




Article

# A Modular Framework for Generic Quantum Algorithms

Alberto Manzano <sup>1,\*</sup>, Daniele Musso <sup>2,\*</sup>, Álvaro Leitao <sup>1</sup> , Andrés Gómez <sup>3</sup> , Carlos Vázquez <sup>1</sup> , Gustavo Ordóñez <sup>4</sup> and María R. Nogueiras <sup>5</sup>

<sup>1</sup> Department of Mathematics and CITIC, Universidade da Coruña, 15001 A Coruña, Spain; alvaro.leitao@udc.gal (Á.L.); carlos.vazquez.cendon@udc.es (C.V.)

<sup>2</sup> Department of Physics, Instituto de Ciencias y Tecnologías Espaciales de Asturias (ICTEA), Oviedo University, 33003 Oviedo, Spain

<sup>3</sup> Galicia Supercomputing Center (CESGA), 15705 Santiago de Compostela, Spain; andres.gomez.tato@cesga.es

<sup>4</sup> Moody's Analytics, Moody's, Edinburgh EH3 8RB, UK; Gustavo.Ordenez@moodys.com

<sup>5</sup> Global Risk Analytics, The Hongkong and Shanghai Banking Corporation (HSBC), London B1 1HQ, UK; maria.r.nogueiras@hsbc.com

\* Correspondence: alberto.manzano.herrero@udc.es (A.M.); mussodaniele@uniovi.es (D.M.)

† These authors contributed equally to this work.

**Abstract:** We describe a general-purpose framework to design quantum algorithms. This framework relies on two pillars: a basic data structure called *quantum matrix* and a modular structure based on three quasi-independent modules. These latter include a loading module, a tool-kit of basic quantum arithmetic operations and a read-out module. We briefly discuss the loading and read-out modules, while the arithmetic module is analyzed in more depth. Eventually, we give explicit examples regarding the manipulation of generic oracles and hint at possible applications.

**Keywords:** quantum computing; linear algebra; quantum simulation

**MSC:** 81P68



**Citation:** Manzano, A.; Musso, D.; Leitao, Á.; Gómez, A.; Vázquez, C.; Ordóñez, G.; Nogueiras, M.R. A Modular Framework for Generic Quantum Algorithms. *Mathematics* **2022**, *10*, 785. <https://doi.org/10.3390/math10050785>

Academic Editor: Jan Śladkowski

Received: 14 February 2022

Accepted: 26 February 2022

Published: 1 March 2022

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Copyright:** © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction, Motivation and Main Results

Quantum hardware and software are still in their early development days, so that the design of quantum algorithms typically focuses on low-level operations. Although one should always keep in mind the hardware limitations, especially when describing possible near-term implementations of quantum algorithms, it is convenient to pursue higher levels of abstraction. Apart from its long-term and algorithmic interest, a more abstract and standardized approach serves practical purposes too, for example that of making the benchmarking of quantum computer performances a more solid and transparent process. In turn, this helps pushing forward the research and the development in quantum computation at all levels.

In the present paper, we describe a novel framework for the design of quantum algorithms on a more abstract plane. Although the isolated components of the framework that we present here might not be new, it is the combination of them that is novel. This paper is especially suited for researchers coming from other fields who see quantum computing as a potentially useful tool within their own subject. To this aim, our first proposal consists of the definition of a *quantum matrix*, namely a quantum state organized in two registers:

$$|\psi\rangle = \sum_{i=0}^{I-1} \sum_{j=0}^{J-1} c_{ij} |i\rangle_{n_I} \otimes |j\rangle_{n_J}, \quad (1)$$

where  $| \rangle_{n_j}$  indicates a register composed of  $n_j$  qubits corresponding to  $J = 2^{n_j}$  states, while  $| \rangle_{n_I}$  is a register composed of  $n_I$  qubits corresponding to  $I = 2^{n_I}$  states. The overall state  $|\psi\rangle$ , as defined in (1), is manifestly presented with the structure of a matrix; specifically,

we interpret  $i$  as the index running over the rows and  $j$  as the index running over the columns. The rightmost qubit within a register is associated with the least significant digit of the associated index, in binary notation (thus, we are adopting a little-endian convention). This way of storing the information has a common ground with that of Flexible Representation of Quantum Images (FRQI) and the Novel Enhanced Quantum Representation (NEQR) [1,2]. The main difference with FRQI and NEQR is that we encode the information of the  $(i, j)$  entry of the matrix in the quantum amplitude  $c_{ij}$ . Intuitively, the matrix (1) is a bi-dimensional memory array where  $c_{ij}$  encodes the information stored in the  $|i\rangle_{n_I} \otimes |j\rangle_{n_J}$  memory location (see Figure 1).

|                   | $ 0\rangle_{n_I}$ | $ 1\rangle_{n_I}$ | ... | $ j\rangle_{n_I}$ | ... | $ J\rangle_{n_I}$ |
|-------------------|-------------------|-------------------|-----|-------------------|-----|-------------------|
| $ 0\rangle_{n_I}$ | $c_{00}$          | $c_{01}$          | ... | $c_{0j}$          | ... | $c_{0J}$          |
| $ 1\rangle_{n_I}$ | $c_{10}$          | $c_{11}$          | ... | $c_{1j}$          | ... | $c_{1J}$          |
| ...               | ...               | ...               | ... | ...               | ... | ...               |
| $ i\rangle_{n_I}$ | $c_{i0}$          | $c_{i1}$          | ... | $c_{ij}$          | ... | $c_{iJ}$          |
| ...               | ...               | ...               | ... | ...               | ... | ...               |
| $ I\rangle_{n_I}$ | $c_{I0}$          | $c_{I1}$          | ... | $c_{Ij}$          | ... | $c_{IJ}$          |

**Figure 1.** Quantum matrix structure. The figure details how we organize the quantum amplitudes into a matrix. As explained in the main text, such organization mainly serves practical purposes. Besides, it allows for tensorial generalizations.

Here we must address a key technical feature about how we encode the information into the quantum amplitudes, the so-called *direct embedding* [3]. Namely, the information to be stored into the quantum matrix is directly encoded into the complex amplitudes  $c_{ij}$ . This contrasts with the customary choice in many fields of applications (see [4] for example) of loading the information into the probabilities  $|c_{ij}|^2$ . The reason behind this latter encoding is that most of the read-out algorithms are only capable of extracting the module of the complex amplitude. We make further comments on this point in Section 4. Furthermore, the direct embedding that we adopt has relevant implications in later stages of the quantum algorithms. Most importantly, the information stored into the quantum state is handled and combined more easily, because algebraic operations are not hampered by the presence of square roots. This allows us to define an “arithmetic library” composed of many fundamental arithmetic operations to handle arrays stored into the quantum matrix. Such a “general purpose” library provides a versatile framework for the implementation of wide classes of algorithms. In this work, we provide some simple example algorithms without aiming to be exhaustive. The possibility of implementing arithmetic operations within a quantum framework has been considered in the literature since the early days of quantum computation. Apart from the quantum implementation of logical circuits corresponding to basic operations, like the quantum adder [5,6], the manipulation of “continuous” numbers has also been studied. Let us mention some works which, at least in spirit, are closer to ours [7–10]. Unlike these previous approaches, we use a new embedding and organize the information into a matrix (1); these two aspects combined allow us to work in a transparent and simple manner.

The second and final proposal in this paper is to give a full overview of the complete pipeline, or overall structure, of the generic algorithm admitting implementation within this framework. The first step of every algorithm corresponds to loading some input data. In the quantum case, it is often convenient to split this step into two sub-steps:

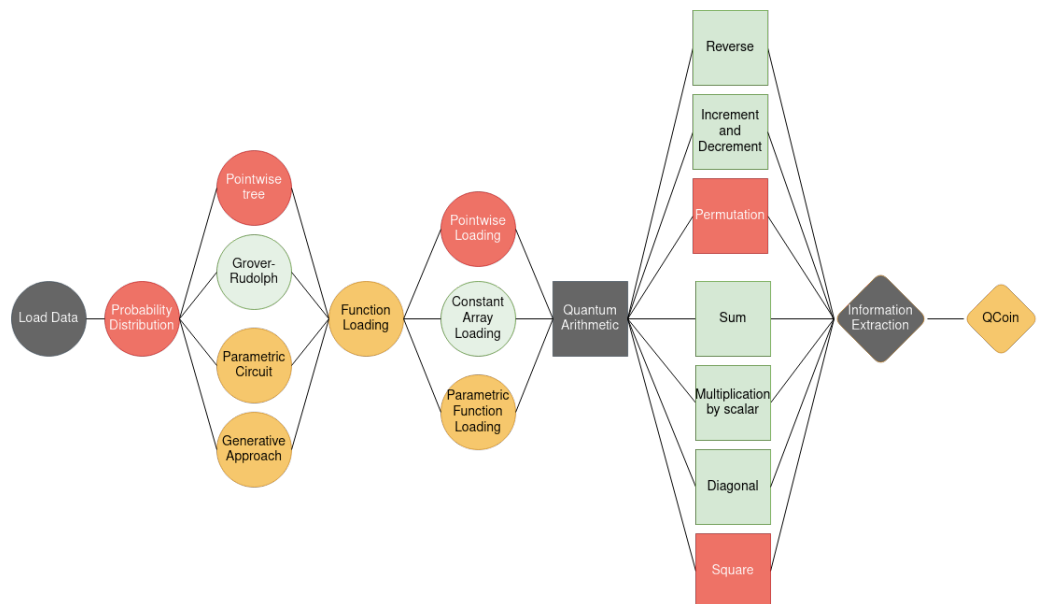
- loading a probability distribution  $p_{ij}$  [3,11,12]

- loading a bi-dimensional function  $f_{ij}$  (possibly by means of methods that load information a line at a time). In reference to the notation introduced for the quantum matrix (1), we have  $c_{ij} = p_{ij}f_{ij}$ , where the indexes are not summed over.

It is not strictly necessary to split the loading into two steps. Yet, we consider such splitting because –typically– we adopt different loading techniques for them: the probability distribution is loaded with a state preparation algorithm (e.g., a multiplexor binary tree), while the function is loaded by means of an auxiliary qubit meant to tell “good” and “bad” states apart. We describe the first step of the pipeline in Section 2.

In Section 3 we describe the second step of the pipeline corresponding to the implementation of various arithmetic operations, typically at the level of entire arrays or sub-arrays, and we refer to it as *quantum arithmetic*. In Section 4 we describe the last step of the pipeline, which corresponds to extracting the information that we have stored in the quantum state, namely the read-out of the state that encodes the result of the algorithm.

One of the advantages of organizing the pipeline as in Figure 2 is that it enjoys a modular structure. Therefore, we can develop and analyze each of the steps independently, thus achieving a better understanding of the problems in each domain. The color coding corresponds to the efficiency of the single modules. It should be stressed that the efficiency of the single modules depicted in the diagram of Figure 2 refers to the current state of the art which can change in the future, that is, it does not necessarily represent a structural limitation. In particular, an efficient algorithm would correspond to an end-to-end green path from left to right across the diagram. In searching for possible implementations for a desired algorithm, the challenge is to improve the necessary blocks so to follow a completely green path. This structure can be easily adapted to quantum neural networks, where the quantum arithmetic layer is interpreted as a layer (or a collection of layers) within the quantum network.



**Figure 2.** Diagrammatic structure of the pipeline. The dark grey nodes represent the three main steps; the nodes in red correspond to efficient algorithms, for those in yellow the efficiency varies from case to case and finally the nodes in green represent those algorithms which are more efficient. The concept of efficiency in the context of the present article is discussed in the Remark 1.

**Remark 1.** Throughout the text we use the terms complexity and efficiency:

- The complexity can be measured in different manners, specifically we measure it as the number of CNOT gates in an algorithm. This is motivated by the fact that CNOTs are sensibly more error-prone and require a longer execution time than single-qubit gates, as commented—for

instance—in [13]. In particular, we call an algorithm “complex” if its complexity scales as an exponential function of  $n$ ,  $n$  being the number of qubits in the circuit.

- The efficiency is much more difficult to define. In the first two modules, data loading and quantum arithmetic, with efficiency we refer to the complexity of the quantum algorithm plus the associated cost of all the classical auxiliary steps (if present). Since there is no standard way of adding the quantum and classical computational costs, we prefer not to quantitatively define efficiency. In the third module, information extraction, we instead use efficiency to refer to the number of calls to the oracle. Note that this is not related to the previous definition of complexity. This said, an overall efficient algorithm would correspond to one that is efficient in the three modules at the same time.

## 2. Data Loading

Data loading is a generic step which is required essentially in any quantum algorithm. The actual data to be loaded can vary in nature and serve different purposes. The data can correspond—for example—to the discretization of a normalized general real function  $f$  defined on a two-dimensional domain. Another example could be the pixels of an image in grayscale. Yet, we can as well think of the loading of more general data corresponding to a complex matrix, as long as the normalization of the quantum state is respected.

The recipe described in the appendix works in a pointwise fashion, exploiting an auxiliary register to store the desired value into the quantum amplitude at each “memory address”, namely, to store it into the associated entry of the quantum matrix. It is important to underline from the outset that this pointwise approach is generically not efficient. In order to attain efficiency at the level of the full algorithm, we need to assume that the loading procedure can be implemented in an alternative and efficient way; in other words, we need to assume the existence of a suitable efficient oracle. Nonetheless, as we will show and stress later, a set of efficient manipulations for generic arrays is possible even when their loading is not efficient. This observation stems directly from the modular structure described in the pipeline of Figure 2.

As already stated in Remark 1 there are two different aspects related to the efficiency of the state preparation, the quantum circuit complexity, on the one side, and the complexity of the pre-processing algorithm (where needed), on the other. An example of pre-processing algorithm would be the computation of the values of the angles in a tree-like loading (see for instance [11,14]). In such an example, although the quantum circuit has a low complexity, the cost of the classical pre-processing makes it inefficient. Here, we are going to discuss only the former, namely, the quantum circuit complexity.

Loading a generic real array is not a trivial problem. In Appendix A.1 we refer to a pointwise loading, without paying attention to its optimization (Despite it not being optimal, we have adopted the pointwise method for its simplicity. We postpone the study of more optimized approaches to the future). To this regard, the state of the art is currently set by two alternative approaches [15]: one based on multiplexors [13,16] and the other one based on Schmidt’s decomposition [17,18]. Both approaches give essentially the same leading CNOT complexity, namely, a number of CNOTs which scales as  $2^n$  for the preparation of the generic  $n$ -qubit state. Although this scaling might seem excessively large, we are loading  $N = 2^n$  points and the complexity of the algorithm is just of order  $O(N)$ .

Let us stress that, in the very specific case where we need to load a constant array, the procedure of Appendix A.2 requires (in the worst case scenario)  $n_I$  X-gates, two Y-rotations and one multi-controlled NOT gate. Such numbers must be compared with their classical counterpart, where the loading of a constant array on a line of the  $I \times J$  matrix requires  $J$  operations, considering that the process of copying a single number from memory is an operation. Therefore, the loading of a constant function is more interesting from the quantum speed-up perspective than the pointwise loading of a generic function. Indeed, in principle, we need exponentially fewer operations on a quantum computer to load a constant array. Interestingly, the number of operations needed does not depend on

the length of the constant array that we want to load, but it does depend on the number of rows of the matrix that we have to control. Here we can directly see the nature of quantum systems in practice, there is an “extra” cost associated to acting on a single element of the system without impacting the others. That makes operations on single elements inefficient and operations on the whole structure very efficient. A detailed analysis of the loading procedure will be explored in future works.

### 3. Quantum Arithmetic

In the present section we provide a collection of tools for the arithmetic handling of arrays encoded into a quantum matrix through direct embedding. These tools have been implemented and tested using QLM (by Atos). In this section we do not aim to address all the possible arithmetic operations implementable within this framework. Nor is in the spirit of this work to make an exhaustive analysis of the performance of the different operations here described. We simply give some particular examples in order to better discuss the working and possibilities of the framework.

Other operations potentially implementable within this framework are—for instance—those described in [19], a Walsh-Hadamard transform [20] or the replacement of the arithmetic unit by neural net processing unit.

#### 3.1. Ordering

The first operations that we introduce are those which allow us to move elements within the quantum matrix. Manipulating single elements in the matrix has a much higher cost than performing operations on the whole structure. For this reason, we first introduce a global reversing operation and then we introduce generic permutations.

##### 3.1.1. Reversing

By reversing, we mean the operation

$$R : (c_{i,0}, c_{i,1}, c_{i,2}, c_{i,3}, \dots, c_{i,J-1}) \longrightarrow (c_{i,J-1}, c_{i,J-2}, c_{i,J-3}, \dots, c_{i,0}), \tag{2}$$

where, for concreteness, we have addressed the reversing operation on the  $i$ -th row of the quantum matrix. Similarly, reversing the entire matrix corresponds to reading its entries from the lower-right corner i.e., in the opposite direction to how matrix entries are usually read. Note that it is straightforward to perform the reversing operation on a column.

We divide the process in three steps:

- Mask the row. In this case we only need to mask the register corresponding to the row (i.e., the  $| \rangle_{n_i}$  register) and leave the column register untouched. For more information on this operation see Appendix A.1.
- Apply  $n_j$  controlled X-gates. The controlled qubits are those of the row register. The target qubits are those of the column register.
- Undo step one, by applying again the same masking operation as before.

Following the steps above, we can perform a reversing operation on any row of the quantum matrix.

If we wanted instead to reverse the whole matrix, the operation would be more efficient than just reversing a row or a column. In that case, there is no need to control on any qubit, we just need to apply an X-gate to each register of the quantum matrix.

As an explicit example, let us think of an  $I \times J$  quantum matrix and, for simplicity, let us consider  $I = 2$ . Hence, we need  $n_i = 1$  and  $n_j = \log_2(J)$ . Suppose that we have loaded the following quantum matrix:

$$| \chi_1 \rangle = \frac{1}{A\sqrt{IJ}} \left( \sum_{j=0}^{J-1} c_{0j} |0\rangle_{n_i} \otimes |j\rangle_{n_j} + \sum_{j=0}^{J-1} c_{1j} |1\rangle_{n_i} \otimes |j\rangle_{n_j} \right). \tag{3}$$

In order to reverse the first row, we start by applying an X-gate to the row register obtaining the state:

$$|\chi_2\rangle = \frac{1}{A\sqrt{IJ}} \left( \sum_{j=0}^{J-1} c_{0j}|1\rangle_{n_I} \otimes |j\rangle_{n_J} + \sum_{j=0}^{J-1} c_{1j}|0\rangle_{n_I} \otimes |j\rangle_{n_J} \right). \tag{4}$$

Now, the row on which we are focusing, namely the one corresponding to  $c_{0j}$  for  $j = 0, \dots, J - 1$ , has all the qubits in the row register set to one (in this case the row register is just the qubit  $|1\rangle_{n_I}$ ). So, by means of controlled operations on  $|\chi_2\rangle$ , we act only on the row  $c_{0j}$ . Specifically, we apply an X-gate controlled on the row register, which acts on all the qubits of the column register. This yields

$$|\chi_3\rangle = \frac{1}{A\sqrt{IJ}} \left( \sum_{j=0}^{J-1} c_{0j}|1\rangle_{n_I} \otimes |J - 1 - j\rangle_{n_J} + \sum_{j=0}^{J-1} c_{1j}|0\rangle_{n_I} \otimes |j\rangle_{n_J} \right). \tag{5}$$

Finally, we apply again an X-gate to the row register obtaining:

$$|\chi_4\rangle = \frac{1}{A\sqrt{IJ}} \left( \sum_{j=0}^{J-1} c_{0j}|0\rangle_{n_I} \otimes |J - 1 - j\rangle_{n_J} + \sum_{j=0}^{J-1} c_{1j}|1\rangle_{n_I} \otimes |j\rangle_{n_J} \right). \tag{6}$$

The last step consists in undoing the mask.

### 3.1.2. Permutations

Permutations of two elements of an array, i.e., swaps of two entries, are demanding operations as we have to manipulate individual elements, instead of whole blocks in the quantum matrix. For simplicity, in what follows we discuss the algorithm referring to a quantum matrix given by a single row. Generalizing to larger matrices is straightforward. It is relevant to point out that also the extension to arrays with higher dimensions, i.e., from bi-dimensional matrices to  $d$ -dimensional tensors, is doable, although it requires additional controlled operations. Note that the additional controlled operations may result in additional complexity (see the definition of complexity in Remark 1). Specifically, consider the state:

$$|\phi_1\rangle = \frac{1}{\|f\|_\infty \sqrt{J}} \sum_{j=0}^{J-1} f_j |j\rangle_{n_I}, \tag{7}$$

where

$$\|f\|_\infty \equiv \max_{j=0, \dots, J} (|f_j|).$$

Moreover, let us write  $|\phi_1\rangle$  in (7) in the notation

$$(f_0, f_1, f_2, f_3, \dots, f_{J-4}, f_{J-3}, f_{J-2}, f_{J-1}), \tag{8}$$

which is more convenient to understand how the different gates act on the order of the components. The strategy presented here to perform a permutation of two arbitrary elements in (8) consists in using a pivot element. That is, we choose a fixed position  $k$  (pivot) and implement the permutations of the component placed at position  $k$  and any other component in the array. Once this is done, the generic swap of two elements can be obtained by means of three operations, at most. For example, if we aim to make a permutation of elements in positions  $i$  and  $j$  in

$$(f_0, \dots, f_i, \dots, f_k, \dots, f_j, \dots, f_{J-1}), \tag{9}$$

we would need to perform the following three steps: First, we permute the positions  $j \longleftrightarrow k$ , obtaining

$$(f_0, \dots, f_i, \dots, f_j, \dots, f_k, \dots, f_{J-1}). \tag{10}$$



Then, we consider the permutation of positions  $i \longleftrightarrow k$  corresponding to the permutation of elements  $i \longleftrightarrow j$ :

$$(f_0, \dots, f_j, \dots, f_i, \dots, f_k, \dots, f_{j-1}) . \tag{11}$$

Finally, we perform again step one, obtaining the desired permuted state, namely

$$(f_0, \dots, f_j, \dots, f_k, \dots, f_i, \dots, f_{j-1}) . \tag{12}$$

Now, the key in the algorithm is to understand how to actually perform in practice the permutations with the pivot. They can be implemented through X-gates and controlled X-gates. Moreover –without losing generality–, we choose the last element of the register as the pivot. If we have  $n_j$  qubits, the single X-gates acting on state (8) have the effects described in Table 1. The symbol **1** represents the identity matrix of order 2.

**Table 1.** The block of elements marked in orange is interchanged with the block of entries marked in yellow. However, note that each gate affects the whole state.

| Gate                                                      | Old State                                                                                    | New State                                                                                                            |
|-----------------------------------------------------------|----------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| $\mathbf{1}^{\otimes n_j-1} \otimes X$                    | $(f_0, f_1, f_2, f_3, \dots, f_{j-4}, f_{j-3}, f_{j-2}, f_{j-1})$                            | $(f_1, f_0, f_3, f_2, \dots, f_{j-3}, f_{j-4}, f_{j-1}, f_{j-2})$                                                    |
| $\mathbf{1}^{\otimes n_j-2} \otimes X \otimes \mathbf{1}$ | $(f_0, f_1, f_2, f_3, \dots, f_{j-4}, f_{j-3}, f_{j-2}, f_{j-1})$                            | $(f_2, f_3, f_0, f_1, \dots, f_{j-2}, f_{j-1}, f_{j-4}, f_{j-3})$                                                    |
| ...                                                       | ...                                                                                          | ...                                                                                                                  |
| $X \otimes \mathbf{1}^{\otimes n_j-1}$                    | $(f_0, f_1, f_2, f_3, \dots, f_{j/2-1}, f_{j/2}, \dots, f_{j-4}, f_{j-3}, f_{j-2}, f_{j-1})$ | $(f_{j/2}, f_{j/2+1}, f_{j/2+2}, f_{j/2+3}, \dots, f_{j-1}, f_0, \dots, f_{j/2-4}, f_{j/2-3}, f_{j/2-2}, f_{j/2-1})$ |

From Table 1 we can see that the single X-gates are performing swaps of blocks of contiguous memory positions. When we act on more significant qubits we are affecting bigger blocks and each gate is affecting the whole state. In this algorithm we are only interested on the effect that the gate has on certain blocks of the array (the ones highlighted). Using multi-controlled X-gates where the controls are applied to all qubits (except the one where we apply the X gate) and acting on state (8), we get the results reported in Table 2.

**Table 2.** The yellow and orange blocks in the second column are the same as in Table 1. The controlled gates interchange the last element of the block marked in orange with the last element of the block marked in yellow. This effect can be seen in the third column, where we highlight in orange and yellow which are the elements that have been affected by the controlled gate.

| Gate                                    | Old State                                                                                    | New State                                                                                    |
|-----------------------------------------|----------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|
| $c^{\otimes n_j-1} \otimes X$           | $(f_0, f_1, f_2, f_3, \dots, f_{j-4}, f_{j-3}, f_{j-2}, f_{j-1})$                            | $(f_0, f_1, f_2, f_3, \dots, f_{j-4}, f_{j-3}, f_{j-1}, f_{j-2})$                            |
| $c^{\otimes n_j-2} \otimes X \otimes c$ | $(f_0, f_1, f_2, f_3, \dots, f_{j-4}, f_{j-3}, f_{j-2}, f_{j-1})$                            | $(f_0, f_1, f_2, f_3, \dots, f_{j-4}, f_{j-1}, f_{j-2}, f_{j-3})$                            |
| ...                                     | ...                                                                                          | ...                                                                                          |
| $X \otimes c^{\otimes n_j-1}$           | $(f_0, f_1, f_2, f_3, \dots, f_{j/2-1}, f_{j/2}, \dots, f_{j-4}, f_{j-3}, f_{j-2}, f_{j-1})$ | $(f_0, f_1, f_2, f_3, \dots, f_{j-1}, f_{j/2}, \dots, f_{j-4}, f_{j-3}, f_{j-2}, f_{j/2-1})$ |

In this case it is clear that the effect of the controlled operations is to permute the last elements of each highlighted block.

We need to combine both operations, X-gates and multi-controlled X-gates, to perform the permutation of any element with the pivot (the last element, according to our choice). The strategy can be implemented recursively in the following way:

1. Move the last element of the array to the block where the element we wish to permute is located. This is done through a suitable multi-controlled X-gate.
2. If at this point the two elements that we wanted to interchange have been actually swapped, then undo all previous operations (both X-gates and multi-controlled X-gates) except for the last one and finish. These operations are needed to bring back to their original position all the other elements except the pair that has been swapped. Otherwise continue.
3. Swap the blocks on which we have acted at step 1. This is done through a single X-gate and serves the purpose of moving to the right the block on which we need to focus.
4. Go back to step 1.

For the sake of clarity, let us give a simple explicit example. Consider the state

$$|\psi\rangle = (f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7), \tag{13}$$

and suppose we want to permute the first element  $f_0$  with the pivot element  $f_7$ . We can proceed as follows:

$$\begin{aligned} (X \otimes c \otimes c) & \left( \begin{matrix} f_0, f_1, f_2, f_3, & f_4, f_5, f_6, f_7 \end{matrix} \right) = \left( f_0, f_1, f_2, f_7, f_4, f_5, f_6, f_3 \right), \\ (X \otimes \mathbf{1} \otimes \mathbf{1}) & \left( \begin{matrix} f_0, f_1, f_2, f_7, & f_4, f_5, f_6, f_3 \end{matrix} \right) = \left( f_4, f_5, f_6, f_3, f_0, f_1, f_2, f_7 \right), \\ (c \otimes X \otimes c) & \left( f_4, f_5, f_6, f_3, f_0, f_1, f_2, f_7 \right) = \left( f_4, f_5, f_6, f_3, f_0, f_7, f_2, f_1 \right), \\ (\mathbf{1} \otimes X \otimes \mathbf{1}) & \left( f_4, f_5, f_6, f_3, f_0, f_7, f_2, f_1 \right) = \left( f_6, f_3, f_4, f_5, f_2, f_1, f_0, f_7 \right), \\ (c \otimes c \otimes X) & \left( f_6, f_3, f_4, f_5, f_2, f_1, f_0, f_7 \right) = \left( f_6, f_3, f_4, f_5, f_2, f_1, f_7, f_0 \right). \end{aligned}$$

Now that we have effectively swapped the element 0 and the element 7 we just have to relocate the rest of the elements (Step 3).

$$\begin{aligned} (\mathbf{1} \otimes X \otimes \mathbf{1}) & \left( f_6, f_3, f_4, f_5, f_2, f_1, f_7, f_0 \right) = \left( f_4, f_5, f_6, f_3, f_7, f_0, f_2, f_1 \right), \\ (c \otimes X \otimes c) & \left( f_4, f_5, f_6, f_3, f_7, f_0, f_2, f_1 \right) = \left( f_4, f_5, f_6, f_3, f_7, f_1, f_2, f_0 \right), \\ (X \otimes \mathbf{1} \otimes \mathbf{1}) & \left( f_4, f_5, f_6, f_3, f_7, f_1, f_2, f_0 \right) = \left( f_7, f_1, f_2, f_0, f_4, f_5, f_6, f_3 \right), \\ (X \otimes c \otimes c) & \left( f_7, f_1, f_2, f_0, f_4, f_5, f_6, f_3 \right) = \left( f_7, f_1, f_2, f_3, f_4, f_5, f_6, f_0 \right). \end{aligned}$$

In general, the number of multi-controlled X gates that we need to apply for the permutation of the last element of the array with the first one, (the worst case scenario) is of order  $2^{n-1} + 1$ , the length of the array being  $2^n$ .

### 3.1.3. Cyclic Permutations

Cyclic permutations correspond to the two transformations:

$$\text{Left} : (f_0, f_1, f_2, \dots, f_{j-2}, f_{j-1}) \longrightarrow (f_1, f_2, f_3, \dots, f_{j-1}, f_0), \tag{14}$$

$$\text{Right} : (f_0, f_1, f_2, \dots, f_{j-2}, f_{j-1}) \longrightarrow (f_{j-1}, f_0, f_1, \dots, f_{j-3}, f_{j-2}), \tag{15}$$



where we follow the same notation adopted in Section 3.1.2. These operators have been discussed in depth in [21] and their implementation can be immediately extended to our framework, upon adding suitable controls.

### 3.2. Addition

In this subsection we discuss both the sum of whole arrays and the sum of their components (reduction).

#### 3.2.1. Sum

Consider the state  $|\psi_1\rangle$  given in (A4) (In what follows, we adopt the notation  $\supset$  to consider just some specific terms that are relevant for our purposes within a bigger quantum state):

$$|\psi_1\rangle \supset \frac{1}{\|f\|_\infty \sqrt{IJ}} \sum_{j=0}^{J-1} \sum_{i=0}^{I-1} f_{ij} |i\rangle_{n_I} \otimes |j\rangle_{n_J}, \tag{16}$$

where we have omitted the auxiliary register  $| \rangle_a$  for convenience. Applying a Hadamard gate on the first qubit of the row register, namely

$$|\psi_2\rangle = (\mathbf{1}^{\otimes n_I-1} \otimes H \otimes \mathbf{1}^{\otimes n_J}) |\psi_1\rangle, \tag{17}$$

we get the sum and difference of the the rows grouped in pairs, explicitly

$$|\psi_2\rangle \supset \frac{1}{\sqrt{2}\|f\|_\infty \sqrt{IJ}} \sum_{j=0}^{J-1} \sum_{i=0}^{\frac{I}{2}-1} \left[ (f_{2i,j} + f_{2i+1,j}) |2i\rangle_{n_I} \otimes |j\rangle_{n_J} + (f_{2i,j} - f_{2i+1,j}) |2i+1\rangle_{n_I} \otimes |j\rangle_{n_J} \right]. \tag{18}$$

In the first row of (18), we get the sum of the first and second row of (16). In the second row of (18), instead, we get the difference between the first and the second row of (16). In the third row of (18), we get the sum of the third and fourth row of (16), and the same structure continues on.

An analogous sum/difference operation can be performed in columns. Note that, in order to consider the correct number of  $\frac{1}{\sqrt{2}}$  factors, we need to count the Hadamard gates that we apply. Eventually, to sum two rows that are not in the same pair, we can take advantage of the permutations described in Section 3.1.2.

#### 3.2.2. Reductions

By *reduction* we mean the summation of all the elements of an array

$$\text{Reduction} : (f_0, f_1, \dots, f_{J-1}) \longrightarrow (f_0 + f_1 + \dots + f_{J-1}, \dots), \tag{19}$$

where the result of the reduction is stored into the first entry of the array. Consider again the state defined in (16), that is,

$$|\psi_1\rangle \supset \frac{1}{\|f\|_\infty \sqrt{IJ}} \sum_{j=0}^{J-1} \sum_{i=0}^{I-1} f_{ij} |i\rangle_{n_I} \otimes |j\rangle_{n_J}. \tag{20}$$

In order to perform a reduction by rows (i.e., summing the elements of each row and storing the result on the first column), we just need to apply a Hadamard gate to every qubit of the column register

$$|\psi_2\rangle = (\mathbf{1}^{\otimes n_I} \otimes H^{\otimes n_J}) |\psi_1\rangle, \tag{21}$$

which gives:

$$|\psi_2\rangle \supset \frac{1}{\sqrt{2}\|f\|_\infty \sqrt{IJ}} \sum_{i=0}^{I-1} \left( \sum_{j=0}^{J-1} f_{ij} \right) |i\rangle_{n_I} \otimes |0\rangle_{n_J}. \tag{22}$$

The parenthesis implies that we have the reduction of each row in the first column (which corresponds to  $|0\rangle_{n_J}$ ). In the rest of the columns, we get other reductions with different combinations of signs, as implemented by the Walsh-Hadamard operator in (21).

If we were to do a reduction by columns, instead of by rows, we will need to apply the Walsh-Hadamard gate to the row register, instead of the column register. Correspondingly, we will get the reduction of the columns on the first row.

### 3.3. Products

In this subsection we consider the product of a whole array by a constant and the component-wise product of two arrays. Eventually, the scalar product of two arrays can be obtained composing the product of two arrays and a reduction. Similarly, the squaring of an array can be obtained as the product of the array by itself.

#### 3.3.1. Multiplication by a Constant

In order to multiply a row or a column by a constant, we need an extra qubit which we denote  $|0\rangle_{\text{mul}}$ . Consider the state defined in (A4), but this time supplemented with the extra qubit:

$$|\psi_1\rangle \supset \frac{1}{\|f\|_\infty \sqrt{IJ}} |0\rangle_{\text{mul}} \otimes \sum_{j=0}^{J-1} \sum_{i=0}^{I-1} f_{ij} |i\rangle_{n_I} \otimes |j\rangle_{n_J}. \tag{23}$$

The multiplication operation consists merely in a controlled rotation. The rotation needs to be applied onto the auxiliary register  $|0\rangle_{\text{mul}}$ , it introduces a factor  $\alpha = \cos \frac{\theta}{2}$ , so we are initially restricted to multiplication by numbers between 0 and 1. This limitation can be circumvented by means of suitable manipulations of the normalization constants.

Depending on the controls that we apply, we can multiply a row, a column or a specific individual entry by  $\alpha$ . For example, assume that we want to multiply the first row by  $\alpha$ . In order to act solely on the first row, we first have to mask it by applying

$$|\psi_2\rangle = (X^{n_I} \otimes \mathbf{1}^{n_J}) |\psi_1\rangle; \tag{24}$$

explicitly, we obtain:

$$|\psi_2\rangle \supset \frac{1}{\|f\|_\infty \sqrt{IJ}} |0\rangle_{\text{mul}} \otimes \sum_{j=0}^{J-1} f_{0j} |I-1\rangle_{n_I} \otimes |j\rangle_{n_J} + \dots \tag{25}$$

The next step is to perform the controlled Y-rotation

$$|\psi_3\rangle = [\mathcal{R}_y(\theta) \otimes C^{\otimes n_I} \otimes \mathbf{1}^{\otimes n_J}] |\psi_2\rangle, \tag{26}$$

where we have indicated the controls of the controlled rotations with the symbol C. Thus, (26) is to be interpreted as a controlled Y-rotation acting on  $|0\rangle_{\text{mul}}$  and controlled by the row register. Explicitly, (26) gives

$$\begin{aligned} |\psi_3\rangle \supset & \frac{1}{\|f\|_\infty \sqrt{IJ}} \cos\left(\frac{\theta}{2}\right) |0\rangle_{\text{mul}} \otimes \sum_{j=0}^{J-1} f_{0j} |I-1\rangle_{n_I} \otimes |j\rangle_{n_J} \\ & + \frac{1}{\|f\|_\infty \sqrt{IJ}} \sin\left(\frac{\theta}{2}\right) |1\rangle_{\text{mul}} \otimes \sum_{j=0}^{J-1} f_{0j} |I-1\rangle_{n_I} \otimes |j\rangle_{n_J} + \dots \end{aligned} \tag{27}$$

Eventually, we have to unmask the state

$$|\psi_4\rangle = (X^{n_I} \otimes \mathbf{1}^{n_J})|\psi_3\rangle, \tag{28}$$

which yields

$$\begin{aligned} |\psi_4\rangle \supset & \frac{1}{\|f\|_\infty \sqrt{IJ}} \cos\left(\frac{\theta}{2}\right) |0\rangle_{\text{mul}} \otimes \sum_{j=0}^{J-1} f_{0j} |0\rangle_{n_I} \otimes |j\rangle_{n_J} \\ & + \frac{1}{\|f\|_\infty \sqrt{IJ}} \sin\left(\frac{\theta}{2}\right) |1\rangle_{\text{mul}} \otimes \sum_{j=0}^{J-1} f_{0j} |0\rangle_{n_I} \otimes |j\rangle_{n_J} \\ & + \frac{1}{\|f\|_\infty \sqrt{IJ}} |0\rangle_{\text{mul}} \otimes \sum_{j=0}^{J-1} \sum_{i=1}^{I-1} f_{ij} |i\rangle_{n_I} \otimes |j\rangle_{n_J}. \end{aligned} \tag{29}$$

The relevant information is marked by  $|0\rangle_{\text{mul}}$ .

### 3.3.2. Array Multiplication

In the present section, we describe the theoretical proposal for a more advanced operation: the multiplication of arrays. Its (overall) efficiency is related to that of the loading process. Let us assume to dispose of the following oracles:

$$O_f(|0\rangle \otimes |j\rangle) = f_j |0\rangle \otimes |j\rangle + \sqrt{1 - f_j^2} |1\rangle \otimes |j\rangle, \tag{30}$$

$$O_g(|0\rangle \otimes |j\rangle) = g_j |0\rangle \otimes |j\rangle + \sqrt{1 - g_j^2} |1\rangle \otimes |j\rangle, \tag{31}$$

which load the arrays  $f$  and  $g$  respectively. Moreover, consider the swap operator:

$$S(|\alpha\rangle \otimes |\beta\rangle \otimes |j\rangle) = |\beta\rangle \otimes |\alpha\rangle \otimes |j\rangle. \tag{32}$$

In order to build the multiplication operator we start from the state:

$$|\lambda_0\rangle = |0\rangle_g \otimes |0\rangle_f \otimes |j\rangle, \tag{33}$$

where we have two auxiliary qubits:  $|0\rangle_f$  to load  $f$  and  $|0\rangle_g$  to load  $g$ . First we load  $f$ :

$$|\lambda_1\rangle = (\mathbf{1} \otimes O_f) |\lambda_0\rangle = f_j |0\rangle_g \otimes |0\rangle_f \otimes |j\rangle + \sqrt{1 - f_j^2} |0\rangle_g \otimes |1\rangle_f \otimes |j\rangle. \tag{34}$$

In the second step we swap qubits  $| \rangle_f$  and  $| \rangle_g$ :

$$|\lambda_2\rangle = S|\lambda_1\rangle = f_j |0\rangle_f \otimes |0\rangle_g \otimes |j\rangle + \sqrt{1 - f_j^2} |1\rangle_f \otimes |0\rangle_g \otimes |j\rangle. \tag{35}$$

The third and last step consists in applying the oracle  $O_g$ :

$$\begin{aligned} |\lambda_3\rangle = (\mathbf{1} \otimes O_g) |\lambda_2\rangle = & g_j f_j |0\rangle_f \otimes |0\rangle_g \otimes |j\rangle + g_j \sqrt{1 - f_j^2} |1\rangle_f \otimes |0\rangle_g \otimes |j\rangle \\ & + \sqrt{1 - g_j^2} f_j |0\rangle_f \otimes |1\rangle_g \otimes |j\rangle + \sqrt{1 - g_j^2} \sqrt{1 - f_j^2} |1\rangle_f \otimes |1\rangle_g \otimes |j\rangle. \end{aligned} \tag{36}$$

The multiplication of arrays  $f$  and  $g$  is encoded in the registers marked by  $|0\rangle_f \otimes |0\rangle_g$  (In order to return back to the original ordering of the auxiliary qubits,  $|0\rangle_g \otimes |0\rangle_f$ , one can consider an extra swap  $S$ ):

$$|\lambda_3\rangle = (\mathbf{1} \otimes O_g) S (\mathbf{1} \otimes O_f) \supset g_j f_j |0\rangle_f \otimes |0\rangle_g \otimes |j\rangle. \tag{37}$$

This procedure can be extended to the multiplication of more than two arrays. It is worth noticing that this method depends on the loading complexity, that is, the efficiency of the employed oracles.

As a final comment, when in Section 1 we split the loading of the integrand  $f \cdot p$  in two parts, one associated to the distribution and one associated to the function, we were in fact performing the multiplication of the two arrays  $\{p_j\}$  and  $\{f_j\}$ .

### 3.3.3. Squaring and Scalar Product

The square of an array and the scalar product of two arrays can be obtained from operations that we have already defined above. The former is trivially just the multiplication of an array by itself. On the other hand, if we perform the reduction of the product of two arrays, we get their scalar product. As their construction depends on the steps commented in Section 3.3.2, the efficiency of the square of an array and the scalar product of two arrays is strongly dependent on the loading strategy for the arrays.

## 4. Information Extraction

With *information extraction* we loosely refer to any technique which allows us to read some information encoded in the quantum matrix. Within this broad category, we can identify two main groups of algorithms.

The first group includes those algorithms focusing on the estimation of the phase of a quantum state. The most well known example in the literature is the *Quantum Phase Estimation* (QPE) algorithm [22,23]. However, the overall structure of the QPE algorithm does not fit directly within the structure that we are proposing here. The reason for this is that, in order to implement QPE, we would need additional qubit registers.

The second group includes algorithms designed to estimate probability amplitudes. The most well known example in the literature is the *Quantum Amplitude Estimation* (QAE) algorithm [24]. More recent approaches try to demand less computational resources by removing the implicit QPE present in the QAE algorithm. Examples of works in this direction include the *Quantum Amplitude Estimation Simplified* (QAES) algorithm, the *Iterative Quantum Amplitude Estimation* (IQAE) algorithm and the *Maximum Likelihood Amplitude Estimation* (MLAE) algorithm [25–27]. QAE algorithm does not fit in our framework either, in fact it depends on QPE. Nevertheless, QAES, IQAE and MLAE do not use QPE and they can be naturally implemented in our framework. In these approaches, the required number  $N$  of oracle calls to obtain a precision  $\epsilon$  in the estimated probability amplitude is of order  $N = O(1/\epsilon)$  while, with a naive (i.e., unamplified) sampling, the number of oracle calls would grow as  $N = O(1/\epsilon^2)$ . We recall that, from Remark 1, the number of calls to an oracle (i.e., the efficiency of the information extraction module) is not related with the number of CNOT gates used in the first two modules (i.e., the complexity). To compute the total number of CNOT gates in the circuit, which represents the overall complexity of the quantum algorithm, we need to consider the number of CNOT gates used in the first two modules repeated as many times as the information extraction algorithm requires. This is the reason why we say that the three modules are *quasi-independent*.

The information extraction strategies deserve an in-depth investigation on their own. Within the here proposed framework, it would be of particular interest an algorithm able to read both the probability amplitude and the phase at the same time. We will explore this possibility in future works [28].

## 5. Examples

In this section we give two explicit examples of manipulations for a given oracle.

### 5.1. Constant Shift of a Given Oracle

Given an oracle  $O_f$  which loads the function  $f$ , we are interested in providing an efficient implementation of an oracle  $O_{\bar{f}}$  for the shifted function

$$\bar{f}_j = f_j - s, \quad (38)$$

where  $s$  is a generic real constant. Note that constructing the oracle for  $\bar{f}$  is non-trivial whenever  $f$  is non-constant. The constant shift of a constant function amounts to just a

global rotation. However, the constant shift of a non-constant function requires in general a position-dependent rotation.

More specifically, consider the state

$$|\phi_1\rangle = \frac{1}{\|f\|_\infty \sqrt{2J}} \sum_{j=0}^{J-1} \left( f_j |0\rangle_a \otimes |0\rangle_1 \otimes |j\rangle_{n_j} + \sqrt{1-f_j^2} |1\rangle_a \otimes |0\rangle_1 \otimes |j\rangle_{n_j} \right) + \frac{1}{\sqrt{2J}} \sum_{j=0}^{J-1} \left( |0\rangle_a \otimes |1\rangle_1 \otimes |j\rangle_{n_j} + |1\rangle_a \otimes |1\rangle_1 \otimes |j\rangle_{n_j} \right), \tag{39}$$

which is the result of applying the oracle  $O_f$  to the initial state  $|\psi_0\rangle$  given in (A2). Note that the state  $|\phi_1\rangle$  can be interpreted as having loaded the function  $f$  on the first row of the  $2 \times J$  quantum matrix.

Following the steps described in Appendix A.2, we can load the constant function  $s$  into the second row of  $|\phi_1\rangle$ . Specifically, we load a constant array on the states associated to  $i = 1$ . It is important to remember that the quantum register  $|i\rangle_{n_i}$  stores the row address of the arrays, while  $|j\rangle_{n_j}$  stores the column address.

More explicitly, we obtain

$$|\phi_2\rangle = \frac{1}{\|f\|_\infty \sqrt{2J}} \sum_{j=0}^{J-1} \left( f_j |0\rangle_a \otimes |0\rangle_1 \otimes |j\rangle_{n_j} + \sqrt{1-f_j^2} |1\rangle_a \otimes |0\rangle_1 \otimes |j\rangle_{n_j} \right) + \frac{1}{\sqrt{2J}} \sum_{j=0}^{J-1} \left( s |0\rangle_a \otimes |1\rangle_1 \otimes |j\rangle_{n_j} + \sqrt{1-s^2} |1\rangle_a \otimes |1\rangle_1 \otimes |j\rangle_{n_j} \right). \tag{40}$$

Following Section 3.2.1, we apply a Hadamard gate to the row qubit combining the two rows of the stored matrix, namely

$$|\phi_3\rangle = (\mathbf{1} \otimes H \otimes \mathbf{1}^{\otimes n_j}) |\phi_2\rangle = \frac{1}{2\|f\|_\infty \sqrt{J}} \sum_{j=0}^{J-1} \left( f_j |0\rangle_a \otimes |0\rangle_1 \otimes |j\rangle_{n_j} + f_j |0\rangle_a \otimes |1\rangle_1 \otimes |j\rangle_{n_j} + \sqrt{1-f_j^2} |1\rangle_a \otimes |0\rangle_1 \otimes |j\rangle_{n_j} + \sqrt{1-f_j^2} |1\rangle_a \otimes |1\rangle_1 \otimes |j\rangle_{n_j} + s |0\rangle_a \otimes |0\rangle_1 \otimes |j\rangle_{n_j} - s |0\rangle_a \otimes |1\rangle_1 \otimes |j\rangle_{n_j} + \sqrt{1-s^2} |1\rangle_a \otimes |0\rangle_1 \otimes |j\rangle_{n_j} - \sqrt{1-s^2} |1\rangle_a \otimes |1\rangle_1 \otimes |j\rangle_{n_j} \right). \tag{41}$$

Let us focus just on the components  $|0\rangle_a \otimes |1\rangle \otimes |j\rangle_{n_j}$ , namely

$$|\phi_3\rangle \supset |\phi_3^{(-)}\rangle \equiv \frac{1}{2\|f\|_\infty \sqrt{J}} \sum_{j=0}^{J-1} (f_j - s) |0\rangle_a \otimes |1\rangle_1 \otimes |j\rangle_{n_j}. \tag{42}$$

This means that we have stored the difference array  $f - s$ , i.e., the array  $f$  shifted by the constant  $s$ , in the second line of the matrix. Note that, at the same time, we have stored the sum array in the first line,

$$|\phi_3\rangle \supset |\phi_3^{(+)}\rangle \equiv \frac{1}{2\|f\|_\infty \sqrt{J}} \sum_{j=0}^{J-1} (f_j + s) |0\rangle_a \otimes |0\rangle_1 \otimes |j\rangle_{n_j}. \tag{43}$$

### 5.2. Approximated Linear Shift of a Given Oracle

Let us consider a state  $|\phi_4\rangle$  obtained by applying a CNOT operator to the state  $|\phi_3\rangle$  where the control qubit is the most significant digit in the register  $|j\rangle_{n_j}$  and the target qubit

is the row register. This operation flips the row index of the columns corresponding to the second half of the range  $\{0, \dots, J - 1\}$ , namely those in the range  $\{\frac{J}{2}, \dots, J - 1\}$ . We then have that the  $|0\rangle_a \otimes |1\rangle \otimes |j\rangle_{n_j}$  components of  $|\phi_4\rangle$  contain the function  $f$  shifted by  $-s$  in the lower half of points and by  $s$  in the higher half, namely

$$|\phi_4\rangle \supset |\phi_4^{(-)}\rangle \equiv \frac{1}{2\|f\|_\infty\sqrt{J}} \sum_{j=0}^{\frac{J}{2}-1} (f_j - s) |0\rangle_a \otimes |1\rangle_1 \otimes |j\rangle_{n_j} \tag{44}$$

$$+ \frac{1}{2\|f\|_\infty\sqrt{J}} \sum_{j=\frac{J}{2}}^{J-1} (f_j + s) |0\rangle_a \otimes |1\rangle_1 \otimes |j\rangle_{n_j} .$$

This corresponds to a shift by a step function, representing the most rudimentary approximation to a linear/RELU function.

So far, we have considered a two row matrix. However, if we want to iterate the step shift just described (for instance to the purpose of getting a finer approximation to a linear shift), we need to consider a larger matrix. Let us start over again with a matrix with  $n_j = 2$  (that is, 4 rows) and repeat all the steps that led us to  $|\phi_4\rangle$ ; we add a tilde to indicate that now we are working with a different matrix. We get

$$|\tilde{\phi}_4\rangle \supset |\tilde{\phi}_4^{(-)}\rangle \equiv \frac{1}{2\|f\|_\infty\sqrt{2J}} \sum_{j=0}^{\frac{J}{2}-1} (f_j - s) |0\rangle_a \otimes |01\rangle_2 \otimes |j\rangle_{n_j} \tag{45}$$

$$+ \frac{1}{2\|f\|_\infty\sqrt{2J}} \sum_{j=\frac{J}{2}}^{J-1} (f_j + s) |0\rangle_a \otimes |01\rangle_2 \otimes |j\rangle_{n_j} .$$

Now, let us load a constant array with the value  $s' = \frac{2}{3}s$ . We can then shift  $|\tilde{\phi}_4\rangle$  by  $s'$  and then apply a CNOT operator whose control qubit is the second most significant digit in the column register  $|j\rangle_{n_j}$  and whose target qubit is the most significant digit in the row register  $|i\rangle_2$ . This leads us to eventually choosing the components  $|0\rangle_a \otimes |11\rangle_2 \otimes |j\rangle_{n_j}$ , so we get

$$|\tilde{\phi}_5\rangle \supset |\tilde{\phi}_5^{(-)}\rangle \equiv \frac{1}{2\|f\|_\infty\sqrt{2J}} \sum_{j=0}^{\frac{J}{4}-1} \left(f_j - \frac{5}{3}s\right) |0\rangle_a \otimes |11\rangle_2 \otimes |j\rangle_{n_j} \tag{46}$$

$$+ \frac{1}{2\|f\|_\infty\sqrt{2J}} \sum_{j=\frac{J}{4}}^{\frac{J}{2}-1} \left(f_j - \frac{1}{3}s\right) |0\rangle_a \otimes |11\rangle_2 \otimes |j\rangle_{n_j}$$

$$+ \frac{1}{2\|f\|_\infty\sqrt{2J}} \sum_{j=\frac{J}{2}}^{\frac{3}{4}J-1} \left(f_j + \frac{1}{3}s\right) |0\rangle_a \otimes |11\rangle_2 \otimes |j\rangle_{n_j}$$

$$+ \frac{1}{2\|f\|_\infty\sqrt{2J}} \sum_{j=\frac{3}{4}J}^{J-1} \left(f_j + \frac{5}{3}s\right) |0\rangle_a \otimes |11\rangle_2 \otimes |j\rangle_{n_j} .$$

Although the process can be iterated, it becomes inefficient if the step-wise approximation of the linear shift is required to have a precision which scales as  $\frac{1}{N}$ ,  $N$  being the number of discretisation points.

### 6. Discussion and Conclusions

The main goal of this work is to propose and describe a generic framework for the design of quantum algorithms based on direct embedding. Its modular structure, as depicted in Figure 2, is appealing and handy in a number of ways. For example, under this framework the main components of a quantum algorithm, namely: data loading,



arithmetic manipulations and read-out, can be studied and discussed separately. This holds true also for considerations related to efficiency, the current status of which is reflected by the color coding of Figure 2; specifically, an end-to-end efficient pipeline would be represented by a left-to-right path within the diagram that encounters only green boxes. Thus, the modular structure of the pipeline for the generic quantum algorithm helps to organize the research effort, compare and interpret different algorithms, and identify possible bottle-necks. Furthermore, it is possible to combine this framework with other existing routines. For instance, it is possible to adopt one's favorite amplitude amplification and estimation technique for the information-extraction part.

On a more technical level, the direct embedding of information into the quantum amplitudes avoids having to deal with square roots and thereby it opens the way to easier arithmetic manipulations of the data stored in the quantum state. In particular, we defined the *quantum matrix*, a two-dimensional array which can be thought of in analogy to a memory register: the basis states correspond to the row and column addresses of the memory locations, while the entries of the matrix are the quantum amplitudes representing the loaded information. As it has been previously illustrated, this construction allows for the neat and flexible manipulation of arrays. We have also covered some basic arithmetic manipulations, for which we provided descriptions and implementation details. All in all, we set up a theoretical proposal for a package of arithmetic operations in a quantum framework. Its full potential and development requires further investigation and work in the three modules.

Quantum matrices can be naturally generalized to multi-dimensional arrays. All the proposed arithmetic manipulations, as well as the loading and read-out techniques, can be extended in a straightforward way to the higher-dimensional and more general tensor setting. However, this comes at the cost of the potential necessity of additional controlled operations needed for "masking" the array and act only on a desired subset of entries. In other words, the cost of an operation is related to the co-dimension of the subset of entries to which it applies.

Finally, we also provided two specific examples of applications that are interesting on their own, beyond the discussions of the present work. Namely, the shift of a generic oracle by a constant and the shift by a step-wise approximate linear function. We note that their efficient implementation depends on the efficiency of the oracle to which the shift is applied. A constant shift for an oracle implements a vertical offset and it is useful—for example—in iterative algorithms where at each iteration an output oracle needs to be centered vertically, i.e., along the  $y$  axis.

**Author Contributions:** Conceptualization, A.M., D.M. and Á.L.; Formal analysis, A.M. and D.M.; Methodology, A.M., D.M. and Á.L.; Project administration, A.G. and C.V.; Resources, A.G.; Supervision, A.G., C.V., G.O. and M.R.N.; Writing—original draft, A.M. and D.M. All authors conceived the research, explored preliminary directions, provided suggestions, discussed the results, and contributed to the manuscript. All authors have read and agreed to the published version of the manuscript.

**Funding:** All authors acknowledge the European Project NExt ApplicationS of Quantum Computing (NEASQC), funded by Horizon 2020 Program inside the call H2020-FETFLAG-2020-01(Grant Agreement 951821). A. Manzano, Á. Leitaó and C. Vázquez wish to acknowledge the support received from the Centro de Investigación de Galicia "CITIC", funded by Xunta de Galicia and the European Union (European Regional Development Fund-Galicia 2014–2020 Program), by grant ED431G 2019/01. Part of the computational resources for this project were provided by the Galician Supercomputing Center (CESGA).

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** The analysis presented here has neither exploited nor produced any dataset.

**Acknowledgments:** We would like to thank Vedran Dunjko, Dario Ferraro, Simon Martiel and Javier Mas for fruitful discussions on some aspects of the present work.

**Conflicts of Interest:** The authors declare no conflict of interest.

### Appendix A. Details on Data Loading

#### Appendix A.1. Pointwise Loading of a Matrix

In this subsection we show how to load a generic matrix (i.e., a two dimensional array) into a quantum matrix (1) in a pointwise fashion. We will be considering states of the form:

$$|\hat{\psi}\rangle = \sum_{i=0}^{I-1} \sum_{j=0}^{J-1} c_{ij} |0\rangle_a \otimes |i\rangle_{n_I} \otimes |j\rangle_{n_J}, \tag{A1}$$

which correspond to the quantum matrix introduced in (1) with the addition of an auxiliary register  $| \rangle_a$ . In what follows, let us assume for simplicity that the auxiliary register is one-dimensional, i.e., it consists of just one qubit. The other registers operate as described earlier when discussing Equation (1).

The pipeline of a quantum algorithm starts by loading an initial state. For example, this can represent a probability distribution and the most simple such case is the uniform distribution. Let us consider it explicitly. In order to load the uniform distribution, we apply the Walsh-Hadamard gate  $1 \otimes H^{\otimes n_I} \otimes H^{\otimes n_J}$  to the base state  $|0\rangle_a \otimes |0\rangle_{n_I} \otimes |0\rangle_{n_J}$ , thus obtaining:

$$|\psi_0\rangle = \frac{1}{\sqrt{IJ}} \sum_{i=0}^{I-1} \sum_{j=0}^{J-1} |0\rangle_a \otimes |i\rangle_{n_I} \otimes |j\rangle_{n_J}. \tag{A2}$$

Note that the loading of the distribution has not made use of the auxiliary qubit.

The next step in the pipeline is to load a real matrix  $f$  into the quantum matrix. In order to load a point  $f_{ij}$  in the corresponding register we need to act in such a way that we only impact the targeted quantum state. For that purpose we need have to perform three steps:

- Mask the state. Masking the state consists in converting the state  $|0\rangle_a \otimes |i\rangle_{n_I} \otimes |j\rangle_{n_J}$  into the state  $|0\rangle_a \otimes |I-1\rangle_{n_I} \otimes |J-1\rangle_{n_J} \equiv |0\rangle_a \otimes |11\dots 11\rangle_{n_I} \otimes |11\dots 11\rangle_{n_J}$ . In terms of qubits, this requires to apply a NOT gate to all the qubits that are zero for the original state, see Figure A1. The reason for this masking operation can be understood in the next step.
- Apply a suitable controlled  $Y$ -rotation on the auxiliary qubit. The controls have to be applied on all the qubits except for the auxiliary one. Here we can see that the complexity of the algorithm depends on the number of qubits that we have to control. The angle for the rotation needs to be  $\theta = \arccos\left(\frac{f_{ij}}{\|f\|_\infty}\right)$ , where  $\|f\|_\infty = \max_{i,j}(|f_{ij}|)$ . The factor  $\|f\|_\infty$  is needed to keep the amplitudes bounded so that the associated probabilities do not exceed 1.
- Undo step one. This consists in the application of the same mask already used at step one.

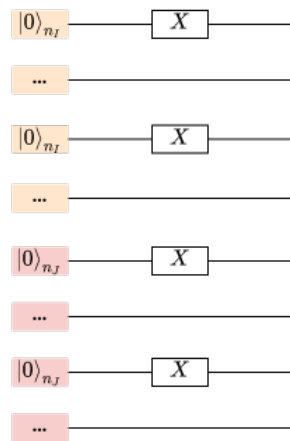


Figure A1. In yellow the row register. In red the column register.

Following this strategy we can load each of the values  $f_{ij}$ , thus getting the state:

$$|\psi_1\rangle = \frac{1}{\|f\|_\infty \sqrt{IJ}} \sum_{j=0}^{J-1} \sum_{i=0}^{I-1} \left( f_{ij} |0\rangle_a \otimes |i\rangle_{n_I} \otimes |j\rangle_{n_J} + \sqrt{1 - f_{ij}^2} |1\rangle_a \otimes |i\rangle_{n_I} \otimes |j\rangle_{n_J} \right). \quad (A3)$$

Usually we focus only on the states marked with  $|0\rangle_a$ , namely

$$|\psi_1\rangle \supset \frac{1}{\|f\|_\infty \sqrt{IJ}} \sum_{j=0}^{J-1} \sum_{i=0}^{I-1} f_{ij} |0\rangle_a \otimes |i\rangle_{n_I} \otimes |j\rangle_{n_J}. \quad (A4)$$

Appendix A.2. Loading a Constant Array

Loading a constant array follows pretty much the same strategy as the pointwise loading. For the purpose of giving an explicit example, we are going to describe the loading of a constant array, taking a real value  $c \leq 1$ , into a row of the quantum matrix. We start again loading a uniform distribution, thus obtaining (A2). Then, we use a similar structure for loading the array as the one discussed before, namely

- Mask the state. In this case we only need to mask the register corresponding to the row (the  $| \rangle_{n_I}$  register) and leave the column register untouched. As we mask only one register, we need fewer gates than for the pointwise loading described above.
- Apply a suitable controlled  $Y$ -rotation on the auxiliary qubit. The angle for the rotation needs to be  $\theta = \arccos(c)$ . The controls have to be made only in the row registers. Here we can see that the number of controls to load a constant array is drastically reduced with respect to the generic function.
- Undo step one, by applying the same mask already considered there.

If we were to load the constant array in the register  $|i\rangle_{n_I}$  we would get:

$$|\xi_1\rangle = \dots + \frac{1}{A\sqrt{IJ}} \sum_{j=0}^{J-1} \left( c |0\rangle_a \otimes |i\rangle_{n_I} \otimes |j\rangle_{n_J} + \sqrt{1 - c^2} |1\rangle_a \otimes |i\rangle_{n_I} \otimes |j\rangle_{n_J} \right) + \dots, \quad (A5)$$

where  $A$  is a normalization constant.

As it can be intuitively anticipated, the loading complexity grows together with the lack of symmetry of the loaded state. The constant case, being highly symmetric, is easy. In between the constant and the generic state with no symmetry, one can encounter lower degrees of symmetry, like for example functions which are piece-wise constant. We remind the reader that we adopted piece-wise constant functions in Section 5.2 to approximate a linear function and observed how the complexity grew with the approximation accuracy.

For further discussions on the relation between the loading complexity and the symmetry of the loaded state we refer to [15].

## References

1. Le, P.; Ilyyasu, A.; Dong, F.; Hirota, K. A flexible representation of quantum images for polynomial preparation, image compression and processing operations. *Quantum Inf. Process.* **2011**, *10*, 63–84. [[CrossRef](#)]
2. Zhang, Y.; Lu, K.; Gao, Y.; Wang, M. NEQR: A novel enhanced quantum representation of digital images. *Quantum Inf. Process.* **2013**, *12*, 2833–2860. [[CrossRef](#)]
3. Kubo, K.; Nakagawa, Y.O.; Endo, S.; Nagayama, S. Variational quantum simulations of stochastic differential equations. *arXiv* **2020**, arXiv:2012.04429.
4. Stamatopoulos, N.; Egger, D.J.; Sun, Y.; Zoufal, C.; Iten, R.; Shen, N.; Woerner, S. Option Pricing using Quantum Computers. *Quantum* **2020**, *4*, 291. [[CrossRef](#)]
5. Draper, T.G. Addition on a Quantum Computer. *arXiv* **2000**, arXiv:quant-ph/0008033.
6. Cuccaro, S.A.; Draper, T.G.; Kutin, S.A.; Petrie Moulton, D. A new quantum ripple-carry addition circuit. *arXiv* **2004**, arXiv:quant-ph/0410184.
7. Wang, S.; Wang, Z.; Cui, G.; Fan, L.; Shi, S.; Shang, R.; Li, W.; Wei, Z.; Gu, Y. Quantum Amplitude Arithmetic. *arXiv* **2020**, arXiv:quant-ph/2012.11056.
8. Vedral, V.; Barenco, A.; Ekert, A. Quantum networks for elementary arithmetic operations. *Phys. Rev. A* **1996**, *54*, 147–153. [[CrossRef](#)]
9. Lloyd, S.; Braunstein, S.L. Quantum Computation over Continuous Variables. *Phys. Rev. Lett.* **1999**, *82*, 1784–1787. [[CrossRef](#)]
10. Xiaopeng, C.; Yu, S. *QBLAS: A Quantum Basic Linear Algebra and Simulation Library*; GitHub: San Francisco, CA, USA, 2019.
11. Grover, L.; Rudolph, T. Creating superpositions that correspond to efficiently integrable probability distributions. *arXiv* **2002**, arXiv:quant-ph/0208112.
12. Nakaji, K.; Uno, S.; Suzuki, Y.; Raymond, R.; Onodera, T.; Tanaka, T.; Tezuka, H.; Mitsuda, N.; Yamamoto, N. Approximate amplitude encoding in shallow parameterized quantum circuits and its application to financial market indicator. *arXiv* **2021**, arXiv:quant-ph/2103.13211.
13. Shende, V.; Bullock, S.; Markov, I. Synthesis of quantum-logic circuits. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2006**, *25*, 1000–1010. [[CrossRef](#)]
14. Herbert, S. No quantum speedup with Grover-Rudolph state preparation for quantum Monte Carlo integration. *Phys. Rev. E* **2021**, *103*, 063302. [[CrossRef](#)] [[PubMed](#)]
15. Brugière, T. Methods for Optimizing the Synthesis of Quantum Circuits. Ph.D. Thesis, Université Paris-Saclay, Gif-sur-Yvette, France, 2020.
16. Mottonen, M.; Vartiainen, J.J. Decompositions of general quantum gates. *arXiv* **2005**, arXiv:quant-ph/0504100.
17. Plesch, M.; Brukner, C. Quantum-state preparation with universal gate decompositions. *Phys. Rev. A* **2011**, *83*, 032302. [[CrossRef](#)]
18. Pathak, A. *Elements of Quantum Computation and Quantum Communication*, 1st ed.; Taylor & Francis, Inc.: London, UK, 2013.
19. Childs, A.M.; Wiebe, N. Hamiltonian Simulation Using Linear Combinations of Unitary Operations. *Quantum Info. Comput.* **2012**, *12*, 901–924. [[CrossRef](#)]
20. Shukla, A.; Vedula, P. A hybrid classical-quantum algorithm for solution of nonlinear ordinary differential equations. *arXiv* **2021**, arXiv:quant-ph/2112.00602.
21. Li, X.; Yang, G.; Torres, C.M.; Zheng, D.; Wang, K.L. A class of efficient quantum incrementer gates for quantum circuit synthesis. *Int. J. Mod. Phys. B* **2014**, *28*, 1350191. [[CrossRef](#)]
22. Nielsen, M.A.; Chuang, I.L. Quantum computation and quantum information. *Phys. Today* **2001**, *54*, 60.
23. Kitaev, A.Y. Quantum measurements and the Abelian stabilizer problem. *arXiv* **1995**, arXiv:quant-ph/9511026.
24. Brassard, G.; Høyer, P.; Mosca, M.; Tapp, A. Quantum amplitude amplification and estimation. *Quantum Comput. Inf.* **2002**, *305*, 53–74. [[CrossRef](#)]
25. Wie, C.R. Simpler quantum counting. *arXiv* **2019**, arXiv:1907.08119.
26. Grinko, D.; Gacon, J.; Zoufal, C.; Woerner, S. Iterative quantum amplitude estimation. *NPJ Quantum Inf.* **2021**, *7*, 52. [[CrossRef](#)]
27. Suzuki, Y.; Uno, S.; Raymond, R.; Tanaka, T.; Onodera, T.; Yamamoto, N. Amplitude estimation without phase estimation. *Quantum Inf. Process.* **2020**, *19*, 75. [[CrossRef](#)]
28. Manzano, A.; Musso, D.; Leitao, A.; Gómez, A.; Vázquez, C.; Ordóñez, G.; Nogueiras, M. Real Quantum Amplitude Estimation. 2022. (In progress).