UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE FÍSICA

Study of Tensor Network Applications in Complex Networks

Francisco Santos Sousa e Costa

Mestrado Integrado em Engenharia Física

Dissertação orientada por:
João Carlos Caetano de Freitas Pires da Cruz
Hygor Piaget Monteiro Melo

2022

Resumo da Tese

Em Física, problemas de muitos corpos são problemas que envolvem sistemas compostos por um grande número de partículas que interagem entre si e estão correlacionadas umas com as outras. Resolver este tipo de problemas é bastante difícil uma vez que a quantidade de informação necessária para descrever tal sistema aumenta exponencialmente com o número de partículas do sistema. Para além disso, a função de onda do sistema seria demasiado complexa para ser calculada. Posto isto, é necessário arranjar métodos diferentes para resolver o problema. Considerando um sistema de eletrões independentes uns dos outros e com dois estados possíveis, então, para descrever este sistema, seriam necessárias 2^N dimensões, sendo que N corresponde ao número de eletrões que compõem o sistema. No entanto, em sistemas reais as partículas não são independentes umas das outras. Existe correlacionamento entre as partículas, o que faz com que a dimensão necessária para descrever o sistema diminua. Em teoria, 2^N estados totais seria uma quantidade de informação intratável. Todavia, o correlacionamento entre as partículas faz com que os sistemas reais observáveis não explorem o espaço de Hilbert por completo, mas, focam-se numa pequena parte deste espaço.

Este correlacionamento pode ser visto como uma generalização das coordenadas. Este conceito proveniente da mecânica analítica permite fazer uma renormalização das unidades de forma a descrever o mesmo sistema usando menos dimensões. O mesmo se passa em sistemas cujas partículas estão correlacionadas, uma vez que esse correlacionamente por si só, reduz as dimensões necessárias para descrever o sistema. Certas redes complexas têm propriedades que nos permitem renormalizar a sua escala, obtendo-se a mesma rede, sem sofrer nenhuma alteração. A este tipo de redes, chamam-se redes livre de escala e possuem uma geometria fractal, ou seja, ao mudar-se a escala da rede, esta mantêm-se inalterada. De forma a estudar o método que perde menos informação quando se renormaliza a escala de redes complexas, como por exemplo uma rede Barabási, optou-se por utilizar redes de tensores, mais propriamente um algoritmo denominado por Matrix Product State ou MPS. Escolheu-se utilizar o MPS pela sua habilidade de descrever a função de onda de sistemas quânticos. Considerando um sistema de dois spins correlacionados, sabe-se que a função de onda deste sistema é dada por $\psi^{n_1 n_2} = \sum_i A_i^{n_1} A_i^{n_2}$, ou seja pelo produto da amplitude de cada spin representada por cada A_i. A mesma expressão representa também a MPS de um sistema de duas partículas com spin. A área das redes tensoriais tem uma notação mais visual, ao contrário de outros métodos matemáticos. Esta notação consiste em nós e ligações. Um vetor é representado por um nó com uma ligação enquanto uma matriz é representada por um nó com duas ligações, como mostra a figura 1. O número de ligações representa o rank do tensor em questão, isto é, um tensor de rank 3 terá três ligações, onde cada uma simboliza um índice, como por exemplo i ou j. A vantagem desta notação é que

permite representar cálculos complexos de forma esquemática. A função de onda apresentada anteriormente seria representada, nesta notação, como dois nós que partilham ambos a mesma ligação, neste caso a ligação cujo índice é igual a i. Para além disso, ambos os nós possuem também uma ligação que não é partilhada, sendo que cada uma representa um índice diferente e mudo.

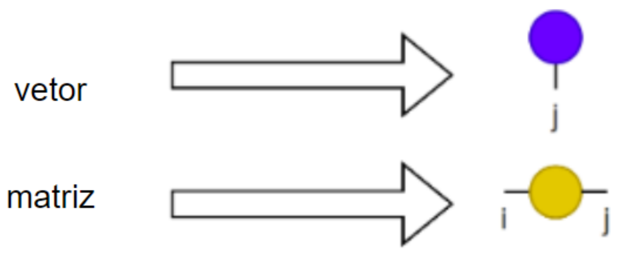


Figure 1: Representação gráfica de um vetor e de uma matriz.

Uma MPS pode ser vista como um comboio de tensores, onde, inicialmente não existe distinção entre as carruagens, isto é, a função de onda do sistema é representada graficamente por um único tensor compacto. De forma a transformar a MPS num conjunto de tensores correlacionados é necessário realizar a decomposição em valores singulares (SVD) sucessivamente. À medida que se aplica o SVD, está-se a "cortar" o tensor compacto. Tal como está representado na figura 2, o tensor original transforma-se num conjunto de tensores representados por nós, onde cada site do tensor original passa a dois tensores resultantes da aplicação do SVD. O tensor que representa os vetores próprios e o tensor que representa os valores próprios. O tensor que representa cada site da MPS tem duas ligações. A primeira representa a dimensão física do site e a segunda representa a *bond dimension* e liga-se ao tensor dos valores próprios. A *bond dimension* é a dimensão do espaço necessária para descrever o sistema e é a característica que iremos estudar.

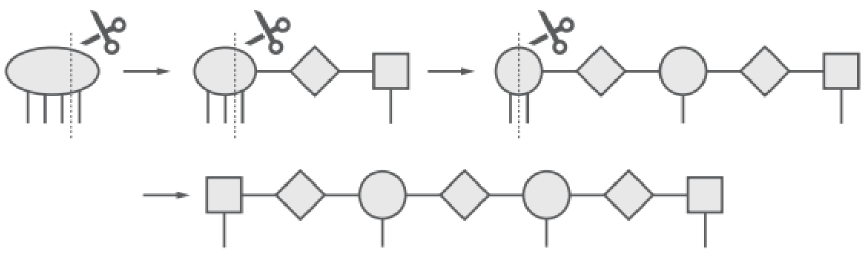


Figure 2: Representação gráfica de uma MPS onde se está a aplicar o SVD.

Uma das vantagens de utilizar este algoritmo é o facto de o SVD poder ser aplicado a qualquer tensor. Para além disso, parte da informação que compõe o tensor pode ser desprezada visto que a aplicação do SVD numa matriz A origina uma matriz U, uma matriz Σ e uma matriz V. $A = U\Sigma V^t$, onde a matriz A é uma matriz $m \times n$, a matriz U é uma matriz $m \times m$, a matriz Σ é uma matriz $m \times n$ e a matriz V^t é uma matriz $n \times n$. A matriz U e V são denominadas de matrizes de vetores próprios esquerdos e direitos, respetivamente. A matriz Σ é denominada de matriz de valores próprios. Ao analisar a matriz dos valores próprios, consegue-se retirar informação sobre a importância de cada valor próprio visto que estes se encontram organizados por grau de importância . Posto isto, a informação mais relevante para descrever o sistema

encontra-se nos primeiros valores próprio sendo que se pode truncar estas matrizes e desprezar a informação negligenciável.

Ao utilizar a representação e a notação da MPS é possível descrever o mesmo sistema usando apenas $N2m^3$ parâmetros ao invés de 2^N parâmetros, onde N é o número de partículas do sistema e m é a *bond dimension*. Esta redução drástica do número de coeficientes necessários para descrever o sistema permite resolver problemas de muitos corpos, onde a quantidade de memória necessária é um factor fundamental. Esta compressão é valida sob a condição de que a MPS está a representar os vetores próprios de baixa energia de Hamiltonians locais a uma dimensão.

De forma a estudar o método que perde menos informação quando se renormaliza a escala de uma rede complexa, utilizou-se uma rede Barabási. Escolheu-se um tipo de rede complexa uma vez que estas representam naturalmente sistemas reais como por exemplo sistemas orgânicos, ecológicos e sociais. Gerou-se este tipo de rede e retirou-se a conectividade de cada nó da rede, isto é, se o primeiro nó se encontra ligado ao segundo, terceiro, até ao último. Se o nó partilhar uma ligação com outro então regista-se o valor 1 na conectividade entre esses dois nós. Caso não partilhem nenhuma ligação então deve registar-se um 0. Uma vez obtida a conectividade de todos os nós, utilizou-se essa informação como input para a MPS. Para o mesmo input, aplicou-se o algoritmo várias vezes, onde se fez variar o valor da *bond dimension*, visto que é a característica que queremos estudar. Ao aplicar o algoritmo está-se a treinar a MPS com base na conectividade dos nós da rede complexa. De forma a treinar a MPS, primeiro aplica-se o SVD e depois o algoritmo DMRG. Neste algoritmo realizam-se vários varrimentos da MPS onde os valores aleatórios que originalmente preenchiam a MPS são alterados de acordo com o input utilizado. Uma vez treinada a MPS, é necessário avaliá-la, através de medições de cada site da MPS. Dado vez que a dimensão física de cada site é 2, mede-se o primeiro site e obtém-se uma dos dois possíveis valores, 0 ou 1. De seguida, mede-se o segundo site com a condicionante do valor obtido no primeiro site. Repete-se este processo até se ter medido todos os sites da MPS e obtém-se um resultado de output. É necessário calcular a fidelidade do resultado e comparar com a fidelidade original da MPS de forma a avaliar o estado de convergência da mesma.

Treinou-se a MPS utilizando o mesmo input, para diferentes valores da *bond dimension*, e conclui-se que para valores da *bond dimension* superiores a 3, a MPS não converge. Para valores de 2 e 3, a MPS convergiu e em ambos os casos obtiveram-se 2 resultados de output, o que faz com que seja possível concluir que o valor da *bond dimension* ideal é entre 2 e 3, corroborando o valor teórico de 2.7, para uma rede Barabási. O facto de se terem obtido 2 valores de output leva à conclusão de que o espaço vetorial da MPS é composto por dois vetores linearmente independentes, os vetores da base. Apesar de se ter cumprido o objetivo inicial, é de salientar que o nível de complexidade deste método é demasiado elevado face a outros métodos como modelos generativos ou redes neuronais, que conseguem obter os mesmos resultados de formas mais simples, mas, no entanto, mais devagar. Quando se passar para o dominio da computação quântica, os algoritmos baseados em redes de tensores serão mais vantagosos visto que a complexidade desaparece, uma vez que se está a computar em algoritmos de natureza quântica.

Palavras-Chave: Redes de Tensores, MPS, Bond Dimension, DMRG.

Abstract

Many-body quantum problems are still very complex to solve and today's solutions do not take into account that the objects of the systems are entangled with each other. By using tensor networks, we are able to describe the same system using drastically fewer coefficients. Instead of 2^N parameters, in certain systems, we only require $N2m^3$. In the Matrix Product State (MPS) algorithm, it is included the Singular Value Decomposition (SVD), which allows us to truncate the tensors and keep only the crucial information about the system, and the Density Matrix Renormalization Group (DMRG), which allows us to obtain the lowest energy MPS wave function of our system. The goal was to study the optimal value for the bond dimension when renormalizing the scale of a complex network. The connectivity of each node of a Barabási network was used as the input for the algorithm and the MPS was trained based on whether the nodes share a connection or not. When measuring each site after training, we obtained two different outputs corresponding to the two linear independent vectors that form the space of the MPS. The MPS was trained using different values for the bond dimension. Nevertheless, only the values equal to 2 and 3 produced viable results since it did not converge for different values of the bond dimension. The connectivity of a Barabási network follows a power law proportional to x^{-p}. When the Barabási network was characterized we obtained that its connectivity followed a power law proportional to $x^{-2.7}$, which is between the theoretical values of 2 and 3. The value obtained for $p = 2.7$ further proves the value obtained for the bond dimension between 2 and 3, meaning that the dimension of the space needed to fully describe the system is between said values. Finally, two different methods are suggested which obtain the same results in a simpler and quicker way.

Keywords: Tensor Networks, MPS, Bond Dimension, DMRG.

Acknowledgement

First and foremost, I would like to express my gratitude towards my advisors, João Cruz and Hygor Melo, for all the effort and help they gave me. Without them, this dissertation would not be possible.

Secondly, my family, that was there for me every day of my life, to help me with anything I needed without hesitation.

Last, but not least, all my friends, past and present, that have been a part of my journey so far.

Content

1 Introduction **1**

1.1 Motivation 1

1.2 Objectives 2

2 State of the Art **3**

2.1 Many-Body Quantum System Problem 3

2.2 Complex Networks 4

2.3 Tensor Networks 6

2.3.1 Notation 6

2.3.2 Matrix Product State 7

2.3.3 Matrix Product Operator 8

2.3.4 Projected Entangled Pair States 9

2.3.5 Tree Tensor Networks 10

2.3.6 Multiscale Entanglement Renormalization Ansatz 11

3 Problem Statement **12**

4 Literature Review of MPS **15**

4.1 Tensor Networks 15

4.1.1 Low Entanglement States 16

4.2 Matrix Product State 16

4.2.1 Singular Value Decomposition 17

4.2.2 MPS Gauge 18

4.2.3 From a general quantum state to a MPS 19

4.3 Matrix Product Operator 21

4.4 Ground state calculations with MPS 23

5 Methods **25**

6 Results **37**

7 Discussion and Conclusion **41**

Appendices **47**

A Code **49**

List of Figures

1 Representação gráfica de um vetor e de uma matriz. ii
2 Representação gráfica de uma MPS onde se está a aplicar o SVD. ii

2.1 One configuration of the Ising model being re-scaled. On the left image, it is represented one configuration of the Ising model; In the middle image, one is re-scaling and forming groups of 9 spins; On the right image, one has the same configuration in a different scale length. Figure taken from Ref. [8] 4
2.2 Example of a complex network. Figure taken from Ref. [12] 5
2.3 Diagram representation of N-rank tensors. Figure inspired by Ref. [18]. 6
2.4 Examples of tensor contractions. Figure inspired by Ref. [18]. 6
2.5 a)Graphical representation of $\frac{\partial H}{\partial \psi_j} \neq 0$; b) Graphical representation of a generic partial derivation with respect to a tensor A. Figure inspired by Ref. [9]. 7
2.6 Graphical representation of the diagram notation of the vector T. Figure taken from Ref. [20]. 7
2.7 Graphical representation of the MPS applied to a 4-party state. Figure taken from Ref. [21]. 8
2.8 Graphical representation of a MPO. Figure taken from Ref. [22]. 9
2.9 Graphical representation of the product of two MPOs for a tensor ona single site i. Figure taken from Ref. [23]. 9
2.10 Graphical representation of a 4x4 PEPS with open boundary conditions. Figure taken from Ref. [25]. 10
2.11 Graphical illustration of a TNN. Figure taken from Ref. [28]. 10
2.12 Graphical illustration of a 1D MERA. Figure taken from Ref. [9]. 11

3.1 Example of a small complex network. Figure taken from Ref. [31]. 12
3.2 A simple pendulum of mass m and length l. Figure taken from Ref. [34]. 13

4.1 Graphical representation of GG^{-1}. Figure taken from Ref. [35]. 19
4.2 Insertion of GG^{-1} in between the first and second tensor of a general MPS. Figure taken from Ref. [35]. 19
4.3 Graphical representation of the first tensor, the singular value and ψ after the SVD. 20
4.4 Graphical representation of a construction of a MPS of an general quantum state in an iterative way by successive applications of the SVD; Visual interpretation of the equation 4.12. Figure taken from Ref. [42]. 21
4.5 Graphical representation of a MPO. Figure taken from Ref. [28]. 22

4.6 Graphical representation of a MPO of a 4 site system. 22

5.1 Example of a Barabási graph with 10 nodes and the code used to generate it. . . 26
5.2 Code used to extract the connections of each node in order to be used as an input for the MPS. 26
5.3 Code used to extract the connections of each node in order to be used as an input for the MPS. 27
5.4 Pseudo code for a randomized SVD. Figure taken from Ref. [50]. 27
5.5 (a) Code used to generate and fill the MPS with random values drawn from a normal Gaussian distribution. (b) Example of the sampling MPS with 3 nodes. . 28
5.6 (a) Result of performing the SVD onto site 3 and absorbing $U\Sigma$ into site 2; The first site remains unaltered. (b) Result of performing the SVD in all 3 sites and normalizing the first one. 29
5.7 Resulting sampling MPS after being filtered by our input. 30
5.8 Illustration of the DMRG being applied one time. After we reach (c), we need to repeat (b) and (a) and that completes one run of the algorithm. 30
5.9 Summary of the DMRG in practise. A single run of the DMRG consists of performing all 3 steps. Step 3 is illustrated in figure 5.10. 31
5.10 (a) Visualisation of the right sweep. Since we are working with the 3 sites MPS example, the right sweep will only consist of doing the matrix multiplication between sites 2 and 3. (b) Result of applying step 4 of the prototype shown in figure 5.4 when the center is site number 1. 31
5.11 Calculation of the SVD applied to the normalized local tensor represented in figure 5.2(b). 32
5.12 (a) Σ and V^T being absorbed into a single tensor that is going to be absorbed by the tensor representing site 2 of the sampling MPS. (b) Absorption of the resulting tensor into the tensor representing the original site 2 of the sampling MPS. 33
5.13 Result of the sampling MPS after performing steps 1 and 2 of the summary shown in figure 5.9. Site 1 is represented by U, calculated in figure 5.11. Site 2 is represented by the tensor calculated in figure 5.12(b). Site 3 is represented by the original tensor shown in figure 5.6(b) 33
5.14 Resulting sampling MPS after 1 run of the DMRG algorithm. 34
5.15 Result of $S \times S^{\dagger}$, where S is the tensor representing site 1 of the MPS after the first run of the DMRG shown in figure 5.14. 35
5.16 Result of the contraction between the tensor representing site 1, [1 0], and the tensor representing site 2, shown in figure 5.14. 35
5.17 Result of contraction between [1 0], that represents site collapsing to $|0>$, knowing that site 1 collapsed to $|1>$, and the tensor that represents site 3, shown in figure 5.14. 36

6.1 Graphic of the LN of the number of connections vs the LN of the number of nodes. 37

A.1 Code used to generate the Barabási network and extract the connectivity of each node. 49
A.2 Code used to for the configurations of the algorithm. 50

A.3 Code used to set up the Logger. 51
A.4 Code used to process the data. 52
A.5 Code used to generate the sampling MPS and the absorption of the result of the SVD. 52
A.6 Code used to perform the SVD 53
A.7 Code used to evaluate the MPS. 54
A.8 Code used to perform the DMRG algorithm. 55
A.9 Code used to perform the right sweep. 56
A.10 Code used to perform the left sweep. 56
A.11 Code used to calculate the local tensor. 57
A.12 (a) Performing the DMRG from the right to the left. (b) Performing the DMRG from the left to the right 57
A.13 (a) Code used to set the configurations. (b) Code used to run the DMRG. 58
A.14 Code used to set the path configuration and to start the code. 58
A.15 Code used to measure the MPS and evaluate the convergence of the MPS. . . . 59

Symbols and Acronyms

TN	Tensor Networks
MPS	Matrix Product State
SVD	Singular Value Decomposition
$DMRG$	Density Matrix Renormalization Group
$MERA$	Multiscale Entanglement Renormalization Group
RN	Renormalization Group
MPO	Matrix Product Operator
$PEPS$	Projected Entangled Pair States
TTN	Tree Tensor Networks

Chapter 1

Introduction

1.1 Motivation

Tensor Networks (TN), have an increasing important role in the progress of understanding entanglement in many-body quantum systems. “Tensor Networks are mathematical representations of quantum many-body systems based on their entanglement structure” and “different tensor network structures describe different physical situations, such as low-energy states of gapped 1D systems, 2D systems and scale-invariant systems” [1]. Moreover, TN are also relevant in areas such as quantum gravity, where Multiscale Entanglement Renormalization Ansatz, (MERA), can be linked to geometry space through the anti-de Sitter/conformal field theory, (AdS/CFT), where AdS is a geometric space with negative curvature and CFT is a quantum field theory with conformal symmetry which includes scale invariance [2].

Today’s challenges focus, mainly, on the application of machine learning to real systems, such as text language [3]. However, this does not take into account that the components of the systems are entangled with each other and for that reason, it violates the main principle of the application of the industry’s most commonly used algorithms, neural networks, and decision trees. These algorithms rely on the independence of the system’s components and the fulfilment of Kolmogorov’s axioms for the space of probabilities. One possible way to overcome this limitation is by using tensor networks, which are algorithms developed in Physics to study and explain the ground states of many body quantum systems with entanglement.

Nowadays, Guifre Vidal and his research team are at the front of tensor networks applied to many-body quantum systems. Using MERA to exploit the spacial structure of entanglement allows them to produce an efficient description of the ground state of the system [4].

1.2 Objectives

In this project we aim to study the minimization of information loss in scale renormalization of complex networks using tensor networks. Primarily, in this work, the goal is to learn what tensor networks are and only then, I will apply a tensor network to a complex network. In the next phase, the objective is to apply this algorithm to a Barabási network, which is an example of an inflationary system, in order to find the the minimum value of the dimension of the space that describes the network and loses less information.

Chapter 2

State of the Art

2.1 Many-Body Quantum System Problem

The many-body quantum problem is a physics problem where one needs to describe the properties of quantum systems made of interacting particles. Considering that the particles have more than one possible state, the wave function of the system has an enormous amount of information and it is extremely difficult or even impossible to calculate.

The fact that the dimension of the space of possible configurations increases exponentially with N, where N is the number of particles, it makes this problem very complex. Nonetheless, one can use some analytical approximations and numerical methods to address the issue. Examples are *Hartree-Fock* methods and approximations [5], the Monte Carlo methods [6] and numerical methods based on the renormalization group (RG) paradigm [7].

A many-body quantum system has many quantum degrees of freedom. The square of the wave function gives the amplitude of probability of each single system configuration, which means that a generic state of the system is described by the N-rank tensor, $\psi = \sum_{\vec{\alpha}} \psi_{\alpha_1\alpha_2...\alpha_N} |\alpha_1\alpha_2...\alpha_N >$, $\vec{\alpha} = \alpha_1\alpha_2...\alpha_N$. In the mean field approach, one assumes that the quantum correlations are negligible, which means that wave function of the many-body system is a result of a tensor product of N individual wave functions. This implies that, the dimension of space has an exponential growth with N.

Another method is the Real-Space Renormalization Group. This method is an approximation based on the hypothesis that the ground of a system is composed of low-energy states of the system's bipartitions. Using this hypothesis it is possible to describe the ground state properties of many-body systems with large size N. To better understand RG, Ref. [8] uses the Ising model as an example, where one re-scales the problem and obtaining the same result in different lengths. In figure 2.1, it is shown one configuration of the Ising model in a certain scale, on the left, and when one re scales the problem, middle figure, groups of nine spins are formed and the dominant spin defines the spin of the group. In an Ising model, the critical temperature represents the temperature at which there are no longer spontaneous magnetizations, meaning that the atomic spins start to align with each other. If one is at the critical temperature, T_c, then the system should be scale invariant.

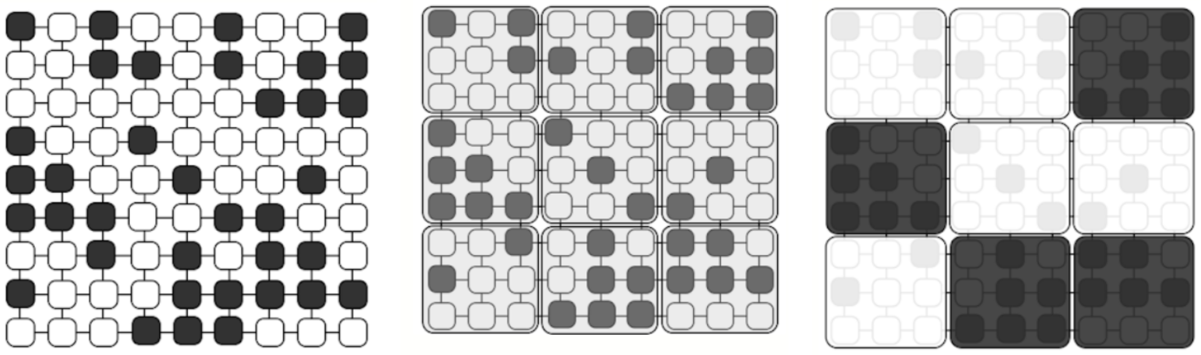


Figure 2.1: One configuration of the Ising model being re-scaled. On the left image, it is represented one configuration of the Ising model; In the middle image, one is re-scaling and forming groups of 9 spins; On the right image, one has the same configuration in a different scale length. Figure taken from Ref. [8]

Nonetheless, the base assumption of RG is not always true. In fact, there are important physical systems that violate the hypothesis and in order to demonstrate why it fails, Ref. [9] uses a simple example, the free particle in an infinite box potential, so that the reader can understand. The reason why RG fails is because only keeping the lowest energy states sometimes is not the best truncation method. In order to overcome this issue, the Density Matrix Renormalized Group method, (DMRG), was developed. The main difference between the RG and DMRG is the truncation rule. The DMRG method keeps the relevant degrees of freedom in a renormalization procedure targeting low-energy eigenstates of 1D Hamiltonians, i.e. DMRG method attempts to find the lowest-energy matrix product state wave function of the Hamiltonian.

2.2 Complex Networks

An arbitrary graph is a set of nodes (entities) and edges (connections) with any arbitrary topology. The type of graph we are interested in is called a complex network and they are called complex because one cannot predict their behavior based only on their individual components. Figure 2.2 shows an example of a complex network. To understand complex networks one must understand their structure and functionality. Despite all the different possibilities, real networks have many characteristics in common. Real Networks are for example the nervous system and the World Wide Web, where the nodes represent the individuals and the connections represent social relationships or conversations [10]. A complex network can be a scale-free network. A network is called free-scale if the connections per node of the network, or in other words, the degree distribution, follows a power law [11].

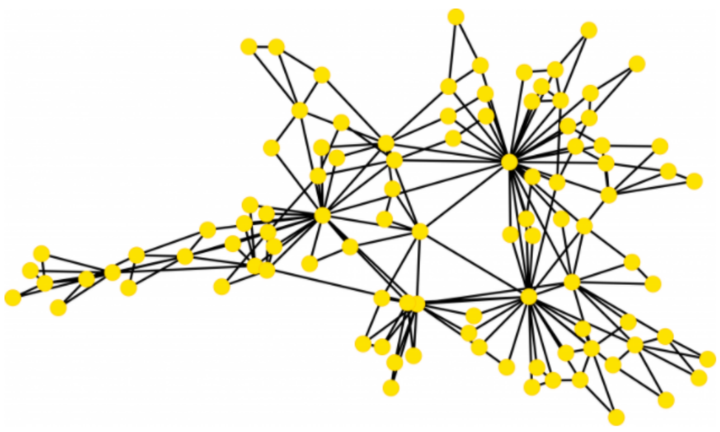


Figure 2.2: Example of a complex network. Figure taken from Ref. [12]

These systems are called complex because one cannot predict their collective behavior from the individual components that form the system. A simple example that can help understand networks in general, is given by Ref. [13], where they tackle the *Konigsberg* problem. The *Konigsberg* problem askes how can one person cross the 7 bridges of a specific city without crossing the same bridge twice. Instead of seeing the map of the city it is much easier to represent the city in terms of a network, where each bridge is represented by the connections and the land is represented by the nodes.

In complex networks, there are three basic concepts one must understand first: the average path length, clustering coefficient and degree distribution. Suppose we have a network and two of the nodes are labelled as i and j, then d_{ij} represents the distance between the two nodes and is defined as the number of edges along the shortest path that connects both nodes. D represents the diameter of the network and is defined as the maximum distance d_{ij}, where i and j are any pair of nodes. Furthermore, the average path length is defined as the mean shortest distance between all pair of nodes. The degree of a node is defined as the the total number of connections/edges that the node has to other nodes. This means that, the higher the degree of the node, the more connections to others nodes it has [14]. The degree of a node is one of the centrality measures and it can translate the importance of the node to the network [15].

The clustering coefficient is defined as the the average pairs of neighbours of a node that are also neighbours of each other. If two nodes are neighbours of a third node then there is a high chance they share an edge.

One of the main reasons why complex networks is a successful framework is due to their natural ability to represent discrete systems. Complex networks are used in Communications and Economy, Medicine, Linguistics and Social Networks. [16]

In the branch of linguistics, the use of linguistics data is crucial for automated systems such as the Web search engines and machine translators. Some linguistic structures can be seen as networks and complex networks play a role in this area where linguistics networks can be formed to study the semantics between the words. Furthermore, the linguistics networks can also be formed using the position of the words in a text instead of using dictionaries. These networks are called superficial and the words are connected if they appear as neighbours in a text and it can be used, for instance, to choose the synonym most expected in a given context [17].

2.3 Tensor Networks

2.3.1 Notation

An N-rank tensor is a mathematical object with N indexes and it takes the form in 2.1.

$$T_{\alpha_1 \dots \alpha_N} \tag{2.1}$$

An example of N-rank tensor is the vector, which is a 1-rank tensor and takes the form of v_j. A 2-rank tensor is a matrix, M_{ij} and a third-rank tensor takes the form of T_{ijk} for example. In the figure 2.3, the diagram notation of this N-rank tensors is represented, from which it is possible to conclude that a vector corresponds to a node with 1 edge, a matrix corresponds to a node with 2 edges and a 3-rank tensor corresponds to a node with 3 edges. Meaning that the notation of an N-rank tensor would be a node with N edges.

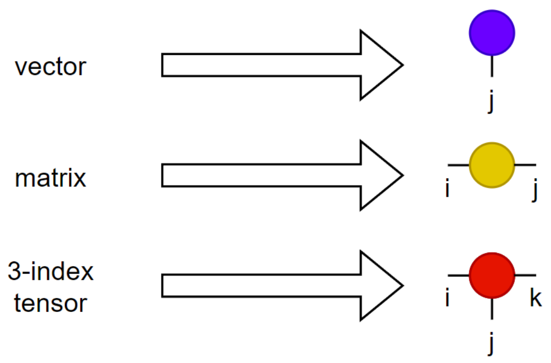


Figure 2.3: Diagram representation of N-rank tensors. Figure inspired by Ref. [18].

When talking about tensor diagrams there are 2 rules. The first one states that tensors are notated by solid shapes and their indices are notated by lines that emerge from the solid shape. The second rule states that when one connects two index lines it means that a contraction or summation over those indices is being performed [19]. This idea is similar to the Einstein convention. Some examples of contractions are given in the figure 2.4.

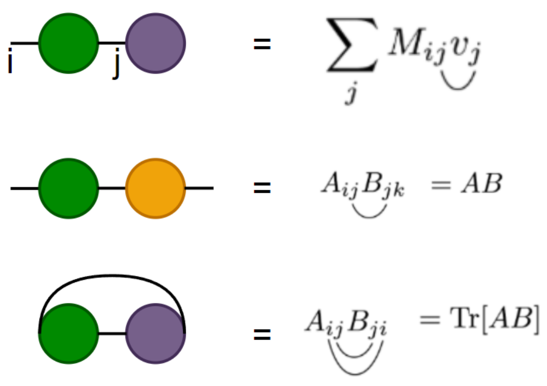


Figure 2.4: Examples of tensor contractions. Figure inspired by Ref. [18].

Since it is always possible to choose a bipartition of n indexes in two disjoint groups, one can recast the tensor in a matrix form and make use of the linear algebra operations in matrices. One crucial operation is the Singular Value Decomposition (SVD), which states that a tensor T_{ij} is equal to $\sum_k S_{ik}V_{kk}D_{kj}$, where V is a diagonal and positive matrix, S and D are unitary matrices. This combined with the fast decay of singular values allows the truncation of the matrices S and V.

Another useful operation is the differentiation. The derivative of a linear function of tensors with respect to a particular tensor A is equal to the linear function where the tensor A has been removed, as expected. Some examples of this operation are presented in figure 2.5 [9].

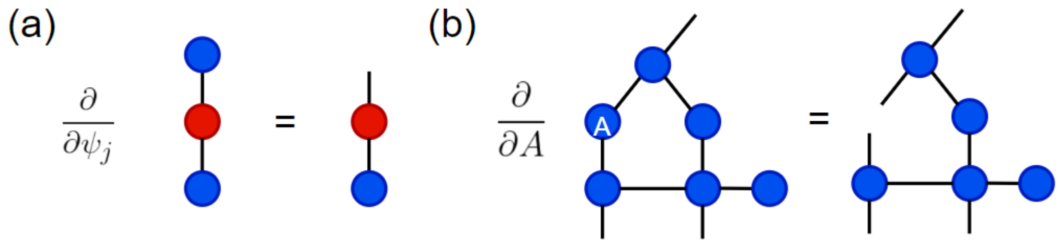

(a)
$\frac{\partial}{\partial \psi_j}$
=
(b)
$\frac{\partial}{\partial A}$
A
=

Figure 2.5: a)Graphical representation of $\frac{\partial H}{\partial \psi_j} \neq 0$; b) Graphical representation of a generic partial derivation with respect to a tensor A. Figure inspired by Ref. [9].

Furthermore, the gauging operation is a useful operation in tensor network algorithms and by using the definition of unitary operator, one can insert an unitary operator between two contracted tensors and change the tensors entries without changing the tensor structure.

2.3.2 Matrix Product State

One of the most well-known tensor networks is the Matrix Product State or MPS and essentially it is equivalent to the state produced by a DMRG [9]. A MPS or a train tensor can be expressed as $T^{s_1 s_2 s_3 s_4 s_5 s_6} = \sum_\alpha A^{s_1}_{\alpha_1} A^{s_2}_{\alpha_2} A^{s_3}_{\alpha_3} A^{s_4}_{\alpha_4} A^{s_5}_{\alpha_5} A^{s_6}_{\alpha_6}$, for a tensor T with six indices or in the diagram notation showed in figure 2.6 [20].

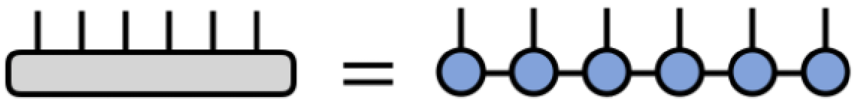


Figure 2.6: Graphical representation of the diagram notation of the vector T. Figure taken from Ref. [20].

In the MPS approach one can assume that the system can be described with an auxiliary dimension that is bigger than one but does not grown exponential with N. This parameter is called the bond dimension of the tensor and it can be thought of as the parameter that plays the role of the cut dimension in the RG methods. If we consider a tensor with N indices, each with d dimension, then normally one would need d^N parameters to fully represent the tensor.

Nonetheless, if we represent the tensor using MPS networks with a m bond dimension then it would only require Ndm^3 parameters to represent the tensor. We can reduce even further the computational cost by using MPS gauges. For more information on this subject consult [9].

The MPS representation of a tensor allows the efficient performance of operations on a large tensor by manipulating much smaller factors that compose the MPS tensor. This algorithm is very useful for dealing with ground states of the 1D gapped systems. For example, if we consider a many-body quantum system of N 1/2 spin particles. This can be seen as a tensor with N indices, where each index has two possible values. Hence, one would need $O(2^N)$ coefficients to represent this tensor. The advantage of using MPS is that one can replace this tensor by a network of connected tensors in order to reduce the number of coefficients needed drastically [9].

An example of the application of MPS to an arbitrary 4-party state $|\psi>$ is given by [21] and it states that the key component is to use SVD in an iterative way. As shown in figure 2.7, we choose a bipartition that separates one edge from the others and so on until we end up with a 1D representation of the state.

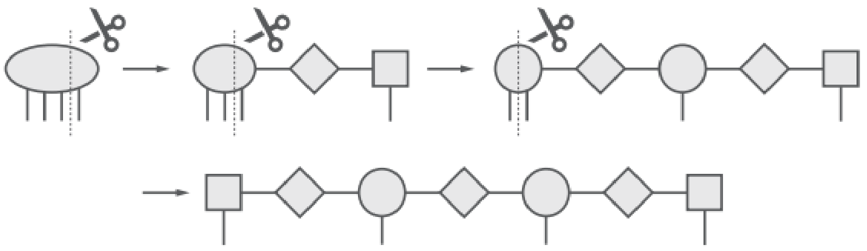


Figure 2.7: Graphical representation of the MPS applied to a 4-party state. Figure taken from Ref. [21].

This examples is just an illustration since it is not practical. Normally, MPS-generating algorithms are used. One example is, as mentioned before, the DMRG. More examples and applications are given in [21] for the interested reader.

2.3.3 Matrix Product Operator

The Matrix Product Operator or MPO arises from the application of the concepts involved in the MPS to operators, where one can use the concepts mention in 2.3.2 to represent operators and apply the MPO to the wave function represented by the MPS. The MPO have the ability to represent Hamiltonians in a very compact way making it easier to compute the energy expectation value. Moreover, MPO also describe very efficiently many-body quantum systems at finite temperatures and can encode several operators that can be used to perform different tasks [9].

A MPO can takes the form of $M^{s_1s_2s_3s_4s_5s_6}_{s'_1s'_2s'_3s'_4s'_5s'_6} = \sum_\alpha A^{s_1\alpha_1}_{s'_1} A^{s_2\alpha_2}_{\alpha_1s'_2} A^{s_3\alpha_3}_{\alpha_2s'_3} A^{s_4\alpha_4}_{\alpha_3s'_4} A^{s_5\alpha_5}_{\alpha_4s'_5} A^{s_6\alpha_6}_{\alpha_5s'_6}$ or in the diagram from represented in figure 2.8 [22].

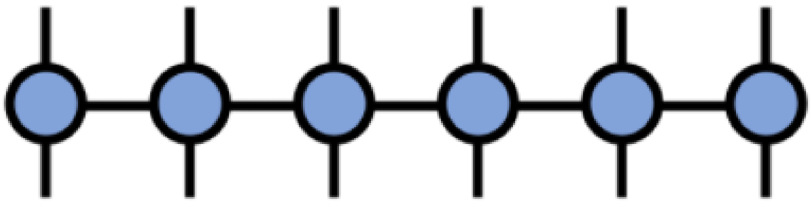


Figure 2.8: Graphical representation of a MPO. Figure taken from Ref. [22].

One can perform arithmetic operations with MPOs such as summation and products. Suppose one has two MPOs, $\hat{A}$ and $\hat{B}$ and we want to know the sum $\hat{A}+\hat{B}=\hat{R}$. Considering the MPO components A_i, B_i and R_i, where i ranges from 1 to L, and treating them as matrices of operators one obtains the following equations.

$$R_1 = \begin{pmatrix} A_1 & B_1 \end{pmatrix} \tag{2.2}$$

$$R_{1<i<L} = \left(\begin{smallmatrix} A_i & 0 \\ 0 & B_i \end{smallmatrix}\right) \tag{2.3}$$

$$R_L = \begin{pmatrix} A_L \\ B_L \end{pmatrix} \tag{2.4}$$

When making the product of two MPOs, $\hat{A}$ and $\hat{B}$, where $\hat{A}\hat{B}=\hat{R}$. In this operation, the lower index of each A_i is contracted with the upper index of each B_i. The left and the right index of the tensors are merged into one and it results in a MPO with bond dimensions $w_i^r = w_i^a * w_i^b$. A graphical representation is shown in figure 2.9.

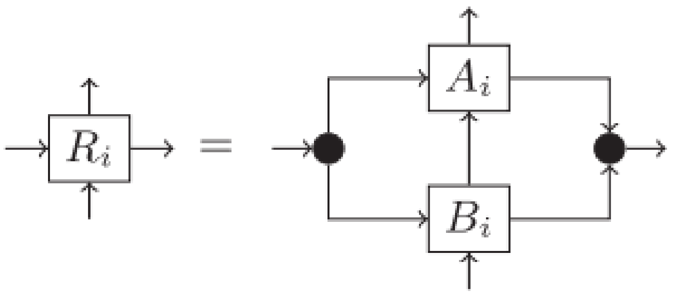


Figure 2.9: Graphical representation of the product of two MPOs for a tensor ona single site i. Figure taken from Ref. [23].

2.3.4 Projected Entangled Pair States

Projected Entangled Pair States or PEPS are 2D array tensors and they are a generalization of MPS tensor networks from 1D. PEPS tensor networks present a way of parameterizing many-body wave functions on any lattice which provides a way of writing a complex wave function in a compress form, similarly to MPS. However, the main difference is that PEPS can represent a diversity of ground states of interacting systems. This means that higher dimensions correlated many-body systems can be represented by PEPS. Many examples of PEPS such as the String-net model and The RVB state are given in [24]. An example of a 2D PEPS is given in figure 2.10.

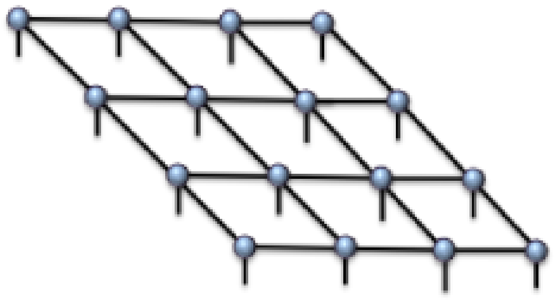


Figure 2.10: Graphical representation of a 4x4 PEPS with open boundary conditions. Figure taken from Ref. [25].

For higher dimension systems, i.e., for PEPS there is the necessity to impose that all the spacial directions of the lattice are translational invariant. Furthermore, PEPS also satisfy the area-law scaling of the entanglement theory, where the entropy of the ground states of the local Hamiltonians of spin scales with the boundary of the block instead of being an extensive property [26].

In order to extract information from the TN, one must compute the expectation values of the local variables. In PEPS it is necessary to use approximations to compute the expectation values such as the DMRG methods or finite systems or the Boundary MPS methods for infinite systems [25]. One downside of PEPS is that they have an exponential computational cost. More ensign on PEPS is given by [24]-[27].

2.3.5 Tree Tensor Networks

Tree Tensor Networks or TTN are structures that have a shape of a tree and have a finite correlation length and they satisfy the 1D area-law. In other words, TNN is also a generalization of MPS but, unlike PEPS, it is a graph with no loops, like a tree. This tree shape brings some advantages such as the capacity to efficiently compute the partition function. An example of a TNN is given in figure 2.11 [28].

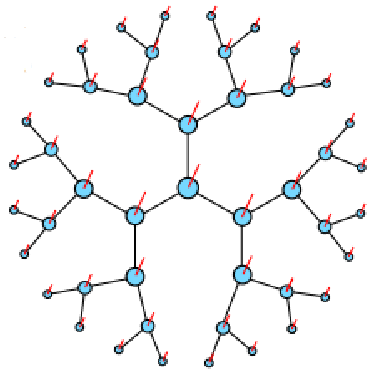


Figure 2.11: Graphical illustration of a TNN. Figure taken from Ref. [28].

2.3.6 Multiscale Entanglement Renormalization Ansatz

Multiscale Entanglemnet Renormalization Ansatz or MERA are structures very similar to TNN but they include disentanglers as shown in figure 2.12, which are unitary operators that account for entanglement amongst neighbouring sites. This disentanglers give MERA the capacity to handle the entanglement entropy of 1D systems [9].

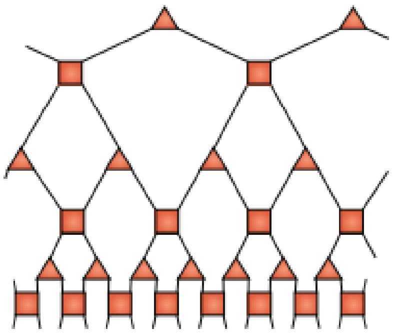


Figure 2.12: Graphical illustration of a 1D MERA. Figure taken from Ref. [9].

A MERA is a network of isomorphic tensors in D+1 dimensions, where the extra dimension is related to the RG flow,[29] and is very useful to solve problems such as finding the ground state of a strongly correlated Hamiltonian, finding a quantum circuit in quantum computing and finding the time evolution of a given quantum state.

AdS/CFT is a conjecture correspondence between D-dimensional conformal field theories (CFT) in Minkowski space and D+1-dimensional asymptotically anti-de Sitter (AdS) spacetimes. As mention before, tensor networks play a crucial roll in the study of many-body systems, in MERA's case CTFs. More about this subject can be found in [30]. In conclusion, MERA provides a concrete implementation of the emergence of spacetime, in the form of a correspondence boundary and bulk regions reminiscent of AdS/CFT. One can think of the tensor network as a quantum circuit that can start running from the top, where it starts with a simple input state and it builds the boundary state, or from the bottom, where the boundary state is renormalized by coarse-graining [30].

Chapter 3

Problem Statement

When attempting to describe a quantum many body system, one must take into account the dimensionality of the problem. If we consider that all particles of the system are independent of each other then the problem in hands has a dimension $D = v^N$, where v is the physical dimension of the single-particle phase space, i.e., the single-particle degree of freedom, and N the number of particles in the system, meaning we would need D dimensions to explain our system. Considering the system shown in figure 3.1, since we have a system with 8 particles, one would need $D = v^8$ to explain it.

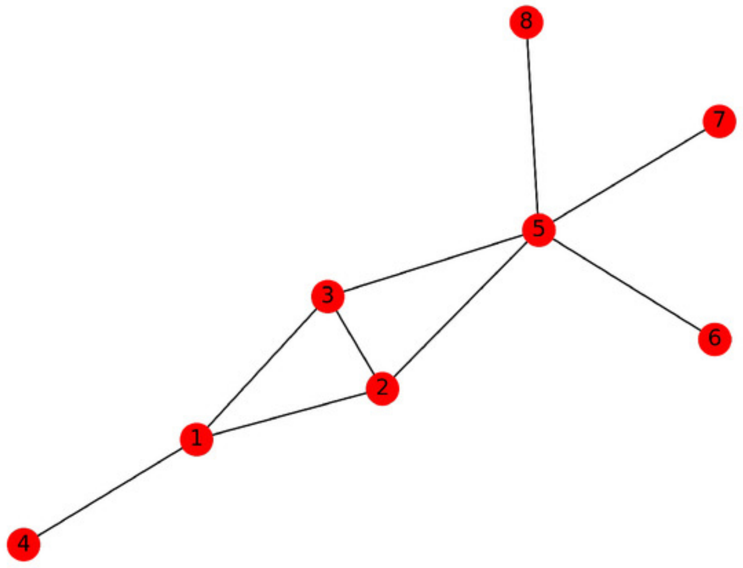


Figure 3.1: Example of a small complex network. Figure taken from Ref. [31].

Nonetheless, in real quantum systems, the particles can be dependent of each other and have entanglement. This entanglement by itself reduces the dimensions required to describe the system and these quantum systems get grouped in a small part of the Hilbert space. To better understand this phenomenon one must first understand the concept of generalized coordinates from analytical mechanics.

In analytical mechanics, when one is studying the dynamics of bodies, the choice of the right set of coordinates is very important in order to simplify the problem at hands. "Generalized coordinates are defined to be a set of convenient coordinates, usually independent of one another, used to describe the configuration of a particular system" [32]. One can think of a simple example such as the motion of a pendulum.

Suppose we have a particle of mass m in free fall. It is considered that the particle is moving in the y-direction. As a consequence, it is only needed one coordinate, the height of the particle, to describe the motion of the particle and obtain the gravitational potential for example. In this case, the number of degrees of freedom of our system is equal to 2, one space coordinate and its time derivative. Considering now the case of the pendulum with length l and a bob mass m which is a very well known case studied in any classical mechanics classroom and suppose one is using the Cartesian coordinates to obtain the Lagrangian of the system, then the desired result would be given by equation 3.1, with the constrain on the motion of the system given by $x^2 + y^2 = l^2$ [33].

$$\mathcal{L} = T - U = \frac{1}{2}m\dot{x}^2 + \frac{1}{2}m\dot{y}^2 - mgy \tag{3.1}$$

As it can be seen, this system requires 2 coordinates to be described in Cartesian coordinates. Nevertheless, as seen in [34], to describe the motion of a pendulum, it is more useful to work with the angle, θ, that the pendulum makes with the y axis, as shown in figure 3.2.

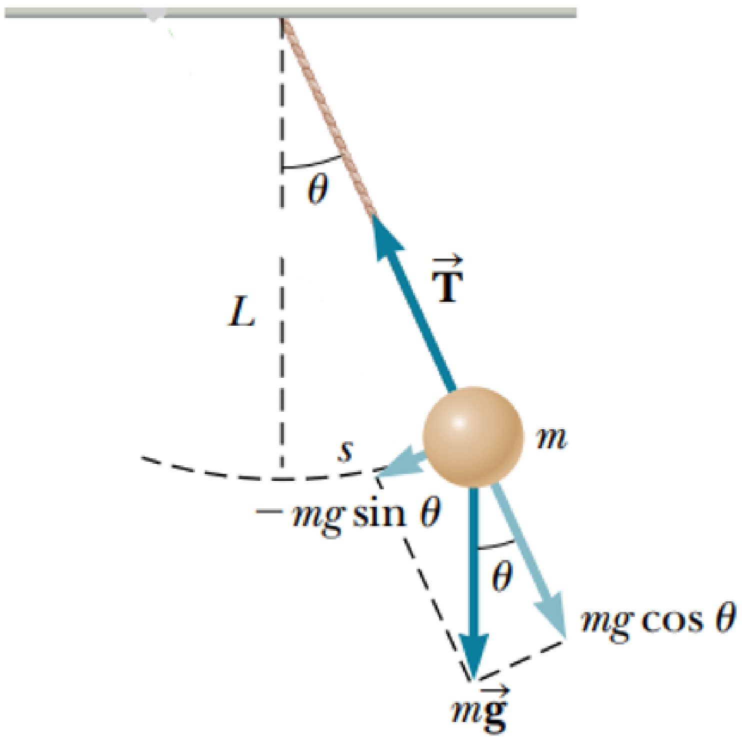


Figure 3.2: A simple pendulum of mass m and length l. Figure taken from Ref. [34].

Making the following substitutions $x = l\sin\theta$ and $y = l(1-\cos\theta)$ and deriving x and y, then the Lagrangian becomes the equation shown in 3.2

$$\mathcal{L} = \frac{1}{2}ml^2\dot{\theta}^2 - mgl(1-\cos\theta) \tag{3.2}$$

Analyzing 3.2, one can conclude that even thought the number of degrees of freedom remains the same, the complexity of the problem is reduced [**45**]. Similarly to this example, in our system with N entangled particles, the entanglement actually reduces the degrees of freedom of the system and consequently the dimensionality of our problem.

Now that we know how to reduce the number of dimensions needed, the next step in solving the problem in hands is to find a way to translate our network, for example, the one shown in figure 3.1, in a low-dimension space. The way to do it is through the connectivity of each particle represented by the nodes in the network.

The connectivity of a node is given by how many connections that node has. Figure 3.1 gives a good example to understand the connectivity of a complex network. The nodes 4, 6, 7 and 8 have connectivity equal to one while the first, second and third node have a connectivity equal to 3. The node with the biggest connectivity is number 5 with five connections. By using the connectivity of each particle, one is projecting the network into the desired low dimension space since the dimensions of this particular space only depend on this connectivity.

If one has N nodes, each one can have v states. For example, if our system is formed by N $\frac{1}{2}$ spin electrons, then, each electron can be up or down meaning the $v = 2$ possible states for each particle and one would need $D = 2^N$ dimensions to describe the desired network. Similarly to the generalized coordinates in the pendulum, by using the connectivity, one is reducing the dimension of the space in which we are projecting the complex network leaving it with a dimension b, also known as the bond dimension.

Considering the case where the particles of the system are not entangled with each other and $v = 1$, then one would have 2^N space. Nonetheless, since we are dealing with entangled particles our space becomes b dimensional, M^b. The only problem now is that we do not know the value of b.

Through entanglement between the particles and the decomposition of any given matrix using the SVD method into $U\Sigma V^T$, where both U and V^T are unitary matrices, we transform our space from a 2^N space to a 2^b space.

Chapter 4

Literature Review of MPS

4.1 Tensor Networks

First of all, before entering the algorithm itself, one must understand and study the basics of tensor networks. A quick introduction was given in chapter 2.3. Nonetheless, in this chapter, we aim to fully explain and understand tensor networks.

As stated by the motivation behind this project, despite us knowing the equations to solve most of the quantum mechanics problems, solving those equations is much harder and complex since the wave function, regardless of the system, has very high dimensions which requires exponential complexity. Be that as it may, this high level of complexity is, to a limit, an illusion, since, despite of the infinite dimensions of the Hilbert space, physical systems in nature do not explore all these possibilities but rather focus on a small part of this space as it was stated in the first part of chapter 3. According to the Principle of Locality [35], the lower energy states of a physical system, or the grounds states, have very little entanglement, meaning this correlation only influences the nearby neighbors of each particle. Considering a general wave function, $|\psi >= \sum_{n_1 n_2 n_3} \psi^{n_1 n_2 n_3} |n_1 n_2 n_3 >$, where $|n >$ can represent spins $\frac{1}{2}$ e.g. $|\uparrow>, |\downarrow>$, or particles, e.g. $|1 >$, $|0 >$. This wave function can be represented through a tensor in its algebraic form by $\psi^{n_1 n_2 n_3}$ or in its graphical form by the left side of the image 2.6, where, in this case, we would only have three legs since our tensor has only three indices.

Tensor networks provide a language that allows us to work only with the relevant physical states of a quantum system. As presented in chapter 2, there are different types of tensor networks with different geometries of entanglement. A tensor is a mathematical concept that generalizes the idea of functions of multiple parameters that are linear to each other. A tensor network is simply a collection of tensors that are connected by contractions, that simplify a complicated quantum state essentially through the compression of data that preserves the most salient properties of the quantum state [36].

4.1.1 Low Entanglement States

Considering a system of two spins, then, one can say that it has no entanglement if the wave function of the system can be factorized into the product of amplitudes for the first and second spin.

$$\psi^{n_1 n_2} = A^{n_1} A^{n_2} \tag{4.1}$$

If one thinks about it in a physical way, then, this can be translated into doing measurements on either the first or the second spin and obtaining a result that has zero correlations or, in other words, the measurements can be done independently of each other. In general, a quantum system has low entanglement, therefore, one cannot factorize the amplitudes of the wave function simply into their product. Nonetheless the wave function amplitude can be written as a sum of the product of amplitudes as shown in the equation 4.2.

$$\psi^{n_1 n_2} = \sum_i A_i^{n_1} A_i^{n_2} \tag{4.2}$$

Equation 4.2 is a generalisation of 4.1, where in 4.1 the sum only has one term meaning the system has no entanglement. In equation 4.2, there is a sum over i terms. A state that has a low entanglement only has a few terms on the sum since it is the index i that is generating the entanglement, consequently a system with low entanglement has a small number of terms in the sum.

4.2 Matrix Product State

The equation 4.2 actually represents the matrix product state of a system with 2 spins. However, the matrix product state of a general system with l spins would be represented as the sum of products of amplitudes on each of the subsystems as it is shown by 4.3

$$\psi^{n_1 n_2 n_3 \ldots n_l} = \sum_i A_{i_1}^{n_1} A_{i_1 i_2}^{n_2} A_{i_2 i_3}^{n_3} \ldots A_{i_l}^{n_l} \qquad , \tag{4.3}$$

where the additional indexes i_1,i_2, ..., i_l are called bond dimension, which is the dimension of the space where one is projecting the network in a MPS form as stated in chapter 3. In order to calculate the components of $|\psi>$ one simply needs to calculate the product of matrices, hence the name matrix product state.

This particular representation implies a one dimension structure and each site can be entangle either to the right or to the left with the exception of the first and last, that can only be entangle to the right and to the left, respectively, as it was explained in 2.3.2. One could argue that, in this representation, there is no entanglement between site 1 and site 3. Nevertheless, since site 1 and site 2 are entangled, and site 2 and site 3 are also entangled, then there is some entanglement communicated along [35].

It is important to stress out that a tensor is just a group of numbers where its indexes are defined as the labels labeling the elements. Considering the explanation given in 2.3.1 about the diagram representation of tensors, one is now ready to understand tensor networks, which is defined as the contraction of many tensors [28].

4.2.1 Singular Value Decomposition

Recalling the conclusion of chapter 3, the way to transform the space in which we are projecting the network into 2^b is through the Singular Value Decomposition. Hence, one starts with a tensor that is untreatable in regards to dimensions and transforms it into a tensor that is computable by decomposing it through the application of the SVD into each site of the tensor. Naturally, the first thing one must learn and understand in order to study the MPS algorithm is the singular value decomposition, SVD, also known as Schmidt decomposition. The Spectral Theorem for symmetric matrices states that an $m \times n$ symmetric matrix can be written as $A = U\Sigma V^t$ for an orthogonal matrix U and a diagonal matrix Σ with entries σ_i. In linear algebra, the SVD is the decomposition of a matrix, real or complex and it seeks to generalize the Spectral Theorem to nonsymmetric matrices. Given the matrix A, one can then rewrite it as

$$A = \sum_{j=1}^{m} \sigma_j u_i v_j^t = U\Sigma V^t \qquad , \tag{4.4}$$

for unitary U and V matrices and for a diagonal matrix. The SVD can be applied to any given matrix A and the proof can be found at [37]. If A is a m×n matrix, therefore U would be m×m matrix, Σ a m×n matrix and V a n×n matrix. The SVD of the matrix A would then become the expression shown in 4.5

$$A = (u_1, \dots, u_k, u_{k+1}, \dots, u_m) \begin{bmatrix} \sigma_1 & \dots & 0 & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & \sigma_k & 0 & \dots & 0 \\ 0 & \dots & 0 & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & 0 & \dots & 0 \end{bmatrix} \begin{bmatrix} v_1^t \\ \vdots \\ v_k^t \\ v_{k+1}^t \\ \vdots \\ v_n^t \end{bmatrix} \qquad , \tag{4.5}$$

where it is assumed that $k < min(m,n)$ [37]. Since the matrix A only has k linear independent columns, there will only be k nonzero values in the Σ matrix due to the fact that the rank of the A matrix has to be k. The vectors of the matrix U are called the left singular vectors and this vectors are hierarchically arranged meaning that the vector σ_1 is more important than the rest of the vectors that make up the U matrix in terms of its ability to describe variance in the columns of the matrix A. The vectors of the matrix V are called the right singular vectors and contains information about the row of the matrix A.The scalars σ that make up the matrix Σ are called the singular values and are also hierarchically ordered where its magnitude decreases, i.e. $\sigma_1 \geq \sigma_2 \geq \sigma_k \geq 0$. Accordingly, the first column of U and the first column of V that corresponds to σ_1 are more important than the other columns in describing the information in the matrix A, and this relative importance is given by the respective singular value σ [38].

If one actually expands the expression 4.5, it can easily be seen that the matrix A, that is equal to $U\Sigma V$, as stated in 4.4, is also equal to $\hat{U}\hat{\Sigma}\hat{V}^t$, as one can see in 4.6

$$A = \sigma_1 u_1 v_1^t + \sigma_2 u_2 v_2^t + \cdots + \sigma_k u_k v_k^t + 0 \qquad , \tag{4.6}$$

where $\hat{U}$ is a m×k matrix,$\hat{\Sigma}$ is a k×k matrix and $\hat{V^t}$ is a k×n matrix. This SVD form is known as the reduced SVD or the "economic" SVD.

Nonetheless, if most of the information of A is captured in the first few singular values, i.e. if we have negligibly σ, with low energy, then one can truncate the SVD at a rank R where we would remain with the first R columns of U and the first R rows of V and a R×R sub-matrix of Σ. In this case, the best approximation of A with a rank R would be given by 4.7

$$A \approx \tilde{U}\tilde{\Sigma}\tilde{V^T} \qquad , \tag{4.7}$$

where $\tilde{U}$ is a m×R matrix, $\tilde{\Sigma}$ is a R×R matrix and $\tilde{V}$ is a R×n matrix [39].

This approximation is known as the Eckart-Young Theorem and it guarantees that the best R approximation to the matrix A, i.e., the best approximation of rank R of the matrix A, is given by the first R columns of the resulting matrices from the SVD. For the more curious reader, the proof can be found in 9.2.3 of [37]. The question now is how do we choose our R in order to truncate the SVD without compromising the accuracy, but at the same time, reducing the complexity or the dimensions of our SVD [40]. In this thesis, we will not go into further detail. However, the goal is to choose R in a way that we keep all the relevant information about A without keeping the noise and there is an optimal value for R that is explored in [41]. An example of the application of the SVD is given in the chapter 2.2.1 of [28], where the truncation error in tensor networks algorithms is also mentioned and is given by 4.8

$$\epsilon = \sqrt{\sum_{a=\chi'}^{\chi-1} \lambda_a^2} \qquad , \tag{4.8}$$

where χ is the rank of the matrix, χ' the optimal rank approximation and λ is called the singular value spectrum.

4.2.2 MPS Gauge

If one introduces an arbitrary matrix G and its inverse, G^{-1}, it becomes obvious that $GG^{-1} = 1$, as represented in the figure 4.1. In the graphical form this just becomes a line or the Kronecker Delta.

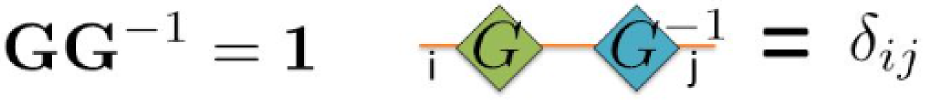


Figure 4.1: Graphical representation of GG^{-1}. Figure taken from Ref. [35].

Now if one considers a general MPS, like the one shown in the figure 4.2, one can insert an arbitrary product of GG^{-1} in between any two tensors and it would still be the same product. This is called the gauge degree of freedom. Nonetheless, now one can multiple G onto the left or the first tensor, and multiple G^{-1} onto the right or the second tensor and obtain a different matrix product state representation of the same state.

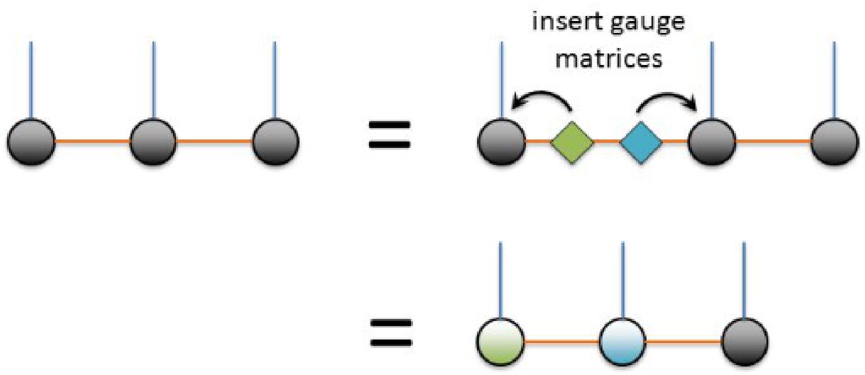


Figure 4.2: Insertion of GG^{-1} in between the first and second tensor of a general MPS. Figure taken from Ref. [35].

4.2.3 From a general quantum state to a MPS

Since our quantum state is represented by the connectivity of the complex network, now one must transform it into a MPS in order to obtain 2^b and as we saw in the previous chapter, the way to do it is through the SVD. Nevertheless, the algorithm itself is not enough, so here we give a general example of how to obtain a MPS from a general quantum state.

Considering one has a one dimensional lattice of L sites, each one with a dimension d, then the quantum state is described by 4.9

$$|\psi> = \sum_{\sigma_1,\dots,\sigma_L} c_{\sigma_1,\dots,\sigma_L} |\sigma_1,\dots,\sigma_L> \qquad , \tag{4.9}$$

where $c_{\sigma_1,\dots,\sigma_L}$ represent the coefficients associated with each site. Assuming this state is normalized, one has essentially four ways of writing an MPS. The one we will describe in detail is the Left-canonical matrix product state. The first step is to reshape the vector that represents the state and has d^L components and transform it into a $d \times d^{L-1}$ matrix, ψ. The coefficients of the state relate to ψ through $\psi_{\sigma_1,\dots,\sigma_L} = c_{\sigma_1,\dots,\sigma_L}$. The second step is to apply the SVD to the matrix ψ. Recalling 4.4, one would get the following SVD when applied to ψ 4.10.

$$c_{\sigma_1,\ldots,\sigma_L} = \psi_{\sigma_1,(\sigma_2,\ldots,\sigma_L)} = \sum_{a_1}^{r_1} U_{\sigma_1,a_1} S_{a_1,a_1} (V^t)_{a_1,(\sigma_2,\ldots,\sigma_L)} \tag{4.10}$$

The rank, r_1, is smaller or equal to the original rank, d. Nevertheless, now the matrix ψ has a dimension of $r_1 d \times d^{L-2}$. One must do this process in a repetitively way, i.e., do a set of singular value decomposition in order to arrive at the MPS. After all the SVDs are applied the result one gets is represented in 4.11. A much more detail demonstration of this derivation is presented in [42].

$$c_{\sigma_1,\ldots,\sigma_L} = \sum_{a_1,\ldots,a_L} A^{\sigma_1}_{a_1} A^{\sigma_2}_{a_1,a_2} \ldots A^{\sigma_{L-1}}_{a_{L-2},a_{L-1}} A^{\sigma_L}_{a_{L-1}} \tag{4.11}$$

And replacing 4.11 into 4.9, one gets the final result 4.12.

$$|\psi >= \sum_{\sigma_1,\ldots,\sigma_L} A^{\sigma_1} A^{\sigma_2} A^{\sigma_{L-1}} A^{\sigma_L} |\sigma_1, \ldots, \sigma_L > \tag{4.12}$$

In other words, by applying the SVD to the original vector that represents the quantum state, ψ, one is basically cutting the system between the first site of the lattice and the rest of the vector. The SVD pulls apart this first site from the tensor and originates the first site of the MPS. After the first SVD, one gets one tensor isolated from the rest of ψ, connected by the singular value from the SVD, as shown in the figure 4.3

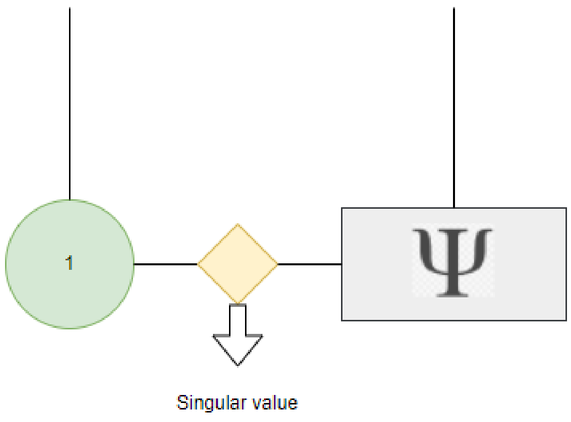


Figure 4.3: Graphical representation of the first tensor, the singular value and ψ after the SVD.

By applying consecutively the SVD, this process is repeated and the result would be a train of tensors labeled as 1,…, L, since we have a lattice with L sites and i between these tensors there would be the singular values. This form of representing the MPS is called the "Vidal" form. Nonetheless, the singular values work as a gauge, hence one could just absorb them into the tensors itself. That means these singular values are absorbed to the right, meaning the singular value represented in the figure 4.3 would be absorbed into ψ or into the second tensor if one has already done more iterations of the SVD. This form of representing the MPS is called the left-canonical matrix product state and its graphical representation is shown in the figure 4.4.

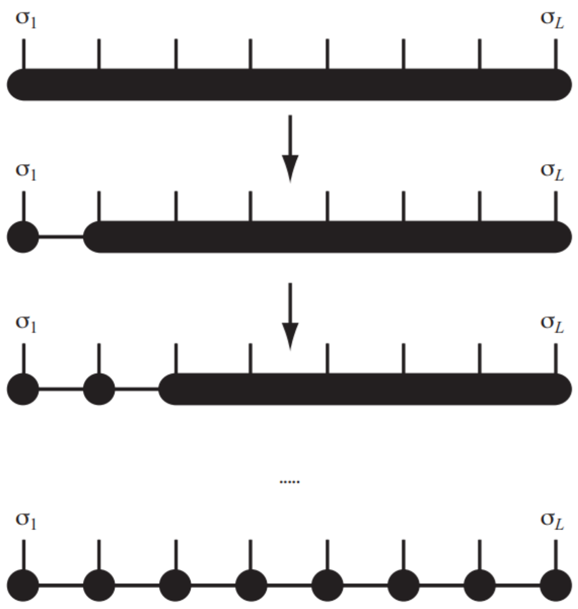


Figure 4.4: Graphical representation of a construction of a MPS of an general quantum state in an iterative way by successive applications of the SVD; Visual interpretation of the equation 4.12. Figure taken from Ref. [42].

Similarly to the process described above, one can start applying the SVD from the right instead of the left and obtain an equivalent MPS. This form of representing the MPS is called the right-canonical form. The fourth way is called the mixed-canonical matrix product state and one uses both the left-canonical and the right-canonical form mixing both of them. In the mix-canonical MPS, the final form of the MPS is similar to the one shown in the figure 4.4. Nonetheless, since it is a mix of both right and left canonical form, one gets the same train of tensors, but with a diamond shape in the middle, that represents the diagonal singular value matrix. Furthermore, the matrices to the left of the diamond are left-normalized and the matrices to the right are right-normalized according to the type of canonical form it was used to obtain the MPS. A graphical representation of the mix-canonical form can be found at 4.1.3. of [42]. Moreover, in the DMRG form, which is very similar to the mix-canonical form, one chooses a particular site, normally in the middle, i.e. if the MPS has three sites, one would choose the site number 2 and every singular value to the left would be absorbed into the first site, and every singular value to the right would be absorbed into the third site. When using this form, every tensor to the left or to the right of the chosen site gives the Kronecker Delta when contracted with themselves.

One way to make tensor networks and MPSs less abstract and more practical is to use MPSs to represent non-trivial physical states such as the Affleck-Kennedy-Lieb-Tasaki state, which is a model that is a generalization of a spin-1 Heisenberg model. Since it falls out of the scope of the project, we will not go into further detail. However, the curious reader will be redirected to 2.2.3. of [28].

4.3 Matrix Product Operator

As stated in chapter 2.3.3, the concept of the Matrix Product Operator emerges from the concepts involved in MPS's. Analogous to the MPS, where one can write down a state as a product of amplitudes of each site, one can also write down an general operator, acting on L sites, as a product of operators acting on each individual site. A MPO can be viewed as a way of writing a general operator as an entangled product of operators. Instead of using equation 4.3 to represent a wave function, one can an useful generalization to represent a MPO as shown in equation 4.13 [43].

$$\sum_{s_i;s'_i} M_{s'_1 s1} M_{s'_2 s2} \dots M_{s'_L sL} |s'_1 >< s1| \otimes |s2 >< s2| \dots \tag{4.13}$$

The figure 4.5 shows a graphical representation of a Matrix Product State, where one can clearly see two vertical lines. These lines represent the physical states of M, one for the ingoing physical state and one for the outgoing state. MPO's can be used to represent several non-trivial physical models such as an Hamiltonian or time evolution operators. In [44], the construction of the MPO for the Ising Hamiltonian is shown in a transverse field.

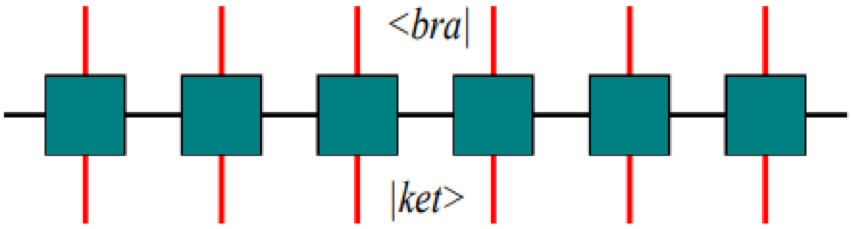


Figure 4.5: Graphical representation of a MPO. Figure taken from Ref. [28].

Considering the Heisenberg Hamiltonian as an example given by equation 4.14, in which the sum, $\sum_{ij}$, is over every site of the lattice and the factor $\frac{1}{2}$ is a correction factor [45].

$$H = \frac{1}{2} \sum_{ij} J_{ij} \vec{S_i} \vec{S_j} \tag{4.14}$$

Writing this Hamiltonian as a MPO, where i and j represent the spins of the nearest neighbors that are coupled together, and writing it as a sum of products of terms in each of the sites, then the bond dimension of the MPO would be equivalent to the number of terms in the Hamiltonian acting with each other.Considering one has a system with 4 sites, then the MPO of this system would be the one shown in figure 4.6.

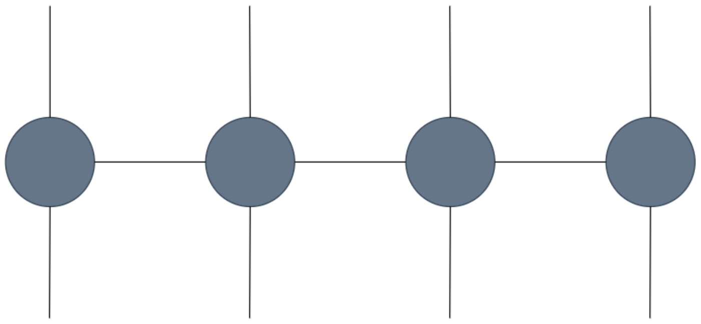


Figure 4.6: Graphical representation of a MPO of a 4 site system.

In order to determine the bond dimension of this MPO, one divides it in the middle, i.e. between site 2 and site 3. Consequently, one would end up with two systems, the left system and the right system. There would be a left Hamiltonian describing the left system and a right Hamiltonian describing the right system and, because we spited the system between the second and third site there would also be a term representing the coupling between the left and the right system. To describe this system, one would need 5 terms as it is shown in equation 4.15, making clear that the bond dimension of the MPO is equal to 5.

$$H_L \otimes 1_R + 1_L \otimes H_R + \sum_{\alpha=x,y,z} S_2^\alpha S_3^\alpha \tag{4.15}$$

An operator acting on a state or in other words a MPO acting on a MPS, gives simply a MPS. Nonetheless, the dimension of the MPS is not the same as the original MPS. If the MPO has a dimension D_1 and the MPS has a dimension D_2, then the dimension of the resulting MPS, D_3, is going to be the product of both dimensions, $D_3 = D_1 \times D_2$, which means that the entanglement of the original state increased.

Since operating on a MPS increases its bond dimension, for example adding two MPSs or applying an operator to it, one must find a way to decrease the bond dimension and as mention in 4.2.1 one way to do it is by SVD compression. One must write the MPS in the Vidal form and truncate the singular values. Another way to achieve the desired result is through variational compression, where one solves a minimization problem following the gradient algorithm. Nevertheless, if one works with the MPS in the mixed canonical form, there is no need to differentiate the sites of the MPS and follow the gradient, one just uses the Density-Matrix Renormalization Group or DMRG built in the sweep algorithm where the DMRG is applied site by site [46]. The difference between the SVD compression and the variational compression is that, in the latest one, the optimization of each site depends on the values of the remaining sites, i.e., this optimization uses the full environment, while in the SVD compression, the tensors are optimized independently meaning one is doing a local update. Despite not being the optimal choice, the SVD compression is preferred since it has less computational cost.

4.4 Ground state calculations with MPS

In order to find the ground state of an Hamiltonian, one must find the best approximation to that state through the MPS that minimizes the energy, given in equation 4.16.

$$E = \frac{<\psi|\hat{H}|\psi>}{<\psi|\psi>} \tag{4.16}$$

The first step would be to represent the Hamiltonian as a MPO, which we saw in the previous section. Nonetheless, a richer approach to the subject is given in chapter 6 of [42]. Once the Hamiltonian is represented in a MPO form, one applies the MPO to the MPS in a mixed canonical state. To find the MPS that minimizes E. As stated in [47], by introducing a Lagrangian multiplier E, one must solve the minimization of the function shown in equation 4.17

$$E[|\psi>] =<\psi|\hat{H}|\psi> -E <\psi|\psi> \tag{4.17}$$

where $|\psi>$ will be the ground state and E the ground state energy. Solving this minimization is a NP-hard problem, meaning it would be possible, given the answer, to verify it in polynomial time. By using the MPS notation, it is possible to use an algorithm based on the DMRG described in chapter 3 of [47] to find the ground state of a given Hamiltonian, where $|\psi>$ will be the ground state and E the ground state energy.

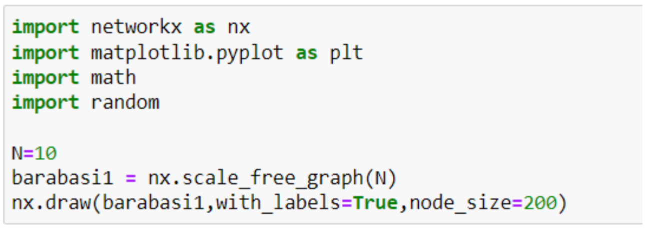
Chapter 5

Methods

As we progressed in understanding the library and all its potential, we got closer to the goal, which was to generate a Barabási network. Before getting into the Barabási network, we generated a Erdős–Rényi network. We first understood what a Erdős–Rényi network was and developed our own code to generate this type of network and only then did we used the library's function.

The Erdős–Rényi model is a model used for generating random graphs. In this model, one can choose the number of nodes of the graph and the probability of creating a connection, independently from the rest of the connections. We proceeded to study this model since it provides a random graph and it was a way to slowly understand and get to the free scaled graphs, which is the case of a Barabási network.

The Barabási-Albert model is a model that generates free scaled random graphs and has a power-law degree distribution. This model differs from simpler networks and resemble real life one's, like the internet, for its growth and preferential attachment characteristics. In this model, the nodes are added one by one and the added node has a certain probability of connecting to the existing nodes. In other words, a new node has a certain probability of connecting with one of the already existing nodes and that probability scales with the number of connections the node already has. As it is expected, if a node already has a high number of edges, then the new node will connect to it with a higher probability than the remaining nodes. As it can be seen in figure 5.1, the node labeled 2 has the most edges, hence, if a new node is added it would connect with it with a probability given by $p_2 = \frac{k_2}{\sum_j k_j}$, where k_2 is the degree of node 2 and the sum is over all nodes [48].



```python
import networkx as nx
import matplotlib.pyplot as plt
import math
import random

N=10
barabasi1 = nx.scale_free_graph(N)
nx.draw(barabasi1,with_labels=True,node_size=200)
```

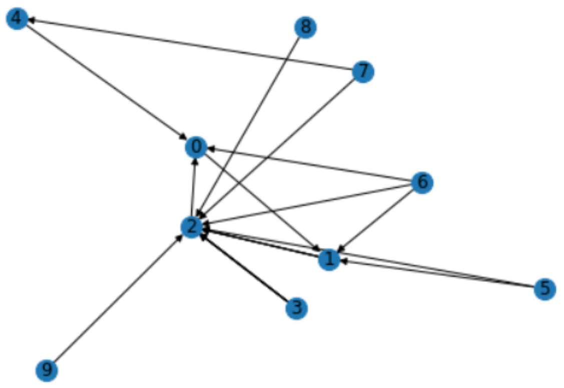
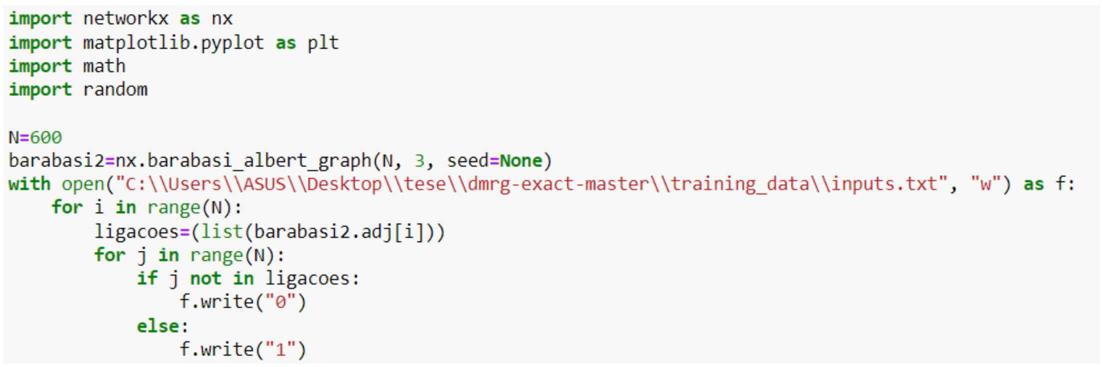


Figure 5.1: Example of a Barabási graph with 10 nodes and the code used to generate it.

Once the graph is generated, we must extract the right information out of it. The characteristic we are interested in is the connectivity of the nodes, as it was explained in chapter 3. In 5.1 the *nx.scale_free_graph()* function was used. However, to generate the one used to obtain the connectivity, we opted for *nx.barabasi_albert_graph()*. Out of the generated network, we used the code showed in figure 5.2 to obtain a list of the connections for each node. The idea behind it was to use the connectivity of each node as an input for the MPS. Theoretically, we would obtain the connections adjacent to node number 1 and run the MPS algorithm and repeat the process for the other nodes. Nonetheless, we obtain the connectivity of all nodes at once and store the information in a text file to be processed later.



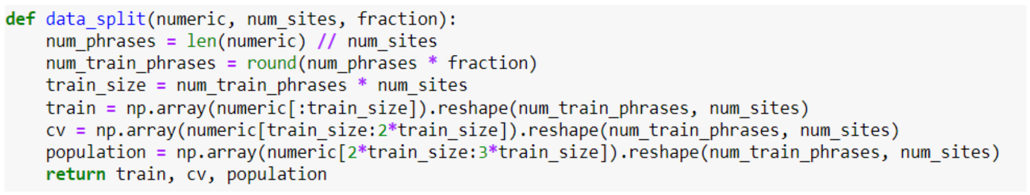
```python
import networkx as nx
import matplotlib.pyplot as plt
import math
import random

N=600
barabasi2=nx.barabasi_albert_graph(N, 3, seed=None)
with open("C:\\Users\\ASUS\\Desktop\\tese\\dmrg-exact-master\\training_data\\inputs.txt", "w") as f:
    for i in range(N):
        ligacoes=(list(barabasi2.adj[i]))
        for j in range(N):
            if j not in ligacoes:
                f.write("0")
            else:
                f.write("1")
```

Figure 5.2: Code used to extract the connections of each node in order to be used as an input for the MPS.

Once we obtain the input, we started working on the MPS algorithm itself. The original code was taken from [49] and we first understood it and adapted it to our goals. All the code used can be found in appendix A As mentioned the first step is to process the data, in this case the connectivity, in order to mold it in such a format that it can be used as an input. Before running the algorithm, one must first set some configurations such as defining the number of sites our

MPS is going to have, the respective bond dimension, the fraction of our data that is going to be used for train and test and the number of sweeps of the DMRG. Some of these configurations came already filled up with the algorithm such as the number of sweeps. Nevertheless, the number of sites had to be change in order to match the number of nodes of our complex network, in this case it was 600 nodes, the test_fraction had to be adapted. The bond dimension was the characteristic we studied in order to determine how it influences the output of the algorithm.

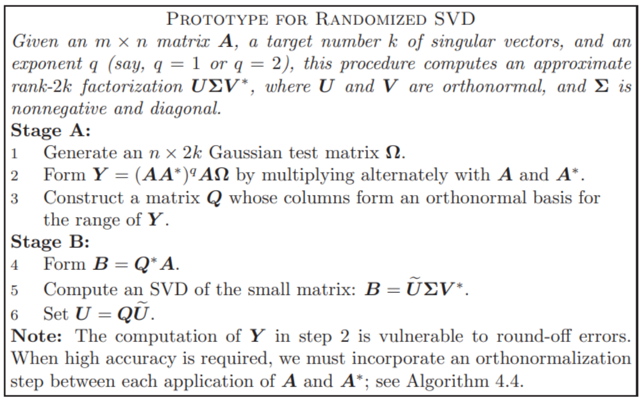
Since the connectivity of each node in regard to the others is either 0, if the node is not connected, or 1, if the nodes indeed share an edge, the dimension of each site of the MPS is going to be 2. This dimension can be seen as a physical dimension, where, in the case of electrons, it would represent either the spin up or the spin down. After importing the data, it was split into 3 sections, the train, cv and test, using the test_fraction configuration and the number of sites.



```python
def data_split(numeric, num_sites, fraction):
    num_phrases = len(numeric) // num_sites
    num_train_phrases = round(num_phrases * fraction)
    train_size = num_train_phrases * num_sites
    train = np.array(numeric[:train_size]).reshape(num_train_phrases, num_sites)
    cv = np.array(numeric[train_size:2*train_size]).reshape(num_train_phrases, num_sites)
    population = np.array(numeric[2*train_size:3*train_size]).reshape(num_train_phrases, num_sites)
    return train, cv, population
```

Figure 5.3: Code used to extract the connections of each node in order to be used as an input for the MPS.

As it can be seen in the code shown in figure 5.3, the data was split into train, cv and population, which later gets renamed to test. Regarding the train batch, and taking into account that the fraction was defined to be 1, i.e. we use all our data from the complex network, the data gets divided it into 3. Which means that the train is composed by the connectivity of the first 200 nodes, and since each node has 600 values, we end up with a list of 200 lists, one for each node, each list with 600 values. In other words, each list of 600 values that corresponds to the connectivity of a single node will become one MPS and the same process occurs with the cv and the population.

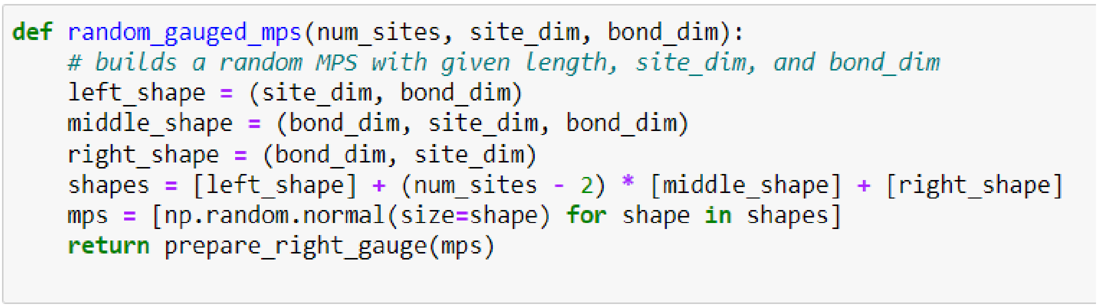


```
PROTOTYPE FOR RANDOMIZED SVD
Given an $m \times n$ matrix $\boldsymbol{A}$, a target number $k$ of singular vectors, and an
exponent $q$ (say, $q = 1$ or $q = 2$), this procedure computes an approximate
rank-$2k$ factorization $\boldsymbol{U\Sigma V}^*$, where $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthonormal, and $\boldsymbol{\Sigma}$ is
nonnegative and diagonal.
Stage A:
1   Generate an $n \times 2k$ Gaussian test matrix $\boldsymbol{\Omega}$.
2   Form $\boldsymbol{Y} = (\boldsymbol{AA}^*)^q \boldsymbol{A\Omega}$ by multiplying alternately with $\boldsymbol{A}$ and $\boldsymbol{A}^*$.
3   Construct a matrix $\boldsymbol{Q}$ whose columns form an orthonormal basis for
    the range of $\boldsymbol{Y}$.
Stage B:
4   Form $\boldsymbol{B} = \boldsymbol{Q}^*\boldsymbol{A}$.
5   Compute an SVD of the small matrix: $\boldsymbol{B} = \widetilde{\boldsymbol{U}}\boldsymbol{\Sigma V}^*$.
6   Set $\boldsymbol{U} = \boldsymbol{Q}\widetilde{\boldsymbol{U}}$.
Note: The computation of $\boldsymbol{Y}$ in step 2 is vulnerable to round-off errors.
When high accuracy is required, we must incorporate an orthonormalization
step between each application of $\boldsymbol{A}$ and $\boldsymbol{A}^*$; see Algorithm 4.4.
```

Figure 5.4: Pseudo code for a randomized SVD. Figure taken from Ref. [50].

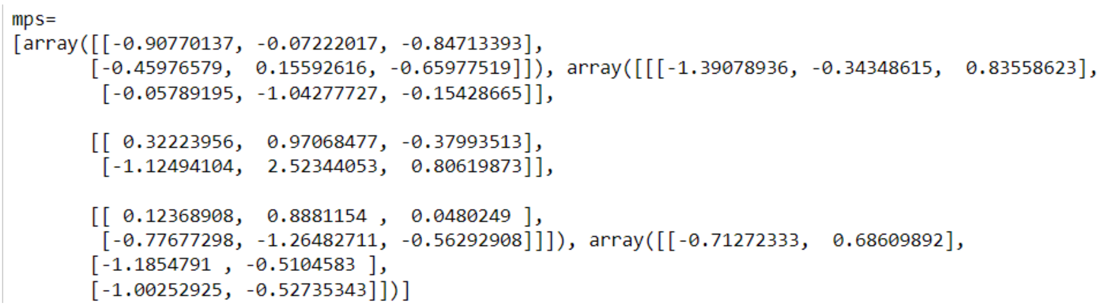
Now that our input data is properly processed, we can enter the algorithm itself. We start by building a MPS with a given length, number of sites and bond dimension. Recalling 4.9, our input would be represented as the coefficients $\sigma_1, \ldots, \sigma_{600}$. For very large-scale matrices, instead of applying the typical truncated SVD, as it was mention in chapter 4.2.1, one applies the randomized SVD. The randomized SVD will reduce the original matrix into a smaller one by multiplying it with a sampling matrix [51]. Since we are missing the sampling matrix, generating one was the following goal. In the algorithm, to apply the randomized SVD, we use a variation of the prototype showed in 5.4.

Similarly to stage A of the prototype, we start by generating the test matrix. The test matrix is going to have the same length as our MPS, 600 sites. The way to do it is by creating a tensor, analogous to the one showed in the first step of figure 4.4. This tensor is going to have a length of 600 sites and each site will be represented by a (2,3), (3,2,3) or (3.2) tensor depending on whether it is the left site, the middle sites or the right site, accordingly. The 3 corresponds to the bond dimension chosen and the 2 refers to the physical dimension of each site, which translates the total number of possible states each site can be in, i.e. 0 and 1 or, in terms of connectivity, means that a pair of nodes is connected or not.



```python
def random_gauged_mps(num_sites, site_dim, bond_dim):
    # builds a random MPS with given length, site_dim, and bond_dim
    left_shape = (site_dim, bond_dim)
    middle_shape = (bond_dim, site_dim, bond_dim)
    right_shape = (bond_dim, site_dim)
    shapes = [left_shape] + (num_sites - 2) * [middle_shape] + [right_shape]
    mps = [np.random.normal(size=shape) for shape in shapes]
    return prepare_right_gauge(mps)
```

(a)



```
mps=
[array([[-0.90770137, -0.07222017, -0.84713393],
       [-0.45976579,  0.15592616, -0.65977519]]), array([[[-1.39078936, -0.34348615,  0.83558623],
        [-0.05789195, -1.04277727, -0.15428665]],

       [[ 0.32223956,  0.97068477, -0.37993513],
        [-1.12494104,  2.52344053,  0.80619873]],

       [[ 0.12368908,  0.8881154 ,  0.0480249 ],
        [-0.77677298, -1.26482711, -0.56292908]]]), array([[-0.71272333,  0.68609892],
       [-1.1854791 , -0.5104583 ],
       [-1.00252925, -0.52735343]])]
```

(b)

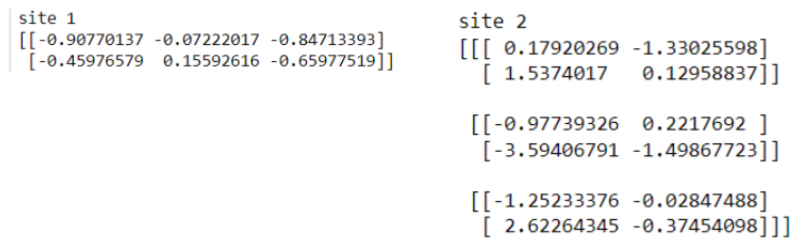
Figure 5.5: (a) Code used to generate and fill the MPS with random values drawn from a normal Gaussian distribution. (b) Example of the sampling MPS with 3 nodes.

After creating this sampling tensor, one must fill it with random values using the *random.normal()* function of the numpy library. This function will draw random samples from a normal Gaussian distribution. For demonstration purposes and in order to first understand the algorithm better a test run was made using only 3 nodes. The sampling MPS would be similar to the one shown in the figure 5.5, with the respective code used to generate it.

The next step, following the prototype shown in 5.4 would be to construct the matrix Q and the way we do it is by taking the sampling matrix and apply the SVD to it. As explained in 4.2.1 and recalling the chapter 4.2.3, we built a MPS from the sampling tensor by performing the SVD starting from the right which will result in a MPS in the right-canonical form. Similar to the example given, the singular values of the sampling tensor will be carried out through the process onto the ψ tensor as they get absorbed, in this case to the left.

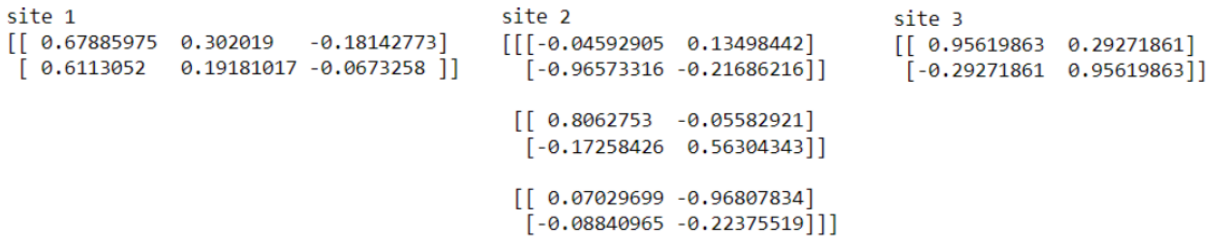
Looking back on the figure 4.4, after applying the SVD to the sampling matrix for the first time, one would obtain $U\Sigma V^T$. As we carry on applying the SVD multiple times, and since we are leaving the MPS in a right-canonical form, the V^T is going to be the tensor that represents the site in question of our MPS and the $U\Sigma$ is going to be absorbed by the following site of ψ, which is the remaining sampling tensor we did not yet decompose.

After applying the SVD to all sites of the sampling tensor, we end up with the last stage of figure 4.4, with the exception that the first site still needs to be normalized, since all the singular values and left singular vectors were absorbed. While applying the SVD, one must reshape the local tensor in order to decompose the site, as mentioned in 4.2.3, and reshape it afterwards, in order to restore the original shape. The figure 5.6 (a) shows the resulting sampling MPS after applying the SVD to site number 3. By comparing it with 5.5 (b), one can conclude that site number 1 remains unaltered, while site 3 corresponds to V^T and site 2 is simply the matrix product between $U\Sigma$ and the original tensor representing the site. On the other hand, the figure 5.6 (b) shows the final form of the sampling MPS after applying the SVD to all the sites and normalizing the first one.



```
site 1
[[-0.90770137 -0.07222017 -0.84713393]
 [-0.45976579  0.15592616 -0.65977519]]

site 2
[[[ 0.17920269 -1.33025598]
  [ 1.5374017   0.12958837]]

 [[-0.97739326  0.2217692 ]
  [-3.59406791 -1.49867723]]

 [[-1.25233376 -0.02847488]
  [ 2.62264345 -0.37454098]]]

site 3
[[ 0.95619863  0.29271861]
 [-0.29271861  0.95619863]]
```

(a)



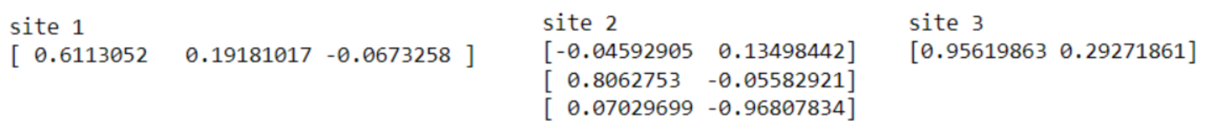
```
site 1
[[ 0.67885975  0.302019    -0.18142773]
 [ 0.6113052   0.19181017 -0.0673258 ]]

site 2
[[[-0.04592905  0.13498442]
  [-0.96573316 -0.21686216]]

 [[ 0.8062753  -0.05582921]
  [-0.17258426  0.56304343]]

 [[ 0.07029699 -0.96807834]
  [-0.08840965 -0.22375519]]]

site 3
[[ 0.95619863  0.29271861]
 [-0.29271861  0.95619863]]
```

(b)

Figure 5.6: (a) Result of performing the SVD onto site 3 and absorbing $U\Sigma$ into site 2; The first site remains unaltered. (b) Result of performing the SVD in all 3 sites and normalizing the first one.

Recalling the prototype shown in the figure 5.4, we finished stage A and enter stage B, which consists of performing the DMRG algorithm. Before step 4, we first need to filter our sampling MPS by using the input from the complex network. The addition of an extra step that does the matrix product between the sampling MPS and the train_batch achieves that goal. The tensor A is going to represent the train_batch. Taking the 3 sites MPS as an example once again, the matrix A would be a tensor with 3 values of either 1 or 0. Taking A to be $A = [100]$ and Q to be the sampling matrix in the MPS form shown in the figure 5.6 (b), by performing the matrix product between both tensors, one is filtering the sampling tensor using the connectivity of the nodes. In this particular example, one would take the second line of the tensor representing site

1 and the first lines of the tensors representing sites 2 and 3. This process is illustrated in figure 5.7 and it can be seen as choosing one of the two possible physical dimensions. In the example of the spin of electrons, we would be choosing one of the dimensions of each individual site of the MPS, according to the spin up, 1, or the spin down, 0.



```
site 1                                   site 2                        site 3
[ 0.6113052   0.19181017 -0.0673258 ]    [-0.04592905  0.13498442]     [0.95619863 0.29271861]
                                         [ 0.8062753  -0.05582921]
                                         [ 0.07029699 -0.96807834]
```

Figure 5.7: Resulting sampling MPS after being filtered by our input.

Now that we filtered the sampling MPS, we are ready to perform the sweeps of the DMRG algorithm. Recollecting the explanation given about leaving a MPS in the DMRG form in the end of chapter 4.2.3, we choose a site to be the “center” of the MPS. Every singular value to the left of the center gets absorbed into the tensor representing the site on the left of it. The same happens to the singular values on right of the center. The DMRG is going to be applied a certain number of times, according to the configuration we chose beforehand. Each time we apply the DMRG algorithm, we are performing several sweeps, as many as double the number of sites our MPS has.

Since we start by choosing the center to be the first site, we do steps 4 to 6 of the prototype and then change the center to be the second site and go all the way until we reach the last site. After that, we circle back repeating the same process until the center is yet again the first site. When we reach the first site again, we have completed a single run of the DMRG and the process is repeat for as many times as we defined in the configurations, in this case 100 times. This process is illustrated in figure 5.8.

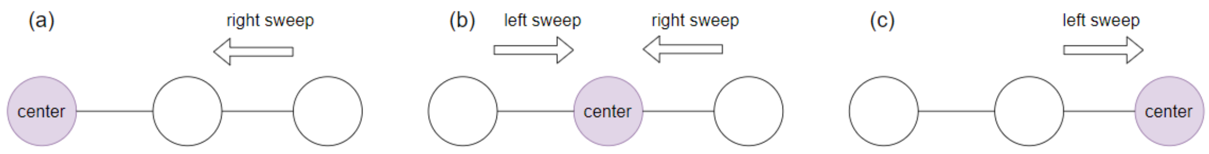
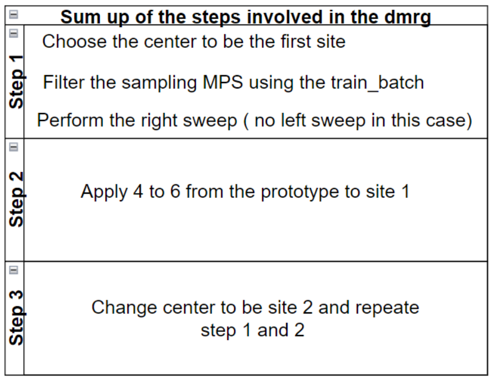


Figure 5.8: Illustration of the DMRG being applied one time. After we reach (c), we need to repeat (b) and (a) and that completes one run of the algorithm.

A single sweep consists of a combination of a right sweep and a left sweep. In the DMRG form, one chooses a site from the MPS, for example X, and performs a sweep from the first site until the site before X and a sweep from the last site until the one right after X. In the example with a MPS with only 3 sites, when performing a sweep on site number 1, there is no left sweep, since there are no sites to the left of site number 1. The right sweep goes from site number 3 until we reach site number 1, as it is shown in figure 5.8.

After filtering the sampling MPS based on the input, we start doing the sweeps. Besides the sweeps, computing the DMRG also includes steps 4 to 6 of the prototype represented in 5.4, as mentioned before. A table summing up the steps involved in the application of the DMRG is represented in figure 5.9. The process shown represents a single run of the algorithm and it is to be repeated 100 times. It is important to note that the sampling MPS does not suffer any alterations as it is being filtered. Instead, we filter it while performing the sweep and store the resulting tensor, leaving the original sampling MPS unaltered. In other words, it would be as if we created a copy of the original sampling MPS, filtered it, as it was explained in figure 5.7 and performed the sweeps.



Sum up of the steps involved in the dmrg

Step 1	Choose the center to be the first site Filter the sampling MPS using the train_batch Perform the right sweep (no left sweep in this case)
Step 2	Apply 4 to 6 from the prototype to site 1
Step 3	Change center to be site 2 and repeate step 1 and 2

Figure 5.9: Summary of the DMRG in practise. A single run of the DMRG consists of performing all 3 steps. Step 3 is illustrated in figure 5.10.

As we begin the DMRG algorithm, and after completing the right sweep, we do step 4, where we calculate the tensor product between $Q[1]^T$ and $A[1]$. Recalling when we filtered the MPS, we set A to be $A = [100]$. Hence, $A[1]$ is going to be equal to $|1>$ or $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$, in the 3 sites MPS example. $Q[1]$ is going to be the tensor representing the right sweep. We obtain $B[1]$ by doing $A[1] \otimes Q[1]^T$ and the result can be found in figure 5.10(b). Figure 5.10(a), shows the tensor obtained when performing the right sweep from site number 3 until site number 1. It is important to note that both tensors, $Q[3]$ and $Q[2]$ do not have the right dimensions in order to be multiple. Therefore an extra dimension is added to $Q[3]$ and later removed from the final result. In figure 5.10(a) the result is presented with that extra dimension while in figure 5.10(b), the same result is used to perform step 4 without said extra dimension.

$$\begin{bmatrix} -0.04592905 & 0.13498442 \\ 0.8062753 & -0.05582921 \\ 0.07029699 & -0.96807834 \end{bmatrix} \times \begin{bmatrix} 0.95619863 & 0 \\ -0.29271861 & 0 \end{bmatrix} = \begin{bmatrix} -0.08342974 & 0 \\ 0.78730159 & 0 \\ 0.35059243 & 0 \end{bmatrix}$$

(a)

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} -0.08342974 & 0.78730159 & 0.35059243 \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} 0 & 0 & 0 \\ -0.08342974 & 0.78730159 & 0.35059243 \end{bmatrix}$$

(b)

Figure 5.10: (a) Visualisation of the right sweep. Since we are working with the 3 sites MPS example, the right sweep will only consist of doing the matrix multiplication between sites 2 and 3. (b) Result of applying step 4 of the prototype shown in figure 5.4 when the center is site number 1.

In the example shown with 3 sites, there is only 1 MPS, but, as it was mentioned in the processing data, we start with an input of 600 lists, each one representing the connectivity of each node. Only the first 200 are used in the training of the MPS. Nevertheless, it means we are working with 200 MPSs. Each individual MPS uses the connectivity of a single node as input, therefore, all the steps performed until this point happen simultaneously in all 200 MPSs. The same sampling MPS is used for all of them, but rather it is the input that varies, i.e. it is the filter that changes.

When the algorithm finishes step 4, shown in figure 5.10, it presents the local tensor presented in (b). Since the algorithm is operating on the 200 MPS at the same time, this local tensor is going to be calculated 200 times. In step 5, the sampling MPS is going to be altered, meaning, when we change the center to be site 2, it is going to be a different tensor representing the site. The local tensor that was calculated is going to be used to alter the sampling MPS. Consequently, all 200 local tensors are going to be summed up and it will need to be normalized.

$$\|local\_tensor\| = \sqrt{(-0.08342974)^2 + (0.78730159)^2 + (0.35059243)^2} = 0.8658633643785765 \tag{5.1}$$

$$\frac{\begin{bmatrix} 0 & 0 & 0 \\ -0.08342974 & 0.78730159 & 0.35059243 \end{bmatrix}}{0.8658633643785765} \tag{5.2}$$

Returning to the example with 3 sites. No sum is going to take place because there only exists one MPS. Nonetheless, the local tensor calculated in figure 5.10 (b) is still going to be normalized. The calculation of the norm of the local tensor shown in figure 5.10(b) is represented in equation 5.1 and in 5.2 it is shown the normalization of the local tensor, which is going to be used in step 5 and forward.

Step number 5 involves applying the SVD to this tensor, as it can be seen in stage B the prototype in figure 5.4. Using the normalized local tensor calculated, we use *np.linalg.svd()* to obtain the SVD. Figure 5.11 shows the result of the SVD.

$$\overset{\text{Local tensor}}{\begin{bmatrix} 0 & 0 & 0 \\ -0.0963544 & 0.9092677 & 0.40490503 \end{bmatrix}} = \overset{\text{U}}{\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}} \overset{\Sigma}{\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}} \overset{V^T}{\begin{bmatrix} -0.0963544 & 0.9092677 & 0.40490503 \\ -0.99534709 & 0.0880215 & -0.03919676 \end{bmatrix}}$$

Figure 5.11: Calculation of the SVD applied to the normalized local tensor represented in figure 5.2(b).

Once we obtain the matrices U, Σ and V^T, we are going to replace the tensor that was representing site 1 of the sampling MPS with the matrix U. Σ and V^T are going to be absorbed by the tensor representing site number 2 of the sampling MPS. In figure 5.12, the resulting tensor of site 2 can be found, complemented with the respective calculation, as well as the full sampling MPS. This is the first time that the sampling MPS is indeed altered.

$$\begin{array}{ccc} \Sigma & & V^T \\ \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} & \times & \begin{bmatrix} -0.0963544 & 0.9092677 & 0.40490503 \\ -0.99534709 & -0.0880215 & -0.03919676 \end{bmatrix} \end{array} = \begin{bmatrix} -0.0963544 & 0.9092677 & 0.40490503 \\ 0 & 0 & 0 \end{bmatrix}$$

(a)

$$\begin{bmatrix} -0.0963544 & 0.9092677 & 0.40490503 \\ 0 & 0 & 0 \end{bmatrix} \times$$

```
[[[-0.04592905  0.13498442]
  [-0.96573316 -0.21686216]]

 [[ 0.8062753  -0.05582921]
  [-0.17258426  0.56304343]]

 [[ 0.07029699 -0.96807834]
  [-0.08840965 -0.22375519]]]
```

$$=$$

```
[[[ 0.76600916 -0.45574983]
  [-0.09967017  0.44225322]]

 [[ 0.          0.        ]
  [ 0.          0.        ]]]
```

(b)

Figure 5.12: (a) Σ and V^T being absorbed into a single tensor that is going to be absorbed by the tensor representing site 2 of the sampling MPS. (b) Absorption of the resulting tensor into the tensor representing the original site 2 of the sampling MPS.

The final tensor represented in figure 5.12(b) is calculated as shown in the list below.

- $0.76600916 = -0.0963544 \times -0.04592905 + 0.9092677 \times 0.8062753 + 0.40490503 \times 0.07029699$;
- $-0.45574983 = -0.0963544 \times 0.13498442 + 0.9092677 \times -0.05582921 + 0.40490503 \times -0.96807834$;
- $-0.09967017 = -0.0963544 \times -0.96573316 + 0.9092677 \times -0.17258426 + 0.40490503 \times -0.08840965$;
- $0.44225322 = -0.0963544 \times -0.21686216 + 0.9092677 \times 0.56304343 + 0.40490503 \times -0.22375519$.

As mentioned before, this is the first time the sampling MPS is being altered since it was decomposed using the SVD. The sampling MPS after the SVD is shown in figure 5.7 and it is the tensor representing site 2 that is going to absorb Σ and V^T. Figure 5.13, represents the sampling MPS after steps 1 and 2 of the summary shown in 5.9. As it can be seen, site number 1 is represented by U shown in figure 5.11. The tensor representing site 2 is the consequence of absorbing Σ and V^T, as explained, and site 3 remains unaltered.

Site 1

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

Site 2

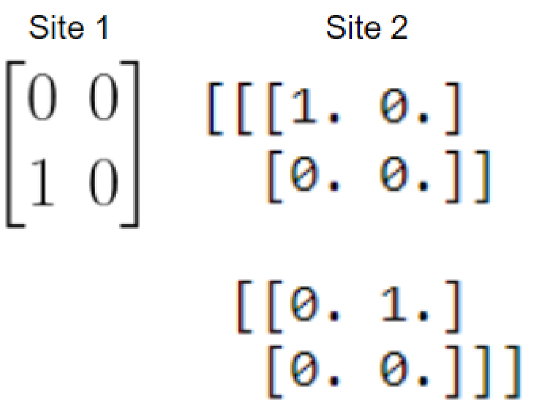
```
[[[ 0.76600916 -0.45574983]
  [-0.09967017  0.44225322]]

 [[ 0.          0.        ]
  [ 0.          0.        ]]]
```

Site 3

$$\begin{bmatrix} 0.95619863 & 0.29271861 \\ -0.29271861 & 0.95619863 \end{bmatrix}$$

Figure 5.13: Result of the sampling MPS after performing steps 1 and 2 of the summary shown in figure 5.9. Site 1 is represented by U, calculated in figure 5.11. Site 2 is represented by the tensor calculated in figure 5.12(b). Site 3 is represented by the original tensor shown in figure 5.6(b)

We now repeat steps 1, 2 and 3 of the summary represented in figure 5.9 until the center is again site 1, which 1 run of the DMRG. In figure 5.14, one can find the resulting sampling MPS after one run of the DMRG. Once the algorithm was understood, we run it with our original input, using the complex network generated instead of the example with the 3 sites MPS. Since that MPS has 600 sites, it is not viable to show it here.



$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

```
[[[1. 0.]
  [0. 0.]]

 [[0. 1.]
  [0. 0.]]]
```

Figure 5.14: Resulting sampling MPS after 1 run of the DMRG algorithm.

Before starting the next run of the DMRG, we need to evaluate the MPS first. Using the train_batch and the cv_batch, that were mentioned in the processing data, the algorithm calculates the fidelity of the MPS and compares it with the previous one. In the first run of the DMRG there is not a previous fidelity to compare it with. Nevertheless, from the second run forward, the new fidelity and the previous one are compared, where the former fidelity gets subtracted to the new one. If the difference is bigger than zero, i.e. if the new fidelity is better than the last one, the DMRG keeps running. However, if the change in the parameter is not greater than zero, it means either the fidelity is exactly the same and we neither improved nor worsen the MPS, or it got slightly worse. If indeed the fidelity got worse, in the next run, when it gets compared again, it will likely be improved.

For this reason, the algorithm has a parameter named patience, and every time the difference is equal or smaller than zero, this parameter increases. If the patience reaches a certain number, the algorithm stops, even if it has not yet completed the 100 runs. Nonetheless, if the difference in fidelity keeps being equal or less than zero and in the next run it is verified that it improves, then, the patience gets reset. In order for it to reach said number and stop the algorithm, the difference in the fidelity needs to be constantly equal or less than zero.

Another way to evaluate the MPS is by starting at the final MPS of each run and try to obtain the connectivity of the nodes in the complex network. We start with the final MPS after the first run of the DMRG shown in figure 5.14 and we will extract the probability of each singular value out of each site. While extracting the probability out of the second site, we cannot do it independently of the result of the first site because there are correlations among the sites.

We start by going to the first site and calculate the amplitude of probability of the tensor by multiplying the tensor, lets assume S, with its conjugate transpose, $S^\dagger$. The result of $S \times S^\dagger$, where S is the tensor representing the first site of the MPS, can be found in figure 5.15.

$$\begin{array}{l} |0> \Longrightarrow \\ |1> \Longrightarrow \end{array} \overset{S}{\begin{bmatrix}0&0\\1&0\end{bmatrix}} \times \overset{S^\dagger}{\begin{bmatrix}0&1\\0&0\end{bmatrix}} = \overset{|S|^2}{\begin{bmatrix}0&0\\0&1\end{bmatrix}}$$

Figure 5.15: Result of $S \times S^\dagger$, where S is the tensor representing site 1 of the MPS after the first run of the DMRG shown in figure 5.14.

Now that we obtained $|S|^2$, we take the values in the diagonal, in this case 0 and 1. These values represent the coefficients $c0_1$ and $c1_1$, respectively. Assuming we were working with $|\varphi >= \alpha|0> + \beta|1>$, then, in this case, $\alpha = 0$ and $\beta = 1$. Consequently, by measuring the state, we would obtain $|1>$ 100% of the times. In this case it is simple because $\alpha = 0$. Nonetheless, if α was different than zero and always respecting $|\alpha|^2 + |\beta|^2 = 1$, we would obtain $|0>$ with a probability $|\alpha|^2$ and $|1>$ with a probability of $|\beta|^2$. For that reason, we perform the measurement 10 times simultaneously, as if we have 10 MPSs exactly the same.

We obtained 1 as the measurement of the first site and we store that value in a list so we can compare it to the input later. As we move to the second site of the MPS, we already altered the first site since we performed a measurement and the system collapsed to $|1>$ or, as it is shown in figure 5.15, [1 0]. The tensor representing the first site no longer is $\begin{bmatrix}0&0\\1&0\end{bmatrix}$, but rather it is [1 0], and before measuring the second site, we need to obtain the tensor representing the second site, knowing the result of the measurement of the first site. What we are aiming to do now, is obtain the measurement of site 2 under the condition that site 1 is $|1>$.

Following the same method used in figure 5.12(b) and the list used to explain it, we are going to contract [1 0] with the tensor representing site 2, shown in figure 5.14. The result of the contraction between both tensors can be found in figure 5.16.

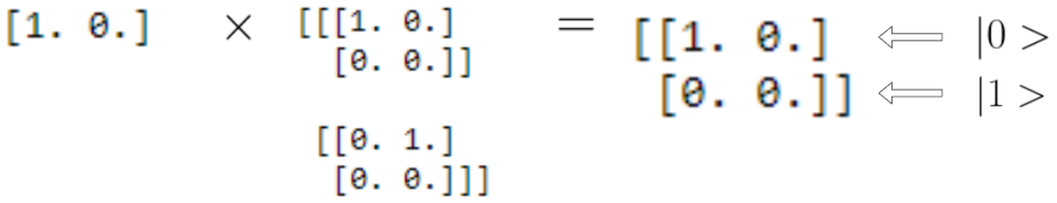

```
[1. 0.]    ×   [[[1. 0.]          =   [[1. 0.]   ⟸  |0 >
                 [0. 0.]]               [0. 0.]]  ⟸  |1 >

                [[0. 1.]
                 [0. 0.]]]
```

Figure 5.16: Result of the contraction between the tensor representing site 1, [1 0], and the tensor representing site 2, shown in figure 5.14.

After obtaining the tensor shown in figure 5.16 that represents site 2 knowing the measurement of site 1, we repeat the process shown in figure 5.15 to said tensor. By doing $\begin{bmatrix}1&0\\0&0\end{bmatrix} \times \begin{bmatrix}1&0\\0&0\end{bmatrix} = \begin{bmatrix}1&0\\0&0\end{bmatrix}$, we obtain $|S_2|^2$. Consequently, we obtain $\alpha = 1$, the system collapsed to $|0>$ and the tensor in that site becomes [1 0].We store 0 as the result of the measurement in site 2 knowing that the measurement in the first site was 1. Following the same logic, we now must calculate the tensor representing site 3 under the condition of the resulting measures of both sites 1 and 2. As we move to the third site, we have already changed both the tensor representing the first and second site. $[1\ \ 0]$ represents the first site of the MPS and $[1\ \ 0]$ represents the second site of the MPS under the condition of the first measurement.

We are now going to obtain the tensor that represents the measurement in site 2 under the condition of the measurement in site 1. To obtain this tensor we contract the tensor representing the collapsed site, $[1\ \ 0]$, with the tensor representing the collapsed site 2, $[1\ \ 0]$. The resulting matrix, $\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$, represents both measurements, i.e., this tensor is going to be used to calculate the tensor in site 3 knowing that site 1 collapsed to $|1>$ and site 2 collapsed to $|0>$. In figure 5.17, we calculate said tensor.

$$\begin{bmatrix}1 & 0\end{bmatrix} \times \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} = \begin{bmatrix}1 & 0\end{bmatrix}$$

Figure 5.17: Result of contraction between $[1\ \ 0]$, that represents site collapsing to $|0>$, knowing that site 1 collapsed to $|1>$, and the tensor that represents site 3, shown in figure 5.14.

Since it is the last site of the MPS, before calculating $|S_3|^2$, we first need to do the outer product of the site with itself. By calculating $\begin{bmatrix}1 & 0\end{bmatrix} \otimes_{outer} \begin{bmatrix}1 & 0\end{bmatrix}$, we obtain $\begin{bmatrix}1 & 0\\ 0 & 0\end{bmatrix}$. The resulting $|S_3|^2$ of this matrix returns $\alpha = 1$ and the system collapses to $|0>$.

Recalling that site 1 collapsed to $|1>$ and site 2 and site 3 collapsed to $|0>$, we obtain the list shown in 5.3, representing the connectivity of the nodes used as input.

$$[1\ \ 0\ \ 0] \tag{5.3}$$

This result corresponds to the input used to train the 3 sites MPS. This happened because it is a very simple example with only 3 sites and, as mentioned before, when performing the sweeps, since there was only one MPS, there was no need to calculate the mean of the resulting tensor from the sweeps. In addition, the MPS in the example was trained using only the connectivity of the first node, which explains the recovered result being an exact match. Nevertheless, with the 600 sites MPS, the MPS will be trained with the connectivity of the first 200 sites, which will imply that the resulting measurement on the MPS will possibly translate the Barabási network. Furthermore the values of α and β will not be exactly zero or one, but will rather be a value in between. For that reason, when we perform the measurements on the MPS, we do it multiple times simultaneously.

Chapter 6

Results

The first goal was to characterize the Barabási network and since we use the connectivity of the nodes as an input, we opted to characterize it by using the distribution of the connectivity. knowing that the connectivity of a Barabási network follows a power law $P(k) \approx k^{-\gamma}$, where k is the connectivity and theoretically γ is a value between 2 and 3, we can calculate γ for the Barabási network.

In order to obtain γ from the original network, we need to know how many connections each node has. Once we obtain that information, we count how many nodes have the same number of connections vs the logarithm of the number of nodes that have said number of connections. By integrating P(k) from a fix value x to infinity and after applying the logarithm, we obtain that the slope of the line will correspond to the value of $-(\gamma - 1)$, i.e., $\gamma = -slope + 1$. Figure 6.1, represents the graphic of the LN of the number of connections as a function of the LN of the number of nodes.

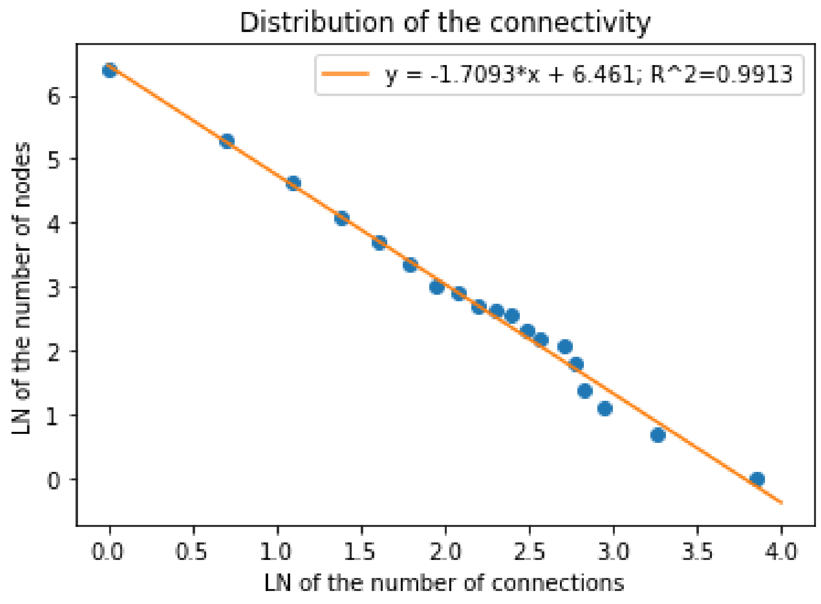


Figure 6.1: Graphic of the LN of the number of connections vs the LN of the number of nodes.

By analysing the graphic 6.1 and using the excel tools to calculate the linear regression, we can conclude that the slope of the line is , which corresponds to the value of γ. Consequently, the Barabási network used has a connectivity distribution that follows the power law $P(k) \approx k^{-2.7}$, which is between the theoretical values.

Once the Barabási network was characterized, we started analysing the output of the algorithm as explained in the final paragraph of chapter 5. As mentioned, the MPS was trained using the connectivity of the first 200 nodes and when performing a measurement in all 600 sites, the MPS will collapse to one possible configuration. To conclude if we can characterize the network based on 1 single node, i.e., if my measuring one site we can obtain the configuration of the remaining sites, we will measured the MPS multiple times, and compare the output.

Recalling the explanation given in the end of chapter 5, the output of measuring the MPS is similar to the one shown in 5.3, but instead of having 3 sites, it has 600. The first result we obtained can be found in 6.1.

$$\begin{gathered}
[101010010000100110000000100001100000000000000000000000000001000100000000100000 \\
000100000100100000000100100000000000100000100000000000001000010100010000000000 \\
001000000001000110100000000100000000000000000000000000000100000000000000000000 \\
001000000000000000000000000000000000000000000000100000000000000000000000001 \\
000000000000100000000000000100000000010000000000000000000000000100000000000010 \\
000000000000000000100000000000000000100000000000000000000000001000000000000000000 \\
000001000000000000000000000000000001010000000000000000000000000000000000000000 \\
00000000000000000000000000010000000000100000000010000000000000000000100000100000]
\end{gathered} \tag{6.1}$$

After measuring the MPS another time, we noticed, as expected that the result was different. Nonetheless, after hundreds of measurements, we compared all the measurements and, for this particular Barabási network, we found only 2 possible outputs. The one shown in 6.1 and the one shown in 6.2, where it can be seen that the first value shifts from a 1 to a 0. This result implies that, for this particular Barabási network, we are only required to measure the first node in order to obtain the result of the complete MPS, if we know the full configuration *a priori*.

In order to understand the meaning of the outputs, lets suppose our MPS represents text, so each site would be representing a word from the text used to train the MPS. The measurements on the sites indicate Whether the words that are being represented by site 1 and 2 are related or not, i.e., the 1 measured in site 1 indicates that both words are correlated while the 0 measured in site 2 indicates that the words being represented by site 2 and 3 do not share a correlation. Analogously, the 1 measured in site 1 indicates that the first and second node of our Barabási network are connected while the 0 measured in site 2 indicates that nodes 2 and 3 are not connected.

If our network represented text, then it would be probable that the word being represented by the first site is correlated with more than 1 word, which would lead to different outputs when measuring the MPS.

$$\begin{gathered}[0111010111110100100001000001000000000001000010100100000000100000010000000010\\ 0000000100000000100000000010000000000110100000000000000001000000000010000000000\\ 0000000000000000000000000000000000100000100000000000001010010000000000000000\\ 0000000000001001000000010010000000000000010000000000000100000000000000000000\\ 0000000100000100110000000000000000000001000000010000000000000010001000000000000\\ 0000100000000000000000000001000000000000000000000000000001000000000000000000000\\ 0000000000000000000000000000100000000000010000000000000000000000000000000000000\\ 000000000000000000000000000000110000000000000000000000000000000000000000000]\end{gathered}\tag{6.2}$$

After measuring the MPS 600 times, we analysed the frequency at which each possible outcome appears. For the output shown in 6.1, we obtain a frequency of $\frac{295}{600}$ that corresponds to $\approx$ 49.17%. Consequently, the frequency of the output shown in 6.2 is given by $100 - 49.17 = 50.83\%$. Moreover, we can conclude that, by measuring the first site, we can automatically know the result for the rest of the MPS. Suppose we have a system of 600 entangled electrons, with either spin up or down, then, measuring the spin of the first electron and depending if its up, 1, or down, 0, we know the remaining spins of the other electrons. In addition, we also know with what frequency we are going to measure a 0 or a 1 as the result of the first spin.

As explained in the chapter 5, we first set the bond dimension to be 3 and the previous results uses that value. Changing the bond dimension, we verify that, despite the input being the same, the output itself is different. Besides that, we also noticed that, the output went from being 2 possible sequences to 3. This happens since the sampling MPS itself changed when the bond dimension changed from 3 to 4. Recalling the data processing of chapter 5, the dimension of the tensors that compose the sampling MPS directly depend on the bond dimension, which will make the MPS different in dimensions and with more random values, since the tensors are going to have dimensions (2,4), (4,2,4) and (4,2), for the first site, the middle sites and the last site, respectively. Table 6.1, shows the bond dimension used, the number of different outputs said dimension produces, and the frequency of each output. Note that all these results are produced using the same Barabási network.

Bond dimension	2		3		4			5			
Number of different outputs	2		2		3			4			
Frequency of the outputs	48.70%	51.30%	49.17%	50.83%	29.17%	32.33%	38.50%	24.67%	23.00%	27.33%	25.00%

Table 6.1: Table displaying for each bond dimension used, the respective number of different outputs and the frequency of each output.

Comparing the different outputs from different bond dimensions, we found that the output shown in 6.1 appears across all bond dimensions, with a probability of 51.30%, 49.17%, 38.50% and 25.00% for bond dimensions 2, 3, 4 and 5, respectively. The output shown in 6.2 was obtained using a bond dimension equal to 3 and appears just one other time with a frequency of 24.67% for a bond dimension equal to 5. All remaining outputs are unique outputs.

Up to this point, we only used one of the methods described in the evaluation of the MPS.

As mentioned before, the algorithm uses the train_batch and the cv_batch to determine the fidelity of the MPS and depending on the value obtained, it runs for a longer period of time or it converges more rapidly. In order to determine the value of the bond dimension that better describes the Barabási network, i.e., the value of the dimension of the space required that best describes the Barabási network, we compared the number of runs of the DMRG necessary for the MPS to converge. Table 6.2 shows the number of runs of the DMRG necessary for the MPS to converge.

Bond dimension	2	3	4	5
Number of runs of the DMRG	21	21	100	100

Table 6.2: Table displaying the necessary number of runs of the DMRG in order for the MPS to converge.

By analysing table 6.2, we can conclude that the minimal dimension of the space that best describes the Barabási network is between 2 and 3, which are also the values of the bond dimension that produces only 2 outputs. Furthermore, we believe that, the fact that one output appears for all values of the bond dimension, means that, it represents the system's configuration that is most likely to be encountered. In addition, we can also conclude that, for bond dimensions 4 and 5, the MPS did not converge, but rather the algorithm stopped at the designated 100 runs.

From table 6.1, we can conclude that, for both bond dimensions 2 and 3, by measuring only one of the sites we are capable of obtaining the value of the remaining sites. For bond dimensions equal to 4 and 5, in order to obtain the configuration of the collapsed system, we need to measure 2 sites. To verify if this occurrence was a particularity of this Barabási network configuration we generated other Barabási networks and checked if the outputs of the measurements were enough to describe the system based on just 1 or 2 sites. By repeating this process, we can deduce that, although the number of outputs varies for different Barabási networks, using the same bond dimension, the system always collapsed to a limited number of outputs, lower than 5. For example, using a different Barabási network, the system collapsed to 4 different outputs instead of 3, when we set the bond dimension to 4.

The last characteristic we wanted to study is whether we can describe the system based only on the result of 1 site, for other types of networks such as the Erdős–Rényi network. We obtain the same type of result, where the system collapsed to a few outputs with a certain frequency, for different bond dimensions. Similarly to the Barabási network, for bond dimensions 2 and 3, the system collapsed to 2 outputs. For bond dimensions 4 and 5 the system collapsed to 3 and 4 outputs, respectively. Nevertheless, in contract to what we obtain for the Barabási network, there was not an output that appears across all bond dimensions. Furthermore, for a bond dimension equal to 3, when using the Barabási network as input, the MPS converge after 21 runs of the algorithm. However, when using the Erdős-Rényi network as input, the MPS took longer to converge for said bond dimension.

Chapter 7

Discussion and Conclusion

The goal of this project was to study the minimization of information loss when re-normalizing complex networks. Our first approach was to use the MERA algorithm. However, we opted to use the Matrix Product State algorithm since it did not use the disentangler operators, which fell out of the scope of this project. Using the MPS algorithm allowed us to start with a space with a dimension equal to v^N, where v is the physical dimension and N is the number of particles of the system in question, and reduces the dimension of the space needed in order to describe the system to 2^b.

Similarly to general coordinates from analytical mechanics, we use the entanglement between the particles and the SVD in order to find the set of coordinates that best describes our system or, in our case, to reduce the dimension of the space. By using the Singular Value Decomposition, which is one main part of the DMRG algorithm, we are truncating the singular value matrix and the right and left singular vectors matrices to an optimal value where the information lost is negligible. This optimal value is called the bond dimension.

We started with a general quantum state, a complex network to be precise, and transformed it into a MPS by applying the SVD in a repetitive way. Once we obtained the MPS, we performed several sweeps and by measuring each individual site of the MPS, we arrived at an output. Using the algorithm explained in chapter 5, we were able to understand the full potential of the algorithm itself and we realized that we could apply it to a Barabási network and study the optimal value that still allowed us to describe the system. By studying the best value for the bond dimension necessary for the MPS to converge, we came to the conclusion that, for a Barabási network, a bond dimension between 2 and 3 is the best value for the dimension of the space required to describe the complex network, where the truncated information is negligible.

For bond dimensions higher than 3, the MPS did not converge. When we measure the MPS and it collapses to one of the outputs, we obtain one of the linearly independent vectors that form the base of the space. Since we obtained two different outputs, we can conclude that our space is composed of 2 linear independent vectors and those vectors are represented in 6.1 and 6.2. We started with an input of 360000 zeros and ones and ended up with an output of 600 zeros and ones, resulting in a reduction of the information needed. We could not compare the bond dimension values bigger than 3 since the algorithm did not converge for those values of bond dimension. Nevertheless, for bond dimensions equal to 2 and 3, we concluded that, for both values, the MPS converges with the same speed, and both values resulted in two linear independent base vectors. Furthermore, the theoretical value obtain was $p \approx 2.7$, which further corroborates the results for the value of the bond dimension obtained.

Throughout this project, we studied the potential of tensor networks and realized that it, indeed, solves the problem we aimed to solve. Nonetheless, it is a method where the complexity level is not justified since we can achieve the same result using simpler methods such as generative models or graph neural networks. However, it is proved to be a quicker method, when compared with the ones mentioned before. Furthermore, the fact that tensor networks have a quantum nature makes it a great candidate for quantum computation, outperforming other methods with a classical nature. The statement that tensor networks is more complex than graph neural networks and generative models is only valid when working with classical computers. If the day comes when we are able to work in a quantum computer, then tensor networks will be simpler since it is being applied in its natural environment. In [36], the reader can find a book already explaining how to use tensor networks to simulate quantum systems and circuits using tensor networks to represent operators and gates, such as the Hadamard operator and the Pauli matrices. Moreover, there are already algorithms to make use of this operators such as the Shor's algorithm, the Bernstein–Vazirani algorithm and the Deutsch–Jozsa algorithm [52].

Bibliography

[1] R. Orús. “Tensor networks for complex quantum systems”. In: *Nature Reviews Physics* 1.9 (Aug. 2019), pp. 538–550. DOI: 10.1038/s42254-019-0086-7. URL: https://doi.org/10.1038%5C%2Fs42254-019-0086-7.

[2] G. Vidal. “Class of Quantum Many-Body States That Can Be Efficiently Simulated”. In: *Physics Review Letters* 101(11) (Sept. 2008), p. 110501. DOI: 10.1103/PhysRevLett.101.110501. URL: https://link.aps.org/doi/10.1103/PhysRevLett.101.110501.

[3] M. Stoudenmire. “Introduction to Tensor Networks for Machine Learning”. In: *Condensed Matter Physics In the City* (June 2020). URL: https://www.youtube.com/watch?v=vgd0J4VujBE.

[4] G. Vidal. “Entanglement Renormalization”. In: *Understanding Quantum Phase Transitions. Series: Condensed Matter Physics.* 2010, pp. 31–58. DOI: 10.1201/b10273-7.

[5] C. Fischer. “General Hartree-Fock program”. In: *Computer Physics Communications* 43.3 (1987), pp. 355–365. ISSN: 0010-4655. DOI: https://doi.org/10.1016/0010-4655(87)90053-1. URL: https://www.sciencedirect.com/science/article/pii/0010465587900531.

[6] R. Rubinstein and D. Kroese. *Simulation and the Monte Carlo method.* 3rd ed. Wiley, 2011.

[7] S. White. “Density matrix formulation for quantum renormalization groups”. In: *Physics Review Letters* 69 (19 Nov. 1992), pp. 2863–2866. DOI: 10.1103/PhysRevLett.69.2863. URL: https://link.aps.org/doi/10.1103/PhysRevLett.69.2863.

[8] D. Ashton. *The Renormalisation Group.* 2022. URL: https://blog.dougashton.net/2012/04/the-renormalisation-group/.

[9] S. Montangero. *Introduction to tensor network methods.* Springer, 2018.

[10] H. Sayama and C. Laramee. “Generative Network Automata: A Generalized Framework for Modeling Adaptive Network Dynamics Using Graph Rewritings”. In: *Understanding Complex Systems* 2009 (Feb. 2009). DOI: 10.1007/978-3-642-01284-6_15.

[11] P. Mistani, S. Pakravan, and F. Gibou. “Towards a tensor network representation of complex systems”. In: Oct. 2018.

[12] V. Parigi. *Quantum Complex Networks — Quantum Optics group.* 2022. URL: http://www.lkb.upmc.fr/quantumoptics/quantum-complex-networks.

[13] A. Mata. “Complex Networks: a Mini-review”. In: *Brazilian Journal of Physics* 50.5 (2020), pp. 658–672. DOI: 10.1007/s13538-020-00772-9.

[14] M. Ahuja and K. Sharma. “Complex Networks: A Review”. In: *International Journal of Computer Applications* 101.15 (2014), pp. 31–35. DOI: 10.5120/17765-8882.

[15] F. Bloch and M. Jackson. “Centrality Measures in Networks”. In: *SSRN Electronic Journal* (2016). DOI: 10.2139/ssrn.2749124.

[16] Jinhu Lü et al. “Theory and applications of complex networks: Advances and challenges”. In: *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2013, pp. 2291–2294. DOI: 10.1109/ISCAS.2013.6572335.

[17] L. Oliveira et al. “Analyzing and modeling real-world phenomena with complex networks: a survey of applications”. In: *Advances in Physics* 60.3 (June 2011), pp. 329–412. DOI: 10.1080/00018732.2011.572452. URL: https://doi.org/10.1080%5C%2F00018732.2011.572452.

[18] Flatiron Institute. *The Tensor Network*. 2022. URL: https://tensornetwork.org/diagrams/.

[19] B. Schutz. *Geometrical methods of mathematical physics*. World Publishing Corporation, 2009.

[20] Flatiron Institute. *The Tensor Network*. 2022. URL: https://tensornetwork.org/mps/.

[21] J. Biamonte and V. Bergholm. *Tensor Networks in a Nutshell*. 2017. DOI: 10.48550/ARXIV.1708.00006. URL: https://arxiv.org/abs/1708.00006.

[22] Flatiron Institute. *The Tensor Network*. 2022. URL: https://tensornetwork.org/mpo/.

[23] C. Hubig, P. McCulloch, and U. Schollwöck. “Generic construction of efficient matrix product operators”. In: *Physics Review B* 95 (3 Jan. 2017), p. 035129. DOI: 10.1103/PhysRevB.95.035129. URL: https://link.aps.org/doi/10.1103/PhysRevB.95.035129.

[24] J. Cirac et al. “Matrix product states and projected entangled pair states: Concepts, symmetries, theorems”. In: *Reviews of Modern Physics* 93.4 (Dec. 2021). DOI: 10.1103/revmodphys.93.045003. URL: https://doi.org/10.1103%5C%2Frevmodphys.93.045003.

[25] R. Orús. “A practical introduction to tensor networks: Matrix product states and projected entangled pair states”. In: *Annals of Physics* 349 (Oct. 2014), pp. 117–158. DOI: 10.1016/j.aop.2014.06.013. URL: https://doi.org/10.1016%5C%2Fj.aop.2014.06.013.

[26] F. Verstraete, V. Murg, and J. Cirac. “Matrix product states, projected entangled pair states, and variational renormalization group methods for quantum spin systems”. In: *Advances in Physics* 57.2 (Mar. 2008), pp. 143–224. DOI: 10.1080/14789940801912366. URL: https://doi.org/10.1080%5C%2F14789940801912366.

[27] Y. Pang et al. *Efficient 2D Tensor Network Simulation of Quantum Systems*. 2020. DOI: 10.48550/ARXIV.2006.15234. URL: https://arxiv.org/abs/2006.15234.

[28] S. Ran et al. *Tensor Network Contractions: Lecture Notes in Physics*. SpringerOpen, 2020.

[29] G. Vidal. “Entanglement Renormalization”. In: *Physics Review Letters* 99(22) (Nov. 2007), p. 220405. DOI: 10.1103/PhysRevLett.99.220405. URL: https://link.aps.org/doi/10.1103/PhysRevLett.99.220405.

[30] N. Bao et al. "Consistency conditions for an AdS multiscale entanglement renormalization ansatz correspondence". In: *Physics Review D* 91(12) (June 2015), p. 125036. DOI: 10.1103/PhysRevD.91.125036. URL: https://link.aps.org/doi/10.1103/PhysRevD.91.125036.

[31] Y. Meng and X. Liu. "Finding Central Vertices and Community Structure via Extended Density Peaks-Based Clustering". In: *Information* 12.12 (2021). ISSN: 2078-2489. DOI: 10.3390/info12120501. URL: https://www.mdpi.com/2078-2489/12/12/501.

[32] F. Amirouche and F. Amirouche. *Fundamentals of multibody dynamics.* Birkhäuser, 2006.

[33] E. Neil. "Classical Mechanics and Math Methods II. Generalized Coordinates". In: (2020).

[34] R. Serway and J. Jewett. *Physics for scientists and engineers.* 9th ed. 2013.

[35] G. Chan. *Matrix product states, DMRG, and tensor networks.* Youtube, Cornell Laboratory of Atomic and Solid State Physics. 2015. URL: https://www.youtube.com/watch?v=Q8bFmV6tHBs.

[36] J. Biamonte. *Lectures on Quantum Tensor Networks.* 2019. DOI: 10.48550/ARXIV.1912.10049. URL: https://arxiv.org/abs/1912.10049.

[37] K. Lange. "Singular Value Decomposition". In: *Numerical Analysis for Statisticians* (2010), pp. 129–142. DOI: 10.1007/978-1-4419-5945-4_9.

[38] S. Brunton. *Singular Value Decomposition (SVD): Mathematical Overview, Singular Value Decomposition.* Youtube, Steve Brunton. 2020. URL: https://www.youtube.com/watch?v=nbBvuuNVfco.

[39] S. Brunton. *Singular Value Decomposition (SVD): Matrix Approximation, Singular Value Decomposition.* Youtube, Steve Brunton. 2020. URL: https://www.youtube.com/watch?v=xy3QyyhiuY4t=2s.

[40] S. Brunton. *SVD and Optimal Truncation, Singular Value Decomposition.* Youtube, Steve Brunton. 2020. URL: https://www.youtube.com/watch?v=9vJDjkx825k.

[41] M. Gavish and D. Donoho. *The Optimal Hard Threshold for Singular Values is 4/sqrt(3).* 2013. DOI: 10.48550/ARXIV.1305.5870. URL: https://arxiv.org/abs/1305.5870.

[42] U. Schollwöck. "The density-matrix renormalization group in the age of matrix product states". In: *Annals of Physics* 326.1 (Jan. 2011), pp. 96–192. DOI: 10.1016/j.aop.2010.09.012. URL: https://doi.org/10.1016%5C%2Fj.aop.2010.09.012.

[43] I. McCulloch. "From density-matrix renormalization group to matrix product states". In: *Journal of Statistical Mechanics: Theory and Experiment* 2007.10 (Oct. 2007), P10014–P10014. DOI: 10.1088/1742-5468/2007/10/p10014. URL: https://doi.org/10.1088/1742-5468/2007/10/p10014.

[44] B. Pirvu et al. "Matrix product operator representations". In: *New Journal of Physics* 12.2 (Feb. 2010), p. 025012. DOI: 10.1088/1367-2630/12/2/025012. URL: https://doi.org/10.1088/1367-2630/12/2/025012.

[45] A. Pires. "Theoretical Tools for Spin Models in Magnetic Systems". In: (2021). DOI: 10.1088/978-0-7503-3879-0.

[46] U. Schollwöck. “The density-matrix renormalization group”. In: *Reviews of Modern Physics* 77.1 (Apr. 2005), pp. 259–315. DOI: `10.1103/revmodphys.77.259`. URL: `https://doi.org/10.1103%5C%2Frevmodphys.77.259`.

[47] M. Wall and L. Carr. “Out-of-equilibrium dynamics with matrix product states”. In: *New Journal of Physics* 14.12 (Dec. 2012), p. 125015. DOI: `10.1088/1367-2630/14/12/125015`. URL: `https://doi.org/10.1088/1367-2630/14/12/125015`.

[48] R. Albert and A. Barabási. “Statistical mechanics of complex networks”. In: *Reviews of Modern Physics* 74(1) (Jan. 2002), pp. 71–75. DOI: `10.1103/RevModPhys.74.47`. URL: `https://link.aps.org/doi/10.1103/RevModPhys.74.47`.

[49] URL: `https://github.com/TunnelTechnologies/DMRG-exact`.

[50] N. Halko, P. Martinsson, and J. Tropp. “Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions”. In: (2009). DOI: `10.48550/ARXIV.0909.4061`. URL: `https://arxiv.org/abs/0909.4061`.

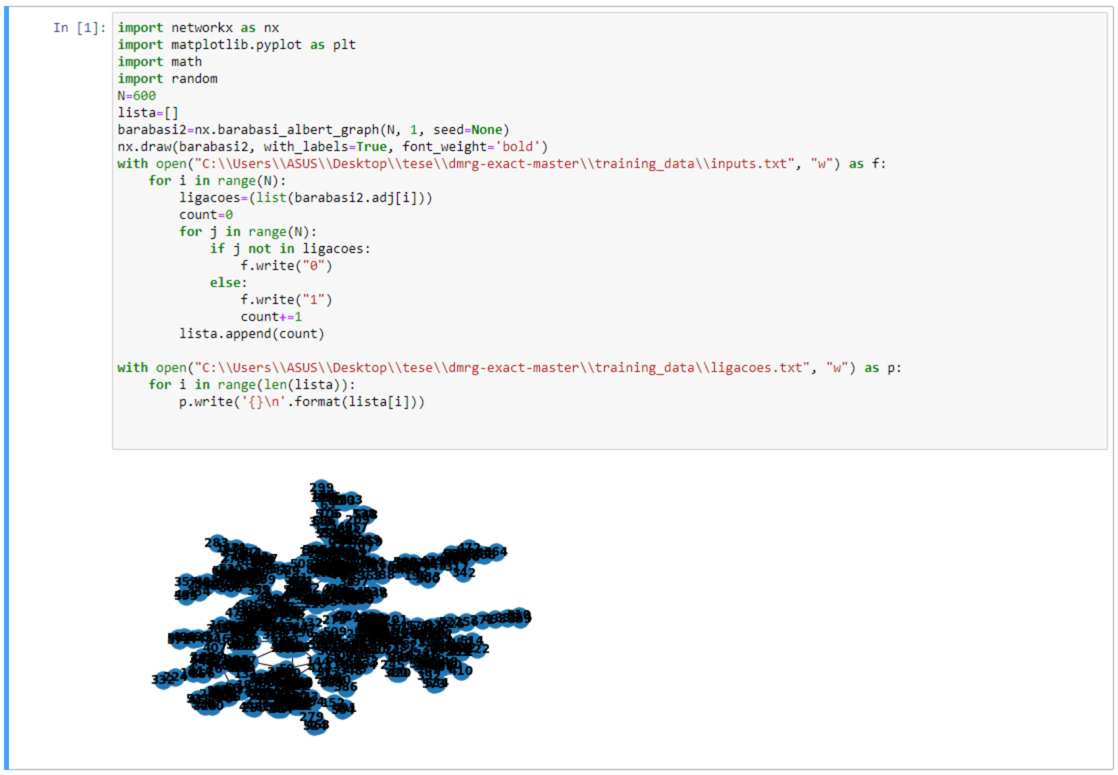
[51] A. Cichocki et al. “Tensor Networks for Dimensionality Reduction and Large-scale Optimization: Part 1 Low-Rank Tensor Decompositions”. In: *Foundations and Trends® in Machine Learning* 9.4-5 (2016), pp. 249–429. DOI: `10.1561/2200000059`. URL: `https://doi.org/10.1561%5C%2F2200000059`.

[52] A. Harrow, A. Hassidim, and S. Lloyd. “Quantum Algorithm for Linear Systems of Equations”. In: *Physics Review Letters* 103(15) (Oct. 2009), p. 150502. DOI: `10.1103/PhysRevLett.103.150502`. URL: `https://link.aps.org/doi/10.1103/PhysRevLett.103.150502`.

Appendices

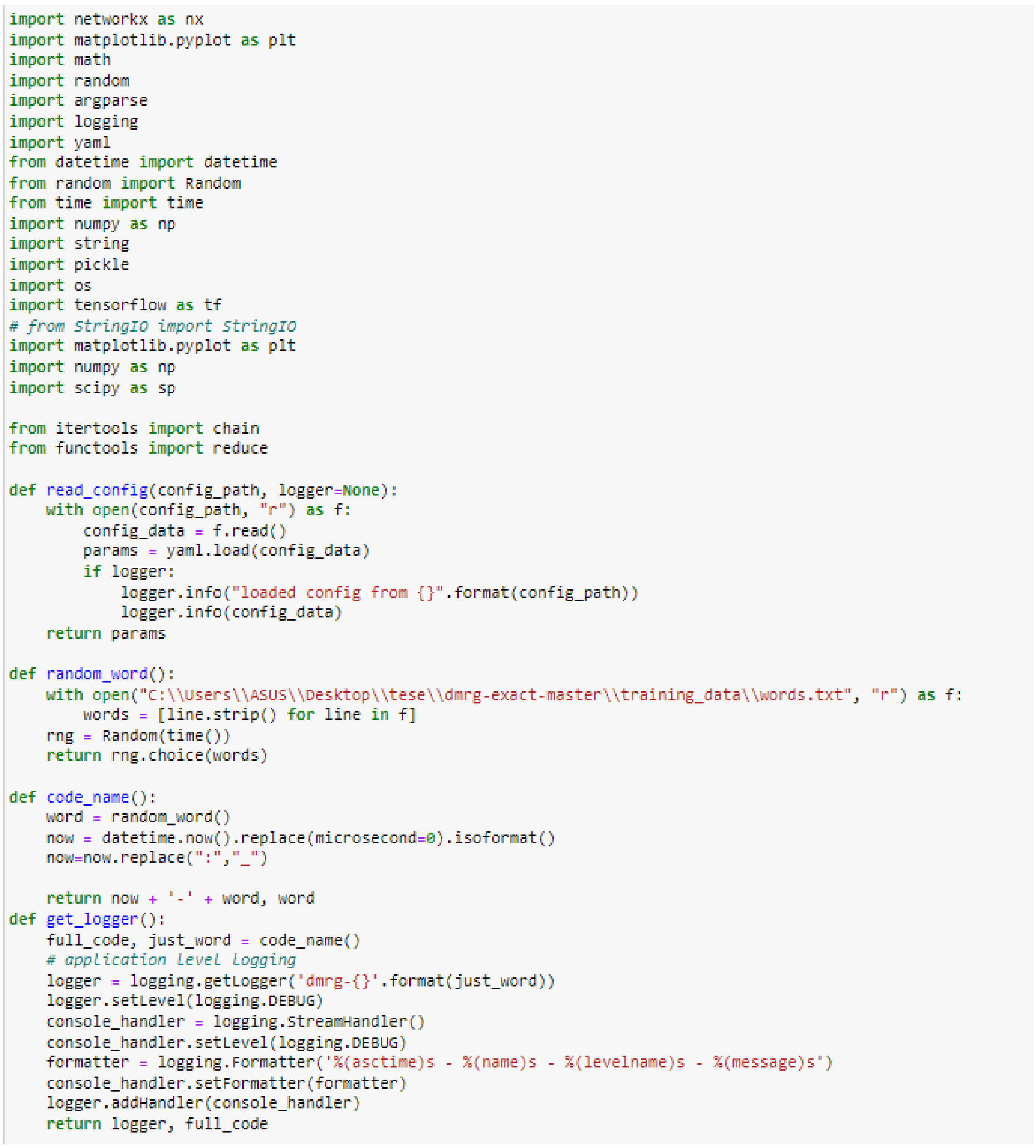
Appendix A

Code



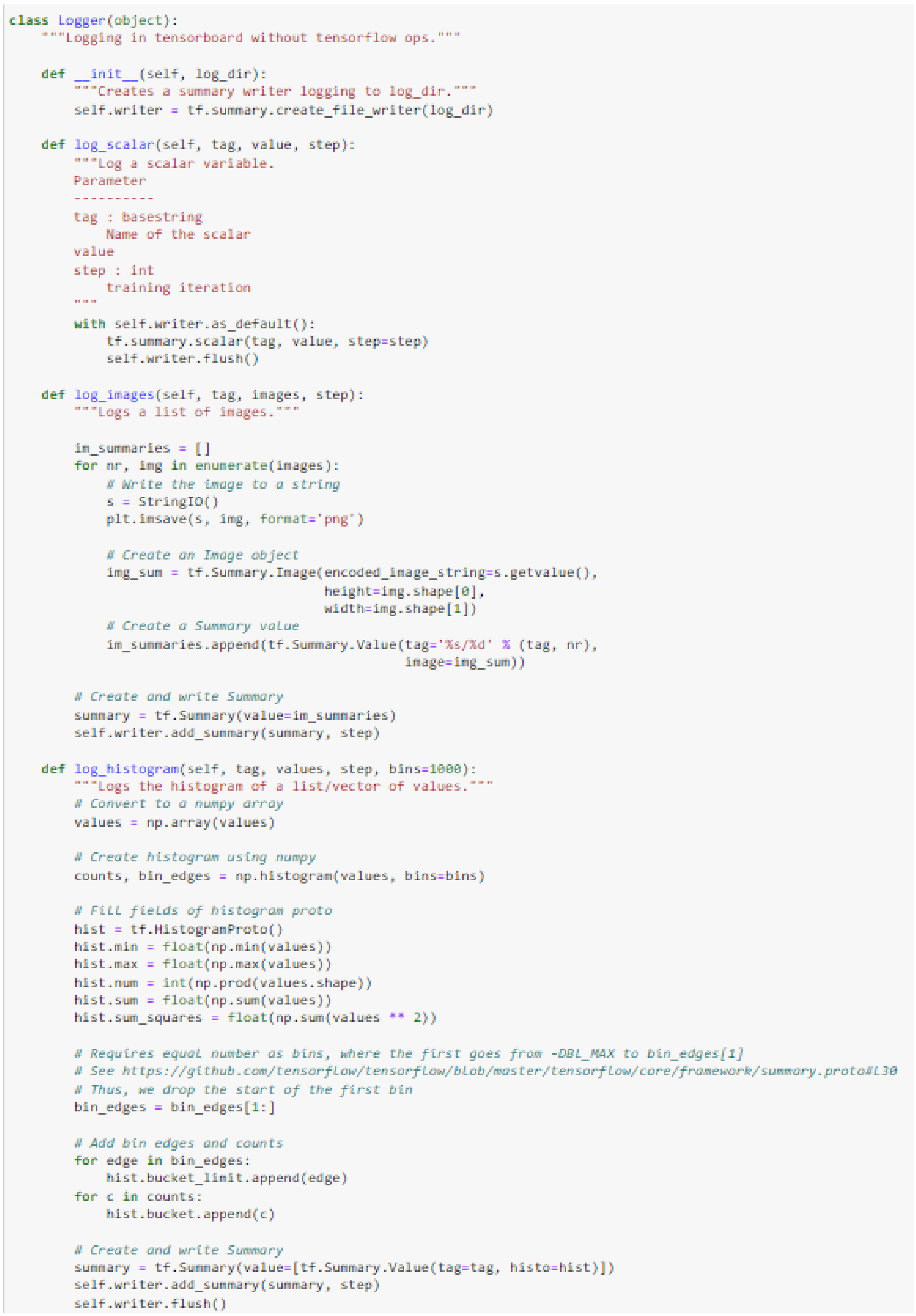
```
In [1]: import networkx as nx
        import matplotlib.pyplot as plt
        import math
        import random
        N=600
        lista=[]
        barabasi2=nx.barabasi_albert_graph(N, 1, seed=None)
        nx.draw(barabasi2, with_labels=True, font_weight='bold')
        with open("C:\\Users\\ASUS\\Desktop\\tese\\dmrg-exact-master\\training_data\\inputs.txt", "w") as f:
            for i in range(N):
                ligacoes=(list(barabasi2.adj[i]))
                count=0
                for j in range(N):
                    if j not in ligacoes:
                        f.write("0")
                    else:
                        f.write("1")
                        count+=1
                lista.append(count)

        with open("C:\\Users\\ASUS\\Desktop\\tese\\dmrg-exact-master\\training_data\\ligacoes.txt", "w") as p:
            for i in range(len(lista)):
                p.write('{}\n'.format(lista[i]))
```

Figure A.1: Code used to generate the Barabási network and extract the connectivity of each node.



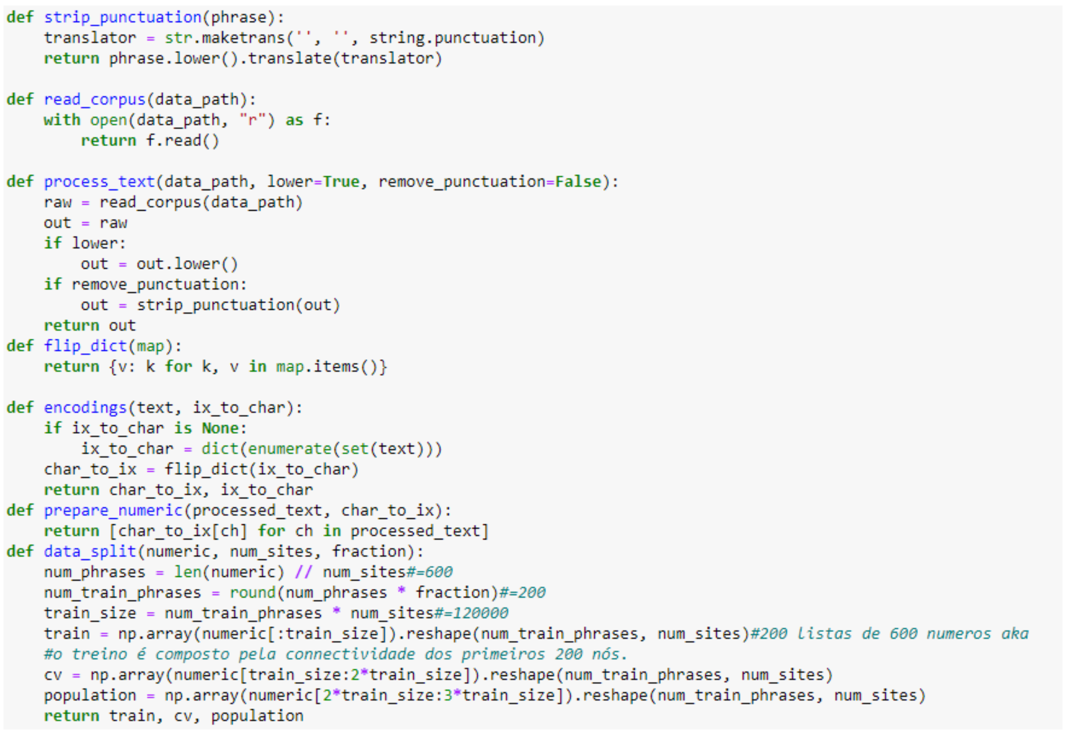
```python
import networkx as nx
import matplotlib.pyplot as plt
import math
import random
import argparse
import logging
import yaml
from datetime import datetime
from random import Random
from time import time
import numpy as np
import string
import pickle
import os
import tensorflow as tf
# from StringIO import StringIO
import matplotlib.pyplot as plt
import numpy as np
import scipy as sp

from itertools import chain
from functools import reduce

def read_config(config_path, logger=None):
    with open(config_path, "r") as f:
        config_data = f.read()
        params = yaml.load(config_data)
        if logger:
            logger.info("loaded config from {}".format(config_path))
            logger.info(config_data)
    return params

def random_word():
    with open("C:\\Users\\ASUS\\Desktop\\tese\\dmrg-exact-master\\training_data\\words.txt", "r") as f:
        words = [line.strip() for line in f]
    rng = Random(time())
    return rng.choice(words)

def code_name():
    word = random_word()
    now = datetime.now().replace(microsecond=0).isoformat()
    now=now.replace(":","_")

    return now + '-' + word, word
def get_logger():
    full_code, just_word = code_name()
    # application level logging
    logger = logging.getLogger('dmrg-{}'.format(just_word))
    logger.setLevel(logging.DEBUG)
    console_handler = logging.StreamHandler()
    console_handler.setLevel(logging.DEBUG)
    formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
    console_handler.setFormatter(formatter)
    logger.addHandler(console_handler)
    return logger, full_code
```

Figure A.2: Code used to for the configurations of the algorithm.



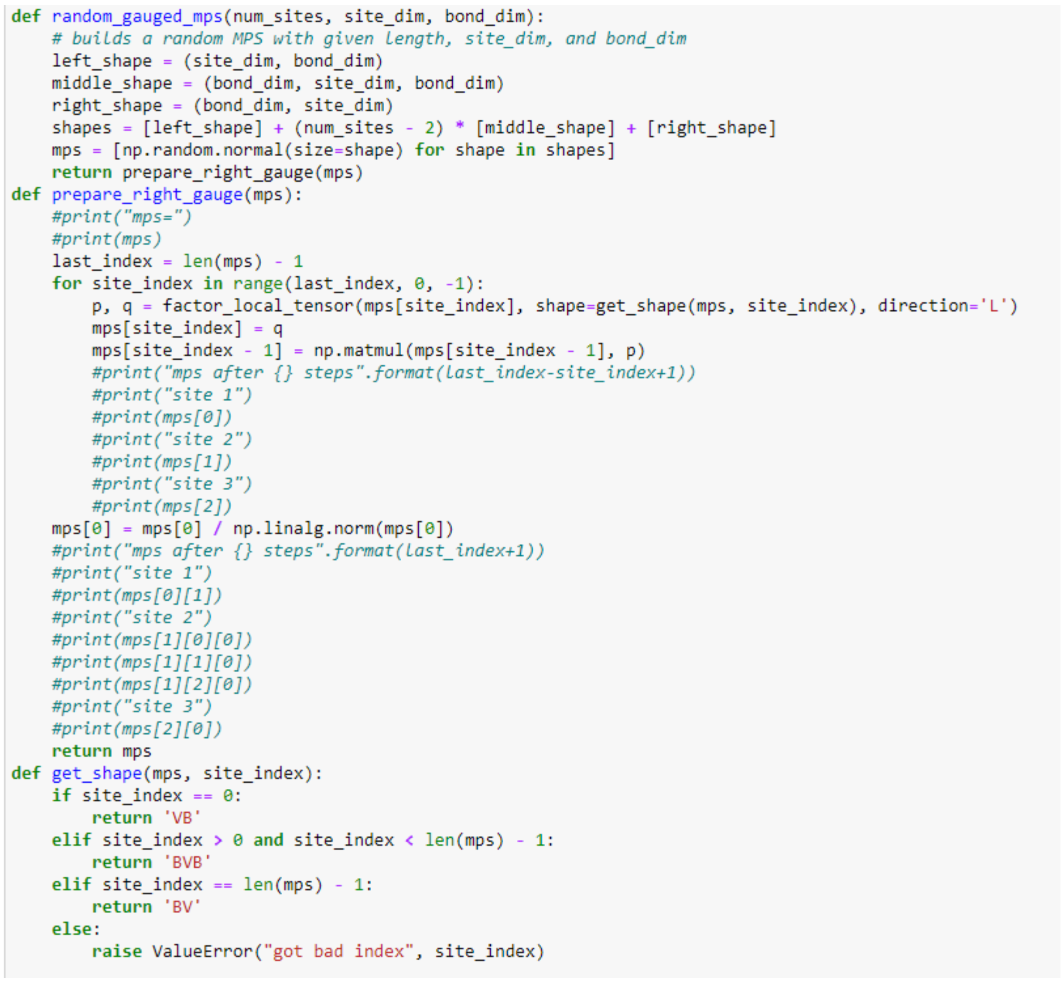
```python
class Logger(object):
    """Logging in tensorboard without tensorflow ops."""

    def __init__(self, log_dir):
        """Creates a summary writer logging to log_dir."""
        self.writer = tf.summary.create_file_writer(log_dir)

    def log_scalar(self, tag, value, step):
        """Log a scalar variable.
        Parameter
        ----------
        tag : basestring
            Name of the scalar
        value
        step : int
            training iteration
        """
        with self.writer.as_default():
            tf.summary.scalar(tag, value, step=step)
            self.writer.flush()

    def log_images(self, tag, images, step):
        """Logs a list of images."""

        im_summaries = []
        for nr, img in enumerate(images):
            # Write the image to a string
            s = StringIO()
            plt.imsave(s, img, format='png')

            # Create an Image object
            img_sum = tf.Summary.Image(encoded_image_string=s.getvalue(),
                                       height=img.shape[0],
                                       width=img.shape[1])
            # Create a Summary value
            im_summaries.append(tf.Summary.Value(tag='%s/%d' % (tag, nr),
                                                 image=img_sum))

        # Create and write Summary
        summary = tf.Summary(value=im_summaries)
        self.writer.add_summary(summary, step)

    def log_histogram(self, tag, values, step, bins=1000):
        """Logs the histogram of a list/vector of values."""
        # Convert to a numpy array
        values = np.array(values)

        # Create histogram using numpy
        counts, bin_edges = np.histogram(values, bins=bins)

        # Fill fields of histogram proto
        hist = tf.HistogramProto()
        hist.min = float(np.min(values))
        hist.max = float(np.max(values))
        hist.num = int(np.prod(values.shape))
        hist.sum = float(np.sum(values))
        hist.sum_squares = float(np.sum(values ** 2))

        # Requires equal number as bins, where the first goes from -DBL_MAX to bin_edges[1]
        # See https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/summary.proto#L30
        # Thus, we drop the start of the first bin
        bin_edges = bin_edges[1:]

        # Add bin edges and counts
        for edge in bin_edges:
            hist.bucket_limit.append(edge)
        for c in counts:
            hist.bucket.append(c)

        # Create and write Summary
        summary = tf.Summary(value=[tf.Summary.Value(tag=tag, histo=hist)])
        self.writer.add_summary(summary, step)
        self.writer.flush()
```

Figure A.3: Code used to set up the Logger.



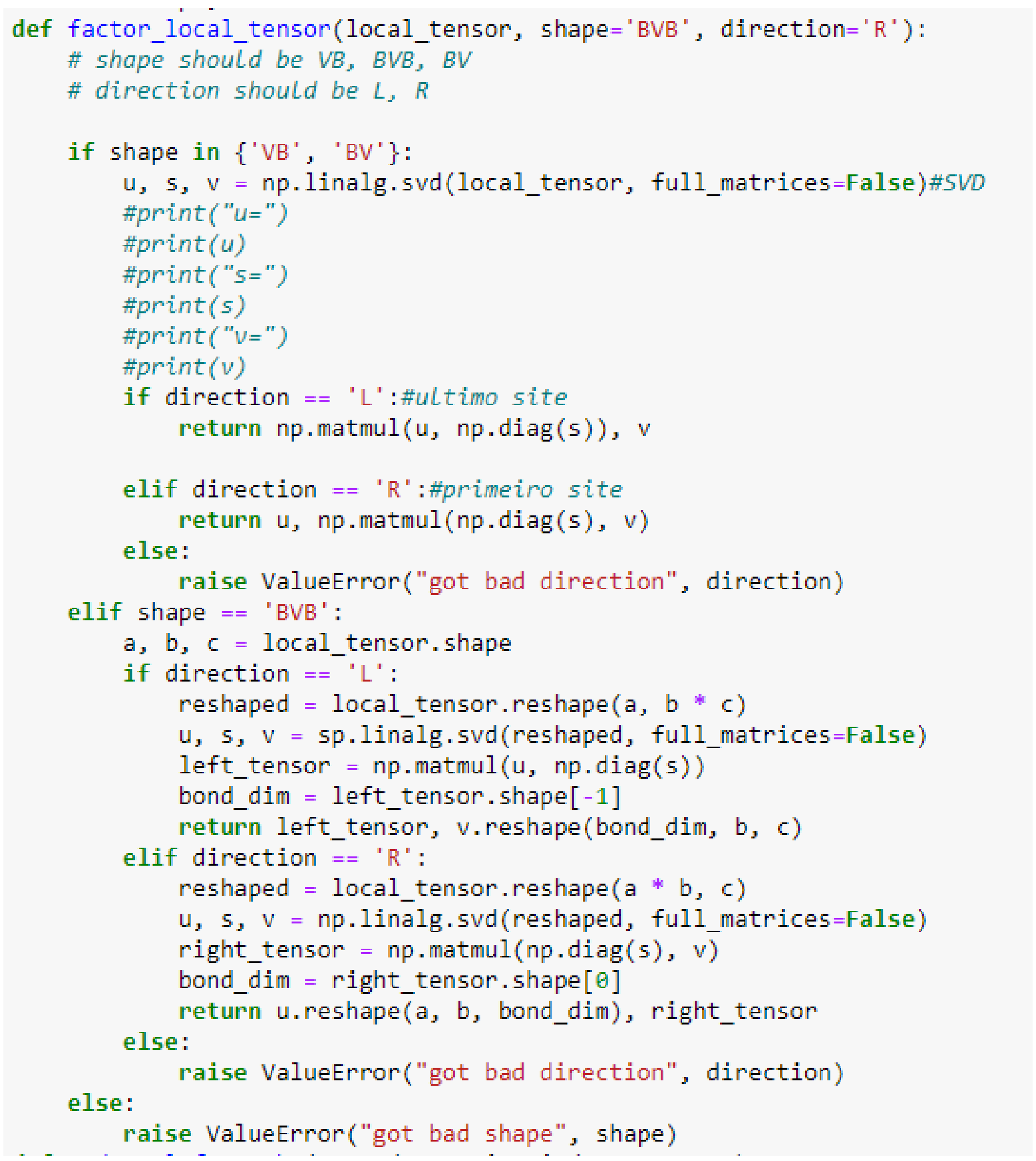
```python
def strip_punctuation(phrase):
    translator = str.maketrans('', '', string.punctuation)
    return phrase.lower().translate(translator)

def read_corpus(data_path):
    with open(data_path, "r") as f:
        return f.read()

def process_text(data_path, lower=True, remove_punctuation=False):
    raw = read_corpus(data_path)
    out = raw
    if lower:
        out = out.lower()
    if remove_punctuation:
        out = strip_punctuation(out)
    return out
def flip_dict(map):
    return {v: k for k, v in map.items()}

def encodings(text, ix_to_char):
    if ix_to_char is None:
        ix_to_char = dict(enumerate(set(text)))
    char_to_ix = flip_dict(ix_to_char)
    return char_to_ix, ix_to_char
def prepare_numeric(processed_text, char_to_ix):
    return [char_to_ix[ch] for ch in processed_text]
def data_split(numeric, num_sites, fraction):
    num_phrases = len(numeric) // num_sites#=600
    num_train_phrases = round(num_phrases * fraction)#=200
    train_size = num_train_phrases * num_sites#=120000
    train = np.array(numeric[:train_size]).reshape(num_train_phrases, num_sites)#200 listas de 600 numeros aka
    #o treino é composto pela connectividade dos primeiros 200 nós.
    cv = np.array(numeric[train_size:2*train_size]).reshape(num_train_phrases, num_sites)
    population = np.array(numeric[2*train_size:3*train_size]).reshape(num_train_phrases, num_sites)
    return train, cv, population
```

Figure A.4: Code used to process the data.



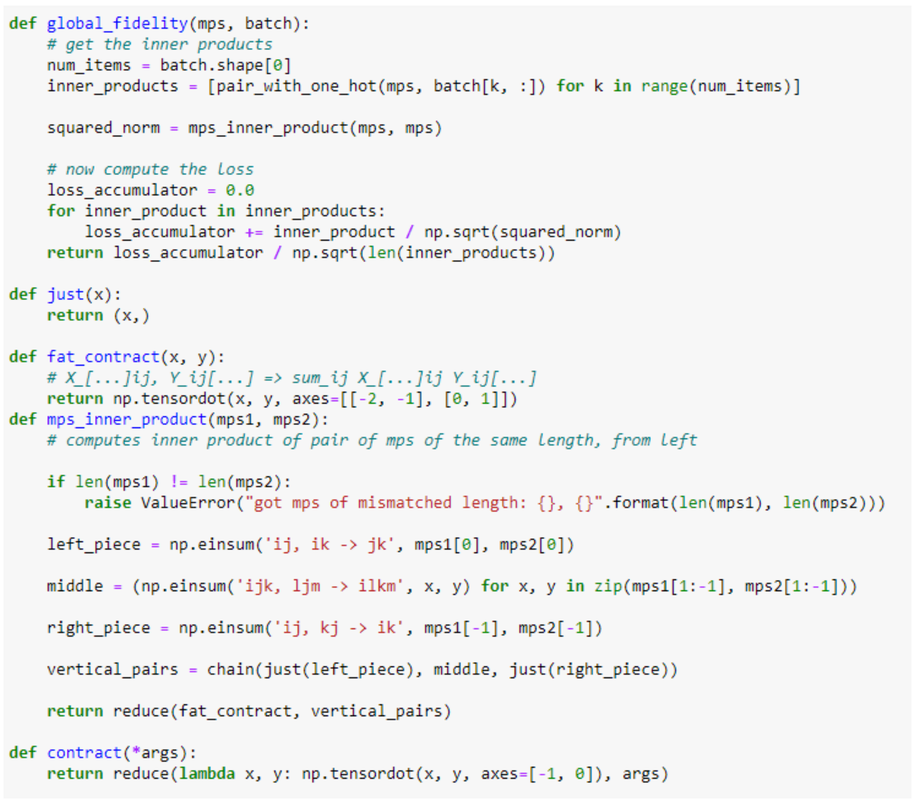
```python
def random_gauged_mps(num_sites, site_dim, bond_dim):
    # builds a random MPS with given length, site_dim, and bond_dim
    left_shape = (site_dim, bond_dim)
    middle_shape = (bond_dim, site_dim, bond_dim)
    right_shape = (bond_dim, site_dim)
    shapes = [left_shape] + (num_sites - 2) * [middle_shape] + [right_shape]
    mps = [np.random.normal(size=shape) for shape in shapes]
    return prepare_right_gauge(mps)
def prepare_right_gauge(mps):
    #print("mps=")
    #print(mps)
    last_index = len(mps) - 1
    for site_index in range(last_index, 0, -1):
        p, q = factor_local_tensor(mps[site_index], shape=get_shape(mps, site_index), direction='L')
        mps[site_index] = q
        mps[site_index - 1] = np.matmul(mps[site_index - 1], p)
        #print("mps after {} steps".format(last_index-site_index+1))
        #print("site 1")
        #print(mps[0])
        #print("site 2")
        #print(mps[1])
        #print("site 3")
        #print(mps[2])
    mps[0] = mps[0] / np.linalg.norm(mps[0])
    #print("mps after {} steps".format(last_index+1))
    #print("site 1")
    #print(mps[0][1])
    #print("site 2")
    #print(mps[1][0][0])
    #print(mps[1][1][0])
    #print(mps[1][2][0])
    #print("site 3")
    #print(mps[2][0])
    return mps
def get_shape(mps, site_index):
    if site_index == 0:
        return 'VB'
    elif site_index > 0 and site_index < len(mps) - 1:
        return 'BVB'
    elif site_index == len(mps) - 1:
        return 'BV'
    else:
        raise ValueError("got bad index", site_index)
```

Figure A.5: Code used to generate the sampling MPS and the absorption of the result of the SVD.



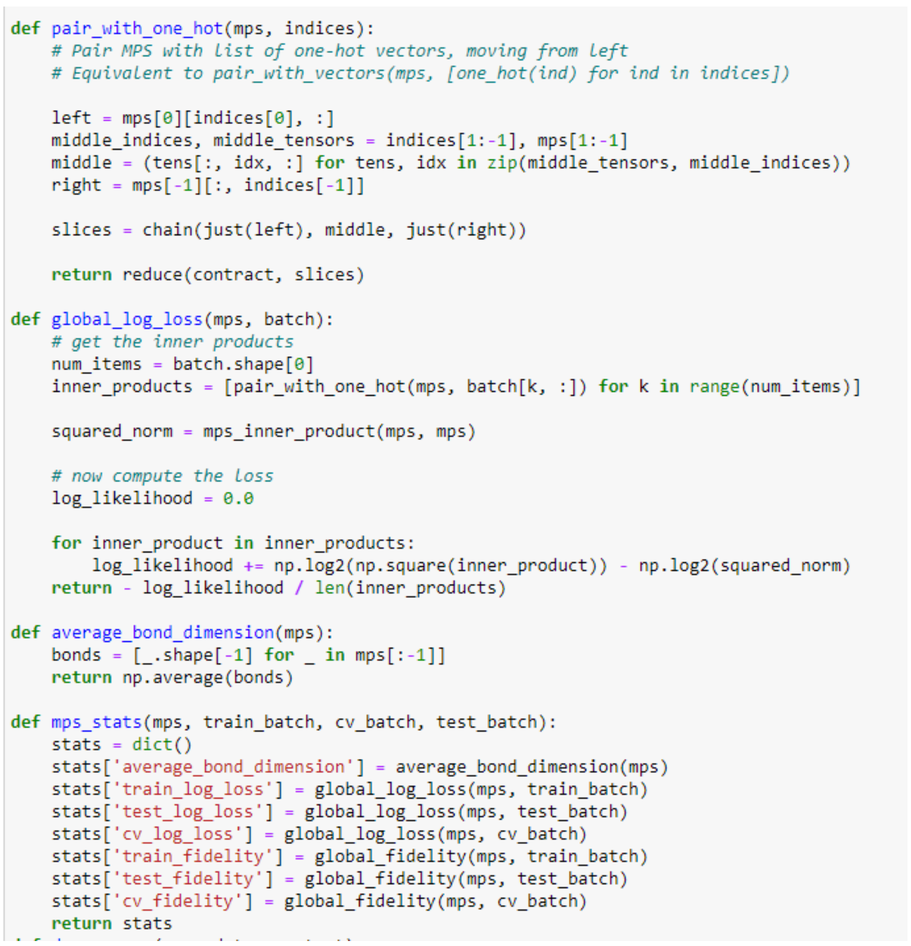
```python
def factor_local_tensor(local_tensor, shape='BVB', direction='R'):
    # shape should be VB, BVB, BV
    # direction should be L, R

    if shape in {'VB', 'BV'}:
        u, s, v = np.linalg.svd(local_tensor, full_matrices=False)#SVD
        #print("u=")
        #print(u)
        #print("s=")
        #print(s)
        #print("v=")
        #print(v)
        if direction == 'L':#ultimo site
            return np.matmul(u, np.diag(s)), v

        elif direction == 'R':#primeiro site
            return u, np.matmul(np.diag(s), v)
        else:
            raise ValueError("got bad direction", direction)
    elif shape == 'BVB':
        a, b, c = local_tensor.shape
        if direction == 'L':
            reshaped = local_tensor.reshape(a, b * c)
            u, s, v = sp.linalg.svd(reshaped, full_matrices=False)
            left_tensor = np.matmul(u, np.diag(s))
            bond_dim = left_tensor.shape[-1]
            return left_tensor, v.reshape(bond_dim, b, c)
        elif direction == 'R':
            reshaped = local_tensor.reshape(a * b, c)
            u, s, v = np.linalg.svd(reshaped, full_matrices=False)
            right_tensor = np.matmul(np.diag(s), v)
            bond_dim = right_tensor.shape[0]
            return u.reshape(a, b, bond_dim), right_tensor
        else:
            raise ValueError("got bad direction", direction)
    else:
        raise ValueError("got bad shape", shape)
```

Figure A.6: Code used to perform the SVD



```python
def global_fidelity(mps, batch):
    # get the inner products
    num_items = batch.shape[0]
    inner_products = [pair_with_one_hot(mps, batch[k, :]) for k in range(num_items)]

    squared_norm = mps_inner_product(mps, mps)

    # now compute the loss
    loss_accumulator = 0.0
    for inner_product in inner_products:
        loss_accumulator += inner_product / np.sqrt(squared_norm)
    return loss_accumulator / np.sqrt(len(inner_products))

def just(x):
    return (x,)

def fat_contract(x, y):
    # X_[...]ij, Y_ij[...] => sum_ij X_[...]ij Y_ij[...]
    return np.tensordot(x, y, axes=[[-2, -1], [0, 1]])
def mps_inner_product(mps1, mps2):
    # computes inner product of pair of mps of the same length, from left

    if len(mps1) != len(mps2):
        raise ValueError("got mps of mismatched length: {}, {}".format(len(mps1), len(mps2)))

    left_piece = np.einsum('ij, ik -> jk', mps1[0], mps2[0])

    middle = (np.einsum('ijk, ljm -> ilkm', x, y) for x, y in zip(mps1[1:-1], mps2[1:-1]))

    right_piece = np.einsum('ij, kj -> ik', mps1[-1], mps2[-1])

    vertical_pairs = chain(just(left_piece), middle, just(right_piece))

    return reduce(fat_contract, vertical_pairs)

def contract(*args):
    return reduce(lambda x, y: np.tensordot(x, y, axes=[-1, 0]), args)
```

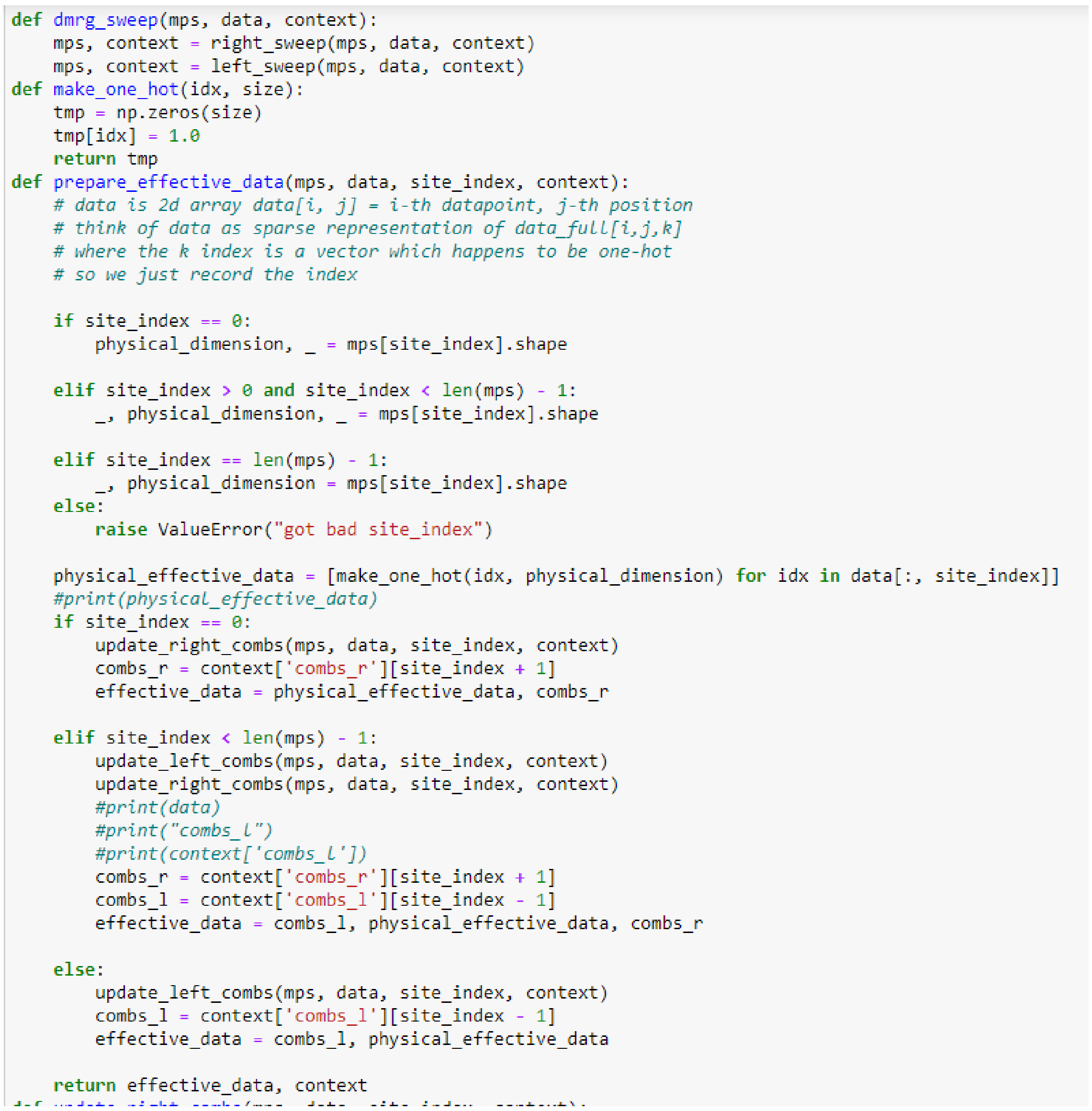
(a)



```python
def pair_with_one_hot(mps, indices):
    # Pair MPS with list of one-hot vectors, moving from left
    # Equivalent to pair_with_vectors(mps, [one_hot(ind) for ind in indices])

    left = mps[0][indices[0], :]
    middle_indices, middle_tensors = indices[1:-1], mps[1:-1]
    middle = (tens[:, idx, :] for tens, idx in zip(middle_tensors, middle_indices))
    right = mps[-1][:, indices[-1]]

    slices = chain(just(left), middle, just(right))

    return reduce(contract, slices)

def global_log_loss(mps, batch):
    # get the inner products
    num_items = batch.shape[0]
    inner_products = [pair_with_one_hot(mps, batch[k, :]) for k in range(num_items)]

    squared_norm = mps_inner_product(mps, mps)

    # now compute the loss
    log_likelihood = 0.0

    for inner_product in inner_products:
        log_likelihood += np.log2(np.square(inner_product)) - np.log2(squared_norm)
    return - log_likelihood / len(inner_products)

def average_bond_dimension(mps):
    bonds = [_.shape[-1] for _ in mps[:-1]]
    return np.average(bonds)

def mps_stats(mps, train_batch, cv_batch, test_batch):
    stats = dict()
    stats['average_bond_dimension'] = average_bond_dimension(mps)
    stats['train_log_loss'] = global_log_loss(mps, train_batch)
    stats['test_log_loss'] = global_log_loss(mps, test_batch)
    stats['cv_log_loss'] = global_log_loss(mps, cv_batch)
    stats['train_fidelity'] = global_fidelity(mps, train_batch)
    stats['test_fidelity'] = global_fidelity(mps, test_batch)
    stats['cv_fidelity'] = global_fidelity(mps, cv_batch)
    return stats
```

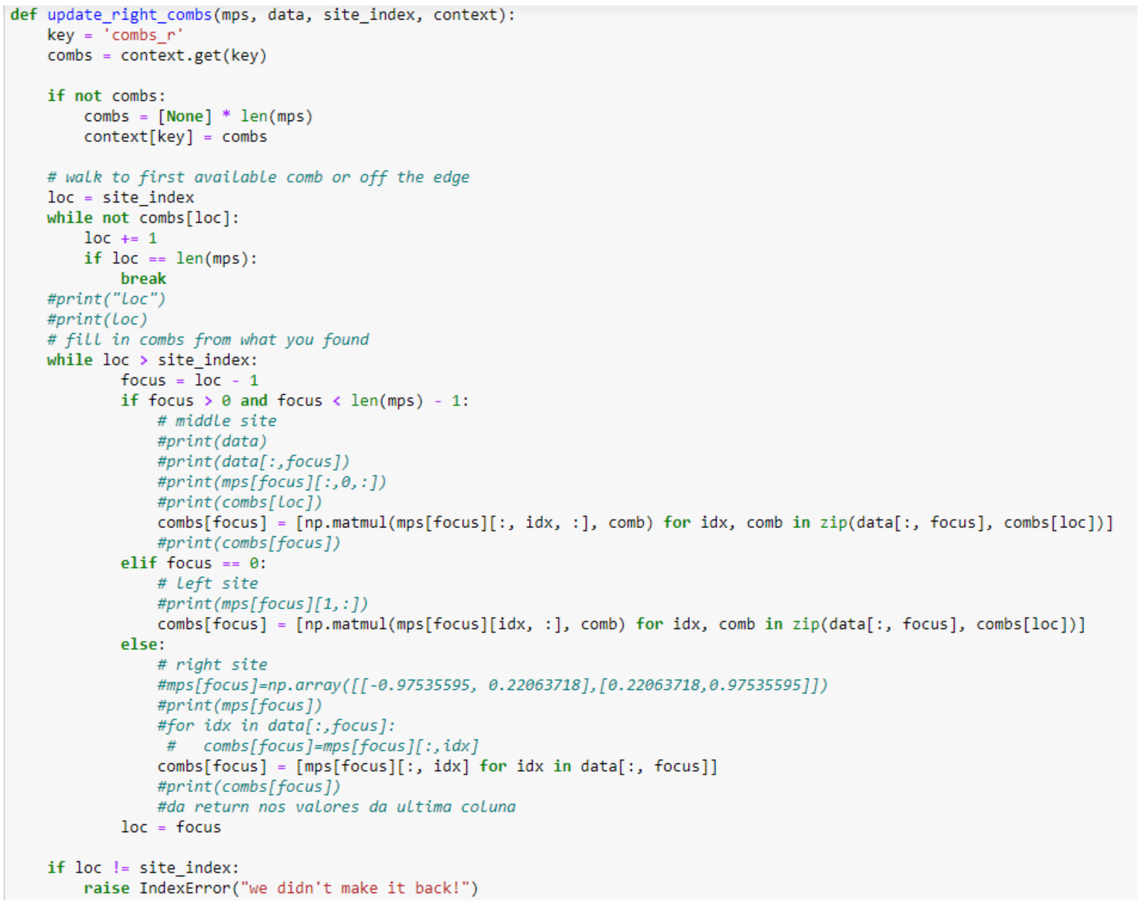
(b)

Figure A.7: Code used to evaluate the MPS.



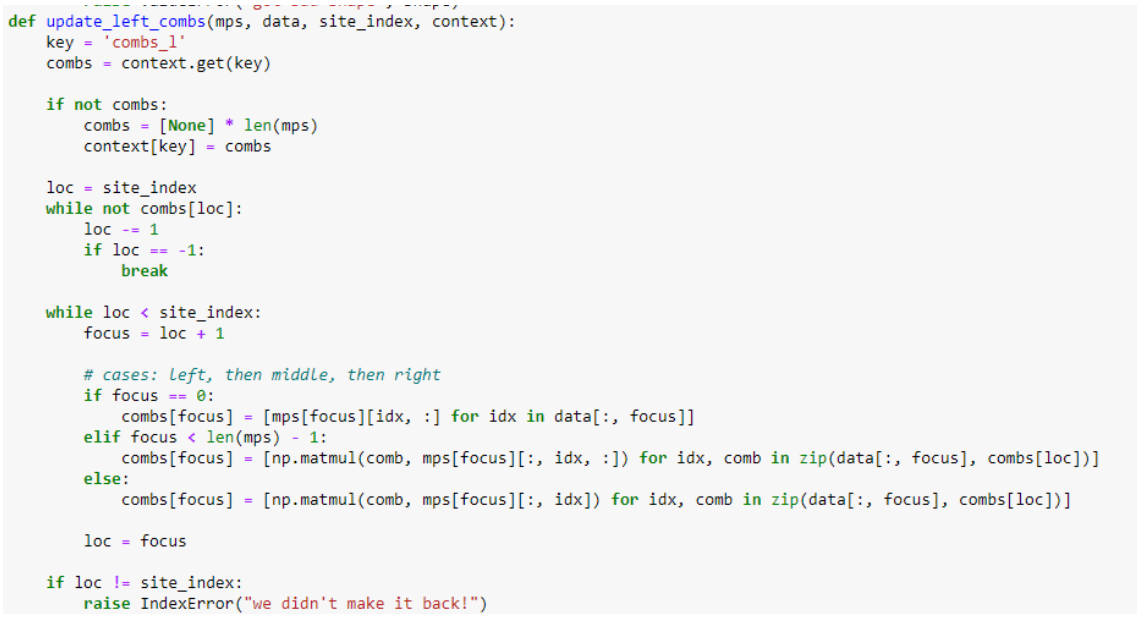
```
def dmrg_sweep(mps, data, context):
    mps, context = right_sweep(mps, data, context)
    mps, context = left_sweep(mps, data, context)
def make_one_hot(idx, size):
    tmp = np.zeros(size)
    tmp[idx] = 1.0
    return tmp
def prepare_effective_data(mps, data, site_index, context):
    # data is 2d array data[i, j] = i-th datapoint, j-th position
    # think of data as sparse representation of data_full[i,j,k]
    # where the k index is a vector which happens to be one-hot
    # so we just record the index

    if site_index == 0:
        physical_dimension, _ = mps[site_index].shape

    elif site_index > 0 and site_index < len(mps) - 1:
        _, physical_dimension, _ = mps[site_index].shape

    elif site_index == len(mps) - 1:
        _, physical_dimension = mps[site_index].shape
    else:
        raise ValueError("got bad site_index")

    physical_effective_data = [make_one_hot(idx, physical_dimension) for idx in data[:, site_index]]
    #print(physical_effective_data)
    if site_index == 0:
        update_right_combs(mps, data, site_index, context)
        combs_r = context['combs_r'][site_index + 1]
        effective_data = physical_effective_data, combs_r

    elif site_index < len(mps) - 1:
        update_left_combs(mps, data, site_index, context)
        update_right_combs(mps, data, site_index, context)
        #print(data)
        #print("combs_l")
        #print(context['combs_l'])
        combs_r = context['combs_r'][site_index + 1]
        combs_l = context['combs_l'][site_index - 1]
        effective_data = combs_l, physical_effective_data, combs_r

    else:
        update_left_combs(mps, data, site_index, context)
        combs_l = context['combs_l'][site_index - 1]
        effective_data = combs_l, physical_effective_data

    return effective_data, context
def update_right_combs(mps, data, site_index, context):
```

Figure A.8: Code used to perform the DMRG algorithm.



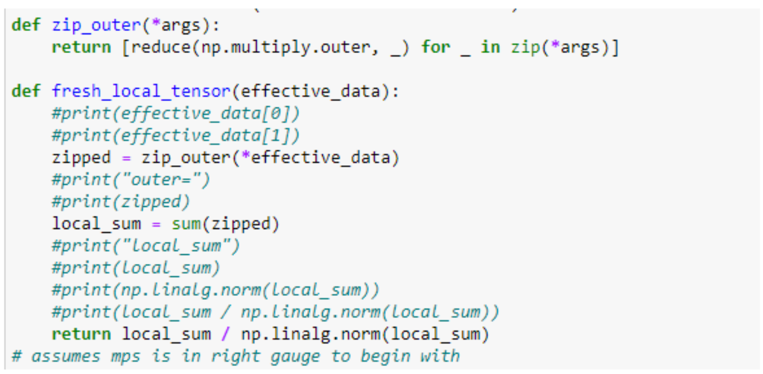
```python
def update_right_combs(mps, data, site_index, context):
    key = 'combs_r'
    combs = context.get(key)

    if not combs:
        combs = [None] * len(mps)
        context[key] = combs

    # walk to first available comb or off the edge
    loc = site_index
    while not combs[loc]:
        loc += 1
        if loc == len(mps):
            break
    #print("loc")
    #print(loc)
    # fill in combs from what you found
    while loc > site_index:
            focus = loc - 1
            if focus > 0 and focus < len(mps) - 1:
                # middle site
                #print(data)
                #print(data[:,focus])
                #print(mps[focus][:,0,:])
                #print(combs[loc])
                combs[focus] = [np.matmul(mps[focus][:, idx, :], comb) for idx, comb in zip(data[:, focus], combs[loc])]
                #print(combs[focus])
            elif focus == 0:
                # left site
                #print(mps[focus][1,:])
                combs[focus] = [np.matmul(mps[focus][idx, :], comb) for idx, comb in zip(data[:, focus], combs[loc])]
            else:
                # right site
                #mps[focus]=np.array([[-0.97535595, 0.22063718],[0.22063718,0.97535595]])
                #print(mps[focus])
                #for idx in data[:,focus]:
                 #   combs[focus]=mps[focus][:,idx]
                combs[focus] = [mps[focus][:, idx] for idx in data[:, focus]]
                #print(combs[focus])
                #da return nos valores da ultima coluna
            loc = focus

    if loc != site_index:
        raise IndexError("we didn't make it back!")
```

Figure A.9: Code used to perform the right sweep.



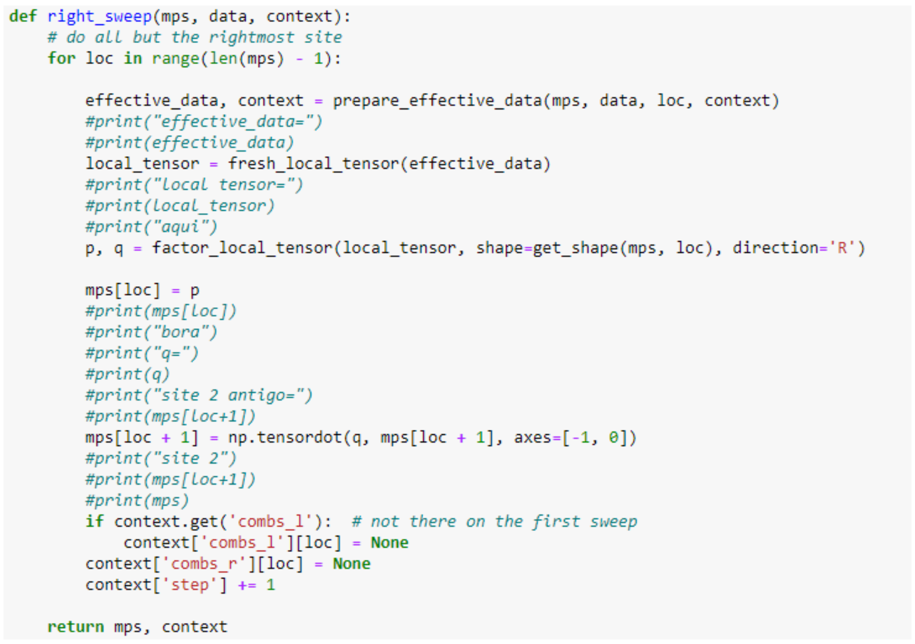
```python
def update_left_combs(mps, data, site_index, context):
    key = 'combs_l'
    combs = context.get(key)

    if not combs:
        combs = [None] * len(mps)
        context[key] = combs

    loc = site_index
    while not combs[loc]:
        loc -= 1
        if loc == -1:
            break

    while loc < site_index:
        focus = loc + 1

        # cases: left, then middle, then right
        if focus == 0:
            combs[focus] = [mps[focus][idx, :] for idx in data[:, focus]]
        elif focus < len(mps) - 1:
            combs[focus] = [np.matmul(comb, mps[focus][:, idx, :]) for idx, comb in zip(data[:, focus], combs[loc])]
        else:
            combs[focus] = [np.matmul(comb, mps[focus][:, idx]) for idx, comb in zip(data[:, focus], combs[loc])]

        loc = focus

    if loc != site_index:
        raise IndexError("we didn't make it back!")
```

Figure A.10: Code used to perform the left sweep.



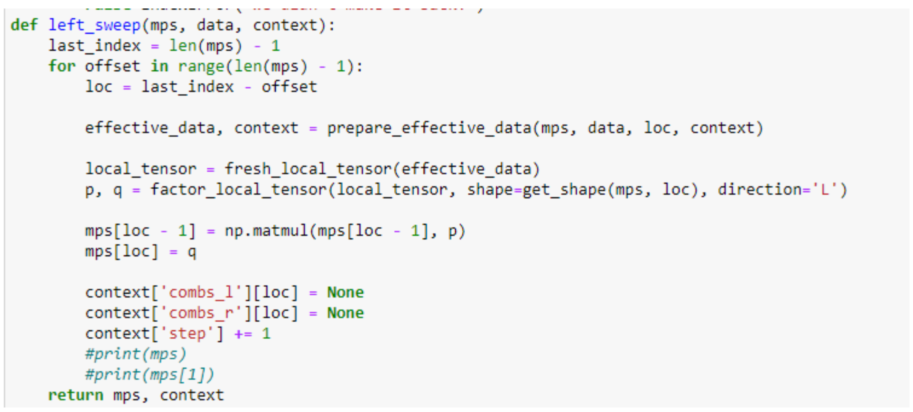
```
def zip_outer(*args):
    return [reduce(np.multiply.outer, _) for _ in zip(*args)]

def fresh_local_tensor(effective_data):
    #print(effective_data[0])
    #print(effective_data[1])
    zipped = zip_outer(*effective_data)
    #print("outer=")
    #print(zipped)
    local_sum = sum(zipped)
    #print("local_sum")
    #print(local_sum)
    #print(np.linalg.norm(local_sum))
    #print(local_sum / np.linalg.norm(local_sum))
    return local_sum / np.linalg.norm(local_sum)
# assumes mps is in right gauge to begin with
```

Figure A.11: Code used to calculate the local tensor.



```
def right_sweep(mps, data, context):
    # do all but the rightmost site
    for loc in range(len(mps) - 1):

        effective_data, context = prepare_effective_data(mps, data, loc, context)
        #print("effective_data=")
        #print(effective_data)
        local_tensor = fresh_local_tensor(effective_data)
        #print("local tensor=")
        #print(local_tensor)
        #print("aqui")
        p, q = factor_local_tensor(local_tensor, shape=get_shape(mps, loc), direction='R')

        mps[loc] = p
        #print(mps[loc])
        #print("bora")
        #print("q=")
        #print(q)
        #print("site 2 antigo=")
        #print(mps[loc+1])
        mps[loc + 1] = np.tensordot(q, mps[loc + 1], axes=[-1, 0])
        #print("site 2")
        #print(mps[loc+1])
        #print(mps)
        if context.get('combs_l'):  # not there on the first sweep
            context['combs_l'][loc] = None
        context['combs_r'][loc] = None
        context['step'] += 1

    return mps, context
```

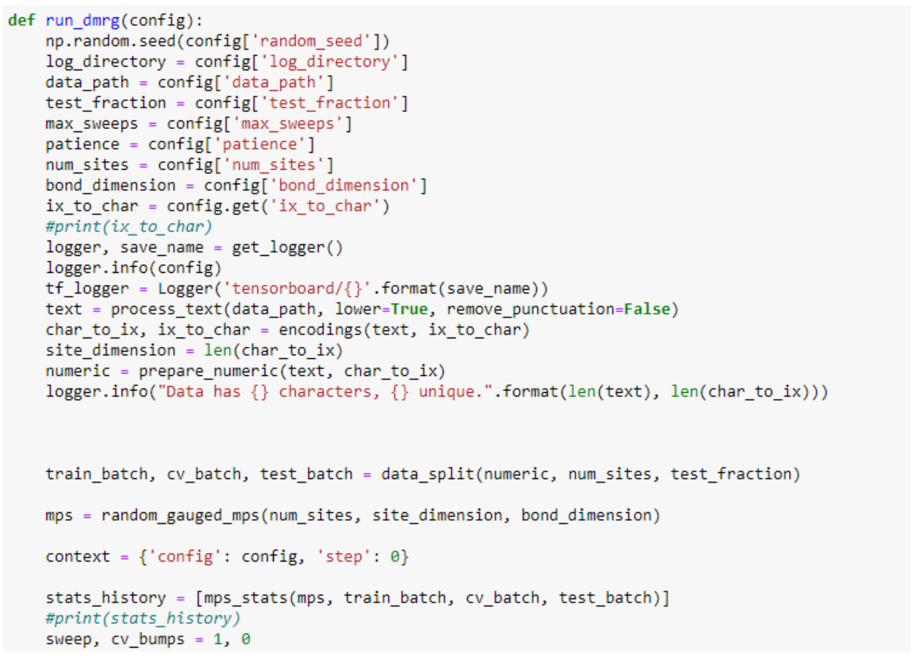
(a)



```
def left_sweep(mps, data, context):
    last_index = len(mps) - 1
    for offset in range(len(mps) - 1):
        loc = last_index - offset

        effective_data, context = prepare_effective_data(mps, data, loc, context)

        local_tensor = fresh_local_tensor(effective_data)
        p, q = factor_local_tensor(local_tensor, shape=get_shape(mps, loc), direction='L')

        mps[loc - 1] = np.matmul(mps[loc - 1], p)
        mps[loc] = q

        context['combs_l'][loc] = None
        context['combs_r'][loc] = None
        context['step'] += 1
        #print(mps)
        #print(mps[1])
    return mps, context
```

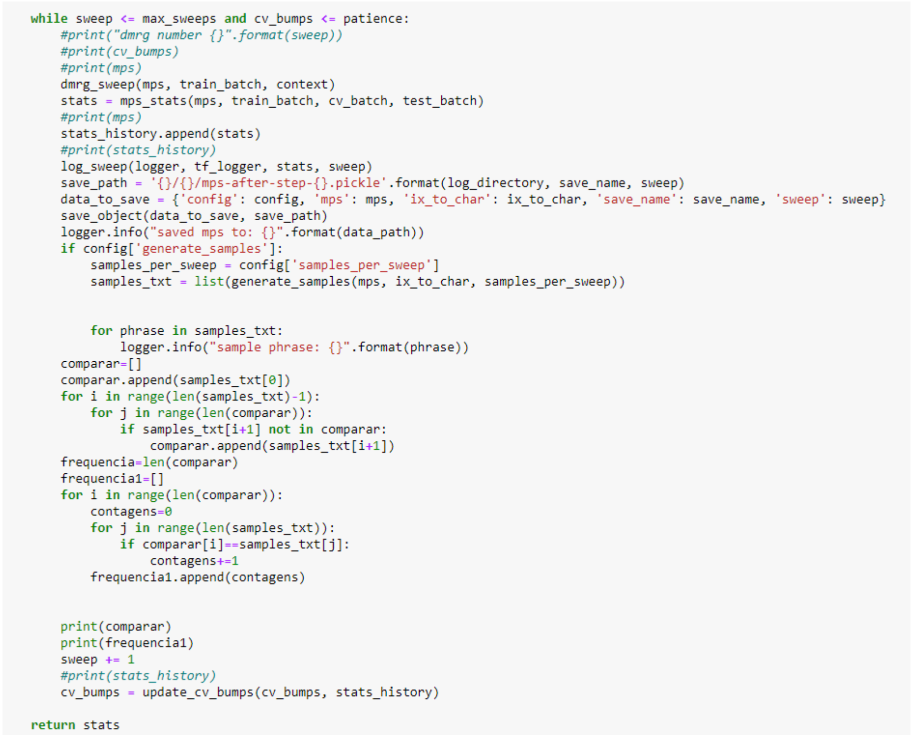
(b)

Figure A.12: (a) Performing the DMRG from the right to the left. (b) Performing the DMRG from the left to the right



```python
def run_dmrg(config):
    np.random.seed(config['random_seed'])
    log_directory = config['log_directory']
    data_path = config['data_path']
    test_fraction = config['test_fraction']
    max_sweeps = config['max_sweeps']
    patience = config['patience']
    num_sites = config['num_sites']
    bond_dimension = config['bond_dimension']
    ix_to_char = config.get('ix_to_char')
    #print(ix_to_char)
    logger, save_name = get_logger()
    logger.info(config)
    tf_logger = Logger('tensorboard/{}'.format(save_name))
    text = process_text(data_path, lower=True, remove_punctuation=False)
    char_to_ix, ix_to_char = encodings(text, ix_to_char)
    site_dimension = len(char_to_ix)
    numeric = prepare_numeric(text, char_to_ix)
    logger.info("Data has {} characters, {} unique.".format(len(text), len(char_to_ix)))



    train_batch, cv_batch, test_batch = data_split(numeric, num_sites, test_fraction)

    mps = random_gauged_mps(num_sites, site_dimension, bond_dimension)

    context = {'config': config, 'step': 0}

    stats_history = [mps_stats(mps, train_batch, cv_batch, test_batch)]
    #print(stats_history)
    sweep, cv_bumps = 1, 0
```

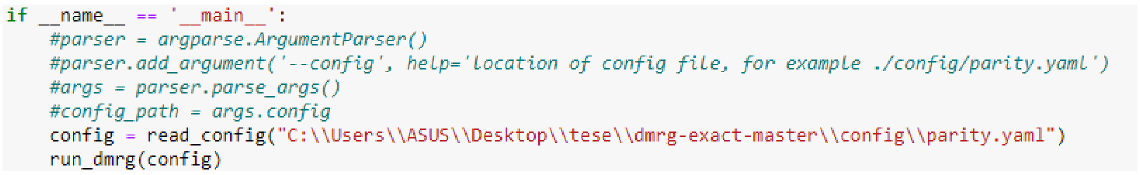
(a)



```python
    while sweep <= max_sweeps and cv_bumps <= patience:
        #print("dmrg number {}".format(sweep))
        #print(cv_bumps)
        #print(mps)
        dmrg_sweep(mps, train_batch, context)
        stats = mps_stats(mps, train_batch, cv_batch, test_batch)
        #print(mps)
        stats_history.append(stats)
        #print(stats_history)
        log_sweep(logger, tf_logger, stats, sweep)
        save_path = '{}/{}/mps-after-step-{}.pickle'.format(log_directory, save_name, sweep)
        data_to_save = {'config': config, 'mps': mps, 'ix_to_char': ix_to_char, 'save_name': save_name, 'sweep': sweep}
        save_object(data_to_save, save_path)
        logger.info("saved mps to: {}".format(data_path))
        if config['generate_samples']:
            samples_per_sweep = config['samples_per_sweep']
            samples_txt = list(generate_samples(mps, ix_to_char, samples_per_sweep))


            for phrase in samples_txt:
                logger.info("sample phrase: {}".format(phrase))
        comparar=[]
        comparar.append(samples_txt[0])
        for i in range(len(samples_txt)-1):
            for j in range(len(comparar)):
                if samples_txt[i+1] not in comparar:
                    comparar.append(samples_txt[i+1])
        frequencia=len(comparar)
        frequencia1=[]
        for i in range(len(comparar)):
            contagens=0
            for j in range(len(samples_txt)):
                if comparar[i]==samples_txt[j]:
                    contagens+=1
            frequencia1.append(contagens)


        print(comparar)
        print(frequencia1)
        sweep += 1
        #print(stats_history)
        cv_bumps = update_cv_bumps(cv_bumps, stats_history)

    return stats
```

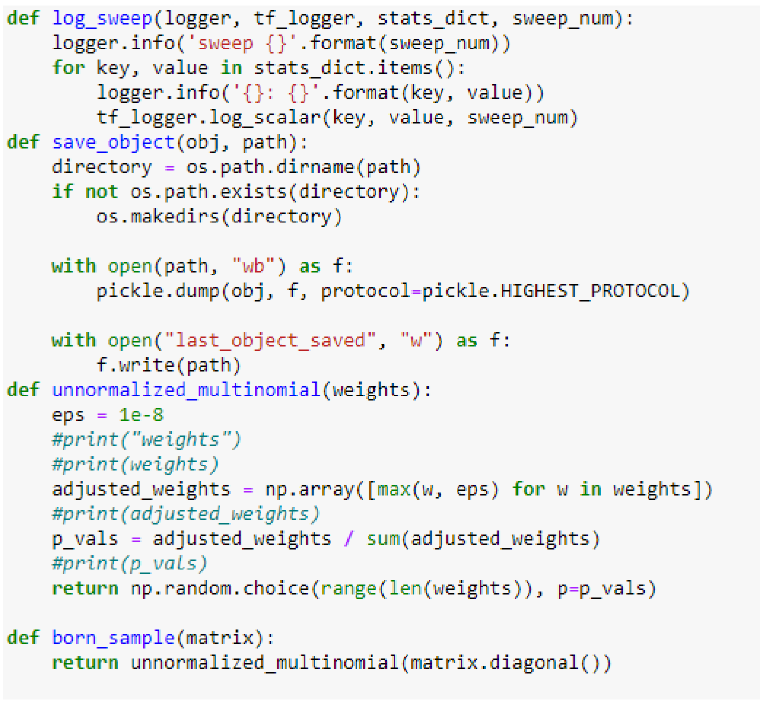
(b)

Figure A.13: (a) Code used to set the configurations. (b) Code used to run the DMRG.



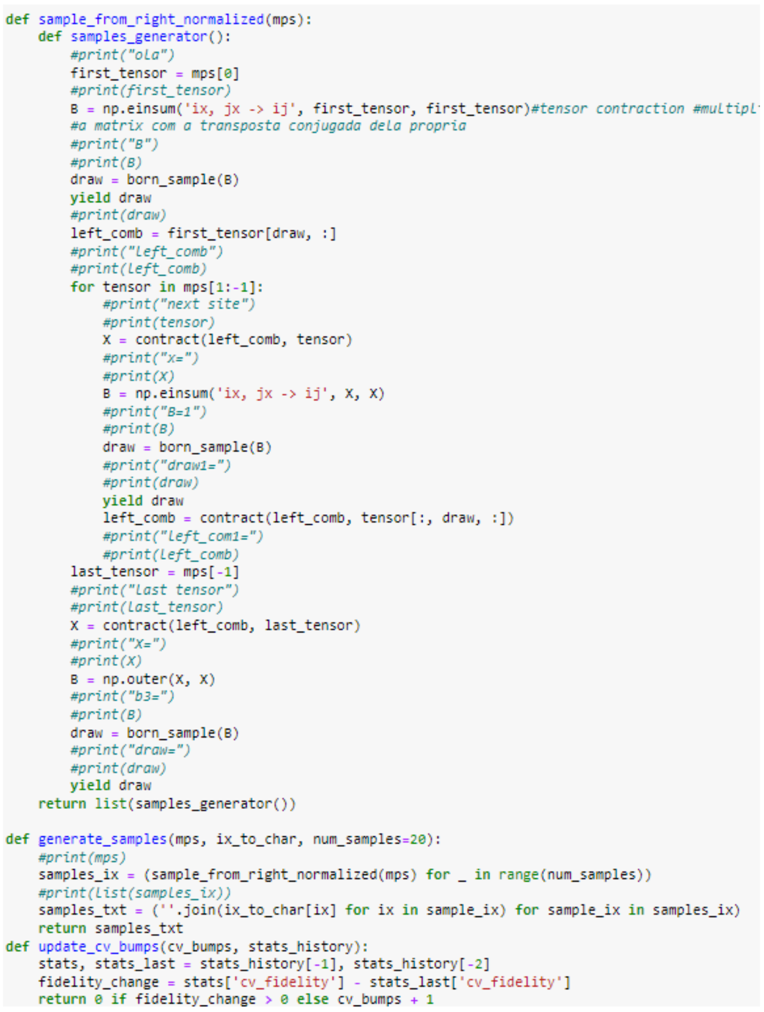
```python
if __name__ == '__main__':
    #parser = argparse.ArgumentParser()
    #parser.add_argument('--config', help='location of config file, for example ./config/parity.yaml')
    #args = parser.parse_args()
    #config_path = args.config
    config = read_config("C:\\Users\\ASUS\\Desktop\\tese\\dmrg-exact-master\\config\\parity.yaml")
    run_dmrg(config)
```

Figure A.14: Code used to set the path configuration and to start the code.



```python
def log_sweep(logger, tf_logger, stats_dict, sweep_num):
    logger.info('sweep {}'.format(sweep_num))
    for key, value in stats_dict.items():
        logger.info('{}: {}'.format(key, value))
        tf_logger.log_scalar(key, value, sweep_num)
def save_object(obj, path):
    directory = os.path.dirname(path)
    if not os.path.exists(directory):
        os.makedirs(directory)

    with open(path, "wb") as f:
        pickle.dump(obj, f, protocol=pickle.HIGHEST_PROTOCOL)

    with open("last_object_saved", "w") as f:
        f.write(path)
def unnormalized_multinomial(weights):
    eps = 1e-8
    #print("weights")
    #print(weights)
    adjusted_weights = np.array([max(w, eps) for w in weights])
    #print(adjusted_weights)
    p_vals = adjusted_weights / sum(adjusted_weights)
    #print(p_vals)
    return np.random.choice(range(len(weights)), p=p_vals)

def born_sample(matrix):
    return unnormalized_multinomial(matrix.diagonal())
```

(a)



```python
def sample_from_right_normalized(mps):
    def samples_generator():
        #print("ola")
        first_tensor = mps[0]
        #print(first_tensor)
        B = np.einsum('ix, jx -> ij', first_tensor, first_tensor)#tensor contraction #multipl
        #a matrix com a transposta conjugada dela propria
        #print("B")
        #print(B)
        draw = born_sample(B)
        yield draw
        #print(draw)
        left_comb = first_tensor[draw, :]
        #print("left_comb")
        #print(left_comb)
        for tensor in mps[1:-1]:
            #print("next site")
            #print(tensor)
            X = contract(left_comb, tensor)
            #print("x=")
            #print(X)
            B = np.einsum('ix, jx -> ij', X, X)
            #print("B=1")
            #print(B)
            draw = born_sample(B)
            #print("draw1=")
            #print(draw)
            yield draw
            left_comb = contract(left_comb, tensor[:, draw, :])
            #print("left_com1=")
            #print(left_comb)
        last_tensor = mps[-1]
        #print("last tensor")
        #print(last_tensor)
        X = contract(left_comb, last_tensor)
        #print("X=")
        #print(X)
        B = np.outer(X, X)
        #print("b3=")
        #print(B)
        draw = born_sample(B)
        #print("draw=")
        #print(draw)
        yield draw
    return list(samples_generator())

def generate_samples(mps, ix_to_char, num_samples=20):
    #print(mps)
    samples_ix = (sample_from_right_normalized(mps) for _ in range(num_samples))
    #print(list(samples_ix))
    samples_txt = (''.join(ix_to_char[ix] for ix in sample_ix) for sample_ix in samples_ix)
    return samples_txt
def update_cv_bumps(cv_bumps, stats_history):
    stats, stats_last = stats_history[-1], stats_history[-2]
    fidelity_change = stats['cv_fidelity'] - stats_last['cv_fidelity']
    return 0 if fidelity_change > 0 else cv_bumps + 1
```

(b)

Figure A.15: Code used to measure the MPS and evaluate the convergence of the MPS.