Assignment 2: Entanglement entropy and compression of quantum states

Instructor: Lesik Motrunich
TA: Liam O'Brien

Ph 121C: Computational Physics Lab, Spring 2024
California Institute of Technology
Due: 4pm Tuesday, April 30, 2024

1 Introduction to many-body entanglement

In this assignment we will study a property of quantum many-body states called entanglement. It may not seem as directly physically motivated as the measures we encountered in the previous assignment, like decay of correlations or measurements of the order parameter, but it turns out that a great deal of the physics of a quantum system is closely related to its entanglement structure. In particular, one quantity called the entanglement entropy is a crucial indicator of a system's low-energy behavior, in addition to determining the degree of difficulty of simulating it on classical architecture like your computer. That the same quantity is important in both situations is not coincidental: in fact, it is typically the case that at low energies nature - that is, physically-motivated Hamiltonians - tends to favor precisely those quantum states which are more amenable to classical simulation. This fortunate state of affairs is rigorously described by the "area law," an upper bound on entanglement (proved in one dimension) which has attracted much recent interest from the quantum information and condensed matter physics communities. This principle is fundamental to our efforts in this course to overcome the limitations of exact diagonalization.

We will learn how to exploit nature's tendency to favor "simple" quantum wavefunctions by developing a compression scheme based on our physical understanding. The most evident advantage here is storage: recall that

a generic eigenstate of a quantum system of even moderate size quickly becomes too big to save on a computer. What simplifications can be made to reduce the exponential scaling of the number of coefficients, while still leaving the interesting properties of a state intact? This question is nontrivial: for example, if you have an ED wavefunction from the previous assignment saved to disk as binary file you will likely find it cannot be compressed very much by tar or other generic tools, and moreover the quantum state would be opaque to us in this form. We will instead make use of the entanglement structure of the physical system to write an ansatz for one-dimensional quantum ground states called matrix product states (MPS). This ansatz is of paramount importance to a modern understanding of quantum systems in one dimension and has a strong physical motivation in the connection between entanglement and locality.

2 Singular value decomposition

2.1 Matrix approximation

Suppose you have a large collection of m data points of some type, each containing a measurement of a large number n of attributes. In order to study the data, one can think of each data point as a vector in an n-dimensional space, where each attribute is assigned to a basis vector. Then the measured value of data point $x_i, i \in \{1, \ldots, m\}$, for attribute $\hat{e}_j, j \in \{1, \ldots, n\}$, is the j-th entry in the vector x_i, viewed as a row vector. It is interesting, both practically and in order to understand the processes behind the data, to ask whether the points (with average subtracted) display clustering in certain directions, described by linear relationships between the basis states. If so, the subspace spanned by the dominant directions can provide a good low-dimensional projection of the data despite having reduced dimension. This is the goal of "principal component analysis," (PCA) in which one uses a low-dimensional ellipsoid to approximate data. While we are not specifically interested in PCA, it is similar to the wavefunction compression we study in spirit and some technical details; however in our case the low-dimensional representation can be exact.

To determine such principal vectors, combine the data points into an $m \times n$ data matrix M of rank $r \leq \min(m, n)$. Determining a subspace of dimension $k < r$ which optimally describes the data then corresponds to

finding a rank- k matrix $\tilde{M}$ which minimizes the distance $\|\tilde{M} - M\|_F$. (Here $\|A\|_F = \sqrt{\mathrm{tr}\,(A^\dagger A)}$ is the so-called Frobenius norm of matrix A.) This is accomplished through a general method of matrix factorization known as the singular value decomposition (SVD):

$$M = USV^\dagger; \quad U^\dagger U = V^\dagger V = \mathbb{1}, \quad S \text{ diagonal.} \tag{1}$$

The SVD can be thought of as a generalization of the spectral theorem to nonsquare matrices, where the left singular vectors $\boldsymbol{u}_\alpha, \alpha = \{1, \ldots, r\}$, stored in $U = \left[\begin{array}{cccc} \boldsymbol{u}_1 & \boldsymbol{u}_2 & \cdots & \boldsymbol{u}_r \end{array} \right]$, need not be the same (or even live in the same space) as the right singular vectors $\boldsymbol{v}_\alpha, \alpha = \{1, \ldots, r\}$, stored in $V = \left[\begin{array}{cccc} \boldsymbol{v}_1 & \boldsymbol{v}_2 & \cdots & \boldsymbol{v}_r \end{array} \right]$. The singular values end up-typically in descending order-on the diagonal of S, with the number of nonzero singular values being the rank of M.

Importantly, the spectrum of singular values may always be chosen to be real and non-negative, and in this form it is uniquely defined for any matrix. The existence of the SVD is one of the most important theorems in linear algebra. You can read more about the SVD - how it is computed and its additional properties and applications - in the handouts posted on the course website. Any linear algebra package has SVD functions requiring computational effort comparable to that of diagonalization. The important thing to know for this assignment is that such an SVD can always be found. The above formula (1) holds for a general complex matrix M; if M is real-valued, then the SVD matrices U and V are real-valued. Separate optimized routines exist for finding the SVD of complex matrices and of real matrices.

We can now state the solution to the problem of approximating a matrix M. It is not too difficult

to show that in order to compute the matrix $\tilde{M}$ of rank $k < r$ which optimally approximates M, it suffices to find the SVD of the matrix M as above and then simply restrict to the largest k singular values. This reduces the dimensions of the U, V matrices from $(m, r), (n, r)$ to (m, k), (n, k), respectively. That is, we construct $\tilde{S} = \mathrm{diag}\,(\lambda_1, \lambda_2, \ldots, \lambda_k), \tilde{U} = [\boldsymbol{u}_1 \boldsymbol{u}_2 \cdots \boldsymbol{u}_k]$, and $\tilde{V} = \left[\begin{array}{cccc} \boldsymbol{v}_1 & \boldsymbol{v}_2 & \cdots & \boldsymbol{v}_k \end{array} \right]$ to produce

$$\tilde{M} = \tilde{U}\tilde{S}\tilde{V}^\dagger \tag{2}$$

and one can show that the corresponding Frobenius measure of error becomes

$$\|\tilde{M} - M\|_F = \sqrt{\sum_{\alpha=k+1}^{r} \lambda_\alpha^2} \tag{3}$$

2.2 Image compression

A nice demonstration of how the SVD works is to compress an image using matrix approximation. A digital image can be thought of as a collection of two-dimensional matrices of numerical values, one for each color channel. A grayscale image is equivalently just a real-valued data matrix. In order to gain some intuition of how the truncated SVD compresses a matrix and the ways in which errors manifest, one can factorize an image, reduce the rank as described above, and multiply again to obtain the approximate image. Despite the fact that truncation based on the SVD is optimal according to the Frobenius norm, it turns out that matrices of reduced rank are typically not ideal for compression of visual data, as you can observe in Fig. 1.

There is not always much intuitive information that can be learned from inspection of the U and V matrices in the SVD. However, the singular value spectrum on the diagonal of S provides a useful measure of the degree of complexity, or information content. By this we mean that an image that is highly structured, or more simply composed, will show a steeper decline in singular values. To use examples from art, a Mondrian or Rothko painting is somewhat more orderly than a Pollock; this distinction may be quantified (perhaps dubiously) using the singular value spectrum from the SVD. We will formalize this concept in the following sections.

Figure 1: Feynman playing the bongos, and his rank-1 approximation. The approximate Feynman is clearly the product of one profile $\boldsymbol{u}_1$ in the vertical direction, and one, $\boldsymbol{v}_1^\top$, in the horizontal. The Frobenius error of the approximation is only 36.5%.

3 Bipartite entanglement

3.1 Schmidt decomposition

The SVD has a direct implication that is crucial to our efforts to efficiently study quantum wavefunctions. Consider a state $|\psi\rangle$ of a system with open boundary conditions which can be partitioned into a subsystem $\mathcal{A}$ and its complement $\mathcal{A}^c$, with Hilbert space $\mathcal{H} = \mathcal{H}_\mathcal{A} \otimes \mathcal{H}_{\mathcal{A}^c}$. Let $\mathcal{H}_\mathcal{A}$ be spanned by $|a_i\rangle, i = 1, \ldots, \dim(\mathcal{H}_\mathcal{A})$, and $\mathcal{H}_{\mathcal{A}^c}$ by $|b_j\rangle, j = 1, \ldots, \dim(\mathcal{H}_{\mathcal{A}^c})$. Naively, one would expect that
the simplest general way to write the wavefunction is

$$|\psi\rangle = \sum_{i,j} c_{ij} |a_i\rangle \otimes |b_j\rangle \tag{4}$$

Written in this way, $|\psi\rangle$ can be associated with a bilinear operator through

a formal procedure of matricization $\mathfrak{M}$ that simply amounts to $|b_j\rangle \to \langle b_j|$, which is evidently a bijection:

$$|\psi\rangle = \sum_{i,j} c_{ij} |a_i\rangle |b_j\rangle \quad \xrightarrow{\mathfrak{M}} \quad M_{|\psi\rangle} = \sum_{i,j} c_{ij} |a_i\rangle \langle b_j| \tag{5}$$

‘ ‘Now we have $M_{|\psi\rangle}$, adim $(\mathcal{H}_{\mathcal{A}}) \times$dim $(\mathcal{H}_{\mathcal{A}^c})$ matrix, and the SVD obtains a basis $|u_\alpha\rangle, \alpha = 1, \ldots, r$, for $\mathcal{H}_{\mathcal{A}}$ and corresponding basis $|v_\alpha\rangle$ for $\mathcal{H}_{\mathcal{A}^c}$, where $r = \min(\dim(\mathcal{H}_{\mathcal{A}}), \dim(\mathcal{H}_{\mathcal{A}^c}))$, permitting

$$M_{|\psi\rangle} = \sum_{\alpha=1}^{r} \lambda_\alpha |u_\alpha\rangle \left\langle v_\alpha \left| \xrightarrow{\mathfrak{M}^{-1}} \right| \psi \right\rangle = \sum_{\alpha=1}^{r} \lambda_\alpha |u_\alpha\rangle \otimes |v_\alpha\rangle \tag{6}$$

The singular values λ_α of $M_{|\psi\rangle}$ are called the Schmidt values of the state $|\psi\rangle$, and the orthonormal basis states are left or right S chmidt vectors.

We can approximate the state $|\psi\rangle$ by reducing the Schmidt rank to k across the cut, throwing away the smallest $r - k$ Schmidt vectors. However, as this is a quantum wavefunction, we must be careful to maintain normalization. Initially, $\langle \psi \mid \psi \rangle = \sum_{\alpha,\alpha'} \lambda_\alpha \lambda_{\alpha'} \langle u_{\alpha'} \mid u_\alpha \rangle \langle v_{\alpha'} \mid v_\alpha \rangle = \sum_\alpha \lambda_\alpha^2 = 1$, and as the norm of the approximate state is

$$\sqrt{\langle \tilde{\psi} \mid \tilde{\psi} \rangle} = \sqrt{\sum_{\alpha=1}^{k} \lambda_\alpha^2} = \sqrt{1 - \sum_{\alpha=k+1}^{r} \lambda_\alpha^2} \tag{7}$$

the state is normalized by rescaling all remaining Schmidt values by this amount.

3.2 Entanglement entropy

The general form (6) of a bipartite wavefunction reveals a great deal of information about the relationship between the two subsystems. If and only if $r = 1$ (that is, the only nonzero Schmidt value is $\lambda_1 = 1$), then $|\psi\rangle$ is a product state between $\mathcal{A}$ and $\mathcal{A}^c$, and is unentangled across the boundary. If $r > 1$ there is some degree of entanglement, which we can quantify in a natural way. Consider the following Schmidt decomposition parameterizing a manifold of states with $r = 2$: $|\psi(t)\rangle = \sqrt{1-t}\,|u_1\rangle\,|v_1\rangle + \sqrt{t}\,|u_2\rangle\,|v_2\rangle$, with $t \in [0, 1/2]$. If $t = \epsilon \ll 1$, the subsystems are nearly uncorrelated, and an observable of the following form almost factorizes:

$$\langle \psi(\epsilon) | O_{\mathcal{A}} \otimes O_{\mathcal{A}^c} | \psi(\epsilon) \rangle = \langle u_1 | O_{\mathcal{A}} | u_1 \rangle \langle v_1 | O_{\mathcal{A}^c} | v_1 \rangle + \mathcal{O}(\epsilon). \tag{8}$$

This can be understood visually by referencing Fig. 1, in which the approximate image clearly factorizes into the vertical and horizontal directions. A small perturbation would cause only a small change. On the other hand, for $t = 1/2$ the state is an equally weighted superposition of terms, and is as far from factorizable as is possible since one cannot designate either term as dominant. Accordingly, the state is said to be maximally entangled for $t = 1/2$.

We can make the preceding discussion more precise using an idea from information theory.

Specifically, recalling that $\sum_\alpha \lambda_\alpha^2 = 1$, we can regard $p(\alpha) = \lambda_\alpha^2$ as a discrete probability distribution. Then the notion described above corresponds to the Shannon entropy of the distribution:

$$H[p] = -\sum_\alpha p(\alpha) \log p(\alpha) \Longrightarrow S = -\sum_\alpha \lambda_\alpha^2 \log \lambda_\alpha^2 \tag{9}$$

The quantity S is called entanglement entropy, and is well-defined for a state $|\psi\rangle$ and specified bipartition, because the Schmidt spectrum is unique.

In the example above, $S(|\psi(0)\rangle) = 0$ indicates a product state and $S(|\psi(1/2)\rangle) = \log 2$ a maximally entangled state (under the constraint $r = 2$). These results generalize naturally. Remember that for a given bipartition $(\mathcal{A}, \mathcal{A}^c)$ of a system, the Schmidt rank is $r = \min(\dim(\mathcal{H}_{\mathcal{A}}), \dim(\mathcal{H}_{\mathcal{A}^c}))$; then product states have $S = 0$, and for maximally entangled states $S = \log r$. These values bound the entanglement entropy of all possible quantum states under this bipartition.

3.3 Scaling of entanglement entropy

So far the discussion of entanglement entropy has been a bit abstract, but this quantity is in fact closely related to the physics of a many-body quantum system. To make this connection, we consider $S(\ell)$, where $\ell = |\mathcal{A}| \leq |\mathcal{A}^c|$ (without loss of generality; S is invariant under $\mathcal{A} \leftrightarrow \mathcal{A}^c$), and $\mathcal{A}$ is contiguous. The scaling of $S(\ell)$ with ℓ in ground states depends strongly on the nature of the Hamiltonian, and in fact reveals some of its fundamental properties. The proofs of these results - those that are known - are difficult, and we will argue from simpler physical grounds.

The naive bound on entanglement entropy depends on the dimension of $\mathcal{H}_\mathcal{A}$. From the previous section, a maximally entangled state has

$$S(\ell) = \log r = \log \dim \left(\mathcal{H}_\mathcal{A}\right) = \log \dim \left(\left(\mathbb{C}^d\right)^{\otimes \ell}\right) = \ell \log d \tag{10}$$

That is, $S \sim \ell$. This is referred to as volume law entanglement scaling, which is saturated by typical states in the many-body Hilbert space. This includes highly-excited eigenstates away from the band edges of local Hamiltonians, as these generically do not avoid typicality.

However, sub-volume law entanglement scaling applies to ground states of local Hamiltonians with a finite excitation gap Δ in the thermodynamic limit. This is a consequence of a finite correlation length $\xi \sim 1/\Delta$ which controls the exponential decay of correlations between spatially separated observables. Because only degrees of freedom in $\mathcal{A}$ that are close (relative to ξ) to the boundary $\partial\mathcal{A}$ can be correlated with the complement, those degrees of freedom sufficiently far away contribute no entanglement under this bipartition. Therefore we expect the entanglement entropy to scale not as $\ell = |\mathcal{A}|$ but instead with the size of the boundary, $|\partial\mathcal{A}|$. In one dimension, $\partial\mathcal{A}$ is simply the endpoints of $\mathcal{A}$; thus $S(\ell) \sim \kappa(\xi)$, independent of ℓ for large $\ell > \xi$. This result is known as the area law of entanglement, and was proved by Matt Hastings in 2007. It formalizes the notion that ground states occupy a specialized and extremely small region of Hilbert space.

More exotic entanglement scaling is exhibited by ground states of Hamiltonians tuned to critical points. Here there is no energy gap, and accordingly the correlation length diverges. There is no length scale above which the system can be considered trivial, and at long distances the system displays scale invariance. A scale-invariant theory in one dimension is also conformally invariant, and is described by the technology of conformal field theory (CFT), in which correlation functions decay according to power laws and distant lattice sites do indeed contribute to entanglement. The scaling of entanglement entropy in a CFT was calculated by Calabrese and Cardy in 2004, who found a relatively mild logarithmic violation of the area law, given in the thermodynamic limit by $S(\ell) \sim \frac{c}{3} \log \ell$ to leading order. In the finite periodic case the logarithmic violation remains but is corrected to use the chord length: $S(\ell) \sim \frac{c}{3} \log \left(\frac{L}{\pi} \sin \frac{\pi \ell}{L}\right)$, demonstrating the entanglement entropy's long-range knowledge of the system.

In contrast to the gapped case where κ depends on ξ, the coefficient for critical entanglement scaling is universal: c is the "central charge," an

important characterization of the CFT describing

the critical point. Central charges are known for many CFTs; in particular, $c = 1/2$ for free fermions and $c = 1$ for free bosons. Note that these emergent descriptions do not necessarily
mirror the lattice theory: for example, the Heisenberg model of spins-which maps to microscopic fermions - is described at low energies by a free boson CFT.

As a final remark, one might question the statement of a necessarily finite correlation length in gapped systems, based on the asymptotically finite value of $\left\langle \sigma_1^z \sigma_{1+r}^z \right\rangle$ in the ordered phase of the quantum Ising model. Previously it was claimed that the true ground state is $\sim |\uparrow\uparrow\cdots\uparrow\rangle + |\downarrow\downarrow\cdots\downarrow\rangle$, which respects the Ising symmetry. But this state, which has Schmidt rank $r = 2$ across all cuts, is clearly long-range entangled and in the thermodynamic limit is a macroscopic superposition state, which is not observed in nature. What happens is that as $L \to \infty$, fluctuations spontaneously choose one of the degenerate symmetry-breaking product states and the connected correlation function $C_{\text{conn}}^{zz}(r) = \left\langle \sigma_1^z \sigma_{1+r}^z \right\rangle - \langle \sigma_1^z \rangle \left\langle \sigma_{1+r}^z \right\rangle$ which detects quantum correlations indeed decays exponentially. However, there is a signature of the twofold ground state degeneracy in the entanglement entropy, in the form of a subleading term: a constant $\log 2$ in the ferromagnet phase.

4 Matrix product states

4.1 Compressing an ED wavefunction into an MPS

We will utilize the SVD and the physical principles of entanglement scaling to write quantum states in an extremely compact form known as a matrix product state (MPS). This will eventually allow us to extend our simulations to much larger system sizes. The MPS is an ansatz suitable for wavefunctions in one dimension which follow an area law; although this is true for vanishingly few states, it does apply to the ground states (and low-energy eigenstates) of local Hamiltonians.

Following Sec. 3.2, we may consider one-dimensional systems with open boundary conditions to be bipartite, with a boundary consisting of a single bond between two neighboring lattice sites. Performing the Schmidt decomposition exposes the entanglement across this bond, and according to the area law the Schmidt rank r is much less than its theoretical maximum and

is in fact bounded by some constant χ. An MPS is obtained by repeating the bipartition between every pair of neighboring sites, relying each time on the strong bound on the number of Schmidt values. Initially the wavefunction can be written

$$|\psi\rangle = \sum_{\sigma_1,\ldots,\sigma_L} C_{\sigma_1,\ldots,\sigma_L} |\sigma_1,\ldots,\sigma_L\rangle \tag{11}$$

where the tensor of coefficients C has L indices for the spin degrees of freedom $\sigma_i, i = 1,\ldots,L$. We partition the system into regions $\mathcal{A} = \{\sigma_1\}$ and $\mathcal{A}^c = \{\sigma_2,\ldots,\sigma_L\}$, and apply the matricization of Sec. 3.1 to $|\psi\rangle$, using parentheses to indicate grouping of the indices of C. Performing an SVD,

$$C_{\sigma_1,(\sigma_2,\ldots,\sigma_L)} = \sum_{\alpha} U_{\sigma_1,\alpha} S_{\alpha,\alpha} \left(V^\dagger\right)_{\alpha,(\sigma_2,\ldots,\sigma_L)} \tag{12}$$

where $\dim(\alpha) \leq \chi$. We rename $U_{\sigma_1,\alpha} = (A^1)^{\sigma_1}_{\alpha}$ and multiply the diagonal matrix into $V^\dagger$ to obtain $W_{\alpha,(\sigma_2,\ldots,\sigma_L)} \equiv S_{\alpha,\alpha} \left(V^\dagger\right)_{\alpha,(\sigma_2,\ldots,\sigma_L)}$. Applying inverse matricization, the state is rewritten

$$M_{|\psi\rangle} = \sum_{\sigma_1,\ldots,\sigma_L} \sum_{\alpha} \left(A^1\right)^{\sigma_1}_{\alpha} W_{\alpha,(\sigma_2,\ldots,\sigma_L)} |\sigma_1\rangle \langle\sigma_2,\ldots,\sigma_L| \tag{13}$$

$$\xrightarrow{\mathfrak{M}^{-1}} |\psi\rangle = \sum_{\sigma_1,\ldots,\sigma_L} \sum_{\alpha} \left(A^1\right)^{\sigma_1}_{\alpha} W_{\alpha,\sigma_2,\ldots,\sigma_L} |\sigma_1,\ldots,\sigma_L\rangle \tag{14}$$

Now simply iterate the process, regrouping the spin indices on W to expose σ_2, which is grouped with the Schmidt index from the previous cut, applying $\mathfrak{M}$, and performing another SVD:

$$W_{\alpha,\sigma_2,\ldots,\sigma_L} \xrightarrow{\mathfrak{M}} W_{(\alpha,\sigma_2),(\sigma_3,\ldots,\sigma_L)} = \sum_{\beta} U_{(\alpha,\sigma_2),\beta} S_{\beta,\beta} \left(V^\dagger\right)_{\beta,(\sigma_3,\ldots,\sigma_L)} \tag{15}$$

Again relabel $U_{(\alpha,\sigma_2),\beta} = (A^2)^{\sigma_2}_{\alpha,\beta}$ and $W_{\beta,(\sigma_3,\ldots,\sigma_L)} \equiv S_{\beta,\beta} \left(V^\dagger\right)_{\beta,(\sigma_3,\ldots,\sigma_L)}$ and apply $\mathfrak{M}^{-1}$. At the next step, you will reshape (matricize) this new $W : W_{\beta,(\sigma_3,\sigma_4,\ldots,\sigma_L)} \to W_{(\beta,\sigma_3),(\sigma_4,\ldots,\sigma_L)}$ and SVD this, and so on. At each step, we can use the singular values to truncate the bond dimension by keeping only χ leading singular values. After $L-1$ steps we eventually obtain the following form of the state:

$$|\psi\rangle = \sum_{\sigma_1,\ldots,\sigma_L} \sum_{\alpha,\beta,\ldots,\omega} \left(A^1\right)^{\sigma_1}_{\alpha} \left(A^2\right)^{\sigma_2}_{\alpha,\beta} \cdots \left(A^L\right)^{\sigma_L}_{\omega} |\sigma_1,\ldots,\sigma_L\rangle \tag{16}$$

This is the MPS representation of $|\psi\rangle$. At the last step $\left(A^L\right)^{\sigma_L}_{\omega} \equiv S_{\omega,\omega} \left(V^\dagger\right)_{\omega,\sigma_L}$. Counting the dimension of each index shows that the MPS form of $|\psi\rangle$ uses at most $2\chi^2(L-2)+4\chi$ coefficients to represent the state exactly, an exponential reduction compared to (11).

4.2 Performing calculations with MPS

In this section we draw heavily from Ch. 4 of [1. The benefit of MPS is not only its concise description of certain quantum states; it also turns out to be a useful platform for many interesting physical calculations like overlaps, matrix elements, and correlations. These operations are conveniently described by the graphical calculus of tensor networks, a formalism for performing summations over many indices, as in (16). For example, computing the overlap between states $\langle\phi \mid \psi\rangle$, where $|\phi\rangle$ has B-tensors, one finds

$$\langle\phi \mid \psi\rangle = \sum_{\sigma_1,\ldots,\sigma_L} \sum_{\alpha,\beta,\ldots,\omega} \sum_{\alpha',\beta',\ldots,\omega'} \left(B^{L\dagger}\right)^{\sigma_L}_{\omega'} \cdots \left(B^{2\dagger}\right)^{\sigma_2}_{\beta',\alpha'} \left(B^{1\dagger}\right)^{\sigma_1}_{\alpha'} \left(A^1\right)^{\sigma_1}_{\alpha} \left(A^2\right)^{\sigma_2}_{\alpha,\beta} \cdots \left(A^L\right)^{\sigma_L}_{\omega} \tag{17}$$

Instead of continuing in this way, we draw diagrams with each tensor represented by a filled shape and attached legs corresponding to each tensor index. A scalar has no legs, a vector one leg, a matrix two legs, and so on. Summation is indicated between two tensors by connecting the legs representing the index to be summed over. The process of performing sums over indices in a tensor network diagram is called contraction. Thus, Fig. 2 reproduces (16): summation over virtual indices arising from the SVD by contracting the appropriate legs on adjacent tensors. The imposition of physical spin configurations is implicitly understood from the uncontracted physical legs.

Figure 2: Figures taken from [1]. By convention when writing MPS, physical indices σ_i point vertically and virtual indices (here, $a_1, a_2, \ldots$) connect

horizontally. The lefthand diagram shows the tensors A^1, A^ℓ, A^L, with no operation performed, and the righthand diagram replicates (16).

Now we return to the problem of the overlap (17). Because $\langle \phi \mid \psi \rangle$ is a scalar quantity the diagram we draw can have no uncontracted indices, including physical legs. In order to indicate Hermitian conjugation we flip the tensors vertically, so the physical indices of $\langle \phi |$ point downward. Now the form in Fig. 3 should suggest itself; the contracted vertical legs are the spin indices σ_i, and
the contracted top and bottom horizontal legs are the virtual indices $\alpha', \beta', \ldots, \omega'$ on $B^\dagger$-tensors and $\alpha, \beta, \ldots, \omega$ on A-tensors. In the case that $\langle \phi | = \langle \psi |$, the tensor network in Fig. 3 computes the squared norm $\langle \psi \mid \psi \rangle$, which can be used for normalization in other computations.

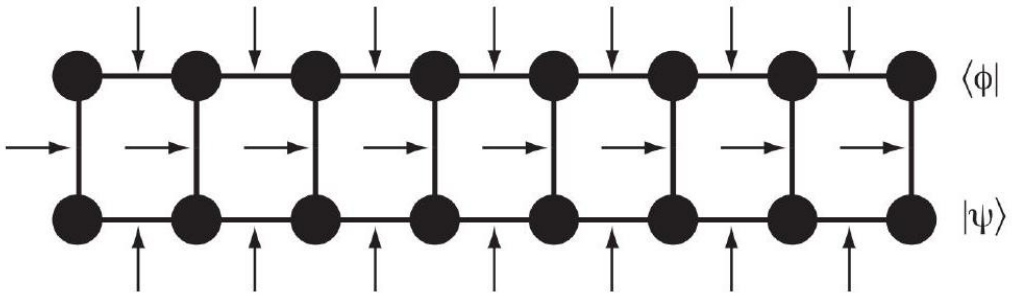

Figure 3: Figure taken from [1]. This is the tensor network for the overlap (17), with arrows pointing to all of the indices $\{\sigma_1, \ldots, \sigma_L\}, \{\alpha, \beta, \ldots, \omega\}, \{\alpha', \beta', \ldots, \omega'\}$ which are summed over in the contraction. The top row of tensors are the $(B^j)^\dagger$ in (17), and the bottom the A^j.

A generalization of the tensor network for overlaps allows us to easily compute matrix elements of local observables. Recall that some local observable $O(j)$ acting on site j may be written $O_j = \sum_{\sigma_j, \sigma_j'} O^{\sigma_j', \sigma_j} \left|\sigma_j'\right\rangle \langle \sigma_j|$, where σ_j' is the same degree of freedom as σ_j but distinguished by belonging to the adjoint state. In the tensor network, we need only insert the matrix for O_j into the contraction at the appropriate location, now performing a double sum over σ_j and σ_j' since we do not find the factor $\delta^{\sigma_j}_{\sigma_j'}$ that was used to obtain (17). This generalizes to products of local observables, as shown in Fig. 4 which computes the matrix element $\langle \phi \left| O_j O_k \right| \psi \rangle$. Because the Hamiltonian is a sum of local terms, by adding the results of multiple contracted tensor networks we can compute the matrix element $\langle \phi | H | \psi \rangle$.

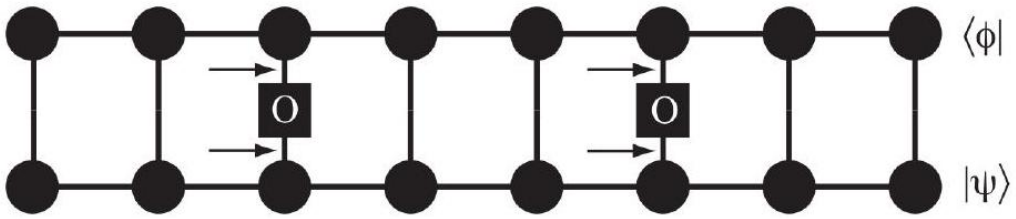

Figure 4: Figure taken from [1. Each of the lines connecting the spin indices σ_j can be thought of as a delta function $\delta^{\sigma_j}_{\sigma'_j}$; in this sense, the network in Fig. 3 computes the matrix element of the identity $\mathbb{I} = \mathbb{I}_1 \otimes \cdots \otimes \mathbb{I}_L$. In this diagram we compute a matrix element of the more general observable $O_j O_k$.

4.3 Practical considerations in tensor network contraction

The outcome of a contraction is independent of the order in which the sums are performed, but for practical reasons some patterns are clearly preferred. For example, by contracting virtual indices first we again obtain tensors with 2^L coefficients, which eliminates the benefits of using MPS. In general, optimal contraction of tensor networks is known to be an NP-complete problem. However for MPS the task is simpler, and we can define a general procedure that is nearly optimal. By

contracting the tensors vertically at a site j-that is, $(A^j)^{\sigma_j}$, $(B^j)^{\sigma'_j}$, and O_j if applicable - we obtain a 4-tensor of dimension χ^4. This is not too large, so we can do so at each site, then starting from the left contract the virtual indices (α, α'), then (β, β'), and so on. Because the cost of each vertical contraction does not depend on the system size and the number of horizontal contractions
scales as L, this "vertical-first" pattern scales linearly with system size. This is an efficient enough strategy to use on computers, and is clearly far superior to the exponential scaling of ED operations.

4.4 Canonical form for MPS

This is a more technical material to make some preparations for the most challenging Assignment 4 (the text and this part of the assignment is not

polished - added this year to make the Assignment 4 easier later - suggestions for improvements are welcome!). Consider a chain of length L with open boundary conditions and a quantum state in the MPS form,

$$|\psi\rangle = \sum_{\sigma_1,\sigma_2,\ldots,\sigma_L} c_{\sigma_1,\sigma_2,\ldots,\sigma_L} |\sigma_1,\sigma_2,\ldots,\sigma_L\rangle \tag{18}$$

$$c_{\sigma_1,\sigma_2,\ldots,\sigma_L} = \sum_{a_1,a_2,\ldots,a_{L-1}} A^{(1)\sigma_1}_{a_1} A^{(2)\sigma_2}_{a_1,a_2} \ldots A^{(\ell)\sigma_\ell}_{a_{\ell-1},a_\ell} \ldots A^{(L-1)\sigma_{L-1}}_{a_{L-2},a_{L-1}} A^{(L)\sigma_L}_{a_{L-1}}. \tag{19}$$

(For some later formulas to make the left end look similar to "bulk", it is convenient to introduce a_0 taking only a single value $a_0 = 1$ and write the first factor as $A^{(1)\sigma_1}_{a_1} \equiv A^{(1)\sigma_1}_{a_0=1,a_1}$; and similarly for the right end.) As sketched in previous subsections and detailed in the notes, efficient calculations with MPS can be formulated using so-called "transfer matrices" (a.k.a., "double-tensor")

$$E^{(\ell)}_{\left(a'_{\ell-1},a_{\ell-1}\right),\left(a'_\ell,a_\ell\right)} \equiv \sum_{\sigma_\ell} \left(A^{(\ell)\sigma_\ell}_{a'_{\ell-1},a'_\ell}\right)^* A^{(\ell)\sigma_\ell}_{a_{\ell-1},a_\ell} \tag{20}$$

Using transfer matrices already reduces the computational costs with $|\psi\rangle$ from exponential in L to $\sim L\chi^2$ (where χ is the bond dimension), but this can be even further reduced using so-called "canonical forms" for the MPS that make the transfer matrices particularly simple to use and are also important for other purposes. The most useful canonical form is so-called "canonical form with an orthogonality center", say, located between sites j and $j+1$, and stated as follows:

$$c_{\sigma_1,\sigma_2,\ldots,\sigma_L} = \sum_{a_1,a_2,\ldots,a_{L-1}} A^{(1)\sigma_1}_{a_1} A^{(2)\sigma_2}_{a_1,a_2} \ldots A^{(j)\sigma_j}_{a_{j-1},a_j} S_{a_j,a_j} A^{(j+1)\sigma_{j+1}}_{a_j,a_{j+1}} \ldots A^{(L-1)\sigma_{L-1}}_{a_{L-2},a_{L-1}} A^{(L)\sigma_L}_{a_{L-1}} \tag{21}$$

with $A^{(\ell)\sigma_\ell}$ matrices satisfying

$$\text{left-canonical condition for } \ell \le j: \quad \sum_{\sigma_\ell} \left(A^{(\ell)\sigma_\ell}\right)^\dagger A^{(\ell)\sigma_\ell} = 1, \tag{22}$$

$$\text{right-canonical condition for } \ell \ge j+1: \quad \sum_{\sigma_\ell} A^{(\ell)\sigma_\ell} \left(A^{(\ell)\sigma_\ell}\right)^\dagger = 1, \tag{23}$$

$$\text{non-negative diagonal matrix}: \quad S_{a_j,a_j} \ge 0 \text{ (Schmidt values)}. \tag{24}$$

The above form is slightly different from the standard MPS form because of the extra diagonal matrix S, but this can be recast into the standard form by redefining either $A^{(j)}$ or $A^{(j+1)}$ matrices "multiplying" S into them. See the notes for the left- and right-canonicity conditions written out with all indices spelled out, and also for the simplifications in calculations of the wavefunction normalization and expectation values of observables at j. What is most important for us is that this canonical form immediately gives Schmidt values for the Schmidt decomposition across the cut between j and $j+1$ as S_{a_j,a_j}, see the notes for details.

For any state, we can calculate the Schmidt values across any cut by expanding the state in the full computational basis $|\sigma_1, \sigma_2, \ldots, \sigma_L\rangle$ and performing SVD as described earlier. However,
if we have a state in an MPS form, we can achieve the calculation of the Schmidt values without ever expanding in the full computational basis by using a procedure that brings the MPS into the canonical form as follows. (Note that the original MPS need not be in the canonical form; e.g., when we act on a trial MPS with a Hamiltonian as we will do in Assignment 4, the result can be written as an MPS, but any canonical form is usually lost and needs to be restored for consistent truncations guided by the true Schmidt values of the wavefunction.) We first sweep from the left end to the right. Suppose matrices $A^{(1)}, \ldots, A^{(\ell-1)}$ are already left-canonical. Next we turn $A^{(\ell)}$ left-canonical by doing SVD of the following matrix:

$$W_{(a_{\ell-1},\sigma_\ell),(\sigma_{\ell+1},a_{\ell+1})} \equiv \sum_{a_\ell} A^{(\ell)\sigma_\ell}_{a_{\ell-1},a_\ell} A^{(\ell+1)\sigma_{\ell+1}}_{a_\ell,a_{\ell+1}} = \sum_k U_{(a_{\ell-1},\sigma_\ell),k} S_{kk} \left(V^\dagger\right)_{k,(\sigma_{\ell+1},a_{\ell+1})} \tag{25}$$

where the last equation is the result of the SVD. A reshaped U becomes the new $\tilde{A}^{(\ell)}$ [specifically, $\tilde{A}^{(\ell),\sigma_\ell}_{a_{\ell-1},k} \equiv U_{(a_{\ell-1},\sigma_\ell),k}$] satisfying the left canonicity condition, while a reshaped V with S_{kk} 's absorbed into it become the new $\tilde{A}^{(\ell+1)}$ [specifically, $\tilde{A}^{(\ell+1)\sigma_{\ell+1}}_{k,a_{\ell+1}} \equiv S_{kk} V^*_{(\sigma_{\ell+1},a_{\ell+1}),k}$]. Dropping tildes on $\tilde{A}^{(\ell)}, \tilde{A}^{(\ell+1)}$, we now have matrices $A^{(1)}, \ldots A^{(\ell-1)}, A^{(\ell)}$ as left-canonical, and we then move to the next site $\ell+1$. Note that the S_{kk} 's we find at each step are NOT Schmidt values for the wavefunction, since we really are not using information about the wavefunction outside the treated two-site blocks, and the rest of the matrices to the right are not right-canonical. Only at the very last step of the sweep when we are treating the block $(\ell, \ell+1) = (L-1, L)$ the resulting $\tilde{A}^{(\ell+1)}$ is the only matrix to the right and is right-canonical (if

we are keeping S_{kk} 's separate at this step); hence only at this point during the sweep the S_{kk} 's are Schmidt values for the cut between $L-1$ and L, and we have succeeded to obtain the canonical form with the orthogonality center between $L-1$ and L. Now we can actually move the orthogonality center to any position (between j and $j+1$ on the chain) by sweeping from the right end to the left making the matrices $A^{(L)}, A^{(L-1)}, \ldots, A^{(j+1)}$ as right-canonical. In this sweep, at each step the generated S_{kk} 's are the true Schmidt values for the whole wavefunction; when we perform the sweep since we are moving to the left keeping the right matrices right-canonical, we absorb these S_{kk} 's by multiplying them into the left matrices of the SVD at each step. The procedure involves a lot of re-shaping ("matricizing") of tensors and book-keeping, and we will practice it on a simple example in this Assignment.

5 Assignment: entanglement entropy and state compression

5.1 Image compression

Select a few grayscale images and treat each as a real-valued matrix, compressing using the SVD. For each image, use approximations of various rank, e.g., $k = \{n/4, n/8, n/16, n/32, \ldots\}$, where n is the smaller image dimension. Show some examples of images that compress well, and poorly, and compute the Frobenius distance (or "discarded norm") between the compressed images and the originals: $d(k) = \|\tilde{A}(k) - A\|_F$, where $\tilde{A}(k)$ is the rank- k approximation to image A. You may also use the singular value spectrum as a guide to truncation if there are evident features. What reduction in storage space are you able to achieve without too badly degrading the image?

```python
import numpy as np
import matplotlib.pyplot as plt
from skimage import img_as_float
import os

print(os.getcwd())

def load_and_convert_image(image_path):
    image = plt.imread(image_path)
    return img_as_float(image)

# define the image paths
image_path = 'hw2/src/parrot.png'

# load and convert the image
image = load_and_convert_image(image_path)
print(image.shape)
# perform the svd on the image


def reconstruct_image(U, s, Vt, k):
    S_k = np.diag(s[:k])
    U_k = U[:, :k]
    Vt_k = Vt[:k, :]
    return np.dot(U_k, np.dot(S_k, Vt_k))

def calculate_frobenius_norms(image_path, ranks):
    image = load_and_convert_image(image_path)
    U, s, Vt = np.linalg.svd(image, full_matrices=False)
    originals = [reconstruct_image(U, s, Vt, k) for k in ranks]
    frobenius_norms = [np.linalg.norm(original - image, 'fro') for original in originals]
    return originals, frobenius_norms

smaller_dim = min(image.shape)
# the ranks go like smaller_dim/(2^i) for i in range(0, 8)
ranks = [smaller_dim // (2**i) for i in range(8)]
reconstructions, frobenius_norms = calculate_frobenius_norms(image_path, ranks)
# show the images corresponding to the reconstructions
fig, axes = plt.subplots(2, 4, figsize=(20, 10))
for i, ax in enumerate(axes.flatten()):
    ax.imshow(reconstructions[i])
    ax.set_title(f'k = {ranks[i]}')
```

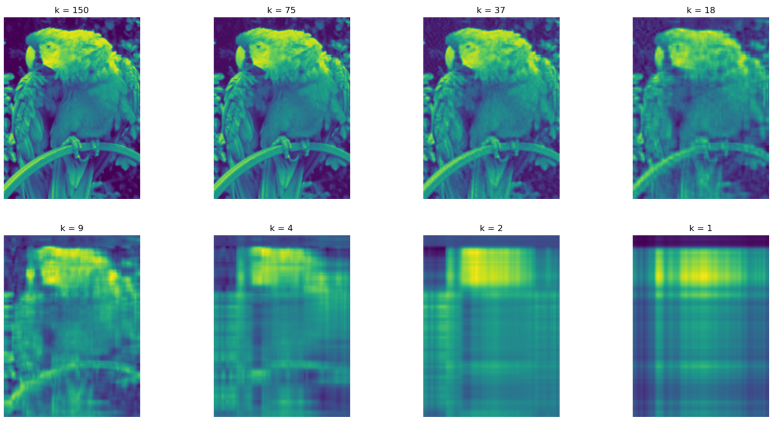

Figure 1: Parrot image

```
    ax.axis('off')
plt.savefig('reconstructions.png')
```

In my opinion, k=10 does not degrade the image to badly. This corresponds to a reduction in storage space of 88.3%.

```
# Updated dimensions for the original image
m = 200  # Height of the image
n = 150  # Width of the image
k = 10   # Rank for SVD compression

# Recalculate the storage requirements
storage_original = m * n
storage_compressed = k * (m + n + 1)

# Calculate the new savings
savings = 1 - (storage_compressed / storage_original)
savings_percentage = savings * 100  # Convert to percentage

storage_original, storage_compressed, savings_percentage
```

We also computed the Frobenius norm as a function of the rank k for the parrot image. The results are shown in the plot below. The plot shows that the Frobenius norm decreases as the rank k increases, which is expected. The Frobenius norm quantifies the difference between the original image and the compressed image, so a lower value indicates a better approximation. The

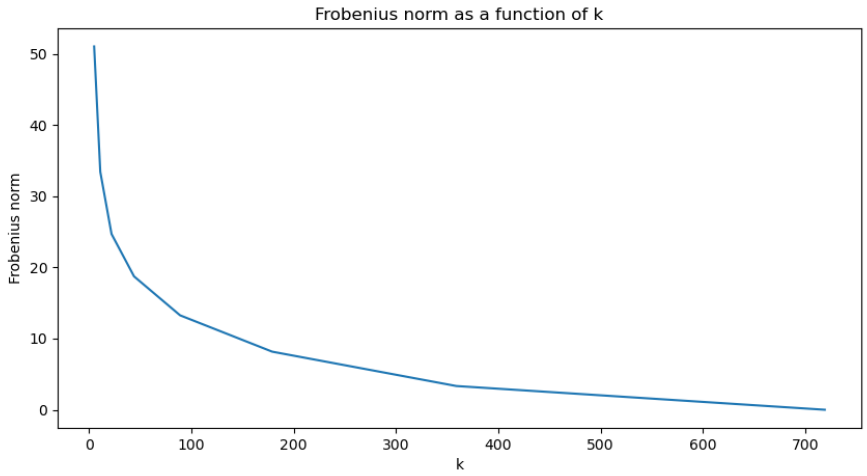

Figure 2: Frobenius norms for the parrot image

plot also shows that the rate of decrease slows down as k increases, which is also expected. This is because the largest singular values capture most of the information in the image, and the additional singular values contribute less to the overall image. Another example:

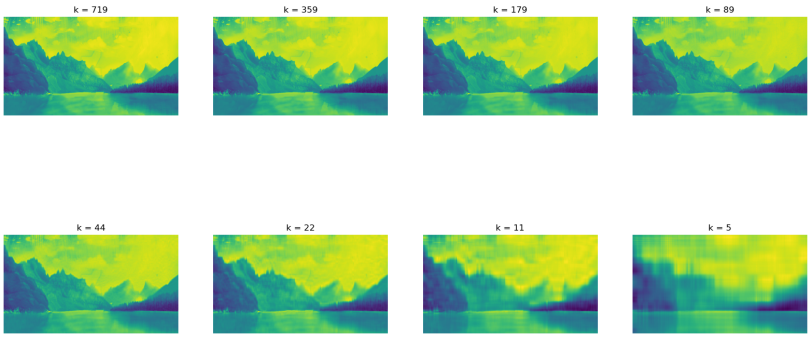

Figure 3: Landscape image

5.2 Entanglement entropy in ground states

In the rest of this assignment, you will use eigenstates of the quantum Ising Hamiltonian found by your ED code from Assignment 1, and will calculate singular value spectra and entanglement
entropy (EE) for different subsystems, studying size dependence on ℓ and L. Use a sparse eigensolver for the ground state to reach larger system sizes.

Find the ground state of the quantum Ising chain for a fixed chain length L and control parameter h/J; as in Assignment 1, solve at representative points in the ferromagnetic and paramagnetic phases and at the critical point. First impose open boundary conditions. Consider a segment A of ℓ sites starting at the left end of the chain and calculate the entanglement entropy $S(\ell; L)$ of this region. Plot $S(\ell; L)$ for $1 \leq \ell \leq L - 1$, and repeat this for several system sizes L; as a summary of these studies, plot also $S(L/2, L)$ versus L.

```
def entanglement_entropy(rho):
    """Calculate the entanglement entropy of a reduced density matrix."""
    # Normalize rho
    rho /= np.trace(rho)
    # Eigenvalues of the reduced density matrix
    eigvals = np.linalg.eigh(rho)[0]
    # Filter out zero eigenvalues to avoid log(0)
    nonzero_eigvals = eigvals[eigvals > 1e-10]
    # Entanglement entropy calculation
    entropy = -np.sum(nonzero_eigvals * np.log(nonzero_eigvals))
    return entropy

def calculate_reduced_density_matrix(state, L, ell):
    """Calculate the reduced density matrix for segment A of length ell."""
    # Reshape the state vector into a matrix
    state_matrix = state.reshape(2**ell, 2**(L - ell))
    # Calculate the reduced density matrix
    rho_A = np.dot(state_matrix, state_matrix.conj().T)
    return rho_A


L = [4, 6, 8]
h = [0.3, 1, 1.7]

entropies = {l: [] for l in L}
# Initialize data collection
S_L2_vs_L = {hi: [] for hi in h}  # This will hold entropy values for each h across all L


for l in L:
    for hi in h:
        # Calculate the ground state using the sparse_hamiltonian
        H = sparse_hamiltonian(l, hi, periodic=False)
        eigenvalues, eigenvectors = scipy.sparse.linalg.eigsh(H.astype(np.float64), k=1, which='SA')
        ground_state = eigenvectors[:, 0]

        for ell in range(1, l):
            rho_A = calculate_reduced_density_matrix(ground_state, l, ell)
```

```
            entropy = entanglement_entropy(rho_A)
            entropies[l].append(entropy)

        rho_A_L2 = calculate_reduced_density_matrix(ground_state, l, l//2)
        entropy_L2 = entanglement_entropy(rho_A_L2)
        S_L2_vs_L[hi].append((l, entropy_L2))  # Store the tuple (L, entropy)
# Define line styles for different values of h and colors for L
line_styles = ['-', '--', ':']
colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k']  # Extend if needed
markers = ['o', '^', 's']  # Example: circle, triangle, square

if len(colors) < len(L):
    raise ValueError("Not enough colors for the number of L values")

# Plot S(ell; L) for 1 <= ell <= L-1 for various L and h
plt.figure(figsize=(10, 6))
for l_index, l in enumerate(L):
    color = colors[l_index]  # Color for each L
    for h_index, hi in enumerate(h):
        start_index = h_index * (l - 1)
        end_index = start_index + l - 1
        line_style = line_styles[h_index % len(line_styles)]
        plt.plot(range(1, l), entropies[l][start_index:end_index], label=f'L = {l}, h = {hi}',
                 linestyle=line_style, color=color)

plt.xlabel(r'Segment length $\ell$')
plt.ylabel(r'Entanglement Entropy $S(\ell; L)$')
plt.legend()
plt.title(r'Entanglement Entropy $S(\ell; L)$ vs. Segment length $\ell$')
plt.grid(True)
plt.savefig('entanglement_entropy.png')
```

I have attached both plots. The following interpretations can be made. It can be noticed that when the segment length is half of the system size, an equal cut of the full Hamiltonian is made, and this maximizes the entanglement entropy for the given system size; the entanglement entropy falls off when an uneven cut is made.

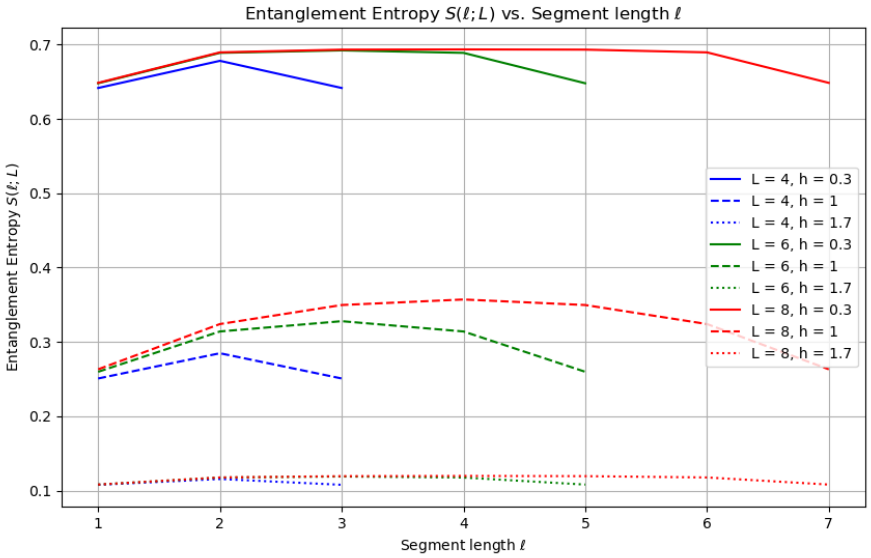

Figure 4: Entanglement entropy for different system sizes

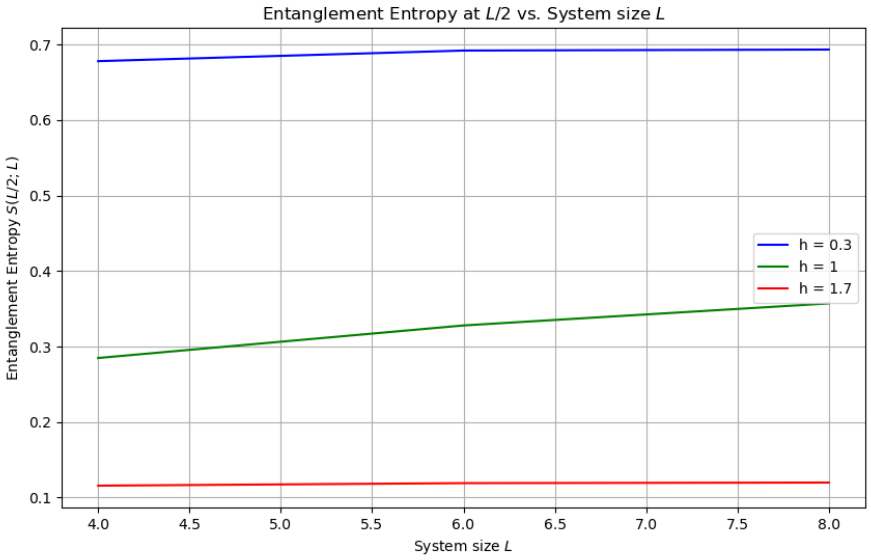

Figure 5: Entanglement entropy at the midpoint of the chain as a function of system size

```
# Create the plot
plt.figure(figsize=(10, 6))
for h_index, hi in enumerate(h):
    color = colors[h_index % len(colors)]
    # Unpack L and entropy values for plotting
    L_vals, entropies = zip(*S_L2_vs_L[hi])
    plt.plot(L_vals, entropies, label=f'h = {hi}', color=
   color, linestyle='-')

plt.xlabel(r'System size $L$')
plt.ylabel(r'Entanglement Entropy $S(L/2; L)$')
plt.title(r'Entanglement Entropy at $L/2$ vs. System size $L$
   ')
plt.legend()
plt.grid(True)
plt.savefig('entanglement_entropy_L2.png')
```

In the ferromagnetic phase, there are two possible ground states, so the entanglement entropy is near $\log 2 \approx 0.7$. In the paramagnetic phase, there is only one possible ground state, so the entanglement entropy is near $\log 1 \approx 0$. At the critical value for h, this is somewhere in the middle.

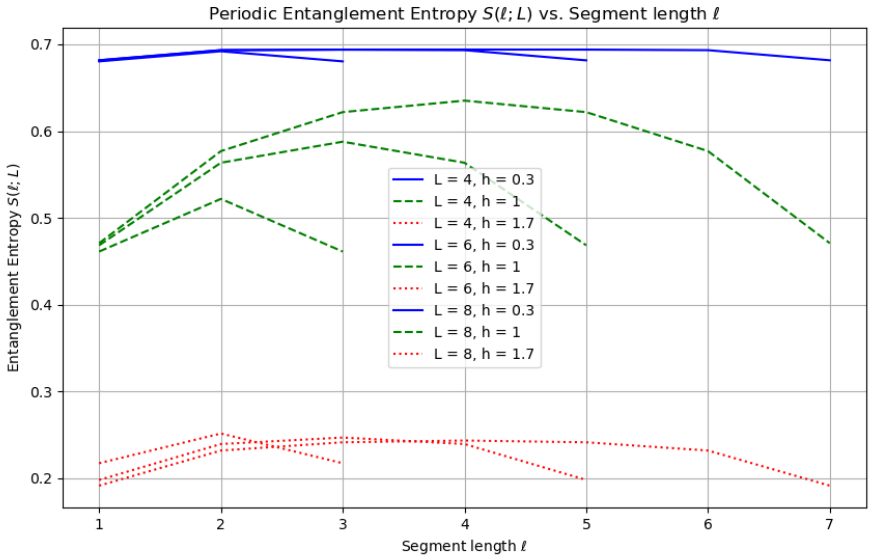

Figure 6: Entanglement entropy for different system sizes with periodic boundary conditions

Now consider the case with periodic boundary conditions and repeat the same calculations (the steps are identical to the open b.c. case). Why is the EE larger in the periodic case and roughly by what factor?

The area law tells us that the entanglement entropy will scale with the size of the boundary between the two regions. In the case of periodic boundary conditions, there can be 2 boundaries come as shown in the screenshot, while with open boundary conditions, there is only 1 boundary site in the middle of the chain. This is why the entanglement entropy is larger in the periodic case, roughly by a factor of 2. For the periodic case, I just changed

```
H = sparse_hamiltonian(l, hi, periodic=True)  # Corrected to
    periodic=True as needed
```

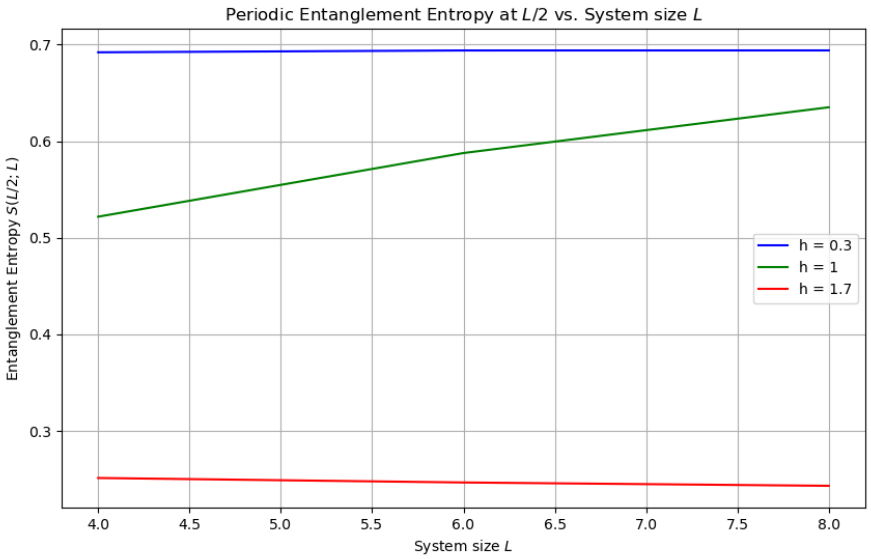

Figure 7: Entanglement entropy at the midpoint of the chain as a function of system size with periodic boundary conditions

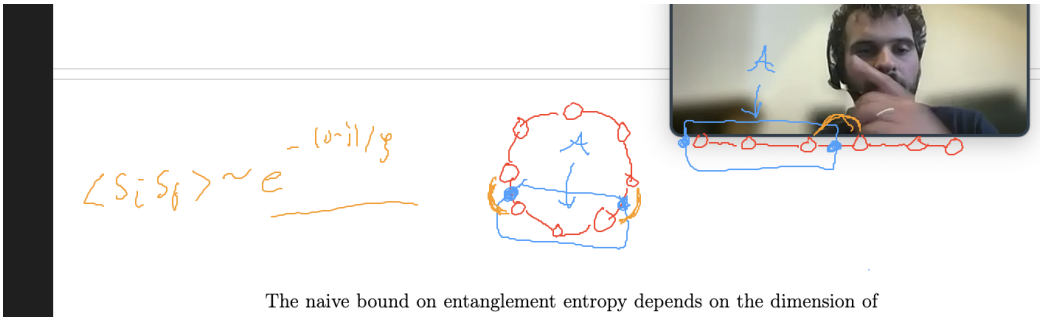

Figure 8: Periodic vs. open boundary conditions

For the largest system size, fit $S(\ell; L)$ at the critical point to the following form:

$$S(\ell; L) = \frac{c}{3} \log \left(\frac{L}{\pi} \sin \frac{\pi \ell}{L} \right) + C \tag{26}$$

This form reduces to $S(\ell) \sim \frac{c}{3} \log \ell$ in the thermodynamic limit ($1 \ll \ell \ll L$) but also nicely captures finite-size effects. (Remember to use the same log base when defining the entanglement entropy and when fitting this form.)

I did a fit for this at the critical point and L=8.

```
# Correct the definition of the fit function
def fit_function(ell, c, C):
    # Assuming max(L) is the largest value in your L array
    largest_L = max(L)
    return (c / 3) * np.log((largest_L / np.pi) * np.sin(np.pi * ell / largest_L)) + C

# Correct the extraction of critical data
# This assumes that entropies[max(L)] contains the entropies for the largest system size at the critical point h=1
critical_ell = np.array(range(1, max(L)))  # This should generate 8 values if max(L) is 8
critical_ee = np.array(entropies[max(L)])  # Make sure this has the same number of values as critical_ell


# Fit the data to the equation
params, params_covariance = curve_fit(fit_function, critical_ell, critical_ee)

# Plot the fit results
plt.figure()
plt.plot(critical_ell, critical_ee, 'bo', label='Data')
plt.plot(critical_ell, fit_function(critical_ell, *params), 'r-', label=f'Fit: ${params[0]:.2f}/3 \log((L/\pi) \sin(\pi \ell / L)) + {params[1]:.2f}$')
plt.xlabel(r'Segment length $\ell$')
plt.ylabel(r'Entanglement Entropy $S(\ell; L)$')
plt.legend()
plt.title('Fit of Entanglement Entropy at Critical Point')
plt.grid(True)
plt.savefig('entanglement_entropy_fit.png')

```

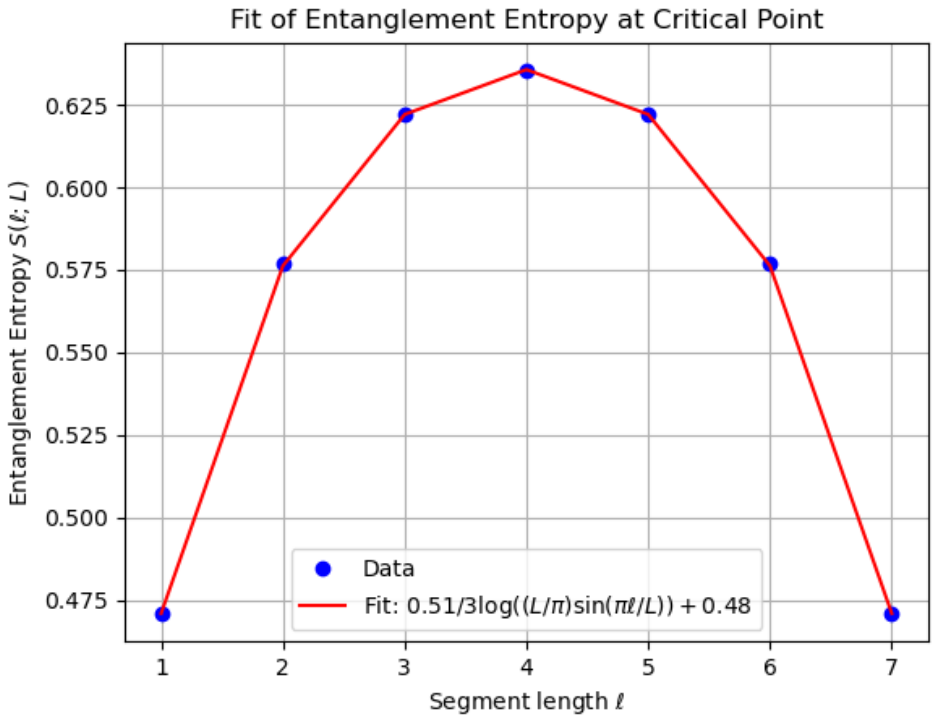

Figure 9: Fit of the entanglement entropy at the critical point

```
# Output the fit parameters
print(f"Fitted c: {params[0]}")
print(f"Fitted C: {params[1]}")
```

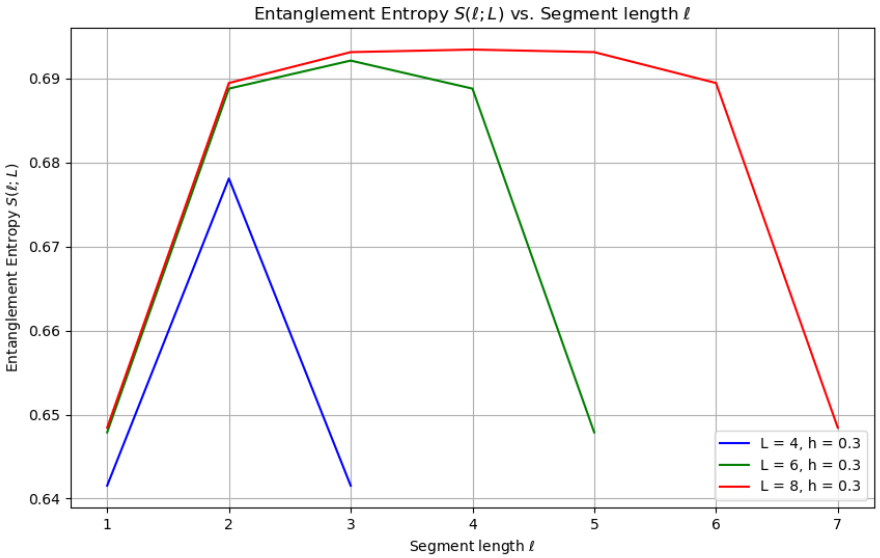

Figure 10: Entanglement entropy for the highest excited state

Have your eigensolver find the highest excited state of the Hamiltonian and perform a similar study; here it is enough to consider just one value of $h/J \neq (h/J)_c$ and one choice of boundary conditions. Why do you think the highest-energy state also satisfies the area law?

I did the same thing for the highest excited state but with the open boundary conditions.

```
eigenvalues, eigenvectors = scipy.sparse.linalg.eigsh(H.
    astype(np.float64), k=1, which='LA')
```

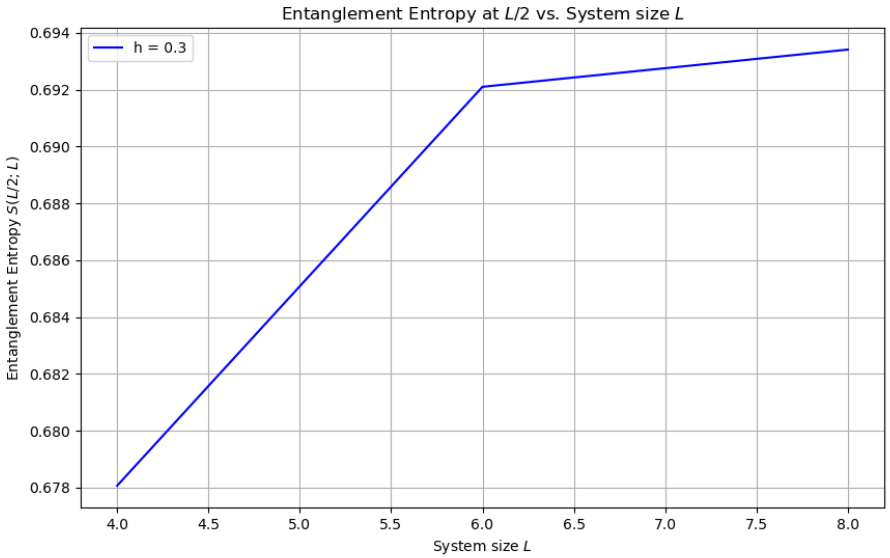

Figure 11: Entanglement entropy at the midpoint of the chain for the highest excited state

5.3 Truncation error of Schmidt decomposition

Consider again the Ising ground state at representative values of the control parameter h/J, using open boundary conditions. Perform the Schmidt decomposition at the middle of the chain and compute the approximate ground state arising from truncating the factorization at k ranging from 1 to $2^{L/2}$. For each k, calculate the discarded norm (Frobenius error) $d(k)$, as well as the error in energy of the approximate ground state, using $E = \langle\psi|H|\psi\rangle/\langle\psi \mid \psi\rangle$. Plot $\Delta E(k)$ as a function of $d(k)$; you should find a roughly linear relationship. This technique is used in MPS methods to extrapolate exact ground state energy to the $k \to \infty$ limit, using only low- k data.

```
from hw1 import sparse_hamiltonian
import numpy as np
import matplotlib.pyplot as plt

def schmidt_decomposition(psi, L):
    """Performs Schmidt decomposition at the middle of the
   chain."""
    cut_psi = psi.reshape(2**(L//2), -1)
```

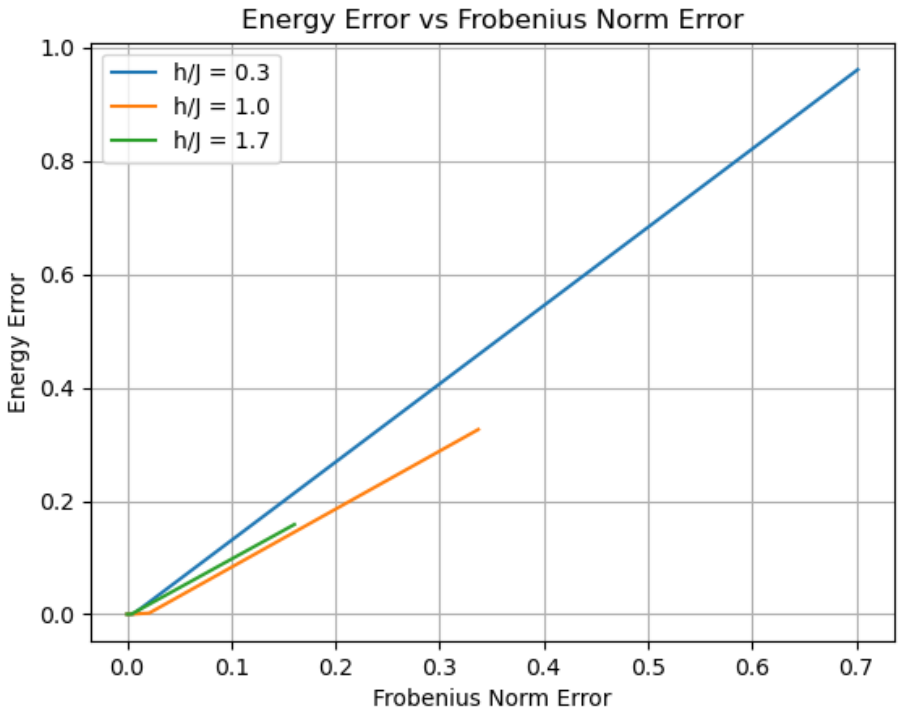

Figure 12: Truncation error of the Schmidt decomposition

```
    U, s, Vt = np.linalg.svd(cut_psi, full_matrices=False)
    return cut_psi, U, s, Vt

def truncated_state(U, S, Vh, k):
    """Reconstructs the state using only the first k singular values."""
    Sk = np.zeros_like(S)
    Sk[:k] = S[:k]
    flat_psi_k = U @ np.diag(Sk) @ Vh
    return flat_psi_k, flat_psi_k.flatten()

h_values = [0.3, 1, 1.7]
L = 8
errors = {hi: [] for hi in h_values}

# Single loop to handle everything per each h
for hi in h_values:
    # Compute the ground state
    H = sparse_hamiltonian(L, hi, periodic=False)
    eigenvalues, eigenvectors = np.linalg.eigh(H.toarray())
    ground_state = eigenvectors[:, 0]

    # Perform Schmidt decomposition
    original, U, s, Vt = schmidt_decomposition(ground_state, L)

    # Loop through all possible k values for truncation
    for k in range(1, 2**(L//2) + 1):
        mat_psi_k, flat_psi_k = truncated_state(U, s, Vt, k)
        norm_error = np.linalg.norm(original - mat_psi_k, 'fro')
        psi_k_energy = (flat_psi_k.conj().T @ H @ flat_psi_k) / (flat_psi_k.conj().T @ flat_psi_k)
        energy_error = np.abs(psi_k_energy - eigenvalues[0])
        errors[hi].append((norm_error, energy_error))

# Plotting the results
for hi, error_list in errors.items():
    norm_errors, energy_errors = zip(*error_list)
    plt.plot(norm_errors, energy_errors, label=f'h/J = {hi:.1f}')

plt.xlabel('Frobenius Norm Error')
plt.ylabel('Energy Error')
plt.title('Energy Error vs Frobenius Norm Error')
```

```
plt.legend()
plt.grid()
plt.savefig('energy_vs_frobenius.png')
```

5.4 Entanglement entropy of highly excited states

Now we will calculate the entanglement entropy $S(\ell; L)$ for an eigenstate in the middle of the many-body spectrum. Here, consider just one value of h/J (e.g., in the paramagnetic phase) and one choice of boundary conditions (e.g., periodic b.c.), but perform a systematic study of $S(\ell, L)$ for several L. Note that the sparse eigensolver does not work in the middle of the spectrum, so you will need a full diagonalization and will thus access smaller systems. To avoid complications with resolving degeneracies, consider a few eigenstates corresponding to non-degenerate eigenvalues close to zero energy (which is exactly in the middle of the spectrum in this model). You will see that the entanglement entropy is much larger for states in the middle of the spectrum compared to the band edges and should observe volume law scaling on average.

What can be observed here is that the entanglement entropy for ground

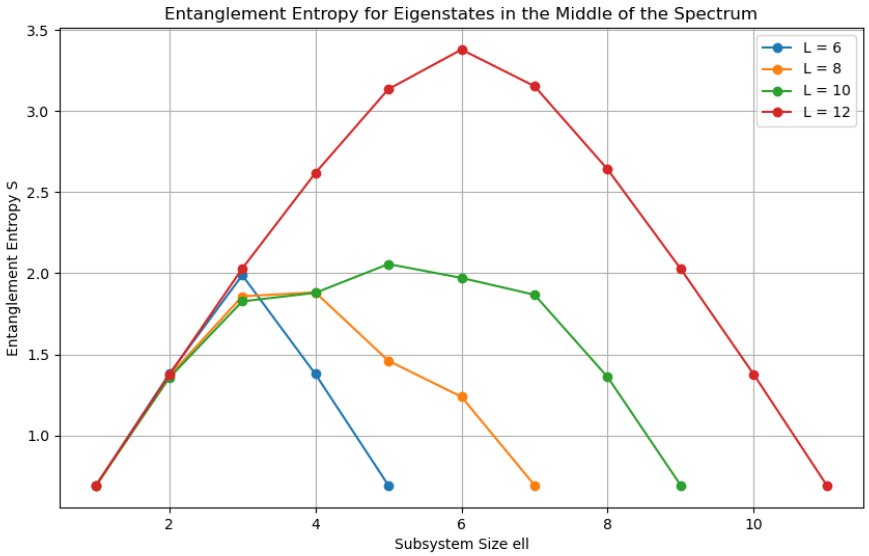

Figure 13: Entanglement entropy for eigenstates in the middle of the spectrum

states ranged from 0.3-0.7, but in this case, it can range from like 1.0-3.5 for the states in the middle of the spectrum. This is a result of there being more excited states with the same eigenvalue to choose from than just the ground

state. For the ferromagnetic phase that we are considering here, there were 2 ground states to choose from so the entanglement entropy was bounded by $\log 2 \approx 0.7$. Also, we see a manifestation of the volume law, where the larger system sizes show a larger entanglement entropy.

```
from hw1 import periodic_dense_hamiltonian_explicit
from p5_2 import entanglement_entropy,
    calculate_reduced_density_matrix
import numpy as np
import matplotlib.pyplot as plt

h = 1.7
L_sizes = [6, 8, 10, 12]  # Different system sizes
entropies = {L: [] for L in L_sizes}  # Dictionary to store
    entropies by L

for L in L_sizes:
    H = periodic_dense_hamiltonian_explicit(L, h)
    eigenvalues, eigenvectors = np.linalg.eigh(H)

    # Select a eigenstate near the middle of the spectrum
    mid_index = len(eigenvalues) // 2
    # make sure that it is not degenerate
    if eigenvalues[mid_index] != eigenvalues[mid_index + 1]
    and eigenvalues[mid_index] != eigenvalues[mid_index - 1]:
        state = eigenvectors[:, mid_index]
    else:
        # If the middle index is degenerate, shift slightly
    to find a non-degenerate state
        offset = 1
        while (mid_index + offset < len(eigenvalues) and
    eigenvalues[mid_index] == eigenvalues[mid_index + offset])
    or \
              (mid_index - offset >= 0 and eigenvalues[
    mid_index] == eigenvalues[mid_index - offset]):
            offset += 1
        state = eigenvectors[:, mid_index + offset]

    # Compute the entanglement entropy for each ell using the
    same procedure as in p5_2.py
    for ell in range(1, L):
        rho_A = calculate_reduced_density_matrix(state, L,
    ell)
        entropy = entanglement_entropy(rho_A)
```

```
        entropies[L].append((ell, entropy))  # Store ell and entropy

# Plotting the results
plt.figure(figsize=(10, 6))
for L in L_sizes:
    ells, L_entropies = zip(*entropies[L])  # Unpack ell and entropy values for each L
    plt.plot(ells, L_entropies, marker='o', linestyle='-', label=f'L = {L}')

plt.xlabel('Subsystem Size ell')
plt.ylabel('Entanglement Entropy S')
plt.title('Entanglement Entropy for Eigenstates in the Middle of the Spectrum')
plt.legend()
plt.grid(True)
plt.savefig('entanglement_entropy_middle_spectrum.png')
```

5.4.1 Optional. Volume law for random states

Generate a random vector in the same Hilbert space of L spins and study its entanglement $S(\ell; L)$. You will get a reasonably random vector by simply choosing an i.i.d. random value - for example, uniformly distributed in $[-1, 1]$ - for each amplitude of the 2^L basis states. However, in order
to actually sample a random vector with a basis-independent distribution uniform over the unit sphere (in the manifold of states that can be written as real-valued wavefunctions), you can choose coefficients from a Gaussian distribution. Remember to normalize your vectors. Repeating this process, you will observe average scaling with the volume law of entanglement.

5.5 Matrix product state representation for ground states

Return to the setup in Sec. 5.2, considering the ground state $|\psi_{\text{gs}}\rangle$ with open b.c., using a sparse eigensolver to reach larger sizes. You should now consider two points, one fairly close to the phase transition, say $h/J = 5/4$, and one exactly at $h/J = (h/J)_c = 1$. Compute MPS approximations of the ED wavefunction, varying the bond dimension k. What is the actual reduction in storage space you achieve? Contract the virtual indices of the tensors of the MPS to obtain exponentially large $\left|\tilde{\psi}_{\text{gs}}(k)\right\rangle$ and compute the overlap $\left\langle \tilde{\psi}_{\text{gs}}(k) \mid \psi_{\text{gs}} \right\rangle$ with the original ED wavefunction.

0.0.1 Answer

My results are shown. The approximation is not that good for k=1, but it is surprisingly good for the ground state as k increases even in small increments. This is likely because we tried to approximate the ground state which does not have much entanglement and can be well approximated as a product state. It also should be noted that the rank 2 approximation is so good because we are dealing with a system of 2 possible spins i.e. no more than 2 singular values are needed for a good approximation to the state.

```
import numpy as np
from scipy.sparse.linalg import eigsh  # Sparse matrix
    eigensolver
from hw1 import sparse_hamiltonian  # Assuming this function
    generates your Hamiltonian
from numpy.linalg import norm

def truncate_svd(matrix, k):
    """
    Truncates the singular value decomposition (SVD) of a
    matrix.

    Parameters:
    matrix (ndarray): The input matrix.
    k (int): The number of singular values to keep.

    Returns:
    ndarray: The truncated left singular vectors.
    ndarray: The truncated singular values.
    ndarray: The truncated right singular vectors.
    """
```

```
h=1.25, k=1, Overlap: 0.693, Storage Reduction: 2048.000
h=1.25, k=2, Overlap: 0.999, Storage Reduction: 546.133
h=1.25, k=3, Overlap: 1.000, Storage Reduction: 264.258
h=1.25, k=4, Overlap: 1.000, Storage Reduction: 154.566
h=1.25, k=5, Overlap: 1.000, Storage Reduction: 105.703
h=1.25, k=6, Overlap: 1.000, Storage Reduction: 76.561
h=1.25, k=7, Overlap: 1.000, Storage Reduction: 57.894
h=1.25, k=8, Overlap: 1.000, Storage Reduction: 45.260
h=1.25, k=9, Overlap: 1.000, Storage Reduction: 37.406
h=1, k=1, Overlap: 0.504, Storage Reduction: 2048.000
h=1, k=2, Overlap: 0.994, Storage Reduction: 546.133
h=1, k=3, Overlap: 0.999, Storage Reduction: 264.258
h=1, k=4, Overlap: 1.000, Storage Reduction: 154.566
h=1, k=5, Overlap: 1.000, Storage Reduction: 105.703
h=1, k=6, Overlap: 1.000, Storage Reduction: 76.561
h=1, k=7, Overlap: 1.000, Storage Reduction: 57.894
h=1, k=8, Overlap: 1.000, Storage Reduction: 45.260
h=1, k=9, Overlap: 1.000, Storage Reduction: 37.406
```

Figure 14: Overlap between the MPS approximation and the exact ground state

```
    u, s, vt = np.linalg.svd(matrix, full_matrices=False)
    return u[:, :k], np.diag(s[:k]), vt[:k, :]


def compute_mps(state, k):
    # this is just the length of the system
    L = int(np.log2(state.size))
    # prepare the state for the initial SVD
    state = state.reshape((2, -1))
    # list to store the mps tensors
    mps_tensors = []
    # initialize a previous bond dimension
    previous_k = 2

    # loop over the system
    for i in range(1, L+1):
        # make the SVD
        u, s, vt = truncate_svd(state, k)
        # current bond dimension
        current_k = min(k, u.shape[1])
```

```
        # debugging
        print(f'Current state shape: {state.shape}, u shape: {u.shape}, s shape: {s.shape}, vt shape: {vt.shape}')

        # for the first iteration
        if i == 1:
            mps_tensors.append(u.reshape((previous_k, -1)))
        # for the middle iterations
        elif i < L:
            # append the rank 3 tensor following the notation of the tensor network diagrams
            mps_tensors.append(u.reshape((previous_k, 2, current_k)))
        # for the last iteration
        else:
            mps_tensors.append(u.reshape((previous_k, -1)))
            break
        # prepare the state for the next iteration
        state = (s @ vt).reshape((2*current_k, -1))
        # update the previous bond dimension
        previous_k = current_k

    return mps_tensors




def reconstruct_state(mps_tensors):
    state = mps_tensors[0]  # Start with the first tensor, assumed to be a matrix.

    for j in range(1, len(mps_tensors)):
        next_tensor = mps_tensors[j]

        # this is for the first construction
        if len(state.shape) == 2:
            # print(f"before state shape: {state.shape}, tensor shape: {next_tensor.shape}")
            state = np.einsum('ij,jkl->ikl', state, next_tensor)
        # this is for the middle cases
        elif len(state.shape) == 3 and len(next_tensor.shape) == 3:
            state = state.reshape(-1, state.shape[2])
```

```
            # print(f"before state shape: {state.shape}, tensor shape: {next_tensor.shape}")
            state = np.einsum('ij,jkl->ikl', state, next_tensor)
        # now for the final case
        elif len(next_tensor.shape) == 2:
            state = state.reshape(-1, state.shape[2])
            next_tensor = next_tensor.reshape(state.shape[1], -1)
            print(f"before state shape: {state.shape}, tensor shape: {next_tensor.shape}")
            state = np.einsum('ij,jk->ik', state, next_tensor)


    return state.reshape(-1)  # Flatten the final state to a 1D array









def calculate_overlap(original, reconstructed):
    return np.abs(np.vdot(original, reconstructed) / (norm(original) * norm(reconstructed)))

# Parameters
L = 16 # System size
h_values = [5/4, 1]  # Near and at the critical point
k_values = np.arange(1, 10, 1)  # Bond dimensions for MPS, from 1 to 20 inclusive


results = {}

for h in h_values:
    H = sparse_hamiltonian(L, h, periodic=False)
    eigenvalues, eigenvectors = eigsh(H.astype(np.float64), k=1, which='SA')
    gs = eigenvectors[:, 0]
    gs /= np.linalg.norm(gs)
    results[h] = {}

```

```
    for k in k_values:
        mps_tensors = compute_mps(gs, k)
        print(mps_tensors)
        reconstructed_gs = reconstruct_state(mps_tensors)
        # print out the gs and reconstructed_gs
        print(f"gs: {gs}")
        print(f"reconstructed_gs: {reconstructed_gs}")
        overlap = calculate_overlap(gs, reconstructed_gs)
        num_params_original = gs.size
        num_params_mps = sum(t.size for t in mps_tensors)
        storage_reduction = num_params_original /
    num_params_mps
        results[h][k] = {
            'ground_state': gs,
            'mps': mps_tensors,
            'overlap': overlap,
            'storage_reduction': storage_reduction
        }

# Printing or processing results as needed
for h in results:
    for k in results[h]:
        print(f"h={h}, k={k}, Overlap: {results[h][k]['
    overlap']:.3f}, Storage Reduction: {results[h][k]['
    storage_reduction']:.3f}")
```

5.5. Efficient calculations with MPS

We will now make use of the efficient measurements afforded by MPS, which are described in Sec. 4.3. Set up such a calculation for $E(k) = \left\langle \tilde{\psi}_{\rm gs}(k)|H|\tilde{\psi}_{\rm gs}(k) \right\rangle / \left\langle \tilde{\psi}_{\rm gs}(k) \mid \tilde{\psi}_{\rm gs}(k) \right\rangle$ using the MPS, comparing your answer with the known energy eigenvalue that was given by your exact diagonalization routine. Also compute the same correlation functions you studied in Assignment 1, for both values of h/J, and examine the convergence of the results in each case for several values of k.

0.0.2 Answer

We know that the quantum Ising model has a Hamiltonian given by

$$H = -J \sum_{i=1}^{L-1} \sigma_i^z \sigma_{i+1}^z - h \sum_{i=1}^{L} \sigma_i^x \tag{1}$$

where σ_i^x, σ_i^z are the Pauli matrices acting on site i, and J, h are the coupling constants. The normalization would be:

$$\sum_{i=1}^{L} I_i \tag{1}$$

where I is the identity matrix.

```
Energy for k=1: -6.0, Exact: -7.296229810558755
Energy for k=2: -7.294090833077591, Exact: -7.296229810558755
Energy for k=3: -7.295889777926114, Exact: -7.296229810558755
Energy for k=4: -7.2962297276160655, Exact: -7.296229810558755
Energy for k=5: -7.296229798693841, Exact: -7.296229810558755
Energy for k=6: -7.2962298105299865, Exact: -7.296229810558755
Energy for k=7: -7.296229810555161, Exact: -7.296229810558755
Energy for k=8: -7.296229810558752, Exact: -7.296229810558755
Energy for k=9: -7.296229810558752, Exact: -7.296229810558755
Energy for k=10: -7.296229810558752, Exact: -7.296229810558755
Energy for k=11: -7.296229810558752, Exact: -7.296229810558755
Energy for k=12: -7.296229810558752, Exact: -7.296229810558755
Energy for k=13: -7.296229810558752, Exact: -7.296229810558755
Energy for k=14: -7.296229810558752, Exact: -7.296229810558755
Energy for k=15: -7.296229810558752, Exact: -7.296229810558755
Energy for k=16: -7.296229810558752, Exact: -7.296229810558755
Energy for k=17: -7.296229810558752, Exact: -7.296229810558755
Energy for k=18: -7.296229810558752, Exact: -7.296229810558755
Energy for k=19: -7.296229810558752, Exact: -7.296229810558755
```

Figure 15: Energy convergence for different values of k

```python
# define the operators
sigma_z = np.array([[1, 0], [0, -1]])
sigma_x = np.array([[0, 1], [1, 0]])

# Define parameters
L = 6
k_values = np.arange(1, 20, 1)
h = 1.0

# Obtain the ground state from the Hamiltonian
H = sparse_hamiltonian(L, h, periodic=False)
eigenvalues, eigenvectors = eigsh(H, k=1, which='SA')
gs = eigenvectors[:, 0]

# Iterate over various bond dimensions k
for k in k_values:
    mps_tensors = compute_mps(gs, k)
    bra = [t.conj().T for t in mps_tensors]  # Prepare the bra state by conjugate transposing every tensor in the lest
    # make a function that takes in a list of mps tensors and its corresponding bra and computes the normalization
```

```
    def compute_contraction(mps_tensors, bra):
        # contract the physical energy_interactions on every tensor to generate a list of 2-tensors
        contraction = np.einsum('ijk,lji->kl', mps_tensors[0], bra[0])
        for j in range(1, len(mps_tensors)):
            if j == len(mps_tensors) - 1:
                contraction = np.einsum('kl,kmo,lmo->', contraction, mps_tensors[j], bra[j])
            else:
                contraction = np.einsum('kl,kmn,oml->no', contraction, mps_tensors[j], bra[j])

        return contraction
    # Compute energy components
    energy_interactions = 0
    energy_field = 0

    # Apply interaction
    for j in range(len(mps_tensors) - 1):
        mod_tensor1 = apply_operator(mps_tensors[j], sigma_z)
        mod_tensor2 = apply_operator(mps_tensors[j + 1], sigma_z)
        # make a new MPS lest that replaces the j-th and j+1-th tensors with the modified ones
        mps_tensors_mod = mps_tensors.copy()
        mps_tensors_mod[j] = mod_tensor1
        mps_tensors_mod[j + 1] = mod_tensor2
        # compute the contraction of the modified tensors
        energy_interactions -= compute_contraction(mps_tensors_mod, bra)


    # apply transverse field
    for j in range(len(mps_tensors)):
        # now we only have one tensor to consider for the field term
        mod_tensor = apply_operator(mps_tensors[j], sigma_x)
        # make a new MPS lest that replaces the j-th and j+1-th tensors with the modified ones
        mps_tensors_mod = mps_tensors.copy()
        mps_tensors_mod[j] = mod_tensor
        # compute the contraction of the modified tensors
        energy_field -= h*compute_contraction(mps_tensors_mod, bra)
```

```
55
56      normalization = compute_contraction(mps_tensors, bra)
57      total_energy = (energy_interactions + energy_field) /
        normalization
58      print(f"Energy for k={k}: {total_energy}, Exact: {
        eigenvalues[0]}")
```

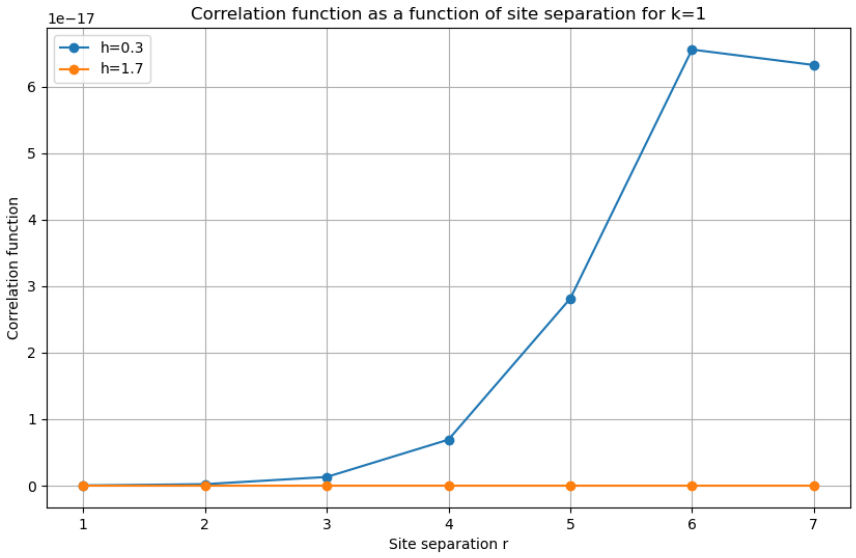

Figure 16: Correlation function for k = 1

Also, I computed the correlation function for various values of k. For all cases, it can be seen that the face with the high h has a low correlation length, whereas the face with low h has a high correlation length. This is because the high h face is in the paramagnetic phase, where the spins are not correlated, whereas the low h face is in the ferromagnetic phase, where the spins are correlated. What we also see is that the rank 1 is not very consistent, but once we get to rank 2 or higher, the correlation length stays at a consistent value of 0 or 1 for the paramagnetic and ferromagnetic phases, respectively.

```
import numpy as np
from scipy.sparse.linalg import eigsh
# from hw1.src.hw1 import sparse_hamiltonian
import scipy.sparse
import matplotlib.pyplot as plt

def truncate_svd(matrix, k):
    """
    Truncates the singular value decomposition (SVD) of a
   matrix.

```

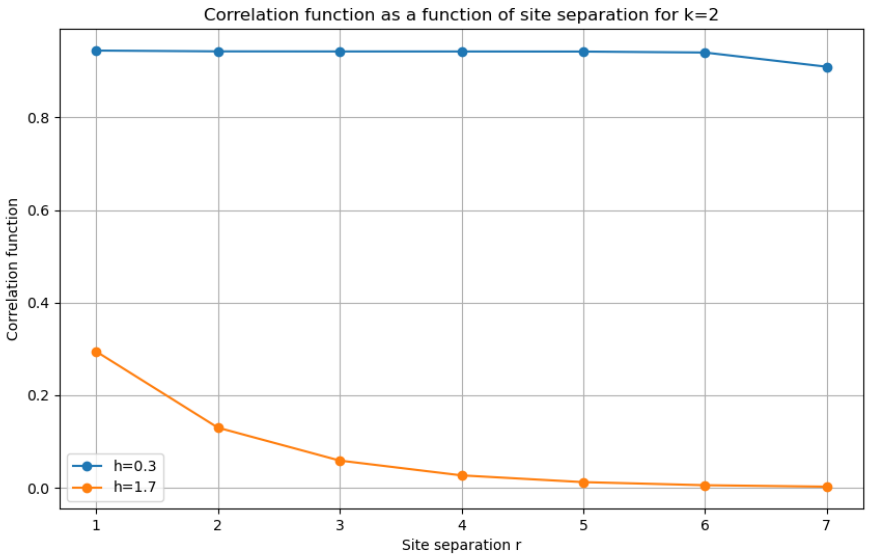

Figure 17: Correlation function for k = 2

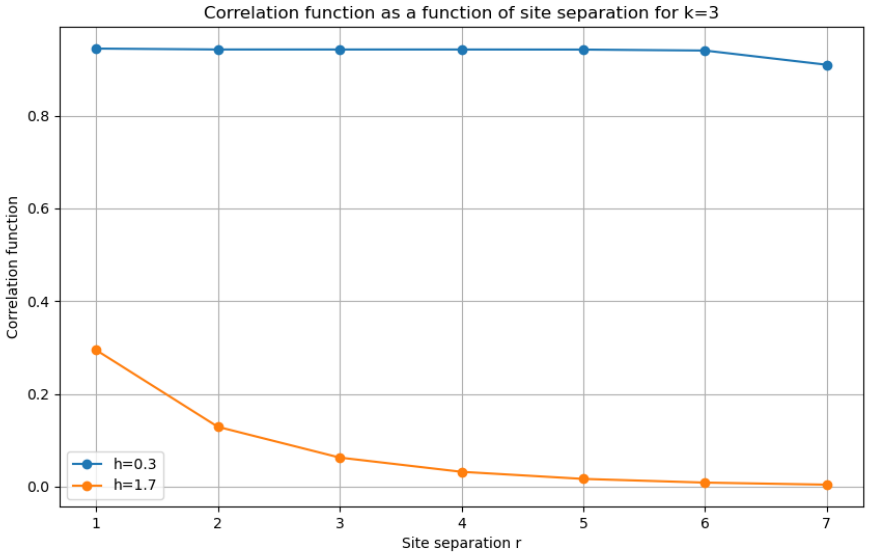

Figure 18: Correlation function for k = 3

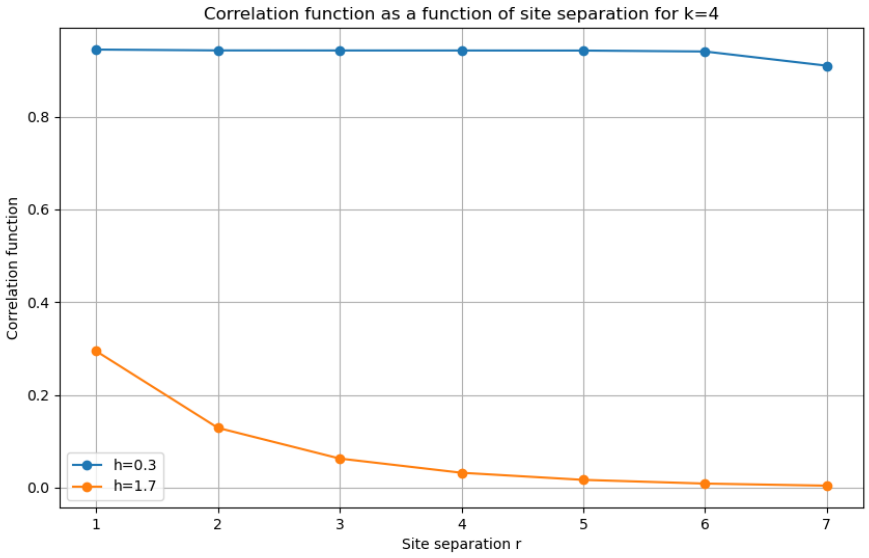

Figure 19: Correlation function for k = 4

```
    Parameters:
    matrix (ndarray): The input matrix.
    k (int): The number of singular values to keep.

    Returns:
    ndarray: The truncated left singular vectors.
    ndarray: The truncated singular values.
    ndarray: The truncated right singular vectors.
    """
    u, s, vt = np.linalg.svd(matrix, full_matrices=False)
    return u[:, :k], np.diag(s[:k]), vt[:k, :]


def compute_mps(state, k):
    # this is just the length of the system
    L = int(np.log2(state.size))
    # prepare the state for the initial SVD
    state = state.reshape((2, -1))
    # list to store the mps tensors
    mps_tensors = []
    # initialize a previous bond dimension
    previous_k = 2
```

```
33
34      # loop over the system
35      for i in range(1, L+1):
36          # make the SVD
37          u, s, vt = truncate_svd(state, k)
38          # current bond dimension
39          current_k = min(k, u.shape[1])
40          # debugging
41          # print(f'Current state shape: {state.shape}, u shape
    : {u.shape}, s shape: {s.shape}, vt shape: {vt.shape}')
42
43          # for the first iteration
44          if i == 1:
45              mps_tensors.append(u.reshape((1, previous_k, -1))
    )
46          # for the middle iterations
47          elif i < L:
48              # append the rank 3 tensor following the notation
     of the tensor network diagrams
49              mps_tensors.append(u.reshape((previous_k, 2,
    current_k)))
50          # for the last iteration
51          else:
52              mps_tensors.append(u.reshape((previous_k, -1, 1))
    )
53              break
54          # prepare the state for the next iteration
55          state = (s @ vt).reshape((2*current_k, -1))
56          # update the previous bond dimension
57          previous_k = current_k
58
59      return mps_tensors
60
61
62  def binary_string(n, L):
63      binary = bin(n)[2:]
64      while len(binary) < L:
65          binary = '0' + binary
66      return binary
67
68  def sparse_hamiltonian(L, h, J=1, periodic=False):
69      row = []
70      col = []
71      data = []
72      # loop over the index i
```

```
    for i in range(2**L):
        # turn that index into a binary string
        bi = binary_string(i, L)

        # to the diagonal element first
        total_interaction = 0
        # initialize the loop range based on the boundary conditions
        loop_range = L if periodic else L - 1
        for k in range(loop_range):
            # Handle periodic boundary by wrapping the index
            next_k = (k + 1) % L if periodic else k + 1
            # Add neighbor interaction based on the spin values
            total_interaction += -J * (1 if bi[k] == bi[next_k] else -1)
        row.append(i)
        col.append(i)
        data.append(total_interaction)

        # now consider the off-diagonal elements connected by a single spin flip
        # make a function that takes the bit i and returns a list of basis states connected by a single flip
        def flip(i):
            connected_bases_states = []
            for j in range(L):
                flip_i = i ^ (1 << j)
                connected_bases_states.append(flip_i)
            return connected_bases_states

        # loop over the connected basis states
        for flip_i in flip(i):
            row.append(i)
            col.append(flip_i)
            data.append(-h)

    # Create a sparse matrix from the lists
    H = scipy.sparse.coo_matrix((data, (row, col)), shape=(2**L, 2**L))
    return H

def apply_operator(tensor, operator):
    return np.einsum('jk,ikl->ijl', operator, tensor)

```

```
# define the operators
sigma_z = np.array([[1, 0], [0, -1]])
sigma_x = np.array([[0, 1], [1, 0]])
def mps_correlations(mps_tensors):
    # Calculate the correlation function of a state.
    contractions = []
    L = len(mps_tensors)
    bra = [t.conj().T for t in mps_tensors]
    mod_tensor1 = apply_operator(mps_tensors[0], sigma_z)
    for r in range(1, L):
        # just modify the tensor at site r
        mod_tensor_r = apply_operator(mps_tensors[r], sigma_z
    )
        mps_tensors_mod = mps_tensors.copy()
        mps_tensors_mod[0] = mod_tensor1
        mps_tensors_mod[r] = mod_tensor_r
        # compute the contraction
        contraction = compute_contraction(mps_tensors_mod,
    bra)
        contractions.append(contraction)
    return contractions
def compute_contraction(mps_tensors, bra):
    # contract the physical energy_interactions on every
    tensor to generate a list of 2-tensors
    contraction = np.einsum('ijk,lji->kl', mps_tensors[0],
    bra[0])
    for j in range(1, len(mps_tensors)):
        if j == len(mps_tensors) - 1:
            contraction = np.einsum('kl,kmo,lmo->',
    contraction, mps_tensors[j], bra[j])
        else:
            contraction = np.einsum('kl,kmn,oml->no',
    contraction, mps_tensors[j], bra[j])

    return contraction
# Define parameters
L = 8
k_values = np.arange(1, 5, 1)
h_values = [0.3, 1.7]

for k in k_values:
    plt.figure(figsize=(10, 6))
    plt.title(f'Correlation function as a function of site
    separation for k={k}')
    plt.xlabel('Site separation r')
```

```
    plt.ylabel('Correlation function')
    for h in h_values:
        # Obtain the ground state from the Hamiltonian
        H = sparse_hamiltonian(L, h, periodic=False)
        eigenvalues, eigenvectors = eigsh(H, k=1, which='SA')
        gs = eigenvectors[:, 0]
        mps_tensors = compute_mps(gs, k)
        correlations = mps_correlations(mps_tensors)
        r_values = range(1, L)
        plt.plot(r_values, correlations, 'o-', label=f'h={h}'
    )
    plt.legend()
    plt.grid()
    plt.savefig(f'5-5_correlation_k{k}.png')
```

5.5.2 Bringing an MPS to a canonical form with an orthogonality center

Consider an MPS state of the form Eq. (19) with bond dimension 2 and matrices

$$A^{(1)\uparrow} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \end{pmatrix}, \quad A^{(1)\downarrow} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -1 \end{pmatrix}, \tag{27}$$

$$A^{(\ell)\uparrow} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}, \quad A^{(\ell)\downarrow} = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 & 0 \\ 1 & -1 \end{pmatrix}, \quad 2 \leq \ell \leq L-1, \tag{28}$$

$$A^{(L)\uparrow} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad A^{(L)\downarrow} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \tag{29}$$

(Aside note: this state is related to so-called cluster state in quantum information theory.) You can verify analytically that this is a very special state where at each $\ell = 2, \ldots, L-1$ the matrices $A^{(\ell)\sigma}$ satisfy both left- and right-canonicity condition, while $A^{(1)\sigma}$ satisfy the left-canonicity and $A^{(L)\sigma}$ satisfy the right-canonicity.
Given the MPS matrices:

$$A^{(1)\uparrow} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \end{bmatrix}, \quad A^{(1)\downarrow} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -1 \end{bmatrix}$$

We compute the left-canonical condition:

$$\left(A^{(1)\uparrow}\right)^\dagger A^{(1)\uparrow} + \left(A^{(1)\downarrow}\right)^\dagger A^{(1)\downarrow}$$

Calculating each term:

$$\left(A^{(1)\uparrow}\right)^\dagger = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix}, \quad \left(A^{(1)\downarrow}\right)^\dagger = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\-1\end{bmatrix}$$

$$\left(A^{(1)\uparrow}\right)^\dagger A^{(1)\uparrow} = \frac{1}{2}\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}, \quad \left(A^{(1)\downarrow}\right)^\dagger A^{(1)\downarrow} = \frac{1}{2}\begin{bmatrix}1 & -1\\-1 & 1\end{bmatrix}$$

Summing these:

$$\frac{1}{2}\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix} + \frac{1}{2}\begin{bmatrix}1 & -1\\-1 & 1\end{bmatrix} = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$$

This shows that the sum is the identity matrix, satisfying the left-canonical condition. Proving the right canonical condition will be similar. ChatGPT was also able to convince me that the middle matrices satisfy both the left and right canonicity conditions.
Hence, this state can be viewed also as being in the canonical form whose orthogonality center can be viewed as located anywhere along the chain. What are the Schmidt values across any cut? (Hint: matrix S is not explicitly present but what such matrix can you trivially insert? The state is not normalized but the normalization can be trivally achieved.)

You can trivially insert the identity matrix.

For some reasonable L, have the computer write out the full wavefunction in the computational basis by multiplying out the matrices. By reusing your previous routines, verify the analytical predictions for the Schmidt values across any cut.

I did this and I verified that the Schmidt values across any cut are 1.

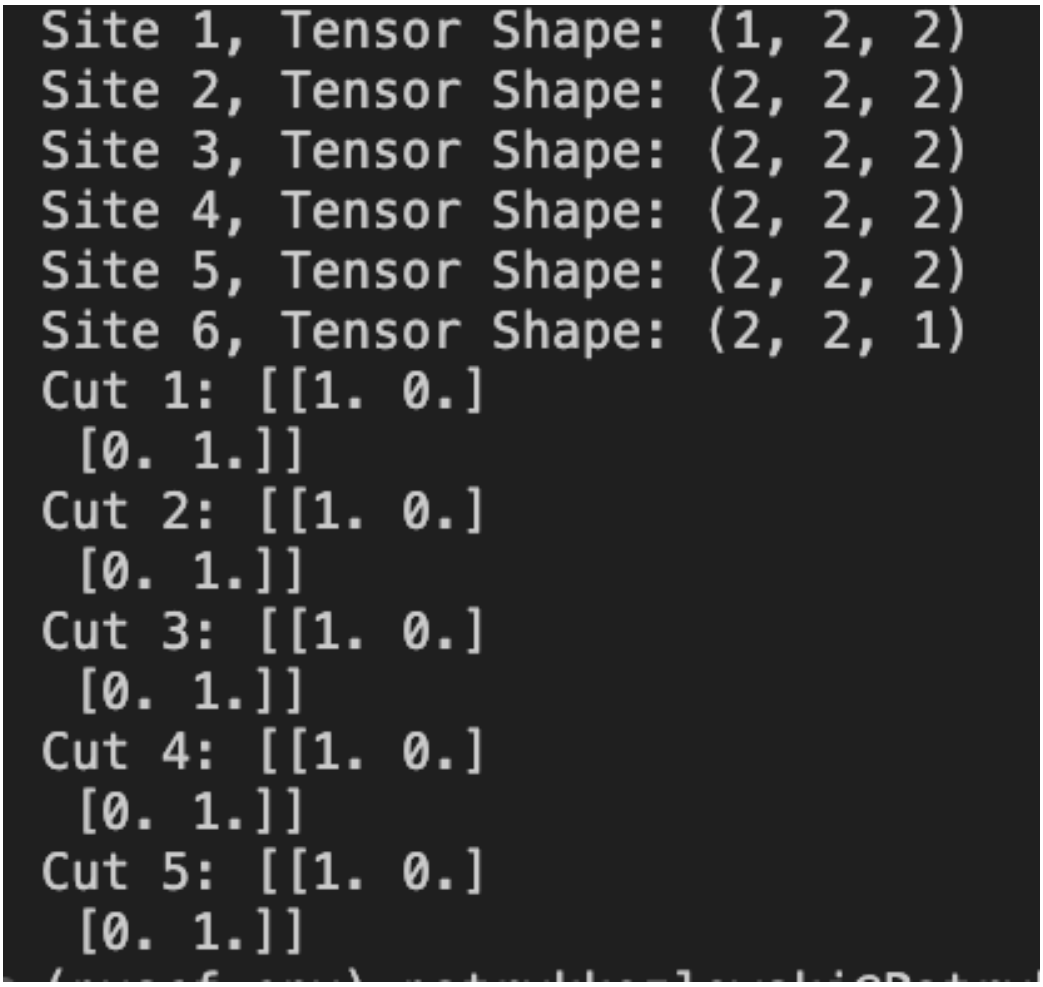

Figure 20: Schmidt values across any cut

```python
import numpy as np
from p5_5 import truncate_svd

# Define MPS matrices for a chain
L = 6  # Number of sites
mps_tensors = []

# Initial tensor setup
A_1_up = np.array([1, 1]) / np.sqrt(2)
```

```
A_1_down = np.array([1, -1]) / np.sqrt(2)
mps_tensors.append(np.stack((A_1_up, A_1_down), axis=0).
    reshape(1, 2, 2))  # First tensor

# # initial tensor setup for second part
# A_1_up = np.array([1, 2])
# A_1_down = np.array([2, -1])
# mps_tensors.append(np.stack((A_1_up, A_1_down), axis=0).
    reshape(1, 2, 2))  # First tensor

# Middle tensors
for site in range(1, L-1):
    A_ell_up = np.array([[1, 1], [0, 0]]) / np.sqrt(2)
    A_ell_down = np.array([[0, 0], [1, -1]]) / np.sqrt(2)
    mps_tensors.append(np.stack((A_ell_up, A_ell_down), axis
    =0))

# Last tensor
A_L_up = np.array([1, 0])
A_L_down = np.array([0, 1])
mps_tensors.append(np.stack((A_L_up, A_L_down), axis=0).
    reshape(2, 2, 1))  # Last tensor

# # last tensor for second part
# A_L_up = np.array([1, 3])
# A_L_down = np.array([3, 1])
# mps_tensors.append(np.stack((A_L_up, A_L_down), axis=0).
    reshape(2, 2, 1))  # Last tensor


# Display tensor shapes
for idx, tensor in enumerate(mps_tensors):
    print(f"Site {idx+1}, Tensor Shape: {tensor.shape}")

# Calculate Schmidt values at each cut
for cut in range(1, L):
    # Contract tensors to the left of the cut
    left_state = mps_tensors[0].reshape(-1, 2)  # Start with
    the first tensor reshaped for matrix multiplication
    # first make a loop from 1 to cut - 1
    for i in range(1, cut):
        left_state = np.einsum('ab,bcd->acd', left_state,
    mps_tensors[i])
        left_state = left_state.reshape(-1, mps_tensors[i].
    shape[2])  # Ensure correct shape
```

```
47
48      # Contract tensors to the right of the cut
49      right_state = mps_tensors[cut].reshape(2, -1)  # Start
    with the tensor at the cut
50      # now make a loop from cut+1 to L-1
51      for j in range(cut + 1, L):
52          right_state = np.einsum('abc,ad->cbd', mps_tensors[j
    ], right_state)
53          right_state = right_state.reshape(mps_tensors[j].
    shape[0], -1)  # Ensure correct shape
54
55      # Compute the singular values directly from reshaped left
     and right states
56      schmidt_matrix = np.dot(left_state, right_state)
57      u, s, vt = truncate_svd(schmidt_matrix, 2)  # Adjust the
    truncation
58      print(f"Cut {cut}: {s}")
```

Let us now consider a modified state where we will change the matrices at the left and right ends as follows:

$$A^{(1)\uparrow} = \begin{pmatrix} 1 & 2 \end{pmatrix}, \quad A^{(1)\downarrow} = \begin{pmatrix} 2 & -1 \end{pmatrix}, \tag{30}$$

$$A^{(L)\uparrow} = \begin{pmatrix} 1 \\ 3 \end{pmatrix}, \quad A^{(L)\downarrow} = \begin{pmatrix} 3 \\ 1 \end{pmatrix}, \tag{31}$$

without changing any of the intermediate $A^{(\ell)\sigma}, 2 \leq \ell \leq L-1$. Even though these intermediate matrices satisfy the canonicity conditions, these conditions are NOT satisfied by the boundary matrices and the full state is NOT in any canonical form. Hence you cannot read off any Schmidt values from the present form. Again, have the computer write out the full wavefunction in the computational basis for some reasonable L and calculate brute force the Schmidt values across all cuts. (Note that this wavefunction is not normalized, so if you want to calculate the entanglement entropy or observables, you would first normalize it appropriately, which is easy to do numerically.)

I did this and I calculated the Schmidt values across all cuts. The Schmidt values are not 1, which is expected because the state is not in a canonical form.

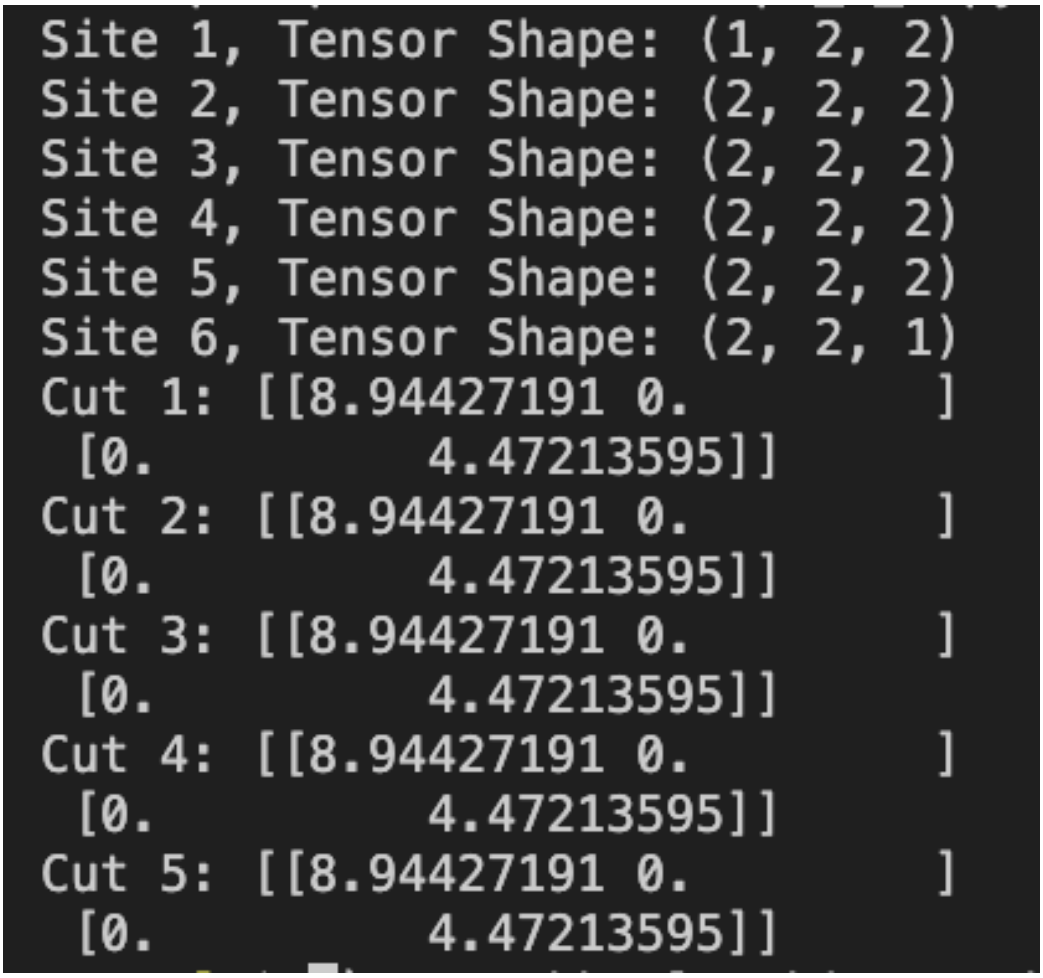

Figure 21: Schmidt values across all cuts

```python
import numpy as np
from p5_5 import truncate_svd

# Define MPS matrices for a chain
L = 6  # Number of sites
mps_tensors = []

```

```
# # Initial tensor setup
# A_1_up = np.array([1, 1]) / np.sqrt(2)
# A_1_down = np.array([1, -1]) / np.sqrt(2)
# mps_tensors.append(np.stack((A_1_up, A_1_down), axis=0).reshape(1, 2, 2))  # First tensor

# initial tensor setup for second part
A_1_up = np.array([1, 2])
A_1_down = np.array([2, -1])
mps_tensors.append(np.stack((A_1_up, A_1_down), axis=0).reshape(1, 2, 2))  # First tensor

# Middle tensors
for site in range(1, L-1):
    A_ell_up = np.array([[1, 1], [0, 0]]) / np.sqrt(2)
    A_ell_down = np.array([[0, 0], [1, -1]]) / np.sqrt(2)
    mps_tensors.append(np.stack((A_ell_up, A_ell_down), axis=0))

# # Last tensor
# A_L_up = np.array([1, 0])
# A_L_down = np.array([0, 1])
# mps_tensors.append(np.stack((A_L_up, A_L_down), axis=0).reshape(2, 2, 1))  # Last tensor

# last tensor for second part
A_L_up = np.array([1, 3])
A_L_down = np.array([3, 1])
mps_tensors.append(np.stack((A_L_up, A_L_down), axis=0).reshape(2, 2, 1))  # Last tensor


# Display tensor shapes
for idx, tensor in enumerate(mps_tensors):
    print(f"Site {idx+1}, Tensor Shape: {tensor.shape}")

# Calculate Schmidt values at each cut
for cut in range(1, L):
    # Contract tensors to the left of the cut
    left_state = mps_tensors[0].reshape(-1, 2)  # Start with the first tensor reshaped for matrix multiplication
    # first make a loop from 1 to cut - 1
    for i in range(1, cut):
        left_state = np.einsum('ab,bcd->acd', left_state, mps_tensors[i])
```

```
46          left_state = left_state.reshape(-1, mps_tensors[i].
    shape[2])  # Ensure correct shape
47
48      # Contract tensors to the right of the cut
49      right_state = mps_tensors[cut].reshape(2, -1)  # Start
    with the tensor at the cut
50      # now make a lope from cut+1 to L-1
51      for j in range(cut + 1, L):
52          right_state = np.einsum('abc,ad->cbd', mps_tensors[j
    ], right_state)
53          right_state = right_state.reshape(mps_tensors[j].
    shape[0], -1)  # Ensure correct shape
54
55      # Compute the singular values directly from reshaped left
     and right states
56      schmidt_matrix = np.dot(left_state, right_state)
57      u, s, vt = truncate_svd(schmidt_matrix, 2)  # Adjust the
    truncation
58      print(f"Cut {cut}: {s}")
```

Now try bringing this MPS to the canonical form using the procedure described in the text. Start from the left end and move rightwards, performing appropriate SVDs on two-site blocks making all matrices to the left satisfy the left-canonical conditions. After reaching the right end, you will obtain the canonical form with the orthogonality center between $L-1$ and L. Now by moving leftwards performing appropriate SVDs, you can move the orthognality center to any location along the chain. As you do so, you can read off the true Schmidt values for the whole wavefunction at each cut and check against the results of the brute-force calculation in the previous paragraph. This procedure will be used extensively in Assignment 4, so make sure you get it well-tested here getting all the back-and-forth "reshapings" and groupings right.

Note that the cost of running this procedure is linear in L, so doubling the system size will take just two times longer (and not exponentially longer as in the brute-force approach).

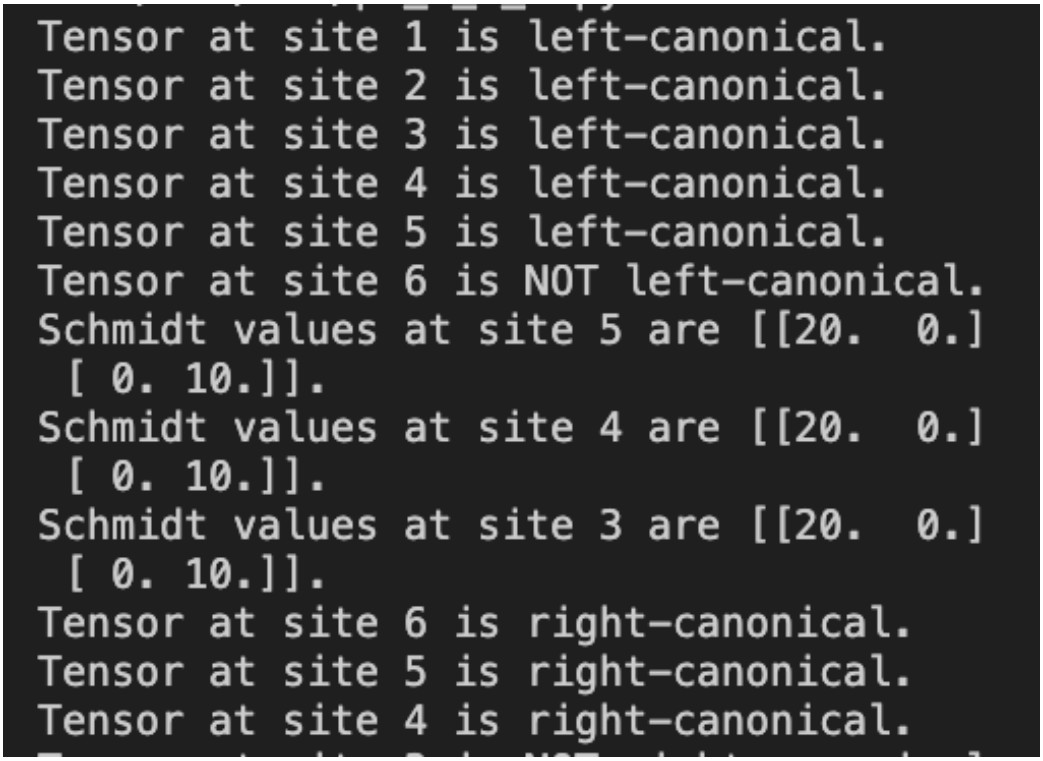

Figure 22: Moving the orthogonality center

My screenshot shows how I first made the MPS list left canonical until the end cut, then I made it rite canonical until the orthogonality center that I chose at the python index 2. I then calculated the Schmidt values across all cuts and compared them to the brute force calculation. The Schmidt values are the same as the brute force calculation up to normalization. Then I explicitly check whether the mps up to that orthogonality center is rite canonical.

```
import numpy as np
from p5_5 import truncate_svd  # Assuming truncate_svd
    performs SVD and truncates based on some criterion

L = 6  # Number of sites
mps_tensors = []

# First tensor initialization
A_1_up = np.array([1, 2])
```

```
A_1_down = np.array([2, -1])
mps_tensors.append(np.stack((A_1_up, A_1_down), axis=0).
    reshape(1, 2, 2))

# Middle tensors
for site in range(1, L-1):
    A_ell_up = np.array([[1, 1], [0, 0]]) / np.sqrt(2)
    A_ell_down = np.array([[0, 0], [1, -1]]) / np.sqrt(2)
    mps_tensors.append(np.stack((A_ell_up, A_ell_down), axis
    =0))

# Last tensor
A_L_up = np.array([1, 3])
A_L_down = np.array([3, 1])
mps_tensors.append(np.stack((A_L_up, A_L_down), axis=0).
    reshape(2, 2, 1))

def left_canonicalize(mps_tensors):
    L = len(mps_tensors)
    for i in range(L-1):
        current_tensor = mps_tensors[i].reshape(-1,
    mps_tensors[i].shape[-1])  # Flatten the tensor for SVD
        u, s, vt = truncate_svd(current_tensor, min(
    current_tensor.shape))  # Perform SVD
        u = u.reshape(mps_tensors[i].shape[0], mps_tensors[i
    ].shape[1], u.shape[-1])  # Reshape U
        mps_tensors[i] = u  # Store the left-canonical form
    of the tensor

        # Update the next tensor by absorbing the S part of
    SVD and the entire next tensor
        if i < L-1:
            next_tensor = mps_tensors[i+1].reshape(
    mps_tensors[i+1].shape[0], -1)
            mps_tensors[i+1] = np.dot(s @ vt, next_tensor).
    reshape(-1, mps_tensors[i+1].shape[1], mps_tensors[i+1].
    shape[2])

    # The last tensor absorbs the remaining transformation
    last_shape = mps_tensors[-1].shape
    mps_tensors[-1] = np.dot(s @ vt, mps_tensors[-1].reshape(
    last_shape[0], -1)).reshape(last_shape)

# Call the function
left_canonicalize(mps_tensors)
```

```
def check_left_canonical(mps_tensors):
    for idx, tensor in enumerate(mps_tensors):
        # Reshape the tensor for easier multiplication
        # We consider the left and physical indices as 'rows' and the right index as 'columns'
        tensor_reshaped = tensor.reshape(-1, tensor.shape[-1])
        # Compute the product of the tensor and its conjugate transpose
        product = np.dot(tensor_reshaped.conj().T, tensor_reshaped)
        # Check if the product is close to the identity matrix
        identity = np.eye(tensor_reshaped.shape[-1])
        if np.allclose(product, identity):
            print(f"Tensor at site {idx+1} is left-canonical.")
        else:
            print(f"Tensor at site {idx+1} is NOT left-canonical.")

check_left_canonical(mps_tensors)

def right_canonicalize(mps_tensors, ortho_center):
    L = len(mps_tensors)
    for i in range(L - 1, ortho_center, -1):
        current_tensor = mps_tensors[i].reshape(mps_tensors[i].shape[0], -1)
        u, s, vt = truncate_svd(current_tensor, min(current_tensor.shape))
        # print out the schmidt valgus at each cut
        print(f"Schmidt values at site {i} are {s}.")
        vt = vt.reshape(min(current_tensor.shape), mps_tensors[i].shape[1], -1)
        mps_tensors[i] = vt
        # update the next tensor by absorbing u and s into it to prepare for the next svd
        if i > ortho_center:
            next_tensor = mps_tensors[i - 1].reshape(mps_tensors[i - 1].shape[-1], -1)
            mps_tensors[i - 1] = np.dot(u @ s, next_tensor)
    # treat the last case differently
    if i == ortho_center:
        mps_tensors[i - 1] = np.dot(u @ s, mps_tensors[i - 1].reshape(-1, mps_tensors[i - 1].shape[-1])).reshape(
```

```
    mps_tensors[i - 1].shape)
74
75
76
77
78 # make another function that checks if the list is right
    canonical
79 def check_right_canonical(mps_tensors):
80     # iterate through the list in a reversed order
81     for idx, tensor in reversed(list(enumerate(mps_tensors)))
    :
82         tensor_reshaped = tensor.reshape(tensor.shape[0], -1)
83         product = np.dot(tensor_reshaped, tensor_reshaped.
    conj().T)
84         identity = np.eye(tensor_reshaped.shape[0])
85         if np.allclose(product, identity):
86             print(f"Tensor at site {idx+1} is right-canonical
    .")
87         else:
88             print(f"Tensor at site {idx+1} is NOT right-
    canonical.")
89 right_canonicalize(mps_tensors, 2)
90 check_right_canonical(mps_tensors)
```

References

[1] Schollwöck, Ulrich. "The density-matrix renormalization group in the age of matrix product states." Annals of Physics 326, no. 1 (2011): 96-192.