Vrije Universiteit Brussel

Faculteit Wetenschappen and Bio-ingenieurswetenschappen
Vakgroep Fysica

Chaos and simulation of quantum spin-chains

Bachelor Project

Emil Pedersen

5 June 2020

2019 – 2020

Promotor: Prof. Dr. Ben Craps
Co-Promotor: Dr. Surbhi Khetrapal

This bachelor's thesis came about (in part) during the period in which higher education was subjected to a lockdown and protective measures to prevent the spread of the COVID-19 virus. The process of formatting, data collection, the research method and/or other scientific work the thesis involved could therefore not always be carried out in the usual manner. The reader should bear this context in mind when reading this bachelor's thesis, and also in the event that some conclusions are taken on board.

Contents

1 Introduction

Quantum entanglement is a fundamental property of quantum physics which does not exist in classical physics. Entanglement has become an increasingly important way of characterizing quantum many-body systems. An important question is how entanglement can be created and spreads starting from a system in a non-entangled product-state. Recent results have also shown that the time-evolution of entanglement can be an interesting tool to distinguish integrable and chaotic systems [7]. One way to quantify entanglement is via the entanglement entropy and mutual information.

A major obstacle when studying general quantum many-body systems numerically is that the Hilbert space grows exponentially with the system size, which means that (exact) simulation requires exponential resources on classical computers (when representing the states by their coefficients in the Hilbert space). Recently there has been significant development in the field of tensor networks, with the discovery of many efficient methods for simulating quantum many-body systems representing their states as tensor networks [2]. One such method for one-dimensional lattice systems is time-evolving block decimation (TEBD). When constructed correctly, tensor networks can also reveal the internal entanglement "structure" of the state. In one-dimensional systems these interesting tensor networks are matrix product states (MPS).

The Ising model is an interesting and common example of a simple one-dimensional quantum many-body system (a spin chain), because it can be both integrable and chaotic depending on the values of the free parameters of the Hamiltonian. In the integrable case, analytic results are obtainable for the time-evolution of the entanglement entropy.

The main aim of this project was to write a program implementing the TEBD algorithm for efficiently simulating the time-evolution of entanglement entropy and mutual information in the Ising model. I developed the program completely myself without using any code from other sources. I then used my program to study the time-evolution of entanglement in some integrable and nonintegrable cases of the Ising model with chains of different lengths. I simulated the time evolution of entanglement entropy starting from random product-states as initial states and looked at the average evolution, extending the results of [5] to longer chains. And I simulated the evolution of entanglement entropy and mutual information starting from a specific product-state as initial state, to compare the results of [7] in a different spin chain model, also discovering other interesting differences between the integrable and nonintegrable cases.

This first section of the report will explain the necessary theory to understand the implemented algorithms and the discussion of the results. I

will give an introduction to tensor networks and matrix product states and how to calculate entanglement entropy in matrix product states. Then I will explain TEBD and define the Ising model. In section 2 I will then explain the most important considerations for the numerical implementation, and in section 3 I will present and discuss the results obtained with my program. Sections 4 and 5 contain the conclusion and acknowledgments, respectively. In the appendices A there is a short worked-out example of how to construct a canonical MPS (as described in 1.3.3), and I present my code implementing the algorithms explained in the text.

1.1 Density matrices and entanglement entropy

For a pure state $|\Psi\rangle \in H_A \otimes H_B$ the density matrix is simply $\rho = |\Psi\rangle\langle\Psi|$. The bipartite von Neumann entanglement entropy of subsystem A is defined as

$$S(A) = -\operatorname{Tr}(\rho_A \log \rho_A) \quad , \tag{1}$$

where $\rho_A = \operatorname{Tr}_B(\rho)$ is the reduced density matrix of subsystem A. The convention from quantum informatics is to use the base-2 logarithm, which will also be used in the rest of this project. Performing the trace in (1) in the orthonormal basis of H_A where ρ_A is diagonal, with eigenvalues λ_k, gives

$$S(A) = \sum_k -\lambda_k \log \lambda_k \quad . \tag{2}$$

It can be proven that ρ_B has the same eigenvalues as ρ_A (see section 1.3.2 (23)) so $S(A) = S(B)$. The entanglement entropy $S(A) = S(B)$ is a measure for the amount entanglement between subsystems A and B. If $|\Psi\rangle = |\Psi_A\rangle \otimes |\Psi_B\rangle$ is a product-state the reduced density matrix is $\rho_A = |\Psi_A\rangle\langle\Psi_A|$ and has only one nonzero eigenvalue $\lambda_0 = 1$ so the entanglement entropy $S(A) = 0$. So $S(A) = 0$ if and only if $|\Psi\rangle$ is a product state for the bipartition (A,B) and thus has no entanglement between A and B.

A useful quantity that can be defined in terms of the entanglement entropy is the mutual information. The mutual information of two subsystems A and B is defined as

$$I(A,B) = S(A) + S(B) - S(A \cup B) \quad . \tag{3}$$

The mutual information is a measure for the amount of information subsystems A and B have in common (due to their entanglement).

1.2 Tensor networks and Matrix product states

1.2.1 Tensor networks and diagrams

In what follows a tensor is defined as a multidimensional array of complex numbers, without extra requirements. The rank of a tensor is its number of indices. An index contraction is the sum of all possible values of the repeated indices of a set of tensors. For example, the matrix product of three matrices (= rank-2 tensors) A_{ij}, B_{jk} and C_{kl} is

$$D_{il} = \sum_{j,k} A_{ij} B_{jk} C_{kl} \quad , \tag{4}$$

which is the contraction over the indices j and k of A_{ij}, B_{jk} and C_{kl}. A tensor network is a collection of tensors with some (possibly all) of their indices contracted. The indices which are not contracted are called the open indices of the tensor network. Tensor networks can be represented visually by tensor network diagrams. In tensor network diagrams tensors are represented by shapes, and their indices are represented by lines emerging from the shapes. Contractions are represented by connecting the lines of the contracted indices between two tensors. (A line can thus be seen as an identity or kronecker-delta tensor δ_{ij}). So a tensor network is represented by a set of shapes, connected to and by lines. The following diagram represents the matrix product (4):

$$\sum_{j,k} A_{ij} B_{jk} C_{kl} = \quad$$

. (5)

In a quantum N-body system, given a local basis $\{|i_r\rangle \mid i_r = 0, 1, ..., d_r - 1\}$ (with d_r the dimension of the local subspace) for each site r, any state $|\Psi\rangle$ of the complete system can be written uniquely in terms of the tensor product basis as

$$|\Psi\rangle = \sum_{i_1,...,i_N} \Psi^{i_1 \cdots i_N} |i_1\rangle \otimes ... \otimes |i_N\rangle \quad . \tag{6}$$

The state is uniquely described by the coefficients $\Psi^{i_1 \cdots i_N}$, which form a rank-N tensor $\Psi^{i_1 \cdots i_N}$. This requires $\prod_{r=1}^{N} d_r$ independent (up to normalization) complex numbers, which is a computationally inefficient description of the state, exponentially large in the system size. Tensor Networks allow to break up such large tensors like $\Psi^{i_1 \cdots i_N}$ into networks of smaller lower rank tensors which are more efficient to work with and, when constructed correctly, can reveal some of the internal (entanglement) structure of the state.

Similarly any operator $\hat{O}$ can be represented uniquely as

$$\hat{O} = \sum_{j_1,\ldots,j_N} \sum_{i_1,\ldots,i_N} O^{j_1\ldots j_N}_{i_1\ldots i_N} |j_1\rangle \otimes \ldots \otimes |j_N\rangle \langle i_1| \otimes \ldots \otimes \langle i_N| \tag{7}$$

and the operator is uniquely defined by the coefficients $O^{j_1\ldots j_N}_{i_1\ldots i_N}$, which form a rank-$2N$ tensor. The action of this operator $|\Phi\rangle = \hat{O}|\Psi\rangle$ can be represented by contraction of corresponding tensors as

$$\Phi^{j_1\ldots j_N} = \sum_{i_1,\ldots,i_N} O^{j_1\ldots j_N}_{i_1\ldots i_N} \Psi^{i_1\ldots i_N} \quad . \tag{8}$$

Like the coefficients of an N-body state, the coefficient tensor of an operator is exponentially large in the system size and therefore a computationally inefficient representation.
In the tensor network diagrams, I will use the convention that open indices represented by lines emerging vertically upwards from the tensors will be assumed contracted with the local basis of the corresponding site (which sites and basis will be specified when necessary) and open indices represented by lines emerging vertically downwards will be assumed contracted with the local dual-basis. With this convention some example states and operators $|\Psi\rangle$, $\langle\Psi|$, $\hat{O}$ and $|\Phi\rangle = \hat{O}|\Psi\rangle$ with $N = 4$ are represented in Figure 1. The star in Ψ^* denotes complex conjugation of all entries of Ψ.

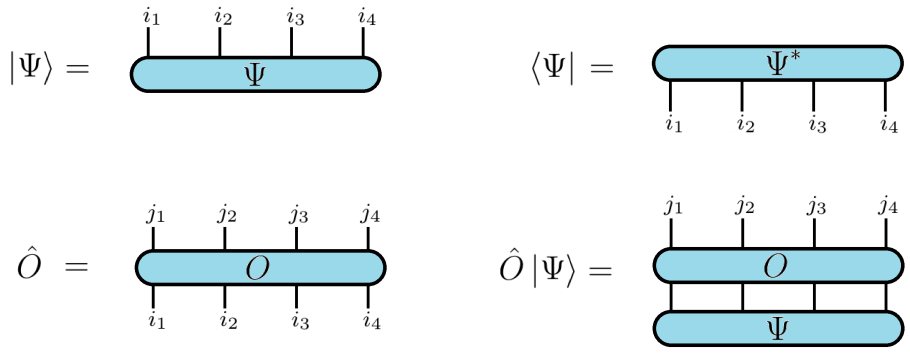


Figure 1: Example tensor network diagrams of 4-site states and operators

In the implementation of many of the algorithms described later, it is often useful to reshape tensors. This can be done by swapping, merging or splitting the indices. I will denote the merging of two indices i and j

of a tensor by (ij) and use the standard convention ("last index changing fastest") that the numerical value of the compound index $k = (ij)$ will be given by:

$$k = d_i i + j \quad , \tag{9}$$

where d_i is the dimension of i. When splitting a (compound) index k into two separate indices i and j, the numerical values will then be (inverse of (9)):

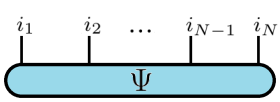
$$\begin{aligned} i &= \left\lfloor \frac{k}{d_i} \right\rfloor \\ j &= \mathrm{rem}(k, d_i) \end{aligned} \quad . \tag{10}$$

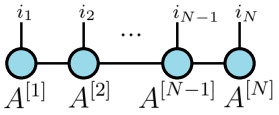
1.2.2 Matrix Product States

Matrix Product States (MPS) are a common form of tensor networks which are behind many powerful algorithms, such as TEBD and DMRG, for simulation of one-dimensional quantum many-body systems. The matrix product form of a rank-N tensor $\Psi^{i_1 \ldots i_N}$ is a decomposition into a contraction of N tensors very similar to a normal matrix product (4):

$$\Psi^{i_1 \ldots i_N} = \sum_{\alpha_1, \ldots, \alpha_N} A^{[1] i_1}_{\alpha_1} A^{[2] i_2}_{\alpha_1 \alpha_2} \ldots A^{[N-1] i_{N-1}}_{\alpha_{N-2} \alpha_{N-1}} A^{[N] i_N}_{\alpha_{N-1}} \quad . \tag{11}$$

In TN diagram notation:

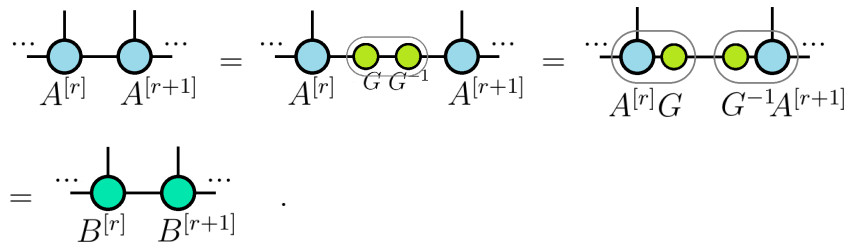
$|\Psi\rangle =$ 

$=$ 

, (12)

where the ellipsis in the first diagram denotes the missing indices $i_3, ..., i_{N-2}$ and the ellipsis in the second diagram denotes the missing tensors $A^{[3]}, ...,$ $A^{[N-2]}$ (and their indices). The ranges of the α_r indices are called the bond-dimensions of the MPS and will be denoted D_r. It is obivous that MPS representations of states are not unique. For example inserting the identity matrix $\delta_{ij} = \sum_k G_{ik} G^{-1}_{kj}$, written as the matrix product of a right-invertible matrix G with its right-inverse G^{-1}, between any two neighboring tensors $A^{[r]}$ and $A^{[r+1]}$ in the MPS and absorbing G into $A^{[r]}$ and G^{-1} into $A^{[r+1]}$ will

give a different representation of the same state:



(13)

Here the grey loops around the tensors mark which tensors were decomposed in the previous step or will be contracted in the next step.

1.3 Schmidt decomposition and canonical form

As shown in the previous section MPS are not unique, so the important question when working with MPS is which gauge to pick and how to obtain it. The form required for TEBD to work is the canonical form. This is a very useful form from which the entanglement structure of the state can be efficiently read-off and it can be obtained for any pure state. The canonical form is based on the Schmidt decomposition of vectors in tensor-product-spaces and the Schmidt decomposition, in turn, is a direct consequence of the singular value decomposition of matrices.

1.3.1 Singular value decomposition

The Singular Value Decomposition (SVD) of an $m \times n$ matrix M is a decomposition of the form

$$M = USV^\dagger \quad , \tag{14}$$

where U is an $m \times m$ unitary matrix, V is an $n \times n$ unitary matrix and S is an $m \times n$ real non-negative diagonal matrix. The diagonal entries of S are called the singular values of M. The SVD exists for any matrix. The SVD is not unique, but it is always possible to choose it such that the singular values are in descending order along the diagonal of S, in which case S (but not U and V) is unique. In the rest of this report I will assume the singular values are always sorted in this descending order.
If $m \neq n$ the last columns or rows (depending on whether $m < n$ or $m > n$, respectively) of S will all be zero and therefore the corresponding columns of V or U, respectively, won't contribute in the product $USV^\dagger$. So it is also possible to perform a "compact" SVD, where one discards these columns of U or V: so $M = USV^\dagger$, where U is now an $m \times r$ matrix with orthonormal

columns, S is an $r \times r$ real non-negative diagonal matrix (with diagonal entries in descending order), V is an $n \times r$ matrix with orthonormal columns, and $r = \min(m, n)$.

1.3.2 Schmidt decomposition

Consider Hilbert spaces H_A and H_B of dimension m and n, respectively. A state $|\Psi\rangle \in H = H_A \otimes H_B$ can then be written uniquely in terms of orthonormal bases $\{|e_0\rangle, ..., |e_{m-1}\rangle\}$ and $\{|f_0\rangle, ..., |f_{n-1}\rangle\}$ of H_A and H_B, respectively, as

$$|\Psi\rangle = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} M_{ij} |e_i\rangle \otimes |f_j\rangle \quad . \tag{15}$$

Then, viewing M_{ij} as an $m \times n$ matrix, it is possible to perform a compact SVD $M = USV^\dagger$, with U an $m \times r$ matrix with orthonormal columns, S an $r \times r$ real non-negative diagonal matrix, V an $n \times r$ matrix with orthonormal columns, and $r = \min(m, n)$. Denoting the singular values by λ_k, the column vectors of U by u_k and the complex-conjugated column vectors of V by v_k, this can be written as

$$M = \sum_{k=0}^{r-1} \lambda_k u_k v_k^T \quad . \tag{16}$$

And so, setting

$$|u_k\rangle = \sum_{i=0}^{m-1} (u_k)_i |e_i\rangle \quad ,$$

$$|v_k\rangle = \sum_{j=0}^{n-1} (v_k)_j |f_j\rangle \quad ,$$

it follows from (15) and (16) that

$$|\Psi\rangle = \sum_{k=0}^{r-1} \lambda_k |u_k\rangle \otimes |v_k\rangle \quad . \tag{17}$$

This is called the Schmidt decomposition of $|\Psi\rangle$. The λ_k are called the Schmidt coefficients and the $|u_k\rangle$ and $|v_k\rangle$ are called the left and right Schmidt vectors of $|\Psi\rangle$. The Schmidt coefficients are not necessarily all nonzero so (17) can also be written

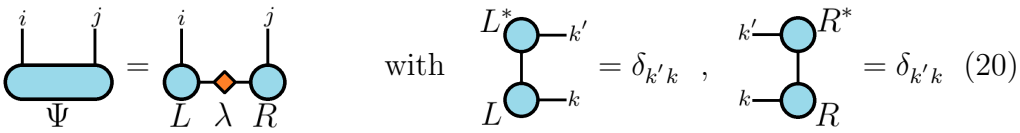
$$|\Psi\rangle = \sum_{k=0}^{\chi-1} \lambda_k |u_k\rangle \otimes |v_k\rangle \quad , \tag{18}$$

where $\chi \leq r$ is the Schmidt rank: the number of nonzero Schmidt coefficients.

In summary: for any state $|\Psi\rangle \in H_A \otimes H_B$ it is possible, using the SVD, to find orthonormal sets $\{|u_0\rangle, ..., |u_{r-1}\rangle\} \subset H_A$ and $\{|v_0\rangle, ..., |v_{r-1}\rangle\} \subset H_B$, where $r = \min(m, n)$, and unique non-negative real numbers $\{\lambda_0, ..., \lambda_{r-1}\}$, which satisfy (18). Writing $|u_k\rangle = \sum_{i=0}^{m-1} L_k^i |i\rangle$ and $|v_k\rangle = \sum_{j=0}^{n-1} R_k^j |j\rangle$ gives:

$$|\Psi\rangle = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \sum_{k=0}^{\chi-1} L_k^i \lambda_k R_k^j |i\rangle \otimes |j\rangle \tag{19}$$

or, in TN diagram notation,



(20)

Here the diamond-shape denotes that λ is diagonal.

The reduced density matrices to either of the subspaces H_A or H_B can be read off directly in diagonal form from the Schmidt decomposition of a pure state. The density matrix of a pure state $|\Psi\rangle$ is simply $\rho = |\Psi\rangle \langle\Psi|$ or, substituting the Schmidt decomposition (18),

$$\rho = \sum_{l=0}^{\chi-1} \sum_{k=0}^{\chi-1} \lambda_l \lambda_k |u_l\rangle \otimes |v_l\rangle \langle u_k| \otimes \langle v_k| \quad . \tag{21}$$

The left Schmidt vectors $|u_k\rangle \in H_A$ are an orthonormal set which can therefore be extended (when $r < m$) to an orthonormal basis of H_A. So the partial trace Tr_A of (21) can be taken in this new basis:

$$\begin{aligned} \mathrm{Tr}_A \rho &= \sum_{j=0}^{m-1} \langle u_j| \left(\sum_{l=0}^{\chi-1} \sum_{k=0}^{\chi-1} \lambda_l \lambda_k |u_l\rangle \otimes |v_l\rangle \langle u_k| \otimes \langle v_k| \right) |u_j\rangle \\ &= \sum_{j=0}^{\chi-1} \lambda_j^2 |v_j\rangle \langle v_j| \quad , \end{aligned} \tag{22}$$

(where the last equality follows from the orthonormality $\langle u_k|u_j\rangle = \delta_{kj}$). The diagonal form of the other reduced density matrix $\rho_A = \mathrm{Tr}_B \rho = \sum_{j=0}^{\chi-1} \lambda_j^2 |u_j\rangle \langle u_j|$ can be obtained in exactly the same way by performing Tr_B in the orthonormal basis of the right Schmidt vectors. It also follows that the von Neumann entanglement entropy (2) of subsystem A or

B can be obtained directly from the Schmidt coefficients as

$$S(A) = -\sum_{j=0}^{\chi-1} \lambda_j^2 \log\left(\lambda_j^2\right) = S(B) \quad . \tag{23}$$

Which proves that $S(A) = S(B)$.

1.3.3 Canonical form of an MPS

Consider a quantum N-body system in a state $|\Psi\rangle \in H_1 \otimes ... \otimes H_N$ with coefficients $\Psi^{i_1 ... i_N}$. Using the Schmidt decomposition it is possible to compute an MPS representation of this state, called the canonical form or Vidal gauge, where the Schmidt coefficients of the decompositions around each site can be read off directly from the tensors, using the following algorithm [2] (a simple worked out example is given in the appendix A.1):
The first step is to compute the Schmidt decomposition between the first site [1] and the rest of the sites [2:N]:

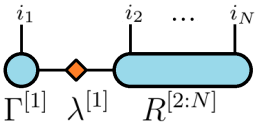
$$|\Psi\rangle = \sum_{\alpha_1=0}^{\chi_1-1} \lambda_{\alpha_1}^{[1]} \left|\tau_{\alpha_1}^{[1]}\right\rangle \otimes \left|\tau_{\alpha_1}^{[2:N]}\right\rangle \quad , \tag{24}$$

where $\lambda_{\alpha_1}^{[1]}$ are the Schmidt coefficients and $\left|\tau_{\alpha_1}^{[1]}\right\rangle \in H_1$ and $\left|\tau_{\alpha_1}^{[2:N]}\right\rangle \in H_{[2:N]} = H_2 \otimes ... \otimes H_N$ are the left and right Schmidt vectors, respectively. Rewriting the left Schmidt vectors as

$$\left|\tau_{\alpha_1}^{[1]}\right\rangle = \sum_{i_1=0}^{d_1-1} \Gamma_{\alpha_1}^{[1]i_1} |i_1\rangle \quad , \tag{25}$$

in the local basis of site 1, and substituting this in (24) gives

$$|\Psi\rangle = \sum_{i_1=0}^{d_1-1} \sum_{\alpha_1=0}^{\chi_1-1} \Gamma_{\alpha_1}^{[1]i_1} \lambda_{\alpha_1}^{[1]} |i_1\rangle \otimes \left|\tau_{\alpha_1}^{[2:N]}\right\rangle =$$



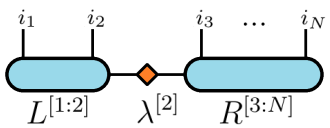
. (26)

The second step is to write

$$\left|\tau_{\alpha_1}^{[2:N]}\right\rangle = \sum_{i_2=0}^{d_2-1} |i_2\rangle \otimes \left|\omega_{\alpha_1}^{[3:N]i_2}\right\rangle \tag{27}$$

and perform the Schmidt decomposition between [1:2] and [3:N]:

$$|\Psi\rangle = \sum_{\alpha_2=0}^{\chi_2-1} \lambda_{\alpha_2}^{[2]} \left|\tau_{\alpha_2}^{[1:2]}\right\rangle \otimes \left|\tau_{\alpha_2}^{[3:N]}\right\rangle =$$

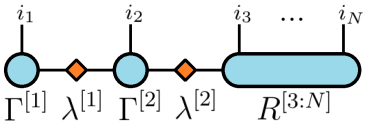


. (28)

Then you can write the $\left|\omega_{\alpha_1}^{[3:N]i_2}\right\rangle$ out in terms of these right Schmidt vectors and coefficients $\lambda_{\alpha_2}^{[2]}\left|\tau_{\alpha_2}^{[3:N]}\right\rangle$ as

$$\left|\omega_{\alpha_1}^{[3:N]i_2}\right\rangle = \sum_{\alpha_2=0}^{\chi_2-1} \Gamma_{\alpha_1\alpha_2}^{[2]i_2}\lambda_{\alpha_2}^{[2]}\left|\tau_{\alpha_2}^{[3:N]}\right\rangle \quad . \tag{29}$$

Plugging this into (27) and that into (26) gives

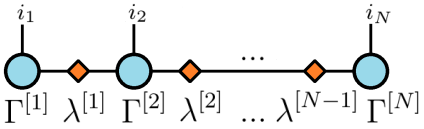
$|\Psi\rangle =$ 

(30)

Iterating the second step (equations (27), (28), (29) and (30)) for all other subsystem bipartitions of the form ([1:r],[$r+1$:N]) until ([1:$N-1$],[N]) and then writing the last right Schmidt vectors as

$$\left|\tau_{\alpha_{N-1}}^{[N]}\right\rangle = \sum_{i_N=0}^{d_N-1} \Gamma_{\alpha_{N-1}}^{[N]i_N}\left|i_N\right\rangle \quad , \tag{31}$$

gives the following MPS representation:

$$\begin{aligned}|\Psi\rangle &= \sum_{i_1,\ldots,i_N}\sum_{\alpha_1,\ldots,\alpha_{N-1}} \Gamma_{\alpha_1}^{[1]i_1}\lambda_{\alpha_1}^{[1]}\Gamma_{\alpha_1\alpha_2}^{[2]i_2}\lambda_{\alpha_2}^{[2]}\ldots\lambda_{\alpha_{N-1}}^{[N-1]}\Gamma_{\alpha_{N-1}}^{[N]i_N}\left|i_1\right\rangle\otimes\left|i_2\right\rangle\otimes\ldots\otimes\left|i_N\right\rangle \\ &= \quad \text{  } \quad .\end{aligned}$$

(32)

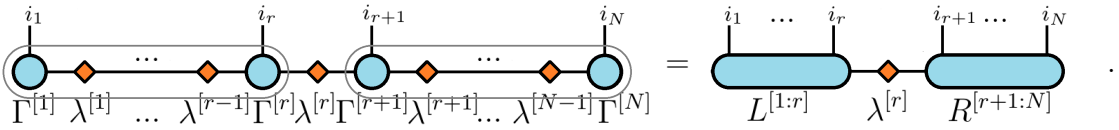
By construction, the Schmidt decomposition for any bipartition of the form ([1:r],[$r+1$:N]) is easily obtained from the canonical form (32): the Schmidt coefficients are the entries of $\lambda^{[r]}$, the left Schmidt vectors $\left|\tau_{\alpha_r}^{[1:r]}\right\rangle$ are the contractions of all tensors to the left of $\lambda^{[r]}$, and the right Schmidt vectors $\left|\tau_{\alpha_r}^{[r+1:N]}\right\rangle$ are the contractions of all tensors to the right of $\lambda^{[r]}$. Explicitly:

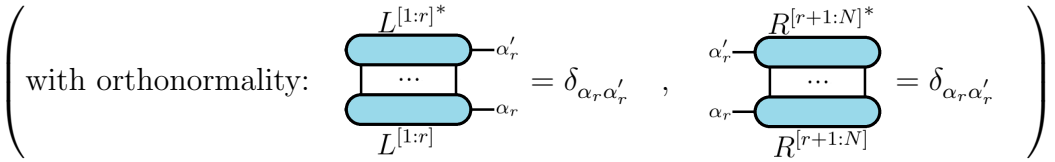
$$\begin{aligned}\left|\tau_{\alpha_r}^{[1:r]}\right\rangle &= \sum_{i_1,\ldots,i_r}\sum_{\alpha_1,\ldots,\alpha_{r-1}} \Gamma_{\alpha_1}^{[1]i_1}\lambda_{\alpha_1}^{[1]}\ldots\lambda_{\alpha_{r-1}}^{[r-1]}\Gamma_{\alpha_{r-1}\alpha_r}^{[r]i_r}\left|i_1\right\rangle\otimes\ldots\otimes\left|i_r\right\rangle \\ \left|\tau_{\alpha_r}^{[r+1:N]}\right\rangle &= \sum_{i_{r+1},\ldots,i_N}\sum_{\alpha_{r+1},\ldots,\alpha_N} \Gamma_{\alpha_r\alpha_{r+1}}^{[r+1]i_{r+1}}\lambda_{\alpha_{r+1}}^{[r+1]}\ldots\lambda_{\alpha_{N-1}}^{[N-1]}\Gamma_{\alpha_{N-1}}^{[N]i_N}\left|i_{r+1}\right\rangle\otimes\ldots\otimes\left|i_N\right\rangle\end{aligned}$$

gives the Schmidt decomposition

$$|\Psi\rangle = \sum_{\alpha_r=0}^{\chi_r-1} \lambda^{[r]}_{\alpha_r} \left|\tau^{[1:r]}_{\alpha_r}\right\rangle \otimes \left|\tau^{[r+1:N]}_{\alpha_r}\right\rangle \quad . \tag{33}$$

Or in TN diagram notation:

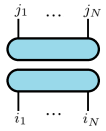




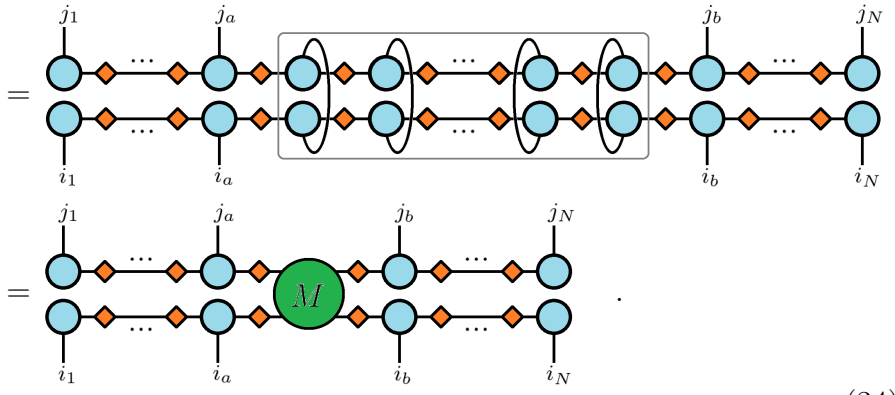
This property is also taken as the definition of canonical form. The canonical form is still not fully unique, because Schmidt vectors aren't unique and so the Γ-tensors are not unique, but it is much more specific than general MPS and (33) is what allows algorithms such as TEBD to work.
This canonical MPS is an exact representation of $|\Psi\rangle$ and the bond-dimensions $D_r = \chi_r$ are equal to the Schmidt ranks of the decompositions across ($[1{:}r]$,$[r{+}1{:}N]$). This means that the maximal bond-dimension $D = \max\{D_r | r = 1, ..., N\}$ can still be exponential in the system size (e.g. assuming all $\chi_r = d_r = d$ are equal and that N is even, then D is the bond-dimension of the central bond $D_{N/2} = d_{[1:N/2]} = d^{N/2}$). But it is known that low-energy states of systems with gapped one-dimensional Hamiltonians (such as the Ising Hamiltonian (49)) can be approximated arbitrarily accurately by MPS with a bond-dimension D that is polynomial in the system size. Fixing a maximum bond-dimension and discarding the Schmidt coefficients which exceed it or discarding all Schmidt coefficients smaller than a given threshold λ_c, can already significantly reduce size without significant loss of precision in such (low entanglement) states.

1.3.4 Mutual information in canonical form

In section 3.3 I will study the mutual information (3) between two intervals $A = [1 : a]$ and $B = [b : N]$ at the ends of the Ising spin-chain. Given the MPS in canonical form, both $S([1 : a])$ and $S([b : N])$ can be obtained directly from $\lambda^{[a]}$ and $\lambda^{[b-1]}$ with (23) but $S(A \cup B) = S([1 : a] \cup [b : N])$ requires a different method: representing the outer product $|\Psi\rangle\langle\Psi|$ in a TN

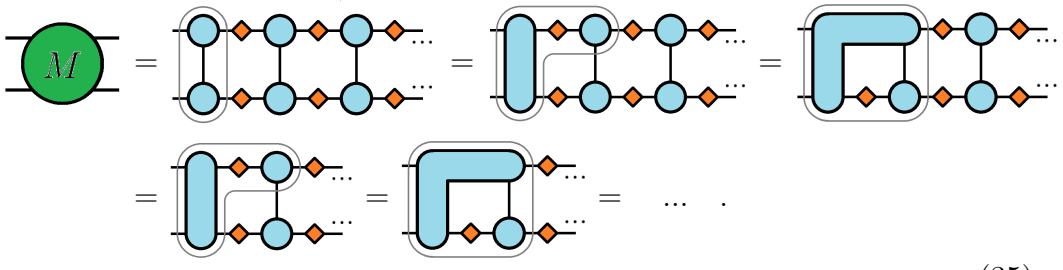
diagram by placing the ket network right above the bra network  , the partial trace $\mathrm{Tr}_{[a+1:b-1]}$ is given by the following diagram

$$\rho_{[1:a]\cup[b:N]} = \mathrm{Tr}_{[a+1:b-1]}(|\Psi\rangle\langle\Psi|)$$



(34)

The canonical Schmidt decomposition property (33) is only implied for continuous intervals containing either of the ends of the chain so (although it is possible, for example, for a product-state) M is not necessarily an identity-matrix (or diagonal matrix) and (34) cannot be used to directly obtain the eigenvalues of the reduced density matrix. However M can be computed efficiently and it is then possible to contract (34) and numerically diagonalize the result. In order to efficiently contract all the tensors in the middle into M this must be done in the right order: in order to minimize the dimensions of the contractions in each step and minimize the total dimension of the open indices in each contraction. This can be done by starting at the end ($r = a + 1$ or $r = b - 1$) and alternately contracting the next Γ and Γ^* tensors with the previous result:



(35)

1.4 Time-evolving block decimation

Time-evolving Block Decimation (TEBD) is an approximate iterative method to simulate time-evolution of one-dimensional lattice quantum systems with nearest-neighbor interactions, which takes advantage of and preserves the canonical form. In this project I will be studying the Ising model with Hamiltonian (49). This Hamiltonian contains only 1-site terms and 2-site nearest-neighbor terms. It is then possible to absorb the 1-site terms in the 2-site terms and write the Hamiltonian in the form

$$\hat{H} = \sum_{r=1}^{L-1} h^{[r:r+1]} \quad , \tag{36}$$

where all $h^{[r:r+1]}$ are 2-site Hermitian operators each acting only on sites r and $r+1$.

The time-evolution operator $e^{-\frac{i}{\hbar}t\hat{H}}$ requires computing the exponential of this sum, but any two adjacent terms $h^{[r-1:r]}$ and $h^{[r:r+1]}$ don't commute (and for non-commuting operators it is no longer true that the exponential of the sum is conveniently equal to the product of the exponentials). However it is possible to approximate the exponential using the Trotter decomposition:

$$e^{(\hat{A}+\hat{B})t} = \left(e^{\hat{A}\frac{t}{n}}e^{\hat{B}\frac{t}{n}}\right)^n + \mathrm{O}\left(\left(\frac{t}{n}\right)^2\right) \quad . \tag{37}$$

For a nearest-neighbor Hamiltonian (36) one can write

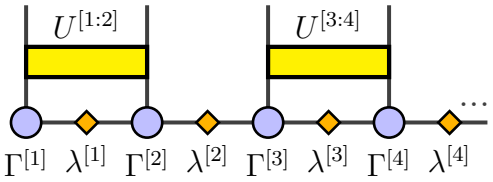
$$\begin{aligned} \hat{H} &= \hat{H}^{[\text{even}]} + \hat{H}^{[\text{odd}]} \\ \hat{H}^{[\text{even}]} &= \sum_{r \text{ even}}^{L-1} h^{[r:r+1]} \\ \hat{H}^{[\text{odd}]} &= \sum_{r \text{ odd}}^{L-1} h^{[r:r+1]} \end{aligned} \tag{38}$$

Then $\hat{H}^{[\text{odd}]}$ and $\hat{H}^{[\text{even}]}$ don't commute but all terms in $\hat{H}^{[\text{odd}]}$ act on disjoint subspaces and therefore commute among each other, and similarly all terms in $\hat{H}^{[\text{even}]}$ commute with each other. This means that the following equalities are exact

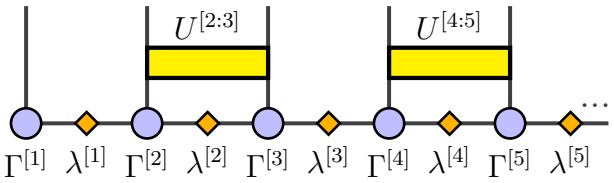
$$\begin{aligned} e^{-\frac{i}{\hbar}\frac{t}{n}\hat{H}^{[\text{even}]}} &= \prod_{r \text{ even}}^{L-1} e^{-\frac{i}{\hbar}\frac{t}{n}h^{[r:r+1]}} \\ e^{-\frac{i}{\hbar}\frac{t}{n}\hat{H}^{[\text{odd}]}} &= \prod_{r \text{ odd}}^{L-1} e^{-\frac{i}{\hbar}\frac{t}{n}h^{[r:r+1]}} \end{aligned} \tag{39}$$

Defining $U^{[r:r+1]} = e^{-\frac{i}{\hbar}\frac{t}{n}h^{[r:r+1]}}$, for all $r = 1, ..., L-1$, these are all nearest-neighbor 2-site unitary gates, and the action of $e^{-\frac{i}{\hbar}\frac{t}{n}\hat{H}^{[\text{even}]}}$ and $e^{-\frac{i}{\hbar}\frac{t}{n}\hat{H}^{[\text{odd}]}}$ on a canonical MPS of $|\Psi\rangle$ can be represented:

$e^{-\frac{i}{\hbar}\frac{t}{n}\hat{H}^{[\text{odd}]}}|\Psi\rangle =$



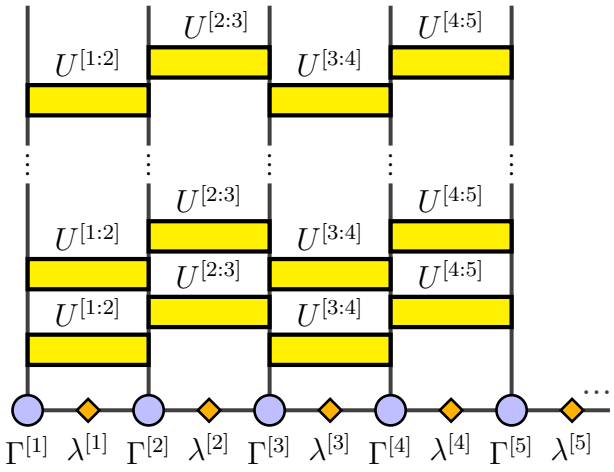
$e^{-\frac{i}{\hbar}\frac{t}{n}\hat{H}^{[\text{even}]}}|\Psi\rangle =$



(40)

The full time-evolution operator $e^{-\frac{i}{\hbar}t\hat{H}}$ can then be approximated (with error $\propto \left(\frac{t}{n}\right)^2$) in terms of the gates, using the Trotter-decomposition (37) for $\hat{H} = \hat{H}^{[\text{odd}]} + \hat{H}^{[\text{even}]}$, by n times alternately applying $e^{-\frac{i}{\hbar}\frac{t}{n}\hat{H}^{[\text{even}]}}$ and $e^{-\frac{i}{\hbar}\frac{t}{n}\hat{H}^{[\text{odd}]}}$ to the state, as illustrated in the following TN diagram:

$e^{-\frac{i}{\hbar}t\hat{H}}|\Psi\rangle \approx$

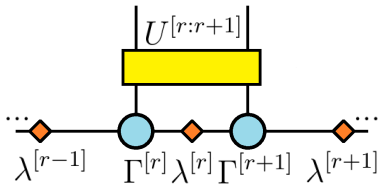


(41)

This is the principle of TEBD: reduce the time-evolution to the iterative application of local unitary gates, with time-step $\delta t = \frac{t}{n}$. The crucial point is that these nearest-neighbor 2-site unitary gates can easily be applied to an MPS in canonical form while preserving the canonical form. Consider a

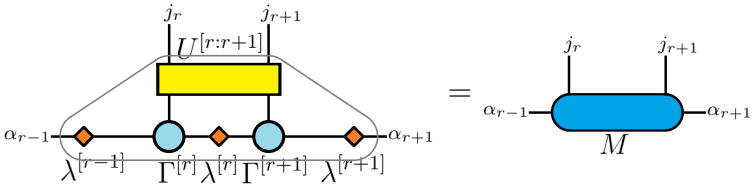
canonical MPS (32) and an arbitrary 2-site unitary gate $U^{[r:r+1]}$ acting on sites r and $r+1$:

$$U^{[r:r+1]}\,|\Psi\rangle = \quad .$$



(42)

Because unitary transformations preserve orthonormality it is not necessary to re-update all tensors in the MPS to ensure (33) will still hold: for any bipartition where r and $r+1$ are contained in the same subsystem, (33) will still be true for the result because $U^{[r:r+1]}$ is then a unitary transformation of the left or right (depending on which subsystem) Schmidt vectors mapping them to a new orthonormal set of Schmidt vectors of the resulting state. So only $\Gamma^{[r]}$, $\lambda^{[r]}$ and $\Gamma^{[r+1]}$ have to be updated in a way so that (33) is valid for the bipartion around r. Contracting the tensor-representation $U^{[r:r+1]j_r j_{r+1}}_{i_r i_{r+1}}$ of $U^{[r:r+1]}$ with $\lambda^{[r-1]}_{\alpha_{r-1}}$, $\Gamma^{[r]i_r}_{\alpha_{r-1}\alpha_r}$, $\lambda^{[r]}_{\alpha_r}$, $\Gamma^{[r+1]i_{r+1}}_{\alpha_r\alpha_{r+1}}$ and $\lambda^{[r+1]}_{\alpha_{r+1}}$ into a single tensor $M^{j_r j_{r+1}}_{\alpha_{r-1}\alpha_{r+1}}$:



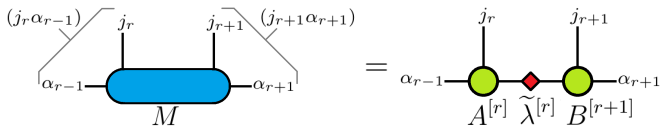
(43)

and using (33) for bonds $r-1$ and $r+1$ gives

$$U^{[r:r+1]}\,|\Psi\rangle = \sum_{\alpha_{r-1}\alpha_{r+1}} \sum_{j_r j_{r+1}} M^{j_r j_{r+1}}_{\alpha_{r-1}\alpha_{r+1}} \left|\tau^{[1:r-1]}_{\alpha_{r-1}}\right\rangle \otimes |i_r\rangle \otimes |i_{r+1}\rangle \otimes \left|\tau^{[r+2:N]}_{\alpha_{r+1}}\right\rangle \quad . \tag{44}$$

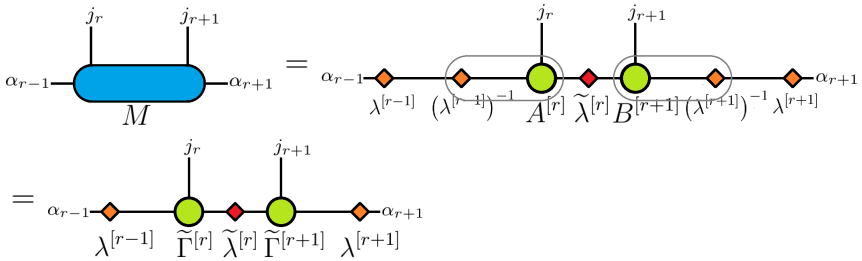
The Schmidt vectors $\left|\tau^{[1:r-1]}_{\alpha_{r-1}}\right\rangle$ and $\left|\tau^{[r+2:N]}_{\alpha_{r+1}}\right\rangle$ and their tensor products with $|i_r\rangle$ and $|i_{r+1}\rangle$ are all orthonormal so M is the coefficient tensor of the resulting state in this orthonormal basis. The Schmidt decomposition across $([1:r],[r+1:N])$ can now be computed from M using a SVD by temporarily merging the indices $(j_r\alpha_{r-1})$ and $(j_{r+1}\alpha_{r+1})$:

$$M^{j_r j_{r+1}}_{\alpha_{r-1}\alpha_{r+1}} = A^{[r]j_r}_{\alpha_{r-1}\alpha_r} \widetilde{\lambda}^{[r]}_{\alpha_r} B^{[r+1]j_{r+1}}_{\alpha_r\alpha_{r+1}} \tag{45}$$



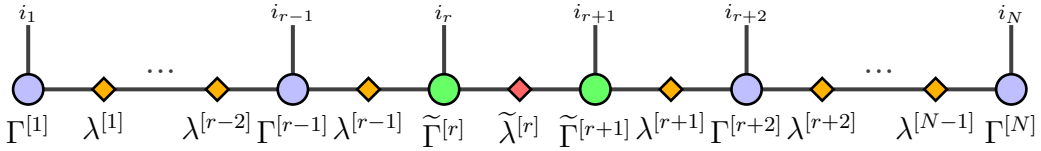
(46)

This then ensures that (33) will hold around site r. By inserting $\lambda^{[r-1]}_{\alpha_{r-1}} \left(\lambda^{[r-1]}_{\alpha_{r-1}}\right)^{-1}$ and $\left(\lambda^{[r+1]}_{\alpha_{r+1}}\right)^{-1} \lambda^{[r+1]}_{\alpha_{r+1}}$ on the sides and absorbing the inverses into A and B these Schmidt coefficients are brought back:



Inserting this new representation of M back into (44) and replacing the Schmidt vectors by their original canonical MPS tensors gives

$U^{[r:r+1]} |\Psi\rangle =$



(47)

in canonical form. This shows that the unitary gates in each even or odd step of (41) can be applied independently in parallel (simply compute the new $\widetilde{\Gamma}^{[r]j_r}_{\alpha_{r-1}\alpha_r}$, $\widetilde{\lambda}^{[r]}_{\alpha_r}$ and $\widetilde{\Gamma}^{[r+1]j_{r+1}}_{\alpha_r\alpha_{r+1}}$ from $\Gamma^{[r]}$, $\lambda^{[r]}$ and $\Gamma^{[r+1]}$ and replace them) while preserving canonical form.

Instead of performing the entire process (41) again for each time t for which a result is desired, one can of course choose a small enough time-step $\delta t = \frac{t}{n}$ and perform the evolution once up to the maximal required time with this time-step and analyze the state at each desired point during this evolution. If the bond-dimensions of the MPS aren't truncated at any point, the only error is that of the Trotter decomposition (37), $\propto (\delta t)^2$, and TEBD becomes exact in the limit $\delta t \to 0$.

1.5 The Ising model

The Ising model is a simple model of a one-dimensional spin-$\frac{1}{2}$ chain with only nearest-neighbor interactions, which can show both integrable and chaotic behaviour depending on values of the free parameters in the Hamiltonian. For a spin chain of length L with open ends the mixed-field Ising Hamiltonian

is usually defined as

$$\hat{H} = \sum_{i=1}^{L} g\sigma_x^{[i]} + \sum_{i=1}^{L} h\sigma_z^{[i]} + \sum_{i=1}^{L-1} J\sigma_z^{[i]}\sigma_z^{[i+1]} \quad , \tag{48}$$

where σ_x, σ_y and σ_z are the Pauli matrices, J is the strength of the interaction between neighboring spins, h is the strength of the longitudinal magnetic field and g is the strength of the transverse magnetic field. Following [5], I will consider the following, slightly modified, Hamiltonian

$$\hat{H} = \sum_{i=1}^{L} g\sigma_x^{[i]} + \sum_{i=2}^{L-1} h\sigma_z^{[i]} + (h-J)(\sigma_z^{[1]} + \sigma_z^{[L]}) + \sum_{i=1}^{L-1} J\sigma_z^{[i]}\sigma_z^{[i+1]} \quad , \tag{49}$$

where the longitudinal fields at the ends of the chain are reduced by J to keep the ends more similar to rest of the chain and avoid edge-effects.

Depending on the values of h and g the Ising model can be integrable or nonintegrable. With either of the fields off ($h = 0$ or $g = 0$) the model is integrable and analytic results for the entanglement entropy are obtainable. Examples of nonintegrable points I will use are

$$(h, g, J) = (\frac{\sqrt{5}+1}{4}, \frac{\sqrt{5}+5}{8}, 1) = (0.8090..., 0.9045..., 1) \quad , \tag{50}$$

which is used in [5], and

$$(h, g, J) = (0.5, -1.05, 1) \tag{51}$$

which is highly chaotic [6].

1.6 Quantum quench and the Néel state

A common way to bring a quantum system out of equilibrium is the quantum quench: when a quantum system is prepared in an eigenstate of a Hamiltonian $\hat{H}_0$ and then evolved in time under a different Hamiltonian $\hat{H}_1$, this is called a quantum quench (often also called a sudden quench, to distinguish from a slow quench when the Hamiltonian is continuously changed from $\hat{H}_0$ to $\hat{H}_1$). An often studied question, very important in quantum informatics, is the creation and spreading of entanglement following a quench from a non-entangled product state. This is also what I will be studying in this project.

A simple product state which I will use as initial state for a quench for most of my results is the Néel state, defined as

$$|\Psi_N\rangle = |0\rangle \otimes |1\rangle \otimes |0\rangle \otimes |1\rangle \otimes ... \quad , \tag{52}$$

with every spin pointing in the opposite direction from its neighbors. The Néel state is an eigenstate of the Ising Hamiltonian when the transverse field $g = 0$ is off, but otherwise using it as initial state and evolving the system with the Ising Hamiltonian is an example of a quench.

The Néel state is a product state so all Schmidt decompositions have only one nonzero Schmidt coefficient

$$\lambda_0^{[r]} = 1 \tag{53}$$

and $D_r = \chi_r = 1$, for all r. In the simplest canonical form the only nonzero entries of the $\Gamma^{[r]}$ tensors are

$$\begin{aligned} \Gamma_{0,0}^{[r]0} &= s \qquad \text{if} \quad r \in 2\mathbb{N}+1 \\ \Gamma_{0,0}^{[r]1} &= s \qquad \text{if} \quad r \in 2\mathbb{N} \end{aligned} \tag{54}$$

where $s = 1$. An equivalent canonical form, which is the output of my numerical implementation of the algorithm to compute the canonical form for even L, is obtained by setting

$$s = \begin{cases} -1 & \text{if} \quad r \in \left\{L, \frac{L}{2}+1\right\} \\ +1 & \text{otherwise} \end{cases} \tag{55}$$

in the previous equation (the two minus-signs will just cancel in the contraction).

2 Numerical implementation

I wrote a program in the Python programming language for working with canonical MPS, implementing TEBD and computing the entanglement entropy for different subsystems. The NumPy library was used to work with the tensors as NumPy-arrays and for linear algebra such as the SVD. For multiprocessing I used the built-in multiprocessing capabilities of NumPy (compiled with BLAS), to perform parts of the contractions and linear algebra in parallel, and the Python multiprocessing library, to perform the TEBD steps in parallel. The plots of the results were made using the Matplotlib library.

Since the required computational resources grow exponentially (without truncation) with the length of the chain, most of the computations for longer chains were performed on my uncle's powerful computer [1] and on the VUB Hydra cluster [2].

[1] It has a 16-core AMD Ryzen 9 3950X CPU and 64GB 2666MHz ECC RAM.

[2] Most of my computations with Hydra ran on 20-core Intel Xeon Gold 6148 CPUs. (Different amounts of RAM can be requested on Hydra).

The tensor network diagrams used in the previous section are a very convenient and elegant way to explain all the theory and algorithms in an understandable way. However tensor network diagrams are a very abstract construct not always trivially directly translatable into efficient Python code (especially in cases such as implicit definitions). In the following subsections I will therefore briefly go over the most important considerations when implementing the algorithms described in the previous section in my program. (My code for these algorithms is shown in the appendix A.2). Here I will separate the indices of tensors by commas in order to avoid confusion with multiplication of their values.

2.1 Schmidt decomposition and canonical form

Computing the Schmidt decomposition of a coefficient tensor $\Psi^{i_1,\dots,i_N}$ across a bipartition $([1:r],[r+1:N])$ is simply a matter of reshaping the tensor into a matrix $M_{ij} = \Psi^{(i_1,\dots,i_r)(i_{r+1},\dots,i_N)}$ by merging the proper indices $i = (i_1,\dots,i_r)$ and $j = (i_{r+1},\dots,i_N)$ and then applying the compact SVD to this matrix: $M = USV^\dagger$. The Schmidt coefficients are then $\lambda_k = S_{k,k}$ and the left an right Schmidt vector tensors L and R are then given by $L^i_k = U_{k,i}$ and $R^j_k = V^\dagger_{j,k}$. For numerically computing the compact SVD I used the NumPy function `numpy.linalg.svd()`, and the reshaping was done using the functionality of the NumPy arrays. I also added optional parameters for a maximal bond-dimension D or cutoff value λ_c for the Schmidt coefficients, to discard all Schmidt vectors and coefficients with $k \geq D$ or $\lambda_k < \lambda_c$, respectively. The default value $\lambda_c = 10^{-16}$ (instead of 0) for the cutoff value was chosen because this is orders of magnitude less than the numerical accuracy so singular values less than 10^{-16} would likely be equal to zero if calculated exactly (this was verified experimentally with a few examples) and the error due to mistakenly considering singular values less than 10^{-16} as being equal to zero is negligible compared to other inevitable numerical errors. This Schmidt decomposition function was then used to compute all Schmidt decompositions in the rest of my program.

The algorithm for finding the canonical form explained in 1.3.3 contains a few implicit definitions which have to be translated into explicit instructions in order to be implemented numerically. The first step is to compute the Schmidt decomposition

$$\Psi^{i_1,\dots,i_N} = \sum_{\alpha_1} L^{[1]i_1}_{\alpha_1} \lambda^{[1]}_{\alpha_1} R^{[2:N]i_2,\dots,i_N}_{\alpha_1} \quad ,$$

which is easily done using the function defined above. The definition (25) of $\Gamma^{[1]}$ is then simply the change-of-basis matrix from the left Schmidt vectors

to the computational basis $|i_1\rangle$ so

$$\Gamma^{[1]i_1}_{\alpha_1} = L^{[1]i_1}_{\alpha_1} \quad . \tag{56}$$

Writing

$$\left|\omega^{[3:N]i_2}_{\alpha_1}\right\rangle = \sum_j W^{[3:N]i_2}_{\alpha_1,j} \, |j\rangle$$

in the computational basis $|j\rangle = |i_3\rangle \otimes ... \otimes |i_N\rangle$, with $j = (i_3, ..., i_N)$, equation (27) becomes

$$R^{[2:N](\cdot)}_{\alpha_1} = \sum_{i_2} |i_2\rangle \otimes W^{[3:N]i_2}_{\alpha_1,(\cdot)} = \begin{pmatrix} W^{[3:N]0}_{\alpha_1,(\cdot)} \\ \vdots \\ W^{[3:N]d_{[2]}-1}_{\alpha_1,(\cdot)} \end{pmatrix} \quad . \tag{57}$$

So, for each value of i_2 and α_1, $W^{[3:N]i_2}_{\alpha_1,(\cdot)}$ can be read-off directly as the entries from $d_{[3:N]}i_2$ to $d_{[3:N]}(i_2+1)$ (for the values in $(\cdot)$) of the column-vector $R^{[2:N](\cdot)}_{\alpha_1}$. The next Schmidt decomposition is

$$\Psi^{i_1,...,i_N} = \sum_{\alpha_2} L^{[1:2]i_1,i_2}_{\alpha_2} \lambda^{[2]}_{\alpha_2} R^{[3:N]i_3,...,i_N}_{\alpha_2} \quad .$$

Equation (29) defines the vector $\Gamma^{[2]i_2}_{\alpha_1,(\cdot)}$ as the representation of $W^{[3:N]i_2}_{\alpha_1,(\cdot)}$ in the basis $\{\lambda^{[2]}_{\alpha_2} \left|\tau^{[3:N]}_{\alpha_2}\right\rangle\}$. The change-of-basis matrix from $|i_3\rangle \otimes ... \otimes |i_N\rangle$ to $\{\left|\tau^{[3:N]}_{\alpha_2}\right\rangle\}$ is then $\left(R^{[3:N](i_3,...,i_N)}_{\alpha_2}\right)^\dagger$ so

$$\Gamma^{[2]i_2}_{\alpha_1,\alpha_2} = \sum_{(i_3,...,i_N)} \left(R^{[3:N](i_3,...,i_N)}_{\alpha_2}\right)^* W^{[3:N]i_2}_{\alpha_1,(i_3,...,i_N)} (\lambda^{[2]}_{\alpha_1})^{-1} \quad . \tag{58}$$

Iterating these steps for all other bipartitions of the form $([1:r],[r+1:N])$ and finally setting

$$\Gamma^{[N]i_N}_{\alpha_{N-1}} = R^{[N-1:N]i_N}_{\alpha_{N-1}} \tag{59}$$

then gives the canonical form. This shows that the algorithm can be fully implemented using only (besides the Schmidt decomposition, discussed above) matrix multiplications, reshaping, and index slicing operations, all of which can be performed efficiently with the NumPy arrays.

2.2 Entanglement entropy and mutual information

Computing the entanglement entropy for a bipartition of the form $([1:r],[r+1:N])$ from the $\lambda^{[r]}$-tensor using (23) is trivial, but for the more complicated case described in 1.3.4 efficient implementation is crucial: when computing the reduced density matrix $\rho_{A\cup B}$ with $A=[1:a]$ and $B=[b:N]$ using an inefficient method to compute the central M-tensor can easily use far too much time or memory. (Also because this is inevitably the algorithm that will have the highest memory usage of the entire program). The most efficient strategy I found was to work out and code the most efficient contraction order myself and use the efficient NumPy `numpy.tensordot()` function to perform the individual contractions. (The issue of minimizing the memory-usage without losing time is also how I found the (35) order). So I wrote a function implementing (35) and then contracting M with the ends (34) to compute $\rho_{A\cup B}$. This matrix was then be diagonalised with NumPy's `numpy.linalg.eigh()` function, and the eigenvalues used to compute $S(A\cup B)$ with (2).

2.3 TEBD for the Ising model

In order to implement TEBD it is first necessary to bring the Ising Hamiltonian (49) into 2-site nearest-neighbor form (36). Each 1-site term $g\sigma_x^{[r]}+h\sigma_z^{[r]}$ in the Hamiltonian can either be absorbed into $h^{[r:r+1]}$ or $h^{[r-1:r]}$. Although all possible ways of absorbing the 1-site terms into the 2-site terms should give exactly the same results (except for slightly different errors in the Trotter decomposition), some might be numerically more stable or efficient. I considered two options: absorbing the 1-site $[r]$-terms into $h^{[r:r+1]}$

$$\begin{aligned} h^{[1:2]} &= g\sigma_x^{[1]}+(h-J)\sigma_z^{[1]}+J\sigma_z^{[1]}\sigma_z^{[2]} \\ h^{[L-1:L]} &= g\sigma_x^{[L-1]}+h\sigma_z^{[L-1]}+J\sigma_z^{[L-1]}\sigma_z^{[L]}+g\sigma_x^{[L]}+(h-J)\sigma_z^{[L]} \\ h^{[r:r+1]} &= g\sigma_x^{[r]}+h\sigma_z^{[r]}+J\sigma_z^{[r]}\sigma_z^{[r+1]} \qquad \text{otherwise} \end{aligned} \tag{60}$$

or absorbing all 1-site terms into the $h^{[r:r+1]}$ with odd r

$$\begin{aligned} h^{[1:2]} &= g\sigma_x^{[1]}+(h-J)\sigma_z^{[1]}+J\sigma_z^{[1]}\sigma_z^{[2]}+g\sigma_x^{[i]}+h\sigma_z^{[i]} \\ h^{[L-1:L]} &= \begin{cases} g\sigma_x^{[L-1]}+h\sigma_z^{[L-1]}+J\sigma_z^{[L-1]}\sigma_z^{[L]}+g\sigma_x^{[L]}+(h-J)\sigma_z^{[L]} & \text{if } L \text{ even} \\ J\sigma_z^{[L-1]}\sigma_z^{[L]}+g\sigma_x^{[L]}+(h-J)\sigma_z^{[L]} & \text{if } L \text{ odd} \end{cases} \\ h^{[r:r+1]} &= g\sigma_x^{[r]}+h\sigma_z^{[r]}+J\sigma_z^{[r]}\sigma_z^{[r+1]}+g\sigma_x^{[r+1]}+h\sigma_z^{[r+1]} \qquad r \text{ odd} \\ h^{[r:r+1]} &= J\sigma_z^{[r]}\sigma_z^{[r+1]} \qquad r \text{ even} \end{aligned} \tag{61}$$

The first form has the advantage that all $h^{[r:r+1]}$ (except the edges) are identical, and the second form has the advantage that half (except the edges) of the $h^{[r:r+1]}$ are diagonal. I tried running TEBD with both forms and they produced exactly the same results, as expected, but the second form was slightly faster so I used (61) for the rest of my results in section 3. Since the Hamiltonian is time-independent, these $h^{[r:r+1]}$ and their exponentials $U^{[r:r+1]} = e^{-\frac{i}{\hbar}\delta t\ h^{[r:r+1]}}$ can be computed from the parameters and the time-step δt at the beginning of the computation and can then be reused in all TEBD steps. In order to avoid potential numerical errors because of very small floating-point numbers I used $\hbar = 1$ (this was also used in [5, 8]). As mentioned in 1.4 all gates can be applied in parallel at each even or odd step of TEBD, so I implemented an option in my program to create a thread-pool with a certain amount of threads with the Python multiprocessing library at the beginning of the simulation and reuse these threads to apply gates in parallel at each odd/even TEBD step.

As discussed in the previous section it is crucial to perform the tensor contractions efficiently and in the right order to minimize memory- and time-usage (although here in the TEBD algorithm the main performance bottleneck was the large SVD's). Like in the previous section, this was done by finding and coding the right order of contractions myself and using `numpy.tensordot()` for the individual contractions.

3 Results

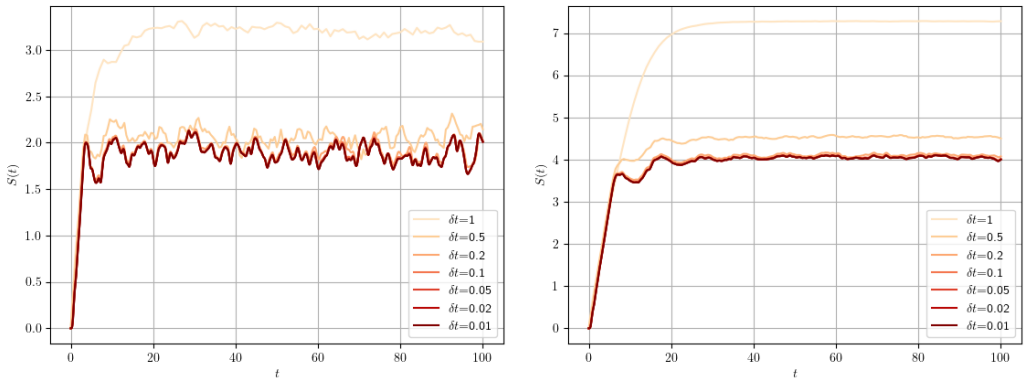
All results presented here are for Ising spin chains with Hamiltonian (49), for different lengths L and different values of h and g, and were computed with my program.

3.1 Evolution of entanglement in Néel state

I used the Néel state $|\Psi\rangle = |0\rangle \otimes |1\rangle \otimes |0\rangle \otimes ...$ (52) as initial state for many of the simulations, and I always used the canonical form (54) with (55) (because it is the output of my program to compute the canonical form). (As a verification of my program a few of the simulations were repeated with the first canonical form (54) with $s = 1$ as initial state and the results were found to be exactly the same, as expected).

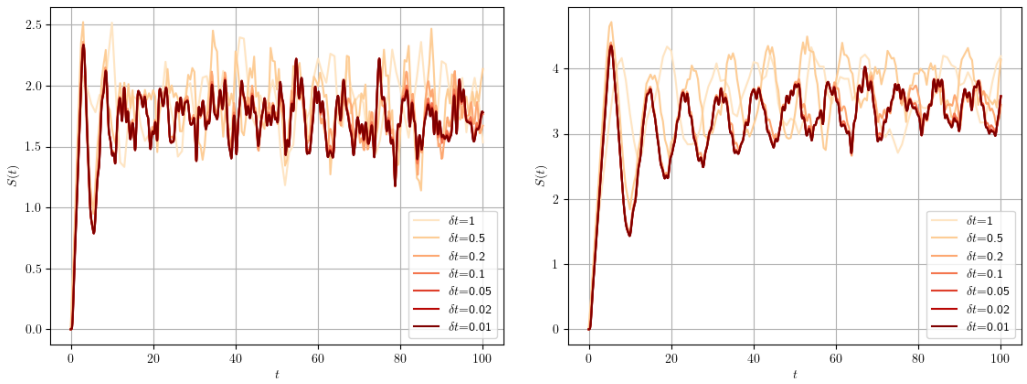
3.1.1 Finding the right time-step

In order to find out which time-step δt to use for the TEBD I ran some simulations, all starting from the Néel state of length $L = 8$ or 16, with different



(a) $L = 8$ (b) $L = 16$

Figure 2: Time-evolution of $S([1 : L/2])$ in spin chains with nonintegrable $(h, g, J) = (\frac{\sqrt{5}+1}{4}, \frac{\sqrt{5}+5}{8}, 1) = (0.8090..., 0.9045..., 1)$

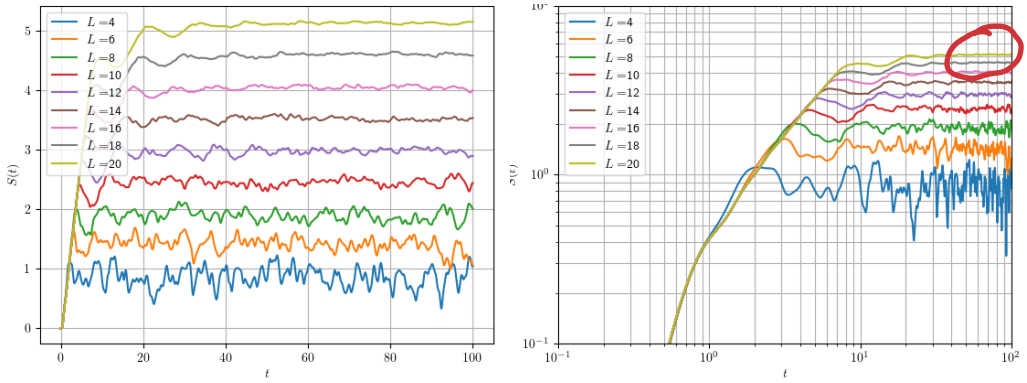


(a) $L = 8$ (b) $L = 16$

Figure 3: Time-evolution of $S([1 : L/2])$ in spin chains with integrable $(h, g, J) = (0, 1, 1)$

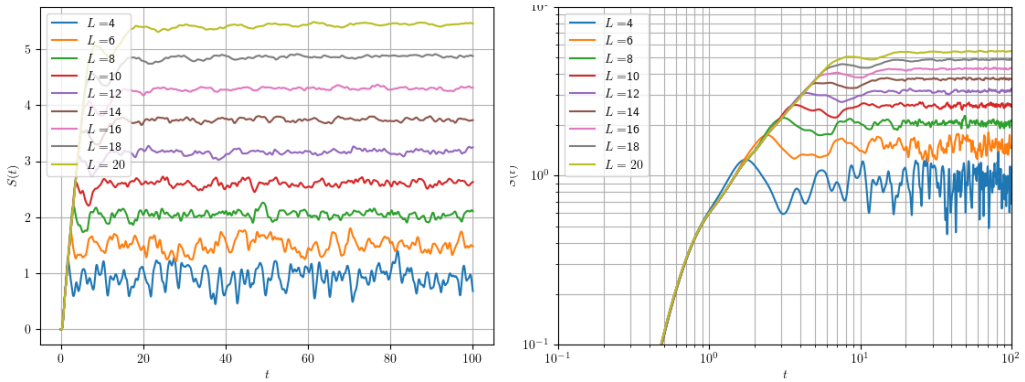
time-steps $\delta t \in \{0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1\}$ and then plotted the resulting time-evolutions of the entanglement entropy $S([1 : L/2])$ of half the chain in Figures 2 (nonintegrable Hamiltonian) and 3 (integrable Hamiltonian). In each of the four plots there is clear convergence with decreasing δt towards the results for $\delta t = 0.01$ and 0.02. In each plot the results for $\delta t = 0.01$ and 0.02 are indistinguishable at the scale of the figures, always differing by strictly less than $|\Delta S|_{max} = 0.006$ from each other, and the difference remains stable (doesn't increase) at large time. This suggests that $\delta t = 0.01$ is a reliable time-step for these simulations and I used it for most of my results.

3.1.2 Nonintegrable cases



(a) linear scale (b) loglog scale

Figure 4: Time-evolution of $S([1:L/2])$ in spin chains with lengths $L = 4$ to 20 with $(h, g, J) = (\frac{\sqrt{5}+1}{4}, \frac{\sqrt{5}+5}{8}, 1) = (0.8090..., 0.9045..., 1)$



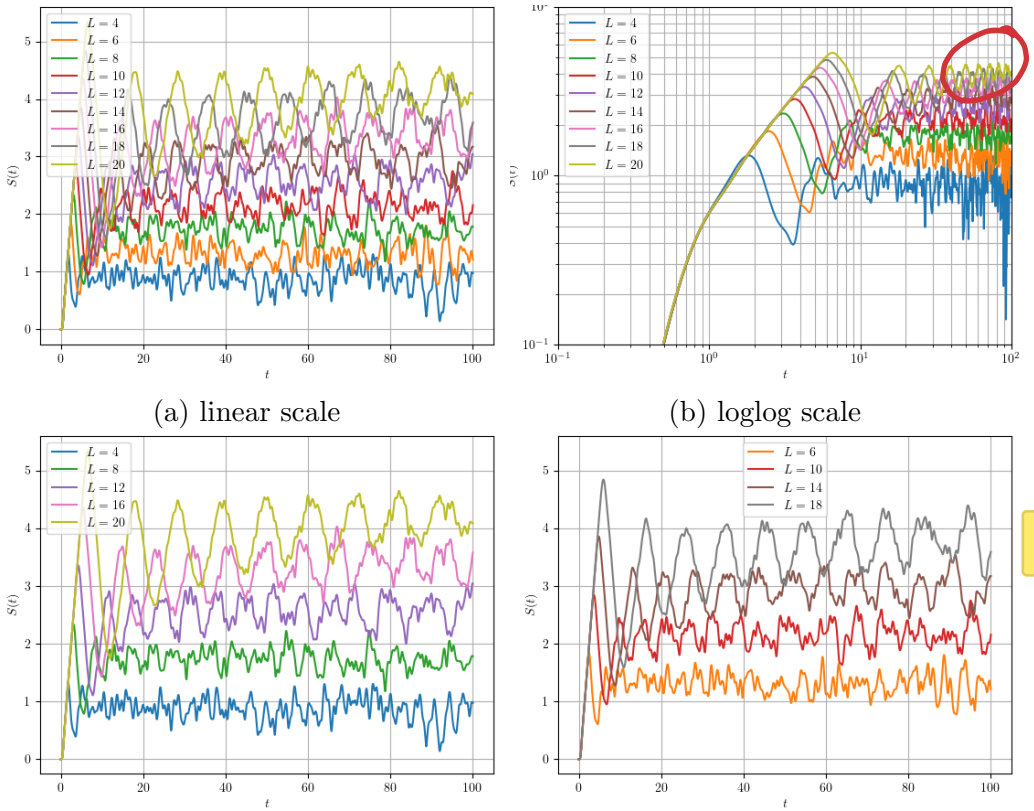
(a) linear scale (b) loglog scale

Figure 5: Time-evolution of $S([1:L/2])$ in spin chains with lengths $L = 4$ to 20 with $(h, g, J) = (0.5, -1.05, 1)$

Then I simulated chains of all even lengths from $L = 4$ to 20 with $\delta t = 0.01$ for some different values of h and g to compare the results. The results for $(h, g, J) = (\frac{\sqrt{5}+1}{4}, \frac{\sqrt{5}+5}{8}, 1)$ (as in (50)) are shown in Figure 4, and agree perfectly with the results obtained by exact diagonalisation for lengths $L \in \{4, 6, 8, 10, 12\}$ in [8] (longer lengths were not simulated in [8]). All lengths start with a similar rapid growth of entanglement which saturates near time $t_L \approx \frac{L}{2}$ and then continues fluctuating irregularly around the saturation value near $S(t \gg t_L) \approx \frac{L}{4}$. These fluctuations are very pronounced for

short lengths and decrease for larger lengths. The results for $(h, g, J) = (0.5, -1.05, 1)$ (as in (51)) are shown in Figure 5. Here the behaviour is very similar to $(h, g, J) = (\frac{\sqrt{5}+1}{4}, \frac{\sqrt{5}+5}{8}, 1)$ but the initial growth is slightly faster and saturation values are slightly higher.

3.1.3 Integrable cases



(a) linear scale

(b) loglog scale

(c) $L = 4, 8, 12, 16, 20$ linear scale

(d) $L = 6, 10, 14, 18$ linear scale

Figure 6: Time-evolution of $S([1 : L/2])$ in spin chains with lengths $L = 4$ to 20 with $(h, g, J) = (0, 1, 1)$

As an example of an integrable case I will consider the case with the longitudinal field off and transverse field on

$$(h, g, J) = (0, 1, 1) \tag{62}$$

(because when $g = 0$ the Néel state is an eigenstate of the Hamiltonian and will therefore remain a product state with zero entanglement entropy for any subsystem). The results for this Hamiltonian are shown in Figure

6. The initial growth is very similar to the integrable cases, but afterwards the behaviour is different: the initial growth ends at a clear peak followed by a rapid (approximately linear) decrease and then rise to a second peak. These oscillations then continue, becoming more irregular over time. The oscillations remain regular for longer times for longer chains.

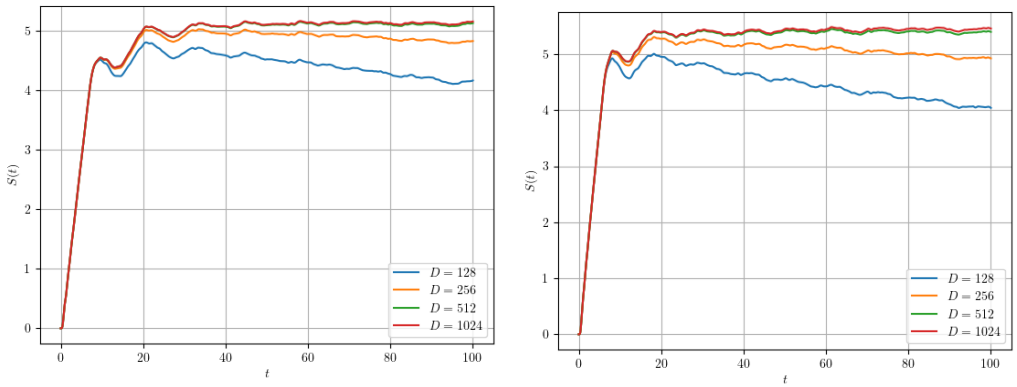
3.1.4 Truncation of bond-dimensions

The results presented so far were all computed without limit on the bond dimension D. For a spin-$\frac{1}{2}$ chain, where each local site has a local state space of dimension $d = 2$, with even length L the worst-case bond-dimensions required for an exact representation are then

$$D_r = \begin{cases} 2^r & \text{when } r \leq \frac{L}{2} \\ 2^{L-r} & \text{when } r > \frac{L}{2} \end{cases} , \tag{63}$$

so the maximum of these bond dimensions is $D = 2^{\frac{L}{2}}$. Looking at the values of the bond dimensions during the previous simulations reveals that the bond dimensions (starting from the Néel state with $D = 1$ at time 0) grow quickly with the entanglement, reaching the maximum $D = 2^{\frac{L}{2}}$ around the same time as the saturation (nonintegrable case) or first peak (integrable case) of $S(t)$, and then remain maximal for the rest of the simulation. For short chains this is not a problem on a fast processor but for longer chains, such as $L = 20$ or more, computational resources become a problem, and efficient truncation is an important advantage of MPS. So when first simulating such longer chains I also tried truncating the bond-dimensions.

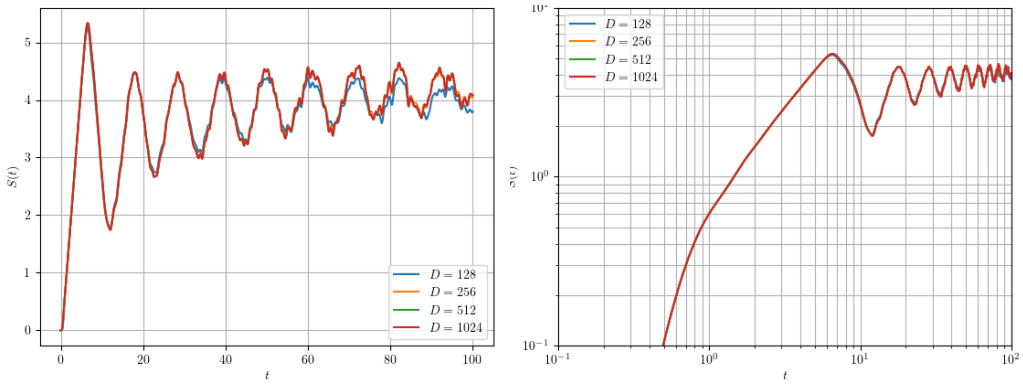
Figure 7 compares the computed evolution of a $L = 20$ chain with different maximal bond-dimensions for nonintegrable cases $(h, g, J) = (\frac{\sqrt{5}+1}{4}, \frac{\sqrt{5}+5}{8}, 1)$ and $(h, g, J) = (0.5, -1.05, 1)$. (Here $D = 2^{10} = 1024$ is the exact maximal bond-dimension for length 20). Truncation with $D = 512$ is relatively accurate, slowly linearly diverging from the exact result (with exact result I am referring to the result for $D = 1024$): the difference between the exact and $D = 512$ results for $S(t)$ grows linearly with time, but remains below 0.03 (for $(h, g, J) = (\frac{\sqrt{5}+1}{4}, \frac{\sqrt{5}+5}{8}, 1)$) or 0.06 (for $(h, g, J) = (0.5, -1.05, 1)$) at $t = 100$. For smaller D the divergence quickly becomes too significant (although the qualitative shapes of the details on the curves remain very similar to the exact results when ignoring the linear divergence). (Using $D = 512$ instead of $D = 1024$ significantly reduced the memory usage of the simulation and reduced the computation time by almost $\sim 25\%$). In section 3.2 in Figure 9 there is also one example for $L = 20$ and $(h, g, J) = (\frac{\sqrt{5}+1}{4}, \frac{\sqrt{5}+5}{8}, 1)$ with truncated bond-dimension $D = 512$ to compare with the exact results,



(a) $(h, g, J) = (\frac{\sqrt{5}+1}{4}, \frac{\sqrt{5}+5}{8}, 1)$ (b) $(h, g, J) = (0.5, -1.05, 1)$

Figure 7: Time-evolution of $S([1 : L/2])$ in spin chain of length $L = 20$ when approximated with different maximal D

and there $D = 512$ already results in a significant divergence from the corresponding exact result. Because of these errors (and because my program was already able to simulate a decent range of lengths without truncation) I continued using exact bond-dimensions for the rest of the results.



(a) linear scale (b) loglog scale

Figure 8: Time-evolution of $S([1 : L/2])$ in spin chain of length $L = 20$ with $(h, g, J) = (0, 1, 1)$ when approximated with different maximal D

Figure 8 compares the computed evolution of a $L = 20$ chain for the integrable $(h, g, J) = (0, 1, 1)$ with different maximal bond-dimensions. Here the truncated results remain much closer to exact than in the nonintegrable cases but still show an error that increases with time. Although truncation is a relatively accurate option here I also continued using exact results for the integrable cases in order to remain compatible with the nonintegrable cases.

I also tried increasing the cutoff value $\lambda_c = 10^{-16}$ to some higher values, but it was difficult to find a good value because for λ_c near $\sim 10^{-5}$ the results rapidly transition from being accurate to very inaccurate, and the exact behaviour is very different for different lengths and different (h, g, J). I therefore didn't experiment more with the cutoff value, always using $\lambda_c = 10^{-16}$ for the rest of the results.

3.2 Entanglement with random initial product states

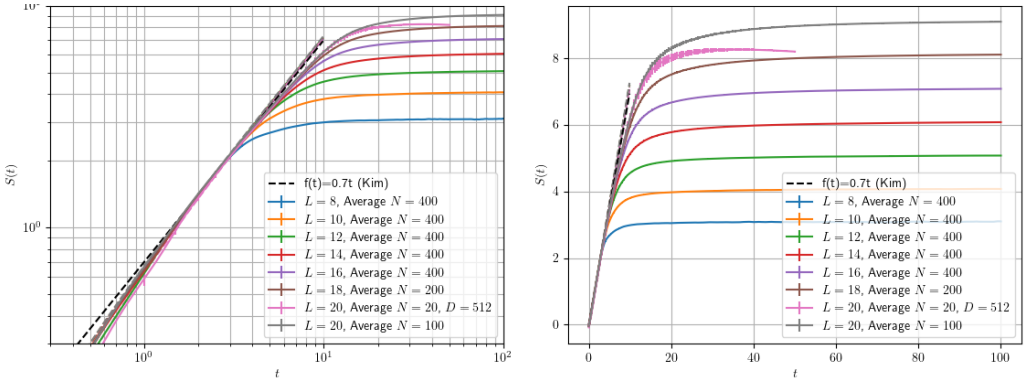
Figure 9 shows the average entanglement entropy $S(t)$ for random initial product states. The random initial product states $|\Psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \otimes ...$ were generated by letting the spin at each site point in a random direction on its Bloch sphere

$$|\psi_r\rangle = \cos\left(\frac{\theta_r}{2}\right)|0\rangle + e^{i\phi_r}\sin\left(\frac{\theta_r}{2}\right)|1\rangle \quad , \tag{64}$$

with random $\theta_r \in [0, \pi]$ and $\phi_r \in [0, 2\pi]$. For each length $L \in \{8, 10, 12, 14, 16, 18, 20\}$, I generated N (for most lengths $N = 400$ but for the larger lengths the computations were slower so there wasn't enough time; see values in the legend of Figure 9) random product states, simulated their evolution to time $T = 100$ with $(h, g, J) = (\frac{\sqrt{5}+1}{4}, \frac{\sqrt{5}+5}{8}, 1)$ and then calculated the average of $S(t)$ over all N for each point in time $t = n\delta t \leq T$ $(n \in \mathbb{N})$. This is almost the same method used in [5] except that they generated $N = 200$ new random states for each point t in time where they wanted a result for the average $S(t)$, in order to avoid correlations in the standard error. I didn't do this because the extra simulations would take too much time and when simulating time-evolution until a time T with an iterative method like TEBD you pass through the time-evolved states at all times $t = n\delta t \leq T$ $(n \in \mathbb{N})$ so it would be a waste not to use this information. In [5] they used exact diagonalization for the time-evolution.

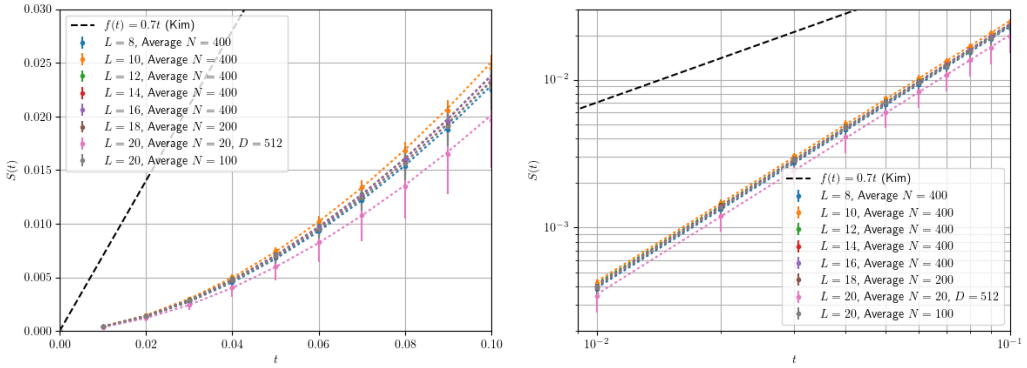
The standard errors on $S(t)$ in Figure 9 are less than 0.05 for all $L \leq 18$ and the error-bars are therefore barely visible in the figure, except for some of the $L = 20$ results. The result for $L = 20$ with maximal bond-dimension $D = 512$ was computed and included in order to verify whether the conclusion from section 3.1.4 that truncation is not a very advantageous option when starting with the Néel state also applies more generally to random product states. The difference with the more accurate result for $L = 20$ without truncation does indeed increase rapidly and significantly faster than in Figure 7.

The results in Figure 9 for $L \in \{8, 10, 12, 14, 16\}$ agree very well with the results of [5] (they only simulated these lengths), and the longer lengths



(a) loglog scale (b) linear scale

Figure 9: Average time-evolution of $S([1 : L/2])$ in spin chains with lengths $L = 4$ to 20 with $(h, g, J) = (\frac{\sqrt{5}+1}{4}, \frac{\sqrt{5}+5}{8}, 1)$



(a) linear scale (b) loglog scale

Figure 10: Average time-evolution of $S([1 : L/2])$ in spin chains with lengths $L = 4$ to 20 with $(h, g, J) = (\frac{\sqrt{5}+1}{4}, \frac{\sqrt{5}+5}{8}, 1)$ for $t \leq 0.1$. The dotted lines are the linear fits through $(\ln(t), \ln(S(t)))$.

continue the same trends. First there is the initial linear growth followed by saturation, very similar to the case for the Néel state in Figure 4. It is quite noticeable that the irregular fluctuations after saturation (present for the Néel state and when looking at $S(t)$ of the individual random simulations instead of the average) are averaged out very smoothly, even for small L. The $S(t)$ saturation-values $S_s \approx \frac{L}{2} - 1$ are higher than for the Néel state but match those of [5]. In [5] they fit a linear function $f(t) = 0.70t$ (black dashed line in Figure 9) to the initial linear growth of $S(t)$ in their results, which agrees for all lengths. Performing linear fits $f(t) = at + b$ through my results in the linear region $0 < t < 3$ (this is the region where $S(t)$ for

all lengths is approximately linear) gives the values in Table 1 (the results are also plotted as dashed lines in Figure 9). The a are all near but slightly higher than the 0.70 from [5] and the b are near but less than zero. Looking at times $t > 1$ in Figure 9, the results are almost exactly equal to $f(t) = 0.70t$ but for $t < 1$ it is already visible in Figure 9a that the average entanglement entropy becomes lower than the linear trend $f(t) = 0.70t$. Zooming further in, it is revealed that at time $t \leq 0.1$ (Figure 10) the average entanglement growth deviates significantly from the values expected for fully linear growth $f(t) = 0.70t$ (black dashed line in the figures), and is clearly nonlinear. Fitting straight lines through $(\ln(t), \ln(S(t)))$ in this region $0.01 \leq t \leq 0.1$ gives the coefficients in Table 2 (plotted as dotted lines in Figure 10), which are not exactly but near quadratic ($a \approx 1.76 \sim 2$).

	a	b
$L = 8$	0.725	-0.0655
$L = 10$	0.728	-0.0641
$L = 12$	0.735	-0.0856
$L = 14$	0.726	-0.0724
$L = 16$	0.724	-0.0643
$L = 18$	0.733	-0.0723
$L = 20$	0.740	-0.0686
$L = 20, D = 512$	0.730	-0.108

Table 1: Linear fit through $(t, S(t))$ data for $0 < t < 3$

	a	b
$L = 8$	1.767	0.292
$L = 10$	1.763	0.372
$L = 12$	1.755	0.303
$L = 14$	1.762	0.319
$L = 16$	1.765	0.326
$L = 18$	1.765	0.306
$L = 20$	1.759	0.288
$L = 20, D = 512$	1.755	0.135

Table 2: Linear fit through $(\ln(t), \ln(S(t)))$ data for $0.01 \leq t \leq 0.1$

In [5] they mention there is a small region at $t \ll 1$ where the entanglement entropy grows $\propto t^2|\log(t)|$ (although deviations from the linear $f(t) = 0.70t$ evolution aren't yet visible in any of their data). In [9] it is predicted from holography that, before the linear regime, the entanglement entropy grows quadratically $\propto t^2$. The results in Figure 10 for $t \leq 0.1$ are not exactly quadratic as shown with the $(\ln(t), \ln(S(t))$-fit but can be approximated well both with the quadratic fits and $\propto t^2|\ln(t)|$ fits. So, with my current results, it is thus not possible to distinguish the $\propto t^2|\ln(t)|$ or $\propto t^2$ growth. But it is clearly a nonlinear growth and an interesting question for further research.

3.3 Mutual information after a quench from the Néel state

In [7] they study mutual information scrambling in the Heisenberg XXZ chain (a spin chain similar to, and slightly more general than the Ising chain), comparing the behaviour for integrable versus nonintegrable parameters. They look at the mutual information $I(A, B)$ between two length-2 intervals $A = [1 : 2]$ and $B = [L-1 : L]$ at the ends of the chain, and look at the behaviour in function of the distance $d = L-4$ between the intervals, arguing that this can be used to distinguish the integrable from nonintegrable cases. I simulated the time-evolution of the mutual information for the same intervals $A = [1 : 2]$ and $B = [L-1 : L]$ for the Ising model (49), with both the integrable and nonintegrable (h, g, J), to see how their results compare to the Ising chain.

3.3.1 The quasiparticle picture

[10, 11] Quasiparticles are excitations in many-body systems, that can prop-

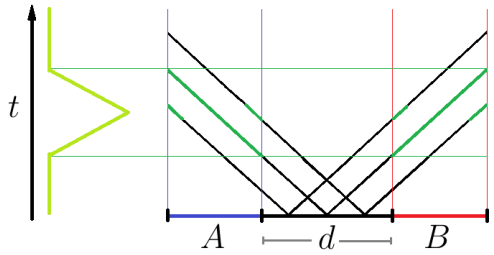
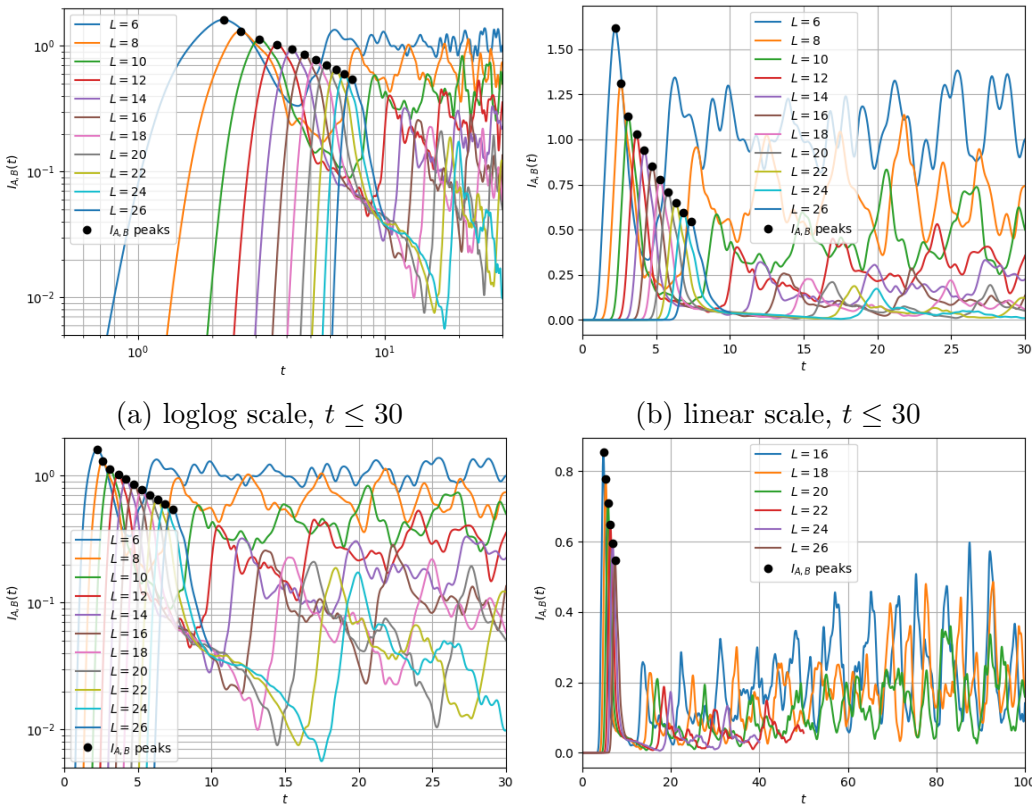


Figure 11: Simple quasiparticle picture of mutual information $I(A, B)$ time-evolution

agate through the system behaving like particles. Integrable quantum many-body systems admit stable quasiparticles with infinite lifetime, while nonintegrable quantum many-body systems only admit quasiparticles with finite lifetime. A quantum quench can act as a source of quasiparticles. When the initial state is a product state, only pairs of quasiparticles created at the same point are entangled. When these entangled particles move in opposite directions through the system they will then spread the entanglement. Considering a subsystem A, a pair of entangled quasiparticles will then contribute to the entanglement entropy of A only when one of the particles is in A and the other one outside. Considering the mutual information $I(A, B)$, a pair of quasiparticles will therefore contribute to the mutual information

when one of the particles is in A and the other in B. Figure 11 shows a space-time diagram of a simple one-dimensional system where entangled pairs of quasiparticles (diagonal black world lines) moving in opposite directions, all with the same speed, are produced throughout the system from a quench at time $t = 0$. The three pairs shown in the figure, all reach a region in time (green part of their trajectory) where one of them is in A and the other in B and they therefore contribute to $I(A, B)$. This will result in a peak in $I(A, B)$ (light-green line in the figure, assuming pairs were created uniformly throughout the system) when a maximal amount of pairs contribute. In the more realistic case of a nontrivial distribution of quasiparticle species with different velocities, the mutual information will be a superposition of peaks from the different species.

3.3.2 Integrable case



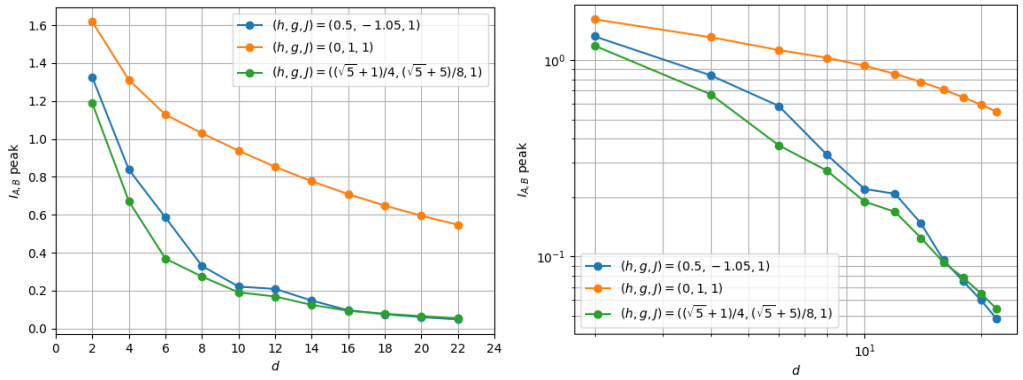
(a) loglog scale, $t \leq 30$

(b) linear scale, $t \leq 30$

(c) linlog scale, $t \leq 30$,

(d) linear scale, $L \geq 16$

Figure 12: Time-evolution of $I([1:2],[L-1:L])$ in spin chains of lengths $L = 6$ to 26 with integrable $(h, g, J) = (0, 1, 1)$



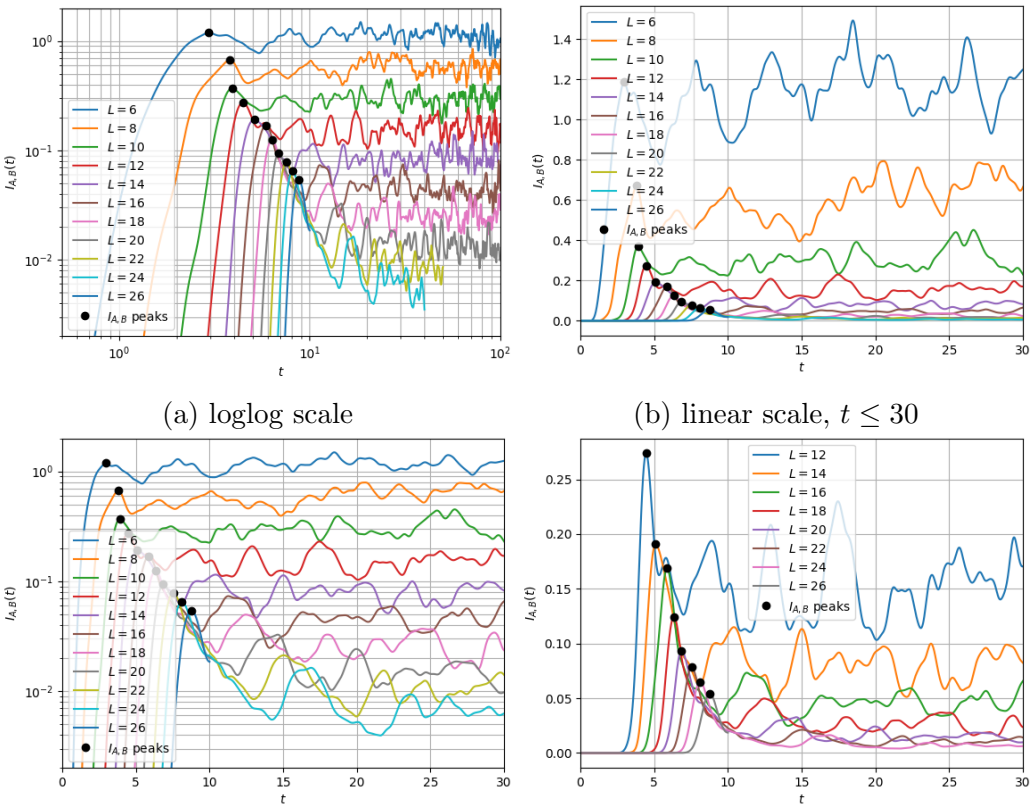
(a) linear scale (b) loglog scale

Figure 13: Primary peak-value of $I([1:2],[L-1:L])$ in function of distance $d = L - 4$, for different (h, g, J)

Figure 12 shows the results for the time-evolution of the mutual information $I([1:2],[L-1:L])$, for lengths $L = 6$ to 26, with the integrable $(h, g, J) = (0, 1, 1)$. (Shorter chains were simulated until time $t = 100$, but length $L = 22$ was simulated until time $t = 50$, $L = 24$ was simulated until $t = 40$, and $L = 26$ was simulated only until $t = 10$, because there was no time for further simulations). For all lengths, the mutual information first remains very low followed by a rapid rise to a high peak (black dots mark the maxima in the figures). The time at which the peak occurs increases approximately linearly with L, while the height of the peak decreases with L (see Figure 13). The presence of these peaks could be explained by the existence of a main species of quasiparticle with maximal velocity. For longer lengths L the particles from the middle of the chain then have to travel a longer distance $\approx \frac{L}{2} - 1$ before they contribute to the mutual information, which means the peak will occur later in time and, if they are scrambled on the way, the height of the peaks will decrease. The first peak is followed by more peaks at regular intervals, with smaller peaks scattered in between. A striking feature of the main peaks is the regular intervals between them: for a fixed L, letting $t_0(L)$ be the time of the primary peak, there are always other large peaks at times $t_n(L) \approx (2n+1)t_0(L)$ $n \in \mathbb{N}$. This timing pattern is similar to the timings of the peaks in the evolution of the entanglement entropy $S([1:L/2])$ (Figure 6). The presence of peaks other than the primary one could be explained by the existence of other species of quasiparticles with lower velocities, although this doesn't explain the regular timings of the main peaks. Another interesting feature, mainly present in longer chains (see Figure 12d) is that, after the big decrease from the primary peak to the next main peak, the height of the peaks temporarily rises with time. The

regular timings of the main peaks could also be explained if the main species of quasiparticle, causing the primary peak, can elastically reflect off the edges of the chain and contribute to the mutual information again when they reach the opposite ends of the chain (but this can't explain the later increases in the heights of the main peaks if these particles are supposed to be scrambled, as suggested above).

3.3.3 Nonintegrable cases



(a) loglog scale

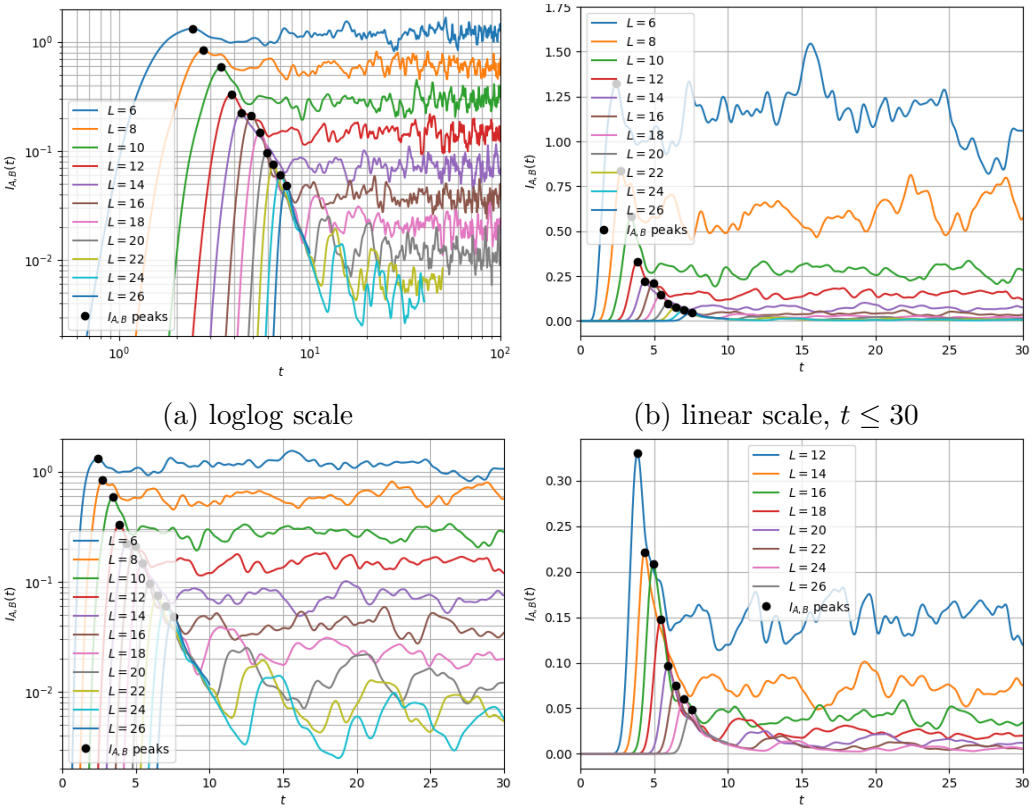
(b) linear scale, $t \leq 30$

(c) linlog scale, $t \leq 30$,

(d) linear scale, $t \leq 30$, $L \geq 12$

Figure 14: Time-evolution of $I([1:2],[L-1:L])$ in spin chains of lengths $L = 6$ to 26 with $(h, g, J) = (\frac{\sqrt{5}+1}{4}, \frac{\sqrt{5}+5}{8}, 1)$

Figures 14 and 15 show the results for the time-evolution of the mutual information $I([1:2],[L-1:L])$, for lengths $L = 6$ to 26, with $(h, g, J) = (\frac{\sqrt{5}+1}{4}, \frac{\sqrt{5}+5}{8}, 1)$ and $(h, g, J) = (0.5, -1.05, 1)$, respectively. (In both cases, shorter chains were simulated until time $t = 100$, but length $L = 22$ was simulated until time $t = 50$, $L = 24$ was simulated until $t = 40$,



(a) loglog scale

(b) linear scale, $t \leq 30$

(c) linlog scale, $t \leq 30$,

(d) linear scale, $t \leq 30$, $L \geq 12$

Figure 15: Time-evolution of $I([1:2],[L-1:L])$ in spin chains of lengths $L = 6$ to 26 with $(h, g, J) = (0.5, -1.05, 1)$

and $L = 26$ was simulated only until $t = 10$, because there was no time for further simulations). In both cases, like in the integrable case, the mutual information first remains very low and then has a peak (black dots in the figures). Compared to the integrable case, the time at which the peak occurs is similarly linear in L, but the height of the peak decreases more rapidly with L. Figure 13 shows the heights of these peaks in function of the distance $d = L - 4$ between the intervals, for both nonintegrable cases and the integrable case. The presence of these well-defined peaks suggests there might still be a main species of quasiparticle in the nonintegrable cases, but the rapid decrease in height (compared to the integrable case) suggests they are scrambled significantly faster than in the integrable case. After the primary peak there are no other clear regular peaks like in the integrable case. But looking closely at the behaviour shortly after the first peak for the longer chains (see Figures 14c and 15c), there do seem to be some smaller peaks

or plateaus in the mutual information which are similar across all larger L (in the same way as the main peaks are identifiable across different L in the integrable case). This could also be peaks due to some other species of quasiparticles, which are again scrambled much faster than in the integrable case.

The behaviour of the peaks in function of d (Figure 13) and the difference between the nonintegrable and integrable cases agrees well with the results of [7], where they similarly observed faster (probably exponential) decrease in the nonintegrable cases, and slower (algebraic $\propto d^{-\alpha}$) decrease for the integrable cases. Therefore they argued that nonintegrable systems are better scramblers than integrable systems, and that scrambling is an interesting way of distinguishing integrable and chaotic systems. In [7] they didn't find many examples of peaks other than the primary ones, but this might simply be because they simulated shorter times than I did.

4 Conclusion

For this project I wrote a program in the Python programming language, implementing TEBD for the Ising model and efficient methods to compute the entanglement entropy and mutual information from an MPS in canonical form. I used the efficient NumPy library for the tensors, contractions and linear algebra and its multiprocessing capabilities, and the Python multiprocessing library for running the TEBD steps in parallel. This, and running the simulations with parallel processing on the VUB Hydra cluster and my uncle's powerful computer, allowed me to simulate (without truncation) the entanglement entropy and mutual information of spin chains up to length $L = 26$ in a time-span of days instead of weeks, and without running out of memory. Due to the ability to simulate relatively long chains efficiently without truncation I ended up simulating all my results without truncation (I also mostly studied cases where truncation is not an accurate option, as verified in section 3.1.4). However, efficient truncation is one of the main advantages of MPS and TEBD (which was invented with low-entanglement systems in mind), so it would definitely be interesting to try out the program for simulating systems where the entanglement remains low enough so truncation is an accurate option, which would allow the simulation of much longer chains (also taking full advantage of the parallelization of the TEBD steps).

In order to verify my program I first simulated the time evolution of the entanglement entropy of half the chain after a quench from the Néel product-state, for different lengths, to reproduce the results of [8]: they matched

exactly. Then I ran the same simulations with different parameters of the Hamiltonian to compare the nonintegrable and integrable cases: the initial monotonic growth of entanglement from zero was similar in both cases, but in the integrable case it was followed by regular periodic oscillations (becoming more irregular over time), while in the nonintegrable cases the entanglement saturated (with small irregular fluctuations around the saturation-value). I also computed the time-evolution of the average entanglement entropy after quenches from random product states, in the nonintegrable case, to reproduce the results of [5]: again the results matched very well, although I found a region of initial nonlinear growth at $t \leq 0.1$ which was not present in their results. This is an interesting point for further research.

Then I simulated the time evolution of the mutual information between two length-2 intervals at the ends of the chain, after a quench from the Néel state. I compared the integrable and nonintegrable cases, observing similar behaviour to that in the Heisenberg XXZ chain studied in [7]: in the integrable cases there are initial peaks in the information with a slow decrease in height in function of the chain length, and in the nonintegrable cases there are similar initial peaks but with a much faster decay in height. These peaks could be explained by quasiparticles, being scrambled much better in the nonintegrable cases than the integrables cases. In my results I also found other peaks after the primary one, including large ones at periodic intervals, in the integrable case. And the nonitegrable cases showed much smaller and irregular peaks after the primary peaks. This also suggests better scrambling in the nonintegrable cases, but definitely requires further research into the patterns of these peaks.

5 Acknowledgments

I would like to thank my promotor Prof. Ben Craps for suggesting and allowing me to complete this fascinating project, and my co-promotor Dr. Surbhi Khetrapal for her guidance and help throughout the year. I would also like to thank Prof. Charles Rabideau, specifically for coming up with the main idea of the algorithm for computing the mutual information described in section 1.3.4.

I would like to thank my uncle for giving me access to his new powerful computer, allowing me to freely run my simulations on a powerful machine. And I would like to thank the SISC HPC team for allowing bachelor students like me to use the professional HPC Hydra cluster for the heavy computations of their bachelor project. Without my uncle's computer or the Hydra cluster my results would have been significantly limited to shorter chains.

References

[1] J. Sakurai, J. Napolitano, *Modern Quantum Mechanics*, second edition, Addison-Wesley (2011).

[2] R. Orús, *A Practical Introduction to Tensor Networks: Matrix Product States and Projected Entangled Pair States*, Annals of Physics **349** (2014) 117-158, https://arxiv.org/abs/1306.2164

[3] U. Schollwöck, *The density-matrix renormalization group in the age of matrix product states*, Annals of Physics **326** (2011) 96, https://arxiv.org/abs/1008.3477v2

[4] G. Vidal, *Efficient classical simulation of slightly entangled quantum computations*, Phys. Rev. Lett. **91** (2003) 147902, https://arxiv.org/abs/quant-ph/0301063v2

[5] H. Kim, D. Huse, *Ballistic spreading of entanglement in a diffusive nonintegrable system*, Phys. Rev. Lett. **111** (2013) 127205, https://arxiv.org/abs/1306.4306

[6] M. Bañuls, J. Cirac, M. Hastings, *Strong and weak thermalization of infinite non-integrable quantum systems*, Phys. Rev. Lett. **106** (2011) 050405, https://arxiv.org/abs/1007.3957

[7] V. Alba, P. Calabrese, *Quantum information scrambling after a quantum quench*, Phys. Rev. B **100** (2019) 115150, https://arxiv.org/abs/1903.09176

[8] F. Tori , *Entanglement entropy evolution in a quantum Ising spin chain with longitudinal and transversal magnetic field*, bachelor project report, VUB, 2019.

[9] H. Liu, S. Suh, *Entanglement Tsunami: Universal Scaling in Holographic Thermalization*, Phys. Rev. Lett. **112** (2014) 011601, https://arxiv.org/abs/1305.7244v2

[10] P. Calabrese, J. Cardy, *Evolution of Entanglement Entropy in One-Dimensional Systems*, Journal of Statistical Mechanics **0504** (2005) P04010, https://arxiv.org/abs/cond-mat/0503393v1

[11] V. Balasubramanian, A. Bernamonti, N. Copland, B. Craps, F. Galli, *Thermalization of mutual and tripartite information in strongly coupled two dimensional conformal field theories*, Phys. Rev. D **84** (2011) 105017, https://arxiv.org/abs/1110.0488

A Appendix

A.1 Computing the canonical MPS representation

The following is an example of applying the algorithm described in section 1.3.3 to a simple state of a system of 3 two-level (for example spin-$\frac{1}{2}$) sites. So $N = 3$ and $d_1 = d_2 = d_3 = 2$.

Consider the (unnormalized, for simplicity) state

$$\begin{aligned} |\Psi\rangle &= -2\,|\uparrow\uparrow\uparrow\rangle + 4\,|\uparrow\downarrow\downarrow\rangle + 2\,|\downarrow\uparrow\downarrow\rangle - |\downarrow\downarrow\uparrow\rangle \\ &= \sum_{i_1,i_2,i_3=0}^{1} \Psi^{i_1,i_2,i_3}\,|i_1\rangle \otimes |i_2\rangle \otimes |i_3\rangle \quad , \end{aligned} \tag{65}$$

with $\Psi^{0,0,0} = -2$, $\Psi^{0,1,1} = 4$, $\Psi^{1,0,1} = 2$, $\Psi^{1,1,0} = -1$ and all other entries 0 (I will separate all indices with commas here in order to avoid confusion with the digits and the multiplication when merging indices). Then the matrix M of (15) for the Schmidt decomposition across the bipartition ([1],[2:3]) is then obtained by simply reshaping Ψ^{i_1,i_2,i_3}: setting $i = i_1$ and merging the indices i_2 and i_3 into $j = d_3 i_2 + i_3$, so that $M_{i_1,2i_2+i_3} = \Psi^{i_1,i_2,i_3}$. This gives

$$M = \begin{pmatrix} -2 & 0 & 0 & 4 \\ 0 & 2 & -1 & 0 \end{pmatrix} \quad ,$$

for which the compact SVD is

$$U = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \;,\; S = \begin{pmatrix} 2\sqrt{5} & 0 \\ 0 & \sqrt{5} \end{pmatrix} \;,\; V^\dagger = \begin{pmatrix} -\frac{1}{\sqrt{5}} & 0 & 0 & \frac{2}{\sqrt{5}} \\ 0 & \frac{2}{\sqrt{5}} & -\frac{1}{\sqrt{5}} & 0 \end{pmatrix} \tag{66}$$

The definition of $\Gamma^{[1]i_1}_{\alpha_1}$ (25) is just the change of basis matrix from the left Schmidt vectors $\left|\tau^{[1]}_{\alpha_1}\right\rangle$ to the local basis $|i_1\rangle$, and the left Schmidt vectors expressed in the local basis $|i_1\rangle$ are the columns of U, so

$$\Gamma^{[1]} = U = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad . \tag{67}$$

The Schmidt coefficients are the diagonal elements of S:

$$\lambda^{[1]}_0 = 2\sqrt{5} \;,\; \lambda^{[1]}_1 = \sqrt{5} \quad .$$

For any complex numbers a, b, c, d: $\binom{1}{0} \otimes \binom{a}{b} = \begin{pmatrix} a \\ b \\ 0 \\ 0 \end{pmatrix}$ and $\binom{0}{1} \otimes \binom{c}{d} = \begin{pmatrix} 0 \\ 0 \\ c \\ d \end{pmatrix}$ so

$$\binom{1}{0} \otimes \binom{a}{b} + \binom{0}{1} \otimes \binom{c}{d} = \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} \tag{68}$$

This is exactly the form of (27), with $\left|\omega_{\alpha_1}^{[3]0}\right\rangle = \binom{a}{b}$ and $\left|\omega_{\alpha_1}^{[3]1}\right\rangle = \binom{c}{d}$ written in the local $|i_3\rangle$ basis. The coefficients of the right Schmidt vectors $\left|\tau_{\alpha_1}^{[2:3]}\right\rangle$ in the local basis $|i_2\rangle \otimes |i_3\rangle$ are the rows of $V^\dagger$ so, using (68) and (66),

$$\left|\omega_0^{[3]0}\right\rangle = \begin{pmatrix} -\frac{1}{\sqrt{5}} \\ 0 \end{pmatrix}, \ \left|\omega_0^{[3]1}\right\rangle = \begin{pmatrix} 0 \\ \frac{2}{\sqrt{5}} \end{pmatrix}, \ \left|\omega_1^{[3]0}\right\rangle = \begin{pmatrix} 0 \\ \frac{2}{\sqrt{5}} \end{pmatrix}, \ \left|\omega_1^{[3]1}\right\rangle = \begin{pmatrix} -\frac{1}{\sqrt{5}} \\ 0 \end{pmatrix} \quad .$$

The matrix M of (15) for the Schmidt decomposition across the bipartition $([1:2],[3])$ is obtained by merging the indices i_1 and i_2 of Ψ^{i_1,i_2,i_3} into $i = 2i_1 + i_2$ and setting $j = i_3$, so that $M_{2i_1+i_2,i_3} = \Psi^{i_1,i_2,i_3}$. This gives

$$M = \begin{pmatrix} -2 & 0 \\ 0 & 4 \\ 0 & 2 \\ -1 & 0 \end{pmatrix} \quad ,$$

for which the compact SVD is

$$U = \begin{pmatrix} 0 & -\frac{2}{\sqrt{5}} \\ -\frac{2}{\sqrt{5}} & 0 \\ -\frac{1}{\sqrt{5}} & 0 \\ 0 & -\frac{1}{\sqrt{5}} \end{pmatrix}, \ S = \begin{pmatrix} 2\sqrt{5} & 0 \\ 0 & \sqrt{5} \end{pmatrix}, \ V^\dagger = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \tag{69}$$

So the Schmidt coefficients are

$$\lambda_0^{[2]} = 2\sqrt{5}, \ \lambda_1^{[2]} = \sqrt{5}$$

and the coefficients of the right Schmidt vectors in the local basis $|i_3\rangle$ are the rows of $V^\dagger$:

$$\left|\tau_0^{[3]}\right\rangle = \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \ \left|\tau_1^{[3]}\right\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

In (29), the vector $\Gamma_{\alpha_1,(\cdot)}^{[2]i_2}$, for fixed α_1 and i_2, is the representation of the corresponding $\left|\omega_{\alpha_1}^{[3]i_2}\right\rangle$ in the basis $\{\lambda_0^{[2]}\left|\tau_0^{[3]}\right\rangle, \lambda_1^{[2]}\left|\tau_1^{[3]}\right\rangle\}$. Since the $\left|\tau_{\alpha_2}^{[3]}\right\rangle$ are the columns of V^*, the change of basis from $|i_3\rangle$ to $\{\left|\tau_0^{[3]}\right\rangle, \left|\tau_1^{[3]}\right\rangle\}$ is obtained by multiplying by $(V^*)^\dagger = V^T$. So, for all values of α_1 and i_2, the vectors $\Gamma_{\alpha_1,(\cdot)}^{[2]i_2}$ are obtained by multiplying $\left|\omega_{\alpha_1}^{[3]i_2}\right\rangle$ by V^T and then dividing the results pointwise by $\lambda_{(\cdot)}^{[2]}$. This gives

$$\Gamma_{0,(\cdot)}^{[2]0} = \begin{pmatrix} 0 \\ -\frac{1}{5} \end{pmatrix}, \ \Gamma_{0,(\cdot)}^{[2]1} = \begin{pmatrix} -\frac{1}{5} \\ 0 \end{pmatrix}, \ \Gamma_{1,(\cdot)}^{[2]0} = \begin{pmatrix} -\frac{1}{5} \\ 0 \end{pmatrix}, \ \Gamma_{1,(\cdot)}^{[2]1} = \begin{pmatrix} 0 \\ -\frac{1}{5} \end{pmatrix} \quad .$$

The definition of $\Gamma^{[N]i_3}_{\alpha_2}$ (31) is just the change of basis matrix from the right Schmidt vectors $\left|\tau^{[3]}_{\alpha_2}\right\rangle$ to the local basis $|i_3\rangle$, and coefficients of the right Schmidt vectors expressed in the local basis $|i_3\rangle$ are the columns of V^*, so

$$\Gamma^{[N]} = V^* = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \quad . \tag{70}$$

Putting everything together gives the following MPS representation

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 2\sqrt{5} & 0 \\ 0 & \sqrt{5} \end{pmatrix} \left(\begin{pmatrix} 0 & -\frac{1}{5} \\ -\frac{1}{5} & 0 \end{pmatrix} \begin{pmatrix} -\frac{1}{5} & 0 \\ 0 & -\frac{1}{5} \end{pmatrix} \right) \begin{pmatrix} 2\sqrt{5} & 0 \\ 0 & \sqrt{5} \end{pmatrix} \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \tag{71}$$

A.2 Python code

The following sections contain the essential parts of my code: my implementations of the algorithms explained in sections 1 and 2. Besides this my program contains plenty of other convenience-functions for better usability and extra information, but these are not essential for the purpose of this project.

A.2.1 Schmidt decomposition

I wrote the following function to compute the (optionally truncated) Schmidt decomposition using the Numpy `numpy.linalg.svd()` function. Here `S` is the coefficient tensor of the state and `r` is the bond around which the Schmidt decomposition of `S` should be computed. All singular values smaller than `cutoff` are discarded and `d_max` allows specifying a maximal bond-dimension. (The default value for `d_max` is -1, which will be ignored and result in no truncation, and the default value for `cutoff` is $\lambda_c = 10^{-16}$).

```
1  import numpy as np
2
3  def schmidt(S,r,d_max=-1,cutoff=(10**(-16))): #schmidt decomposition
4      sz = S.shape
5      N = len(sz)
6      m = 1
7      n = 1
8      for k in range(r):
9          m*=sz[k]
10     for k in range(r,N):
11         n*=sz[k]
12
13     k = min(m,n)
14     M = S.copy().reshape((m,n))
```

```
    u,s,vh = np.linalg.svd(M,full_matrices=False)
    #|u[:,k]>=left schmidt vectors
    #s[k]=schmidt coefficients
    #|vh[k,:]>=right schmidt vectors

    if d_max==-1 or d_max>k:
        d_max = k

    k_cut = d_max
    for i in range(d_max-1,-1,-1):
        k_cut = i+1
        if s[i]>cutoff:
            break

    u = u[:,0:k_cut]
    vh = vh[0:k_cut,:]
    s = s[0:k_cut]

    if k_cut <k:
        s = s / np.sqrt(np.sum(s*s))

    return u,s,np.transpose(vh)
```

Listing 1: Schmidt decomposition

A.2.2 Canonical form

The following function is my implementation of the algorithm in section 1.3.3 for computing the canonical MPS representation of a state with coefficient tensor `S`. (The optional truncation parameters `d_max` and `cutoff` will be passed to all Schmidt decompositions).

```
import numpy as np

def cMPS(S,d_max=-1,cutoff=(10**(-16))): #canonical MPS representation of
    arbitrary state
    sz = S.shape
    N = len(sz)
    dtot = 1
    for k in range(N):
        dtot*=sz[k]

    L1,s1,R1 = schmidt(S,1,d_max=d_max,cutoff=cutoff) # |S>=s1[a1]*|L1[:,a1
    ]>|R1[:,a1]>
    gamma1 = L1.copy() # left schmidt vectors |L1[:,a1]> = gamma1[i1,a1]*|i1
    >
    s1rank = s1.shape[0]

    svs = [s1]
    gammas = []
    dR = dtot//sz[0]
    for n in range(1,N-1):
        dr = dR//sz[n]
        W = [[R1[dr*s:(dr*(s+1)),k] for s in range(sz[n])] for k in range(
    s1rank)] #|R1[:,a1]>=|i2>|W[a1][i2]>

        L2,s2,R2 = schmidt(S,n+1,d_max=d_max,cutoff=cutoff)
        s2rank = s2.shape[0]
```

```
23
24          sfs = np.ones((min(dr,s2rank),),dtype='complex')
25          sfs[0:s2rank]= s2 +(s2==(0+0j)) #avoid zero-division
26
27          gamma = [[(np.matmul(np.conjugate(np.transpose(R2)),w)/sfs)[0:s2rank
    ] for w in ws] for ws in W] #|W[a1][i2]> = gamma[a1][i2][a2]*s2[a2]*|R2
    [:,a2]>
28          # (write |W2[a1][i2]> in basis |R2[:,a2]> --> change of basis-matrix
     = np.conjugate(np.transpose(R2)))
29          R1 = R2
30          s1rank = s2rank
31          svs.append(s2)
32          gammas.append(np.array(gamma))
33          dR = dr
34
35      gammaN = R1
36      return MPS(svs,gamma1,gammas,gammaN)
```

Listing 2: compute canonical MPS form

Here `MPS(svs,gamma1,gammas,gammaN)` is the constructor of a class I wrote to work with MPS in canonical form. The `MPS` class has attributes `MPS.svs` (a list containing the NumPy arrays of the $\lambda^{[1]}$,...,$\lambda^{[N]}$ tensors), `MPS.gamma1` (the NumPy array of the $\Gamma^{[1]}$ tensor), `MPS.gammas` (a list containing the NumPy arrays of the $\Gamma^{[2]}$,...,$\Gamma^{[N-1]}$ tensors) and `MPS.gammaN` (the NumPy array of the $\Gamma^{[N]}$ tensor), which are all initiated with the values of the arguments with the same names in the constructor.

A.2.3 Entanglement entropy from canonical MPS

In the `MPS` class I implemented methods for computing the entanglement entropies and mutual information. The following method computes the entanglement entropy $S([1:\mathtt{r}])$ (23).

```
1  class MPS:
2      ...
3      def get_EE(self,r):
4          l = self.svs[r-1]*self.svs[r-1]
5          S = np.sum(-l*np.log(l))
6          return S
```

Listing 3: get entanglement entropy from canonical MPS

The next method computes the reduced density matrix $\rho_{A\cup B}$ with $A = [1:\mathtt{l}]$ and $B = [\mathtt{r}:N]$, using the algorithm described in 1.3.4.

```
1  class MPS:
2      ...
3      def get_rhoAB(self,l,r):#(A=[1:l])U(B=[r:N])
4          M = np.swapaxes(np.tensordot(self.gammas[l-1]*self.svs[l][None,None
       ,:],np.conjugate(self.gammas[l-1])*self.svs[l][None,None,:],axes
       =([1],[1])),1,2)#contract gamma(l+1),gamma(l+1)*,lambda(l+1),lambda(l+1)
        with indices: aic,bid,c,d->abcd
5          for k in range(l+1,r-1):
6              M = np.tensordot(M,self.gammas[k-1]*self.svs[k][None,None,:],
       axes=([2],[0]))#indices: abcd,cir->abdir
```

```
            M = np.tensordot(M,np.conjugate(self.gammas[k-1])*self.svs[k][
    None,None,:],axes=([2,3],[0,1])) #indices: abdir,dil->abrl

        L = self.gamma1*self.svs[0][None,:] #indices: ia,a->ia
        for k in range(1,l):
            L = np.tensordot(L,self.gammas[k-1]*self.svs[k][None,None,:],
    axes=([k],[0]))#contract L,gamma(k+1),lambda(k+1) with indices: (...)a,
    aib,b->(...)ib

        ds = L.shape
        L = L.reshape((np.product(ds[0:-1]),ds[-1]))
        R = np.swapaxes(self.gammaN,0,1) #indices: ia->ai
        for k in range(len(self.gammas)+1,r-1,-1):
            R = np.tensordot(self.gammas[k-2]*self.svs[k-1][None,None,:],R,
    axes=([2],[0]))#contract gamma(k-1),lambda(k-1),R with indices: aib,b,b
    (...)->ai(...)

        ds = R.shape
        R = R.reshape((ds[0],np.product(ds[1:])))
        temp = np.tensordot(R,M,axes=([0],[2])) #indices: cj,abcd ->jabd
        temp = np.tensordot(L,temp,axes=([1],[1])) #indices: ia,jabd->ijbd
        temp = np.tensordot(temp,np.conjugate(L),axes=([2],[1])) #indices:
    ijbd,mb->ijdm
        temp = np.tensordot(temp,np.conjugate(R),axes=([2],[0])) #indices:
    ijdm,dn->ijmn
        shp = temp.shape
        rhoAB = temp.reshape((shp[0]*shp[1],shp[2]*shp[3]))
        return rhoAB
```

Listing 4: get density matrix of $A \cup B$

This is then used in the following method to compute the required entanglement entropies for the mutual information (3). So the mutual information $I(A = [1 : \mathtt{l}], B = [\mathtt{r} : N])$ is equal to `SA+SB-SAUB`.

```
class MPS:
    ...
    def get_IAB(self,l,r):
        rhoAB = self.get_rhoAB(l,r)
        dAB,W = np.linalg.eigh(rhoAB)
        SAUB = 0
        for k in range(dAB.shape[0]):
            if dAB[k]>0.0:
                SAUB = SAUB - dAB[k]*np.log(dAB[k])

        SA = self.get_EE(l)
        SB = self.get_EE(r-1)
        return SA,SB,SAUB
```

Listing 5: get mutual information from canonical MPS

A.2.4 TEBD

In order to apply the 2-site unitary gates to a canonical MPS while preserving canonical form, as described in section 1.4, I wrote the following function, which accepts a tuple containing only the required tensors $U^{[r:r+1]} = \mathtt{U}$, $\lambda^{[r-1]} = \mathtt{lambdaL}$, $\Gamma^{[r]} = \mathtt{gammaL}$, $\lambda^{[r]} = \mathtt{lambdaC}$, $\Gamma^{[r+1]} = \mathtt{gammaR}$ and $\lambda^{[r+1]} =$

`lambdaR` (and truncation parameters `d_max` and `cutoff`) to compute the new $\widetilde{\Gamma}^{[r]} =$ `gammal`, $\widetilde{\lambda}^{[r]} =$ `s` and $\widetilde{\Gamma}^{[r+1]} =$ `gammar`. If `lambdaL` is empty or `lambdaR` is empty the function will assume that these are the respective ends of the chain and apply the gate accordingly.

```
def apply_Ugate(inpt):
    U,lambdaL,gammaL,lambdaC,gammaR,lambdaR,d_max,cutoff = inpt
    if lambdaL==[]:
        temp1 = np.tensordot(gammaL*lambdaC[None,:],gammaR,axes=([1],[0]))
        temp2 = temp1*lambdaR[None,None,:]
        theta = np.tensordot(U,temp2,axes=([2,3],[0,1]))
        sz = theta.shape
        L,s,R = schmidt(theta,1,d_max=d_max,cutoff=cutoff)
        srank = s.shape[0]
        gammal = L.reshape((sz[0],srank))
        gammar = R.transpose().reshape((srank,sz[1],sz[2]))
        gammar = gammar/lambdaR[None,None,:]
        return gammal,s,gammar
    elif lambdaR==[]:
        temp1 = lambdaL[:,None,None]*gammaL
        temp2 = np.tensordot(temp1*lambdaC[None,None,:],gammaR,axes
    =([2],[1]))
        theta = np.tensordot(temp2,U,axes=([1,2],[2,3]))
        sz = theta.shape
        L,s,R = schmidt(theta,2,d_max=d_max,cutoff=cutoff)
        srank = s.shape[0]
        gammal = L.reshape((sz[0],sz[1],srank))
        gammar = R.reshape((sz[2],srank))
        gammal = gammal/lambdaL[:,None,None]
        return gammal,s,gammar
    else:
        temp1 = lambdaL[:,None,None]*gammaL
        temp2 = np.tensordot(temp1*lambdaC[None,None,:],gammaR,axes
    =([2],[0]))
        theta = np.moveaxis(np.tensordot(temp2*lambdaR[None,None,None,:],U,
    axes=([1,2],[2,3])),[1,2,3],[3,1,2])
        sz = theta.shape
        L,s,R = schmidt(theta,2,d_max=d_max,cutoff=cutoff)
        srank = s.shape[0]
        gammal = L.reshape((sz[0],sz[1],srank))
        gammar = R.transpose().reshape((srank,sz[2],sz[3]))
        gammal = gammal/lambdaL[:,None,None]
        gammar = gammar/lambdaR[None,None,:]
        return gammal,s,gammar
```

Listing 6: apply unitary gate preserving canonical MPS

This tuple input form was chosen with the threading in mind, in order not to have to share memory between the parallel threads or duplicate unnecessary memory (for example if I would pass the full `MPS` to all threads) when applying this function in parallel.

The next listing shows the code for my implementation (for even length chains) of TEBD for the Ising Hamiltonian (49), when using (61) to reorganize the Hamiltonian into 2-site terms. Here `mps0` is the initial `MPS` state, `delta_t` is the time-step δt, `Nsteps` is the number of iterations, `hfield` is the value of h, `gfield` is the value of g, `coupling` is the value of J, `func` is

the function which will be used to compute all the desired data from the `MPS` at each time-step, `d_max` and `cutoff` are the Schmidt truncation parameters and `NUMTHREADS` is the number of threads used to simulate the TEBD steps in parallel (and `h_bar` was an optional parameter to change the value of $\hbar$). This function will then return the final `MPS` and the list `data` containing the computed output of `func` at each time-step.

```
def Ising_TEBD_asymm_edge(mps0,delta_t,Nsteps,hfield,gfield,coupling,func=
    None,d_max=-1,cutoff=(10**(-16)),NUMTHREADS=0,h_bar=1):
    expHodd = np.exp(-1j/h_bar*delta_t*coupling)
    expHoddinv = 1/expHodd
    U_odd = np.diag([expHodd,expHoddinv,expHoddinv,expHodd]).reshape
    ((2,2,2,2))

    H_even = np.array([[2*hfield+coupling,gfield,gfield,0],[gfield,-coupling
    ,0,gfield],[gfield,0,-coupling,gfield],[0,gfield,gfield,coupling-2*
    hfield]])
    d,W = np.linalg.eigh(H_even)
    D = np.diag(np.exp(-1j/h_bar*delta_t*d))
    U_even = np.matmul(W,np.matmul(D,np.transpose(np.conjugate(W)))).reshape
    ((2,2,2,2))

    H_start = np.array([[2*hfield,gfield,gfield,0],[gfield,-2*coupling,0,
    gfield],[gfield,0,0,gfield],[0,gfield,gfield,2*coupling-2*hfield]])
    d,W = np.linalg.eigh(H_start)
    D = np.diag(np.exp(-1j/h_bar*delta_t*d))
    U_start= np.matmul(W,np.matmul(D,np.transpose(np.conjugate(W)))).reshape
    ((2,2,2,2))

    H_end = np.array([[2*hfield,gfield,gfield,0],[gfield,0,0,gfield],[gfield
    ,0,-2*coupling,gfield],[0,gfield,gfield,2*coupling-2*hfield]])
    d,W = np.linalg.eigh(H_end)
    D = np.diag(np.exp(-1j/h_bar*delta_t*d))
    U_end = np.matmul(W,np.matmul(D,np.transpose(np.conjugate(W)))).reshape
    ((2,2,2,2))

    mps = mps0.copy()
    n_sites = len(mps.gammas)+2
    n2 = n_sites//2

    if func==None:
        func = lambda x : x.get_EE(n2)

    data = [func(mps)]

    if NUMTHREADS<1:
        for k in range(Nsteps):
            for n in range(1,n2-1):
                mps.apply_Ugate(U_even,2*n,d_max=d_max,cutoff=cutoff)
            mps.apply_Ugate(U_end,2*(n2-1),d_max=d_max,cutoff=cutoff)
            mps.apply_Ugate(U_start,0,d_max=d_max,cutoff=cutoff)
            for n in range(0,n2-1):
                mps.apply_Ugate(U_odd,2*n+1,d_max=d_max,cutoff=cutoff)
            data.append(func(mps))
        return mps,data

    pool_N = Pool(processes=NUMTHREADS)

    for k in range(Nsteps):
        inputs = [(U_start,[],mps.gamma1,mps.svs[0],mps.gammas[0],mps.svs
```

```
    [1],d_max,cutoff)]
45      for n in range(1,n2-1):
46          r = 2*n
47          inputs.append((U_even,mps.svs[r-1],mps.gammas[r-1],mps.svs[r],
    mps.gammas[r],mps.svs[r+1],d_max,cutoff))
48
49      inputs.append((U_end,mps.svs[n_sites-3],mps.gammas[n_sites-3],mps.
    svs[n_sites-2],mps.gammaN,[],d_max,cutoff))
50
51      results = mergelistN(pool_N.map(apply_Ugates,splitlistN(inputs,
    NUMTHREADS)),NUMTHREADS)
52      gammal,sN,gammaN = results[-1]
53      gamma1,s0,gammar = results[0]
54      mps.gamma1 = gamma1
55      mps.svs[0] = s0
56      mps.gammas[0] = gammar
57      mps.gammas[-1] = gammal
58      mps.svs[-1] = sN
59      mps.gammaN = gammaN
60
61      for n in range(1,n2-1):
62          r = 2*n
63          gammal,s,gammar = results[n]
64          mps.svs[r] = s
65          mps.gammas[r-1]=gammal
66          mps.gammas[r]=gammar
67
68      inputs = []
69      for n in range(0,n2-1):
70          r = 2*n+1
71          inputs.append((U_odd,mps.svs[r-1],mps.gammas[r-1],mps.svs[r],mps
    .gammas[r],mps.svs[r+1],d_max,cutoff))
72
73      results = mergelistN(pool_N.map(apply_Ugates,splitlistN(inputs,
    NUMTHREADS)),NUMTHREADS)
74      for n in range(0,n2-1):
75          r = 2*n+1
76          gammal,s,gammar = results[n]
77          mps.svs[r] = s
78          mps.gammas[r-1]=gammal
79          mps.gammas[r]=gammar
80
81      data.append(func(mps))
82
83  pool_N.close()
84  return mps, data
```

Listing 7: TEBD for Ising model with (61)

The default value of `func` is `lambda x : x.get_EE(n2)`, which will calculate the entanglement entropy $S([1:L/2])$ of half the chain, and for the mutual information $I([1:2],[L-1:L])$ I used `func=lambda x : x.get_IAB(2,L-1)`. (I also combined both in `func=lambda x : (x.get_IAB(2,L-1), x.get_EE(n2))` for many simulations).

If `NUMTHREADS=0` (default value) the TEBD steps won't be simulated in parallel (so only the built-in multiprocessing of NumPy contractions and linear algebra will be used). Here the function

`MPS.apply_Ugate(U,r,d_max=-1,cutoff=10**(-16))` does exactly the same as `apply_Ugate()`, shown above, for computing the new canonical tensors when applying the unitary gate with tensor `U` to the `MPS` at site `r`, but directly takes the input tensors from the `MPS` and updates the `MPS` with the new tensors in-place.

Otherwise, if `NUMTHREADS>0`, a thread pool of `NUMTHREADS` threads will be created and the unitary gate applications will be distributed between these threads and performed in parallel with these threads. The function `apply_Ugates()` takes a list of tuples for `apply_Ugate()` as input and applies `apply_Ugate()` to all tuples in this list and returns a list of the results. (The function `splitlistN(lst,N)` splits a list `lst` into `N` lists, each containing every `N`th element of `lst`, and `mergelistN(lsts,N)` is the inverse).

The 2-site terms `H_even` and `H_odd` are named odd or even oppositely to (61) because the indexing of the sites in the code is zero-based. Using (60) instead of (61) to split the Hamiltonian into 2-site terms, is just a matter of replacing the definitions of `U_even`, `U_odd`, `U_start` and `U_end`, in the beginning of the TEBD function, with

```
1   H_even = np.array([[hfield+coupling,0,gfield,0],[0,hfield-coupling,0,
    gfield],[gfield,0,-hfield-coupling,0],[0,gfield,0,coupling-hfield]])
2   d,W = np.linalg.eigh(H_even)
3   D = np.diag(np.exp(-1j/h_bar*delta_t*d))
4   U_even = np.matmul(W,np.matmul(D,np.transpose(np.conjugate(W)))).reshape
    ((2,2,2,2))
5
6   U_odd = U_even
7
8   H_start = np.array([[hfield,0,gfield,0],[0,hfield-2*coupling,0,gfield],[
    gfield,0,-hfield,0],[0,gfield,0,2*coupling-hfield]])
9   d,W = np.linalg.eigh(H_start)
10  D = np.diag(np.exp(-1j/h_bar*delta_t*d))
11  U_start = np.matmul(W,np.matmul(D,np.transpose(np.conjugate(W)))).
    reshape((2,2,2,2))
12
13  H_end = np.array([[2*hfield,gfield,gfield,0],[gfield,0,0,gfield],[gfield
    ,0,-2*coupling,gfield],[0,gfield,gfield,2*coupling-2*hfield]])
14  d,W = np.linalg.eigh(H_end)
15  D = np.diag(np.exp(-1j/h_bar*delta_t*d))
16  U_end = np.matmul(W,np.matmul(D,np.transpose(np.conjugate(W)))).reshape
    ((2,2,2,2))
```

Listing 8: alternative definitions of unitary gates when using (60)