Computer Physics Communications 228 (2018) 163–177

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

TNSPackage: A Fortran2003 library designed for tensor network state methods☆

Shao-Jun Dong, Wen-Yuan Liu, Chao Wang, Yongjian Han *, G.-C. Guo, Lixin He **

CAS Key Laboratory of Quantum Information, University of Science and Technology of China, Hefei, Anhui, 230026, China
Synergetic Innovation Center of Quantum Information and Quantum Physics, University of Science and Technology of China, Hefei, Anhui, 230026, China

ARTICLE INFO

Article history:
Received 13 June 2017
Received in revised form 27 February 2018
Accepted 3 March 2018
Available online 14 March 2018

Keywords:
Tensor network state
Condensed matter physics

ABSTRACT

Recently, the tensor network states (TNS) methods have proven to be very powerful tools to investigate the strongly correlated many-particle physics in one and two dimensions. The implementation of TNS methods depends heavily on the operations of tensors, including contraction, permutation, reshaping tensors, SVD and so on. Unfortunately, the most popular computer languages for scientific computation, such as Fortran and C/C++ do not have a standard library for such operations, and therefore make the coding of TNS very tedious. We develop a Fortran2003 package that includes all kinds of basic tensor operations designed for TNS. It is user-friendly and flexible for different forms of TNS, and therefore greatly simplifies the coding work for the TNS methods.

Program summary
Program Title: TNSP
Program Files doi: http://dx.doi.org/10.17632/fgggdbrdnx.1
Licensing provisions: GNU General Public License version 3
Programming language: Fortran2003
External routines: BLAS, LAPACK, ARPACK
Nature of problem: The implementation of Tensor Network State (TNS) methods depends heavily on the operations of tensors. Unfortunately, the most popular computer languages for scientific computation, such as Fortran and C/C++ do not have a standard library for such operations, and therefore make the coding of TNS very tedious.
Solution method: We develop a Fortran2003 package that includes all kinds of basic tensor operations designed for TNS, which greatly simplifies the coding work for the TNS methods.
Additional comments including Restrictions and Unusual features: A gcc-4.8.4 or later version is required to compile the code.

1. Introduction

One of the biggest challenges in modern condensed matter physics is to develop efficient numerical methods to solve the strongly correlated many-particle physics. As for strongly interacting systems, where conventional perturbation theory fails, numerical simulation plays a crucial role to reveal the nature of quantum many-body physics. Several popular methods, including the exact diagonalization (ED), Quantum Monte Carlo(QMC) methods [1,2] and the density matrix renormalization group (DMRG) method [3,4] , have been widely used and achieve great success. However, there are some limitations for the previous methods: ED methods suffer the "Exponential Wall" problem and the QMC suffers the notorious sign problem when simulating frustrated systems and fermion systems [5], whereas DMRG is limited to 1D or quasi-1D systems and does not work well for higher dimension systems [6]. It is pressing to develop new efficient numerical algorithms.

Recently, tensor network states (TNS), including matrix product states (MPS) [7], projected entangled pair states (PEPS) [8] etc., are proposed to describe many-body physics inspired by the quantum entanglement theory. The TNS methods provide a promising scheme to

☆ This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

* Corresponding author.
** Corresponding author at: CAS Key Laboratory of Quantum Information, University of Science and Technology of China, Hefei, Anhui, 230026, China.
E-mail addresses: smhan@ustc.edu.cn (Y. Han), helx@ustc.edu.cn (L. He).

https://doi.org/10.1016/j.cpc.2018.03.006
0010-4655/

investigate the systems that are not tractable by the previous methods. In the case of 1D, matrix product states (MPS) which constitute the variational space of DMRG, has been established as the leading method for the simulation of the statics and dynamics of one-dimensional strongly correlated quantum systems [9,10], both at zero [11] and finite temperatures [12]. As a natural extension of MPS to higher dimensions, PEPS show great potential to solve some long-standing problems [13–16]. Besides, TNS supply great flexibility for different problems, such as projected entangled simplex states (PESS) [17] and multi-scale entanglement renormalization ansatz (MERA) [18] which have been successfully applied to simulate frustrated magnets [19,20].

In the TNS scheme, there are many differences for programming codes when adopting different TNS forms including MPS, PEPS, PESS, MERA, string-bond states (SBS) [21] and so on. Therefore unlike the first-principles packages based on density functional theory, it is impossible to develop a universal code that includes all these methods. However, the TNS methods do have very similar features, where they all depend heavily on some basic operations of high rank tensors, even though with different TNS forms. Unfortunately, so far there is no standard library for such tensor operations in the most popular computer languages for scientific computations, e.g., Fortran and C/C++. Directly using the high dimensional arrays, makes the codes tedious, fragile, low efficient and hard to maintain.

To solve this problem, we develop a Fortran2003 package that integrates some most used tensor operations, including multiplication, contraction, permutation, singular value decomposition (SVD) and other matrix decomposition operations, etc. The package is very flexible, supporting all kinds of tensor formats, and data types, including integer, single or double precision real number, single or double precision complex number, logical and character. Some high performance mathematic packages, including Arpack [22], Lapack [23] and Blas [23] have been adopted to speed up the performance in the package. We extensively use advanced language features of Fortran2003, such as an object-oriented programming style, to improve the readability and re-usability of the package. We have successfully implemented the SBS method [24,25] and PEPS methods [16] for different many-particle Hamiltonian and geometries using the package, which greatly reduces the workload for the coding.

The rest of the paper is organized as follows. We first introduce the basic theory of TNS in Section 2 and their applications in many-body physics in Section 3. We summarize some most often used basic operations in TNS methods in Section 4, and then introduce the main features of the TNSpackage in Section 5. In Section 6, we give an example of how to implement a TNS algorithm (here PEPS) using the TNSpackage. We conclude in Section 7.

2. Basic theory of tensor network states

In this section, we introduce the basic theories of TNS. We start with the one-dimension matrix product states (MPS) [7,14].

2.1. Matrix product states (MPS)

Consider a 1D lattice with N sites and a physical degree freedom s on each site. A general quantum state of this system can be expanded as

$$|\Psi\rangle = \sum_{\{s\}} c_{s_1,s_2,\ldots,s_N} |s_1, s_2, \ldots, s_N\rangle \,, \tag{1}$$

where $c_{s_1,s_2,\ldots,s_N}$ is the coefficient for the basis $|s_1, s_2, \ldots, s_N\rangle$, and $\sum_{\{s\}} |c_{s_1,s_2,\ldots,s_N}|^2 = 1$. Generally, the total number of coefficients is d^N which increases exponentially with the size of the system, where d is the physical dimension of the system. To reduce the parameters of the many-body quantum state, the matrix product state (MPS) is introduced in which coefficients $c_{s_1,s_2,\ldots,s_N}$ are expressed as traces of a set of matrices [6,26], i.e.,

$$|\Psi\rangle = \sum_{\{s\}} \mathrm{tr}(A[1]^{s_1} A[2]^{s_2} \cdots A[N]^{s_N}) |s_1, s_2, \ldots, s_N\rangle \,, \tag{2}$$

where $A[i]$ is a tensor corresponding to the site i, and the index s_i is called the physical index of the tensor $A[i]$ since it corresponds to the physical degree freedom on site i. For each basis $|s_1, s_2, \ldots, s_N\rangle$, the physical index of $A[i]$ is fixed as s_i and the resulted tensor, $A[i]^{s_i}$, will be a matrix besides the endpoints of the 1D lattice (the tensor form of the endpoints depends on the boundary condition: for the open boundary, it will be a vector for fixed physical index; for the periodic boundary condition, it is a matrix). For the matrix $A[i]^{s_i}$, it has the matrix indexes l_i and r_i which are called virtual indices of the $A[i]$ since they are not physical and will be contracted in the state. This state is therefore named as matrix product state. The dimension of the virtual indices, denoted by D, is the controlling parameters of the state. The MPS with proper virtual D has been used as a successful variational space in DMRG method [6]. In addition, the ground state of a 1D gapped system and the slightly entangled state can be well approximated by an MPS [27], the precision of the approximation can be well controlled by the virtual dimension D: the larger D, the better approximation.

It is very convenient to use the graphical notation [26] to denote MPS instead of the formula like Eq. (2). In the graphical notation, a tensor is represented by a geometrical shape such as a circle or a diamond, and its indices are represented by outgoing legs. The legs corresponding to physical indices are called physical legs, and the legs corresponding to virtual indices are called virtual legs. As shown in Fig. 1, the upper one represents a single tensor $A[i]$. It has 3 indices, represented by 3 legs, with physical leg pointing upward and virtual legs pointing leftward and rightward. The bottom picture represents an MPS with the periodic condition. The connected legs are called bonds, which represent the virtual indices that are summed over.

From the quantum information point of view, more importantly, with deeper understanding of the relation between area law and the DMRG method [28], the MPS can be interpreted in another way which can be very conveniently extended to higher dimensions:

$$|\Psi\rangle = \sum_{\{s\},\{r\},\{l\}} A[1]^{s_1}_{r_1} |s_1\rangle\langle r_1| \, A[2]^{s_2}_{l_2 r_2} |s_2\rangle\langle l_2 r_2| \cdots A[N]^{s_N}_{l_N} |s_N\rangle\langle l_N| \, EPR_{12} \cdots EPR_{N-1,N} \,, \tag{3}$$

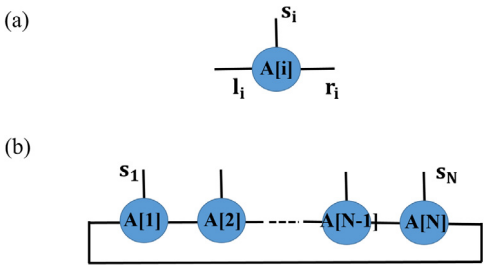

Fig. 1. (a) The graphical representation of tensor $A[i]^{s_i}_{l_i,r_i}$ at ith site. Every leg denotes a tensor index. (b) The graphical representation of the MPS corresponding to Eq. (2). The connected legs are virtual bonds that are summed over.

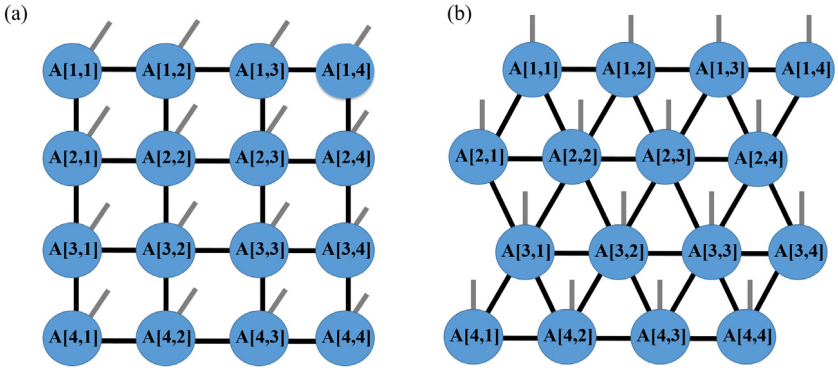

Fig. 2. (a) A PEPS on a 4×4 square lattice, where the tensors have maximum 4 virtual legs. (b) A PEPS on a 4×4 triangular lattice, where the tensor have 6 virtual legs. Each circle corresponds to a site in the lattice. The free legs are the physical indices.

where $|l_i\rangle$ and $|r_i\rangle$ are states in a D dimensional virtual Hilbert space, and

$$EPR_{i,i+1} = \frac{1}{\sqrt{D}} \sum_{j=1}^{D} |r_i = j, l_{i+1} = j\rangle \tag{4}$$

are the maximal entangled state on the bond between site i and site $i+1$. The configuration of the EPR pairs automatically satisfies the area law and can be viewed as the matrix of the MPS state in the virtual space. For each site, there is a projector, $A[i]^{s_i}_{l_i r_i} |s_i\rangle\langle l_i r_i|$, which project the state from the virtual space into the physical space. Any set of projectors will define a quantum many-body state. The projectors then define a variational space. The quantum information theory [29] tells us that the entanglement cannot be increased under the projections, that is, the resulting state (MPS) also satisfies the area law. With this understanding of the MPS, it is natural to extend the MPS to the higher dimension.

2.2. Projected entangled pair state (PEPS)

We now generalize MPS to higher dimension, namely the projected entangled pair state (PEPS). For each 2D lattice, we assign each site i (i is a coordination in 2D) a tensor $A[i]^{s_i}_{d_{i,1}d_{i,2},\ldots}$, where $d_{i,1}d_{i,2}, \ldots$ are virtual indices corresponding to virtual legs to be connected with neighboring sites. Generally, the PEPS can be written as,

$$|\Psi\rangle = \sum_{\{s\},\{d\}} \prod_i A[i]^{s_i}_{d_{i,1}d_{i,2},\ldots} |s_i\rangle\langle d_{i,1}d_{i,2}, \ldots| \prod_{\langle i,j\rangle} EPR_{ij}\,, \tag{5}$$

where EPR_{ij} is the EPR pair on bonds connecting site i and site j.

This can also be expressed compactly as

$$|\Psi\rangle = \sum_{\{s\}} \mathcal{C}\Big[\prod_i A[i]^{s_i}_{d_{i,1}d_{i,2},\ldots}\Big] |s_1, s_2, \ldots, s_N\rangle \tag{6}$$

where $\mathcal{C}$ means to contracts all connected virtual legs.

The number of the virtual legs can be the coordination number of the lattice, e.g., 4 for a square lattice, 3 for a honeycomb lattice and 6 for a triangular lattice. The PEPS of a square lattice and a triangular are graphically represented in Fig. 2. For the square lattice, each tensor in the network has 4 virtual legs that connect to neighbor sites and a physical leg open, whereas for the triangular lattice, there are 6 virtual legs for each tensor. The PEPS satisfies area law [30], therefore it is an efficient representation of the ground states of some strongly correlated many-body systems.

2.3. Other tensor network states (TNS)

The MPS and PEPS both satisfy the area law (MPS in 1D, PEPS in 2D) which can be used to well approximate the ground state of many systems. Along the same way, some other TNSs which also satisfy the area law are constructed, such as SBS [21], PESS [17]. The different TNS has its own advantages in certain situations.

However, there are some systems whose volume entropy is beyond the area law, such as the free fermion system [31]. As a result, the former states cannot be used as the efficient approximation of the states in this situation. Some TNSs which are beyond the area law can also be constructed, e.g., MERA [18].

3. Tensor network states in many-body physics

TNS have many applications in different physical fields, from condensed matter physics to black hole [32]. In this section, we introduce the application of TNS in many-body physics. We start from the method to use TNS to access the ground state of a quantum system.

3.1. Use TNS to access the ground state of a quantum system

As we have mentioned, the ground states of the quantum systems, especially for the gapped systems, can be effectively approximated by the TNS with proper virtual bond dimension D. In this section, we introduce some methods to find an optimal TNS with fixed parameter D to approximate the ground state for a specific Hamiltonian. There are two major methods to optimize the TNS: (i) imaginary time evolution method and (ii) variational method. We take the 1D MPS as an example to explain these methods, which can be generalized to other TNS.

3.1.1. Imaginary time evolution method

In the imaginary time evolution method, we start from a randomly chosen MPS with fixed parameter D. The ground state is obtained by evolving the initial state under imaginary time for long enough time, i.e.,

$$|0\rangle = \lim_{\tau\to\infty} e^{-\tau H}|\Psi\rangle\,, \tag{7}$$

where $|\Psi\rangle$ is the initial state, $|0\rangle$ is the ground state of the Hamiltonian. The algorithm originates from the fact that we can expand the initial state by energy eigenstates of the Hamiltonian, as $|\Psi\rangle = c_n|n\rangle$, where $|n\rangle$ is the eigenstate with nth smallest eigenvalue. We have

$$\begin{aligned} & e^{-\tau H}|\Psi\rangle \\ &= \sum_{n=0} c_n e^{-\tau E_n}|n\rangle \\ &= e^{-\tau E_0}\Big[c_0|0\rangle + \sum_{n=1} e^{-\tau(E_n-E_0)}c_n|n\rangle\Big] \propto |0\rangle\,. \end{aligned} \tag{8}$$

It is easy to see only the lowest energy state $|0\rangle$ survives after imaginary time evolution for long enough time.

Generally, the operator $e^{-\tau H}$ is a global operator. From calculation aspect, we cannot deal with the whole system whose Hilbert space is huge. Therefore, we have to approximate the imaginary time evolution $e^{-\tau H}$ by a set of operators which only act on several local sites. We first divide the time evolution into N segments, i.e.,

$$e^{-\tau H} = \left(e^{-H\Delta\tau}\right)^N\,, \tag{9}$$

where $\Delta\tau = \frac{\tau}{N}$ is a small time step. We then decompose the Hamiltonian H into summation of m parts [14],

$$H = H_1 + \cdots + H_m\,, \tag{10}$$

and each part contains only mutual commuting terms. For example, for the 1D Heisenberg model, we divide the Hamiltonian into $m = 2$ parts. The first part only contains the interaction terms of odd bonds, whereas the second part only contains the even bonds. For a very short time ($\Delta\tau \ll 1$), the time evolution can be well approximated by the production of a series of operators using (2nd order) Trotter decomposition [33] as,

$$\begin{aligned} e^{-\Delta\tau H} &= e^{-\Delta\tau\sum_{i=1}^m H_i} \\ &= \prod_{i=1}^{m} e^{-\Delta\tau H_i}\prod_{i=m}^{1} e^{-\Delta\tau H_i} + \mathcal{O}(\Delta\tau^2). \end{aligned} \tag{11}$$

Using Eq. (9), we have,

$$\begin{aligned} e^{-\tau H} &= \Big[\prod_{i=1}^{m} e^{-\Delta\tau H_i}\prod_{i=m}^{1} e^{-\Delta\tau H_i} + \mathcal{O}(\Delta\tau^2)\Big]^N \\ &= \Big[\prod_{i=1}^{m}\prod_{j} e^{-\Delta\tau H_{i,j}}\prod_{i=m}^{1}\prod_{j} e^{-\Delta\tau H_{i,j}}\Big]^N + \mathcal{O}(\Delta\tau)\,. \end{aligned} \tag{12}$$

We have approximated the original global operator $e^{-\tau H}$ by a successive local actions of $e^{-i\Delta\tau H_{i,j}}$, with controllable precision by $\delta\tau$.

We now apply the imaginary time evolution Eq. (12) to the MPS [11]. For simplicity, we consider the local Hamiltonian $H_{i,i+1}$ only acts on two nearest neighbor ith and $(i+1)$th sites. We show graphically how $e^{-\Delta\tau H_{i,i+1}}$ acts on an MPS state in Fig. 3(a–d). For convenience,

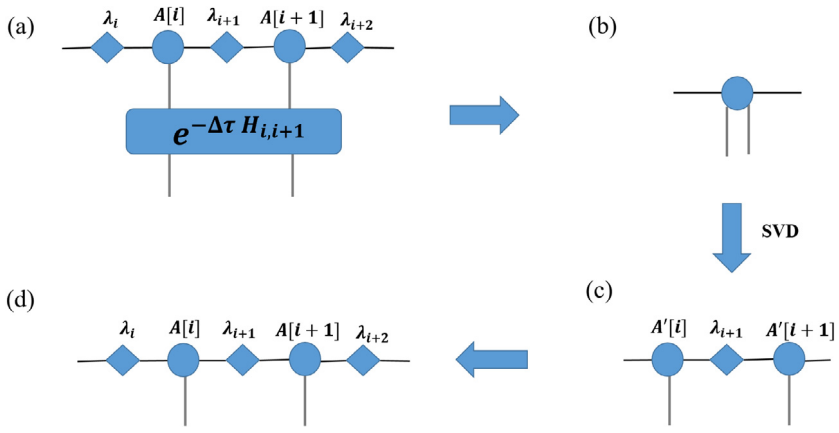

Fig. 3. The procedure of one step imaginary time evolution $e^{-\Delta\tau H_{i,j}}|\Psi_{\mathrm{MPS}}\rangle$ [11]. The diamonds are the environment matrices λ_i, whereas the circles are the tensors $A[i]$. The time evolution operator is a tensor with 4 legs. We first contract the tensors $A[i]$, $A[i+1]$, λ_i, λ_{i+1}, λ_{i+2} and $\exp\{-\Delta\tau H_{i,i+1}\}$ (a); We then perform a SVD to the resulting 4-leg tensor (b); and obtain $A'[i]$, $A'[i+1]$ and λ_i (c); Finally we restore the original format of the MPS, as $A[i] = \lambda_i^{-1}A'[i]$, $A^{[i+1]} = A'[i+1]\lambda_{i+2}^{-1}$ (d).

we slightly modify the form of the MPS, by introducing additional diagonal matrices λ_i $(i = 1, 2, \ldots, N-1)$ into the MPS, which are the Schmidt coefficients of the MPS when it is divided into two parts at site i. λ_i is also known as the "environment". For each site, there is a tensor $A[i]$ represented by a circle, and a matrix λ_i represented by a diamond. The time evolution operation $e^{-\Delta\tau H_{i,i+1}}$ can be expressed as a tensor with four physical legs. The whole processes for one step time evolution are shown in Fig. 3.

First, we contract the tensors $A[i]$, $A[i+1]$, λ_i, λ_{i+1}, λ_{i+2} and $\exp\{-\Delta\tau H_{i,i+1}\}$ [Fig. 3(a)], and the resulting tensor is shown in Fig. 3(b). After the action of the $\exp\{-\Delta\tau H_{i,i+1}\}$, the original format of MPS is modified, where $A[i]$, $A[i+1]$ fuse to a new tensor with four legs. We perform the singular value decomposition (SVD) to separate site i and $i+1$. The left and right SVD tensors are temporarily labeled by $A'[i]$, $A'[i+1]$ respectively, and the matrix λ_{i+1}. The virtual dimension between the site i and $i+1$ will be $D \times d$ where d is the dimension of the physical index. In order to find an optimal approximation of the modified MPS with fixed parameter D, we only keep the D largest singular values in λ_{i+1} and their corresponding vectors $A'[i]$, $A'[i+1]$ [Fig. 3(c)]. Finally, in order to restore the original format of MPS at site i and $i+1$, we need to take off the effect of the environment and update $A[i] = \lambda_i^{-1}A'[i]$ and $A^{[i+1]} = A'[i+1]\lambda_{i+2}^{-1}$ [Fig. 3(d)].

With a set of local imaginary time evolution operators according to Trotter decomposition of a given Hamiltonian H, we can efficiently find the optimal ground state of H in the MPS space with fixed D, which will converge to the real ground state of H with the increasing of D.

3.1.2. Variational method

Besides the imaginary time evolution method, we can also optimize an MPS using a variational method, i.e., to minimize the energy $\langle\Psi|H|\Psi\rangle$, with the constrain $\langle\Psi|\Psi\rangle = 1$. The application of Lagrange multiplier method transforms the question into minimizing the quantity

$$\langle\Psi|H|\Psi\rangle - \lambda(\langle\Psi|\Psi\rangle - 1), \tag{13}$$

with respect to both unnormalized $|\Psi\rangle$ and the multiplier λ.

The stationary solution of Eq. (13) by taking derivatives with respect to λ and tensor elements, leads to the equations [26]

$$\begin{cases} H_{\mathrm{eff}}^{[M]}A[M] = \lambda N_{\mathrm{eff}}^{[M]}A[M] \\ \langle\Psi|\Psi\rangle = 1 \end{cases} \tag{14}$$

where $A[M]$ is the Mth tensor of the MPS, and $H_{\mathrm{eff}}^{[M]}$ and $N_{\mathrm{eff}}^{[M]}$ are tensors defined in Fig. 4. Since the second equation can always be satisfied by rescaling the state, we only need to care about the first equation, which can be expressed graphically in Fig. 4(c). This equation is actually a generalized eigenvalue problem for each M and can be easily solved for a tensor at one time [26]. In practice, we solve the equation for M sequentially from 1 to L and back to 1. We repeat the process until the result converges. With this method, we can find the optimal MPS with fixed D to approximate the ground state of H. By increasing D, the MPS may converge to the real ground state.

3.2. Calculation of the expectation value of physical observables

After we obtain the ground state represented in MPS, we can calculate the physical observables to investigate the properties of the state. To obtain the expectation value of an operator, $\langle O\rangle = \langle\Psi|\hat{O}|\Psi\rangle$, we need to contract the tensor network shown in Fig. 5, where the operator O is treated as a tensor. In practice, we usually express $\hat{O}$ as sum of local terms and calculate them separately [26].

3.3. Other applications of TNS

TNS can also be used to simulate dynamical systems and systems at finite temperature. Interestingly, it has been reported that MERA is an efficient expression of the conformal field on edge of an anti-deSitter(AdS) space [34], which is the holographic duality to the bulk space. This sheds light on our understanding of the nature of quantum gravity [35]. Furthermore, making use of the resemblance of big data processing and statistical physics, TNS has been used in the fields of machine learning [36]. Such applications can be conducted more efficiently using our tensor package.

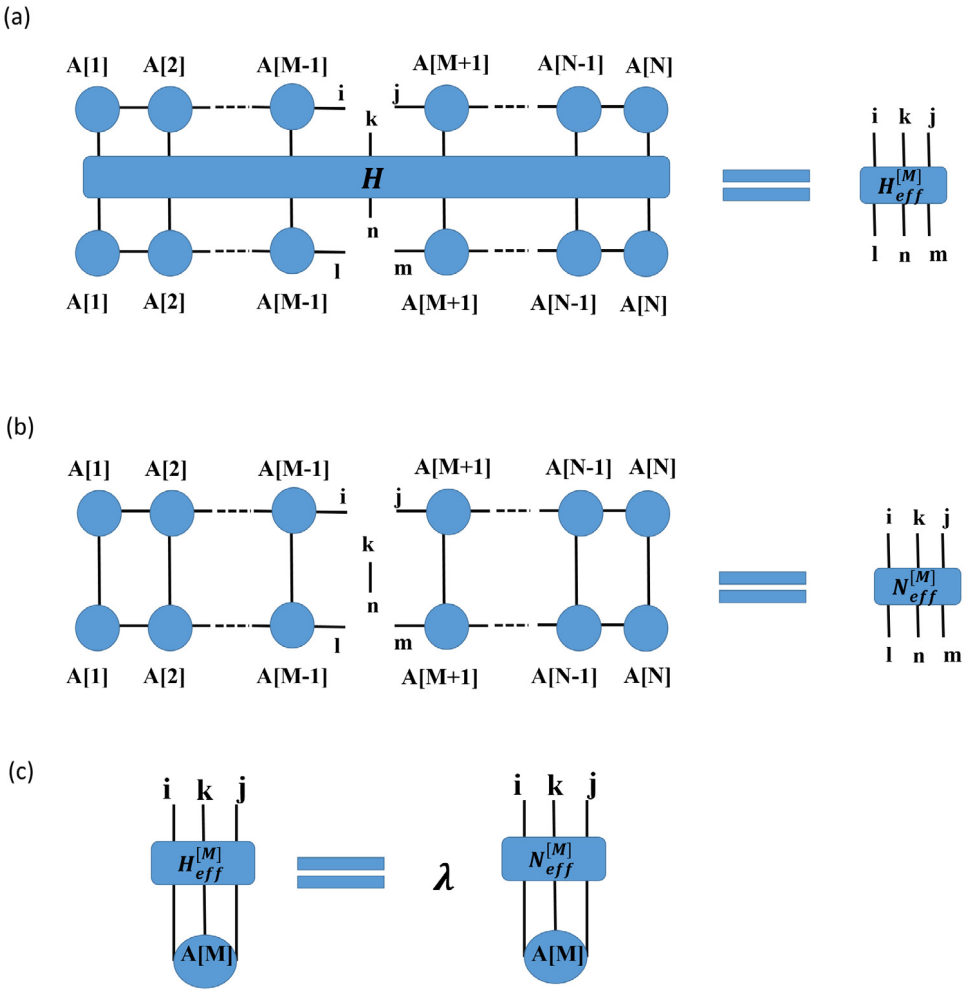

Fig. 4. The tensor network for (a) $H^{[M]}_{\text{eff}}$ and (b) $N^{[M]}_{\text{eff}}$, where the circles are the tensors $A[i]$, and the square represents the Hamiltonian H. The resulting $H^{[M]}_{eff}$ and $N^{[M]}_{\text{eff}}$ are 6-leg tensors, where i, j, k, l, m, n are uncontracted indices. (c) Tensor network representation of equation $H^{[M]}_{\text{eff}}A[M] = \lambda N^{[M]}_{\text{eff}}A[M]$, where the circles are the eigenvectors $A[M]$, and squares represent the effective Hamiltonian $H^{[M]}_{\text{eff}}$ and $N^{[M]}_{\text{eff}}$.

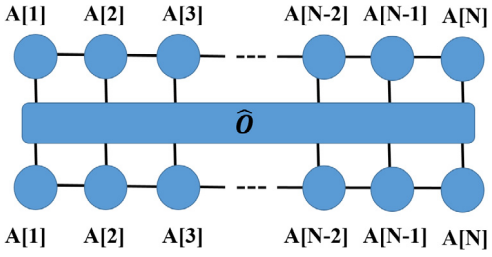

Fig. 5. The tensor network for $\langle O\rangle = \langle \Psi_{\text{MPS}}|O|\Psi_{\text{MPS}}\rangle$, where the circles are the tensors $A[i]$.

4. Basic operations in tensor network state methods

As discussed in previous sections, there are different types of tensors that are used in different TNS methods, and there are also various methods to achieve the ground states and calculate expectation values. Therefore it is hard to develop a universal code for the TNS methods as those widely used first-principles packages based on density functional theory. However, after careful analysis of the TNS methods, we find the TNS methods do share many similar basic operations, especially on the tensors. In this section, we summarize the commonly used basic operations in the TNS methods.

4.1. Contraction of two tensors

Contraction of two tensors is one of the key operations that are widely used in TNS. Generally, when two tensors are connected with legs, we need to sum the bond dimension of these legs. This process is called "contraction". As an example, the contraction of two tensors

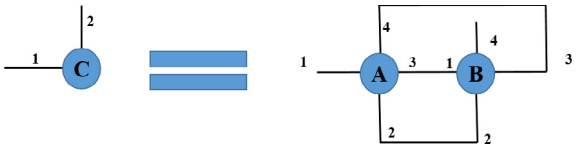

Fig. 6. The contraction of tensors A and B results in tensor C. The numbers label the order of the legs in the tensors.

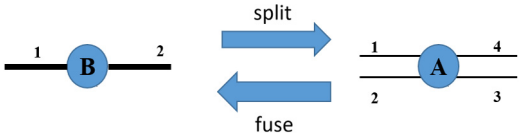

Fig. 7. The fusion of the first leg with the second leg, and the third leg and the fourth leg of A results in a two-leg tensor B. Splitting is a reverse operation of fusion.

$A_{j_1,\alpha_1,\alpha_2,\alpha_3}$ and $B_{\alpha_2,\alpha_1,\alpha_3,j_2}$ are schematically shown in Fig. 6. It tells us that we should sum all connected bonds, namely the 2nd leg of A with 2nd leg of B, the 3rd leg of A with the 1st leg of B and the 4th leg of A with the 3rd leg of B. The resulting tensor after the contraction is a two-leg tensor C, with one leg comes from the first leg of A and the other from the 4th leg of B. The formula of the graphical notation is:

$$C_{j_1,j_2} = \sum_{\alpha_1,\alpha_2,\alpha_3} A_{j_1,\alpha_1,\alpha_2,\alpha_3} B_{\alpha_2,\alpha_1,\alpha_3,j_2} \,. \tag{15}$$

For spins and bosons, the order of the contraction is not important. However, for fermions, one must be careful about the contraction order.

Tensor contraction is one of the most time-consuming parts of the TNS method, especially when the tensors have many legs. One also has to be very careful to contact the correct legs.

4.2. Fuse legs of a tensor

Sometimes we need to reshape a tensor, for example to a matrix to do singular value decomposition (SVD) etc. In this case, we need to combine two or more legs of the tensor into a new leg, and the dimension of the new leg is the production of the dimensions of the original legs. This process is called fusion of the legs, which are graphically shown in Fig. 7, where we fuse the 1st leg with the 2nd leg and the 3rd leg with the 4th leg of tensor A. After fusion, the resulting tensor B is a two-leg matrix.

4.3. Split a leg of a tensor

The splitting of a leg of a tensor to two legs can be viewed as the inverse operation of the fusion operation. Generally, after a fusion operation and some other operations, we need a splitting operation to recover the original form of the tensor. Taking the example of Fig. 7, we can split the two legs of B back to four legs in A. The order of legs in tensor A should be carefully arranged to avoid confusion in future operations.

4.4. Tensor leg permutation

The *permutation* is used to reorder the legs of a tensor. It plays the key role in the fusion operation, because to fuse two legs, one need to permute the two legs to the adjacent positions. The permutation operation can be very time-consuming for high rank tensors.

5. Overview of the package

The TNSpackage is written in Fortran2003. We use an object oriented programming (OOP) style to improve the readability and reusability. So far it has about 42000 lines, and more than 200 subroutines and functions which are grouped into 8 modules: Tensor.f90, function.f90, Dimension.f90, print_in_TData.f90, element_in_TData.f90, modify_in_TData.f90, permutation_in_TData.f90, TData.f90, among which Tensor.f90 is the main module. To use the package, one need to include the Tensor.f90 module in the codes. The package have been tested to be a stable version. With high reusability, the users are able to write their own children objects through inheritance. The procedure polymorphism and data polymorphism of the package make it suitable for coding different algorithms for TNS or for other purposes involving high dimension arrays.

In the package, we define a new data type `Tensor`, whose elements are stored in a one-dimensional array. The new data type `Tensor` supports all data types of Fortran, including integer, single/double precision real number, single/double precision complex number, logical and character. We overwrite most functions of Fortran2003, such as dcmplx, max, '.eq.', '+', '-', '*', '/' and so on, so they can be directly applied to `Tensor`, as they are applied to other fundamental data types.

In Section 4, we summarize the most often used tensor operations in the TNS methods. We design accordingly the functions/subroutines to simplify these tensor operations. We list in Table 1 the most important and often used functions and subroutines for TNS.

We carefully design the data structure of `Tensor` to speed up the performance and make our package more user-friendly. Some high rank tensor operations can be very time consuming if not designed wisely. For example, reshaping tensors (including fuse, and split the legs etc.) may be time-consuming if one really moves the data around in the memory, but in our package the operations of fusing or

Table 1
Most used functions and subroutines in the package.

Functions /subroutines	Purpose	Examples
setName	give names to the dimension of the Tensors	call T%setName(1,'T.L')
fuse	fuse 2 or more legs of a Tensor into one leg	call T%fuse(1,3)
split	reverse operation of fuse	call T%split()
permute	reorder the legs of a Tensor	call T%permute([3,2,1])
contract	contraction of two Tensors	A=contract(B,1,C,2)
SVDTensor	SVD function	SVD=T%SVDTensor()
eig	output the eigenvalue of the Tensor	eig=T%eig()
QRTensor	QR decomposition of a Tensor	QR=T%QRTensor
LQTensor	LQ decomposition of a Tensor	LQ=T%LQTensor
max	output the max element of the Tensor	a=T%max()
min	output the min element of the Tensor	a=T%min()
norm	output the norm of the Tensor	a=T%norm()
pointer	output a pointer which point to the data of the Tensor	call T%pointer(data)

splitting legs of a tensor will cost almost no CPU time as we only modify data structure of the tensor, which does not actually move the data themselves. We also make great effort to optimize the *permeation* operation which is heavily used in the code.

We use high performance mathematic packages including Arpack [22], Lapack [23] and Blas [23] to improve the efficiency for linear algebraic operations, such as contractions (matrix multiplications), QR decompositions, SVD etc. Direct use of these subroutines is tedious and cumbersome. To ease the coding work, these mathematic subroutines have been encapsulated in the TNSpackage, and the users who use these packages through calling subroutines of TNS package, do not need to worry about the details on how they are implemented. For example, the contraction of two tensors (see Eq. (15)), involves the product of two matrices or a matrix to a vector. The TNSpackage calls XGEMM for the product of two matrices and XGEMV for the product for a matrix and a vector, where X is 'S', 'D', 'C', 'Z' standing for single/double real/complex data. The user can just use a short code like "`A=contract(B,1,C,2)`", to contract the first leg of B with the second leg of C, and the package will automatically choose proper subroutines and parameters for the operations, which greatly simplifies the coding work. Similarly, one can do SVD, QR/LQ decompositions and other linear algebraic operations in this simple way.

As shown in Section 4, most of the basic operations of tensors are on some specified legs of the tensors. In TNS, each leg has its physical meaning, and the order of the legs is of great importance. However, in many operations, the tensor legs are permuted, fused or split. As a result, it will be very difficult to trace the correct legs after a few operations on the tensors. One has to be extremely careful to ensure the operation is on the correct legs. To overcome this complexity in coding TNS, we may assign a 'name' to every leg of the Tensor. The 'name' of the tensor legs are characters in the form of '`TensorName.DimensionName`', where `TensorName` is used to track the tensor name which the legs belong to and `dimensionName` is to track the legs in the tensor. We use "." to divide the tensor name and leg name. In the process of contraction, the legs that have been contracted (including their names) will be removed, but the remaining ones are saved in the new `Tensor`. By reading the name of the leg we can easily tell which `Tensor` the leg original comes from and its physical meaning. All the functions/subroutines in Table 1 can be used by specifying legs with `TensorName.DimensionName`. With the help of the `TensorName.DimensionName`, one does not need to consider the actual order of the legs in the tensors, which greatly simplifies the coding work.

6. Example codes for PEPS

In this section, we demonstrate how to use the TNSpackage by an example of solving the Heisenberg model on a square lattice using PEPS. The Hamiltonian reads,

$$H = \sum_{\langle i,j \rangle} \mathbf{S}_i \cdot \mathbf{S}_j \,, \tag{16}$$

where $\sum_{\langle i,j \rangle}$ denotes the summation of the nearest-neighbor pairs. We assume the periodic boundary condition. The wave functions of the Heisenberg model are presented PEPS, and we optimize the wave functions using a simple update time evolution method [37]. We first give examples on some important ingredients of this algorithm to introduce step by step the usage of the TNS package. The full program is given in the Appendix.

6.1. Data structure of PEPS wave functions

We first design a data structure `type Node`, to store all necessary data on each lattice site.

```
type Node
   type(Tensor)::site  ! to store PEPS tensor of each site.
   type(Tensor)::Up    ! to store the tensor of the environment
   type(Tensor)::Down  ! to store the tensor of the environment
   type(Tensor)::Left  ! to store the tensor of the environment
   type(Tensor)::Right ! to store the tensor of the environment
end type Node
```

`Tensor site` is to store the main data of PEPS wave functions, which is a rank 5 tensor, including 4 ranks for the 4 virtual bonds, and one to store the physical indices. Besides the main PEPS tensor, we also define 4 tensors: `Up`, `Down`, `Left` and `Right` to store the environment information for the 4 virtual bonds, each of them is a rank 2 tensor, i.e., a matrix. The graphic representations of `Node A, B` are shown in Fig. 8(a).

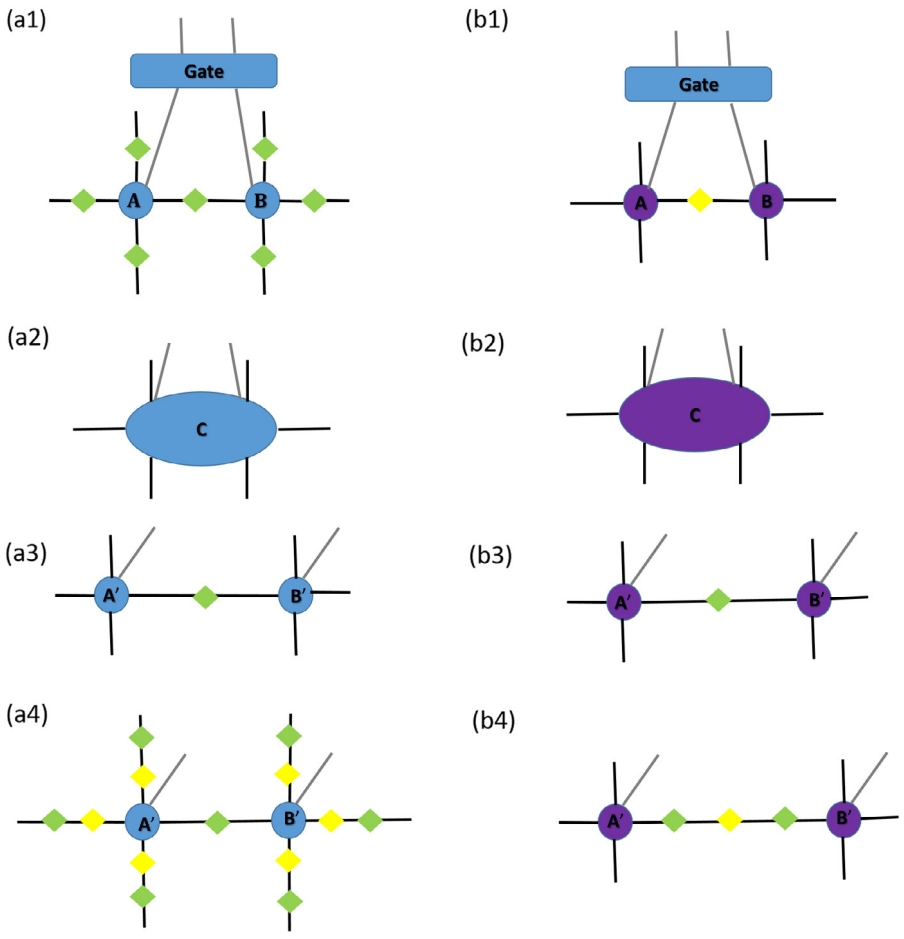

Fig. 8. The green diamonds denote the environment tensors and the yellow ones are the inverse of them. The circles are the node in the tensor network. Left panel: The standard procedure of one step imaginary time evolution. (a1) Contracting tensors A, B, with all environments and the time evolution gate. (a2) The resulting tensor C after contraction. (a3) The resulting tensors obtained by performing SVD on the tensor in (a2). (a4) Restore the original form of tensors in (a1) by contracting the inverse of the environments (the yellow diamonds). Right panel: The implemented scheme of one step imaginary time evolution. The stored tensors A and B have already included the environments. (b1) Contract tensor A, B, the inverse of the environment between them and the time evolution gate. (b2) The result of (b1) which is equivalent to (a2). (b3) The resulting tensors obtained by performing SVD on the tensor in (b2), which is equivalent to (a3). (b4) Contract the new environment (the green ones) and restore the original tensor form in (b1). This scheme may save three environment contractions from the standard procedures. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

6.2. Generate PEPS wave functions

In the following subroutine, we generate a random PEPS as a starting wave function on the L1×L2 lattice. `A(i,j)` with `i=1,...,L1`, and `j=1,...,L2`, are an array of data type `Node`. We first initialize tensor `A(i,j)%site` in each `Node` structure to store the PEPS wave function, where the first 4 legs with dimension `D` is to store the virtual bonds, and the last one to store the physical index, with $d = 2$ (spin up and down). (Please note that our package is not limited to d=2. One may use arbitrary physical dimensions in their codes, which is important for Boson models.) We define the tensor elements as single precision real numbers by specifying the variable `datatype` in the code. For convenience, we assign a name to each leg of `A(i,j)%site` to avoid confusion in future operations. Next we set up the environment tensors `Up`, `Down`, `Left`, `Right` as identity matrices.

```
   subroutine initialPEPS(A,L1,L2,D)
      type(Node),allocatable,intent(inout)::A(:,:)
      integer,intent(in)::L1,L2,D
      integer::i,j
      allocate(A(L1,L2))
      do j=1,L2
         do i=1,L1
1           A(i,j)%site=generate([D,D,D,D,2],datatype)
2           call A(i,j)%site%setName(1,'A'+i+'_'+j+'.Left')
            call A(i,j)%site%setName(2,'A'+i+'_'+j+'.Down')
            call A(i,j)%site%setName(3,'A'+i+'_'+j+'.Right')
            call A(i,j)%site%setName(4,'A'+i+'_'+j+'.Up')
            call A(i,j)%site%setName(5,'A'+i+'_'+j+'.phy')
```

Table 2
"Datatype" and its value.

Datatype	Data type of Tensor
'integer'	integer
'real'	real(kind=4)
'real*4'	real(kind=4)
'real(kind=4)'	real(kind=4)
'double'	real(kind=8)
'real*8'	real(kind=8)
'real(kind=8)'	real(kind=8)
'complex'	complex(kind=4)
'complex*8'	complex(kind=4)
'complex(kind=4)'	complex(kind=4)
'complex*16'	complex(kind=8)
'complex(kind=8)'	complex(kind=8)
'logical'	logical
'character'	character(len=*)

```
3           call A(i,j)%Up%allocate([D,D],datatype)
4           call A(i,j)%Up%eye()
            call A(i,j)%Up%setName(1,'Lambda.Leg1')
            call A(i,j)%Up%setName(2,'Lambda.Leg2')
5           A(i,j)%Down=A(i,j)%Up
            A(i,j)%Left=A(i,j)%Up
            A(i,j)%Right=A(i,j)%Up
         end do
      end do
      return
   end subroutine
```

where

1 Generate `site` as a $D \times D \times D \times D \times 2$ Tensor, whose elements are random real numbers. The data type is which are specified by the parameter `datatype`.
2 Set the name `Ai_j%Left` to the first leg the Tensor. Here `i` and `j` are integers, which are used for the lattice site indices. We have overwrite the operator (+), so `i` and `j` are treated as characters in the operation `'A'+i+'_'+j+'.Left'`.
3 Allocate memory for `A(i,j)%Up`, which is a $D \times D$ matrix, and the data type is real number.
4 Set the `A(i,j)%Up` as a identity matrix.
5 Copy tensor `A(i,j)%Up` to `A(i,j)%Down`.

The data type of a tensor can be integer, real(kind=4), real(kind=8), complex(kind=4), complex(kind=8), logical and character as listed in Table 2. User can use the following code to specify the data type of a Tensor:

```
type(Tensor)::T
   character(len=20)::datatype
   datatype='complex*8'
   call T%setType(datatype)
```

where the character `datatype` specifies the data type of `T`. If one does not specify the data type of the `Tensor`, the package will automatically choose a suitable data type for it. For example, we can copy a tensor to another using the following code:

```
   T1=T2 !copy tensor T2 to T1
```

Here, if the data type of `T1` have been set before by calling `T1%setType(datatype)`, the data type will be determined by the specified data type, otherwise, the data type is set to the same as that of `T2`.

6.3. Define the time evolution operator as a quantum gate

To calculate the ground state of the model using the imaginary time evolution method, we need to define the Hamiltonian in terms of two-body operators $H_{1,2} = \mathbf{S}_1 \cdot \mathbf{S}_2 = (S_1^x \otimes S_2^x) + (S_1^y \otimes S_2^y) + (S_1^z \otimes S_2^z)$, where $S_{1,2}^{x,y,z}$ are the $\frac{1}{2}$ Pauli matrix and $\otimes$ is the direct product operator. The operator $H_{1,2}$ can be viewed as a 4-leg tensor. In the following subroutine, we define a gate operator $\text{Gate} = \exp(-\tau H_{1,2})$, where τ is the length of the time step, to perform time evolution between two sites. We first reshape the 4-leg tensor $H_{1,2}$ to a matrix(2-leg Tensor) by fusing leg 1 with leg 2, and leg 3 with leg 4. We then perform time evolution $e^{-\tau H_{1,2}}$ on the matrix. Finally, we reshape (split) the resulting matrix to its original form of a 4-leg tensor.

```
   subroutine initialH(H)
      type(Tensor),intent(inout)::H
      type(Tensor)::Sx,Sy,Sz
```

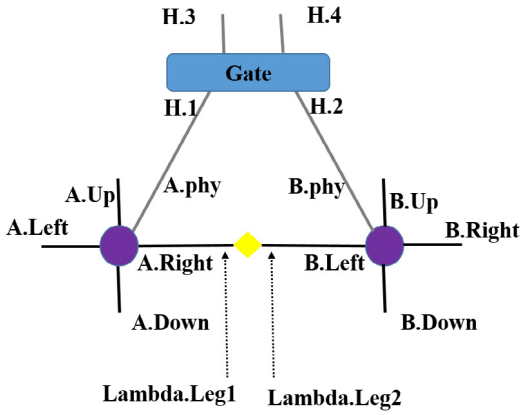

Fig. 9. The contraction of tensors A, B with the inverse of environment and the time evolution gate. The yellow diamond stands for the inverse of the environment, whereas the purple circles are the tensors, with the names of the legs. The physical legs are in gray.

```
1      call pauli_matrix(Sx,Sy,Sz,0.5)
       call H%setType(datatype)
2      H=(Sx.xx.Sx)+(Sy.xx.Sy)+(Sz.xx.Sz)
3      call H%setName('H')
       return
   end subroutine

   type(Tensor) function gate(H,tau)
      type(Tensor),intent(in)::H
      real,intent(in)::tau
      gate=H*tau
4      call gate%fuse(1,2)
       call gate%fuse(2,3)
5      gate=expm(gate)
6      call gate%split()
       return
   end function
```

where

1 Generate the 1/2 pauli matrices for S_x, S_y and S_z.
2 The operator (`.xx.`) is the direct product operator, e.g., `C=A.xx.B` is defined as $C_{i,j,k,l} = A_{i,k} * B_{j,l}$.
3 Set TensorName 'H' to tensor H, and the legs of tensor H are automatically named by order: 'H.1', 'H.2', 'H.3' and 'H.4'.
4 Fuse the 1st and the 2nd legs of tensor 'gate' into a new leg.
5 Calculate e^A, where A is a matrix.
6 Split all the legs of the 'gate' tensor back to it original form (a 4 leg tensor).

6.4. Imaginary time evolution

One step time evolution involves 3 steps as shown in Fig. 8 (a1)–(a4): (i) Contract the environments to the tensors A, B, and then contract with gate. The resulting tensor C is shown in Fig. 8(a2). (ii) Do SVD to tensor C [Fig. 8(a3)]; (iii) Contract the inverse of the environments, so all tensors are back to theirs original formats [Fig. 8(a4)]. In our implementation, we slightly change the procedures, which are shown in Fig. 8(b1)–(b4). In our scheme, tensors A and B are the ones that already have contracted the environments. Therefore at the first step, one should contract the inverse of the environment between the tensor A and the tensor B. At the final step, both tensor A and tensor B need to contract the new environment tensor. In this scheme, one can save three environment contractions in each time step from the standard procedures.

In the following subroutine, we demonstrate how to use the gate tensor to perform time evolution of two sites A and B. For convenience, we assign names to the legs of the two tensors and the gate, where AName is the TensorName of tensor A and BName is the TensorName of B; cha1 and cha2 specify the legs to be contracted during time evolution. Lambda is the environment between A and B. The contraction of time evolution gate is shown in Fig. 9, where AName='A', BName='B', cha1='Right' and cha2='Left'. The yellow diamond is the inverse of the environment (Lambda^{-1}) between tensors A and B.

```
   subroutine step(A,Lambda,B,expmH,CutOff,AName,BName,cha1,cha2)
      type(Tensor),intent(inout)::A,Lambda,B
      type(Tensor),intent(in)::expmH
      character(len=*),intent(in)::cha1,cha2,AName,BName
```

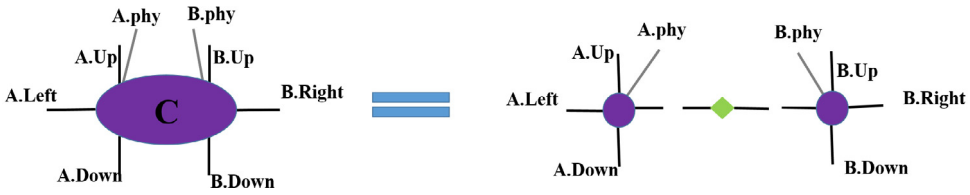

Fig. 10. The SVD of tensor C after time evolution gate operation (Fig. 9). The green diamond stands for the environment Lambda. The new legs that come out from site A and site B do not have names, where other legs keep their original names before gate operation.

```
      integer,intent(in)::CutOff
      type(Tensor)::C,SVD(3)
1     C=contract(A,AName+'.'+cha1,Lambda%invTensor(),'Lambda.Leg1')
2     C=contract(C,'Lambda.Leg2',B,BName+'.'+cha2)
3     C=contract(C,[AName+'.phy',BName+'.phy'],expmH,['H.1','H.2'])
4     call C%setName('H.3',AName+'.phy')
5     call C%setName('H.4',BName+'.phy')
6     SVD=C%SVDTensor(AName,BName,CutOff)
7     Lambda=eye(SVD(2))/SVD(2)%smax()
8     A=SVD(1)*Lambda
9     call A%setName(A%getRank(),AName+'.'+cha1)
      B=Lambda*SVD(3)
10    call B%setName(1,BName+'.'+cha2)
      call Lambda%setName(1,'Lambda.Leg1')
      call Lambda%setName(2,'Lambda.Leg2')
      return
   end subroutine
```

where

1 Contract leg `AName.cha1` from tensor `A` with leg `Lambda.Leg1` from environment tensor `invTensor(Lambda)`, where function `invTensor()` gives the inverse of a matrix.
2 Contract leg `Lambda.Leg2` from tensor `C` with leg `BName.cha2` from tensor `B`.
3 Contract leg `AName.phy` (spin index of tensor `A`) and leg `BName.phy` (spin index of tensor `B`) from tensor `C` with the legs `H.1` and `H.2` from gate tensor `expmH`.
4 Rename leg `H.3` of tensor `C` to `AName.phy`.
5 Rename leg `H.4` of tensor `C` to `BName.phy`.
6 Do `SVD` to tensor `C`, such that `C=SVD(1)*eye(SVD(2))*SVD(3)`. Function `eye()` reformats a vector to a diagonal matrix. The legs from `AName` are regarded as row, and the legs from `BName` are regarded as col. In output, `SVD(1)` are the left singular vectors, `SVD(2)` store the singular values and `SVD(3)` are the right vectors. After SVD, the resulting two tensors automatically restore the structure of tensor `A` and `B`, according to tensor names (see Fig. 10) , except the last leg of tensor `A` and first leg of tensor `B`, whose dimensions are determined by the input parameters `cutoff`. This procedure is shown in Fig. 10.
7 Update environment tensor `Lambda`, which is normalized by its maximum element.
8 Update PEPS tensor `A` after time evolution.
9 Set name `AName.+cha1` to the last leg of the updated tensor `A`, which has been contracted during the time evolution, so it can be used in the next time evolution steps. `A%getRank()` get the rank (number of legs) of tensor `A`.
10 Set name `BName.+cha2` to the first leg of the updated tensor `B`, which has been contracted during the time evolution.

If one does not want to use the leg names for the operations, one may also explicitly specify the legs by their orders in the tenors to be used. However, doing this way will be more complicated and elusive than using tensor and leg names, because one has to track the orders of tensors' legs which may change during the operations.

7. Summary

We have developed a Fortran2003 package namely TNSpackage that integrates most often used tensor operations designed for the TNS methods. Great efforts has been spent to optimize the efficiency of the tensor operations. The package is user-friendly and flexible for different types of TNS, and therefore greatly simplifies the coding work for developing the TNS methods. The package is also very useful for other algorithms that involving high rank tensor operations.

Acknowledgments

The authors thank Prof. Hong An and Chao Yang for valuable discussions. This work was funded by the Chinese National Science Foundation (Grant numbers 11374275, 11474267), the National Key Research and Development Program of China (Grant No. 2016YFB0201202). The numerical calculations have been done on the USTC HPC facilities.

Appendix

In this appendix, we give a complete example of imaginary time evolution of the Heisenberg model on a 4×4 square lattice, with the periodic boundary condition. We use simple update method, proposed in Ref. [37].

```
module example_PEPS
   use Tensor_type
   use usefull_function
   implicit none
   character(len=20)::datatype='real'
   type Node
      type(Tensor)::site
      type(Tensor)::Up
      type(Tensor)::Down
      type(Tensor)::Left
      type(Tensor)::Right
   end type Node
contains

   subroutine initialH(H)
      type(Tensor),intent(inout)::H
      type(Tensor)::Sx,Sy,Sz
      call pauli_matrix(Sx,Sy,Sz,0.5)
      call H%setType(datatype)
      H=(Sx.xx.Sx)+(Sy.xx.Sy)+(Sz.xx.Sz)
      call H%setName('H')
      return
   end subroutine

   type(Tensor) function gate(H,tau)
      type(Tensor),intent(in)::H
      real,intent(in)::tau
      gate=H*tau
      call gate%fuse(1,2)
      call gate%fuse(2,3)
      gate=expm(gate)
      call gate%split()
      return
   end function

   subroutine initialPEPS(A,L1,L2,D)
      type(Node),allocatable,intent(inout)::A(:,:)
      integer,intent(in)::L1,L2,D
      integer::i,j
      allocate(A(L1,L2))
      do j=1,L2
         do i=1,L1
            A(i,j)%site=generate([D,D,D,D,2],datatype)
            call A(i,j)%site%setName(1,'A'+i+'_'+j+'.Left')
            call A(i,j)%site%setName(2,'A'+i+'_'+j+'.Down')
            call A(i,j)%site%setName(3,'A'+i+'_'+j+'.Right')
            call A(i,j)%site%setName(4,'A'+i+'_'+j+'.Up')
            call A(i,j)%site%setName(5,'A'+i+'_'+j+'.phy')
            call A(i,j)%Up%allocate([D,D],datatype)
            call A(i,j)%Up%eye()
            call A(i,j)%Up%setName(1,'Lambda.Leg1')
            call A(i,j)%Up%setName(2,'Lambda.Leg2')
            A(i,j)%Down=A(i,j)%Up
            A(i,j)%Left=A(i,j)%Up
            A(i,j)%Right=A(i,j)%Up
         end do
      end do
      return
   end subroutine

   subroutine step(A,Lambda,B,expmH,CutOff,AName,BName,cha1,cha2)
      type(Tensor),intent(inout)::A,Lambda,B
```

```
    type(Tensor),intent(in)::expmH
    character(len=*),intent(in)::cha1,cha2,AName,BName
    integer,intent(in)::CutOff
    type(Tensor)::C,SVD(3)
    C=contract(A,AName+'.'+cha1,Lambda%invTensor(),'Lambda.Leg1')
    C=contract(C,'Lambda.Leg2',B,BName+'.'+cha2)
    C=contract(C,[AName+'.phy',BName+'.phy'],expmH,['H.1','H.2'])
    call C%setName('H.3',AName+'.phy')
    call C%setName('H.4',BName+'.phy')
    SVD=C%SVDTensor(AName,BName,CutOff)
    Lambda=eye(SVD(2))/SVD(2)%smax()
    A=SVD(1)*Lambda
    call A%setName(A%getRank(),AName+'.'+cha1)
    B=Lambda*SVD(3)
    call B%setName(1,BName+'.'+cha2)
    call Lambda%setName(1,'Lambda.Leg1')
    call Lambda%setName(2,'Lambda.Leg2')
    return
end subroutine

subroutine sampleUpdate(A,expmH,L1,L2,CutOff)
    type(Node),intent(inout)::A(:,:)
    type(Tensor),intent(in)::expmH
    integer,intent(in)::L1,L2,CutOff
    integer::i,j
    do i=1,L1
       do j=1,L2-1
          call step(A(i,j)%Site,A(i,j)%Right,A(i,j+1)%Site&
            ,expmH,CutOff,'A'+i+'_'+j,'A'+i+'_'+(j+1),'Right','Left')
          A(i,j+1)%Left=A(i,j)%Right
       end do
       call step(A(i,L2)%Site,A(i,L2)%Right,A(i,1)%Site&
            ,expmH,CutOff,'A'+i+'_'+L2,'A'+i+'_1','Right','Left')
       A(i,1)%Left=A(i,L2)%Right
    end do
    do j=1,L2
       do i=1,L1-1
          call step(A(i,j)%Site,A(i,j)%Down,A(i+1,j)%Site&
             ,expmH,CutOff,'A'+i+'_'+j,'A'+(i+1)+'_'+j,'Down','Up')
          A(i+1,j)%Up=A(i,j)%Down
       end do
       call step(A(L1,j)%Site,A(L1,j)%Down,A(1,j)%Site&
           ,expmH,CutOff,'A'+L1+'_'+j,'A1_'+j,'Down','Up')
       A(1,j)%Up=A(L1,j)%Down
    end do
    return
end subroutine

subroutine PEPSrun()
    integer::L1,L2,D,i,runningnum,runningTime
    real::tau
    type(Node),allocatable::A(:,:)
    type(Tensor)::H,expmH
    L1=4
    L2=4
    D=4
    tau=0.1
    runningnum=50
    call initialPEPS(A,L1,L2,D)
    call initialH(H)
    expmH=expmTensor(H,tau)
    do runningTime=1,4
       do i=1,runningnum
          call sampleUpdate(A,expmH,L1,L2,D)
          tau=tau*0.1
       end do
```

```
      end do
      return
   end subroutine
end module

program Example
   use example_PEPS
   call PEPSrun()
end
```

References

[1] V.A. Kashurnikov, A.V. Krasavin, J. Exp. Theor. Phys. 105 (1) (2007) 69–78.
[2] Olav F. Syljuåsen, Anders W. Sandvik, Phys. Rev. E 66 (2002) 046701.
[3] Steven R. White, Phys. Rev. Lett. 69 (1992) 2863–2866.
[4] Steven R. White, Phys. Rev. B 48 (1993) 10345–10356.
[5] M. Troyer, U.-J. Wiese, Phys. Rev. Lett. 94 (2005) 170201.
[6] U. Schollwöck, Ann. Physics 326 (2011) 96–192.
[7] Guifré Vidal, Phys. Rev. Lett. 91 (2003) 147902.
[8] F. Verstraete, J.I. Cirac, Renormalization algorithms for quantum-many body systems in two and higher dimensions, eprint arXiv:cond-mat/0407066, July 2004.
[9] C. Karrasch, J.H. Bardarson, J.E. Moore, Phys. Rev. Lett. 108 (2012) 227206.
[10] Y.K. Huang, Pochung Chen, Y.-J. Kao, T. Xiang, Phys. Rev. B 89 (2014) 201102.
[11] G. Vidal, Phys. Rev. Lett. 98 (2007) 070201.
[12] T. Barthel, New J. Phys. 15 (7) (2013) 073010.
[13] F. Verstraete, M.M. Wolf, D. Perez-Garcia, J.I. Cirac, Phys. Rev. Lett. 96 (2006) 220601.
[14] F. Verstraete, V. Murg, J.I. Cirac, Adv. Phys. 57 (2008) 143–224.
[15] L. Wang, Z.-C. Gu, F. Verstraete, X.-G. Wen, Phys. Rev. B 94 (2016) 075143.
[16] W.-Y. Liu, S.-J. Dong, Y.-J. Han, G.-C. Guo, L. He, Gradient optimization for finite projected entangled pair states, eprint arXiv:cond-mat/1611.09467, 2016.
[17] Z.Y. Xie, J. Chen, J.F. Yu, X. Kong, B. Normand, T. Xiang, Phys. Rev. X 4 (2014) 011025.
[18] G. Vidal, Phys. Rev. Lett. 101 (2008) 110501.
[19] H.J. Liao, Z.Y. Xie, J. Chen, Z.Y. Liu, H.D. Xie, R.Z. Huang, B. Normand, T. Xiang, Gapless spin-liquid ground state in the s=$\frac{1}{2}$ kagome antiferromagnet. eprint arXiv:cond-mat/1610.04727, 2016.
[20] G. Evenbly, G. Vidal, Phys. Rev. Lett. 104 (2010) 187203.
[21] A. Sfondrini, J. Cerrillo, N. Schuch, J.I. Cirac, Phys. Rev. B 81 (21) (2010) 214426.
[22] Arpack, http://www.caam.rice.edu/software/ARPACK.
[23] Lapack and blas, http://www.netlib.org/lapack.
[24] S.-J. Dong, W. Liu, X.-F. Zhou, G.-C. Guo, Z.-W. Zhou, Y.-J. Han, L. He, Tunneling frustration induced peculiar supersolid phases in the extended bose-hubbard model, eprint arXiv:1610.06042, October 2016.
[25] C. Wang, M. Gong, Y. Han, G. Guo, L. He, Exotic spin phases in two dimensional spin–orbit coupled models: Importance of quantum fluctuation effects, eprint arXiv:1611.09467, January 2017.
[26] Ulrich. Schollwöck, Philos. Trans. R. Soc. Lond. Ser. A Math. Phys. Eng. Sci. 369 (1946) (2011) 2643–2661.
[27] M.B. Hastings, J. Stat. Mech. Theory Exp. 8 (2007) 08024.
[28] U. Schollwöck, Rev. Modern Phys. 77 (2005) 259–315.
[29] Charles H. Bennett, David P. DiVincenzo, John A. Smolin, William K. Wootters, Phys. Rev. A 54 (1996) 3824–3851.
[30] J. Eisert, M. Cramer, M.B. Plenio, Rev. Modern Phys. 82 (2010) 277–306.
[31] Michael M. Wolf, Phys. Rev. Lett. 96 (2006) 010404.
[32] F. Pastawski, B. Yoshida, D. Harlow, J. Preskill, J. High Energy Phys. 2015 (6) (2015) 149.
[33] N. Hatano, M. Suzuki, in: A. Das, B.K. Chakrabarti (Eds.), Lecture Notes in Physics, in: Lecture Notes in Physics, Vol. 679, Springer Verlag, Berlin, 2005, p. 37.
[34] B. Swingle, Constructing holographic spacetimes using entanglement renormalization, 2012.
[35] Adam R. Brown, Daniel A. Roberts, Leonard Susskind, Brian Swingle, Ying Zhao, Phys. Rev. D 93 (2016) 086006.
[36] A. Cichocki, Era of big data processing: A new approach via tensor networks and tensor decompositions, arXiv e-prints arXiv:1403.2048 March 2014.
[37] H.C. Jiang, Z.Y. Weng, T. Xiang, Phys. Rev. Lett. 101 (2008) 090603.