 * q2* corresponds to *q1.prt() ^ q2.prt(). (q1.prtF() ^ q2.prtF() in fermionic case)*

Parameters

q1, q2: Qnum

Two *Qnums* being multiplied, q1 multiplied by q2.

Return Value

Returns the resulting Qnum q = q1 * q2.

operator<<

```
friend std::ostream& operator<< (std::ostream& os, const Qnum& q);
```

Print out Qnum

Prints out a quantum number Qnum *q* as: (for example)

std::cout << q;

```
 (U1 = 1, P = 0, 1)
```

Which means *q.U1()* is 1, *q.prt()* is 0(*PRT_EVEN*) and *q.prtF()* is 1(*PRTF_ODD*)

Parameters

os: std::ostream

ostream in standard library, see http://www.cplusplus.com/reference/ostream/ostream/?kw=ostream

q: Qnum

Qnum to be printed out.

Return Value

Returns *std::ostream&*

Class uni10::Bond

Bond of a tensor

A tensor is comprised of bonds. The number of bonds corresponds to the rank of a tensor. A bond usually represents states of a particular basis of a physical system. For example, a bond could represents the states of a spin 1 particle, which has three possible states in the basis of Sz, -1, 0, 1. In this case, the dimension of bond is three. If there are some symmetries in the system, each state of bond can has a *Qnum*, which is the eigenvalue of the symmetry operators. For example, the conservation of total Sz in spin system(U1 symmetry), if we choose Sz as our basis, we can have a bond of three *Qnum* of *U1* values corresponding to the Sz eigenvalues.

relavant datatype

uni10::**bondType**

```
enum bondType{ BD_IN = 1, BD_OUT = -1 };
```

Two types of bond, in-coming bond, out-going bond.

member functions

uni10::Bond::**Bond**

```
(1) Bond(bondType, std::vector<Qnum>& qnums);
(2) Bond(const Bond& bd);
```

Construct Bond

Constructs a Bond, initializing depending on the constructor version used:

(1) Constructs a Bond by the given *Qnum* vector *qnums*. *qnums* is the quantum numbers of the states on the bond.

(2) Copy constructor.

Parameters

qnums: std::vector<Qnum>

Quantum number array to fill the bond with.

bd: Bond

Bond to be copied.

uni10::Bond::~**Bond**

```
~Bond();
```

Destruct Bond

Destroys a *Bond*.

uni10::Bond::**assign**

```
void assign(bondType, std::vector<Qnum>& qnums);
```

Assign bond content

Assigns new quantum numbers, *bondType* and dimension to *Bond*, replacing its current content.

Parameters

qnums: std::vector<Qnum>

Quantum number array to fill the bond with.

uni10::Bond::**type**

```
bondType type()const;
```

Return type

Returns the type of a bond, either BD_IN or BD_OUT.

Return Value

The type of the bond.

uni10::Bond::**dim**

```
int dim()const;
```

Return dimension

Returns the dimension of a bond, that is, the number of quantum states.

Return Value

The dimension of the bond.

uni10::Bond::**degeneracy**

```
std::map<Qnum, int> degeneracy()const;
```

Return the numbers of degeneracy of various Qnum

Returns a map, which shows the degeneracy of various *Qnum*.

Return Value

The mapping of *Qnum* to its degeneracy value.

uni10::Bond::**Qlist**

```
std::vector<Qnum> Qlist()const;
```

Return its quantum number array

Returns an array of *Qnum* of states in the bond. the size of the vector is the same as the dimension of the bond.

Return Value

The vector of its *Qnum* array.

uni10::Bond::**change**

```
void change(bondType type);
```

Change type of a bond

Changes the type of the bond, together with the *Qnums* of the bond. If bond is changed from incoming(BD_IN) to out-going(BD_OUT) type or vice versa, the change of its *Qnums* is defined by the minus sign “-” of *Qnum* itself.

Parameters

type: bondType

Type to change to.

uni10::Bond::**combine**

```
Bond& combine(const Bond bd);
```

Combine a bond

Combines another bond *db*, expand the bond dimension by direct product of it *Qnums* and the *Qnums* of the given bond *bd*. The resulting bondType is unchanged.

Parameters

bd: Bond

The bond to be combined.

Return Value

The resulting *Bond.*

static member functions:

uni10::Bond::**combine**

```
(1)  static Bond combine(const std::vector<Bond>& bds);
```

(2) static Bond combine(bondType type, const std::vector<Bond>& bds);

Combine bonds

Combines an array of bonds by successively combining the bonds in the order of bonds in the given array *bds*.

(1) The resulting Bond is of type of the first bond in *bds.*

(2) The resulting Bond is of type of the given *bondType*.

Parameters

bds: std::vector<Bond>

An array of *Bond* to be combined.

type: bondType

The type for the resulting bond to be.

Return Value

The resulting *Bond*.

non-member overloads:

operator==

friend bool operator== (const Bond& b1, const Bond& b2);

Compare two bonds

The equality condition is that:
b1.type() == b2.type() && b1.Qlist() == b2.Qlist()

Parameters

b1, b2: Bond

The bonds to be compared.

Return Value

true if b1 == b2, *false* otherwise.

operator<<

friend std::ostream& operator<< (std::ostream& os, const Bond& b);

Print out Qnum

Prints out a bond as(for example):
std::cout << b;

```
IN : (U1 = 1, P = 0, 0)|1, (U1 = 0, P = 0, 0)|2, (U1 = −1, P = 0, 0)|1, Dim = 4
```

In above example, it is in-coming bond with three Qnums: one U1=1, two U1=0 and one U1=-1. The dimension of the bond is 4.

Parameters

os: std::ostream&

ostream in standard library, see http://www.cplusplus.com/reference/ostream/ostream/?kw=ostream

b: Bond

Bond to be printed out.

Return Value

Returns *std::ostream&*

Class uni10::**Matrix**

A common matrix

Class *Matrix* is made for some linear algebra operations on matrix. In addition, *Matrix* is perfectly cooperate with the class *UniTensor*. A tensor with symmetries contains blocks of various *Qnums*. Each block is a matrix with tensor elements. We can take a block out of a tensor as a *Matrix*, do whatever operations you want on the matrix elements and put *Matrix* back to the tensor. The *Matrix* follows C convention that it is row-major and indices start from 0. For now, *Matrix* only supports datatype *double.*

member functions:

uni10::Matrix::**Matrix**

```
(1) Matrix(int Rnum, int Cnum, bool diag=false);
(2) Matrix(int Rnum, int Cnum, double* elem, bool diag=false);
(3) Matrix(const Matrix& m);
```

Construct Matrix

Constructs a *Matrix*, initializing depending on the constructor version used:

(1) Allocate memory of size Rnum * Cnum(or min(*Rnum, Cnum*) if *diag* is *true*) for matrix elements and set the elements to zero

(2) Allocate memory of size Rnum * Cnum(or min(*Rnum, Cnum*) if *diag* is *true*) for matrix elements and copy the elements from the given *elem*.

(3) copy constructor. The properties of the *Matrix* are copied. It allocates new memory for elements and copied the content from the given *Matrix m*.

Parameters

Rnum: int

The number of rows of the matrix.

Cnum: int

The number of columns of the matrix.

diag: bool, optional(false)

If it’s true, only the memory of the diagonal elements are allocated.

m: Matrix

Another *Matrix* being copied from.

uni10::Matrix::**~Matrix**

```
~Matrix();
```

Destruct Matrix

Destroys the *Matrix* and freeing all the allocated memory for matrix elements.

uni10::Matrix::**row**

```
int row()const;
```

Return number of rows

Returns the number of rows of the *Matrix*.

Return Value

Number of rows

uni10::Matrix::**col**

```
int col()const;
```

Return number of columns

Returns the number of columns of the *Matrix*.

Return Value

Number of columns.

uni10::Matrix::**isDiag**

```
bool isDiag()const;
```

Test whether *Matrix* is diagonal

Returns whether the *Matrix* is diagonal

Return Value

true if diagonal, *false* otherwise.

uni10::Matrix::**elemNum**

```
size_t elemNum()const;
```

Return size

Returns the number of allocated elements. If diagonal, the return value is equal to the number of diagonal elements.

Return Value

The number of elements in *Matrix*

uni10::Matrix::**operator[]**

```
double& operator[](size_t idx);
```

Access element

Returns a reference to the element at position *idx* in the *Matrix*. The value *idx* is serial index counted in row-major from the first element(*idx* = 0) of the *Matrix*.
This function works similar to member function *Matrix::at()*.

Parameters

idx: int

Position of an element in the *Matrix*

Return Value

The element at the specified position in the *Matrix*.

uni10::Matrix::**at**

```
double& at(int i, int j);
```

Access element

Returns a reference to the element in *i*-th row and j-th column of the *Matrix*. The values *i* and *j* are counted from 0.

Parameters

i, j: int

Index of the *Matrix*

Return Value

The element at the index (*i*, *j*) in the *Matrix*.

uni10::Matrix::**elem**

```
double* elem()const;
```

Access element

Returns a pointer of type *double* to the *Matrix* elements.

Return Value

double pointer of the *Matrix* elements

uni10::Matrix::**operator=**

```
Matrix& operator=(const Matrix& mat);
```

Assign Matrix

Assigns new content to the *Matrix* from the given matrix *mat*, replacing the original memory of elements by reallocating new memory fit for *mat*.

Parameters

mat: Matrix

The *Matrix* to be copied from.

uni10::Matrix::**addElem**

```
void addElem(double* elem);
```

Copy elements

Copies the first *elemNum()* elements from the given double pointer elem, replacing the original ones.

Parameters

elem: double pointer

The *Matrix* elements to be copied from.

uni10::Matrix::**save**

```
void save(const std::string& fname);
```

Output to file

Writes the elements of the *Matrix* out to a binary file of file name *fname*. The output file size is *elemNum*()*sizeof(*double*).

Parameters

fname: std::string

File name to write elements out.

uni10::Matrix::**load**

```
void load(const std::string& fname);
```

Input from file

Reads the elements of the *Matrix* from the binary file of file name *fname*. Reads in *double* array of size *elemNum*()*sizeof(*double*) from file stream and replacing the origin elements.

Parameters

fname: std::string

File name to read elements in.

uni10::Matrix::**set_zero**

```
void set_zero();
```

Assign elements

Sets all the double elements of the *Matrix* to zero.

uni10::Matrix::**randomize**

```
void randomize();
```

Assign elements

Randomly assigns double values ranged (0, 1) to the elements of the *Matrix*.

uni10::Matrix::**orthoRand**

```
void orthoRand();
```

Assign elements

Randomly generates orthogonal bases. Assigns to the elements of the *Matrix*.
Let *Nr* = *row*() and *Nc* = *col*().
If the *Nr* < *Nc*, randomly generates *Nr*'s orthogonal bases with each basis having dimension *Nc*, generating *Nr*'s row-vectors of size *Nc*.
If the *Nr* > *Nc*, randomly generates *Nc*'s orthogonal bases with each basis having dimension *Nr*, generating *Nc*'s column-vectors of size *Nr.*

uni10::Matrix::**transpose**

```
void transpose();
```

Transpose Matrix

Exchanges the number of rows and the number of columns and transposes the elements of the *Matrix*.

uni10::Matrix::**diagonalize**

```
std::vector<Matrix> diagonalize();
```

Perform diagonalization on the Matrix

Diagonalizes the *Matrix* and returns the a vector of two matrices of diagonalization.
For a *n* by *n* matrix *A*:
A = *UT* * *D* * *U*
The operation is a wrapper of Lapack function *dsyev*().

Return Value

A vector of Matrices [*D*, *U*] of the diagonalization results.
For *n* by *n Matrix*:
D is *n* by *n* diagonal *Matrix* of eigenvalues.
U is *n* by *n* row-vectors of eigenvectors.

uni10::Matrix::**svd**

```
std::vector<Matrix> svd();
```

Perform SVD on the Matrix

Performs singular value decomposition(SVD) on the Matrix and returns a vector of three resulting matrices of SVD.
For *m* by *n* matrix *A*, it is decomposed as:
A = *U* * *Σ* * *VT*
The operation is a wrapper of Lapack function *dgesvd*().

Return Value

A vector of Matrices [*U*, *Σ, VT*] of the diagonalization results.
For *m* by *n Matrix:*
U is m by m row-major matrix.
Σ is m by n diagonal matrix.
VT is n by n row-major matrix.

uni10::Matrix::**trace**

```
double trace();
```

Trace of the Matrix

Performs trace of the *Matrix.*

Return Value

Trace value of the *Matrix*.

uni10::Matrix::**operator*=**

```
(1) Matrix& operator*= (const Matrix& Mb);
(2) Matrix& operator*= (double a);
```

Perform multiplication

(1) Performs matrix multiplication with another Matrix *Mb.*
(2) Performs element-wise multiplication with a scalar *a* of type double.

Parameters

Mb: Matrix

The given matrix to multiplied with.

a: double

The scalar of type *double* to multiplied with.

Return Value

The resulting *Matrix*.

uni10::Matrix::**operator+=**

```
Matrix& operator+= (const Matrix& Mb);
```

Perform additions of elements

Performs element by element addition.

Parameters

Mb: Matrix

The given matrix to add.

Return Value

The resulting *Matrix*.

non-member overloads:

operator*

```
(1) friend Matrix operator* (const Matrix& Ma, const Matrix& Mb);
(2) friend Matrix operator*(const Matrix& Ma, double a);
(3) friend Matrix operator*(double a, const Matrix& Ma)
```

Perform multiplication

(1) Performs matrix multiplication, *Ma* * *Mb*;
(2) Performs element-wise multiplication with a scalar *a* of type double, *Ma* * *a*
(3) The same as (2), *a * Ma*

Parameters

Ma, Mb: Matrix

Matrices for matrix multiplication *Ma* * *Mb*

a: double

The scalar of type *double* to multiplied with *Matrix*.

Return Value

The resulting *Matrix*.

operator+

```
friend Matrix operator+(const Matrix& Ma, const Matrix& Mb);
```

Matrix addition

Performs element-wise additions on matrix *Ma* and *Mb*, *Ma* + *Mb*.

Parameters

Ma, Mb: Matrix

Matrices for matrix addition *Ma* + *Mb*

Return Value

The resulting *Matrix*.

operator==

```
friend bool operator== (const Matrix& Ma, const Matrix& Mb);
```

Compare Matrices

Compare all the elements in *Ma* and *Mb*, returning *true* if all the elements are the same, false otherwise.

Parameters

Ma, Mb: Matrix

Matrices for comparison.

Return Value

true if all the elements in *Ma* and *Mb* are the same, *false* otherwise.

operator<<

```
std::ostream& operator<< (std::ostream& os, const Matrix& M);
```

Print out Matrix

Prints out a *Matrix* as(for example):
std::cout << M;

```
2 x 3 = 6

 -0.254 -0.858 -0.447

  0.392  0.331 -0.859
```

In the above example, M is a 2 by 3 matrix with number of elements 6 and the following 2 by 3 matrix are its elements.

Parameters

os: std::ostream

ostream in standard library, see http://www.cplusplus.com/reference/ostream/ostream/?kw=ostream

M: Matrix

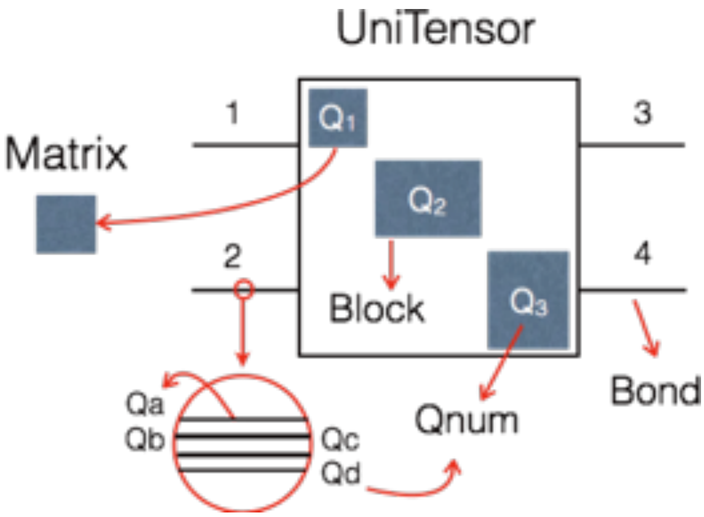
Matrix to be printed out.

Return Value

Returns *std::ostream&*

Class uni10::**UniTensor**

Universal Tensor



Class *UniTensor* is made for tensor contractions and permutes of tensor *Bonds*(Indices) with symmetry block diagonal elements. A tensor is comprised of *Bonds* and blocked elements with various quantum number *Qnum*(*Q1*, *Q2*, *Q3* in the figure). The *Qnums* on the *Bonds* decide the sizes of those *Qnum* blocks and the rank of tensor is decided by the number of Bonds. Each Bond has a label(1, 2, 3, 4 in the figure). Labels are used to manipulate tensors, such as in operations *permute*, *partialTrace* and tensor contraction. For manipulation of tensor elements, one can use *getBlock* function to take out block elements out as a *Matrix*, performs whatever operations on it and puts it back with function *putBlock*. For the operations about *Matrix*, see class *Matrix*.

member functions

uni10::UniTensor::**UniTensor**

(1) UniTensor(double val = 1.0);
(2) UniTensor(const std::vector<Bond>& bonds, const std::string& name = “");
(3) UniTensor(const std::vector<Bond>& bonds, int* labels, const std::string& name = “");
(4) UniTensor(const std::vector<Bond>& bonds, std::vector<int>& labels, const std::string& name = “");
(5) UniTensor(const std::string& fname);
(6) UniTensor(const UniTensor& uT);

Construct UniTensor

Constructs a *UniTensor*, initializing depending on the constructor version used:

(1) Constructs without *Bonds* to create a rank 0 tensor, a scalar. The default value for the scalar is 1.0.
(2) Constructs a tensor with the given *Bond* array *bonds*, allocating memories for blocks of sizes decided by the input *bonds* and naming the tensor *name* if given.
(3) Constructs a tensor with the given *Bond* array *bonds* and int array *labels*. It initializes the same way as (2) above, then assigns *labels* to the *Bonds* of the *UniTensor*.
(4) It is similar to (3). Instead of taking int* *labels*, it takes std::vector type *labels*.
(5) Loads a tensor from the binary file of file name *fname*. Note that, the file *fname* is a binary file of specific format which is generated by the function *UniTensor*::*save*(*fname*).
(6) Copy constructor. The properties of the *UniTensor* are copied. It allocates new memory for elements and copied the content from the given *UniTensor uT*.

Parameters

val: double, optional(1.0)

The value for the rank 0 *UniTensor*(1).

bonds: std::vector<Bond>

An array of bonds of the *UniTensor*. For initializing the structure of tensor.

name: std::string, optional(“”)

Given name of the *UniTensor*.

fname: std::string

The file name of the *UniTensor* being read in

uT: UniTensor

Another *UniTensor* being copied from.

uni10::UniTensor::**~UniTensor**

```
~UniTensor();
```

Destruct UniTensor

Destroys the *UniTensor* and freeing all the allocated memory for tensor content.

uni10::UniTensor::**addLabel**

```
(1) void addLabel(int* newLabels);
(2) void addLabel(const std::vector<int>& newLabels);
```

Parameters

newLabels: int*

newLabels: std::vector<int>

uni10::UniTensor::**label**

```
(1) std::vector<int> label()const;
(2) int label(int idx)const;
```

Parameters

idx: int

The position of the *label* being retrieved.

uni10::UniTensor::**bondNum**

```
size_t bondNum()const;
```

uni10::UniTensor::**inBondNum**

```
int inBondNum()const;
```

uni10::UniTensor::**bond**

```
(1) std::vector<Bond> bond()const;
(2) Bond bond(int idx)const;
```

Parameters

idx: int

The position of the *bond* being retrieved.

uni10::UniTensor::**addRawElem**

```
void addRawElem(double* rawElem);
```

Parameters

rawElem: double*

uni10::UniTensor::**blockNum**

```
size_t blockNum()const;
```

uni10::UniTensor::**blockQnum**

```
(1) std::vector<Qnum> blockQnum()const;
(2) Qnum blockQnum(int idx)const;
```

Parameters

idx: int

The position of the blocked elements being retrieved.

uni10::UniTensor::**getBlocks**

```
std::map<Qnum, Matrix> getBlocks()const;
```

uni10::UniTensor::**getBlock**

```
Matrix getBlock(Qnum qnum, bool diag = false)const;
```

Parameters

qnum: Qnum

diag: bool, optional(false)

uni10::UniTensor::**putBlock**

```
void putBlock(const Qnum& qnum, Matrix& mat);
```

Parameters

qnum: Qnum

mat: Matrix

uni10::UniTensor::**elemNum**

```
size_t elemNum()const;
```

uni10::UniTensor::**at**

```
double at(std::vector<int>idxs)const;
```

Parameters

idxs: std::vector<int>

uni10::UniTensor::**operator[]**

```
double& operator[](size_t idx);
```

Parameters

idx: int

uni10::UniTensor::**rawElem**

```
Matrix rawElem()const;
```

uni10::UniTensor::**printRawElem**

```
void printRawElem()const;
```

uni10::UniTensor::**setName**

```
void setName(const std::string& name);
```

Parameters

name: std::string

uni10::UniTensor::**getName**

```
std::string getName();
```

uni10::UniTensor::**save**

```
void save(const std::string& fname);
```

Parameters

fname: std::string

uni10::UniTensor::**similar**

```
bool similar(const UniTensor& Tb)const;
```

Parameters

Tb: UniTensor

uni10::UniTensor::**elemCmp**

```
bool elemCmp(const UniTensor& uT)const;
```

Parameters

uT: UniTensor

uni10::UniTensor::**check**

```
void check();
```

uni10::UniTensor::**permute**

```
(1) UniTensor& permute(int* newLabels, int inBondNum);
(2) UniTensor& permute(std::vector<int>& newLabels, int inBondNum);
```

Parameters

newLabels: int*

newLabels: std::vector<int>

inBondNum: int

uni10::UniTensor::**transpose()**

```
UniTensor& transpose();
```

uni10::UniTensor::**combineBond**

```
UniTensor& combineBond(const std::vector<int>& combined_labels);
```

Parameters

combined_labels: std::vector<int>

uni10::UniTensor::**partialTrace**

```
UniTensor& partialTrace(int la, int lb);
```

Parameters

la, lb: int

uni10::UniTensor::**trace**

```
double trace()const;
```

uni10::UniTensor::**set_zero**

```
(1) void set_zero();
(2) void set_zero(const Qnum& qnum);
```

Parameters

qnum: Qnum

uni10::UniTensor::**eye**

```
(1) void eye();
(2) void eye(const Qnum& qnum);
```

Parameters

qnum: Qnum

uni10::UniTensor::**randomize**

```
void randomize();
```

uni10::UniTensor::**orthoRand**

```
(1) void orthoRand();
(2) void orthoRand(const Qnum& qnum);
```

Parameters

qnum: Qnum

uni10::UniTensor::**exSwap**

```
std::vector<_Swap> exSwap(const UniTensor& Tb)const;
```

Parameters

Tb: UniTensor

uni10::UniTensor::**addGate**

```
void addGate(std::vector<_Swap> swaps);
```

Parameters

os: std::ostream&

uni10::UniTensor::**operator*=**

```
(1) UniTensor& operator*= (UniTensor& Tb);
(2) UniTensor& operator*= (double a)
```

Parameters

Tb: UniTensor

a: double

uni10::UniTensor::**operator+=**

```
UniTensor& operator+= (const UniTensor& Tb);
```

Parameters

Tb: UniTensor

non-member overloads:

operator*

```
(1) friend UniTensor operator* (UniTensor& Ta, UniTensor& Tb);
(2) friend UniTensor operator* (const UniTensor& Ta, double a);
(3) friend UniTensor operator* (double a, const UniTensor& Ta)
```

Parameters

Ta, Tb: UniTensor

a: double

operator+

```
friend UniTensor operator+ (const UniTensor& Ta, const UniTensor& Tb);
```

Parameters

Ta, Tb: UniTensor

operator<<

```
friend std::ostream& operator<< (std::ostream& os, const UniTensor& uT);
```

Parameters

os: std::ostream

ostream in standard library, see http://www.cplusplus.com/reference/ostream/ostream/?kw=ostream

uT: UniTensor

Class uni10::**Network**

member functions

uni10::Network::**Network**

```
Network(const std::string& fname);
Network(const std::string& fname, const std::vector<UniTensor*>& uTptrs);
```

Parameters

fname: std::string

uTptrs: std::vector<UniTensor*>

uni10::Network::**~Network**

```
~Network();
```

uni10::Network::**putTensor**

```
void putTensor(int idx, const UniTensor* uTptr, bool force=false);
```

Parameters

uTptr: UniTensor*

uni10::Network::**launch**

```
UniTensor launch(const std::string& name=“");
```

Parameters

name: std::string

non-member overloads

operator<<

```
friend std::ostream& operator<< (std::ostream& os, Network& net);
```

Parameters

os: std::ostream

ostream in standard library, see http://www.cplusplus.com/reference/ostream/ostream/?kw=ostream

net: Network