UNIVERSITY OF PADUA
PHYSICS AND ASTRONOMY DEPARTMENT

QFT simulation through MPS-TEBD

Student names: *Marco Ballarin, Francesco Manzali, Beatrice Segalini*

Course: *Quantum Information and Computing 2020* – Professor: *S. Montangero*
Due date: *February 15th, 2021*

Abstract

The Quantum Fourier Transform (QFT) is the quantum analogue of the Fourier Transform, offering an exponential speedup when compared to its classical counterpart, and a plethora of applications in quantum computing, ranging from phase estimation to order-finding problems.

Progress in the physical implementation of quantum circuits has been extremely rapid in the last few years, reaching milestones such as the first 53-qubit computer by IBM (2019) or a claim of quantum supremacy by Google (2019). Nonetheless, the huge complexity of this task means that efficient classical simulation methods are still in high demand.

In this work, we examine the MPS-TEBD approach, which makes use of a highly compressed representation of quantum states through tensor networks to approximately simulate general quantum circuits. After providing the required theoretical foundations, we first implement the QFT quantum circuit using two different `Python`-based libraries: `Cirq` and `Qiskit`. Then, we interface `Qiskit` with the tensor library `quimb` to run the simulation with MPS-TEBD. To gain a better insight on this approach, we also re-implement all its steps from scratch, by using only numerical libraries such as `numpy` and `ncon`.

The MPS-TEBD approach, as implemented with `quimb`, is able to accurately simulate the QFT on a GHZ state of 125 qubits in under 10 minutes, while a direct application to non-compressed dense states would be limited to around ~ 20 qubits before exhausting all the available RAM. We note that correctness is maintained even at high compression, provided that the initial state has a sufficiently "low entanglement", as is the case for GHZ. Moreover, our naïve implementation of MPS-TEBD, while informative, is not able to achieve the scaling of the optimized methods from `quimb`.

The code developed in this project is available on github with its extensive documentation.

Contents

Notation

In this work, we use the following notation for variables:

- A lowercase letter (e.g. c) denotes a scalar.
- A lowercase bold letter (e.g. $\boldsymbol{c}$) denotes a vector.
- An uppercase upright letter (e.g. C) denotes a matrix.
- An uppercase bold letter (e.g. $\mathbf{C}$) denotes a tensor of order > 3.

In this way, the dimensions of a variable can be understood even in the absence of explicit indices.

1. Theory

In this section, we discuss all the relevant theory for this project. In sec. 1.1 we introduce the Quantum Fourier Transform, and derive its quantum circuit representation. Then, in sec. 1.2, we discuss Matrix Product States and the TEBD approach for simulating quantum circuits, and give a brief overview on tensor networks.

1.1. Quantum Fourier Transform

The Quantum Fourier Transform (QFT) is the analogue of the inverse discrete Fourier transform for qubits.

Given a sequence of N complex terms $\{f_k\}_{k=0,\dots,N-1}$, with $f_k \in \mathbb{C}$, the **inverse Discrete Fourier Transform** is a linear transformation $\mathcal{F}^{-1}\colon \mathbb{C}^N \to \mathbb{C}^N$ mapping each $f_k \mapsto \tilde{f}_k \in \mathbb{C}$ to:

$$\tilde{f}_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} f_j \exp\left(\frac{2\pi i}{N} j\, k\right). \tag{1}$$

The quantum analogue, i.e. the **Quantum Fourier Transform** (QFT), is a linear transformation on n qubits, which acts on vectors of the computational basis $\{|j\rangle\}_{j=0,\dots,N-1}$, with $N = 2^n$, according to:

$$\mathrm{QFT}(|j\rangle) = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \exp\left(\frac{2\pi i}{N} j\, k\right) |k\rangle \quad \forall\, |j\rangle \in \mathbb{C}^N. \tag{2}$$

The QFT can be written as a sequence (circuit) of quantum gates, which can be implemented in a quantum computer.

To do so, we start by representing $k \in \{0, \dots, N-1\}$ as a binary number:

$$k = k_{n-1}2^{n-1} + \cdots + k_0 2^0 = \sum_{t=0}^{n-1} k_t 2^t,$$

We consider k_i as the $i-$th digit, with k_0 being the least significant one (i.e., the Least Significant Bit, LSB), while the Most Significant Bit (MSB) is k_{n-1}. This allows us to rewrite the elements of the computational basis as the tensor product of the local bases at each qubit:

$$|k\rangle = \bigotimes_{l=1}^{n} |k_{n-l}\rangle \equiv |k_{n-1} \cdots k_0\rangle \tag{3}$$

With this notation, (2) becomes a nested sequence of summations:

$$\mathrm{QFT}(|j\rangle) = \frac{1}{\sqrt{N}} \sum_{k_{n-1}=0}^{1} \cdots \sum_{k_0=0}^{1} \exp\left(\frac{2\pi i j}{2^n} \sum_{t=0}^{n-1} 2^t k_t\right) |k_{n-1} \cdots k_0\rangle\,. \tag{4}$$

If we perform the change of variables $l = n - t$, the highlighted summation can be written as:

$$\sum_{t=0}^{n-1} \frac{2^t k_t}{2^n} = \sum_{t=0}^{n-1} \frac{k_t}{2^{n-t}} = \sum_{l=1}^{n} \frac{k_{n-l}}{2^l}.$$

Then, converting the exponential of the summation to a product of exponentials, (4) becomes:

$$\mathrm{QFT}(|j\rangle) = \frac{1}{\sqrt{N}} \sum_{k_{n-1}=0}^{1} \cdots \sum_{k_0=0}^{1} \left[\prod_{l=1}^{n} \exp\left(2\pi i j \frac{k_{n-l}}{2^l}\right)\right] |k_{n-1} \cdots k_0\rangle\,.$$

We can now use (3) to separate the qubits:

$$\begin{aligned}\mathrm{QFT}(|j\rangle) &= \frac{1}{\sqrt{N}} \sum_{k_{n-1}=0}^{1} \cdots \sum_{k_0=0}^{1} \bigotimes_{l=1}^{n} \exp\left(2\pi i j \frac{k_{n-l}}{2^l}\right) |k_{n-l}\rangle = \\ &= \frac{1}{\sqrt{N}} \bigotimes_{l=1}^{n} \left[\sum_{k_{n-l}=0}^{1} \exp\left(2\pi i j \frac{k_{n-l}}{2^l}\right) |k_{n-l}\rangle \right] \\ &= \frac{1}{\sqrt{N}} \bigotimes_{l=1}^{n} \left[|0\rangle + \exp\left(2\pi i j \frac{1}{2^l}\right) |1\rangle \right].\end{aligned}$$

Finally, we convert j in binary notation too:

$$j = j_{n-1} j_{n-2} \dots j_0 = \sum_{t=0}^{n-1} j_t 2^t,$$

and we also introduce the fractional binary notation:

$$0.j_l j_{l+1} \dots j_m = \frac{1}{2} j_l + \frac{1}{4} j_{l+1} + \cdots + \frac{1}{2^{m-l+1}} j_m.$$

In this way, the term $j/2^l$ can be rewritten as:

$$\frac{j}{2^l} = j_{n-1} j_{n-2} \dots j_{l+1}.j_l \dots j_0.$$

Thanks to the properties of the exponential, the integer and the fractional part can be factorised. Note that the integer part is unitary, since $\exp(2\pi i j_{n-1} \dots j_{l+1}) = 1$, and so can be removed.

This allows to further expand the tensor product:

$$\begin{aligned}\mathrm{QFT}(|j\rangle) = \frac{1}{\sqrt{N}} \Big[|0\rangle + \exp(2\pi i 0.j_0) |1\rangle \Big]_{n-1} \otimes \Big[|0\rangle + \exp(2\pi i 0.j_1 j_0) |1\rangle \Big]_{n-2} \otimes \cdots \otimes \\ \otimes \Big[|0\rangle + \exp(2\pi i 0.j_{n-1} j_{n-2} \dots j_0) |1\rangle \Big]_0. \qquad (5)\end{aligned}$$

This final expression can be used now to express the QFT operation via elementary gates, leading to the quantum circuit implementation of the algorithm.

We start by noting that the *last* qubit after the transformation (the $[\dots]_{n-1}$ term) only depends on the first one ($|j_0\rangle$), and can be derived by applying a Hadamard gate:

$$\mathrm{QFT}(|j\rangle)_{n-1} \equiv \left|\tilde{j}\right\rangle_{n-1} = \frac{1}{\sqrt{2}} \Big[|0\rangle + \exp(2\pi i 0.j_0) |1\rangle \Big] = \mathrm{H} |j_0\rangle .$$

In fact, there are only two possible cases, since j_0 can be either 1 or 0:

$$\begin{aligned} j_0 = 0: &\qquad \left|\tilde{j}\right\rangle_{n-1} = \frac{1}{\sqrt{2}} \Big[|0\rangle + |1\rangle \Big] = \mathrm{H} |0\rangle \\ j_0 = 1: &\qquad \left|\tilde{j}\right\rangle_{n-1} = \frac{1}{\sqrt{2}} \Big[|0\rangle + \exp(\pi i) |1\rangle \Big] = \frac{1}{\sqrt{2}} \Big[|0\rangle - |1\rangle \Big] = \mathrm{H} |1\rangle \end{aligned}$$

The qubit immediately before ($n-2$-th) is indeed more complex to derive, especially due to its phase. The $0.j_1$ term can be computed, similarly to before, via a Hadamard gate applied to $|j_1\rangle$, however the $0.0j_0$ expression requires a *controlled phase* (CPHASE gate) dependent on $|k_0\rangle$.

We define a k-order CPHASE gate as follows:

$$R_k = \begin{pmatrix} \mathbb{I}_2 & \mathbb{O} \\ \mathbb{O} & \mathrm{R}_z(2\pi i/2^k) \end{pmatrix}.$$

Thus:

$$\left|\tilde{j}\right\rangle_{n-2} = \mathrm{R}_2^{|j_0\rangle}\mathrm{H}\left|j_1\right\rangle.$$

The same argument can be repeated for all the other qubits. For $m \geq 1$, we get:

$$\left|\tilde{j}\right\rangle_m = \mathrm{R}_{m+1}^{|j_0\rangle}\cdots\mathrm{R}_3^{|j_{m-2}\rangle}\mathrm{R}_2^{|j_{m-1}\rangle}\mathrm{H}\left|j_m\right\rangle = \left(\prod_{l=0}^{m-1}\mathrm{R}_{m+1-l}^{|j_l\rangle}\right)\mathrm{H}\left|j_m\right\rangle.$$

Summarising all the previous deductions and computations, one can obtain the final quantum circuit, displayed in Figure (1). It is worth observing that a SWAP operation of order $O(N)$ — or at least a *renaming* of qubits — must be implemented to maintain the original qubit order, since it is inverted by the QFT.

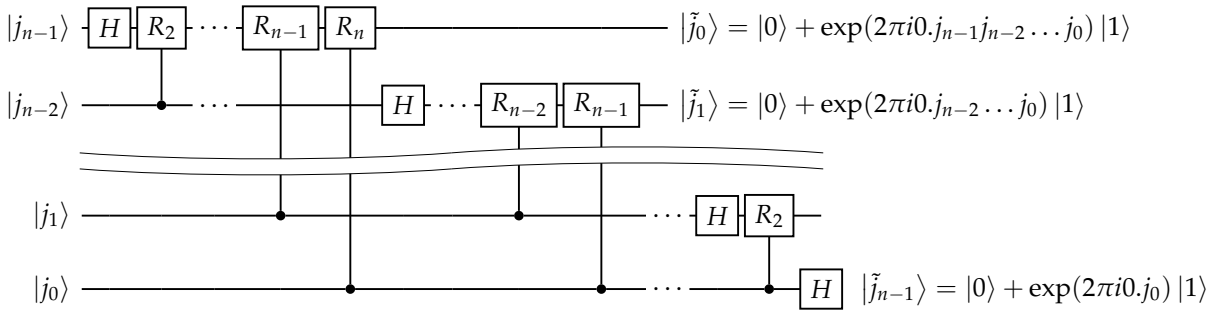

Figure (1) – Circuit implementation for the QFT.

In conclusion, considering the number of quantum gates adopted in this circuit (n^2), one can at first infer that the order of the algorithm is $O(n^2)$, which compared to its classical counterpart (Fast Fourier Transform order: $O(nN)$) is exponentially more efficient.

1.2. Matrix Product States

The Matrix Product States (MPS) are an efficient way of representing a quantum state of 1-dimensional systems. We motivate their use in sec. 1.2.1, where we also present their definition, along with an intuitive way to physically construct them. Then, sec. 1.2.2 introduces the graphical notation of tensor networks, of which MPS are a specific case. In sec. 1.2.3 we discuss the choice of gauge for a tensor network, which can be used to simplify computations, and improve the stability of algorithms. Then, sec. 1.2.4 contains a procedure to convert states from a dense representation to MPS. Finally, sec. 1.2.5 introduces the Time Evolution Block Decimation (TEBD) algorithm, which will be used to apply the QFT to MPS.

1.2.1. Motivation and Construction

A generic wave function of a many-body quantum system can be expressed by listing its coefficients in a chosen base:

$$\left|\psi\right\rangle = \sum_{k=1}^{N} \boldsymbol{c}_k \left|k\right\rangle \tag{6}$$

The number N of needed coefficients scales exponentially with the system's size n. For example, if we have n degrees of freedom (sites) with local dimension d, the most general wave function has d^n coefficients:

$$\left|\psi\right\rangle = \sum_{s_1,\dots,s_n=1}^{d} \mathbf{C}_{s_1\dots s_n}\left|s_1 s_2\dots s_n\right\rangle$$

However, we are usually interested in some specific states, e.g. the low energy ones. Their coefficients are not completely random, and so we may seek a more compressed representation.

In fact, we can argue that (6) is an extremely inefficient way to define a state, because most of the states we are interested in belong to a tiny subset of the whole system's Hilbert space.

A first intuition comes from the fact that most Hamiltonians are *local*, i.e. only sites that are *close* to each other interact significantly. For instance, consider a 1D spin chain with a finite correlation length ξ [18, p. 9]. Two sites A and B which lie at a distance $l_{AB} \gg \xi$ are effectively independent, and their state ψ_{AB} can be well approximated by a product state, thus requiring fewer coefficients to be fully specified.

Moreover, most of the Hilbert space cannot be quickly reached by time-evolution under a *local* Hamiltonian [12, sec. 3.4]. So, every state that can be "reasonably" prepared (either in an experiment or by Nature) belongs to a tiny corner of the whole space of possible states.

This means that specifying a $|\psi\rangle$ by listing d^N coefficients is highly inefficient: it would be better to have some representation that is "specialized" to the corner of the Hilbert space we are most interested in. As we will now see, MPS offer one such representation.

Consider an n-body system, with local dimension d and open boundary conditions. A pure state $|\psi\rangle$ can be written as a Matrix Product State as follows [4]:

$$|\psi\rangle = \sum_{s_1,\dots,s_n=1}^{d} \sum_{\alpha_1,\dots,\alpha_n=1}^{\chi} \mathbf{M}^{[1],s_1}_{1\alpha_1} \mathbf{M}^{[2],s_2}_{\alpha_2\alpha_3} \cdots \mathbf{M}^{[n-1],s_{n-1}}_{\alpha_{n-2}\alpha_{n-1}} \mathbf{M}^{[n],s_n}_{\alpha_{n-1}1} |s_1 s_2 \dots s_n\rangle \tag{7}$$

The core idea is that each tensor $\mathbf{M}^{[i],s_i}_{\alpha_i\alpha_{i+1}}$ is a local description for the $[i]$-th site, which will allow to apply a *local* operator to a certain site without the need to change all the other coefficients.

For a fixed s_i, $\mathbf{M}^{[i],s_i}_{\alpha_i\alpha_{i+1}}$ is a $\chi \times \chi$ complex matrix, meaning that (7) is the sum of basis elements weighted by matrix products — which is why it is called a Matrix Product State. χ is called the MPS **bond dimension**, and a sufficiently high χ is needed if we want to express a truly *general* $|\psi\rangle$ in such form. However, the idea is that MPS with a *lower* χ can still encode all the "interesting" states, albeit clearly not *all* possible states.

More precisely, MPS are suitable to describe states with "low entanglement". Usually, various many-body ground states have "low entanglement", and this statement can be made rigorous [5] for 1-dimensional systems that have a gap between the ground state and the first excited state.

First, let us specify what we mean exactly with entanglement, and in what sense ground states usually have "low entanglement". Then we will show how (7) can be derived from the description of a state constructed with a "limited amount of entanglement", meaning that it is a "specialized" representation for the states we are most interested in.

We consider the so-called bipartite entanglement for a system divided into two parts (A, B), each with A (B) sites of local dimension d. By introducing bases $\{|\phi_i^A\rangle\}$ and $\{|\phi_i^B\rangle\}$ for the two subsystems, which are respectively of dimension d^A and d^B, we can write any state of the whole system in a product basis as follows:

$$|\psi\rangle = \sum_{ij} \alpha_{ij} \left|\phi_i^A\right\rangle \left|\phi_i^B\right\rangle \tag{8}$$

However, using the Schmidt decomposition [10], we can turn the double sum into a single one:

$$|\psi\rangle = \sum_{\alpha}^{r} \lambda_\alpha \left|\phi_\alpha^A\right\rangle \left|\phi_\alpha^B\right\rangle \tag{9}$$

where $r \in \left[1, \min(d^A, d^B)\right]$ is the Schmidt rank. The Schmidt rank is equal to 1 only for a product state, which by definition is not entangled, whereas a Schmidt rank $r > 1$ indicates non-zero entanglement between the two parts. From the Schmidt coefficients $\{\lambda_\alpha\}$, which are real, non-negative,

unique (for a given state) and satisfy $\sum_\alpha \lambda_\alpha^2 = 1$, we can obtain the entanglement entropy. This is simply the Von Neumann entropy evaluated on a subsystem, i.e.:

$$S = -\sum_\alpha \lambda_\alpha \ln \lambda_\alpha \tag{10}$$

Consider now the entanglement entropy as a function of the size N of the left half of the bipartition. It is possible to prove that $S(N)$ obeys an area law for ground states of gapped 1-dimensional systems[9], according to which $S(N)$ is proportional to the size of the boundary that is left after the bipartition.

In 1-dimensional systems, the boundary consists of 2 points and hence the entanglement entropy does not scale with N[1]. Thus, such states are said to have "low entanglement".

Now, we will show a way to construct states with a controllable amount of entanglement, which will in turn naturally lead to the MPS representation (7) [4, sec. 2.2.5].

Consider a system of n sites, each with local dimension d. We want to construct a state for this system such that it has a "limited" amount of entanglement. One way to do so is offered by the Valence Bond Picture [16, sec. 2.2] [15, sec. 1.2.1].

First, we associate to each site a pair of χ-dimensional auxiliary sites, each representing one bond (Figure (2)). In this way, we can independently set the entanglement contained in each bond. This is done by preparing any two auxiliary sites i and $i+1$, which correspond to the same bond (e.g. auxiliary sites 2 and 3 in Figure (2)), in a maximally entangled state:

$$|\omega_\chi\rangle = \frac{1}{\sqrt{\chi}} \sum_{k=1}^{\chi} |k\rangle_i \, |k\rangle_{i+1}$$

In fact, the Von Neumann entropy for $|\omega_\chi\rangle$ is maximal, and equal to $S = \ln \chi$. The number χ is called the **bond dimension**, and fixes the amount of entanglement present in each bond.

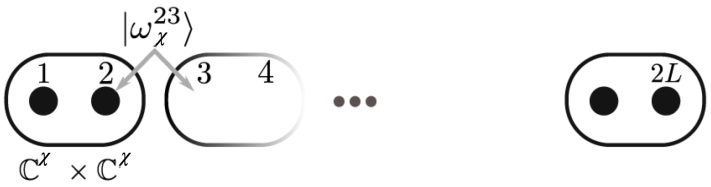

Figure (2) – Construction of maximally entangled bonds $|\omega_\chi\rangle$ between physical sites. Each site consists of two auxiliary sites (with states belonging to $\mathbb{C}^\chi \times \mathbb{C}^\chi$), so that a chain with L sites consists of $2L$ auxiliary sites. Figure taken from [11].

If we assume open boundary conditions, the first and last auxiliary sites do not participate in any bond, and they are respectively set to $|\alpha\rangle$ and $|\beta\rangle$ (boundary states).

Now, we project the states of each pair of auxiliary sites to states of a single physical site (Figure (3)). This is done by applying to each pair $[i]$ a local linear map $\mathbb{C}^\chi \times \mathbb{C}^\chi \to \mathbb{C}^d$ given by:

$$\mathbf{P}^{[i]} = \sum_{s=1}^{d} \sum_{\alpha,\beta=1}^{\chi} \mathbf{M}_{\alpha\beta}^{[i],s} \, |s\rangle^{[i]} \, \langle\alpha\beta|_{\text{aux}}^{[i]} \tag{11}$$

In other words, each element $|\alpha\beta\rangle_{i,\text{aux}}$ of the basis of the i-th pair of auxiliary sites is mapped to a physical state given by:

$$|\alpha\beta\rangle_{\text{aux}}^{[i]} \mapsto \sum_{s=1}^{d} \mathbf{M}_{\alpha\beta}^{[i],s} \, |s\rangle^{[i]}$$

[1] Near the boundary of a system there may be a dependence on N due to boundary effects.

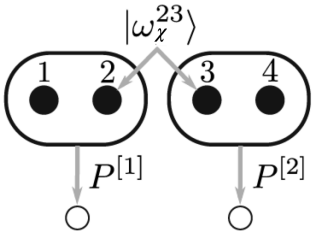

Figure (3) – We construct maps $\mathbf{P}^{[i]} : \mathbb{C}^\chi \times \mathbb{C}^\chi \to \mathbb{C}^d$, mapping states on the auxiliary sites to states on the physical sites. These maps do not increase the system's entanglement. Figure taken from [11].

Consider, for example, a chain of 2 physical sites, and thus 2 pairs of auxiliary sites and exactly one bond between them. The initial state of the extended system is:

$$|\tilde{\psi}\rangle = |\alpha\rangle\,|\omega_\chi\rangle\,|\beta\rangle = \frac{1}{\sqrt{\chi}}\,|\alpha\rangle_1^{[1]} \sum_{k=1}^{\chi} |k\rangle_2^{[1]}\,|k\rangle_3^{[2]}\,|\beta\rangle_4^{[2]}$$

By applying (11) to both sites (and ignoring the normalisation for brevity) we get:

$$\begin{aligned} |\psi\rangle &= (\mathbf{P}^{[1]} \otimes \mathbf{P}^{[2]})\Big(\sum_{k=1}^{\chi} |\alpha k\rangle^{[1]}\,|k\beta\rangle^{[2]}\Big) = \\ &= \sum_{s_1=1}^{d}\sum_{s_2=1}^{d} \mathbf{M}^{[1],s_1}_{\alpha k}\mathbf{M}^{[2],s_2}_{k\beta}\,|s_1 s_2\rangle \end{aligned}$$

which is the MPS representation for a 2-body system, with bond dimension χ. The same reasoning can be extended to n sites (with $n-1$ bonds), which leads back to (7).

Note that applying $\mathbf{P}^{[i]}$ to each physical site $[i]$ is a LOCC transformation [16, sec. 2.2], i.e. it can be executed by means of only Local Operations and Classical Communication. Thus, it cannot add more "quantum entanglement" between different sites. More precisely, the entanglement entropy of $|\psi\rangle$ is bounded by that of $|\tilde{\psi}\rangle$, which is fixed by χ. Consider a bipartition splitting sites i and $i+1$, and let ρ be the left (or right) reduced density matrix of $|\psi\rangle$. Then:

$$\mathcal{S}_{\mathrm{VN}}(\rho) \leq \log\chi$$

So, we can conclude that any MPS with bond dimension χ has a "low entanglement", i.e. a bipartition entanglement not greater than $\log\chi$ along any bipartition.

Therefore, since an MPS can encode *any*[2] state (see [sec. 2.3][16] for a proof), we can say that MPS are a good representation for any "low entanglement" state.

In fact, the MPS representation is particularly useful when χ is small. In particular, we can truncate the bond dimension χ to reduce the computational size of a quantum state, while still retaining most (if not all) of the information.

For example, a chain of n qubits ($d = 2$) in a GHZ state can be exactly encoded by an MPS with bond dimension $\chi = 2$. Explicitly, it is realised by taking, in the above construction, $|\omega_2\rangle = |00\rangle + |11\rangle$ and the mapping $\mathbf{P} = |0\rangle\langle 00|_{\mathrm{aux}} + |1\rangle\langle 11|_{\mathrm{aux}}$ at each site (we ignore the normalisation for brevity). The MPS tensors $\mathbf{M}^{[i]}_{\mathrm{ghz}}$ are all equal to:

$$\mathrm{M}^{[i],0}_{\mathrm{ghz}} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \qquad \mathrm{M}^{[i],1}_{\mathrm{ghz}} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

[2]Note, however, that the MPS decomposition for a particular state is not unique. So, different MPS may not correspond to different states.

The whole tensor $\mathbf{M}$ can be written "all at once" as a vector-valued matrix:

$$\mathbf{M}^{[i]}_{\text{ghz}} = \begin{pmatrix} |0\rangle & \mathbf{0} \\ \mathbf{0} & |1\rangle \end{pmatrix}.$$

The main advantage of this representation comes from the fact that the number of coefficients in an MPS scales as $O(nd\chi^2)$, i.e. linearly with n for a fixed χ, while a dense representation needs $O(d^n)$ coefficients, i.e. a number of coefficients that is exponential in n.

For instance, to represent an n-qubit GHZ state as a dense vector we would need 2^n coefficients, but only $8n$ if the MPS representation is used.

Thus, MPS offer a way to accurately represent low entanglement states (i.e. all the "interesting" ones) which is extremely compressed if compared to the usual dense representation (6).

1.2.2. Tensor networks

Notation such as the one used in (7) can be particularly heavy. For this reason we make use of graphical representations, first introduced by R. Penrose [14].

Consider one of the terms appearing in (7), i.e. $\mathbf{M}^{[i],s_i}_{\alpha\beta}$. This is an object with 3 indices, that is an order-3 tensor. We can graphically represent it as a coloured rectangle with 3 "legs", each representing a different index (Figure (4)).

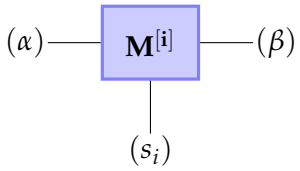

Figure (4) – MPS diagram for the $[i]$-th site MPS tensor $\mathbf{M}^{[i],s_i}_{\alpha\beta}$. The number of "legs" is the number of indices of the tensor, which is 3 for a single $\mathbf{M}$-tensor. The downward pointing leg is conventionally chosen to be the one representing the "physical" index, while the other two represent the "virtual" indices that are contracted in the MPS representation.

Contraction between tensors are represented by joining with a line the two indices being contracted. Our convention for representing Matrix Product States is shown in Figure (5).

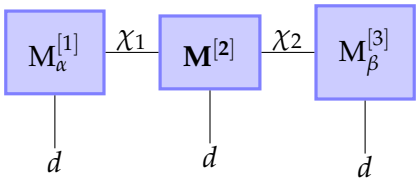

Figure (5) – MPS representation for a 3-body system, encoding the following tensor contraction: $\sum_{\gamma=1}^{\chi_1}\sum_{\delta=1}^{\chi_2}\mathbf{M}^{[1],s_1}_{\alpha\gamma}\mathbf{M}^{[2],s_2}_{\gamma\delta}\mathbf{M}^{[3],s_3}_{\delta\beta}$. Note that the boundary conditions $|\alpha\rangle$ and $|\beta\rangle$ fix the very first and last indices of the $\mathbf{M}$ tensor chain, reducing the first and last $\mathbf{M}$ tensor to just matrices $\mathrm{M}^{[1]}_{\alpha}$ and $\mathrm{M}^{[3]}_{\beta}$ (i.e. objects with 2 indices). Moreover, in general the bond dimensions (χ_1 and χ_2) may be different for each bond, and they are denoted above the links between the contracted indices. All the physical sites have dimension d, which is shown below the downward links, representing the physical indices.

1.2.3. Gauge Freedom

A fundamental operation to do on the Matrix Product States is the application of operators. If we want to measure the expectation value of a local operator on a system of n sites, we would have to perform the full contraction of all the n tensors (Figure (6)).

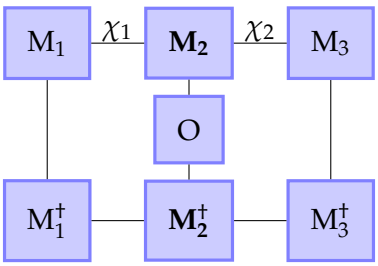

Figure (6) – Expected value of a single-site operator O on a 3-body MPS.

However, the MPS representation is not unique. In fact, we may insert in a bond any two matrices X and X^{-1} whose product equates an identity (Figure (7)). Then, each matrix is contracted with the nearest tensor, changing the numerical representation of the MPS, but not the overall contraction of the chain, i.e. the physical state it is representing. This is the so-called **gauge freedom** of tensor networks.

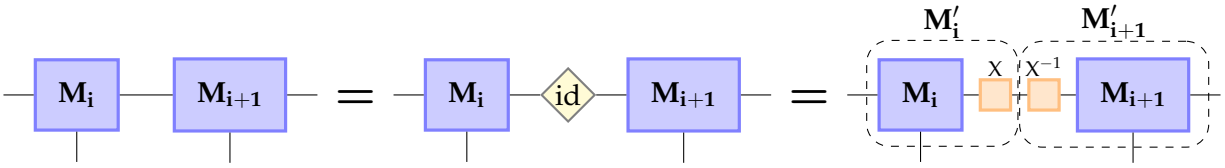

Figure (7) – Any bond can be rewritten as the contraction with an identity matrix, which can then be decomposed into the matrix product of some generic matrix X and its inverse X^{-1}. These can be in turn be contracted into the neighboring tensors. At the end, the tensor coefficients are changed, but the tensor network remains the same, in the sense that the result of any contraction with external tensors is as before.

By choosing the right kind of transformations, we can pick the particular MPS representation which is most suited to our needs. For example, consider the computation in Figure (6). To simplify the contraction, we could choose M_1 and M_2 so that $M_1 M_1^\dagger = \mathrm{id}_{\chi_1}$ and $M_2 M_2^\dagger = \mathrm{id}_{\chi_2}$. In this way, to compute the expected value we would only need to consider the tensor $\mathbf{M_2}$ on which the gate O is acting.

In general, in a (tree) tensor network, if all branches connected to a tensor **A** form an isometry between their open indices and their indices connected to **A** (as it happens for $\mathbf{M_2}$ in the above example), then **A** is said to be a **centre of orthogonality** [6, def. 3.3].

Setting a centre of orthogonality is useful also if one wants to compress **A**, for instance by reducing its dimension, or by decomposing it into smaller tensors. In fact, suppose that $\mathbf{A}'$ is some (smaller) tensor used to locally approximate **A**. Let **H** be the tensor obtained by contracting the whole original network, and similarly let $\mathbf{H}'$ be the result of contracting the whole network with $\mathbf{A}'$ in place of **A**. In our picture, **H** and $\mathbf{H}'$ would be physical states in a dense representation.

Then, if **A** is a centre of orthogonality, the local approximation error $\|\mathbf{A} - \mathbf{A}'\|$ is the same as the global approximation error on the whole network $\|\mathbf{H} - \mathbf{H}'\|$, where $\|\cdots\|$ denotes the Frobenius norm [6, Theorem 3.4]:

$$\mathbf{A} \text{ centre of orthogonality} \Rightarrow \left\|\mathbf{A} - \mathbf{A}'\right\| = \left\|\mathbf{H} - \mathbf{H}'\right\|, \quad \|\mathbf{T}\| = \sqrt{\mathrm{Tr}(\mathbf{T}^\dagger \mathbf{T})} = \sqrt{\sum_{\alpha_1, \alpha_2, \ldots, \alpha_k} |\mathbf{T}_{\alpha_1 \alpha_2 \ldots \alpha_k}|^2} \tag{12}$$

A simple way to fix a tensor **A** as the centre of orthogonality is to iterate QR decompositions in each branch connected to **A** [6, Method 1] [13, sec. 2.5] [17, sec. 4.2.2], as shown in Figure (8) for a 3-body MPS. The resulting network is said to be in the **unitary gauge**.

If by following this procedure one sets the rightmost (leftmost) site as centre of orthogonality, the MPS is said to be in left-canonical (right-canonical) form. In general, MPS are initialised in one of these two forms.

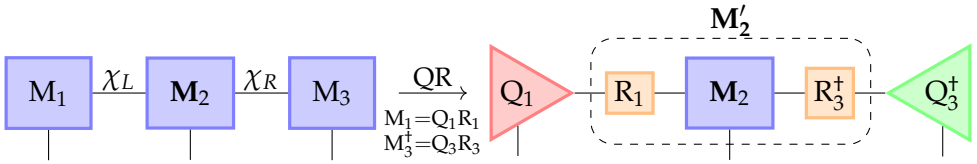

Figure (8) – Procedure for setting a site (e.g. the second) as the centre of orthogonality in an MPS (in the figure, a 3-body MPS) by repeated QR decomposition. We conventionally draw left (right)-orthogonal tensors (e.g. Q_1 and Q_3) as red (green) triangles. They are oriented such that, if they are contracted with their hermitian conjugates along the indices they are "pointing" to, they form a projector. Instead, if they are contracted along all the other indices, they form an identity. For instance, Q_1 is shown "pointing to" its second index. So $\sum_\beta (Q_1)_{i\beta}(Q_1^\dagger)_{\beta j}$ is a projector, while $\sum_\alpha (Q_1^\dagger)_{i\alpha}(Q_1)_{\alpha j} = \delta_{ij}$ is an identity. The opposite holds for Q_3, since it is "pointing to" its first index.

Note that setting a centre of orthogonality through QR decompositions does not completely fix the gauge of the network. In fact, each bond can still be modified by adding a **unitary** matrix and its inverse $UU^\dagger = \mathrm{id}$, without changing the physical state nor moving the centre of orthogonality.

1.2.4. From dense to MPS

Consider a state of an n-body system with local dimension d, written in dense representation as a set of d^n indices $\mathbf{C}_{\alpha_1 \dots \alpha_n}$. This can be interpreted as an order-n tensor, which can be rewritten as an MPS (i.e. a 1d tensor network) through repeated tensor decomposition.

To do so, we first gather all indices except the first into a unique index, effectively *reshaping* the order-n tensor into a matrix:

$$\mathbf{C}_{\alpha_1 \dots \alpha_n} \quad (\alpha_i = 1, \dots, d) \xrightarrow{\text{Reshape}} C_{\alpha_1 \beta} \quad (\alpha_1 = 1, \dots, d;\ \beta = 1, \dots, d^{n-1}) \tag{13}$$

This matrix can be now decomposed using the Singular Value Decomposition (SVD) as follows:

$$C_{\alpha_1 \beta} = \sum_{\gamma=1}^{r_1} U_{\alpha_1 \gamma} S_{\gamma\gamma} V^\dagger_{\gamma\beta} \qquad 1 \leq r_1 \leq \min(d, d^{n-1}) = d$$

Note that the first physical index α_1 appears only in the matrix $U_{\alpha_1\gamma}$, which we now rewrite as a tensor $\mathbf{M}^{[1],\alpha_1}_{1\gamma}$, following the MPS notation. Then we absorb S into V by writing $S_{\gamma\gamma}V^\dagger_{\gamma\beta} \equiv R_{\gamma\beta}$:

$$C_{\alpha_1 \beta} = \sum_{\gamma_1=1}^{r_1} \mathbf{M}^{[1],\alpha_1}_{1\gamma_1} R_{\gamma\beta}$$

By splitting the index β into the physical indices $\alpha_2 \dots \alpha_n$, we can reshape $R_{\gamma\beta}$ to an order-n tensor:

$$C_{\alpha_1 \beta} = \sum_{\gamma=1}^{r_1} \mathbf{M}^{[1],\alpha_1}_{1\gamma_1} \mathbf{R}_{\gamma_1 \alpha_2 \dots \alpha_n}$$

Now we repeat the SVD to extract the second physical index (α_2) from the $\mathbf{R}$ tensor. This is done by regrouping the indices as $\delta = (\gamma, \alpha_2)$ (size d^2) and $\epsilon = (\alpha_3, \dots, \alpha_n)$ (size d^{n-2}), i.e. reshaping $\mathbf{R}$ into a matrix $R_{\delta\epsilon}$, which can then be decomposed as follows:

$$R_{\delta\epsilon} = \sum_{\gamma_2=1}^{r_2} U_{\delta\gamma_2} S_{\gamma_2\gamma_2} V^\dagger_{\gamma_2\epsilon} \qquad 1 \leq r_2 \leq \min(d^2, d^{n-2})$$

Now we split again the $\delta = (\gamma, \alpha_2)$ index, and rename the tensor $\mathbf{U}_{\gamma_1\alpha_2\gamma_2} \equiv \mathbf{M}^{[2],\alpha_2}_{\gamma_1\gamma_2}$, following the MPS notation:

$$\mathbf{C}_{\alpha_1\alpha_2\epsilon} = \sum_{\gamma_1=1}^{r_1} \sum_{\gamma_2=1}^{r_2} \mathbf{M}^{[1],\alpha_1}_{1\gamma_1} \mathbf{M}^{[2],\alpha_2}_{\gamma_1\gamma_2} [S_{\gamma_2\gamma_2} V^\dagger_{\gamma_2\epsilon}]$$

Note that, if the system contains $n \geq 4$ sites, the number of singular values r_2 in the second decomposition can be up to d^2, which is a factor d higher than the maximum range of singular values r_1 at the previous decomposition. This means that, if no approximation is added, the bond dimension increases exponentially, up to $d^{\lfloor n/2 \rfloor}$.

Then, after grouping again S and $V^\dagger$, the whole procedure can be applied once more. By repeating it until all physical indices are split into separate tensors we arrive at the MPS representation (7).

1.2.5. TEBD

The Time Evolving Block Decimation (TEBD) algorithm is a way for efficiently computing the time evolution of a wave function in MPS representation.

The core idea is that the time evolution operator can be split through the Suzuki-Trotter decomposition into 2-sites quantum gates, which can then be **locally** applied to a Matrix Product State [2, sec. 5.2]. The MPS representation can then be recovered through an SVD decomposition, while keeping fixed the bond dimension by truncating the lowest singular values. This is shown graphically in Figures (14) and (15) (pag. 19).

In the case of the QFT on qubits, we have already either 1 or 2-qubits gates. Thus, we can follow the TEBD scheme to apply them to an MPS, as it will be shown when discussing the code.

2. Code development

We implement the QFT algorithm following two approaches:

- First, in sec. 2.1 we simulate it for states in dense representation, by using the `Python`-based quantum circuit frameworks `Qiskit` (sec. 2.1.1) and `Cirq` (sec. 2.1.2). At the start, the QFT circuit is rewritten by using only local gates, i.e. operations on neighbouring qubits, by iteratively swapping consecutive qubits. This will then allow applying the MPS-TEBD algorithm, since it requires the locality of gates.
- In sec. 2.2 we apply the QFT with the MPS-TEBD approach. We first implement it "manually" (sec. 2.2.1) to gain a better insight on the procedure. Only then we use the already available `quimb` library (sec. 2.2.2), which we link to `Qiskit`, so that it can automatically simulate any circuit with the MPS-TEBD approach, without needing to reprogram it.

2.1. Circuits

2.1.1. Qiskit

`Qiskit` [1] is a software interface developed to write quantum circuits and run experiment and simulations. It is divided in four macro-areas of interest, called "elements", but we will focus only on `Terra`, which provides a bedrock for composing quantum programs at level of circuits and pulses, optimising them for the constraints of a given device, and managing the execution of an experiment on real remote-access devices. This is because we are mainly focused on the application of fundamental gates to simulate the QFT.

With `qiskit.Terra`, we can easily define a quantum circuit using

```
qc = QuantumCircuit(q,c),
```

where `q` is the integer number of qubits and `c` the integer number of bits in the circuit. To apply a gate to the circuit we use the following syntax:

```
qc.gate(qubits),
```

where `qc` is a quantum circuit, `gate` is the gate that we want to implement, and `qubits` is the tuple of indexes of the qubits to which we want to apply the gate to. For example, if `qc` is a quantum circuit with 5 qubits to apply an Hadamard gate to the 3$^{\text{rd}}$ one we will write `qc.h(3)`.

After this introduction, we can now proceed to describe the specific implementation of the QFT with the tools provided by this `Python` package.

It is important to state that `Qiskit` allows us to apply long-range gates, i.e. to apply controlled gates to non subsequent qubits, so coding the QFT circuit as presented in Figure (1) is straightforward. However, as we have seen in the MPS theory, in the TEBD algorithm we apply only *local* operations. Thus, for further code development, it is of our interest to *localise* the gates' application. To do that, we exploit the SWAP gate, whose action is just to switch the states of two neighboring qubits. The resulting circuit is shown, for $n = 4$ qubits, in Figure (9).

To understand how the SWAPs are applied, recall that the QFT reverses the order of qubits. This reversal can be removed by iteratively swapping neighboring qubits. For instance, for $n = 4$, if we name the qubits with their index after the QFT, we get the reversed order 3210. The first-place qubit ("3") can be brought to its correct position (the rightmost one) by applying 3 SWAPs: $3210 \to 2310 \to 1230 \to 2103$. A second "pass" of SWAPs can be used to move also the "2" to its right position ($2103 \to 1203 \to 1023$), and a last SWAP brings "0" and "1" to their correct places: $1023 \to 0123$. Now, note that in the original circuit (Figure (1)), the qubit at the i-th place interacts with a CPHASE with all the qubits "to its right". For instance, "3" in 3210 interacts with "2", "1" and "0" in succession. But during the first pass of SWAPs, "3" is moved iteratively "to the right", and becomes neighbour of qubits "2", "1" and "0" in sequence. This property holds for the other qubits too, meaning that the entire circuit can be rewritten by placing a SWAP after each CPHASE, as shown in the figure. In particular, note how all CPHASEs with the same phase are applied on the same "line", and that a "pass" of SWAPs is achieved by diagonals from the top-left to the bottom-right. This wiring makes all gates local (all operations happen on consecutive qubits) and preserves the initial ordering of qubits.

Note that the SWAP gate appears exclusively after a CPHASE gate. Thus, these two gates can be combined in a new 2-qubit gate. This is done in the code by the `cphase_swap_qiskit` (Listing (1)) function.

Then, observe that the entire circuit can be decomposed into "passes", such that at pass l, a CPHASE-SWAP gate is applied at consecutive positions for the first $n - l$ qubits. This allows defining the QFT circuit **recursively** in `qft_circuit_qiskit` (Listing 2).

```
def cphase_swap_qiskit(circuit, control, target, phase):
    circuit.cp(phase, control, target)
    circuit.swap(control, target)
```

Listing (1) – Function which creates the combination of the SWAP gate with the controlled phase one for `Quiskit`.

```
def qft_circuit_qiskit(circuit, n):

    if n == 0:
        return circuit
    elif n==1:
        circuit.h(0)
        return circuit

    circuit.h(0)
    for i in range(n-1):
        cphase_swap_qiskit(circuit, i, i+1, np.pi*1/2**(i+1))

    return qft_circuit_qiskit(circuit, n-1)
```

Listing (2) – `Python` function to initialise the QFT circuit recurrently. The `h` is the Hadamard gate, while the controlled phase gate is applied combined with the SWAP one in the function `cphase_swap_qiskit`.

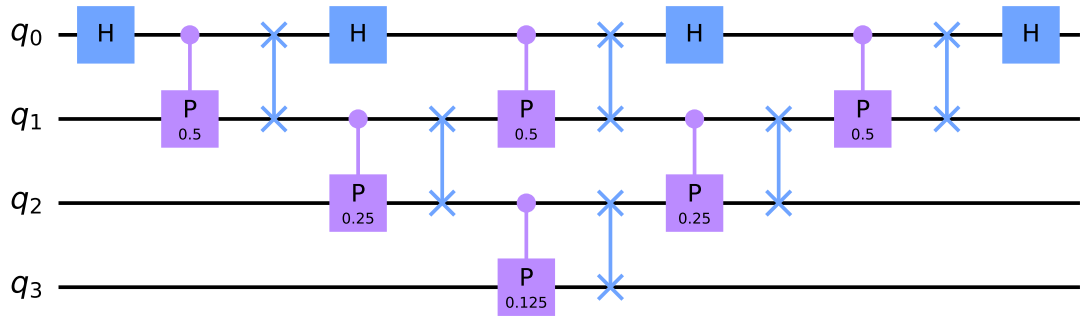

Figure (9) – Quantum Fourier Transform circuit for $n = 4$ qubits.

Finally, in order to be able to extract the metadata from a `Qiskit` quantum circuit, we coded a `Python` class able to collect all the required information, presented in Listing (3). This class is useful to automatically generate a MPS circuit from a quantum circuit, and it is the core of the interface between the tensor network implementation and the quantum circuit one that will be describe later in sec. 2.2.2.

```
class circ_data():

    def __init__(self, circ):

        self.data = [ info for info in circ.data]
        # Gates applied in order
        self.gates = [ qub[0].name for qub in self.data ]
        self.gates_params = [ qub[0].params for qub in self.data ]
        # Indices of application of the gates
        self.indeces = [ self._to_index( qub[1])  for qub in self.data  ]
        # Number of qubits in the circuit
        self.n_qub = circ.num_qubits

    def _to_index(self, x):

        return [ y.index for y in x ]
```

Listing (3) – `Python` class to extract the metadata from a `Qiskit` quantum circuit.

2.1.2. Cirq

`Cirq` [3] is a `Python` software library developed by Google AI for writing, manipulating, and optimising quantum circuits, and then running them on quantum computers and quantum simulators.

For initialising a quantum circuit with `Cirq`, the first step is to create a 1d lattice with nearest-neighbour connectivity of n qubits. This is realised with the function:

```
qubits = cirq.LineQubit.range(n)
```

which returns a list containing n qubits.

Afterwards, since we want to implement the QFT circuit with local gates as shown in sec. 2.1.1, we need to define a gate which combines the CPHASE gate with the SWAP one. This is done in Listing 4.

```
def cphase_and_swap(ctrl, target, phase):
    yield cirq.CZ(ctrl, target) ** phase
    yield cirq.SWAP(ctrl, target)
```

Listing (4) – `Python` function to build the combined action of a controlled phase gate and a swap one.

Finally, in analogy with the `Qiskit` implementation, also with `Cirq` the construction of the QFT quantum circuit is done recursively, with the function `qft_circuit_swap_cirq` shown in Listing 5.

```
def qft_circuit_swap_cirq(qubits, circuit=[]):

    n = len(qubits)
    assert n > 0, "Number of qubits must be > 0"

    if (n == 1):
        circuit.append(cirq.H(qubits[0]))
        return cirq.Circuit(circuit, strategy=cirq.InsertStrategy.EARLIEST)
    else:
        circuit.append(cirq.H(qubits[0]))
        circuit.extend(cphase_and_swap_cirq(qubits[i], qubits[i+1], 1/2**(i+1)) for i
     in range(n-1))

        return qft_circuit_swap_cirq(qubits[:n-1], circuit)
```

Listing (5) – `Python` function to build the QFT circuit recursively with `Cirq`.

2.2. MPS

2.2.1. Manual implementation

In this section, we aim to implement the MPS-TEBD algorithm "manually", i.e. without using already available high-level code. Specifically, we use just two libraries: `numpy`, for arrays and linear algebra methods, and `ncon`, for tensor contractions.

The focus is on getting a low-level understanding on the procedure, and also to provide an independent check for the results that will be obtained later on with the `quimb` library. Since the latter already contains highly optimised methods, in this section we neglect many possible enhancements, seeking to keep the code simple and understandable.

Consider a state $|\psi\rangle$ of an n-body quantum system, with each subsystem having dimension d. We can express $|\psi\rangle$ in the computational basis as follows:

$$|\psi\rangle = \sum_{\sigma_1=0}^{d} \cdots \sum_{\sigma_n=0}^{d} \mathbf{C}_{\sigma_1 \ldots \sigma_n} |\sigma_1 \ldots \sigma_n\rangle$$

The coefficients $\mathbf{C}_{\sigma_1 \ldots \sigma_n}$ are stored in an order-n tensor. For example, the GHZ state for $n = 4$ can be constructed as follows:

```
n=4
d=2
state = np.zeros(d**n)
state[0] = 1  #state = |0000>
state[-1] = 1 #state = |0000> + |1111>
state = state/np.sqrt(2) #Normalize
psi = state.reshape([d] * n) #Reshape as a tensor
```

Before applying quantum gates, we need to convert this **dense** representation into the **MPS** form. This can be done with two functions:

- `to_full_MPS` produces an MPS without loss of information, by always using sufficiently high bond dimensions, up to $d^{\lfloor n/2 \rfloor}$;
- `to_approx_MPS` results in an MPS with a maximum bond dimension constrained by a specified `chi`. If `chi` is sufficiently high, the returned MPS is exact, but in general it will be a "good" approximation of the original state.

We focus on the implementation of `to_approx_MPS`, since it is the most general.

The MPS representation is formed by the repeated SVD decomposition of the original order-n tensor `state_tensor`, following the procedure discussed in sec. 1.2.4. The full execution on a 3-body state is shown graphically in figg. 10 and 11 (pag. 17).

```
def to_approx_MPS(dense_state, n, d=2, chi=2):
    #Reshape into a tensor of order n
    state_tensor=dense_state.reshape([d]*n)
    MPS = []

    last_svd_dim = 1

```

The variable `state_tensor` always contains the rightmost tensor in the network, which at the beginning is just the initial state. `last_svd_dim` represents the dimension of the bond that connects `state_tensor` to the other tensors at its left. Since at the start there is no other tensor beside the initial state, `last_svd_dim` is set to 1.

```
    for i in range(N-1):
        #---Reshape & SVD---#
        U, S, Vh = LA.svd(state_tensor.reshape(last_svd_dim * d, -1)), full_matrices=False)
    #dim1 = last_svd_dim * d
    #dim2 = inferred from dim of state_tensor
    #S has dim (r, r), r <= min(dim1, dim2)

```

At each iteration, the rightmost tensor in the network is split in two tensors, the first consisting of its leftmost site, and the second of all its other sites.
To perform the splitting, `state_tensor` is reshaped to an `dim1` $\times$ `dim2` matrix M whose first dimension is `dim1` $=$ `last_svd_dim` $\cdot\, d$ (and `dim2` can be inferred by the initial dimensions). Then a compact SVD is performed:

$$M = USV^\dagger.$$

S is an $r \times r$ diagonal matrix containing the non-zero singular values in descending order, with $r \leq \min(\texttt{dim1}, \texttt{dim2})$, while U and $V^\dagger$ are semi-unitary matrices of shapes `dim1` $\times\, r$ and $r \times$ `dim2` respectively.

```
        #---Truncate---#
        U  = U[...,:chi]
        #shape (m,r) -> (m,min(chi,r))
        Vh = Vh[:chi, ...]
        #shape (r,n) -> (min(chi,r),n)

        if i > 0: #Tensors not at the boundaries are of order-3
            U = U.reshape(last_svd_dim, d, -1)
            #matrix -> tensor of order-3

        MPS.append(U.copy())
        #Append left site

```

Only the first χ greatest singular values in S are kept, along with the first χ columns (rows) of U ($V^\dagger$).
If U is not the leftmost tensor in the network, i.e. if this is not the first iteration, then it is reshaped to an order-3 tensor. Its first dimension is set to `last_svd_dim`, i.e. the original dimension of the left-bond of `state_tensor`, while the second dimension is d (the site dimension) and the third is inferred. Then U is appended to the `MPS` list, which will contain the full MPS representation of the initial state.

```
        #---Regroup---#
        state_tensor = (np.diag(S[:chi]) @
    Vh)

        r = len(S)
        last_svd_dim = min(r, chi)
        #Update left-bond dimension

    #Outside loop
    MPS.append(state_tensor) #Append the
    rightmost site

    return MPS

```

As the final step in each iteration, the singular values are merged with $V^\dagger$, and the result is now the rightmost tensor in the network, which is stored in `state_tensor`. Then, `last_svd_dim` is updated with the new left-bond dimension for that tensor.

At the end of the loop, the rightmost tensor consists only of the single rightmost site, and so it is included in the `MPS`.

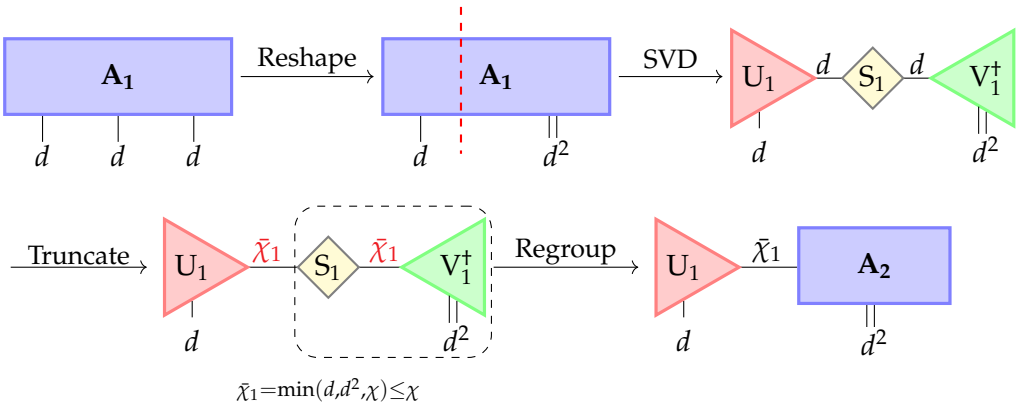

Figure (10) – First iteration of the `to_approx_MPS` function on an order-3 tensor $\mathbf{A_1}$. First, the rightmost tensor, which is $\mathbf{A_1}$, is *reshaped* to a matrix $d \times d^2$. Then a compact *SVD* is performed, resulting in a bond dimension which is at most $\min(d, d^2) = d$. The bond dimension is now *truncated* to $\bar{\chi}_1 = \min(d, d^2, \chi) :=$ `last_svd_dim`, so that it is not greater than χ. Finally, the singular values are *regrouped* into the rightmost tensor.

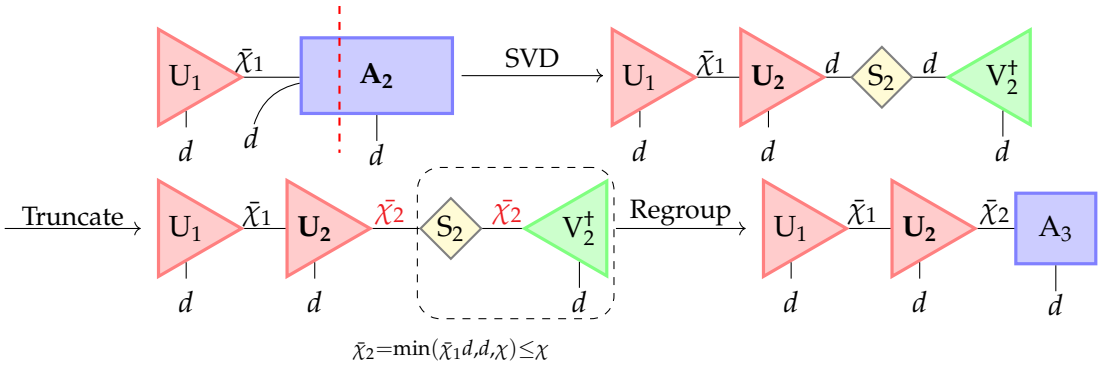

Figure (11) – Second iteration of `to_approx_MPS`. At the start, we have `last_svd_dim` $= \bar{\chi}_1$. The rightmost tensor, $\mathbf{A_2}$, is *reshaped* to a matrix $\bar{\chi}_1 d \times d$, and then a compact *SVD* is performed, leading to a bond dimension which is at most $\min(\bar{\chi}_1 d, d) = d$. This is now *truncated* to $\bar{\chi}_2 = \min(\bar{\chi}_1 d, d, \chi) :=$ `last_svd_dim`. The singular values are then *regrouped* into the rightmost tensor, and the algorithm ends.

Note that the returned MPS is, by construction, left-canonical.

An MPS can be returned to a dense representation by simply contracting all the non-spin indices. This operation is implemented by the `to_dense` function:

```
def to_dense(MPS):
    n = len(MPS)
    first_indices  = [-1, 1]
    middle_indices = [[i, -(i+1), i+1] for i in range(1,n-1)]
    last_indices   = [n-1, -n]
    connect_list = [first_indices, *middle_indices, last_indices]
    return ncon(MPS, connect_list)
```

The `ncon` function performs a tensor contraction, given a list of tensors (`MPS`) and the indices of each tensors (`connect_list`). For example, for $n = 3$ sites, `MPS` contains three tensors, respectively or order 2, 3 and 2. In this case, `connect_list` would be:

```
    connect_list = [[-1, 1], [1, -2, 2], [2, -3]]
```

Each element i of `connect_list` is a list of the indices of the i-th tensor. *Negative* indices represent free indices, while *positive* indices denote contractions. In the above example, the repeated 1 index means that `ncon` will contract the second index of the first tensor with the first index of the second tensor.

A graphical representation is shown in Figure (12).

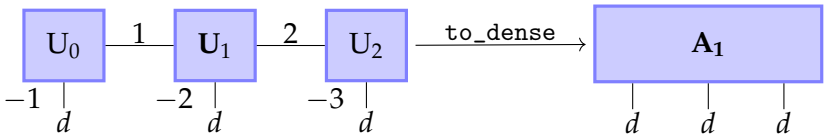

Figure (12) – Example of the contraction schema for the `to_dense` function, in the case of $N = 3$ sites. "Negative" indices (-1, -2 and -3) represent free indices, while the others are repeated indices denoting contractions. Following our convention, indices are ordered left-to-right. For example, the indices of U_1 are, in order, 1, -2 and 2.

To simulate quantum circuits, we need to apply specific gates to an MPS representations. In this work, we focus on only 1 and 2-qubits gates, which are implemented respectively in the functions `apply_one_qubit_gate` and `apply_two_qubit_gate`.

The first one just performs a local tensor contraction:

```
def apply_one_qubit_gate(gate, pos, state):
    """Applies a one qubit gate *gate* to the site at *pos* of the MPS *state*."""

    state = deepcopy(state)
    contraction_indices = [-1 , 1, -3]

    if (pos == 0): #if site is the leftmost
        contraction_indices.pop(0) #remove left free index
    if (pos == len(state)-1): #if site is the rightmost
        contraction_indices.pop(-1) #remove right free index

    #Apply gate at *pos*
    temp = ncon([state[pos], gate], [contraction_indices, [1, -2]])
    state[pos] = temp

    return state
```

The general contraction schema is shown in Figure (13) (middle). If the gate G is applied to one of the boundary sites, one of the free indices (the leftmost or rightmost) must be dropped, since these are order-2 tensors (Figure (13), left and right).

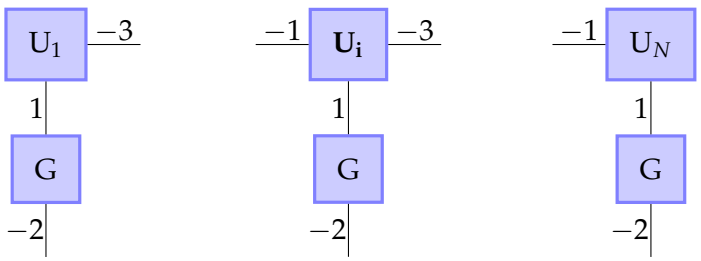

Figure (13) – Tensor contraction schemas for `apply_one_qubit_gate`. Following our convention, indices are ordered left-to-right and top-to-bottom.

On the other hand, `apply_two_qubit_gate` can be used to apply a gate to the neighbouring sites at `pos` and `pos` $+1$ as follows:

1. First, the MPS is rewritten in the unitary gauge, setting the site at `pos` to the centre of orthogonality. This is done to improve the algorithm's stability and minimise the approximation error [7] [17, sec. 4.2].

2. The 2-qubit gate is contracted with the sites at `pos` and `pos+1`.

3. The result is not an MPS anymore, since two sites are "fused" together by the tensor contraction in a single tensor **U**. Thus, SVD is used to split again **U** and recover the MPS representation. By discarding the lowest singular values, the bond dimension can be kept under a specified constraint χ, so that the MPS complexity remains constant. Remembering the properties of the centre of orthogonality (12), it is easy to notice that this procedure is indeed the *globally optimal* one. In fact, the local approximation error, which is naturally introduced by truncating the SVD and for (12) is equal to the global one, is minimised.

Much of the complexity of the implementation lies in managing the boundary sites, which are order-2 tensors and so have fewer free indices. In the following, we will focus on the general case of a gate being applied "in the middle" of an MPS, so that no site is at the boundary, and neglect the instructions needed to remove the missing free indices at the boundaries (which are nonetheless present in the actual code).

```
def apply_two_qubit_gate(gate_matrix, pos,
    state, chi=2):
    n = len(state)
    #Reshape to order-4 tensor
    gate = np.array(gate_matrix.reshape
    (2,2,2,2))

    #Set *pos* to centre of orthogonality
    if pos > 0:
        state = left_canonize(state[:pos+1])
     + state[pos+1:]
    if pos < N-1:
        state = state[:pos] + right_canonize
    (state[pos:])

    #Gate tensor contraction
    left_tens = np.array(state[pos])
    right_tens = np.array(state[pos+1])
    gate_contraction_list = [[-2,-3,1,3],
    [-1,1,2], [2,3,-4]]

    gate_contracted = ncon([gate, left_tens,
     right_tens], gate_contraction_list)

```

First, we set the site at `pos` to the centre of orthogonality. This is done by using the `left_canonize` and `right_canonize` functions, which are examined later.
Then, the two sites the gate is acting on are denoted by `left_tens` and `right_tens` respectively. They are finally contracted with the gate itself, leading to an order-4 tensor stored in `gate_contracted` (Figure (14)).

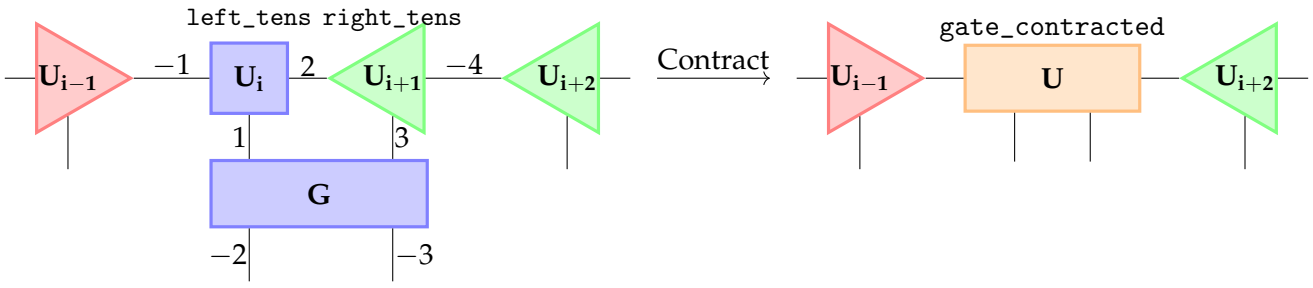

Figure (14) – At the beginning of `apply_two_qubit_gate`, the site at `pos` ($\mathbf{U_i}$) is set to the centre of orthogonality. Then the quantum gate **G** is contracted with the sites at `pos` and `pos` $+1$ ($\mathbf{U_i}$ and $\mathbf{U_{i+1}}$ in the figure).

```
    #---Reshape + SVD---#
    U, S, Vh = LA.svd(gate_contracted.
    reshape(chi_left * d, d * chi_right),
    full_matrices=False)

    #---Truncate & Reshape---#
    U  = U[...,:chi]
    Vh = Vh[:chi, ...]

    state[pos] = U.reshape(min(chi_left, chi
    ), d, -1)

    #---Regroup & Reshape---#
    right_tens = np.diag(S[:chi]) @ Vh
    state[pos+1] = right_tens.reshape(-1, d,
     min(chi_right, chi))

    return state

```

We are now ready to split `gate_contracted` into two sites using SVD, and truncate the bond dimension so that it is not greater than χ.
This procedure is similar to the one already described in `to_approx_MPS`, and it is shown graphically in Figure (15).

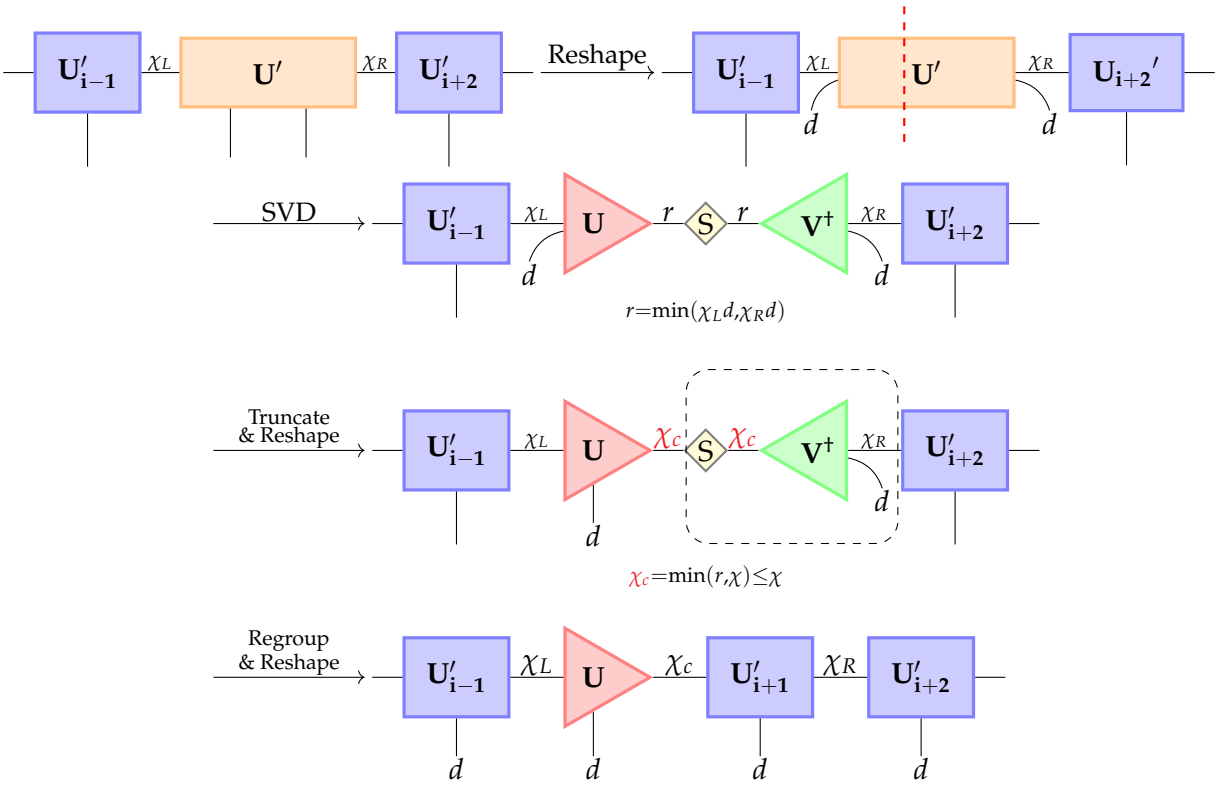

Figure (15) – Final steps of `apply_two_qubit_gate`. First, **U** (i.e. `gate_contracted`), which is now the centre of orthogonality, is reshaped to a matrix $\chi_L d \times \chi_R d$. Then a compact SVD is performed, leading to a bond dimension which is at most $r = \min(\chi_L d, \chi_R d)$. Columns (rows) of **U** (**V**) are removed to truncate the bond dimension from r to $\chi_c = \min(r, \chi) \leq \chi$. Then, the first χ_c elements of S are *regrouped* into $\mathbf{V}^\dagger$ to complete the splitting procedure. The result is a new MPS state, incorporating the action of the quantum gate.

By using the functions so defined, the QFT circuit can be implemented in a way analogous to the one used in `Qiskit` (sec. 2.1.1) or `Cirq` (sec. 2.1.2):

```
def cphase_and_swap(phase):
    #CPHASE (x) SWAP gate
    cphase = np.array(quimb.controlled('z')) ** phase
    swap = np.array(quimb.swap())

    return swap @ cphase

def qft_circuit_swap(state, n, verbosity=False):
    #Computes the QFT of a MPS @state with @N qubits.

```

```
    H = np.array(quimb.hadamard())
    for pos in range(n):
        if verbosity: print("H(0)")
        state = apply_one_qubit_gate(H, 0, state)

        for i in range(n-pos-1):
            if verbosity: print(f"CZS(({i},{i+1}), {1/2**(i+1)})")
            state = apply_two_qubit_gate(cphase_and_swap(1/2**(i+1)), i, state)
            #Apply gate to (i, i+1)

    return state

#Example
n=2
state = np.zeros(2**n)
state[0] = 1
state[-1] = 1
#state = |00> + |11>
state = state/np.sqrt(2) #Normalize

result = qft_circuit_swap(state, n)
#0.707|00> + 0.354-0.354j|01> + 0.354+0.354j|11>
```

In `apply_two_qubit_gate` we use two functions, `left_canonize` and `right_canonize`, to set the centre of orthogonality of the MPS at a specific site. They implement the iterated QR decomposition procedure shown in Figure (8).

The idea is that, for any given bond, we can make the left (right) tensor semi-orthogonal by QR decomposing it, and absorbing the R matrix into the other tensor.

This is done by the `left_compress` and `right_compress` functions. For instance, `left_compress` is as follows:

```
def left_compress(left, right):

    left_dim = left.shape[:-1]
    right_dim = right.shape[1:]

    #---Reshape---#
    left = left.reshape(np.prod(left_dim), -1)
    right = right.reshape(-1, np.prod(right_dim))

    #---QR---#
    q, r = LA.qr(left) #(left) = QR.

    #---Regroup & Reshape---#
    left = q.reshape(*left_dim, -1) #Set (left) to Q (and reshape back)
    right = (r @ right).reshape(-1, *right_dim) #Absorb R into (right)

    return (left, right)
```

The procedure is graphically shown in Figure (16).

The function `right_compress` is analogous. The only difference is that, since this time the `right` tensor is the one that needs to be orthogonal, a transposition must be added:

```
    right_dim = right.shape[1:]
    right = right.reshape(-1, np.prod(right_dim))
    q, r = LA.qr(right.T)
    q = q.T
    r = r.T
```

Mathematically, if $\texttt{right}^T = \mathrm{Q}\,\mathrm{R}$, then $\texttt{right} = \mathrm{R}^T\mathrm{Q}^T$, so that the orthogonal matrix is at the correct position (the rightmost).

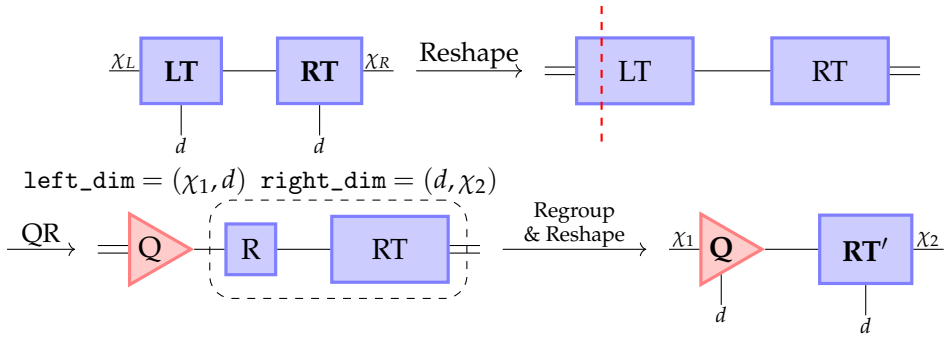

Figure (16) – Graphical representation of the change of gauge applied by the function `left_compress`, which makes the Left Tensor **LT** semi-orthogonal through a QR decomposition.

Then, to `left_canonize` an MPS we merely need to `left_compress` all of its bonds, starting from the leftmost one:

```
def left_canonize(sites):
    n = len(sites)
    sites = sites.copy()

    for i in range(n-1): #For each bond
        left  = sites[i] #starting from the first
        right = sites[i+1]

        left, right = left_compress(left, right) #left_compress it

        sites[i] = left
        sites[i+1] = right

    return sites
```

Similarly, a right-canonical form can be obtained by applying `right_compress` to all bonds, starting from the rightmost one:

```
def right_canonize(sites):
    n = len(sites)
    sites = sites.copy()

    for i in range(n-1): #For each bond
        left  = sites[n-2-i] #starting from the last
        right = sites[n-1-i]

        left, right = right_compress(left, right) #right_compress it

        sites[n-1-i] = right
        sites[n-2-i] = left

    return sites
```

2.2.2. *Quimb*

`Quimb` [8] is a fast `Python` library for quantum information and many-body calculations, including with tensor networks. It has many interesting features, but, for the purpose of this work, we focus on MPS handling and application of MPS quantum gates.

It is indeed possible to create an MPS representation from a dense one using the function

`quimb.tensor.MatrixProductState.from_dense(),`

or from a string composed only of $\{0, 1\}$ using the function

`quimb.tensor.MPS_computational_state().`

For example, to prepare the MPS corresponding to the state $|010\rangle$, one can simply type:

```
MPS = quimb.tensor.MPS_computational_state('010').
```

With `quimb`, we apply a generic quantum gate using the following structure:

```
MPS.gate_( gate(*params), qubits, tags=gate_name, max_bond=chi, contract='swap+split'),
```

where `qubits` is the tuple of the interested qubits and `contract='swap+split'` indicates that, after the application of the gate, we want to come back to the MPS structure, with maximum bond dimension `max_bond=chi`.

To automatise the creation of an MPS circuit, we implement an interface with `Qiskit`, as we aforementioned in sec. 2.1.1.

For this goal, the function presented in Listing (6) is developed: thereby, it is possible to translate a `Qiskit` circuit `qc` into a `quimb` MPS one, making use of the class introduced in Listing (3).

If the input parameter `gates` is not provided, the function retrieves a dictionary containing NOT, CNOT, CPHASE, Hadamard and SWAP quantum gates. Moreover, it is possible to set as initial state `init_state` an MPS or a string with the same syntax of `quimb.tensor.MPS_computational_state()`. If no information on the initial state is supplied, the null state $|0\dots0\rangle$ is considered.

```python
def MPS_circ(qc, gates = None, init_state = None, chi=None, verbosity=False):
    # Checks for inputs parameters
    data = circ_data(qc)
    for gate_name, qub_idx, params in zip( data.gates, data.indeces, data.
    gates_params):
        qubits = tuple(qub_idx)
        if len(qubits)==1:
            # ...
        elif len(qubits)==2:
            if len(params)==0:
                MPS.gate_( gates[ gate_name ], qubits, tags=gate_name, max_bond=chi,
    contract='swap+split')
            else:
                # Parametric gates
                MPS.gate_( gates[ gate_name ](*params), qubits, tags=gate_name,
    max_bond=chi, contract='swap+split')

    return MPS

```

Listing (6) – `Python` function to translate a `Qiskit` circuit into a `quimb` MPS one. For the sake of conciseness, the checks done on the input parameters and the application of 1-qubit gates are not reported.

3. Results

In this section, the previously described code is tested and the main outcomes are examined.

The first analysis (sec. 3.1) concerns the correctness of the results, then a study on execution times (sec. 3.2) is performed to check the code efficiency.

3.1. Correctness

To check the correctness of the code, two different initial states are chosen: the GHZ and a random one, with a fixed number of qubits $n = 7$.

Let us define the **fidelity** between two quantum pure states $|\psi\rangle$ and $|\phi\rangle$ as:

$$\mathcal{F} = |\langle\psi|\phi\rangle|^2. \tag{14}$$

This expression quantifies the *closeness* between the two states and hence it is a good metric to evaluate the performances of the algorithm. The fidelity between the "reference" state (as computed by Qiskit with a dense representation) and the one represented by the MPS with a fixed bond dimension χ is computed, and then displayed in the following plots (Figure (17)) as a function of χ. In particular, we investigate how the fidelity varies as we apply quantum gates to the MPS, i.e. along the TEBD. In particular, we focus on the application of a single CNOT gate and of the entire QFT circuit.

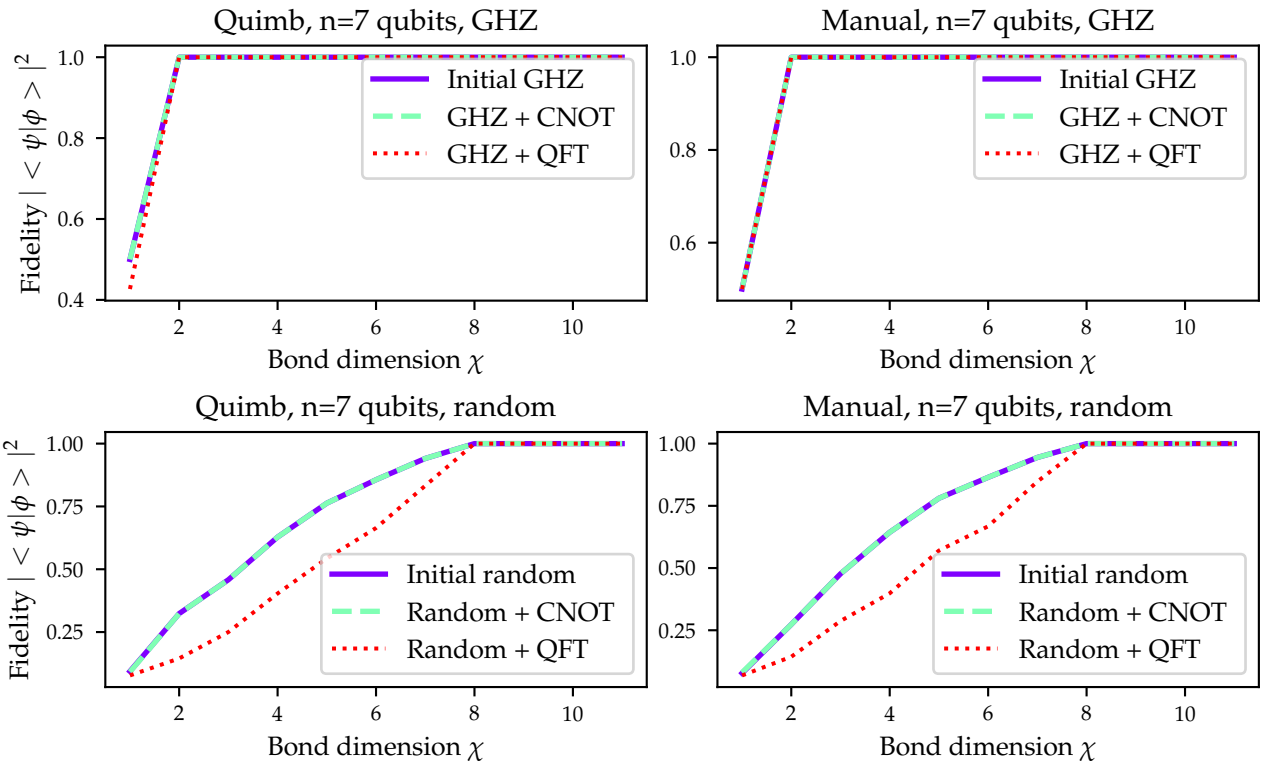

Figure (17) – Fidelity of the MPS state vs. bond dimension χ. On the left, the results of the `quimb` implementation are shown, while on the right, the manual implementation is displayed. The top row's graphs use the GHZ state, the bottom ones a random quantum state.

It can be observed that our manual implementation presents the same behaviour of the `quimb`'s one. This is exactly the expected tendency: indeed, it is sufficient to use a bond dimension of $\chi = 2$ to fully describe the GHZ state, while we need a higher bond dimension in the case of the random state, which is significantly entangled. In fact, the application of several gates (the QFT) worsen the fidelity when the bond dimension is not big enough. This trend is reverted when we reach the **maximum bond dimension** $\chi_{\max} = 2^{\lfloor n/2 \rfloor}$ (in the case analysed, $\chi_{\max} = 8$), when also the application of several gates produces consistent results — i.e., the fidelity becomes equal to 1. In fact, such a high bond dimension is sufficient to encode exactly the whole state, without any compression, as discussed in sec. 1.2.4.

3.2. Efficiency

To evaluate the QFT code's efficiency, in this section the execution time versus the size of the system (i.e., the number of qubits n composing the circuit). In the following analysis we use the GHZ as the initial state the QFT is applied to.

Firstly, it is interesting to observe the differences between the `quimb` and the "manual" implementation for two different constraints on the bond dimension: $\chi = 2$ and $\chi = 2^{\lfloor n/2 \rfloor}$ (which is sufficient for storing *any* possible state of n-qubits). As a result, the plots in Figure (18) are obtained. Note that we focus on "small" systems, with sizes $n < 16$. This is because, to better estimate the timings, we

repeat the measurements 50 times, and so considering higher n would take too long. However, we note that MPS-TEBD, as implemented by `quimb`, can manage much higher n: we tested up to $n = 125$, for which the algorithm runs for about 10 minutes on a GHZ state.

The left log-log plot in Figure (18) shows a polynomial growth of the execution times as the number of qubits increases, with exponents ~ 3.

For a fixed bond dimension χ, the MPS complexity remains constant, meaning that applying a gate to some site(s) is a $O(1)$ operation. However, before applying a gate, to reduce the approximation error, we need to set the orthogonality centre of the tensor to the gate position, which requires $O(n)$ QR decompositions at constant time (as in Figure (8)).

Moreover, the number of gates in the QFT scales as $O(n^2)$ (sec. 1.1). Thus, from theory we would expect a $O(n^3)$ complexity, which can be evaluated by a straight line fit of the execution time versus the size of the system in the log-log plot. The slope b of such line is the complexity exponent, which is the one reported in the plots of Figure (18).

It is worth noticing that in the leftmost graph the two trends have very similar slopes, meaning that the growth rate of the execution time is comparable. However, for the range of qubits considered, the manual implementation is less time consuming with respect to the `quimb` one. This is most probably due to the presence of more overhead in the `quimb` library, which is responsible for checks and optimisations.

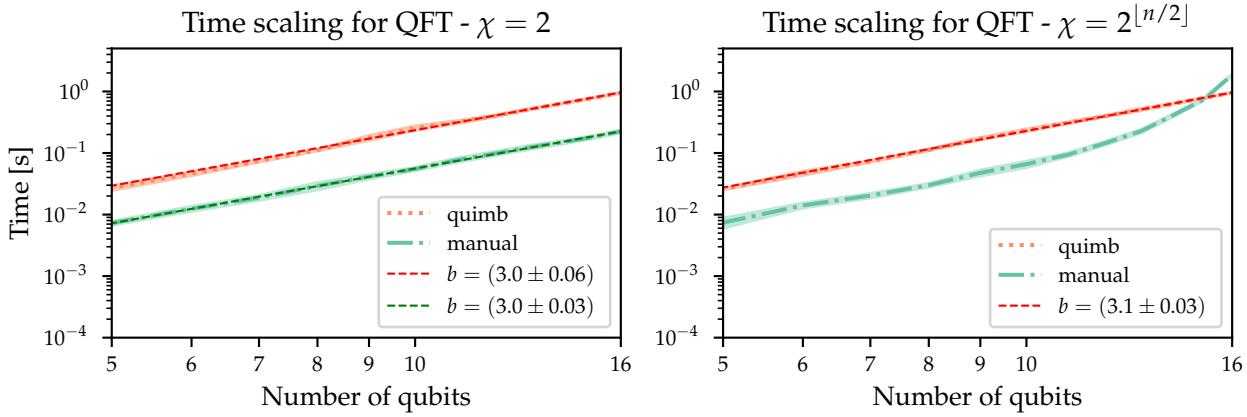

Figure (18) – Log-log plot of execution time vs. number of qubits, for the application to a GHZ state of the QFT circuit, implemented with both a fixed bond dimension $\chi = 2$ (left) and with the maximum bond dimension $\chi = 2^{\lfloor n/2 \rfloor}$ (right) for both the manual and the `quimb` code. The shaded area represents values within 1 standard deviation from the average time, computed along 50 iterations. In the legend, the order of complexity b obtained by fitting the data is reported with its error.

The most striking feature of Figure (18) is, nevertheless, the rightmost graph: in fact, we can observe that, when we consider the maximum bond dimension, the manual implementation's execution time explodes for $n > 10$, leading to a very steep increase in the graph, while the `quimb` method scales similarly as before. This is because `quimb` actively minimises the bond dimensions, while the "manual" algorithm merely truncates any bond with a dimension higher than χ. Thus, when χ is set to the maximum value, "manual" will use an amount of coefficients in the MPS which scales exponentially with n, while `quimb` will note that a $\chi = 2$ suffices for maintaining an exact representation of the GHZ state, and use an $O(n)$ number of coefficients.

As a final analysis, it is interesting to widen the range of n considered, and add a comparison with the quantum circuit libraries `Cirq` and `Qiskit`. In this way, it will be possible to assess which QFT application is the most suitable if we seek numerically exact results. Thus, Figure (19) is produced, where we reduce the number of iterations for the time estimation from 50 to 10 to compensate the larger range of n.

It is interesting to notice that both `Cirq` and `Qiskit` work significantly better than the others for

$n \lesssim 20$. The two packages, in fact, rely on sparse matrices and hence their implementation is very efficient for smaller systems. However, when the size of the system overcomes $n = 17$, we notice a sudden increase in the computational time: the threshold for these two algorithms is met and hence they are not able to compute the QFT result fast.

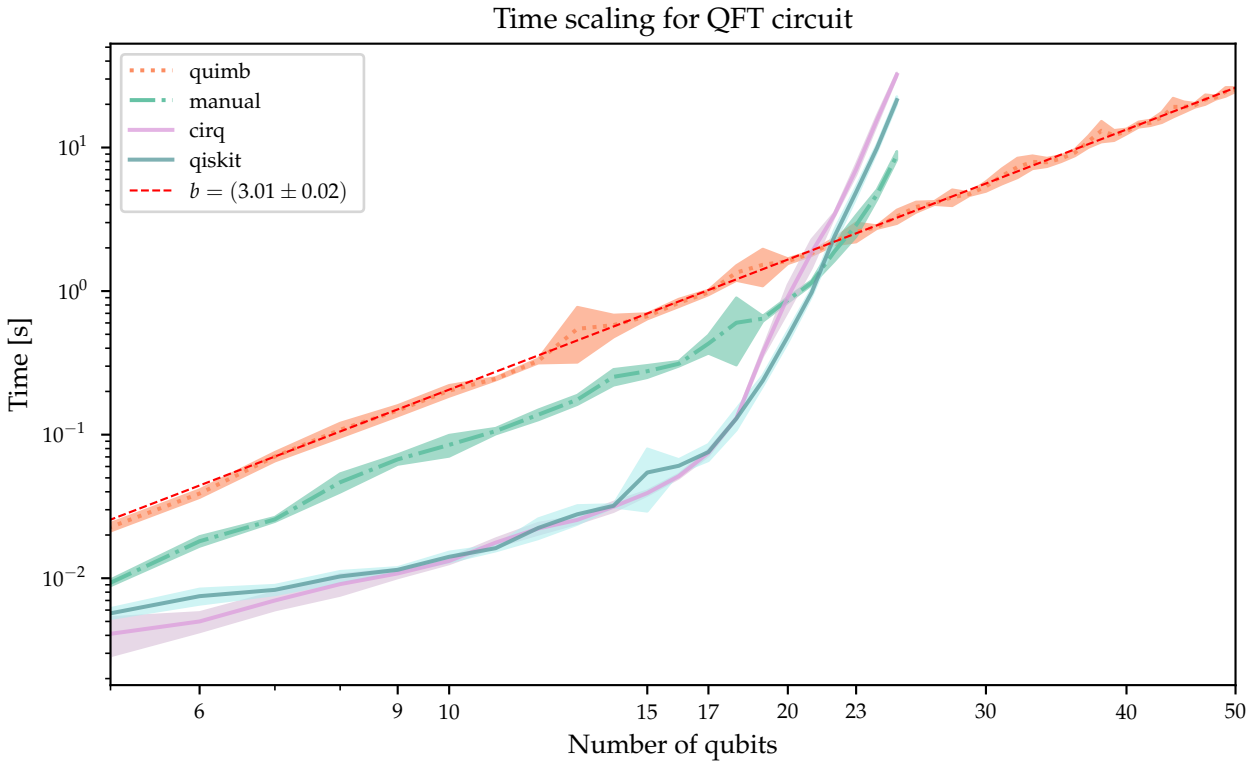

Figure (19) – Log-log plot of execution time vs. number of qubits, for the application to a GHZ state of the QFT circuit, with no constraint on χ, so that numerically exact results are produced. Each line represents a different implementation. The shaded area represents values within 1 standard deviation from the average time, computed along 10 iterations.

The most remarkable outcome in this graph, however, is that we finally obtained proof of the goodness of the `quimb` method. As a matter of facts, it is evident that for $n > 23$, the "manual" implementation's execution time starts to diverge and to decisively increase, due to the inefficiency of not compressing the state to its optimal size. `quimb`, on the other hand, is still able to manage even much bigger number of qubits, with a considerably less relevant increase in the computational time.

4. Conclusion

In this work, we managed to successfully implement the QFT algorithm with quantum circuits for dense states, and with tensor networks for Matrix Product States.

The circuit implementation worked properly and with comparable performances with both the `Python` libraries adopted. Therefore, it can be assessed that both `Cirq` and `Qiskit` are suitable for implementing the QFT circuit.

The code produced correct results also for the MPS-TEBD algorithm, in both the applications we developed ("manual" and `quimb`), despite showing some differences from the performance point of view. The "manual" algorithm with a fixed bond dimension χ proved to be more efficient than the `quimb` one for small systems (possibly due to a lower overhead). However, one advantage of `quimb` is that it is able to automatically optimise the χ. Thus, if one is interested in obtaining numerically exact results, with no bound on χ, `quimb` can scale significantly better than the "manual" implementation.

The interface between `Qiskit` and `quimb` allowed us to create an even more flexible code that can be adapted from one framework to the other, hence linking the tensor network MPS representation to the quantum circuit one.

References

[1] Héctor Abraham et al. *Qiskit: An Open-source Framework for Quantum Computing*. 2019. DOI: 10.5281/zenodo.2562110.

[2] Jacob C Bridgeman and Christopher T Chubb. "Hand-waving and interpretive dance: an introductory course on tensor networks". In: *Journal of Physics A: Mathematical and Theoretical* 50.22 (May 2017), p. 223001. ISSN: 1751-8121. DOI: 10.1088/1751-8121/aa6dc3. URL: http://dx.doi.org/10.1088/1751-8121/aa6dc3.

[3] Cirq Developers. *quantumlib/Cirq: Cirq v0.9.1*. Version v0.9.1. See full list of authors on Github: https://github.com/quantumlib/Cirq/graphs/contributors. Oct. 2020. DOI: 10.5281/zenodo.4064322. URL: https://doi.org/10.5281/zenodo.4064322.

[4] J. Eisert. *Entanglement and tensor network states*. 2013. arXiv: 1308.3318 [quant-ph].

[5] Jens Eisert, Marcus Cramer, and Martin B Plenio. "Colloquium: Area laws for the entanglement entropy". In: *Reviews of Modern Physics* 82.1 (2010), p. 277.

[6] Glen Evenbly. *Tutorial 3: Gauge Freedom*. URL: https://www.tensors.net/tutorial-3 (visited on 02/12/2021).

[7] Matthew Fishman, Steven R. White, and E. Miles Stoudenmire. *The ITensor Software Library for Tensor Network Calculations*. 2020. arXiv: 2007.14822. URL: http://itensor.org/docs.cgi?page=formulas/gate&vers=cppv3.

[8] Johnnie Gray. "quimb: A python package for quantum information and many-body calculations". In: *Journal of Open Source Software* 3.29 (2018), p. 819. DOI: 10.21105/joss.00819. URL: https://doi.org/10.21105/joss.00819.

[9] Matthew B Hastings. "An area law for one-dimensional quantum systems". In: *Journal of Statistical Mechanics: Theory and Experiment* 2007.08 (2007), P08024.

[10] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. 10th. USA: Cambridge University Press, 2011, p. 109. ISBN: 1107002176.

[11] Evert van Nieuwenburg. *An introduction to MPS*. https://evert.info/wp-content/uploads/2019/06/MPSEvert.pdf. May 2019.

[12] Román 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. ISSN: 0003-4916. DOI: 10.1016/j.aop.2014.06.013. URL: http://dx.doi.org/10.1016/j.aop.2014.06.013.

[13] Sebastian Paeckel et al. "Time-evolution methods for matrix-product states". In: *Annals of Physics* 411 (Dec. 2019), p. 167998. ISSN: 0003-4916. DOI: 10.1016/j.aop.2019.167998. URL: http://dx.doi.org/10.1016/j.aop.2019.167998.

[14] Roger Penrose. "Applications of negative dimensional tensors". In: *Combinatorial mathematics and its applications* 1 (1971), pp. 221–244.

[15] Mikel Sanz. "Tensor Networks in Condensed Matter". PhD thesis. Apr. 2011. DOI: 10.14459/2011md1070963.

[16] Pietro Silvi. *Tensor Networks: a quantum-information perspective on numerical renormalization groups*. 2012. arXiv: 1205.4198 [quant-ph].

[17] Pietro Silvi et al. "The Tensor Networks Anthology: Simulation techniques for many-body quantum lattice systems". In: *SciPost Physics Lecture Notes* (Mar. 2019). ISSN: 2590-1990. DOI: 10.21468/scipostphyslectnotes.8. URL: http://dx.doi.org/10.21468/SciPostPhysLectNotes.8.

[18] F. Verstraete, V. Murg, and J.I. 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. ISSN: 1460-6976. DOI: 10.1080/14789940801912366. URL: http://dx.doi.org/10.1080/14789940801912366.