A practical tutorial on DMRG (and DMRG like) algorithm(s)

Nishan Ranabhat, Guglielmo Lami

December 6, 2022

1 Introduction

This tutorial aims to explain how to write an efficient code for single-site Density Matrix Renormalization Group (DMRG) [8, 9] algorithm, to iteratively find the ground state and the ground state energy of a spin chain system. We will follow the Tensor-Network (TN) approach to DMRG, by representing our state in the form of Matrix Product State (MPS) and our operators (hamiltonians specifically) in the form of Matrix Product Operators (MPO).

The tutorial will follow the language of a well written review article on the subject titled "The density-matrix renormalization group in the age of matrix product states" [7]. Most of the material of this article has been burrowed from this article and it is strongly suggested for every one to go through this review article for a deeper and solid knowledge on DMRG and handling MPS based tensors in general. In addition to this there is a very helpful website on the tensor network,[2], which consists useful tutorials on handling tensors in general and also several tensor network based algorithms, DMRG included. This website goes farther than MPS based algorithms.

This tutorial intends to teach general tensor handling. The section 3 on contraction of tensors is useful in this regards as efficient tensor contraction forms the basis of every tensor network code. This article will be accompanied by a code, written in Julia [1]/Python programming language. This article has been written such that specific sections of the article has a direct correspondence with analogous sections in the code for an easy understanding of the code.

2 A quick recap on MPS and MPO

As you have seen during the course, Matrix Product States provide a smart way to efficiently represent low-entangled states in quantum many-body systems. Given a system of N spins, MPS are defined by

$$\psi_{\sigma_1 \ldots \sigma_L} = M^{[1]}_{\alpha_1}(\sigma_1) M^{[2]}_{\alpha_1 \alpha_2}(\sigma_2) \ldots M^{[N]}_{\alpha_{N-1}}(\sigma_N) \tag{1}$$

where we used Einstein convention on the "virtual" indices α (α_i runs from 1 to the "local bond-dimension" χ_i). This object can be graphically represented as in Figure 1. In principle any state $|\psi\rangle = \psi_{\sigma_1 \ldots \sigma_N} |\sigma_1 \ldots \sigma_N\rangle$ can be written as an MPS, but if we set $\chi_i \leq \chi, \forall i$ it is easy to realize that only states with bounded Entangled Entropy (i.e. $S_{vN}(A|B) \leq \log \chi$ for any bi-partition $A|B$ of the system) can be represented. If we do this, we can easily define a function returning a random MPS, i.e. a random set of three-legs tensors M (two virtual legs and one physical leg):

```
def initial_psi(N,chi,d):

    M_set = [0 for x in range(N)]
    M_set[0] = np.random.rand(1,d,chi)
    M_set[N-1] = np.random.rand(chi,d,1)

    for i in range(1,N-1):
        M_set[i] = np.random.rand(chi,d,chi)

    return M_set
```

Code Listing 1: random initialization of a non-normalized MPS with bond (physical) dimension χ (d).

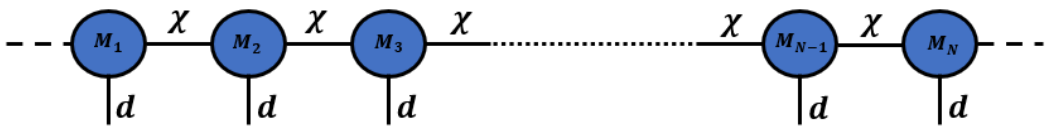

Figure 1: graphical representation of an MPS.

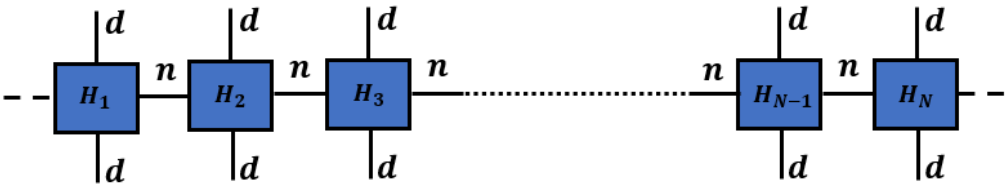

Figure 2: graphical representation of an MPO.

The number of parameters will grows polynomially (not exponentially) with the system size N.

Similarly, any operator O acting on our Hilbert space $\mathcal{H}$

$$\hat{O} = O_{(\sigma_1,\sigma_1')\dots(\sigma_N,\sigma_N')} \left|\sigma_1 \dots \sigma_N\right\rangle \left\langle \sigma_1' \dots \sigma_N'\right| \ . \tag{2}$$

can be decomposed as

$$O_{(\sigma_1,\sigma_1')\dots(\sigma_N,\sigma_N')} = W^{[1]}_{\alpha_1}(\sigma_1,\sigma_1') W^{[2]}_{\alpha_1\alpha_2}(\sigma_2,\sigma_2') \dots W^{[N]}_{\alpha_{N-1}}(\sigma_N,\sigma_N') \ . \tag{3}$$

where $W^{[i]}(\sigma_i,\sigma_i')$ are matrices/vectors (see Figure 2).

2.1 How to design your MPO

As you know, DMRG is nothing but an iterative (local) optimization of energy expectation value

$$\mathcal{E}\big(\left|\psi\right\rangle\big) = \left\langle\psi\right| H \left|\psi\right\rangle \tag{4}$$

with respect to the local tensors M. In order to write your DMRG code, it is necessary to write the hamiltonian you are interested in as an MPO. In the greater part of the cases, this is not a difficult task. Let us start by defining the following matrices/vectors of operators $\tilde{W}^{[i]}_{\alpha\alpha'} = W^{[i]}(\sigma_i,\sigma_i')_{\alpha\alpha'} \left|\sigma_i\right\rangle \left\langle\sigma_i'\right|$, so that

$$\hat{O} = \tilde{W}^{[1]} \dots \tilde{W}^{[N]} \ . \tag{5}$$

It is useful to think at the above equation as the action of a "finite state machine" operating on $\tilde{\chi}$ different (virtual) states

$$\boxed{1} \ , \boxed{2} \ \dots \ \boxed{\tilde{\chi}} .$$

For this purpose, let us define the vectors (of operators)

$$V^{[l]} = \tilde{W}^{[l]} \dots \tilde{W}^{[N]} \qquad l = 1, 2 \dots N \ ,$$

so that

$$V_{\alpha}^{[l-1]} = \tilde{W}_{\alpha\beta}^{[l-1]} V_{\beta}^{[l]} \ .$$

Now, we have to build properly our "transition matrices" $\tilde{W}^{[l]}$ in order to get the operator $\hat{O}$. Let us begin with a simple example. Suppose $\hat{O} = \Sigma^z$ is the total $z-$magnetization operator

$$\Sigma^z = \sum_{i=1}^{N} \sigma_i^z = \sum_{i=1}^{N} \mathbb{1}_1 \otimes ... \otimes \mathbb{1}_{i-1} \otimes \sigma_i^z \otimes \mathbb{1}_{i-1} \otimes ... \otimes \mathbb{1}_N \ .$$

Let us consider a finite state machine operating on two states (i.e. $\tilde{\chi} = 2$) and let us represent it as in Figure 5. The corresponding transition matrix is just:

$$\tilde{W}^{[l]} = \begin{pmatrix} \mathbb{1} & 0 \\ \sigma_l^z & \mathbb{1} \end{pmatrix}$$

with $l = 2, 3, ...N$. The vector $\tilde{W}^{[N]}$ set the starting state of the machine. It can be both $\boxed{1}$ (and in this case we must apply the identity operator $\mathbb{1}_N$) or $\boxed{2}$ (and in this case we must apply the σ_N^z). Therefore

$$\tilde{W}^{[N]} = \begin{pmatrix} \mathbb{1}_N \\ \sigma_N^z \end{pmatrix} \ .$$

It is easy to realize that, by applying the $\tilde{W}^{[l]}$ matrices to $\tilde{W}^{[N]}$ we will get

$$V^{[2]} = \begin{pmatrix} \mathbb{1}_2 \otimes \mathbb{1}_3 \otimes ... \otimes \mathbb{1}_N \\ \sum_{i=2}^{N} \mathbb{1}_2 \otimes ... \otimes \mathbb{1}_{i-1} \otimes \sigma_i^z \otimes \mathbb{1}_{i+1} \otimes ... \otimes \mathbb{1}_N \end{pmatrix} \ .$$

Now, it becomes obvious that to get the operator Σ^z we have to set

$$\tilde{W}^{[1]} = \begin{pmatrix} \sigma_1^z & \mathbb{1}_1 \end{pmatrix} \ .$$

Now let us consider a more complex case, i.e. the following Ising hamiltonian with exponentially decaying couplings

$$H = -\sum_{i=1}^{N} \sum_{j=1}^{i-1} J \lambda^{i-j} \sigma_i^z \sigma_j^z - \sum_{i=1}^{N} \boldsymbol{h} \cdot \boldsymbol{\sigma} \ . \tag{6}$$

The finite state machine plotted in Figure 6 can realize such operator. Indeed, the loop on the state $\boxed{2}$ allow us to get exponential decay. The corresponding transition matrix will be

$$\tilde{W}^{[l]} = \begin{pmatrix} \mathbb{1} & 0 & 0 \\ \sigma_l^z & \lambda \mathbb{1} & 0 \\ -\boldsymbol{h} \cdot \boldsymbol{\sigma}_l & -J\lambda\sigma_l^z & \mathbb{1} \end{pmatrix} \ ,$$

whereas the "initial state" will be

$$\tilde{W}^{[L]} = \begin{pmatrix} \mathbb{1}_N \\ \sigma_N^z \\ -\boldsymbol{h} \cdot \boldsymbol{\sigma}_N \end{pmatrix} \ .$$

It is to realize that we get the following V vector on the site $i = 2$:

$$V^{[2]} = \begin{pmatrix} \mathbb{1}_2 \otimes \mathbb{1}_3 \otimes ... \otimes \mathbb{1}_N \\ \sum_{i=2}^{N} \lambda^{i-2} \mathbb{1}_2 \otimes ... \otimes \mathbb{1}_{i-1} \otimes \sigma_i^z \otimes \mathbb{1}_{i+1} \otimes ... \otimes \mathbb{1}_N \\ \sum_{i=2}^{N} \sum_{j=2}^{i-1} \lambda^{i-j} \mathbb{1}_2 \otimes ... \otimes \mathbb{1}_{j-1} \otimes \sigma_j^z \otimes \mathbb{1}_{j+1} \otimes ... \otimes \mathbb{1}_{i-1} \otimes \sigma_i^z \otimes \mathbb{1}_{i+1} \otimes ... \otimes \mathbb{1}_N - \sum_{i=2}^{N} \boldsymbol{h} \cdot \boldsymbol{\sigma}_i \end{pmatrix} \ ,$$

and therefore we need to set

$$\tilde{W}^{[1]} = \begin{pmatrix} -\boldsymbol{h} \cdot \boldsymbol{\sigma}_1 & -J\lambda\sigma_1^z & \mathbb{1}_1 \end{pmatrix} \ .$$

These tricks can be used also to get the long-range Ising hamiltonian with power-law decaying couplings, i.e.

$$H = -\sum_{i=1}^{N} \sum_{j=1}^{i-1} J \frac{\sigma_i^z \sigma_j^z}{(i-j)^{\alpha}} - \sum_{i=1}^{N} \boldsymbol{h} \cdot \boldsymbol{\sigma} \ , \qquad \alpha > 0 \ .$$

In this case, it is useful to fit the power law $1/r^{\alpha}$ as a sum of decaying exponentials:

$$\frac{1}{r^{\alpha}} = \sum_{k=1}^{n} c_k e^{-r/\xi_k} = \sum_{k=1}^{n} c_k \lambda_k^r \qquad \lambda_k = e^{-1/\xi_k} \ .$$

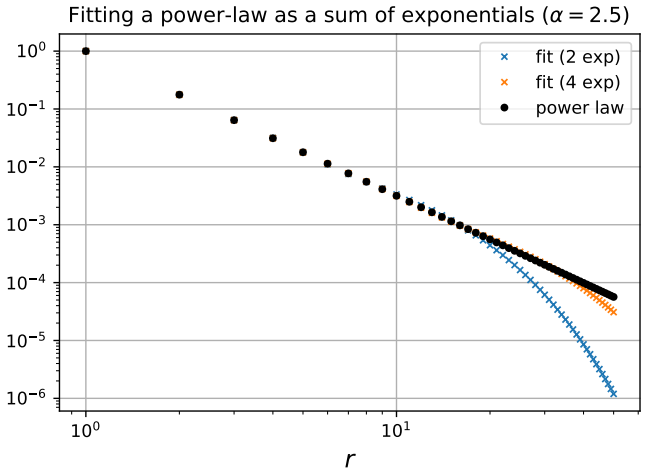

Figure 3: fitting a power-law $1/r^\alpha$ $(r = 1, 2, ...N)$ as a sum of decaying exponentials.

This task can be easily achieved by means of standard least squares routines and the result is usually very good (see Figure 3).

We can now use a finite state machine similar to the one represented in Figure 6, but with more intermediate states, each corresponding to a λ_k. In this case, we will get an MPO with bond dimension equal to $\chi = n+2$.

Let us observe that by setting $n = 1$, $J = J_0/\lambda$ and by taking the limit $\lambda \to 0$, the hamiltonian H in equation 6 becomes nothing but the usual short-range Ising hamiltonian. The code to define our MPO in this case will be simply:

```
def Hamiltonian_Ising(h,N):

    #basic spin matrices
    sX = 0.5*np.array([[0, 1], [1, 0]])
    sY = 0.5*np.array([[0, -1j], [1j, 0]])
    sZ = 0.5*np.array([[1, 0], [0,-1]])
    sI = np.array([[1, 0], [0, 1]])

    #building the local bulk MPO
    H = np.zeros([3,3,2,2])

    H[0,0,:,:] = sI; H[2,2,:,:] = sI; H[2,0,:,:] = -h*sX
    H[1,0,:,:] = sZ; H[2,1,:,:]= -sZ


    #building the boundary MPOs
    HL = np.zeros((1,3,2,2))
    HL[0,:,:,:] = H[2,:,:,:]
    HR = np.zeros((3,1,2,2))
    HR[:,0,:,:] = H[:,0,:,:]

    #put the hamiltonian in a list so that it can be iteteratively recuperate
    Ham = [0 for x in range(N)]

    Ham[0] = HL
    Ham[N-1] = HR
    for i in range(1,N-1):
        Ham[i] = H

    return Ham
```

Code Listing 2: function to generate the short range transverse Ising Hamiltonian as MPO.

One can also easily design an MPO representing a 2D hamiltonian. Let us consider a square lattice of dimension $N_x \times N_y$ $(N = N_x N_y)$. Let us order the sites of the grid following a one-dimensional "snaking" path connecting all the sites (see Figure 4). The sites will be labelled with a single index $i = 1, 2, ...N$. To design our finite state machine, let us observe that if we place a σ^z operator on the site i, then we have to either

1. place another σ^z on the next site following the path, except when i is a multiple of N_y;

2. or place $N_y - 1$ identity matrices $\mathbb{1}$ and then a σ^z on the site $i + N_y$.

Thus, we get the finite state machine plotted in Figure 7, with the *caveat* that the red edge must be "switched-off" when i is a multiple of N_y. Therefore, we get:

$$\tilde{W}^{[l]} = \begin{pmatrix} \mathbb{1} & 0 & 0 & 0 & 0 & 0 \\ \sigma_l^z & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbb{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbb{1} & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbb{1} & 0 & 0 \\ \boldsymbol{h} \cdot \boldsymbol{\sigma} & -J\sigma_l^z \left(1 - \delta_{0,l \bmod N_y}\right) & 0 & 0 & -J\sigma_l^z & \mathbb{1} \end{pmatrix}$$

This is an example of an inhomogeneous MPO, since the MPO matrices $\tilde{W}^{[l]}$ explicitly depend on the sites l (in contrast with the above cases). By using such MPO it is easy to implement also hamiltonian for inhomogeneous spin chains (for example, the Heisenberg model with site dependent couplings).

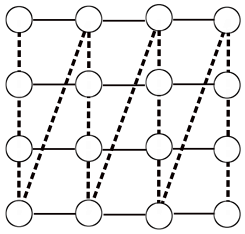

Figure 4: a 2D system described by means of a 1D "snaking" path ($N_x = N_y = 4$).

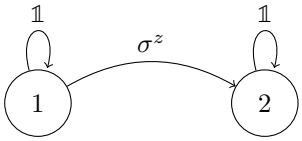

Figure 5: finite state machine representation of the Σ^z operator.

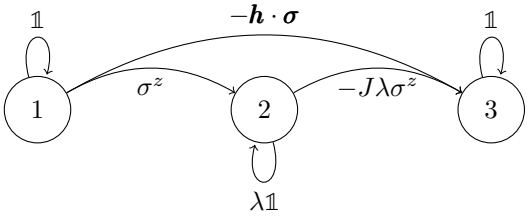

Figure 6: finite state machine representation of the long-range Ising hamiltonian.

3 Tensor contractions

Tensor contraction is a fundamental operation on tensors. A very important idea to keep in mind while working with tensors is the cost of contraction of tensors, as it directly determines the efficiency of your tensor network code. The cost of contraction of a set of tensors is the product of all the dimensions of the tensors involved divided by the dimensions of the indices contracted. Consider two tensors A and B with dimension $d_1 \times d_2 \times D_A$ and $d_1 \times d_2 \times D_B$ respectively, then the cost of contracting two indices of A and B as shown in figure 8 is

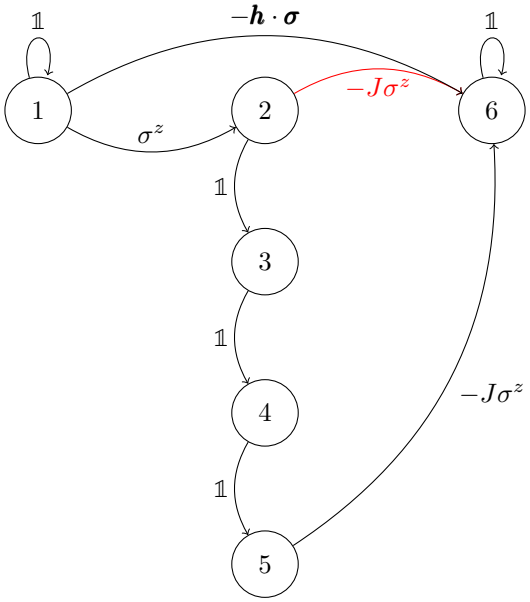

Figure 7: finite state machine representation of the 2D nearest-neighbours Ising hamiltonian ($N_y = 4$).

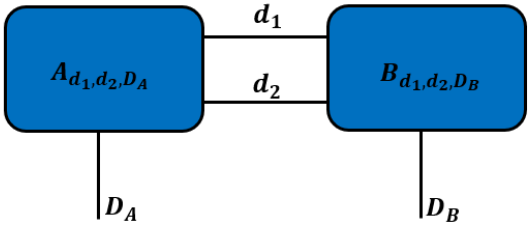

Figure 8: contracting tensors A and B.

$$\text{cost}(A \times B) = \frac{d_1^2 d_2^2 D_A D_B}{d_1 d_2} = d_1 d_2 D_A D_B \tag{7}$$

This can be easily understood by verifying it in a simple matrix multiplication. To show how different orders of tensor contraction lead to different contraction cost lets look at an example that will be useful later in DMRG (or DMRG like) algorithms. Consider the contraction of four tensors L, A, H, and $\tilde{A}$ with dimension $\chi^2 n$, $\chi^2 d$, $n^2 d^2$, and $\chi^2 d$ respectively as shown in figure 9. Also, for now consider $\chi > n$ and $\chi > d$.

We will contract these tensors with two different schemes. The first scheme, shown in figure 10 is a two step process.In the first step we contract the tensors $A, H,$ and, $\tilde{A}$ to create a block, the cost of this contraction is $\chi^4 n^2 d^2$.In the second step we contract the new block to the tensor L, the cost of this contraction is $\chi^4 n^2$. So, in this scheme the total contraction cost is $\chi^4 n^2 d^2 + \chi^4 n^2$.

The second scheme shown in figure 11 is a three step process. In the first step we contract tensor A with L with a contraction cost of $\chi^3 nd$. In the second step we contract tensor B with the new block with the contraction cost of $\chi^2 n^2 d^2$. Finally we complete the contraction by contracting the tensor $\tilde{A}$ to the block with a contraction cost of $\chi^3 nd$. So the total contraction cost is $\chi^3 nd + \chi^2 n^2 d^2 + \chi^3 nd$.

For $\chi \gg n$ and $\chi \gg d$ we see that the second scheme is order of magnitude efficient than the first scheme. Finding a cost effective contraction scheme is therefore the first step toward optimising your tensor network code. Several articles [6, 5] has been published on algorithms for finding the most efficient order for tensor contraction. The website,[2], has all the necessary basics of tensor contractions and a lot more on tensor

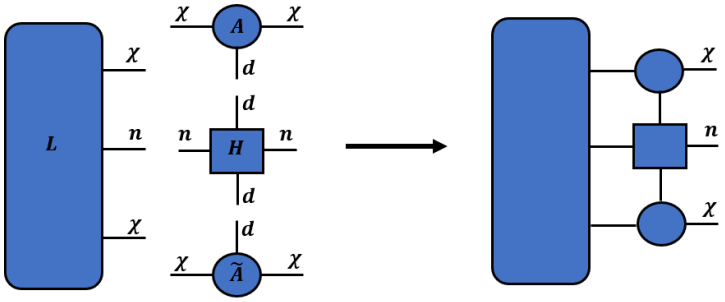

Figure 9: contracting tensors L, A, H, and $\tilde{A}$.

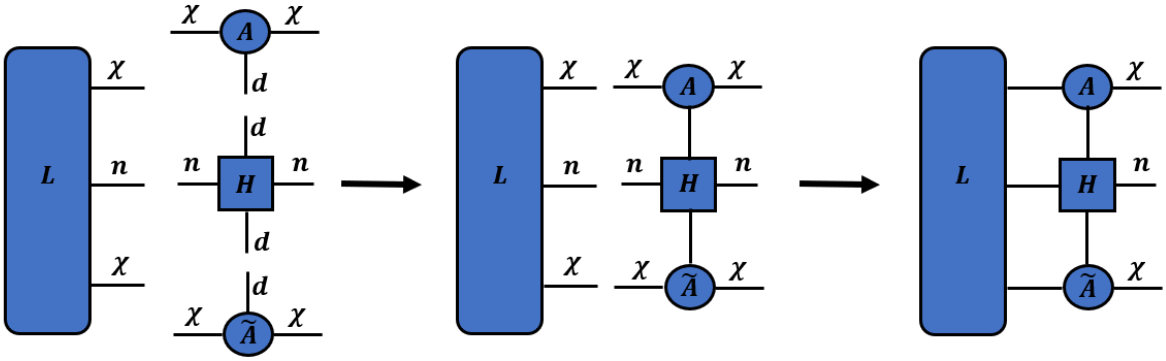

Figure 10: contracting tensors L, A, H, and $\tilde{A}$ with expensive scheme.

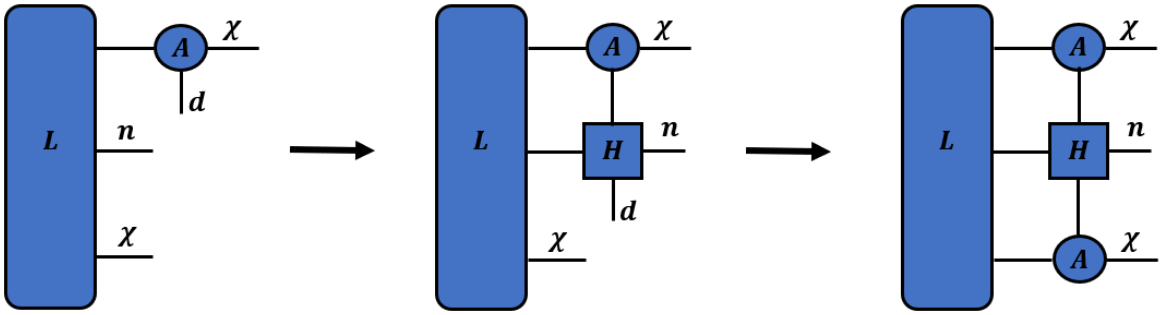

Figure 11: contracting tensors L, A, H, and $\tilde{A}$ with cheaper scheme.

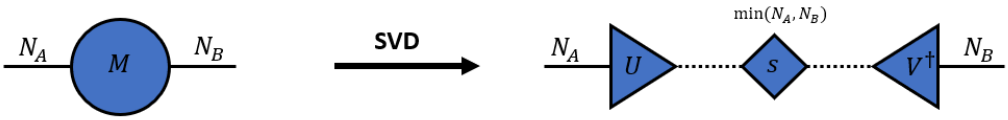

Figure 12: compact singular value decomposition of the matrix M.

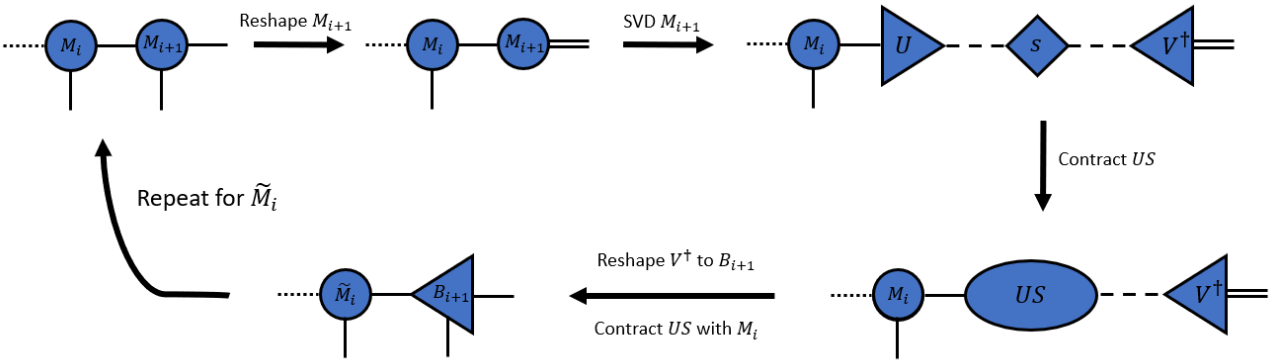

Figure 13: iteration for putting the non-normalized MPS into right canonical state.

networks for beginners.Also, a software called TensorTrace [3] allows you to find the optimal contraction order for any tensor network that you draw in its graphical user interface. For our need the efficient order for tensor contraction can usually be determined by a simple observation. The take home message should be that while contracting tensors we should always make sure that there are as less bigger indices free as possible.

4 Right and Left canonical MPS

```
def right_normalize(B,N):

    for i in range(N-1,0,-1):

        U,S,V = compact_svd(np.reshape(B[i],(B[i].shape[0],B[i].shape[1]*B[i].shape[2])))
        B[i] = np.reshape(V,(-1,B[i].shape[1],B[i].shape[2]))
        B[i-1] = np.tensordot(B[i-1],np.dot(U,np.diag(S)),axes = [2,0])/LA.norm(S)

    U,S,V = compact_svd(np.reshape(B[0],(B[0].shape[0],B[0].shape[1]*B[0].shape[2])))
    B[0] = np.reshape(V,(-1,B[0].shape[1],B[0].shape[2]))

    return B
```

Code Listing 3: function to put the non-normalized MPS set B into right canonical form.

```
def check_right_normalization(B,N):
    for i in range(N):
        summa = np.add(np.dot(B[i][:,0,:],B[i][:,0,:].T), np.dot(B[i][:,1,:],B[i][:,1,:].T))
        print("Test right unitarity: %s" % np.allclose(summa,np.eye(B[i].shape[0])))
```

Code Listing 4: function to check if the set B is in right canonical form..

For DMRG like algorithms it is necessary to bring the MPS to right (or left) canonical state. The MPS can be brought to right(or left) canonical state in more that one ways. Although the QR and LQ decompositions [10] are efficient for bringing the non-normalized MPS to right(or left) normalization, here we will discuss a more expensive but a standard approach following the Singular Value Decomposition (SVD) [11].

Consider an arbitrary matrix M of dimension $N_A \times N_B$. The compact singular value decomposition[1] of this matrix is shown in figure 12. The immediate properties of these three parts of the singular value decomposition are

[1] compact SVD is different from full SVD, while using libraries for SVD it should be remembered that the decomposition is a compact one.

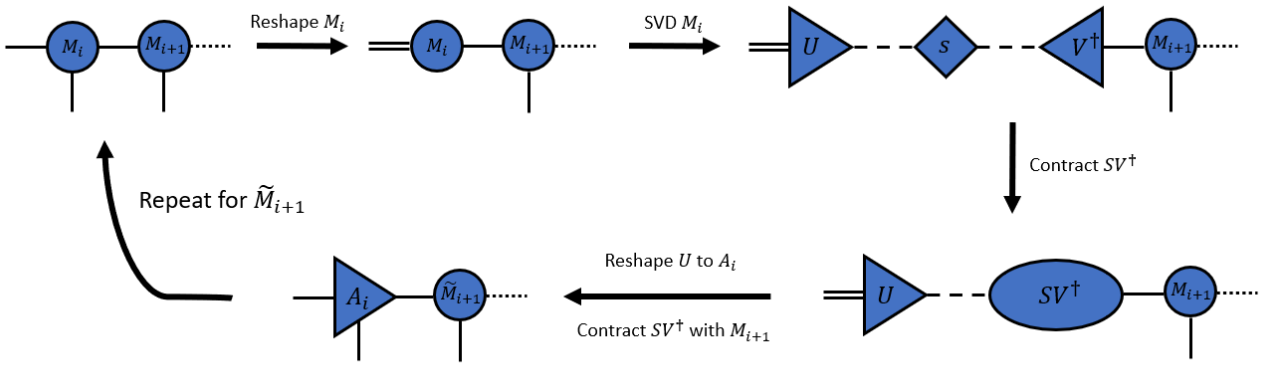

Figure 14: iteration for putting the non-normalized MPS into left canonical state.

- U is a matrix of dimension $N_A \times min(N_A, N_B)$ and is left singular, i.e. $U^\dagger U = \mathbb{1}$, if $N_A \leq N_B$ then we also have $UU^\dagger = \mathbb{1}$;
- S is a diagonal matrix with all non-negative entries, i.e. $S_{i,i} \geq 0$. These non-negative entries are known as singular values and the number of non-zero entries are known as the Schmidt rank;
- $V^\dagger$ is a matrix of dimension $min(N_A, N_B) \times N_B$ and is right singular, i.e. $V^\dagger V = \mathbb{1}$, if $N_A \geq N_B$ then we also have $VV^\dagger = \mathbb{1}$.

The left singular and right singular properties of the U and $V\dagger$ matrices will be used to bring the MPS to left and right canonical states respectively [2].

The process to bring a non-normalized MPS into right canonical form is demonstrated in figure 13. Here we consider only two consecutive tensors M_i and M_{i+1}, the dotted line on the left denotes other tensors down the chain.

- We start by reshaping the M_{i+1} tensor into a matrix by combining the free indices, one of which is a physical index σ and the other is bond index χ.
- The reshaped matrix is decomposed by SVD into U, S, and $V^\dagger$.
- The U and S matrices are contracted into US which is then contracted with M_i to form $\tilde{M}_i$, which is a non-normalized tensor.
- The $V^\dagger$ matrix, which is right normalized (i.e. $V^\dagger V = \mathbb{1}$) is reshaped back into a three legged tensor B_{i+1}.
- In the next iteration we undergo same procedure to $\tilde{M}_i$ and the iteration is repeated until the left most tensor is right normalized and the MPS becomes right canonical.

The process to bring a non-normalized MPS into left canonical form is similar, only this time we iterate from left to right and exploit the left-normalized property of U matrix to form the left normalized tensor A. Figure 14 demonstrates the self explanatory process.

Python code 3 implements the above mentioned process to put the non-normalized MPS set from code 1 into a right canonical MPS. A very similar algorithm can be written for left canonical form. Code 4 checks whether the given set is in right canonical form or not. A similar implementation can be done to check the left canonical form.

5 R and L tensors

Like right(or left) MPS, the R(or L) tensors are also necessary to begin the DMRG like algorithms. To build these tensors we will need right and left MPS. To begin with lets build the successive R tensors. Figure 15 demonstrates the building of successive R tensors. Like before the dotted lines on the left signifies other tensors down the chain.

[2] once again it is noted that QR and LQ decompositions are cheaper than SVD

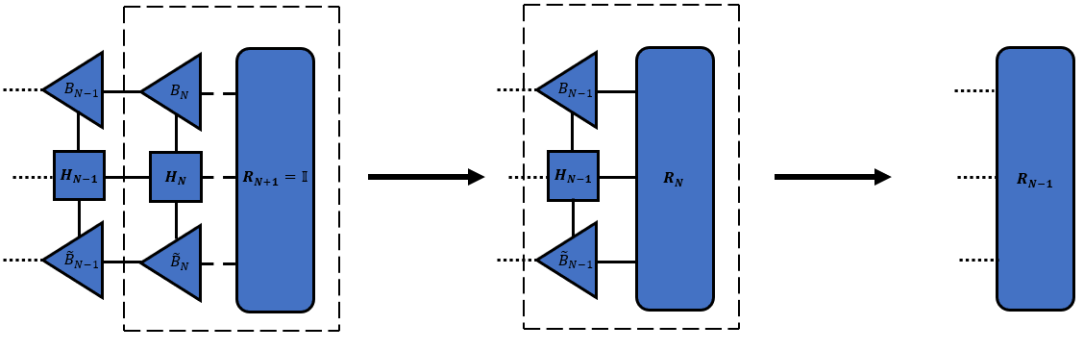

Figure 15: procedure to form successive R matrices.

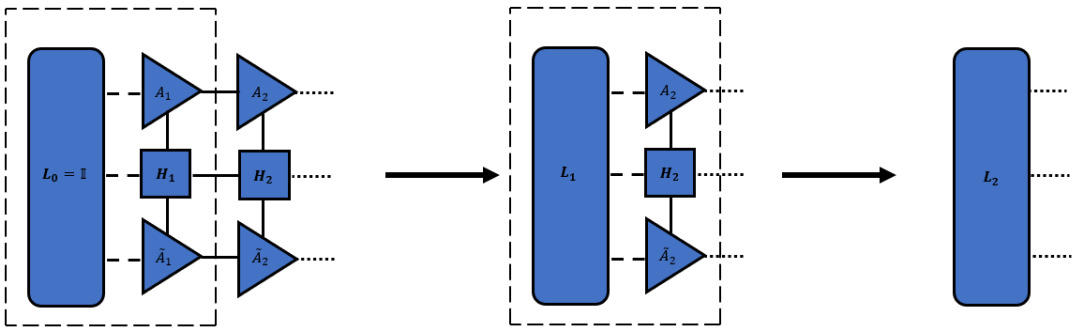

Figure 16: procedure to form successive L matrices.

- We start with R_{N+1} tensor which is nothing but a three dimensional analog of identity.
- This three legged tensor is contracted with the MPS tensors B_N and $\tilde{B}_N$ and MPO tensor H_N as demonstrated in figure 15 to form R_N. Refer to section 3 for the efficient construction of R_N.
- The above process is iterated to build successive R tensors $R_{N-1}, R_{N-2}, ...R_2$.
- We needn't build R_1 as it is not necessary for the DMRG algorithm.

```
def contract_right(R,W,B):

    R1 = np.tensordot(B.conj(),R,axes = [2,0])
    R1 = np.tensordot(R1,W,axes = [[1,2],[2,1]])
    R1 = np.tensordot(R1,B,axes = [[1,3],[2,1]])

    return R1

def contract_left(L,W,A):

    L1 = np.tensordot(A.conj(),L,axes = [0,0])
    L1 = np.tensordot(L1,W,axes = [[0,2],[2,0]])
    L1 = np.tensordot(L1,A,axes = [[1,3],[0,1]])

    return L1
```

Code Listing 5: unit functions in the creation of R and L tensors.

The process to build the L tensors is similar, only this time we start from the left most point of the chain. The procedure is demonstrated in figure 16 and is self explanatory.

The functions in the code 5 are the unit functions in the creation of R and L tensors. Iterating these functions over the lattice sites from left right edge and left edge builds the R and L tensors respectively.

6 Initialization

Initialization encodes all the necessary items to begin the DMRG sweeps, in that sense it is a very important step. It is conventional to start the DMRG sweep from left to right in which case we will need the following

- MPS with all but first site in right normalized form (it will be called mixed canonical although there aren't any left normalized tensors left of non-normalized tensor). Refer to section 4 to build this set.
- Set of R tensors R_{N+1},R_N,R_{N-1},...,R_2. Refer to section 5 to build this set.
- Set of L tensors with only the first element, L_0.

```
def Initialize(M_set,B,Ham,N):

    #get the list for R and boundary R
    R = [0 for x in range(N+1)]
    R[N] = np.zeros((1,1,1))
    R[N][0,:,:]=R[N][:,0,:]=R[N][:,:,0]=1

    #putting non-normalized MPS to right canonical form
    B = right_normalize(B,N)

    #generating R tensors
    for j in range(N-1,0,-1):
        R[j] = contract_right(R[j+1],Ham[j],B[j])

    # get the list for L and initialize boundary L
    L = [0 for x in range(N+1)]
    L[-1] = np.zeros((1,1,1))
    L[-1][0,:,:]=L[-1][:,0,:]=L[-1][:,:,0]=1

    #get the list for A
    A = [0 for x in range(N)]

    #get initial M
    M = M_set[0]

    return A,B,L,R,M
```

Code Listing 6: function to initialize before DMRG sweeps.

Code 6 initializes the system by building the necessary sets of tensors before running the DMRG sweeps.

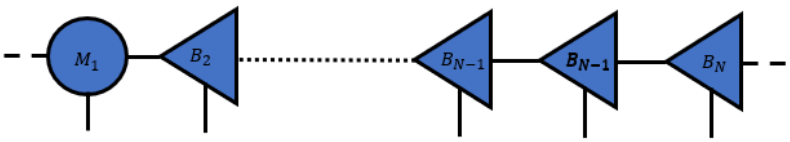

Figure 17: Mixed canonical MPS before the DMRG sweep.

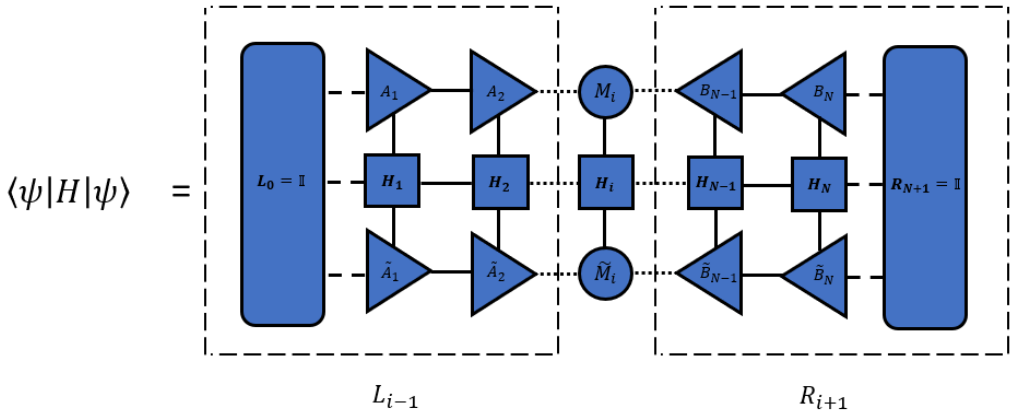

Figure 18: Pictorial representation of $\langle\psi|\, H\, |\psi\rangle$.

7 DMRG sweeps

Before explaining the DMRG sweeps step-wise, it is helpful to review the general idea behind DMRG in MPS framework. It is concerned with variational minimization of the energy functional

$$\mathcal{E}\big(\,|\psi\rangle\,\big) = \langle\psi|H|\psi\rangle\ . \tag{8}$$

Figure 18 shows the pictorial representation of the energy functional where the state is in the mixed canonical form with i^{th} lattice site tensor in non-normalized form and tensors left and right to it in left and right normalized state respectively. With $|\psi\rangle$ represented in MPS format DMRG achieves this minimization iteratively by fixing tensors corresponding to all but one lattice and local minimization is obtained with respect to this tensor. To do this a Lagrange multiplier is applied as

$$\mathcal{L}(|\psi\rangle\,,\lambda) = \langle\psi|\,H\,|\psi\rangle + \lambda(\langle\psi|\,|\psi\rangle - 1) \tag{9}$$

and this function is locally minimized with respect to local MPS tensors iteratively:

$$\frac{\partial\mathcal{L}(|\psi\rangle\,,\lambda)}{\partial M_i} = 0\,. \tag{10}$$

Equation 10 is equivalent to solving the local eigenvalue problem

$$H_i^{\text{eff}} M_i = \epsilon_0 M_i \tag{11}$$

where ϵ_0 is the minimum eigenvalue of H_i^{eff} matrix (until this point H_i^{eff} is still a six legged tensor and not a matrix, details on how to compute this eigenvalue equation will follow shortly). Figure 19 shows the pictorial representation of the of eigenvalue equation of H_i^{eff}. The idea is to iteratively perform this local

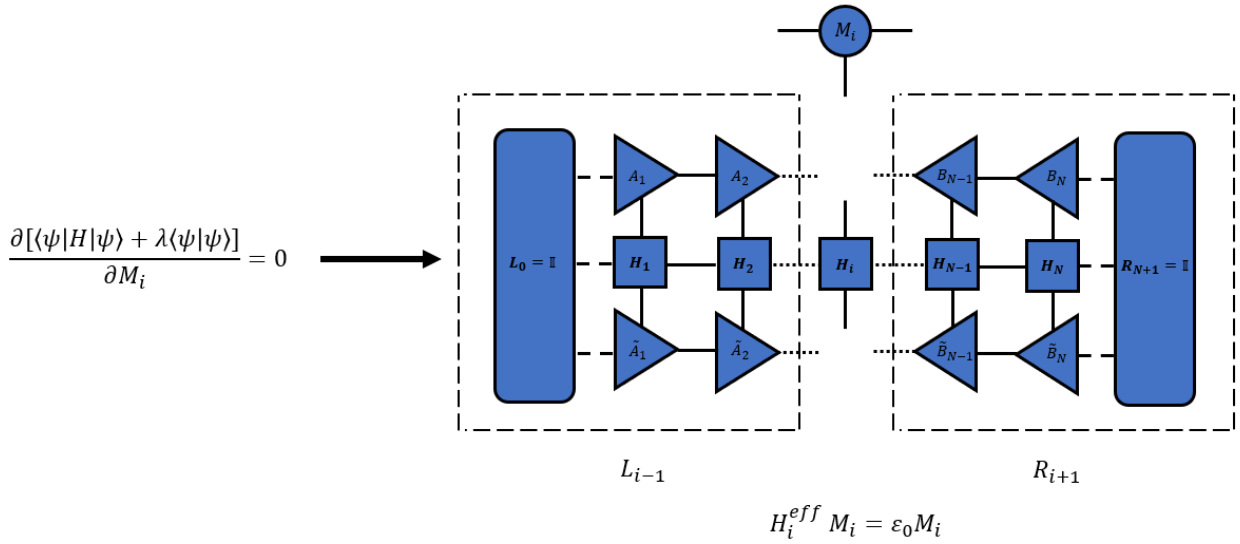

Figure 19: Pictorial representation of local minimization of H_i^{eff}.

minimization along the lattice sites known as DMRG sweep.It is conventional to start the sweep from left to right and then right to left until the local energy (the lowest eigenvalue of effective hamiltonian) converges. It is more comprehensible to explain the steps involved in DMRG sweeps explicitly rather than just mentioning the algorithm. Here we will explain the steps involved in following subsections.

7.1 Optimization at first site

We begin the sweep by optimizing the left most lattice site. For this we build the effective hamiltonian at 1^{st} site, H_1^{eff} as shown in figure 20 which is a six legged tensor. To perform the eigenvalue minimization we reshape it into a matrix by bundling together the upper three and lower three indices together. We also reshape the three legged tensor M_1^o into a vector.We then use an eigensolver that takes matrix H_1^{eff} and vector M_1^o as input and gives the lowest eigenvalue ϵ_0 and eigenvector M_1^f as output. This however is a brute force method as in real life calculations the matrix H_i^{eff} can be very large and thus be time and memory inefficient to build such a matrix. We should then resort to iterative solvers, here we have used Lanczos eigensolver based on Lanczos algorithm[4]. Lanczos eigensolver just needs the product $H_i^{eff} M_i$ as the input and not the full matrix. In MPS formalism this product can be done much more efficiently and hence the Lanczos eigensolver is a cheaper approach to do the local eigenvalue equation, figure 21 demonstrate this. The minimization gives the lowest eigenvalue ϵ_0 and eigenvector M_1^f.

Note: It should be noted that it is inefficient to obtain full convergence of the iterative Lanczos eigensolver locally. The idea is to obtain the global convergence.

7.2 Restoration of MPS

The matrix M_1^f is reshaped back to three legged tensor and left normalized tensor A_1 is formed out of it by SVD (or QR decomposition). The residuals $SV^\dagger$ is contracted with B_2 to form the non-normalized tensor M_2^o. This process is demonstrated in figure 22. Similarly the L_1 tensor build as shown in figure 23.

7.3 Sweeping through the loop

Following the same procedure as for first lattice site we minimize the effictive hamiltonian at second lattice site, restore the MPS, build L_2 tensor and iteratively move to the next lattice site until we reach the right most lattice site where our right sweep ends. This process is demonstrated in figure 24.

Once at the right edge we start the left DMRG sweep from right to left. This is technically similar to the right sweep, only this time we begin at site final site N and end at first site. Figures 25, 26, 27, and 28 demonstrates the self explanatory process.

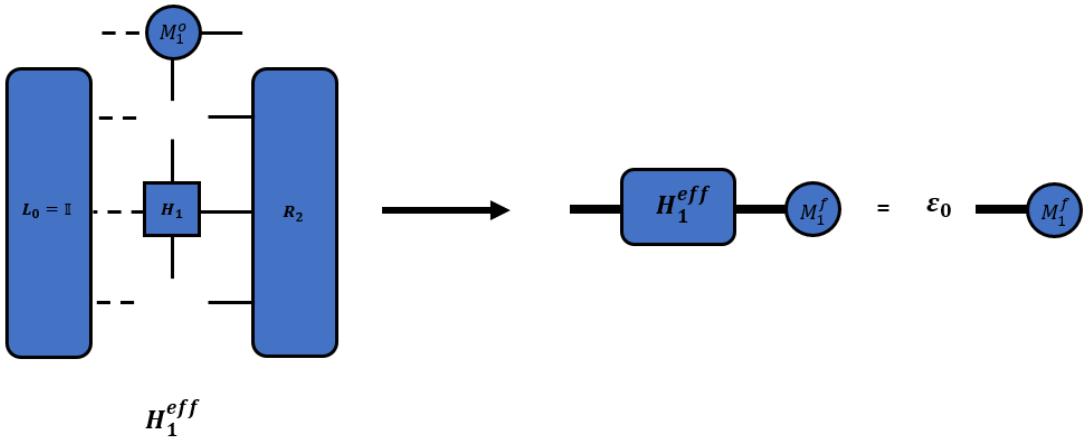

Figure 20: Pictorial representation of local minimization of at the first lattice site at the beginning of right DMRG sweep.

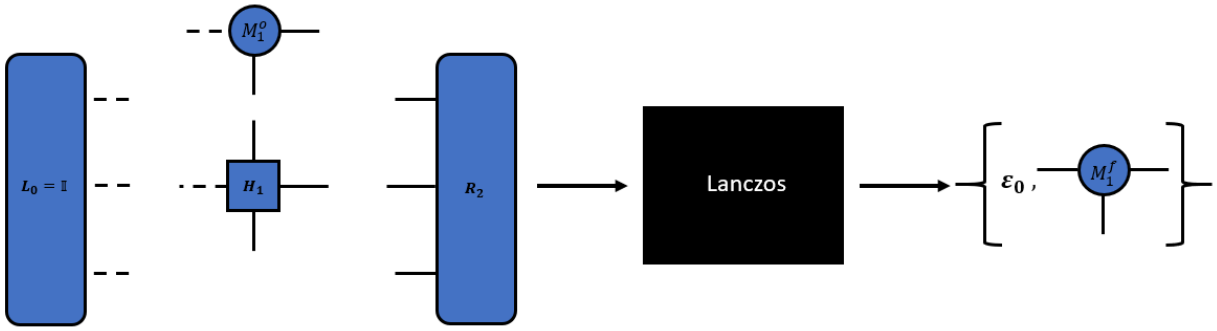

Figure 21: Pictorial representation of local minimization of at the first lattice site with Lanczos eigensolver.

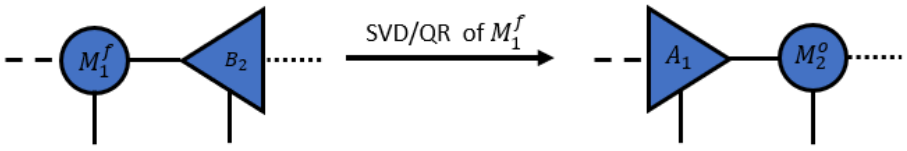

Figure 22: Restoration of MPS at first lattice site in right DMRG sweep.

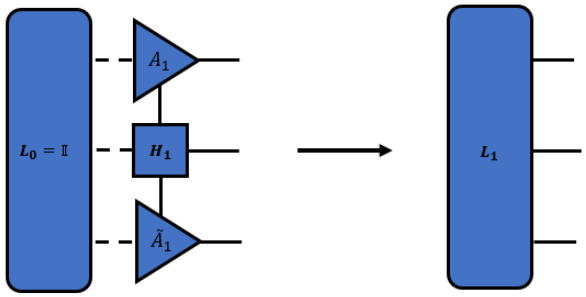

Figure 23: building L_1 tensor.

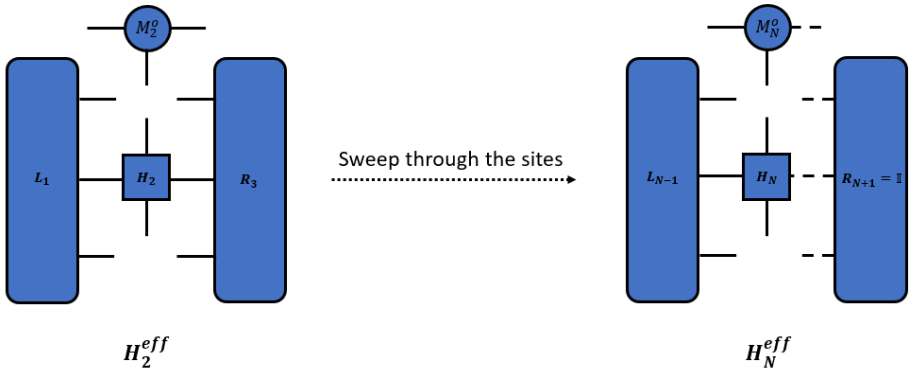

Figure 24: right iterative sweep for DMRG.

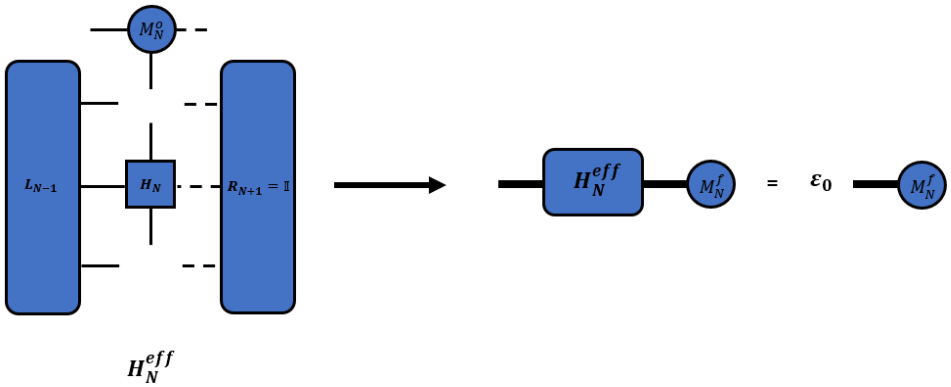

Figure 25: pictorial representation of local minimization of at the final lattice site at the beginning of left DMRG sweep.

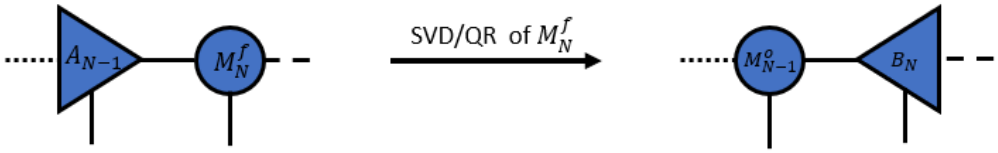

Figure 26: restoration of MPS at final lattice site in left DMRG sweep.

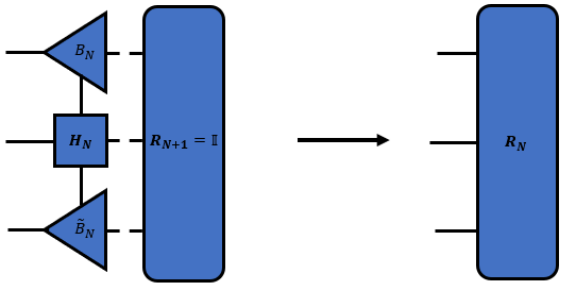

Figure 27: building R_N tensor.

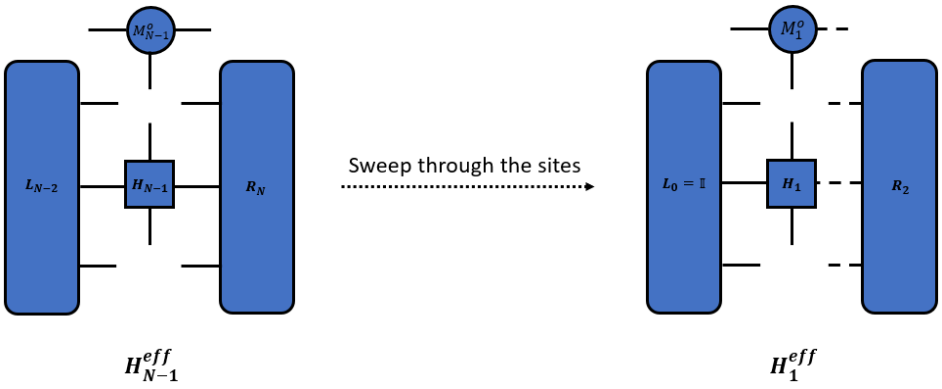

Figure 28: left iterative sweep for DMRG.

```python
def right_DMRG_sweep(L,R,Ham,M,A,B,chi,N,krydim,maxit):

    for i in range(N):

        shp_M = M.shape

        #reshape M into a vector
        psivec = np.reshape(M,(shp_M[0]*shp_M[1]*shp_M[2]))

        #local minimization at site i with Lanczos algorithm
        eig_vec, eig_val = EigenLanczOneSite(psivec,L[i-1],Ham[i],R[i+1],krydim,maxit)

        #reshape eig_vec into matrix for SVD
        vec = np.reshape(eig_vec,(shp_M[0]*shp_M[1],shp_M[2]))

        #SVD and truncation
        U,S,V = svd_truncate(vec,chi)

        #reshape U into A
        A[i] = np.reshape(U,(shp_M[0],shp_M[1],-1))

        #create L[i]
        L[i] = contract_left(L[i-1],Ham[i],A[i])

        if i != N-1:

            #create M tensor SV and B[i+1]
            SV = np.dot(np.diag(S),V)
            M = np.tensordot(SV,B[i+1],axes = [1,0])

            #delete R[i+1]
            R[i+1] = 0.0

    return A,B,L,R,M

def left_DMRG_sweep(L,R,Ham,M,A,B,chi,N,krydim,maxit):

    for i in range(N-1,-1,-1):

        shp_M = M.shape

        #reshape M into a vector
        psivec = np.reshape(M,(shp_M[0]*shp_M[1]*shp_M[2]))

        #local minimization at site i with Lanczos algorithm
        eig_vec, eig_val = EigenLanczOneSite(psivec,L[i-1],Ham[i],R[i+1],krydim,maxit)

        #reshape eig_vec into matrix for SVD
        vec = np.reshape(eig_vec,(shp_M[0],shp_M[1]*shp_M[2]))

        #SVD and truncation
        U,S,V = svd_truncate(vec,chi)

        #reshape U into A
        B[i] = np.reshape(V,(-1,shp_M[1],shp_M[2]))
        #print(B[i].shape)

        #create L[i]
        R[i] = contract_right(R[i+1],Ham[i],B[i])

        if i != 0:

            #create M tensor SV and B[i+1]
            US = np.dot(U,np.diag(S))
            M = np.tensordot(A[i-1],US,axes = [2,0])

            #delete L[i-1]
            L[i-1] = 0.0

    return A,B,L,R,M
```

Code Listing 7: Functions for right and left DMRG sweeps through the lattice sites.

Code 7 consists the implementation of the right and left DMRG sweeps. Iterating the functions over several runs will provide us the ground state and energy. These functions implements a local solver **EigenLanczOneSite** which finds the ground state and energy using iterative Lanczos method. Refer to code 8 for the Lanczos eigen-

solver.

```
def EigenLanczOneSite(psivec,L,W,R,krydim,maxit):

    if LA.norm(psivec) == 0:
        psivec = np.random.rand(len(psivec))

    psi = np.zeros([len(psivec),krydim+1])
    A = np.zeros([krydim,krydim])
    dval = 0

    for k in range(maxit):

        psi[:,0] = psivec/max(LA.norm(psivec),1e-16)

        for p in range(1,krydim+1):

            psi[:,p] = MpoToMpsOneSite(L,W,R,psi[:,p-1])

            for g in range(p-2,p):
                if g >= 0:
                    A[p-1,g] = np.dot(psi[:,p],psi[:,g])
                    A[g,p-1] = np.conj(A[p-1,g])

            for g in range(p):
                psi[:,p] = psi[:,p] - np.dot(psi[:,g],psi[:,p])*psi[:,g]
                psi[:,p] = psi[:,p] / max(LA.norm(psi[:,p]),1e-16)

        [dtemp,utemp] = LA.eigh(A)
        psivec = psi[:,range(0,krydim)] @ utemp[:,0]

    psivec = psivec/LA.norm(psivec)
    dval = dtemp[0]

    return psivec, dval
```

Code Listing 8: Lanczos based eigen solver.

8 Algorithm/pseudo-code

We can sum up everything in the following algorithms. It is more comprehensible to separate the algorithms into different procedures each of which can be efficiently written as a separate function while coding. Finally, it is to be understood that these are just pseudo-codes: their implementation on different programming languages might be different.

Algorithm 1 Initialize random non-normalized MPS with bond dimension χ and physical dimension d for system of N lattice sites

procedure INITIAL(N, χ, d)
  $M[1] = \mathbf{rand}(1, d, \chi)$ ▷ left boundary tensor of MPS
  $M[N] = \mathbf{rand}(\chi, d, 1)$ ▷ right boundary tensor of MPS
  for $i \in [2, N-1]$ **do**
    $M[i] = \mathbf{rand}(\chi, d, \chi)$
  return $\{M\}_{i=0}^{N}$ ▷ Returns non-normalized MPS

Algorithm 2 Bring the non-normalized MPS to right canonical form

procedure RIGHT NORMALIZE($\{M\}_{i=1}^{N}, N$)
  for $i \in [N-1, 1]$ **do**
    $M[i] \leftarrow U[i]S[i]B[i]$ ▷ using SVD, also can use LQ decomposition
    $M[i-1] \leftarrow M[i-1]U[i]S[i]$
  return $\{B\}_{i=1}^{N}$ ▷ Returns right-normalized MPS

Algorithm 3 Unit algorithm to build R and L tensors, refer to figures 15, 16

procedure RIGHT CONTRACTION($R[i+1], H[i], B[i]$)
  $R_{m_{i-1},h_{i-1},\tilde{m}_{i-1}}[i] \leftarrow \tilde{B}^{\tilde{\sigma}_i}_{\tilde{m}_i,\tilde{m}_{i-1}}[i] H_{h_i,h_{i-1},\sigma_i,\tilde{\sigma}_i}[i] B^{\sigma_i}_{m_i,m_{i-1}}[i] R_{m_i,h_i,\tilde{m}_i}[i+1]$ ▷ m_i is i^{th} bond index
  return $R[i]$
procedure LEFT CONTRACTION($L[i-1], H[i], A[i]$)
  $L_{m_i,h_i,\tilde{m}_i}[i] \leftarrow \tilde{A}^{\tilde{\sigma}_i}_{\tilde{m}_i,\tilde{m}_{i-1}}[i] H_{h_i,h_{i-1},\sigma_i,\tilde{\sigma}_i}[i] A^{\sigma_i}_{m_i,m_{i-1}}[i] L_{m_{i-1},h_{i-1},\tilde{m}_{i-1}}[i-1]$ ▷ m_i is i^{th} bond index
  return $L[i]$

Algorithm 4 Initialize before DMRG sweeps.

procedure INITIALIZE($\{M\}_{i=1}^N, \{H\}_{i=1}^N, N$)
  $L[0] \leftarrow \mathbb{1}$
  $R[N+1] \leftarrow \mathbb{1}$
  $\{B\}_{i=1}^N \leftarrow$ RIGHT NORMALIZE($\{M\}_{i=1}^N, N$) ▷ refer to algorithm 2
  for $i \in [N, 2]$ **do** ▷ we don't need $R[1]$
    $R[i] \leftarrow$ RIGHT CONTRACTION($R[i+1], H[i], B[i]$) ▷ refer to algorithm 3
  return $L[0], \{R\}_{i=2}^N, \{B\}_{i=1}^N$

Algorithm 5 Perform DMRG Sweeps.

procedure RIGHT DMRG SWEEP($M[1], \{B\}_{i=2}^N, \{H\}_{i=1}^N, L[0], \{R\}_{i=2}^N, N$)
  for $i \in [1, N]$ **do**
    $M[i] \leftarrow H^{eff}[i]M[i] = \epsilon_o[i]M[i]$, with $H^{eff}[i] = L[i-1]H[i]R[i]$ ▷ with Lanczos
    $A[i]S[i]V^\dagger[i] \leftarrow M[i]$ ▷ with SVD/QR
    $L[i] \leftarrow$ CONTRACT LEFT($L[i-1], H[i], A[i]$)
    if $i \neq N$ **then**
      $M[i+1] \leftarrow S[i]V^\dagger[i]B[i+1]$
      Delete $R[i+1]$
  return $\{L\}_{i=0}^{N-1}, \{A\}_{i=1}^{N-1}, M[N]$
procedure LEFT DMRG SWEEP($M[N], \{A\}_{i=1}^{N-1}, \{H\}_{i=1}^N, R[N+1], \{L\}_{i=1}^{N-1}, N$)
  for $i \in [N, 1]$ **do**
    $M[i] \leftarrow H^{eff}[i]M[i] = \epsilon_o[i]M[i]$, with $H^{eff}[i] = L[i-1]H[i]R[i]$ ▷ with Lanczos
    $U[i]S[i]B[i] \leftarrow M[i]$ ▷ with SVD/LQ
    $R[i] \leftarrow$ CONTRACT RIGHT($R[i+1], H[i], B[i]$)
    if $i \neq 1$ **then**
      $M[i-1] \leftarrow A[i-1]U[i]S[i]$
      Delete $L[i-1]$
  return $\{R\}_{i=2}^N, \{B\}_{i=2}^N, M[1]$

9 Full code in Python

In this section we will show our complete single site DMRG code, implemented in Python. Some of the functions were not available in the main text above. For a detailed open access code for DMRG (also TDVP and more) in julia language refer to the github page https://github.com/NishanRanabhat/TenMB

```
#call necessary libraries
import numpy as np
from numpy import linalg as LA
import scipy.linalg as la
from scipy.sparse.linalg import eigsh, eigs

#initialize a random non-normalized MPS
def initial_psi(N,chi,d):

    M_set = [0 for x in range(N)]
    M_set[0] = np.random.rand(1,d,chi)
    M_set[N-1] = np.random.rand(chi,d,1)

    for i in range(1,N-1):
        M_set[i] = np.random.rand(chi,d,chi)

    B = M_set.copy()

    return M_set,B

#gives the compact(or thin) SVD of a matrix
def compact_svd(mat):

    U,S,V = LA.svd(mat)

    if mat.shape[0] > mat.shape[1]:
        U = U[:,:mat.shape[1]]

    elif mat.shape[0] < mat.shape[1]:
        V = V[:mat.shape[0],:]

    return U,S,V

#performs SVD and truncates the results by a cutoff value chi
def svd_truncate(T,chi):

    U,S,V = compact_svd(T)

    if len(S) > chi:

        S = S[0:chi]
        U = U[:,0:chi]
        V = V[0:chi,:]

    S = S/LA.norm(S)

    return U,S,V

#puts non-normalized MPS in right canonical form
def right_normalize(B,N):

    for i in range(N-1,0,-1):

        U,S,V = compact_svd(np.reshape(B[i],(B[i].shape[0],B[i].shape[1]*B[i].shape[2])))
        B[i] = np.reshape(V,(-1,B[i].shape[1],B[i].shape[2]))
        B[i-1] = np.tensordot(B[i-1],np.dot(U,np.diag(S)),axes = [2,0])/LA.norm(S)

    U,S,V = compact_svd(np.reshape(B[0],(B[0].shape[0],B[0].shape[1]*B[0].shape[2])))
    B[0] = np.reshape(V,(-1,B[0].shape[1],B[0].shape[2]))

    return B

#nearest neighbor transverse field Ising Hamiltonian as the set of MPOs
def Hamiltonian_Ising(h,N):

    sX = 0.5*np.array([[0, 1], [1, 0]])
    sY = 0.5*np.array([[0, -1j], [1j, 0]])
    sZ = 0.5*np.array([[1, 0], [0,-1]])
    sI = np.array([[1, 0], [0, 1]])
```

```
71
72      #building the local bulk MPO
73      H = np.zeros([3,3,2,2])
74
75      H[0,0,:,:] = sI; H[2,2,:,:] = sI; H[2,0,:,:] = -h*sX
76      H[1,0,:,:] = sZ; H[2,1,:,:]= -sZ
77
78
79      #building the boundary MPOs
80
81      HL = np.zeros((1,3,2,2))
82      HL[0,:,:,:] = H[2,:,:,:]
83      HR = np.zeros((3,1,2,2))
84      HR[:,0,:,:] = H[:,0,:,:]
85
86      #put the hamiltonian in a list so that it can be iteteratively recuperate
87      Ham = [0 for x in range(N)]
88
89      Ham[0] = HL
90      Ham[N-1] = HR
91      for i in range(1,N-1):
92          Ham[i] = H
93
94      return Ham
95
96
97  #the following five functions are all the necessary tensor contraction routines in this code
98  def contract_right(R,W,B):
99
100     R1 = np.tensordot(B.conj(),R,axes = [2,0])
101     R1 = np.tensordot(R1,W,axes = [[1,2],[2,1]])
102     R1 = np.tensordot(R1,B,axes = [[1,3],[2,1]])
103
104     return R1
105
106 def contract_left(L,W,A):
107
108     L1 = np.tensordot(A.conj(),L,axes = [0,0])
109     L1 = np.tensordot(L1,W,axes = [[0,2],[2,0]])
110     L1 = np.tensordot(L1,A,axes = [[1,3],[0,1]])
111     return L1
112
113 def contract_left_noop(L,A):
114
115     L1 = np.tensordot(A.conj(),L,axes = [0,0])
116     L1 = np.tensordot(L1,A,axes = [[0,2],[1,0]])
117     return L1
118
119 def contract_left_nonmpo(L,W,A):
120
121     L1 = np.tensordot(A.conj(),L,axes = [0,0])
122     L1 = np.tensordot(L1,W,axes = [0,0])
123     L1 = np.tensordot(L1,A,axes = [[1,2],[0,1]])
124
125     return L1
126
127 def MpoToMpsOneSite(L,W,R,M):
128
129     M = np.reshape(M,(L.shape[2],W.shape[3],R.shape[2]))
130
131     fin = np.tensordot(W,R,axes = [1,1])
132     fin = np.tensordot(M,fin,axes = [[1,2],[1,4]])
133     fin = np.tensordot(L,fin, axes = [[1,2],[1,0]])
134
135     fin = np.reshape(fin,(L.shape[0]*W.shape[2]*R.shape[0]))
136
137     return fin
138
139 #Initialize the system before DMRG sweeps
140 def Initialize(M_set,B,Ham,N):
141
142     #get the list for R and boundary R
143     R = [0 for x in range(N+1)]
144     R[N] = np.zeros((1,1,1))
145     R[N][0,:,:]=R[N][:,0,:]=R[N][:,:,0]=1
146
```

```
    #putting non-normalized MPS to right canonical form
    B = right_normalize(B,N)

    #generating R tensors
    for j in range(N-1,0,-1):
        R[j] = contract_right(R[j+1],Ham[j],B[j])

    # get the list for L and initialize boundary L
    L = [0 for x in range(N+1)]
    L[-1] = np.zeros((1,1,1))
    L[-1][0,:,:]=L[-1][:,0,:]=L[-1][:,:,0]=1

    #get the list for A
    A = [0 for x in range(N)]

    #get initial M
    M = M_set[0]

    return A,B,L,R,M

#Local Lanczos eigensolver
def EigenLanczOneSite(psivec,L,W,R,krydim,maxit):

    if LA.norm(psivec) == 0:
        psivec = np.random.rand(len(psivec))

    psi = np.zeros([len(psivec),krydim+1])
    A = np.zeros([krydim,krydim])
    dval = 0

    for k in range(maxit):

        psi[:,0] = psivec/max(LA.norm(psivec),1e-16)

        for p in range(1,krydim+1):

            psi[:,p] = MpoToMpsOneSite(L,W,R,psi[:,p-1])

            for g in range(p-2,p):
                if g >= 0:
                    A[p-1,g] = np.dot(psi[:,p],psi[:,g])
                    A[g,p-1] = np.conj(A[p-1,g])

            for g in range(p):
                psi[:,p] = psi[:,p] - np.dot(psi[:,g],psi[:,p])*psi[:,g]
                psi[:,p] = psi[:,p] / max(LA.norm(psi[:,p]),1e-16)

        [dtemp,utemp] = LA.eigh(A)
        psivec = psi[:,range(0,krydim)] @ utemp[:,0]

    psivec = psivec/LA.norm(psivec)
    dval = dtemp[0]

    return psivec, dval

#The following two functions implements right and left DMRG sweeps
def right_DMRG_sweep(L,R,Ham,M,A,B,chi,N,krydim,maxit):

    for i in range(N):

        shp_M = M.shape

        #reshape M into a vector
        psivec = np.reshape(M,(shp_M[0]*shp_M[1]*shp_M[2]))

        #local minimization at site i with Lanczos algorithm
        eig_vec, eig_val = EigenLanczOneSite(psivec,L[i-1],Ham[i],R[i+1],krydim,maxit)

        #reshape eig_vec into matrix for SVD
        vec = np.reshape(eig_vec,(shp_M[0]*shp_M[1],shp_M[2]))

        #SVD and truncation
        U,S,V = svd_truncate(vec,chi)

        #reshape U into A
        A[i] = np.reshape(U,(shp_M[0],shp_M[1],-1))
```

```
223
224         #create L[i]
225         L[i] = contract_left(L[i-1],Ham[i],A[i])
226
227         if i != N-1:
228
229             #create M tensor SV and B[i+1]
230             SV = np.dot(np.diag(S),V)
231             M = np.tensordot(SV,B[i+1],axes = [1,0])
232
233             #delete R[i+1]
234             R[i+1] = 0.0
235
236     return A,B,L,R,M
237
238 def left_DMRG_sweep(L,R,Ham,M,A,B,chi,N,krydim,maxit):
239
240     for i in range(N-1,-1,-1):
241
242         shp_M = M.shape
243
244         #reshape M into a vector
245         psivec = np.reshape(M,(shp_M[0]*shp_M[1]*shp_M[2]))
246
247         #local minimization at site i with Lanczos algorithm
248         eig_vec, eig_val = EigenLanczOneSite(psivec,L[i-1],Ham[i],R[i+1],krydim,maxit)
249
250
251         #reshape eig_vec into matrix for SVD
252         vec = np.reshape(eig_vec,(shp_M[0],shp_M[1]*shp_M[2]))
253
254         #SVD and truncation
255         U,S,V = svd_truncate(vec,chi)
256
257         #reshape U into A
258         B[i] = np.reshape(V,(-1,shp_M[1],shp_M[2]))
259         #print(B[i].shape)
260
261         #create L[i]
262         R[i] = contract_right(R[i+1],Ham[i],B[i])
263
264         if i != 0:
265
266             #create M tensor SV and B[i+1]
267             US = np.dot(U,np.diag(S))
268             M = np.tensordot(A[i-1],US,axes = [2,0])
269
270             #delete R[i+1]
271             L[i-1] = 0.0
272
273     return A,B,L,R,M
274
275
276 #This part initializes and executes the DMRG sweeps
277 N = 100
278 chi = 40
279 d = 2
280 num_DMRG = 4
281 h = 1.0
282 krydim = 4
283 maxit = 6
284
285 M_set,B = initial_psi(N,chi,d)
286
287 Ham = Hamiltonian_Ising(h,N)
288
289 A,B,L,R,M = Initialize(M_set,B,Ham,N)
290
291 for i in range(num_DMRG):
292     A,B,L,R,M = right_DMRG_sweep(L,R,Ham,M,A,B,chi,N,krydim,maxit)
293     A,B,L,R,M = left_DMRG_sweep(L,R,Ham,M,A,B,chi,N,krydim,maxit)
```

Code Listing 9: full single site DMRG code in Python.

References

[1] Jeff Bezanson et al. "Julia: A fresh approach to numerical computing". In: *SIAM review* 59.1 (2017), pp. 65–98. URL: https://doi.org/10.1137/141000671.

[2] Glen Evenbly. *Tensors.net*. URL: https://www.tensors.net/.

[3] Glen Evenbly. *TensorTrace: an application to contract tensor networks*. 2019. arXiv: 1911.02558 [quant-ph].

[4] Cornelius Lanczos. "An Iteration Method for the Solution of the Eigenvalue I Problem. of Linear Differential and Integral Operators". In: *Journal of Research of the National Bureau of Standards* 45 (4 1950), pp. 255–282. URL: https://doi.org/10.6028%2Fjres.045.026.

[5] Ling Liang et al. "Fast Search of the Optimal Contraction Sequence in Tensor Networks". In: *IEEE Journal of Selected Topics in Signal Processing* 15.3 (2021), pp. 574–586. DOI: 10.1109/JSTSP.2021.3051231.

[6] Robert N. C. Pfeifer, Jutho Haegeman, and Frank Verstraete. "Faster identification of optimal contraction sequences for tensor networks". In: *Phys. Rev. E* 90 (3 Sept. 2014), p. 033315. DOI: 10.1103/PhysRevE.90.033315. URL: https://link.aps.org/doi/10.1103/PhysRevE.90.033315.

[7] Ulrich Schollwöck. "The density-matrix renormalization group in the age of matrix product states". In: *Annals of Physics* 326.1 (2011). January 2011 Special Issue, pp. 96–192. ISSN: 0003-4916. DOI: https://doi.org/10.1016/j.aop.2010.09.012. URL: https://www.sciencedirect.com/science/article/pii/S0003491610001752.

[8] Steven R. White. "Density matrix formulation for quantum renormalization groups". In: *Phys. Rev. Lett.* 69 (19 Nov. 1992), pp. 2863–2866. DOI: 10.1103/PhysRevLett.69.2863. URL: https://link.aps.org/doi/10.1103/PhysRevLett.69.2863.

[9] Steven R. White. "Density-matrix algorithms for quantum renormalization groups". In: *Phys. Rev. B* 48 (14 Oct. 1993), pp. 10345–10356. DOI: 10.1103/PhysRevB.48.10345. URL: https://link.aps.org/doi/10.1103/PhysRevB.48.10345.

[10] Wikipedia contributors. *QR decomposition*. [Online; accessed 23-April-2021]. 2021. URL: https://en.wikipedia.org/wiki/QR_decomposition.

[11] Wikipedia contributors. *Singular Value Decomposition*. [Online; accessed 7-May-2021]. 2021. URL: https://en.wikipedia.org/wiki/Singular_value_decomposition.