Uni10: an open-source library for tensor network algorithms

Ying-Jer Kao[1,2], **Yun-Da Hsieh**[1], **Pochung Chen**[3]

[1] Department of Physics , National Taiwan University, Taipei, 10607, Taiwan
[2] Center for Advanced Study in Theoretical Science, National Taiwan University, Taipei, 10607, Taiwan
[3] Department of Physics, National Tsing Hua University, Hsin-Chu, 30013, Taiwan

E-mail: `yjkao@phys.ntu.edu.tw`

Abstract. We present an object-oriented open-source library for developing tensor network algorithms written in C++ called Uni10. With Uni10, users can build a symmetric tensor from a collection of bonds, while the bonds are constructed from a list of quantum numbers associated with different quantum states. It is easy to label and permute the indices of the tensors and access a block associated with a particular quantum number. Furthermore a network class is used to describe arbitrary tensor network structure and to perform network contractions efficiently. We give an overview of the basic structure of the library and the hierarchy of the classes. We present examples of the construction of a spin-1 Heisenberg Hamiltonian and the implementation of the tensor renormalization group algorithm to illustrate the basic usage of the library. The library described here is particularly well suited to explore and fast prototype novel tensor network algorithms and to implement highly efficient codes for existing algorithms.

1. Introduction

Tensor network (TN) algorithms[1] have emerged in recent years as a promising tool to study the physics of quantum many-body systems. The most well-known example is the density matrix renormalization group (DMRG) algorithm for quasi-one-dimensional systems[2], which variationally optimizes the matrix product state (MPS)[3]. DMRG has been successfully used to study frustrated quantum spin systems[4], and topological ordered systems[5]. Borrowing the concepts of entanglement from the quantum information science, proposals to extend MPS to higher dimensions have also been put forward. Two-dimensional generalizations such as projective entangled-pair states (PEPS)[6, 7] and the multi-scale entanglement renormalization ansatz (MERA) [8, 9], have been used to study many-body physics in two dimensions. Recently, ideas from TNs have also been used to study systems other than the condensed matter systems such as the holographic principles of AdS/CFT, and loop quantum gravity[1]. In short, TN algorithms have become an important tool to study non-perturbative aspects of many strongly interacting systems. From a broader perspective, tensors are widely used in different branches of sciences and engineering, such as pattern and image recognition, signal processing, machine learning and computational neuroscience to represent very complicated multidimensional data with multiple aspects [10]. It is then nature to use TNs to represent these multidimensional data and to use tensor decomposition algorithms to find the low-dimension and low-rank tensors that best approximate the complicated data.

To motive the need for a tensor network library, consider the contraction of a 2D tensor network. Currently, there exist many different schemes, such as the corner transfer matrix method (CTM)[11, 12], the tensor renormalization group (TRG)[13], plaquette renormalization[14], high-order tensor

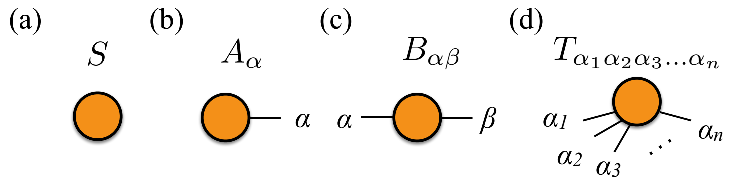


Figure 1: (Color online.) Tensor network diagrams: (a) scalar, (b) vector, (c) matrix and (d) rank-n tensor

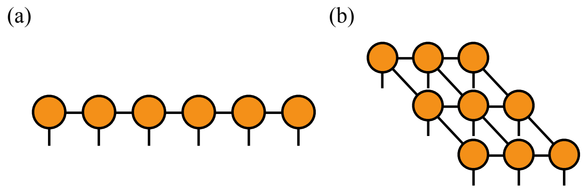


Figure 2: (Color online.) Two examples of tensor network diagrams: (a) Matrix Product State (MPS) for 6 sites with open boundary conditions; (b) Projected Entangled Pair State (PEPS) for a 3×3 lattice with open boundary conditions.

renormalization group (HOTRG)[15], and many others. All these algorithms involves contracting tensors, i.e., summing over common indices between tensors. However, programming tensor network algorithms is tedious and prone to errors. The task of keeping track of tensor indices while performing contraction of a complicated tensor network can be daunting. It is hence desirable to have a platform that provides the bookkeeping and optimization to speed up the software development .

In this paper, we describe a fully open-source implementation of a library designed for the development of the tensor network algorithms called the *Universal Tensor Network library* or **Uni10**. This software distinguishes itself from other available software solutions by providing the following advantages:

- Fully implemented in C++.
- Aimed toward applications in tensor network algorithms.
- Provides basic tensor operations with an easy-to-use interface.
- Provides a class `Network` to process and store the graphical representations of the networks.
- Provides an engine to construct and analyze the contraction tree for a given network.
- Provides a collection of Python wrappers called **pyUni10** which interact with the compiled C++ library to take advantage of the Python language for better code readability and faster prototyping, without the sacrifice of speed.
- Provides behind-the-scene optimization and acceleration.

The paper is organized as follows: Sec. 2 gives a brief introduction to the tensor networks and their graphical representation; Sec. 3 summarizes the key components of the Uni10 library and their usage; Sec. 4 gives a sample code implementing the TRG method using pyUni10; we conclude in Sec. 5.

2. Tensors, tensor networks, and tensor network diagrams

Tensors can be regarded as the multi-dimensional generalizations of matrices. For a given tensor $\mathbf{T}$, the rank is the number of indices. Therefore, a rank-0 tensor is a scalar (S) , a rank-1 tensor is a vector (A_α), and a rank-2 tensor is a matrix ($B_{\alpha\beta}$). The number of all possible values for an index is the dimension of the index. We can perform index contraction for a set of tensors by summing over all of the possible

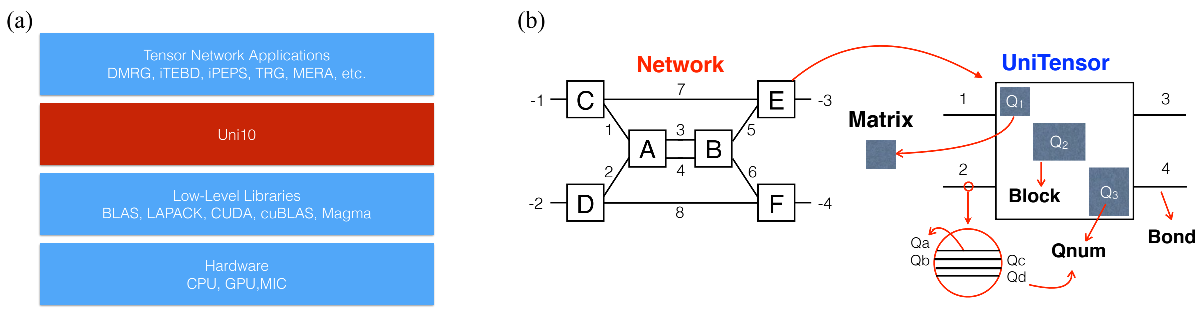


Figure 3: (a) Uni10 serves as a middleware between the tensor network applications and the low-level numerical libraries. (b) Hierarchy of the uni10 classes.

values for the common indices. For example, a matrix multiplication can be regarded the contraction of two rank-2 tensors

$$T_{\alpha\beta} = \sum_{\gamma=1}^{\chi} A_{\alpha\gamma} B_{\gamma\beta}, \tag{1}$$

where χ is the dimension of the index γ. A more complicated example is to contract four tensors to give a complex number,

$$G = \sum_{\alpha\beta\gamma\delta\xi\eta\zeta} A_{\alpha\beta\gamma\delta} B_{\beta\gamma\delta\xi} C_{\xi\eta\zeta\delta} D_{\eta\zeta\alpha}. \tag{2}$$

It is convenient to introduce a graphical representation of tensors and TNs using the tensor diagrams (Fig. 1). Tensors are represented by a circle or a rectangle, and indices are represented by lines emerging from these shapes. A TN is thus represented by a set of shapes interconnected by lines. The internal lines connecting tensors correspond to contracted indices, and external lines correspond to open indices in the TN. Figure 2 shows the graphical representation for MPS and PEPS respectively.

3. Uni10

3.1. Overview

The key object in Uni10 is the Abelian symmetric tensor class `UniTensor` together with auxiliary classes for quantum numbers `Qnum`, blocks `Block`, and bonds `Bond`. In addition, the library provides a network class `Network` to store and process the details of the graphical representations of the networks. Table 1 lists the major classes in Uni10, and Fig 3(b) shows the hierarchy of the different classes.

For a given network, Uni10 provides an engine construct and analyze the pair-wise contraction tree. A heuristic algorithm is implemented to search for an optimal pairwise contraction order based on the computation and memory constraints of the system. The library also provides a collection of wrappers for Python called **pyUni10** to take advantage of the for Python language that is designed for better code readability, and faster prototyping.

In the following, we will use the pyUni10 module to demonstrate some features of Uni10. The following `import` statement should be inserted at the beginning of all the Python codes,

```
from pyUni10 import *
```

3.2. Symmetric Tensors

In order to define a symmetric tensor, one must first construct an object that carries a quantum number. In Uni10, a quantum number object is created through the `Qnum` class. To create a `Qnum` object named `q10` to represent a U(1) quantum number $U_1 = 1$ with even parity one can use the code:

```
q10 = Qnum(1, PRT_EVEN)
```

Table 1: Major Uni10 classes

Class	Description
`UniTensor`	A symmetric tensor
`Bond`	A bond with a list of quantum numbers $\{q_1, q_2, \ldots\}$
`Qnum`	A composite $U(1) \times Z_2 \times Z_2^F$ quantum number
`Matrix`	A matrix
`Block`	A map from a quantum number to a matrix
`Network`	A network

While to create a quantum number object `q_11` with $U_1 = -1$ with odd parity one can use the code:

```
q_11 = Qnum(-1, PRT_ODD)
```

Using the quantum number objects, one can proceed to define a bond which contains these quantum numbers. In Uni10, two types of bonds are defined: the *incoming* and *outgoing* bonds. This can be indicated using `BD_IN` and `BD_OUT` constants. To define an incoming bond of a three-fold degenerate quantum number 0 the following code can be used:

```
Q=Qnum(0)
Qlist=[Q,Q,Q]
Bi=Bond(BD_IN,Qlist)
```

In cases where none of the symmetries is used, Uni10 provides a simple interface to create a non-symmetric bond. To create a bond with dimension four the following code can be used:

```
Bo=Bond(BD_OUT,4)
```

Where a default quantum number is assigned to all the quantum states. When no symmetries are used, there is no need to distinguish between the incoming and outgoing bonds. However, the incoming bonds are always used as the column indices for the internal storage of Uni10, and the specification is preserved in the API.

A tensor is created using these bonds as shown in the code below:

```
A=UniTensor([Bi,Bi,Bo,Bo])
```

In Uni10, each bond is associated with an integer label. A default set of labels will be assigned during the tensor creation but the user can set the labels in any way that is convenient. In Uni10, the tensor operations are overloaded, for example, the expression in Eq. (2) is represented in Uni10 as

```
G= A*B*C*D
```

Here bonds with the same label will be summed. A more efficient way to contract tensors is using the `Network` class, which will be described in details in Sec. 3.4.

3.3. Example: Heisenberg Hamiltonian

Here we demonstrate how to construct a two-site Heisenberg Hamiltonian for $S = 1$,

$$H = \mathbf{S}_1 \cdot \mathbf{S}_2. \tag{3}$$

First we create incoming and outgoing bonds with dimension 3,

```
bdi = uni10.Bond(uni10.BD_IN, 3 )
bdo = uni10.Bond(uni10.BD_OUT, 3 )
```

Next we create a tensor named `H` with two incoming and two outgoing bonds of dimension 3 that will be used to store the Hamiltonian of the Heisenberg model,

```
H = UniTensor([bdi, bdi, bdo, bdo])
```

We can fill in the elements of `H` by putting in the elements directly,

```
heisenberg_s1=[\
1, 0, 0, 0, 0, 0, 0, 0, 0,\
0, 0, 0, 1, 0, 0, 0, 0, 0,\
0, 0,-1, 0, 1, 0, 0, 0, 0,\
0, 1, 0, 0, 0, 0, 0, 0, 0,\
0, 0, 1, 0, 0, 0, 1, 0, 0,\
0, 0, 0, 0, 0, 0, 0, 1, 0,\
0, 0, 0, 0, 1, 0,-1, 0, 0,\
0, 0, 0, 0, 0, 1, 0, 0, 0,\
0, 0, 0, 0, 0, 0, 0, 0, 1\
]
H.setRawElem(heisenberg_s1)
```

Next, we show how to build the same Hamiltonian in the U(1) symmetric basis. This can be achieved following the steps described above by specifying the quantum numbers on the bonds. First, create `Qnum` objects holding U(1) quantum numbers 0, 1, and -1,

```
q0 = uni10.Qnum(0)
q1 = uni10.Qnum(1)
q_1 = uni10.Qnum(-1)
```

Then create incoming and outgoing bonds carrying these quantum numbers,

```
bdi = uni10.Bond(uni10.BD_IN, [q1, q0, q_1] )
bdo = uni10.Bond(uni10.BD_OUT, [q1, q0, q_1] )
```

Finally, create a tensor `H_U1` from the incoming and outgoing bonds, and fill in the elements,

```
H_U1 = uni10.UniTensor([bdi, bdi, bdo, bdo], "H_U1")
H_U1.setRawElem(heisenberg_s1)
```

The `setRawElem` method will fill in Hamiltonian tensor according to the local basis on the bonds, but the tensor is stored in the symmetric basis, i.e., in terms of blocks. Uni10 provides a compact way to output the contents of the tensor with the **`print`** command,

```
print H_U1
```

The output includes the type, the quantum numbers and their degeneracies of each bond, the blocks corresponding to the total quantum numbers formed by the individual bonds, and the elements in each block,

```
**************** 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) 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
```

3.4. Network

TNs are collections of tensors and it is important to store the information about the connection between tensors. An important feature of Uni10 is the `Network` class. Figure 5(a) shows a typical TN with three tensors W, WT, and H. The labels indicates how these tensors are connected. Two tensors connected through an internal line will share a common label. This TN can be represented by a text file with entries showing the tensors and the labels for each tensors [Fig. 5(b)]. Uni10 provides a simple interface to create a network named `net` by reading in the network file,

```
net = uni10.Network("network.net")
```

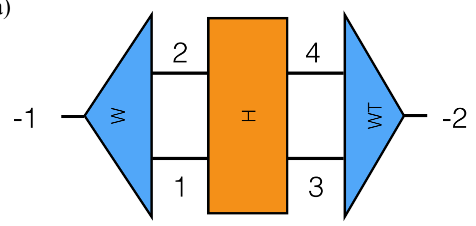
Put tensors to `net`,

```
net.putTensor("H", H_U1)
net.putTensor("W", W)
net.putTensor("WT", WT)
```

Finally, perform contractions inside the tensor network,

```
T=net.launch()
```

the resulting tensor `T` has an incoming bond labeled -1, and an outgoing bond labeled -2, as indicated in the `network.net`.

(a)


(b)

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

Figure 4: (Color online.) (a) A network with three tensors. (b) The network file `network.net` representing the connection between the tensors.

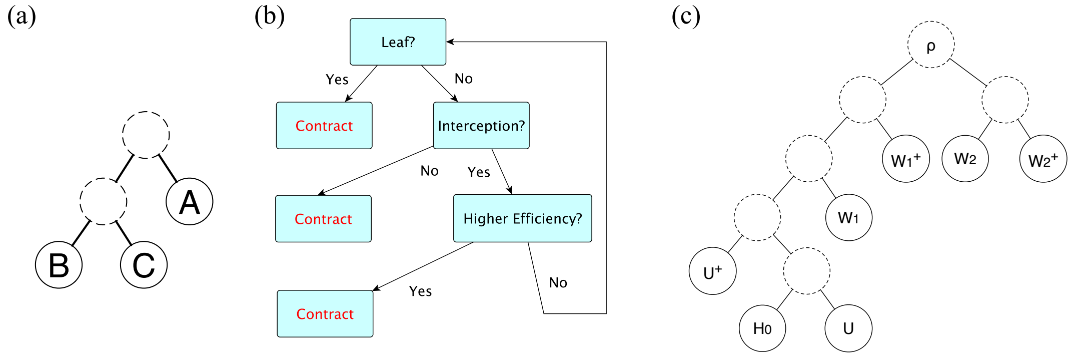


Figure 5: (Color online.) (a) Pairwise contraction of three tensors. (b) Contraction algorithm. (c) Pairwise contraction tree.

3.5. Network Contraction Algorithm

Finding the optimal pairwise contraction order of a general TN is NP-hard[16], and search algorithms are available to find the optimal order in some special cases[17]. Here, we describe a network contraction algorithm which aims to find a locally optimized pair-wise contractions order to contract the whole tensor network. The algorithm has two major features: First, it is fast, and the time complexity is of order $O(\log N)$ to $O(N)$, depending on the balance of the contraction tree. Second, the algorithm uses heuristics to optimize the contraction order and users can provide initial guess to improve the optimization process.

We measure the efficiency of a pairwise contraction of two tensors A and B by defining the contraction efficiency ϵ in terms of the numbers of tensor elements $N(A)$, $N(B)$ and $N(A*B)$,

$$\epsilon(A,B) \equiv (N(A)+N(B))/N(A*B). \tag{4}$$

It is clear that $\epsilon(A,B)=\epsilon(B,A)$. For a given pairwise contraction of A and B, we insert a third tensor C by using a local optimization algorithm to determine the most efficient way for the three tensor to contract. We compare $\epsilon(A,B)$, $\epsilon(B,C)$ and $\epsilon(C,A)$ and contract first the pair with the largest efficiency, followed by the contraction of the resulting tensor with the remaining tensor [Fig. 5(a)]. This procedure is repeated until all the tensors are inserted into the network [Fig. 5(b)]. For a given tensor network, a binary tree for the contraction order will be generated [Fig. 5(c)]. Since the algorithm can only find the local optimal contraction order, the final contraction order may not be optimal. To overcome this, users can propose a contraction order in the network file to serve as the initial guess for the search algorithm.

4. Sample Code: TRG

Here we present a code sample to demonstrate how to compute the magnetization along the z-axis $\langle\sigma_z\rangle$ using TRG for the transverse Ising model. In the following example, the tensors `GaL` and `GbL` of the ground state wave function are obtained through some update scheme, such as iTEBD. The detailed description of the TRG algorithm can be found in Ref. [13]. The goal of the algorithm is to compute the expectation value of an operator $\hat{O}$,

$$\langle O\rangle = \frac{\langle\Psi|\hat{O}|\Psi\rangle}{\langle\Psi|\Psi\rangle} \tag{5}$$

for a given TNS wave function in the two sublattice form,

$$|\Psi\rangle = \sum_{s_i,s_j} \mathrm{Tr} \prod_{i\in A, j\in B} \Gamma^A_{u_i d_i l_i r_i}[s_i]\Gamma^B_{u_j d_j l_i r_j}[s_j]|s_i s_j\rangle. \tag{6}$$

To compute the norm (denominator) and numerator of Eq. (5), we first define two functions `makeT`, and `makeImpurity` to form double tensors and impurity tensors[13] from Γ^A and Γ^B,

```
def makeT(GL):
    GLT = copy.copy(GL)
    GLT.transpose()
    GL.setLabel([0, 1, 2, 3, 4])
    GLT.setLabel([-1, -2, -3, -4, 0])
    T = GLT * GL
    for i in xrange(1, 5):
        T.combineBond([-i, i])
    return T

def makeImpurity(GL, Ob):
    GLT = copy.copy(GL)
    GLT.transpose()
    Ob.setLabel([0, 5])
    GL.setLabel([5, 1, 2, 3, 4])
    GLT.setLabel([-1, -2, -3, -4, 0])
    I = Ob * GL
    I = GLT * I
    for i in xrange(1, 5):
        I.combineBond([-i, i])
    return I
```

The function `trgSVD` performs the singular value decomposition for the A and B sublattices, and perform truncation,

```
def trgSVD(wch, T, chi):
    if wch % 2 == 0:
        T.permute([-4, -3, -1, -2], 2)
    else:
        T.permute([-1, -4, -2, -3], 2)

    svd = T.getBlock().svd()
    chi = chi if chi < svd[1].row() else svd[1].row()
    bdi_chi = uni10.Bond(uni10.BD_IN, chi)
    bdo_chi = uni10.Bond(uni10.BD_OUT, chi)
    S0 = uni10.UniTensor([T.bond(0), T.bond(1), bdo_chi])
    S1 = uni10.UniTensor([bdi_chi, T.bond(2), T.bond(3)])
    svd[1].resize(chi, chi)
    for i in xrange(chi):
        svd[1][i] = np.sqrt(svd[1][i])
    S0.putBlock(svd[0].resize(svd[0].row(), chi) * svd[1])
    S1.putBlock(svd[1] * svd[2].resize(chi, svd[2].col()))
    return S0, S1
```

The function `trgContract` performs the contraction of four tensors given the TN described in `TRG_net`,

```
def trgContract(Ss, TRG_net):
    for i in xrange(4):
        TRG_net.putTensor(i, Ss[i])
    return TRG_net.launch()
```

The function `updateAll` performs the renormalization of the tensors,

```
def updateAll(Ts, Imps, chi, TRG_net):
    S = []
    I = []
    for r in xrange(2):
        S.append(trgSVD(r, Ts[r%len(Ts)], chi))
    for r in xrange(4):
        I.append(trgSVD(r, Imps[r], chi))
    a, b, c, d = 0, 1, 2, 3
    Ts = [trgContract([S[a][0], S[b][0], S[a][1], S[b][1]], TRG_net)]
    maxel = max([Ts[0][i] for i in xrange(Ts[0].elemNum())])
    Ts[0] *= (1.0 / maxel)

    Imps=[trgContract([S[a][0],I[b][0],I[a][1],S[b][1]],TRG_net)*(1.0/maxel),\
            trgContract([S[a][0],S[b][0],I[c][1],I[b][1]],TRG_net)*(1.0/maxel),\
            trgContract([I[c][0],S[b][0],S[a][1],I[d][1]],TRG_net)*(1.0/maxel),\
            trgContract([I[a][0],I[d][0],S[a][1],S[b][1]],TRG_net)*(1.0/maxel)]

    return Ts, Imps
```

The function `trgExpectation` compute the expectation value using the impurity tensors,

```
def trgExpectation(Ts, exp_net):
    for step in xrange(4):
        exp_net.putTensor(step, Ts[step % len(Ts)])
    return exp_net.launch()[0]
```

The main function reads in the tensor networks (Fig. 6),

```
TRG_net = uni10.Network("TRG.net")
exp_net = uni10.Network("expectation.net")
```

Construct the σ_z operator,

```
bdi = uni10.Bond(uni10.BD_IN, 2)
bdo = uni10.Bond(uni10.BD_OUT, 2)
Mz = uni10.UniTensor([bdi, bdo], "Mz")
Mz.setElem([1, 0, 0, -1])
```

Prepare the double tensors and impurity tensors,

```
Ts = [makeT(GaL), makeT(GbL)]
Imps = [makeT(GaL), makeT(GbL), makeImpurity(GaL, Mz), makeT(GbL)]
```

Finally, perform the renormalization steps,

```
for i in xrange(8):
    Ts, Imps = updateAll(Ts, Imps, chi2, TRG_net)
M=trgExpectation(Imps, exp_net)
norm=trgExpectation(Ts, exp_net)
```

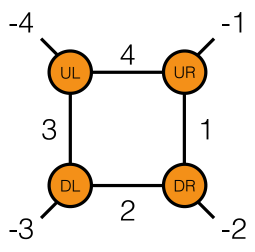
For brevity, the code segments associated with initialization is omitted. The complete pyUni10 source code for the transverse Ising model can be found at the Uni10 website [18].

5. Conclusions

We have presented a new, open-source C++ library for the tensor network algorithms. This library is suitable for implementing a wide range of TN algorithms. In this work we have described the basic structure of the library, and how to construct a spin-1 Heisenberg Hamiltonian. For more in-depth documentation, and more elaborate examples using pyUni10 API listed in Table 1, the reader should visit the Uni10 website at `http://www.uni10.org`. Further optimization of Uni10 using speed accelerators such as Graphic Processing Unit (GPU) and Many Integrated Core (MIC) architectures and Matlab wrappers will be available in future releases.

Acknowledgments
We thank Tama Ma and Sukhbinder Singh for their help in the development of Uni10. We acknowledge inspiring discussions with A. Sandvik, G. Evenbly, G. Vidal, and Ian McCulloch. The support

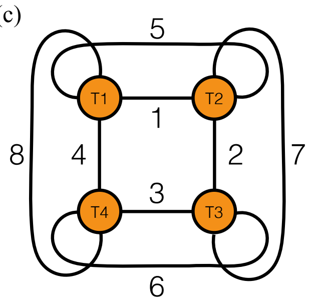
(a)



(b)

```
UR: 4, 1; -1
DR: 1, 2; -2
DL: -3; 3, 2
UL: -4; 4, 3
TOUT: -1, -2, -3, -4
```

(c)



(d)

```
T1: 8, 1, 4, 5
T2: 7, 5, 2, 1
T3: 2, 6, 7, 3
T4: 4, 3, 8, 6
TOUT:
```

Figure 6: (Color online.) (a) A TN for tensor renormalization in the TRG algorithm. (b) Network file `TRG.net`. (c) A TN for expectation values. (d) Network file `expectation.net`.

from MOST in Taiwan through Grants No. 100-2112-M-002-013-MY3, 100-2923-M-004-002 -MY3, 102-2112-M-002-003-MY3, 101-2112-M-007-010-MY3 as well as the support from National Taiwan University Grant No. 101R891004 are acknowledged.

Appendix

Here we list the commands available in pyUni10 (v0.9.1). Full documentation, including API and other examples, is available at Uni10 Website [18].

Table 1: pyUni10 global constants, functions, classes and class methods.

Constants	
`BD_IN`	Defines an incoming bond
`BD_OUT`	Defines an outcoming bond
`PRT_EVEN`	Defines particle parity even in a bosonic system
`PRT_ODD`	Defines particle parity odd in a bosonic system
`PRTF_EVEN`	Defines particle parity even in a fermionic system
`PRTF_ODD`	Defines particle parity odd in a fermioninc system
Functions	
`contract()`	Performs tensor contraction
`otimes()`	Performs tensor product
`combine()`	Combines bonds
`takeExp()`	Returns the exponential of a matrix
`Qnum` class	
`Qnum()`	Creates a quantum number
`QnumF()`	Creates a fermionic quantum number
`Qnum.U1()`	Obtains the U(1) quantum number
`Qnum.assign()`	Assigns a quantum number to `Qnum`
`Qnum.assignF()`	Assigns a fermonic quantum number to `Qnum`
`Qnum.prt()`	Obtains the particle number parity
`Qnum.prtF()`	Obtains the fermionic number parity
`Bond` class	
`Bond()`	Creates a bond
`Bond.Qlist()`	Returns the `Qnum` objects in a `Bond`
`Bond.assign()`	Assigns `Qnum` to `Bond`
`Bond.change()`	Changes the type of `Bond`

`Bond.combine()`	Combine bonds
`Bond.degeneracy()`	Returns the degeneracy of quantum number
`Bond.dim()`	Returns total dimension of `Bond`
`Bond.type()`	Returns type of `Bond`
`Matrix` class	
`Matrix()`	Creates a matrix
`Matrix.col()`	Returns number of columns of `Matrix`
`Matrix.eigh()`	Returns the eigenvalues and eigenvectors
`Matrix.elemNum()`	Returns the total number of elements
`Matrix.getElem()`	Returns the reference to matrix elements
`Matrix.identity()`	Sets `Matrix` to 1 on the diagonal and 0 otherwise
`Matrix.isDiag()`	Returns whether `Matrix` is diagonal
`Matrix.load()`	Loads elements of `Matrix` from a file
`Matrix.norm()`	Returns the L^2 norm of `Matrix`
`Matrix.orthoRand()`	Generates an orthogonal basis with random elements and assigns to `Matrix`
`Matrix.randomize()`	Sets the elements of `Matrix` with random number in $[0, 1)$
`Matrix.resize()`	Resizes `Matrix`
`Matrix.row()`	Returns number of rows of `Matrix`
`Matrix.save()`	Saves `Matrix` to a file
`Matrix.setElem()`	Sets the elements of `Matrix`
`Matrix.set_zero()`	Sets all the elements in `Matrix` to zero
`Matrix.sum()`	Returns the sum of all elements
`Matrix.svd()`	Performs singular value decomposition of `Matrix`
`Matrix.trace()`	Returns the trace of `Matrix`
`Matrix.transpose()`	Returns the transpose of `Matrix`
`UniTensor` class	
`UniTensor()`	Creates a symmetric tensor
`UniTensor.assign()`	Restructures `UniTensor` with new bonds
`UniTensor.blockNum()`	Returns the total number of blocks
`UniTensor.blockQnum()`	Returns the quantum number of a block in `UniTensor`
`UniTensor.bond()`	Returns a bond in `UniTensor`
`UniTensor.bondNum()`	Returns the total number of bonds
`UniTensor.combineBond()`	Combines bonds
`UniTensor.elemCmp()`	Compares the elements of two tensors
`UniTensor.elemNum()`	Returns the total number of elements
`UniTensor.getBlock()`	Returns the block of a given quantum number
`UniTensor.getBlocks()`	Returns the map from quantum numbers to blocks
`UniTensor.getElem()`	Returns the reference of the elements
`UniTensor.getName()`	Returns the name of `UniTensor`
`UniTensor.getRawElem()`	Returns the raw elements of `UniTensor` with row(column) basis defined by the incoming (outgoing) bonds
`UniTensor.identity()`	Sets blocks to identity matrix.
`UniTensor.inBondNum()`	Returns the number of incoming bonds in `UniTensor`
`UniTensor.label()`	Returns label(s) of `UniTensor`
`UniTensor.orthoRand()`	Randomly generates orthogonal bases and assigns to blocks
`UniTensor.partialTrace()`	Takes partial trace
`UniTensor.permute()`	Permutes the order of bonds
`UniTensor.printRawElem()`	Prints raw elements of `UniTensor`
`UniTensor.putBlock()`	Puts matrix into block
`UniTensor.randomize()`	Sets the elements of `UnTensor` with random number in $[0, 1)$
`UniTensor.save()`	Saves `UniTensor` to file
`UniTensor.setElem()`	Assigns the elements to `UniTensor`, replacing the originals
`UniTensor.setLabel()`	Assigns a new label of a bond
`UniTensor.setName()`	Sets the name of `UniTensor`
`UniTensor.setRawElem()`	Assigns raw elements to `UniTensor`
`UniTensor.set_zero()`	Sets all elements to zero
`UniTensor.similar()`	Checks if the bonds of two tensors are the same
`UniTensor.trace()`	Takes the trace

`UniTensor.transpose()`	Transposes all blocks associated with quantum numbers
`Network` class	
`Network()`	Creates a network
`Network.launch()`	Performs the full contraction of `Network`
`Network.profile()`	Returns the total memory usage of `Network`
`Network.putTensor()`	Put tensors into `Network`
`Network.putTensorT()`	Put transposed tensors into into `Network`

References

[1] Orús R 2014 *The European Physical Journal B* **87** 280–18
[2] White S R 1993 *Phys. Rev. B* **48**(14) 10345–10356
[3] Schollwöck U 2011 *Annals of Physics* **326** 96–192
[4] Depenbrock S, McCulloch I P and Schollwöck U 2012 *Phys. Rev. Lett.* **109** 067201
[5] Zaletel M P, Mong R S K and Pollmann F 2013 *Phys. Rev. Lett.* **110** 236801
[6] Verstraete F, Murg V and Cirac J 2008 *Advances in Physics* **57** 143–224
[7] Verstraete F and Cirac J I 2004 Renormalization algorithms for quantum-many body systems in two and higher dimensions (*Preprint* `cond-mat/0407066`)
[8] Vidal G 2007 *Phys. Rev. Lett.* **99** 220405
[9] Evenbly G and Vidal G 2009 *Phys. Rev. B* **79**(14) 144108
[10] Cichocki A 2014 Era of Big Data Processing: A New Approach via Tensor Networks and Tensor Decompositions (*Preprint* `arXiv:1403.2048`)
[11] Jordan J, Orus R, Vidal G, Verstraete F and Cirac J I 2008 *Phys. Rev. Lett.* **101** 250602
[12] Orús R and Vidal G 2009 *Physical Review B* **80** 094403
[13] Jiang H C, Weng Z Y and Xiang T 2008 *Phys. Rev. Lett.* **101** 090603
[14] Wang L, Kao Y J and Sandvik A W 2011 *Physical Review E* **83** 056703
[15] Xie Z Y, Chen J, Qin M P, Zhu J W, Yang L P and Xiang T 2012 *Physical Review B* **86** 045139
[16] Lam C C, P S and Rephael W 1997 *Parallel Processing Letters* **07** 157–168
[17] Pfeifer R N C, Haegeman J and Verstraete F 2014 *Physical Review E* **90** 033315
[18] URL `http://www.uni10.org`