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.
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::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

```
1) #include <iostream>
2) #include <uni10.hpp>
3)
4) int main(){
5)   // U1 = 1, parity even.
6)   uni10::Qnum q10(1, uni10::PRT_EVEN);
7)   // U1 = -1, parity odd.
8)   uni10::Qnum q_11(-1, uni10::PRT_ODD);
9)
10)  std::cout<<"q10: "<<q10<<std::endl;
```

```
11)  std::cout<<"q_11: "<<q_11<<std::endl;
12)  std::cout<<"q_11: U1 = "<<q_11.U1()<<", parity = "<<q_11.prt()<<std::endl;
13)  q_11.assign(-2, uni10::PRT_EVEN);
14)  std::cout<<"q_11(after assign): "<<q_11<<std::endl;
15)  // check the for fermionic
16)  std::cout<<"isFermioinc: "<<uni10::Qnum::isFermionic()<<std::endl ;
17)
18)  // Fermionic system
19)  std::cout<<"----- Fermionic -----\n";
20)  // fermionic parity even, U1 = 1, parity even.
21)  uni10::Qnum f0_q10(uni10::PRTF_EVEN, 1, uni10::PRT_EVEN);
22)  // fermionic parity odd, U1 = 1, parity even.
23)  uni10::Qnum f1_q10(uni10::PRTF_ODD, 1, uni10::PRT_EVEN);
24)
25)  std::cout<<"f0_q10: "<<f0_q10<<std::endl;
26)  std::cout<<"f1_q10: "<<f1_q10<<std::endl;
27)  std::cout<<"f1_q10: fermionic parity = " <<f1_q10.prtF()<<std::endl;
28)  std::cout<<"isFermioinc: "<<uni10::Qnum::isFermionic()<<std::endl;
29)
30)  return 0;
31)}
```

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&*

Example: **egQ2**

source: http://uni10.org/examples/egQ2.cpp

```
1) int main(){
2)   // U1 = 1, parity even.
3)   uni10::Qnum q10(1, uni10::PRT_EVEN);
4)   // U1 = -1, parity odd.
5)   uni10::Qnum q_11(-1, uni10::PRT_ODD);
6)
7)   std::cout<<"q10: "<< q10 << std::endl;
8)   std::cout<<"q_11: "<< q_11 << std::endl;
9)
10)  std::cout<<"----- Operations -----\n";
11)  std::cout<<"-q_11 = " << -q_11 << std::endl;
12)  std::cout<<"q10 * q_11 = " << q10 * q_11 <<std::endl;
13)  std::cout<<"q10 * (-q_11) = " << q10 * (-q_11) <<std::endl;
14)
15)  return 0;
16)}
```

Output:

```
q10: (U1 = 1, P = 0, 0)
q_11: (U1 = -1, P = 1, 0)
----- Operations -----
-q_11 = (U1 = 1, P = 1, 0)
q10 * q_11 = (U1 = 0, P = 1, 0)
q10 * q_11 = (U1 = 2, P = 1, 0)
```

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

Construct Bond

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

(1) Constructs a Bond with dimension *dim* and without symmetry quantum number

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

(3) 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**

```
(1) void assign(bondType, int dim);
(2) void assign(bondType, const std::vector<Qnum>& qnums);
```

Assign bond content

Assigns new quantum numbers, *bondType* and dimension *dim* 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&*

Example: **egB1**

source: http://uni10.org/examples/egB1.cpp

```
1) #include <iostream>
2) #include <uni10.hpp>
3)
4) int main(){
5)   uni10::Qnum q1(1);
6)   uni10::Qnum q0(0);
7)   uni10::Qnum q_1(-1);
8)   // Create an array of Qnums for the states of a bond.
9)   std::vector<uni10::Qnum> qnums;
10)  qnums.push_back(q1);
11)  qnums.push_back(q1);
12)  qnums.push_back(q0);
13)  qnums.push_back(q0);
14)  qnums.push_back(q0);
15)  qnums.push_back(q_1);
16)
17)  // Constrcut Bond with Qnum array
18)  uni10::Bond bd(uni10::BD_IN, qnums);
19)  // Print out a Bond
20)  std::cout<<"Bond bd: \n"<<bd<<std::endl;
21)  std::cout<<"Bond type: "<<bd.type()<<"(IN)"<<", Bond dimension:
     "<<bd.dim()<<std::endl<<std::endl;
22)
23)  // List the degeneracy of states
24)  std::cout<<"Degeneracies: "<<std::endl;
25)  std::map<uni10::Qnum, int> degs = bd.degeneracy();
26)  for(std::map<uni10::Qnum,int>::const_iterator it=degs.begin(); it!
     =degs.end(); ++it)
27)      std::cout<<it->first<<": "<<it->second<<std::endl;
28)  std::cout<<std::endl;
29)
30)  std::vector<uni10::Qnum> qlist = bd.Qlist();
31)  std::cout<<"Qnum list: "<<std::endl;
32)  for(int i = 0; i < qlist.size(); i++)
33)      std::cout<<qlist[i]<<", ";
```

```
34) std::cout<<std::endl<<std::endl;
35)
36) // Change bond type
37) bd.change(uni10::BD_OUT);
38) std::cout<<"bd changes to BD_OUT:\n"<<bd<<std::endl;
39)
40) bd.change(uni10::BD_IN);
41)
42) // Combine bond
43) qnums.clear();
44) qnums.push_back(q1);
45) qnums.push_back(q0);
46) qnums.push_back(q0);
47) qnums.push_back(q_1);
48) uni10::Bond bd2(uni10::BD_IN, qnums);
49) std::cout<<"Bond bd2: \n"<<bd2<<std::endl;
50)
51) // bd.combine(bd2);
52) std::cout<<"bd2.combine(bd): \n"<<bd2.combine(bd)<<std::endl;
53)
54) std::cout<<"Degeneracies of bd2 after combining bd: "<<std::endl;
55) degs = bd2.degeneracy();
56) for(std::map<uni10::Qnum,int>::const_iterator it=degs.begin(); it!
    =degs.end(); ++it)
57)     std::cout<<it->first<<": "<<it->second<<std::endl;
58) std::cout<<std::endl;
59)
60) return 0;
61)}
```

Output:

```
Bond bd:
IN : (U1 = 1, P = 0, 0)|2, (U1 = 0, P = 0, 0)|3, (U1 = -1, P = 0, 0)|1, Dim = 6

Bond type: 1(IN), Bond dimension: 6

Degeneracies:
(U1 = -1, P = 0, 0): 1
(U1 = 0, P = 0, 0): 3
(U1 = 1, P = 0, 0): 2

Qnum list:
(U1 = 1, P = 0, 0), (U1 = 1, P = 0, 0), (U1 = 0, P = 0, 0), (U1 = 0, P = 0, 0),
(U1 = 0, P = 0, 0), (U1 = -1, P = 0, 0),

bd changes to BD_OUT:
OUT: (U1 = -1, P = 0, 0)|2, (U1 = 0, P = 0, 0)|3, (U1 = 1, P = 0, 0)|1, Dim = 6

Bond bd2:
IN : (U1 = 1, P = 0, 0)|1, (U1 = 0, P = 0, 0)|2, (U1 = -1, P = 0, 0)|1, Dim = 4

bd2.combine(bd):
IN : (U1 = 2, P = 0, 0)|2, (U1 = 1, P = 0, 0)|3, (U1 = 0, P = 0, 0)|1, (U1 = 1, P
= 0, 0)|2, (U1 = 0, P = 0, 0)|3, (U1 = -1, P = 0, 0)|1, (U1 = 1, P = 0, 0)|2, (U1
```

```
= 0, P = 0, 0)|3, (U1 = −1, P = 0, 0)|1, (U1 = 0, P = 0, 0)|2, (U1 = −1, P = 0,
0)|3, (U1 = −2, P = 0, 0)|1, Dim = 24

Degeneracies of bd2 after combining bd:
(U1 = −2, P = 0, 0): 1
(U1 = −1, P = 0, 0): 5
(U1 = 0, P = 0, 0): 9
(U1 = 1, P = 0, 0): 7
(U1 = 2, P = 0, 0): 2
```

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. Notice that, the matrix being diagonalized must be symmetry.
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 n row-major matrix.
Σ is n by n diagonal matrix.
VT is n by m 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&*

Example: **egM1**

source: http://uni10.org/examples/egM1.cpp

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

int main(){
 // Spin 1/2 Heisenberg hamiltonian
 double elem[] = {1.0/4,      0,      0,     0,
                        0, -1.0/4,  1.0/2,     0,
                        0,  1.0/2, -1.0/4,     0,
                        0,      0,      0, 1.0/4};
 uni10::Matrix H(4, 4, elem);
 std::cout<<H;
 // Diagonlize H
 std::vector<uni10::Matrix> results = H.diagonalize();
 std::cout<<"The eigen values: \n\n"<<results[0];
 std::cout<<"The eigen vectors: \n\n"<<results[1];

 // Access element in a diagonal matrix
 uni10::Matrix D = results[0];
 std::cout<<"D.at(1, 1) = "<<D.at(1, 1)<<std::endl;;
 std::cout<<"D[2] = " << D[2]<<std::endl;
 // Assign element
 std::cout<<"\nAssign D.at(3, 3) = 7.0\n\n";
 D.at(3, 3) = 7.0;
 std::cout<<D;

 // Access element
 std::cout<<"H.at(1, 2) = "<<H.at(1, 2)<<std::endl;;
 std::cout<<"H[5] = " << H[5]<<std::endl;

 // Make a pure density matrix from ground state
 uni10::Matrix U = results[1];
 // Generate ground state by taking the first H.rol() elements from U.
 uni10::Matrix GS(1, H.col(), U.elem());
 // Transposed GS
 uni10::Matrix GST = GS;
 GST.transpose();

 std::cout<<"\nThe ground state: \n\n";
 std::cout<< GS;
 std::cout<< GST;

 // Compose a pure density matrix from ground state
 uni10::Matrix Rho = GST * GS;
 std::cout<<"\nPure density matrix of ground state: \n\n";
 std::cout<< Rho;

 // Measure ground state energy
 std::cout<<"\nThe ground state energy: " << (Rho * H).trace() << std::endl;
}
```

Output:

```
4 x 4 = 16

  0.250  0.000  0.000  0.000

  0.000 -0.250  0.500  0.000

  0.000  0.500 -0.250  0.000

  0.000  0.000  0.000  0.250

The eigen values:

4 x 4 = 4, Diagonal

 -0.750  0.000  0.000  0.000

  0.000  0.250  0.000  0.000

  0.000  0.000  0.250  0.000

  0.000  0.000  0.000  0.250

The eigen vectors:

4 x 4 = 16

  0.000  0.707 -0.707  0.000

  1.000  0.000  0.000  0.000

 -0.000  0.707  0.707  0.000

  0.000  0.000  0.000  1.000

D.at(1, 1) = 0.250
D[2] = 0.250

Assign D.at(3, 3) = 7.0

4 x 4 = 4, Diagonal

 -0.750  0.000  0.000  0.000

  0.000  0.250  0.000  0.000

  0.000  0.000  0.250  0.000

  0.000  0.000  0.000  7.000

H.at(1, 2) = 0.500
H[5] = -0.250

The ground state:

1 x 4 = 4
```

```
  0.000  0.707 -0.707  0.000

4 x 1 = 4

  0.000

  0.707

 -0.707

  0.000


Pure density matrix of ground state:

4 x 4 = 16

  0.000  0.000  0.000  0.000

  0.000  0.500 -0.500  0.000

  0.000 -0.500  0.500  0.000

  0.000  0.000  0.000  0.000


The ground state energy: -0.750
```

Example: **egM2**

source: http://uni10.org/examples/egM2.cpp

```
1) #include <iostream>
2) #include <uni10.hpp>
3)
4) int main(){
5)   uni10::Matrix M(4, 5);
6)   M.randomize();
7)   std::cout<<M;
8)   // carry out SVD
9)   std::vector<uni10::Matrix> rets = M.svd();
10)
11)  // write matrice out to file
12)  rets[0].save("mat_U");
13)  rets[1].save("mat_Sigma");
14)  rets[2].save("mat_VT");
15)
16)  uni10::Matrix U(rets[0].row(), rets[0].col(), rets[0].isDiag());
17)  uni10::Matrix S(rets[1].row(), rets[1].col(), rets[1].isDiag());
18)  uni10::Matrix VT(rets[2].row(), rets[2].col(), rets[2].isDiag());
19)
```

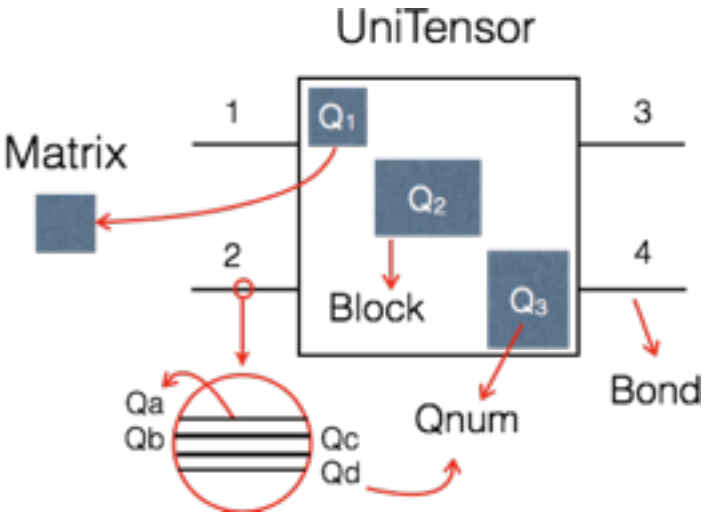
```
20) // read in the matrice we just write out
21) U.load("mat_U");
22) S.load("mat_Sigma");
23) VT.load("mat_VT");
24) std::cout<< S;
25) std::cout<< U * S * VT;
26)}
```

Output:

```
4 x 5 = 20

  0.840  0.394  0.783  0.798  0.912

  0.198  0.335  0.768  0.278  0.554

  0.477  0.629  0.365  0.513  0.952

  0.916  0.636  0.717  0.142  0.607

4 x 4 = 4, Diagonal

  2.736  0.000  0.000  0.000

  0.000  0.555  0.000  0.000

  0.000  0.000  0.449  0.000

  0.000  0.000  0.000  0.382

4 x 5 = 20

  0.840  0.394  0.783  0.798  0.912

  0.198  0.335  0.768  0.278  0.554

  0.477  0.629  0.365  0.513  0.952

  0.916  0.636  0.717  0.142  0.607
```

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 ten sor 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);

Assign labels

Assign the labels *newLabels* to each bond of the *UniTensor*, replacing the origin labels on the bonds.

Parameters

newLabels: int*
int pointer to the array of labels.
newLabels: std::vector<int>
vector of the array of labels.

uni10::UniTensor::**label**

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

Access labels

Returns the label of the bond at the position *idx*, the order of bonds are decided at initialization. If no input given, return the array of labels of type std::<int>vector.

Parameters

idx: int
The position of the bond from which the label is retrieved.

Return Value

The value of label (2) or the array of labels (1).

uni10::UniTensor::**bondNum**

size_t bondNum()const;

Access the number of bonds

Returns the number of bonds.

Return Value

The value of the number of bonds of the *UniTensor*.

uni10::UniTensor::**inBondNum**

int inBondNum()const;

Access the number of in-coming bonds

Returns the number of in-coming bonds.

Return Value

The value of the number of in-coming bonds of the *UniTensor*.

uni10::UniTensor::**bond**

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

Access bonds

Returns the bond of type *Bond* at the position *idx*, the order of bonds is decided at initialization. If no input given, return the array of bonds of type std::<*Bond*>vector.

Parameters

idx: int

The position of the *bond* being retrieved.

Return Value

The bond in type *Bond* (2) or the array of bonds (1).

uni10::UniTensor::**addRawElem**

void addRawElem(double* rawElem);

Assign elements

Assigns elements to the UniTensor from the given “raw elements” rawElem. Raw elements mean non-block-diagonal elements which are written down with a chosen basis. This function will reorganize the raw elements into block-diagonal elements.

Parameters

rawElem: double*

double pointer pointing to the array of input non-block-diagonal elements.

uni10::UniTensor::**blockNum**

size_t blockNum()const;

Access the number of blocks

Returns the number of blocks of elements

Return Value

The value of the number of blocks.

uni10::UniTensor::**blockQnum**

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

Access Qnums of blocks

Returns the quantum number of type *Qnum* of the block at the position *idx*, the order of blocks is in ascending order of *Qnum*. If no input given, returns an array of *Qnum* of all blocks. The returned array of *Qnums* is in ascending order of *Qnum*.

Parameters

idx: int

The position of the block from which the *Qnum* is retrieved.

Return Value

The quantum number in type *Qnum* (2) or the array of *Qnums* in type std::vector<*Qnum*>(1).

uni10::UniTensor::**getBlocks**

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

Access block elements

Returns the element blocks of various *Qnum*s as std::map<*Qnum*, *Matrix*>. The returned std::map is a mapping from Qnum to a corresponding elements as a *Matrix* of the *UniTensor*. The order for iterating through all the *Qnum*’s blocks is defined by the smaller than sign “<” of *Qnum*.

Return Value

A std::map of *Qnum* and *Matrix*, they are the quantum number and the block elements of the *UniTensor*.

uni10::UniTensor::**getBlock**

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

Get specific block elements

Returns the block elements of specific quantum number *qnum* as a *Matrix*. If the *diag* flag is set, only the diagonal elements of the block will be picked out to a diagonal *Matrix*.

Parameters

qnum: Qnum

The quantum number of the block to be retrieved.

diag: bool, optional(false)

A flag to specify whether the block elements are diagonal.

Return Value

Matrix of the block elements of the quantum number *qnum*.

uni10::UniTensor::**putBlock**

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

Assign elements to specific block

Assigns elements of the given matrix *mat* to the specific block of quantum number *qnum*, replacing the origin elements. If the given matrix *mat* is diagonal, it sets all the elements to zero and assigns elements of *mat* to the diagonal elements of the block.

Parameters

qnum: Qnum

The quantum number of the block which is being assigned elements to.

mat: Matrix

The matrix elements to be assigned to.

uni10::UniTensor::**elemNum**

size_t elemNum()const;

Access the number of elements.

Returns the number of total elements of the blocks.

Return Value

The value of the number of elements.

uni10::UniTensor::**at**

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

Access element.

Returns the element at position specified by the indices *idxs*. The given vector *idxs* is of size equal to the number of bonds, specifying the indices at each bond.

Parameters

```
idxs: std::vector<int>
```

An array of indices of the element in the *UniTensor*.

Return Value

The element at indices *idxs*.

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

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

Access element

Returns the element at position *idx*. The position *idx* is counted by lining up all the elements in a sequence. The first bond's dimension is the most significant dimension.

Parameters

```
idx: int
```

The position of the element in the *UniTensor*.

Return Value

The element at indices *idx*.

uni10::UniTensor::**rawElem**

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

Access non-block-diagonal elements

Returns the non-block-diagonal elements(raw elements) as a *Matrix*. The row(or column) bases of the elements are defined by the in-coming bonds(or out-going) bonds.

Return Value

The *Matrix* of the raw elements.

uni10::UniTensor::**printRawElem**

void printRawElem()const;

Print out raw elements

Prints out raw elements as(for example):

```
         2,0    1,0    0,0    1,0    0,0   -1,0    0,0   -1,0   -2,0
     ------------------------------------------------------------------
    |
 2,0|  0.142  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000
    |
 1,0|  0.000  0.952  0.000  0.916  0.000  0.000  0.000  0.000  0.000
    |
 0,0|  0.000  0.000  0.198  0.000  0.335  0.000  0.768  0.000  0.000
    |
 1,0|  0.000  0.636  0.000  0.717  0.000  0.000  0.000  0.000  0.000
    |
 0,0|  0.000  0.000  0.278  0.000  0.554  0.000  0.477  0.000  0.000
    |
-1,0|  0.000  0.000  0.000  0.000  0.000  0.394  0.000  0.783  0.000
    |
 0,0|  0.000  0.000  0.629  0.000  0.365  0.000  0.513  0.000  0.000
    |
-1,0|  0.000  0.000  0.000  0.000  0.000  0.798  0.000  0.912  0.000
    |
-2,0|  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.840
```

In the above example, the UniTensor has two in-coming bonds and two out-going bonds(see example egU1.cpp), with each bond having states with Qnums [q(1), q(0), q(-1)]. The resulting raw elements form a 9 by 9 Matrix. The first row shows the Qnums, U1 and parity, of the columns below and the first column shows the Qnums of the rows on the right.

uni10::UniTensor::**setName**

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

Assign name to the UniTensor

Assigns *name* to the UniTensor.

Parameters

name: std::string

The string of the *name.*

uni10::UniTensor::**getName**

std::string getName();

Access name of the UniTensor

Return the name of the *UniTensor*.

uni10::UniTensor::**save**

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

Write the UniTensor out to file

Writes the UniTensor out to a file of path *fname*. To load in a tensor, use the constructor of *UniTensor*.

Parameters

fname: std::string

The path of the file.

uni10::UniTensor::**similar**

bool similar(const UniTensor& Tb)const;

Test whether the UniTensor is similar to input tensor *Tb*

Two tensors are said to be similar if the bonds of the two tensors are exactly the same.

Parameters

Tb: UniTensor

The *UniTensor* to be compared with.

Return Value

true if the *UniTensor* is similar to *Tb*, *false* otherwise.

uni10::UniTensor::**elemCmp**

bool elemCmp(const UniTensor& Tb)const;

Test whether the elements of the UniTensor are the same as in *Tb*

Compares the elements of the two tensors one by one. Returns true the all elements are the same.

Parameters

uT: UniTensor

Return Value

true if the elements of the *UniTensor* is the same as in *Tb*, *false* otherwise.

uni10::UniTensor::**check**

void check();

Print out the memory usage of the existing UniTensor

Prints out the memory usage as(for example):

```
Existing Tensors: 30
Allocated Elem: 2240
Max Allocated Elem: 4295
Max Allocated Elem for a Tensor: 924
```

In the above example, currently there are 30 tensors and total number of existing elements is 2240. The maximum element number for now is 4295 and the maximum element number of a tensor is 924.

uni10::UniTensor::**permute**

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

Permute the order of bonds

Permutes the order of bonds to the order according to *newLabels*. *inBondNum* specifies the number of in-coming bonds after permuted. By doing so, all the elements are rearranged and the quantum numbers, shapes and even the number of the symmetry blocks may change. For example, default labels for a 4-bond *UniTensor* are [0, 1, 2, 3]. If we want to permute the order of the first two bonds, the *newLabels* are [1, 0, 2, 3].

Parameters

newLabels: int*
int pointer pointing to the array of *newLabels*.
newLabels: std::vector<int>
std::vector of the array *newLabels*
inBondNum: int
The number of in-coming bonds to be for the permuted *UniTensor*.

Return Value

The reference of the permuted *UniTensor*.

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

UniTensor& transpose();

Transpose the block elements

Transpose each block of various quantum numbers. The bonds are changed from in-coming to out-coming or vice versa without changing the quantum numbers on the bonds. That is, the bases of in-coming and out-going are exchanged without any change on the basis itself.

Return Value

The reference of the transposed *UniTensor*.

uni10::UniTensor::**combineBond**

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

Combine bonds together

Combines bonds of label in the array *combined_labels*. The resulting bond has the same label and *bondType* as the bond of the first label in array *combined_labels*.

Parameters

combined_labels: std::vector<int>

std::vector of the array labels of those bonds to be combined together.

Return Value

The reference of the *UniTensor* after combining bonds.

uni10::UniTensor::**partialTrace**

UniTensor& partialTrace(int la, int lb);

Trace out two bonds

Traces out the two bond of label *la* and *lb*.

Parameters

la, lb: int

The labels of the bonds to be traced out.

Return Value

The reference of the *UniTensor* after partial tracing two bonds.

uni10::UniTensor::**trace**

double trace()const;

Trace out in-coming bonds and out-going bonds

Traces out in-coming bonds and out-going bonds and returns the trace value.

Return Value

The trace value.

uni10::UniTensor::**set_zero**

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

Assign elements

Set all the elements to zero(1). If *qnum* is given, set all the elements of the block with quantum number equal to *qnum* to zero.

Parameters

qnum: Qnum

The quantum of the block to be set.

uni10::UniTensor::**eye**

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

Assign elements

Set all the block elements to identity(1). If *qnum* is given, set the block with quantum number equal to *qnum* to identity.

Parameters

qnum: Qnum

The quantum of the block to be set.

uni10::UniTensor::**randomize**

void randomize();

Assign elements

Randomly assigns values to the elements.

uni10::UniTensor::**orthoRand**

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

Assign elements

Randomly generates orthogonal bases. Assigns to the elements of every block or the block with quantum number equal to *qnum*.
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.*

Parameters

qnum: Qnum

The quantum of the block to be set.

uni10::UniTensor::**exSwap**

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

Record swaps

In fermionic system, the operation of Ta * Tb != Tb * Ta. It's not because of compatibility of *Ta* and *Tb*, but it is come from the swap signs of fermionic operators in *Ta* and *Tb*. This function is to record the swaps of the bonds in the *UniTensor* when exchanging the operation order with the *UniTensor Tb*. Returns the array for swaps of the bonds. For adding fermionic swap signs, see function *addGate*().

Parameters

Tb: UniTensor

The tensor to exchange with.

Return Value

The array of the swaps of the bonds.

uni10::UniTensor::**addGate**

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

Add swap gate

Adds swap gate when the order of two bonds are swapped in fermionic system. The meaning of adding swap gate is to multiply the corresponding elements by -1.

Parameters

swaps: std::vector<_Swap>

The swaps of the bonds.

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

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

Perform multiplication

(1) Performs tensor contraction with another *UniTensor Tb*. It contracts out the bonds of the same labels in the *UniTensor* and *Tb*
(2) Performs element-wise multiplication with a scalar *a* of type double.

Parameters

Tb: UniTensor

The given tensor to multiplied with.

a: double

The scalar of type *double* to multiplied with.

Return Value

The reference of the resulting *UniTensor*.

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

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

Perform additions of elements

Performs element by element addition. The tensor *Tb* to be added must be similar to the *UniTensor*. See *UniTensor::similar()*

Parameters

Tb: UniTensor

The given tensor to add.

Return Value

The reference of the resulting *UniTensor*.

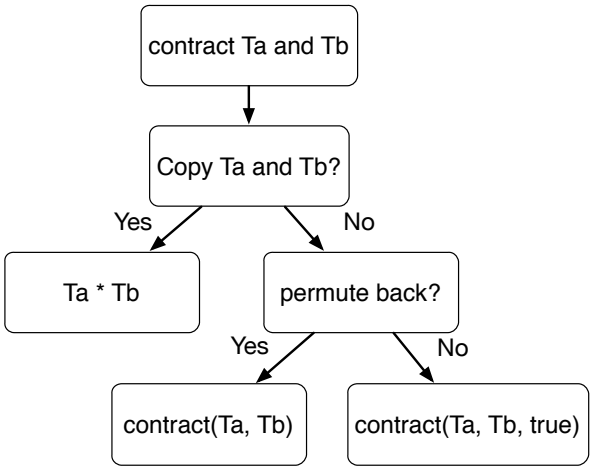
non-member overloads:

contract

friend UniTensor contract(UniTensor& Ta, UniTensor& Tb, bool fast = false);

Perform multiplication

Performs tensor contraction of Ta and Tb. It contracts out the bonds of the same labels in *Ta* and *Tb.* Note that compared with *operator** (*Ta* * *Tb*), It performs contraction without copying *Ta* and *Tb*. Thus it uses less memory. When the flag *fast* is set true, the two tensors *Ta* and *Tb* are contracted without being permuted back to origin labels.



Parameters

Ta, Tb: UniTensor

The tensors to carry out contractions.

fast: bool, optional(false)

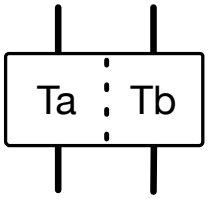
A flag to decide whether two tensors are permuted back to origin labels. If true, two tensor are not permuted back.

Return Value

Resulting *UniTensor*.

outer

friend UniTensor outer(const UniTensor& Ta, const UniTensor& Tb);



Perform multiplication

Performs outer product of Ta and Tb.

Ta, Tb: UniTensor

The tensors to carry out contractions.

Return Value

Resulting *UniTensor*.

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)

Perform multiplication

(1) Performs tensor contraction, Ta * Tb. It contracts out the bonds of the same labels in *Ta* and *Tb.* It copies *Ta* and *Tb* and then call function *contract* to perform the contraction.
(2)&(3) Performs element-wise multiplication with a scalar *a* of type double, a * Ta or Ta * a.

Parameters

Ta, Tb: UniTensor

The tensors to carry out contractions.

a: double

The scalar of type *double* to multiplied with.

Return Value

Resulting *UniTensor*.

operator+

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

Perform additions of elements

Performs element by element addition.

Parameters

Ta, Tb: UniTensor

The tensors to carry out contractions

Return Value

Resulting *UniTensor*.

operator<<

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

Print out UniTensor

Prints out a *UniTensor* as(for example):

std::cout << uT;

```
***************** Demo *****************
             ____________
            |            |
        0___|3          3|___2
            |            |
        1___|3          3|___3
            |            |
            |____________|

================BONDS===============
IN : (U1 = 1, P = 0, 0)|1, (U1 = 0, P = 0, 0)|1, (U1 = -1, P = 0, 0)|1, Dim = 3
IN : (U1 = 1, P = 0, 0)|1, (U1 = 0, P = 0, 0)|1, (U1 = -1, P = 0, 0)|1, Dim = 3
OUT: (U1 = 1, P = 0, 0)|1, (U1 = 0, P = 0, 0)|1, (U1 = -1, P = 0, 0)|1, Dim = 3
OUT: (U1 = 1, P = 0, 0)|1, (U1 = 0, P = 0, 0)|1, (U1 = -1, P = 0, 0)|1, Dim = 3

===============BLOCKS===============
--- (U1 = -2, P = 0, 0): 1 x 1 = 1

  0.840

--- (U1 = -1, P = 0, 0): 2 x 2 = 4

  0.394  0.783

  0.798  0.912

--- (U1 = 0, P = 0, 0): 3 x 3 = 9

  0.198  0.335  0.768

  0.278  0.554  0.477

  0.629  0.365  0.513

--- (U1 = 1, P = 0, 0): 2 x 2 = 4

  0.952  0.916

  0.636  0.717

--- (U1 = 2, P = 0, 0): 1 x 1 = 1

  0.142

Total elemNum: 19
***************** END *****************
```

In the above example, *uT* has four bonds with default labels [0, 1, 2, 3]. The bonds 0 and 1 are in-coming bonds. and the 2, 3 are out-going bonds. Each bond has dimension three and the three states corresponding to three *U1* quantum number [-1, 0, 1]. The following are the block elements of the tensor. There are five blocks of various *U1* [-2, -1, 0, 1, 2] and various sizes. The total element number is 19.

Parameters

os: std::ostream

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

uT: UniTensor

The *UniTensor* to be printed out.

Return Value

Returns *std::ostream&*

Example: **egU1**

source: http://uni10.org/examples/egU1.cpp

```
1) #include <iostream>
2) #include <uni10.hpp>
3)
4) int main(){
5)  // Construct spin 1 Heisenberg model
6)  // Raw element
7)  double heisenberg_s1[] = \
8)      {1, 0, 0, 0, 0, 0, 0, 0, 0,\
9)       0, 0, 0, 1, 0, 0, 0, 0, 0,\
10)      0, 0,-1, 0, 1, 0, 0, 0, 0,\
11)      0, 1, 0, 0, 0, 0, 0, 0, 0,\
12)      0, 0, 1, 0, 0, 0, 1, 0, 0,\
13)      0, 0, 0, 0, 0, 0, 0, 1, 0,\
14)      0, 0, 0, 0, 1, 0,-1, 0, 0,\
15)      0, 0, 0, 0, 0, 1, 0, 0, 0,\
16)      0, 0, 0, 0, 0, 0, 0, 0, 1\
17)     };
18) // without using symmetries of the hamiltonian.
19) uni10::Qnum q0(0);
20) // create array of three quantum number q0, without using symmetries
21) std::vector<uni10::Qnum> qnums;
22) qnums.push_back(q0);
23) qnums.push_back(q0);
24) qnums.push_back(q0);
25) // Create in-coming and out-going bonds, without any symmetry.
26) uni10::Bond bd_in(uni10::BD_IN, qnums);
27) uni10::Bond bd_out(uni10::BD_OUT, qnums);
28) std::vector<uni10::Bond> bonds;
29) bonds.push_back(bd_in);
30) bonds.push_back(bd_in);
31) bonds.push_back(bd_out);
```

```
32) bonds.push_back(bd_out);
33) // Create tensor from the bonds and name it "H".
34) uni10::UniTensor H(bonds, "H");
35) H.addRawElem(heisenberg_s1);
36) std::cout<< H;
37)
38)
39) // Since it has U1 symmetry(total Sz conserved)
40) // add U1 quantum number to the states of bonds.
41) uni10::Qnum q1(1);
42) uni10::Qnum q_1(-1);
43) qnums.clear();
44) qnums.push_back(q1);
45) qnums.push_back(q0);
46) qnums.push_back(q_1);
47)
48) // Create in-coming and out-going bonds
49) bd_in.assign(uni10::BD_IN, qnums);
50) bd_out.assign(uni10::BD_OUT, qnums);
51) bonds.clear();
52) bonds.push_back(bd_in);
53) bonds.push_back(bd_in);
54) bonds.push_back(bd_out);
55) bonds.push_back(bd_out);
56)
57) // Create tensor from the bonds and name it "H_U1".
58) uni10::UniTensor H_U1(bonds, "H_U1");
59) // Add raw elements to tensor
60) H_U1.addRawElem(heisenberg_s1);
61) std::cout<< H_U1;
62)
63) // Check the quantum number of the blocks
64) std::cout<<"The number of the blocks = "<< H_U1.blockNum()<<std::endl;
65) std::vector<uni10::Qnum> block_qnums = H_U1.blockQnum();
66) for(int q = 0; q < block_qnums.size(); q++)
67)     std::cout<< block_qnums[q]<<", ";
68) std::cout<<std::endl<<std::endl;
69) // print out non-block diagonal elements, raw elements
70) H_U1.printRawElem();
71)
72) // Write out tensor
73) H_U1.save("egU1_H_U1");
74) return 0;
75)}
```

Output:

```
****************** H ******************
            _____________
           |             |
       0___|3           3|___2
           |             |
       1___|3           3|___3
           |             |
           |_____________|
```

```
===============BONDS==============
IN : (U1 = 0, P = 0, 0)|3, Dim = 3
IN : (U1 = 0, P = 0, 0)|3, Dim = 3
OUT: (U1 = 0, P = 0, 0)|3, Dim = 3
OUT: (U1 = 0, P = 0, 0)|3, Dim = 3

===============BLOCKS===============
--- (U1 = 0, P = 0, 0): 9 x 9 = 81

  1.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000

  0.000  0.000  0.000  1.000  0.000  0.000  0.000  0.000  0.000

  0.000  0.000 -1.000  0.000  1.000  0.000  0.000  0.000  0.000

  0.000  1.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000

  0.000  0.000  1.000  0.000  0.000  0.000  1.000  0.000  0.000

  0.000  0.000  0.000  0.000  0.000  0.000  0.000  1.000  0.000

  0.000  0.000  0.000  0.000  1.000  0.000 -1.000  0.000  0.000

  0.000  0.000  0.000  0.000  0.000  1.000  0.000  0.000  0.000

  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  1.000

Total elemNum: 81
****************** END ******************


****************** H_U1 ******************
          ____________
         |            |
       0___|3          3|___2
         |            |
       1___|3          3|___3
         |            |
         |____________|

===============BONDS==============
IN : (U1 = 1, P = 0, 0)|1, (U1 = 0, P = 0, 0)|1, (U1 = -1, P = 0, 0)|1, Dim = 3
IN : (U1 = 1, P = 0, 0)|1, (U1 = 0, P = 0, 0)|1, (U1 = -1, P = 0, 0)|1, Dim = 3
OUT: (U1 = 1, P = 0, 0)|1, (U1 = 0, P = 0, 0)|1, (U1 = -1, P = 0, 0)|1, Dim = 3
OUT: (U1 = 1, P = 0, 0)|1, (U1 = 0, P = 0, 0)|1, (U1 = -1, P = 0, 0)|1, Dim = 3

===============BLOCKS===============
--- (U1 = -2, P = 0, 0): 1 x 1 = 1

  1.000

--- (U1 = -1, P = 0, 0): 2 x 2 = 4

  0.000  1.000
```

```
  1.000  0.000

--- (U1 = 0, P = 0, 0): 3 x 3 = 9

 -1.000  1.000  0.000

  1.000  0.000  1.000

  0.000  1.000 -1.000

--- (U1 = 1, P = 0, 0): 2 x 2 = 4

  0.000  1.000

  1.000  0.000

--- (U1 = 2, P = 0, 0): 1 x 1 = 1

  1.000

Total elemNum: 19
***************** END ****************

The number of the blocks = 5
(U1 = -2, P = 0, 0), (U1 = -1, P = 0, 0), (U1 = 0, P = 0, 0), (U1 = 1, P = 0, 0),
(U1 = 2, P = 0, 0),
```

Example: **egU2**

source: http://uni10.org/examples/egU2.cpp

```
1) #include <iostream>
2) #include <uni10.hpp>
3)
4) int main(){
5)   // Construct spin 1 Heisenberg model by reading in the tensor which is
      written out in example egU1
6)   uni10::UniTensor H_U1("egU1_H_U1");
7)
8)   // Get the block of quantum number q0 as a matrix "block0"
9)   uni10::Qnum q0(0);
10)  uni10::Matrix block0 = H_U1.getBlock(q0);
11)  std::cout<<block0;
12)  // Randomly assign "block0" and put it back to H_U1
13)  block0.randomize();
14)  H_U1.putBlock(q0, block0);
15)  //std::cout<<H_U1;
16)
17)  // Permute bonds by its label, the default label of it is [0 1 2 3]
18)  int permuted_label[] = {1, 2, 3, 0};
19)  // Permute bonds to which with label [1, 2, 3, 0] and leaving 1 bond as in-
      coming bonds.
```

```
20) H_U1.permute(permuted_label, 1);
21) //std::cout<<H_U1;
22)
23) // Permute bonds by its label.
24) std::vector<int> combined_label;
25) combined_label.push_back(2);
26) combined_label.push_back(3);  // combined_label = [2, 3]
27) // combine the two bonds with label 2 and 3
28) H_U1.combineBond(combined_label);
29) std::cout<< H_U1;
30)
31) return 0;
32)}
```

Output:

```
3 x 3 = 9

 -1.000  1.000  0.000

  1.000  0.000  1.000

  0.000  1.000 -1.000


************* egU1_H_U1 *************
             ____________
            |            |
        1___|3          9|___2
            |            |
            |           3|___0
            |            |
            |____________|

================BONDS===============
IN : (U1 = 1, P = 0, 0)|1, (U1 = 0, P = 0, 0)|1, (U1 = -1, P = 0, 0)|1, Dim = 3
OUT: (U1 = 2, P = 0, 0)|1, (U1 = 1, P = 0, 0)|1, (U1 = 0, P = 0, 0)|1, (U1 = 1, P
= 0, 0)|1, (U1 = 0, P = 0, 0)|1, (U1 = -1, P = 0, 0)|1, (U1 = 0, P = 0, 0)|1, (U1
= -1, P = 0, 0)|1, (U1 = -2, P = 0, 0)|1, Dim = 9
OUT: (U1 = -1, P = 0, 0)|1, (U1 = 0, P = 0, 0)|1, (U1 = 1, P = 0, 0)|1, Dim = 3

================BLOCKS===============
--- (U1 = -1, P = 0, 0): 1 x 6 = 6

  0.840  0.394  0.000  0.783  1.000  1.000

--- (U1 = 0, P = 0, 0): 1 x 7 = 7

  0.000  0.798  1.000  0.912  1.000  0.198  0.000

--- (U1 = 1, P = 0, 0): 1 x 6 = 6

  1.000  1.000  0.335  0.000  0.768  0.278

Total elemNum: 19
```

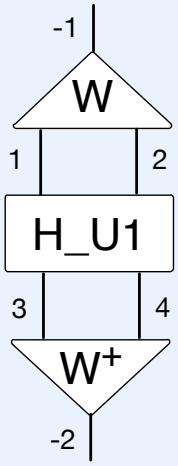
```
****************** END ****************
```

Example: **egU3**

source: http://uni10.org/examples/egU3.cpp

```
1) #include <iostream>
2) #include <uni10.hpp>
3)
4) int main(){
5)   // Construct spin 1 Heisenberg model by reading in the tensor which is
      written out in example egU1
6)   uni10::UniTensor H_U1("egU1_H_U1");
7)
8)   // Randomly create an isometry tensor
9)   uni10::Qnum q0(0);
10)  uni10::Qnum q1(1);
11)  uni10::Qnum q_1(-1);
12)  uni10::Qnum q2(2);
13)  uni10::Qnum q_2(-2);
14)  std::vector<uni10::Qnum> in_qnums;
15)  in_qnums.push_back(q2);
16)  in_qnums.push_back(q1);
17)  in_qnums.push_back(q0);
18)  in_qnums.push_back(q0);
19)  in_qnums.push_back(q_1);
20)  in_qnums.push_back(q_2);
21)  std::vector<uni10::Qnum> out_qnums;
22)  out_qnums.push_back(q1);
23)  out_qnums.push_back(q0);
24)  out_qnums.push_back(q_1);
25)  uni10::Bond bd_in(uni10::BD_IN, in_qnums);
26)  uni10::Bond bd_out(uni10::BD_OUT, out_qnums);
27)  std::vector<uni10::Bond> bonds;
28)  bonds.push_back(bd_in);
29)  bonds.push_back(bd_out);
30)  bonds.push_back(bd_out);
31)  // Create isometry tensor W and transposed WT
```

```
32) uni10::UniTensor W(bonds, "W");
33) W.orthoRand();
34) uni10::UniTensor WT = W;
35) WT.transpose();
36)
37) // Operate W and WT on H_U1, see the contraction labels in the documentation.
38) int label_H[] = {1, 2, 3, 4};
39) int label_W[] = {-1, 1, 2};
40) int label_WT[] = {3, 4, -2};
41) H_U1.addLabel(label_H);
42) W.addLabel(label_W);
43) WT.addLabel(label_WT);
44) //std::cout<<W;
45) std::cout<<W * H_U1 * WT;
46)
47) // Write the tensors W and WT out to file
48) W.save("egU3_W");
49) WT.save("egU3_WT");
50)
51) // Check the memory usage.
52) uni10::UniTensor::check();
53)
54) return 0;
55)}
```



Output:

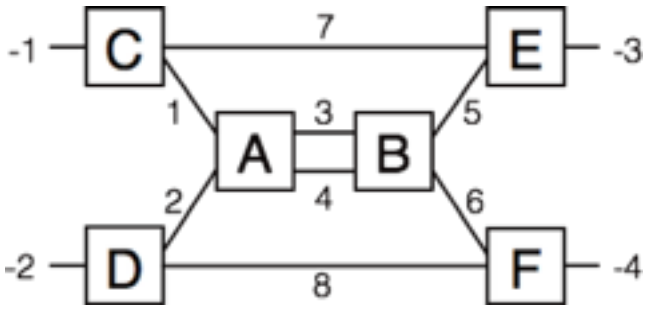
```
************************************
             ____________
            |            |
       -1___|6          6|___-2
            |            |
            |____________|

================BONDS===============
IN : (U1 = 2, P = 0, 0)|1, (U1 = 1, P = 0, 0)|1, (U1 = 0, P = 0, 0)|2, (U1 = -1,
P = 0, 0)|1, (U1 = -2, P = 0, 0)|1, Dim = 6
OUT: (U1 = 2, P = 0, 0)|1, (U1 = 1, P = 0, 0)|1, (U1 = 0, P = 0, 0)|2, (U1 = -1,
P = 0, 0)|1, (U1 = -2, P = 0, 0)|1, Dim = 6

================BLOCKS===============
--- (U1 = -2, P = 0, 0): 1 x 1 = 1

  1.000

--- (U1 = -1, P = 0, 0): 1 x 1 = 1

  0.803

--- (U1 = 0, P = 0, 0): 2 x 2 = 4

 -0.264 -0.427

 -0.427 -1.164

--- (U1 = 1, P = 0, 0): 1 x 1 = 1
```

```
  0.989

--- (U1 = 2, P = 0, 0): 1 x 1 = 1

  1.000

Total elemNum: 8
****************** END ****************

Existing Tensors: 3
Allocated Elem: 43
Max Allocated Elem: 98
Max Allocated Elem for a Tensor: 19
```

Class uni10::**Network**

Tensor Network

Class *Network* is made for contractions of a whole tensor network. Left figure is an example of tensor network. Each blocks are tensors(in this library *UniTensor*) and tensors are connect together by the bonds(lines) of the same labels. Network is comprised by the connections which is specified by the labels. To construct a network, you have to prepare a file on the left, specifying the labels of the tensors and the out-coming labels of the tensor "TOUT: ". Note that "TOUT" line is necessary in the file. Even if the out-coming tensor having no bond, that is a scalar, you still need to keep the line as "TOUT: ", without any label. Labels are separated by blank or comma and in-coming and out-going labels are divided by semi-colon. The last line, starting from "ORDER: " is optional. It suggests the order to construct the pair-wise contraction order and furthermore you can force the contraction order by the parentheses as(for example):




```
A: 1 2; 3 4
B: 3 4; 5 6
C: -1; 7 1
D: -2; 2 8
E: 7 5; -3
F: 6 8 -4
TOUT: -1 -2; -3 -4
ORDER: A B C E D F
```

```
ORDER: ((((A B) C) E) (D F))
```

To use a network, one can use the method *putTensor*() to put tensor to the specific place in the network. After placing all the tensors to the right places. Call the function *launch*() to contract out all the bonds in between. When the function *launch*() will generate a contraction order or follow the contraction order if given, carry out the pair-wise contraction. Note that, in fermionic system, even though the labels of contractions are specified, *A * B* != *B * A*. It is because the swap signs that come from the swapping of the fermionic operators in *A* and *B*. The function launch guarantees that no matter what contraction order of the tensors, the resulting tensor is the same as the tensors are contracted by the order when the labels are given in the first few lines. In our example, launch() guarantees the result *A * B * C * D * E * F*.

member functions

uni10::Network::**Network**

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

Construct UniTensor

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

(1) Constructs with the file *fname*. The file specifies the connections between tensors in the network. See more in the introduction of *Network* above.

(2) Constructs with the file *fname* as (1) and also put those *UniTensor* into the right position in the network. The given array of tensor pointers is in the same order of those tensors specified in the network file *fname.*

Parameters

fname: std::string

Path of the network file.

uTptrs: std::vector<UniTensor*>

Array of *UniTensor* pointers to put into the network.

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

~Network();

Destruct Network

Destroys the Network and freeing all the intermediate tensors.

uni10::Network::**putTensor**

(1) void putTensor(int idx, const UniTensor* uTptr, bool force=false);
(2) void putTensor(std::string tname, const UniTensor* uTptr, bool force=false);

Assign tensor to the Network

(1) Assigns the tensor uT to the position *idx* in the network. The order of the position is given in the network file. For the example on the right side, to put tensor uT to the network at position "C", use *putTensor*(2, *uTptr*).
(2) Assigns the tensor uT to the position of the tensor *tname* in the network file. To put tensor uT to the network at position "C", use *putTensor*("C", *uTptr*).

```
A: 1 2; 3 4
B: 3 4; 5 6
C: -1; 7 1
D: -2; 2 8
E: 7 5; -3
F: 6 8 -4
TOUT: -1 -2; -3 -4
ORDER: A B C E D F
```

If the force flag is set, the tensor will be put in the network without reconstructing the pair-wise contraction sequence.

Parameters

idx: int

The position in the network at which the tensor uT is placed.

tname: std::string

The name of the tensor in the network file fname. It specifies the position where the tensor *uT* is put to the network.

uTptr: UniTensor*

The pointer of the tensor to be added in the network.

force: bool, optional(false)

If false, the contraction sequence will be reconstructed after putting the tensor. If true, contraction sequence remains the same, only the tensor at position *idx* is replaced with *uT*.

uni10::Network::**launch**

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

Contract the tensors in the Network

Performs contractions of the tensors in the network, returns the out-coming *UniTensor* and naming it as *name*.

Parameters

name: std::string, optional("")

The name for the out-coming *UniTensor*.

non-member overloads

operator<<

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

Print out Network

Before the calling the function *net.launch*()
Prints out the *Network* coming from the example network file on the right:
std::cout << net;

```
W1: i[-1] o[0, 1, 3]
W2: i[-2] o[7, 10, 11]
U: i[3, 7] o[4, 8]
Ob: i[1, 4] o[2, 5]
UT: i[5, 8] o[6, 9]
W1T: i[0, 2, 6] o[-3]
W2T: i[9, 10, 11] o[-4]
Rho: i[-3, -4] o[-1, -2]
TOUT:
```

```
W1: -1; 0 1 3
W2: -2; 7 10 11
U: 3 7; 4 8
Ob: 1 4; 2 5
UT: 5 8; 6 9
W1T: 0 2 6; -3
W2T: 9 10 11; -4
Rho: -3 -4; -1 -2
TOUT:
ORDER: W1 W1T W2 W2T U Ob UT Rho
```

The above is the connections of every tensors from the network file. “i” means labels for in-coming bonds and “o” for out-going.
After we call the function *net.launch*(), besides printing out the connections between tensors above, the contraction sequence, or binary tree of the pair-wise contractions is also printed as:

```
*(1):
|   *(70): 7, 9, -4, -2,
|   |   *(70): -1, 7, 9, -3,
|   |   |   *(70): -1, 0, 7, 2, 6, 9,
|   |   |   |   W1(20): -1, 0, 1, 3,
|   |   |   |   *(20): 3, 7, 1, 2, 6, 9,
|   |   |   |   |   *(20): 3, 7, 8, 1, 2, 5,
|   |   |   |   |   |   U(6): 3, 7, 4, 8,
|   |   |   |   |   |   Ob(6): 1, 4, 2, 5,
|   |   |   |   |   UT(6): 5, 8, 6, 9,
|   |   |   W1T(20): 0, 2, 6, -3,
|   |   Rho(924): -3, -4, -1, -2,
|   *(70): -2, 7, 9, -4,
|   |   W2(20): -2, 7, 10, 11,
|   |   W2T(20): 9, 10, 11, -4,
```

The output shows clearly how the whole network is contracted. For example, *U* and *Ob* are contracted as the intermediate tensor with labels [3, 7, 8, 1, 2, 5] and element number 20. And the resulting tensor then contracts with *UT* and so on.

Parameters

os: std::ostream

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

net: Network

The *Network* to be printed out.

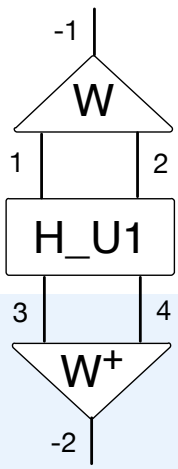
Return Value

Returns *std::ostream&*

Example: **egN1**

network file: http://uni10.org/examples/egN1_network

```
H: 1 2; 3 4
W: -1; 1 2
WT: 3 4; -2
TOUT: -1; -2
ORDER: ((W H) WT)
```



source: http://uni10.org/examples/egN1.cpp

```
56)#include <iostream>
57)#include <uni10.hpp>
58)
59)int main(){
60) // Read in the tensor H_U1 which is written out in example egU1
     and W, WT in example egU3
61) uni10::UniTensor H_U1("egU1_H_U1");
62) uni10::UniTensor W("egU3_W");
63) uni10::UniTensor WT("egU3_WT");
64)
```

```
65) // Create network by reading in network file "egN1_network"
66) uni10::Network net("egN1_network");
67) // Put tensors to the Network net
68) net.putTensor("H", &H_U1);
69) net.putTensor("W", &W);
70) net.putTensor("WT", &WT);
71)
72) // Perform contractions inside the tensor network
73) std::cout<<net.launch();
74) // Print out the network
75) std::cout<<net;
76)
77) return 0;
78)}
```

Output:

```
************************************
             ____________
            |            |
       -1___|6          6|___-2
            |            |
            |____________|

===============BONDS==============
IN : (U1 = 2, P = 0, 0)|1, (U1 = 1, P = 0, 0)|1, (U1 = 0, P = 0, 0)|2, (U1 = -1,
P = 0, 0)|1, (U1 = -2, P = 0, 0)|1, Dim = 6
OUT: (U1 = 2, P = 0, 0)|1, (U1 = 1, P = 0, 0)|1, (U1 = 0, P = 0, 0)|2, (U1 = -1,
P = 0, 0)|1, (U1 = -2, P = 0, 0)|1, Dim = 6

===============BLOCKS===============
--- (U1 = -2, P = 0, 0): 1 x 1 = 1

  1.000

--- (U1 = -1, P = 0, 0): 1 x 1 = 1

  0.803

--- (U1 = 0, P = 0, 0): 2 x 2 = 4

 -0.264 -0.427

 -0.427 -1.164

--- (U1 = 1, P = 0, 0): 1 x 1 = 1

  0.989

--- (U1 = 2, P = 0, 0): 1 x 1 = 1

  1.000

Total elemNum: 8
***************** END ****************
```

```
H: i[1, 2] o[3, 4]
W: i[-1] o[1, 2]
WT: i[3, 4] o[-2]
TOUT: i[-1] o[-2]

*(8): -1, -2,
|   *(12): -1, 3, 4,
|   |   W(12): -1, 1, 2,
|   |   H(19): 1, 2, 3, 4,
|   WT(12): 3, 4, -2,
```