



# Report on Quantum Computing exploratory research

Samuele Pino

Project for Software Engineering 2, Politecnico di Milano

June 18, 2019

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Abstract . . . . .	4
1.2	Quantum circuits . . . . .	5
1.2.1	Matrix representation of circuits . . . . .	5
1.2.2	Some elementary gates . . . . .	6
1.2.3	Quantum gates in series . . . . .	6
1.2.4	Quantum gates in parallel . . . . .	7
1.2.5	Controlled gates . . . . .	8
<b>2</b>	<b>Microsoft Quantum Development Kit</b>	<b>10</b>
2.1	Software overview . . . . .	11
2.1.1	Installation . . . . .	11
2.1.2	Documentation . . . . .	11
2.1.3	Language Q# . . . . .	11
2.1.4	Simulator . . . . .	12
2.2	Sample Q# code: Grover Search . . . . .	13
2.2.1	The code . . . . .	14
2.2.2	Results of multiple runs of the code . . . . .	16
2.2.3	Possible improvements on this implementation . . . . .	16

<b>3</b>	<b>Insights on GSA and its implementation</b>	<b>20</b>
3.1	The problem . . . . .	20
3.2	The phone book implementation . . . . .	21
3.3	Permutation of rows in a matrix . . . . .	22
3.3.1	Simplified case: 2 qubits . . . . .	23
3.3.2	General case: shift and control . . . . .	24
3.3.3	General case: sorting algorithms . . . . .	26
3.4	Feasibility . . . . .	27
<b>4</b>	<b>Implementation of an entangled database for GSA</b>	<b>28</b>
4.1	Octave . . . . .	28
4.2	Q# . . . . .	31
<b>5</b>	<b>Other minor aspects explored during the research</b>	<b>36</b>
<b>6</b>	<b>Conclusions</b>	<b>37</b>
	<b>Appendices</b>	<b>38</b>
<b>A</b>	<b>Quantum gates in Octave</b>	<b>39</b>
A.1	Elementary (existing) gates . . . . .	39
A.1.1	Hadamard and X, Y, Z . . . . .	39
A.1.2	CNOT . . . . .	40
A.1.3	ICNOT: inverted CNOT . . . . .	40
A.1.4	SWAP . . . . .	41
A.1.5	CCNOT . . . . .	41
A.1.6	CSWAP . . . . .	42
A.2	Operations between gates . . . . .	42
A.2.1	Kronecker product (or direct product) . . . . .	42

A.2.2	Gate control (direct sum)	43
A.3	New (derivated) gates	43
A.3.1	DSWAP	43
A.3.2	SHIFT	44
A.3.3	QSD: Quarter Shift Down	44
A.3.4	QSU: Quarter Shift Up	45
A.3.5	CNOT3	45
A.3.6	SWAP3	46
A.3.7	SHIFT3	47
A.3.8	CNOT4	47

# Chapter 1

## Introduction

The present document is a report of the exploration work performed in the quantum computing field by the author.

### 1.1 Abstract

Quantum computing is a topic that has gained some ground in the scientific community over the last decades. It promises polynomial or even exponential speedups for certain applications. In Chapter 1 we provide the reader with some useful background on quantum computing theory. In Chapter 2 we present Quantum Development Kit, a tool for writing and compiling quantum programs in the specific language Q# and some examples. In Chapter 3 we explore some deep implementation details of a quantum entangled state database, useful for the correct performing of a Grover's search, while in Chapter 4 we present a sample implementation of what explained in the previous chapter. Finally, in Chapter 5 we mention other minor aspects explored during the research process.

## 1.2 Quantum circuits

We assume the reader already knows the basic concepts about linear algebra and elementary quantum gates. For more information on these basic topics, we recommend [10, 15].

Here we propose a fast walk-through of some possible compositions of quantum gates in the context of a circuit. For further explanations and a more complete reading on the topic we recommend [15, p. 123–129].

### 1.2.1 Matrix representation of circuits

Operations that make sense in quantum computing are usually performed on more than 2 or 3 qubits and they often give as an output multiple qubits as well. Such computations can be performed by long and complex circuits, therefore we need to be able to decompose them into a sequence of simpler quantum gates. A circuit can be represented by a unitary matrix, which can mathematically describe the operations performed on an array of input qubits.

Consider a qubit array  $|b\rangle = [b_0, b_1]^T$  where  $b_0$  and  $b_1$  are respectively the most and least significative qubits. Therefore a unitary matrix  $U$  applied on a  $|b\rangle$  will have the following representation:

$$\begin{array}{l} b_0 : \text{---} \\ b_1 : \text{---} \end{array} \begin{array}{c} \boxed{U} \\ \text{---} \end{array} \begin{array}{l} \text{---} \\ \text{---} \end{array} \qquad U |b\rangle = \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}$$

where on the right side we have a standard matrix multiplication. Please note that the order of this product is important, as it is *not commutative*.

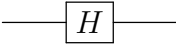
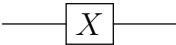
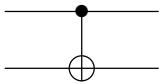
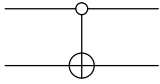
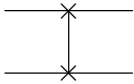
Hadamard gate		$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
NOT gate		$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
CNOT (on $b_0 = 1$ )		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
CNOT (on $b_0 = 0$ )		$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
SWAP gate		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Table 1.1: Some elementary quantum gates in circuit and matrix representation.

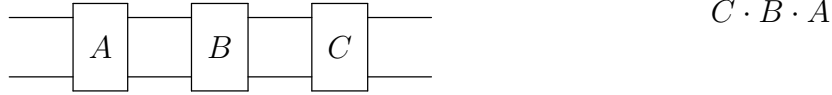
### 1.2.2 Some elementary gates

A computation usually requires a combination of elementary gates in 3 main ways: sequentially, in parallel or conditionally. In Table 1.1 we show some gates (and their matrix form) that will be useful for the rest of this document.

### 1.2.3 Quantum gates in series

The series of quantum gates applied on a qubit line (or on a subset of qubit lines) in a circuit is equivalent to the *dot product* between the matrices of

each gate in reverse order.



Matrices A, B and C must be of the same size (in this example, being applied on 2 bits, they must be  $4 \times 4$ ). The result matrix will be obviously of the same size of A, B and C ( $4 \times 4$ ).

#### 1.2.4 Quantum gates in parallel

Applying distinct quantum gates to disjoint subsets of qubits is equivalent to the *direct product* (or *tensor product*, or *kroncker product*) between the matrices of each gate. Here the order is given by the position of the qubits to which gates are applied (most significative first).



Given  $A \in M^{m \times m}$  and  $B \in M^{n \times n}$ , the result matrix will be  $mn \times mn$  dimensional. We can easily notice that the matrix dimension grows fast with consecutive applications of direct product and the resulting dimension is always a power of 2 in quantum circuits.

A special case is when some qubits have a gate applied, while others have nothing (that is equivalent to an identity matrix).



$$\begin{array}{l}
b_0 : \text{---} \boxed{X} \text{---} \\
b_1 : \text{---}
\end{array}
\quad
X \otimes I_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

### 1.2.5 Controlled gates

The effect of a gate on a subset of qubits (“targets”) can be applied conditionally to the value of one or more other qubits (called “controls”). This operation is equivalent to a *direct sum* between matrices.

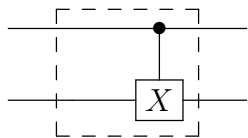
$$\begin{array}{c}
\text{---} \circ \text{---} \\
\quad | \\
\boxed{A} \\
\text{---}
\end{array}
\quad
\begin{array}{c}
\text{---} \bullet \text{---} \\
\quad | \\
\boxed{B} \\
\text{---}
\end{array}
\quad
A \oplus B = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 0 & b_{11} & b_{12} \\ 0 & 0 & b_{21} & b_{22} \end{bmatrix}$$

in this example  $A \oplus B$  means that gate  $A$  is applied to the bottom qubit if the top one is in state  $|0\rangle$ , while  $B$  is applied if the top qubit is in state  $|1\rangle$ . It can be easily generalized to the case of 2 control qubits:

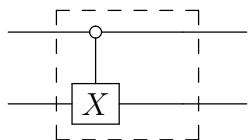
$$\begin{array}{cccc}
\text{---} \circ & \text{---} \circ & \text{---} \bullet & \text{---} \bullet \\
\quad | & \quad | & \quad | & \quad | \\
\text{---} \circ & \text{---} \bullet & \text{---} \circ & \text{---} \bullet \\
\quad | & \quad | & \quad | & \quad | \\
\boxed{A} & \boxed{B} & \boxed{C} & \boxed{D}
\end{array}
\quad
A \oplus B \oplus C \oplus D = \begin{bmatrix} A & 0 & 0 & 0 \\ 0 & B & 0 & 0 \\ 0 & 0 & C & 0 \\ 0 & 0 & 0 & D \end{bmatrix}$$

in general if we have  $n_c$  control lines and  $n_t$  target lines, we can obtain a direct sum of up to  $2^{n_c}$  gates, each gate of dimension  $2^{n_t}$ . If we have less than  $2^{n_c}$  gates to control, the missing spots in the direct sum are filled by appropriate

sized identity matrices:



$$I \oplus X = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$



$$X \oplus I = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## Chapter 2

# Microsoft Quantum Development Kit

QDK is Microsoft's open source development kit for quantum computing. It is quite young, as it has been released in January 2018. Despite that, it has some interesting characteristics, like the use of a specific “quantum focused” language: Q#. Differently from other frameworks (like pyQuil, QISKit, ProjectQ...), QDK uses a technology based on Majorana fermions, which is also the reason why right now there are no hardware devices on which run the algorithms. [11]

It is updated very frequently with often drastic improvement in terms of usability and bug fixes. Of course it is not yet a mature environment, as it is still under development.

## 2.1 Software overview

### 2.1.1 Installation

Microsoft QDK can be installed on top of Visual Studio or Visual Studio Code (recommended). The setup is easy and the process it's well explained in the official page: <https://docs.microsoft.com/it-it/quantum/install-guide/vs-2017>.

After installation it is also possible to validate its correctness running a sample program whose aim is to check the possible absence of needed packages (linke NuGet).

### 2.1.2 Documentation

A complete documentation of the software and language can be found on the official website <https://docs.microsoft.com/it-it/quantum>. It contains tutorials on how to run the first quantum program, info about the simulator, Q# syntax and its libraries. Other part of the website also contain a good documentation on the theory about quantum computing.

Moreover the open source libraries are useful to learn the language. Last but not least, there is a vast number of verbosely commented examples (Teleportation, Grover's Search, Integer factorization, simulations...).

### 2.1.3 Language Q#

Using QDK requires some basic knowledge of C# for the classical host computation, usually contained in a "driver.cs" file, that is used to call the simulator with the quantum program, optionally providing inputs.

The quantum part of the program uses Q# that, despite the name, is more

similar to a hardware description language than to an object oriented one. We can define *operations*, callable routines with quantum instructions, that as functions take some input and return an output value. We can also define variables to values bindings (like integers and booleans), perform operations on single qubits (like gates, conditionals and controls).

In general the language is high level oriented: you do not have to design the spatial disposition of gates on qubit lines as you are not bound to a specific architecture, therefore the programmer can focus more on the algorithm than on the implementation details, thanks also to the available libraries.

#### **2.1.4 Simulator**

QDK can be used within a local run of Visual Studio, in this case it can simulate circuits of up to 30 qubits. If more power is needed, it can also be run in Microsoft Azure cloud (through a paid subscription) achieving simulations of more than 40 qubits.

It uses a locally deployed simulation environment based on dotnet. The language abstracts from the actual architecture to be deployed (it uses Qubit objects, not specific low level registers), in order to allow an better re-usability and portability of the code.

It also implements a Toffoli simulator, a special-purpose simulator for quantum algorithms that are limited to X, CNOT, and multi-controlled X.

A trace simulator is also provided. It is useful for debugging classical code and estimating the resources required to run a given instance of a quantum program. Circuits of thousands of qubits can be tested, as the trace simulator executes the program without simulating the state of the quantum computer.

## Hardware and noise analysis

The technology Microsoft is trying to use has not been implemented on hardware yet. Moreover QDK does not provide any functionality for noise analysis or simulation. This is probably connected to the fact that Microsoft is betting on topological qubits, that should be highly resilient to noise and decoherence.

## 2.2 Sample Q# code: Grover Search

QDK comes with a bunch of code samples to help the user understanding the basics of the language and its potentialities. Among these examples we want to mention *qubits teleportation*, *CHSH Game*, *Grover's Algorithm* (called Database Search) and Integer factorization.

In this document we are not going to describe the details of Grover's Algorithm in function, as it is not our purpose (for this please refer to [9]). However it's instructive to read some examples of Q# code, so let's take a quick look at Grover's Algorithm implementation.

```
/// This sample will walk through several examples of searching a database
/// of N elements for a particular marked item using just  $O(1/\sqrt{N})$  queries
/// to the database. [...] We will model the database by an oracle D that
/// acts to map indices to a flag indicating whether a given index is marked
/// .
/// [...]
/// (it) makes full use of the amplitude amplification library and other
/// supporting libraries to implement Grover's algorithm more easily. We
/// also consider a more general instance of the database oracle that allows
/// us to mark multiple elements.
```

For a simpler and less theory requiring example of Q# code see also Section 4.2.

## 2.2.1 The code

```
// Copyright (c) Microsoft Corporation. All rights reserved.  
// Licensed under the MIT License.
```

First an oracle  $D$  is built from the classical database

```
operation DatabaseOracleFromInts (markedElements : Int[], markedQubit :  
    Qubit, databaseRegister : Qubit[]) : Unit {  
  
    body (...) {  
        let nMarked = Length(markedElements);  
  
        for (idxMarked in 0 .. nMarked - 1) {  
            (ControlledOnInt(markedElements[idxMarked], ApplyToEachCA(X, -)))(  
                databaseRegister, [markedQubit]);  
        }  
    }  
  
    adjoint invert;  
    controlled distribute;  
    controlled adjoint distribute;  
}
```

Then we prepare the state oracle

```
operation GroverStatePrepOracleImpl (markedElements : Int[], idxMarkedQubit  
    : Int, startQubits : Qubit[]) : Unit {  
  
    body (...) {  
        let flagQubit = startQubits[idxMarkedQubit];  
        let databaseRegister = Exclude([idxMarkedQubit], startQubits);  
  
        // Apply oracle 'U'  
        ApplyToEachCA(H, databaseRegister);  
  
        // Apply oracle 'D'  
        DatabaseOracleFromInts(markedElements, flagQubit, databaseRegister);  
    }  
  
    adjoint invert;  
    controlled distribute;  
    controlled adjoint distribute;
```

```
|}
```

Finally the library function ‘AmpAmpByOracle’ returns a unitary that implements all steps of Grover’s algorithm.

```
function GroverSearch (markedElements : Int[], nIterations : Int,
    idxMarkedQubit : Int) : (Qubit[] => Unit : Adjoint, Controlled) {

    return AmpAmpByOracle(nIterations, GroverStatePrepOracle(markedElements),
        idxMarkedQubit);
}
```

Putting all together

```
operation ApplyGroverSearch (markedElements : Int[], nIterations : Int,
    nDatabaseQubits : Int) : (Result, Int) {

    // Allocate variables to store measurement results.
    mutable resultSuccess = Zero;
    mutable numberElement = 0;

    // Allocate nDatabaseQubits + 1 qubits. These are all in the |0
    // state.
    using (qubits = Qubit[nDatabaseQubits + 1]) {

        // Define marked qubit to be indexed by 0.
        let markedQubit = qubits[0];

        // Let all other qubits be the database register.
        let databaseRegister = qubits[1 .. nDatabaseQubits];

        // Implement the quantum search algorithm.
        (GroverSearch(markedElements, nIterations, 0))(qubits);

        // Measure the marked qubit. On success, this should be One.
        set resultSuccess = M(markedQubit);

        // Measure the state of the database register post-selected on
        // the state of the marked qubit.
        let resultElement = MultiM(databaseRegister);
        set numberElement = PositiveIntFromResultArr(resultElement);
    }
}
```



```

    // These reset all qubits to the |0 state, which is required
    // before deallocation.
    ResetAll(qubits);
}

// Returns the measurement results of the algorithm.
return (resultSuccess, numberElement);
}

```

### 2.2.2 Results of multiple runs of the code

We performed some runs of the algorithm, shaping the output so that it could be easy intelligible. In particular, to show the potentialities of Grover's Search and the theoretical speedup with respect to the classical one we collected stats on:

- classical search of one element in Figure 2.1
- quantum search of one element in Figure 2.2
- quantum search of multiple elements in Figure 2.1

### 2.2.3 Possible improvements on this implementation

In the above code a standard version of Grover's algorithm is discussed. The whole program takes as input only the elements to search, but there is no information about the elements present in the database. In fact the database is implemented through a register of  $n$  qubits, that are initialized so that when measured all the values from 0 to  $2^n - 1$  have the same flat probability. In this way the register is actually implementing a virtual database, a mathematical object that has no relevance in real cases as it contains no information. We will focus on this issue in Chapter 3.

```
Classical search for marked element in database.
Database size: 1024.
Marked elements: 13 Classical success probability: 0,0009765625

    Attempt 1000: Fail. Found database index 226
Up to now success frequency: 0,000999

    Attempt 2000: Fail. Found database index 0
Up to now success frequency: 0,001

    Attempt 3000: Fail. Found database index 351
Up to now success frequency: 0,000666

    Attempt 4000: Fail. Found database index 1023
Up to now success frequency: 0,0005

    Attempt 5000: Fail. Found database index 743
Up to now success frequency: 0,0006

    Attempt 6000: Fail. Found database index 817
Up to now success frequency: 0,000667

    Attempt 7000: Fail. Found database index 343
Up to now success frequency: 0,000571

    Attempt 8000: Fail. Found database index 619
Up to now success frequency: 0,00075

    Attempt 9000: Fail. Found database index 899
Up to now success frequency: 0,000778

    Attempt 10000: Fail. Found database index 653
Up to now success frequency: 0,0009
```

Figure 2.1: Classical search on a database. The search complexity is  $O(N)$ , in fact the probability of picking a the correct element by random choice is  $1/N$ . In this example the database contains 1024 elements, we look for element 13 and we perform 10 thousands attempts.

```

Quantum search for marked element in database.
Database size: 1024.
Marked elements: 13 Classical success probability: 0,0009765625
Oracle queries per search: 7
Quantum success probability: 0,047108250571215

    Attempt 100: Fail. Found database index 979
Up to now success frequency: 0,009901 => Speedup: 1,448

    Attempt 200: Fail. Found database index 100
Up to now success frequency: 0,029851 => Speedup: 4,367

    Attempt 300: Fail. Found database index 348
Up to now success frequency: 0,033223 => Speedup: 4,86

    Attempt 400: Fail. Found database index 691
Up to now success frequency: 0,047382 => Speedup: 6,931

    Attempt 500: Fail. Found database index 72
Up to now success frequency: 0,045908 => Speedup: 6,716

    Attempt 600: Fail. Found database index 937
Up to now success frequency: 0,043261 => Speedup: 6,328

    Attempt 700: Fail. Found database index 243
Up to now success frequency: 0,045649 => Speedup: 6,678

    Attempt 800: Fail. Found database index 102
Up to now success frequency: 0,042447 => Speedup: 6,209

    Attempt 900: Fail. Found database index 147
Up to now success frequency: 0,043285 => Speedup: 6,332

    Attempt 1000: Fail. Found database index 871
Up to now success frequency: 0,044955 => Speedup: 6,576

```

Figure 2.2: Quantum search on a database. The search complexity is  $O(\sqrt{N})$ . The quantum success probability is predicted to be higher than the classical one, and the statistics confirm the theory as computed during the attempts. In this example the database contains 1024 elements, we look for element 13 and we perform 1 thousand attempts (less than classical ones because the simulation of Oracle queries carries some overhead).

```

Quantum search for marked element in database.
Database size: 1024.
Marked elements: 0,15,48,62 Classical success probability: 0,00390625
Oracle queries per search: 7
Quantum success probability: 0,179720628257257

    Attempt 25: Fail. Found database index 142
Up to now success frequency: 0,192308 => Speedup: 7,033

    Attempt 50: Fail. Found database index 940
Up to now success frequency: 0,137255 => Speedup: 5,02

    Attempt 75: Success. Found database index 48
Up to now success frequency: 0,171053 => Speedup: 6,256

    Attempt 100: Fail. Found database index 814
Up to now success frequency: 0,178218 => Speedup: 6,518

    Attempt 125: Fail. Found database index 416
Up to now success frequency: 0,190476 => Speedup: 6,966

    Attempt 150: Fail. Found database index 296
Up to now success frequency: 0,18543 => Speedup: 6,781

    Attempt 175: Fail. Found database index 333
Up to now success frequency: 0,1875 => Speedup: 6,857

    Attempt 200: Success. Found database index 15
Up to now success frequency: 0,179104 => Speedup: 6,55

    Attempt 225: Fail. Found database index 79
Up to now success frequency: 0,181416 => Speedup: 6,635

    Attempt 250: Fail. Found database index 167
Up to now success frequency: 0,167331 => Speedup: 6,12

```

Figure 2.3: Again quantum search on a database. The database contains 1024 elements, but this time we look for 4 different elements at the same time. As the simulation gets proportionally heavier, this time we perform 1/4 of the previous attempts, bt enough to confirm the matching between the theoretical success probability and the simulated one.

## Chapter 3

# Insights on GSA and its implementation

Many quantum algorithms base their reason to exist in the fact that a small routine of the them (e.g. the search for the next arc to be considered among those coming out of a given node, in Max Flow Analysis) is done by a quantum computer. It is usually nothing more than a search in a list (or more generally in a database) of one or more elements that satisfy a certain condition (the arcs that have not been visited yet, i.e. having infinite weight value).

### 3.1 The problem

Let's start noticing that although we have presented in Section 2.2 an implementation of Grover Search in  $Q\#$ , it actually works on what is known as “virtual database”. Alike real databases, virtual (or implicit) ones are not really databases: given  $n$  as the number of bits, they are nothing more than the set of integer numbers  $[0, 2^n - 1]$ .

Such a “database” can be easily implemented with a quantum register initialized with  $H^{\oplus n}$ . In this way, whenever the register is measured, it collapses to one of all the possible combinations of its bits (i.e.  $[0, 2^n - 1]$ ), being all these combinations all equally probable.

Actually the implementation described in Section 2.2 complies with most of the available literature [9, 12]. It is evident that currently most of the works somehow related to Grover Search Algorithm are devoted to quantum search on virtual databases. [7]

Apparently some people agree that Grover is limited to implicit databases, therefore not convenient or even not useful at all for real databases [14, 16, 3, 1, 2]. On the other hand, someone had a deeper study on the algorithm, understanding the mechanism and implementing (at least mathematically) the encoding and the search on a real database. [5]

### 3.2 The phone book implementation

The work [5] actually finds a way to encode some elements into a real database. It is done setting a register to an entangled state, as sum of the states corresponding to the elements that we want to encode into the database. This database-register is created by applying a particular matrix to it, in which the database elements are encoded within the rows order. An example is shown in Figure 3.1.

Calling  $n$  the number of bits of the primary key (i.e. the contact *name*) and  $m$  the number of bits of the data field (i.e. the contact *number*), the square matrix  $A'$  will have a size of  $K = 2^n \cdot 2^m = 2^{n+m}$  rows. Matrix  $A'$  can be obtained as a direct sum  $H^{\otimes n} \oplus I_{K-2^n}$  (see Section 1.2.5).

Figure 3.1 shows the transformation of matrix  $A'$  into matrix  $A$  through a series of row swaps. Matrix  $A'$  is a 16x16 matrix with a block structure. Matrix  $A$  is the result of applying a sequence of swap operators  $S_{i',j_i}$  to  $A'$ . The sequence is given by the equation:

$$A = \prod_{i=0}^{N-1} S_{i',j_i} A' = S_{0',2} S_{1',7} S_{2',8} S_{3',13} A'$$

Figure 3.1: Successive row swapping operations to transform  $A'$  to  $A$  in for the specific telephone database example. (credits to [5])

Matrix  $A$  can be obtained by applying to  $A'$  a series of swap operators  $S_{ij}$  that perform a swapping between rows  $i$  and  $j$  of a matrix. This operation is not described in the proposed paper, therefore we will try to give an algorithm to perform it (see Section 3.3). This is the key passage that let us prepare an entangled register, ready to be used for the subsequent Grover iterations, as shown in the mentioned work.

This is a remarkable result, as it demonstrates the theoretical consistency of Grover's Algorithm for searching purposes. Critics can be raised against the performance or the convenience of the entire process with respect to the classical one, but these topics have already been discussed elsewhere [14].

### 3.3 Permutation of rows in a matrix

The main issue that we want to address now is how to perform an arbitrary permutation of the rows of a “quantum” matrix. This is a fundamental

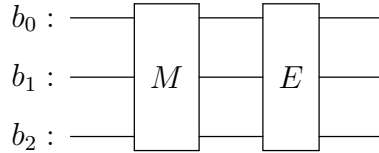
algorithm passage for the correct implementation of Grover iteration. Despite that, we were not able to find any hint in literature on how to perform such permutations, so here we present some ideas that can be a starting point for a future improved and more general solution of the problem.

As it is well known in linear algebra, given a square matrix  $M$  we can obtain  $M'$  (a version of it where  $i$ -th and  $j$ -th rows are swapped) multiplying  $M$  by a matrix  $E$ , where  $E$  is the identity with  $i$ -th and  $j$ -th rows swapped:

$$EM = M'$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{41} & a_{42} & a_{43} & a_{44} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix}$$

Therefore our problem is to find a circuit that implements matrix  $E$ . This technique is consistent with the fact that a circuit applied on an array of qubit can be represented with a matrix multiplying the vector from the left side. Multiplying  $E$  from the left of  $M$  is equivalent to placing the circuit of  $E$  after (on the right of) the circuit of  $M$ .



### 3.3.1 Simplified case: 2 qubits

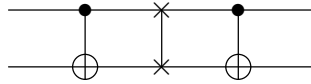
If we have a circuit of only 2 qubits, swapping 2 rows can be relatively easy, as shown in Table 3.1. For example:



Rows to swap	Matrices to apply
1, 2	ICNOT ( <i>NOT gate</i> controlled on first qubit = 0)
1, 3	SWAP · ICNOT · SWAP
1, 4	CNOT · SWAP · ICNOT · SWAP · CNOT
2, 3	SWAP
2, 4	CNOT · SWAP · CNOT
3, 4	CNOT

Table 3.1: Combinations of rows that one could want to swap and the possible combination of 2-qubits gates to achieve it.

$$CNOT \cdot SWAP \cdot CNOT = swap(2, 4)$$



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

### 3.3.2 General case: shift and control

A possible extension of the algorithm to the case of  $n$  qubits would make an intensive use of controlled gates. Control through a qubit equal to 1 is equivalent to the direct sum of an identity matrix and the controlled gate itself. This means that, if we can control a gate, we are able to replicate the behavior of its  $4 \times 4$  matrix in the bottom half of a larger  $8 \times 8$  matrix (in terms of rows swapping). It could be interesting if we could temporarily “move down” the rows of a big matrix, perform our swaps and then move them up again. To easier describe this process we will make some definitions.

*Definition.* Let  $N$  denote the number of rows in the matrix representing an operation  $G$ . The number  $N$  is obviously  $2^n$ , where  $n$  is the number of bits on which gate  $G$  is applied. Let's consider only a gate  $G$  which is decomposable as a direct product of a matrix  $M$  and an identity matrix  $I_k$ . We will call  $k$  the *grade* of operator  $G$ .

Example:

- CNOT and SWAP have *grade* 0
- CNOT3, SWAP3 and SHIFT3 (Section A.3) have *grade* 1
- SHIFT4 (Section A.3) has *grade* 2
- CCNOT has grade 0

We can easily increase by  $h$  the degree of a matrix  $G$  by performing  $kron(G, I_h)$ . This is equivalent, in a register large  $n + h$  qubits, to apply  $G$  to the first  $n$  qubits and nothing to the remaining  $h$ .

## Algorithm

Let's take as an example the problem of permuting rows of an  $8 \times 8$  matrix.

Using CNOT3, SWAP3 and ICNOT3 we can exchange matrix macroblocks (blocks of 2 contiguous lines). We can then operate on the two blocks (each  $2 \times 8$ ) of the lower half-matrix using the CCNOT, CSWAP and controlled ICNOT ports, with granularity of the individual rows. Please note that the CSWAP allows us to exchange two rows of two different blocks ( $2 \times 8$ ), this can be useful in the generalization to more qubits. The same algorithm can be used for  $16 \times 16$  matrices, increasing by one the degree of all previous ports and adding one more bit of control to the existing port (thus obtaining CCCNOT, CCICNOT, CCSWAP...).

Useful gates to perform the shift are SHIFT, QSD and QSU gates, together with their higher grade versions (Section A.3).

### Open issues

Probably the addition of new control qubits at each step of generalization implies an exponential growth in spatial complexity of the circuit.

### 3.3.3 General case: sorting algorithms

This approach, instead of focusing on single row swaps, treats the permutation from initial to final matrix as a single process. You can see the analogy with sorting algorithms applied to arrays (bubble sort, merge sort...).

If there was a way to swap 2 consecutive rows of a matrix, regardless of their position, the problem would be easily solved. In that case we could apply bubble sort as an algorithm to rearrange all the rows as we like.

The only general way that we could devise to exchange consecutive rows is to use  $X$  gates in direct sum with  $I_2$  matrices all over the diagonal. The main drawback of this method is that this configuration is only able to swap rows  $2i + 1$  and  $2i + 2$ , with  $i \geq 0$ . Therefore if we want to swap for example rows 2 and 3 we need to use a SWAP gate in direct sum with an appropriate number of  $I$  matrices down the diagonal. The problem in using SWAP in this configuration is that it works only for swapping rows  $4i + 2$  and  $4i + 3$ , with  $i \geq 0$ . Therefore if we want to swap rows 4 and 5 we need a new different gate (possibly  $8 \times 8$  or bigger) and so on.

### 3.4 Feasibility

Complexity studies on quantum circuits are still under development, however we can say something about the circuits we proposed in this chapter. As the reader can infer from the rapidly increasing size of matrices, the problem of preparing an entangled database seems to grow exponentially with the number of qubits in which the data is encoded. If we use the number of row swaps as a unit of measure for the complexity of this process, we can notice that it is in some way proportional to the number of elements in the database we want to encode and to the number of rows. If we assume constant (or linearly growing with the qubits) the number of needed gates to perform a swap of two near rows, the most relevant contribute to the spatial complexity of gates is still exponential in the number of qubits.

This is probably a less important problem if the algorithm user's main focus is on the search performances rather than on the preparation overhead. However, in our opinion, this approach could weaken Grover's "quantum speedup", that is the main point of the entire quantum computing research field. At the moment we could not find any efficient (less than exponentially complex) way to perform a real implementation of Grover Search and we hope that more in-depth researches will be done on this topic.

## Chapter 4

# Implementation of an entangled database for GSA

In order to give a touch of reality to our research work we would like to present an implementation of the algorithm described in Section 3.3.2. We first provide an implementation in Octave, that simulates the effects of quantum operations through matrix calculation and provides a clear view of the matrices shape. Then we provide an implementation in Q# through Microsoft QDK.

### 4.1 Octave

In this chapter we will implement several operations in Octave, both common ones and ad hoc defined ones. The reader can find the definition of all new introduced gates and operations in Section A.3.

In the following we show first a commented version of the output of the algorithm and then the source code.

Listing 4.1: Entangled DB implemented in Octave (output only)

```

# Format of the database register:
[ n1 | t1 t2 ]
# 1 bit for the name
# 2 bit for the telephone number

# all possible name values
N1 = 2
# all possible telephone number values
N2 = 4
# initialization hadamard matrix. Size: N1xN1
ans =

    0.71    0.71
    0.71   -0.71

# initialization of primed matrix as direct sum. Size: (N1*N2)x(N1*N2)
primed_m =

    0.71    0.71    0.00    0.00    0.00    0.00    0.00    0.00
    0.71   -0.71    0.00    0.00    0.00    0.00    0.00    0.00
    0.00    0.00    1.00    0.00    0.00    0.00    0.00    0.00
    0.00    0.00    0.00    1.00    0.00    0.00    0.00    0.00
    0.00    0.00    0.00    0.00    1.00    0.00    0.00    0.00
    0.00    0.00    0.00    0.00    0.00    1.00    0.00    0.00
    0.00    0.00    0.00    0.00    0.00    0.00    1.00    0.00
    0.00    0.00    0.00    0.00    0.00    0.00    0.00    1.00

# exchange rows
# we want to encode a database with the following values for name and
    telephone
    -----
    | value | name | tel |
    | ----- | ----- | ---- |
    | 0-11(3) | 0 | 3 |
    | 1-10(6) | 1 | 2 |
    | ----- | ----- | ---- |
# exchange rows 1 and 3
m =

    0.00    0.00    1.00    0.00    0.00    0.00    0.00    0.00
    0.71   -0.71    0.00    0.00    0.00    0.00    0.00    0.00

```

```

    0.71    0.71    0.00    0.00    0.00    0.00    0.00    0.00
    0.00    0.00    0.00    1.00    0.00    0.00    0.00    0.00
    0.00    0.00    0.00    0.00    1.00    0.00    0.00    0.00
    0.00    0.00    0.00    0.00    0.00    1.00    0.00    0.00
    0.00    0.00    0.00    0.00    0.00    0.00    1.00    0.00
    0.00    0.00    0.00    0.00    0.00    0.00    0.00    1.00

# now exchange rows 2 and 6
m =

    0.00    0.00    1.00    0.00    0.00    0.00    0.00    0.00
    0.00    0.00    0.00    0.00    0.00    1.00    0.00    0.00
    0.71    0.71    0.00    0.00    0.00    0.00    0.00    0.00
    0.00    0.00    0.00    1.00    0.00    0.00    0.00    0.00
    0.00    0.00    0.00    0.00    1.00    0.00    0.00    0.00
    0.71   -0.71    0.00    0.00    0.00    0.00    0.00    0.00
    0.00    0.00    0.00    0.00    0.00    0.00    1.00    0.00
    0.00    0.00    0.00    0.00    0.00    0.00    0.00    1.00

done

```

Octave script (some log commands are omitted for brevity):

Listing 4.2: Entangled DB implemented in Octave (code)

```

display("#_Format_of_the_database_register:")
n1 = 1;      # 1 bit for the name
n2 = 2;      # 2 bit for the telephone number

N1 = 2^n1
N2 = 2^n2

display("#_initialization_hadamard_matrix._Size:_N1xN1")
init_H = H;
for i = 1:(n1-1)
    init_H = kron(init_H, H); # note that in this case order of kron
                             parameters does not matter
endfor
H

display("#_initialization_of_primed_matrix_as_direct_sum._Size:_(N1*N2)x(N1*
    N2)")

```

```

primed_m = init_H;
for i = 1:n2
    primed_m = IC(primed_m);
endfor
primed_m

display("#_exchange_rows")

m = primed_m;

display("#_exchange_rows_1_and_3")
m = SHIFT3 * m;
m = C(switch_g(1,3)) * m;
m = SHIFT3 * m;
m

display("#_now_exchange_rows_2_and_6")
m = kron(switch_g(1,4),eye(2)) * m;
m = C(switch_g(2,4)) * m;
m = kron(switch_g(1,4),eye(2)) * m;
m

display("done")

```

where *switch\_g()* implements the swapping gates described in Table 3.1.

## 4.2 Q#

Now we would like to implement in a quantum development environment some parts of the Octave code shown in Section 4.1, we will use Microsoft QDK (described in Chapter 2) and the programming language will be Q#.

First let's define an operation to set qubits' state to a desired one.

```

operation Set(desired: Result, q1: Qubit): Unit {
    let current = M(q1);
    if (desired != current) {
        X(q1);
    }
}

```



```
|    }
```

Then let's define a couple of custom gates, like we have done in Octave

```
operation SHIFT(q : Qubit[]) : Unit {
    body(...) {
        X(q[0]);
    }
    controlled auto;
}

operation CSHIFT(q : Qubit[]) : Unit {
    (Controlled SHIFT)([q[0]], [q[1], q[2]]);
}

operation ICNOT(q : Qubit[]) : Unit {
    body(...) {
        X(q[0]); // assuming control qubit is |0>, flip it
        (Controlled X)([q[0]], q[1]);
        X(q[0]); // restore original state of control qubit
    }
    controlled auto;
}
```

And now let's define an useful operator that performs control on *ctrl\_qubits=0*

```
// inverse control on gates that get 1 qubit
operation IC_1q(controls : Qubit[], gate : (Qubit=>Unit is Ctl),
    gate_qubit : Qubit) : Unit {
    ApplyToEach(X, controls); // assuming control qubit is |0>, flip it
    (Controlled gate)(controls, gate_qubit);
    ApplyToEach(X, controls); // restore original state of control qubit
}
```

Finally we can put everything together and implement the Octave code seen in Listing 4.2. To let the reader understand better what's happening we will perform just the first row swap (between rows 1 and 3), but the process for the second swap (between rows 2 and 6) is similar.

Listing 4.3: Building the entangled database in Q#

```

operation PhoneBookSearch (q0 : Result , q1 : Result , q2 : Result) : (
    Result , Result , Result) {

    let n1 = 1;    // number of qubits for the name
    let n2 = 2;    // number of qubits for the telephone number

    using (q = Qubit[n1+n2]) {

        Set(q0, q[0]);
        Set(q1, q[1]);
        Set(q2, q[2]);

        // building the primed matrix

        IC_1q([q[0],q[1]], H, q[2]);

        // exchange rows 1 and 3

        SHIFT([q[0], q[1]]); // SHIFT3
        (Controlled SWAP)([q[0]], (q[1], q[2]));
        (Controlled ICNOT)([q[0]], [q[1], q[2]]);
        (Controlled SWAP)([q[0]], (q[1], q[2]));
        SHIFT([q[0], q[1]]); // SHIFT3

        // returning measurements

        let res0 = M(q[0]);
        let res1 = M(q[1]);
        let res2 = M(q[2]);
        ResetAll(q);
        return (res0 , res1 , res2);
    }
}

```

In Figure 4.1a we see the row swap between 1 and 3 in action, in particular the outputs when entering  $(0, 0, 0)$  and  $(0, 1, 0)$  are swapped, as it should be.

In Figure 4.1b we see the primed matrix  $A'$  (without row swaps). Notice that there is an equal probability of getting 0 or 1 on the least significative qubit when the first 2 are both zero (i.e.  $H$   $2 \times 2$  gate has been applied in

direct sum with an identity matrix), as it should be.

Finally in Figure 4.1c we see both operations performed on the same circuit, therefore producing an indetermination on the second least significative qubit as well, which is the changing one between row 1 (000) and row 3 (010).

Input: (0, 0, 0)  
Output: (0, 1, 0)

Input: (0, 0, 1)  
Output: (0, 0, 1)

Input: (0, 1, 0)  
Output: (0, 0, 0)

Input: (0, 1, 1)  
Output: (0, 1, 1)

Input: (1, 0, 0)  
Output: (1, 0, 0)

Input: (1, 0, 1)  
Output: (1, 0, 1)

Input: (1, 1, 0)  
Output: (1, 1, 0)

Input: (1, 1, 1)  
Output: (1, 1, 1)

Input: (0, 0, 0)  
Output: (0, 0, 0, 58)

Input: (0, 0, 1)  
Output: (0, 0, 0, 47)

Input: (0, 1, 0)  
Output: (0, 1, 0)

Input: (0, 1, 1)  
Output: (0, 1, 1)

Input: (1, 0, 0)  
Output: (1, 0, 0)

Input: (1, 0, 1)  
Output: (1, 0, 1)

Input: (1, 1, 0)  
Output: (1, 1, 0)

Input: (1, 1, 1)  
Output: (1, 1, 1)

(a) Swap op. between rows 1 and 3

(b) Primed matrix  $A'$

Input: (0, 0, 0)  
Output: (0, 0, 45, 0, 55)

Input: (0, 0, 1)  
Output: (0, 0, 48, 0, 52)

Input: (0, 1, 0)  
Output: (0, 0, 0)

Input: (0, 1, 1)  
Output: (0, 1, 1)

Input: (1, 0, 0)  
Output: (1, 0, 0)

Input: (1, 0, 1)  
Output: (1, 0, 1)

Input: (1, 1, 0)  
Output: (1, 1, 0)

Input: (1, 1, 1)  
Output: (1, 1, 1)

(c) Swapped matrix  $A$

Figure 4.1: Partial outputs of the code in Listing 4.3.

## Chapter 5

# Other minor aspects explored during the research

Grover’s Algorithm was originally devised to work with functions that are satisfied by a single input. Actually it has also been “generalized to search in the presence of multiple *winners*”. [6]

It turns out that Grover’s Algorithm can be more useful in “speeding up the solution to NP-complete problems such as 3-SAT” than actual search. [13]

We also considered spatial database search as a possible way of exploiting Grover’s Algorithm, in the specific case of graphs with costs on arcs [4, 8].

Although there are some points of contact with Grover, none of them seemed to me of any use for our Max Flow Analysis problem.

# Chapter 6

## Conclusions

In this work we explored several introductory aspects of quantum computing.

First of all Microsoft QDK, a quantum development environment that let us write programs for future quantum computers, and its language Q#.

Then we got deep in the implementation of the entangled database for Grover Search, a specific part of the whole bigger algorithm that is often ignored and given for granted. In this context we provided not only a theory for performing passage needed to such entanglement, but also a working example of how this algorithm could be implemented with real gates.

Although the original purpose of this research project was to explore a new and unknown topic for the author, without any expectations, it turned out to be a little more. It was the opportunity to look with a critic eye to the state of the art and to propose approaches not yet explored, besides being of course a valuable experience.

# Appendices

# Appendix A

## Quantum gates in Octave

Octave is a free software and a scientific programming language whose syntax is largely compatible with Matlab.

To fill the gap between some theoretical papers (which perform calculations on matrices) and quantum gates (that are eventually how those matrices are implemented) we modeled some quantum matrices as combination of known gates. In this way it was possible to investigate on how such matrices could be really implemented.

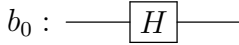
Therefore here we show the implementation of some gates used in other chapters.

### A.1 Elementary (existing) gates

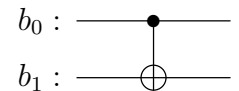
#### A.1.1 Hadamard and X, Y, Z

We will show only  $H$  as an example, but the same applies for  $X$ ,  $Y$  and  $Z$  and in general for  $2 \times 2$  gates.

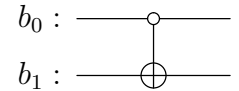


Circuit	Octave code
$b_0 :$ 	<pre>H = [ 1, 1; 1, -1 ] ./ sqrt(2);</pre>

### A.1.2 CNOT

Circuit	Octave code
$b_0 :$ 	<pre>CNOT = C(X);</pre>

### A.1.3 ICNOT: inverted CNOT

Circuit	Octave code
$b_0 :$ 	<pre>ICNOT = IC(X);</pre>

Note that **it is not equivalent** to this circuit:

$b_0 :$  

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

### A.1.4 SWAP

Circuit	Octave code
$  \begin{array}{c}  b_0 : \text{---} \times \text{---} \\  b_1 : \text{---} \times \text{---}  \end{array}  $	<pre> SWAP = [ 1, 0, 0, 0; 0, 0, 1, 0; 0, 1, 0, 0; 0, 0, 0, 1; ]; </pre>

### A.1.5 CCNOT

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

Circuit	Octave code
$  \begin{array}{c}  b_0 : \text{---} \bullet \text{---} \\  b_1 : \text{---} \bullet \text{---} \\  b_2 : \text{---} \oplus \text{---}  \end{array}  $	<pre> CCNOT = C(CNOT); </pre>

### A.1.6 CSWAP

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

Circuit

Octave code



## A.2 Operations between gates

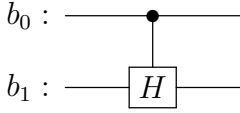
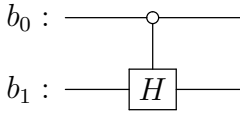
### A.2.1 Kronecker product (or direct product)

Circuit

Octave code



## A.2.2 Gate control (direct sum)

Circuit	Octave code
 <p> <math>b_0</math>: ———●—————  <math>b_1</math>: ———<math>\boxed{H}</math>—————         </p>	<pre>function out = C(gate) size = rows(gate); out = blkdiag(eye(size), gate ); endfunction;</pre>
 <p> <math>b_0</math>: ———○—————  <math>b_1</math>: ———<math>\boxed{H}</math>—————         </p>	<pre>function out = IC(gate) size = rows(gate); out = blkdiag(gate, eye(size) ); endfunction;</pre>

Note that CNOT is a “controlled  $X$ ”.

## A.3 New (derivated) gates

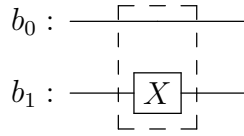
### A.3.1 DSWAP

Performs a swap of the first 2 and the last 2 rows of the matrix, i.e. flips the least significative qubit.

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

Circuit

Octave code



```
| DSWAP = kron(eye(2), X);
```

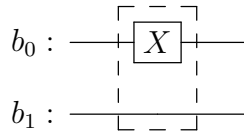
### A.3.2 SHIFT

This gate shifts rows of half the size of the matrix, i.e. flips the most significant qubit.

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

Circuit

Octave code

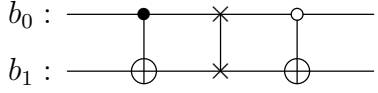


```
| DSWAP = kron(X, eye(2));
```

### A.3.3 QSD: Quarter Shift Down

This gate shifts rows down of a quarter the matrix.

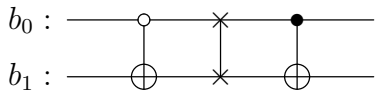
$$QSD = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Circuit	Octave code
	<pre>  QSD = ICNOT*SWAP*CNOT;</pre>

### A.3.4 QSU: Quarter Shift Up

This gate shifts rows down of a quarter the matrix.

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

Circuit	Octave code
	<pre>  QSU = CNOT*SWAP*ICNOT;</pre>

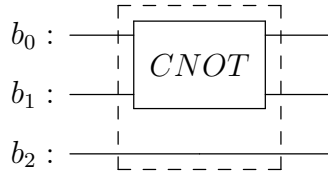
### A.3.5 CNOT3

CNOT of grade 1. Please note that it is different from CCNOT.

$$CNOT3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Circuit

Octave code



```
CNOT3 = kron(CNOT, eye
(2));
```

### A.3.6 SWAP3

$$SWAP3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Circuit	Octave code
$ \begin{array}{l} b_0 : \text{---} \boxed{\times} \text{---} \\ b_1 : \text{---} \boxed{\times} \text{---} \\ b_2 : \text{---} \boxed{\phantom{\times}} \text{---} \end{array} $	<pre>SWAP3 = kron(SWAP, eye (2));</pre>

### A.3.7 SHIFT3

$$SHIFT3 = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Octave code

```
|SHIFT3 = kron(SHIFT, eye(2));
```

### A.3.8 CNOT4

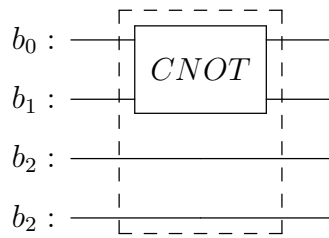
CNOT of grade 2. Please note that it is different from CCCNOT.



$$CNOT4 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Circuit

Octave code



```
CNOT4 = kron(CNOT, eye
(4));
```

# Bibliography

- [1] Grover's algorithm: what to input to oracle? <https://quantumcomputing.stackexchange.com/questions/2149/grovers-algorithm-what-to-input-to-oracle>, 2018.
- [2] Grover's algorithm: where is the list? <https://quantumcomputing.stackexchange.com/questions/2110/grovers-algorithm-where-is-the-list>, 2018.
- [3] How is the oracle in grover's search algorithm implemented? <https://quantumcomputing.stackexchange.com/questions/175/how-is-the-oracle-in-grovers-search-algorithm-implemented>, 2018.
- [4] D. Aharonov, A. Ambainis, J. Kempe, and U. Vazirani. Quantum walks on graphs. In *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing*, STOC '01, pages 50–59, New York, NY, USA, 2001. ACM.
- [5] P. Alsing and N. McDonald. Grover's search algorithm with an entangled database state. *Proceedings of SPIE - The International Society for Optical Engineering*, 05 2011.

- [6] M. Boyer, G. Brassard, P. Hyer, and A. Tapp. Tight bounds on quantum searching. *Fortschritte der Physik*, 46(45):493–505, 1998.
- [7] B. Broda. Quantum search of a real unstructured database. *The European Physical Journal Plus*, 131(2):38, Feb 2016.
- [8] A. M. Childs and J. Goldstone. Spatial search by quantum walk. *Physical Review A*, 70(2):022314, 2004.
- [9] L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 212–219, New York, NY, USA, 1996. ACM.
- [10] A. Helwer. Quantum computing for computer scientists. Available at <https://www.microsoft.com/en-us/research/video/quantum-computing-computer-scientists/>, 2018.
- [11] R. LaRose. Overview and comparison of gate level quantum software platforms. *Quantum*, 3:130, 2019.
- [12] C. Lavor, L. Manssur, and R. Portugal. Grover’s algorithm: quantum database search. *arXiv preprint quant-ph/0301079*, 2003.
- [13] A. Montanaro. Quantum algorithms: an overview. *npj Quantum Information*, 2:15023, 2016.
- [14] G. F. Viamontes, I. L. Markov, and J. P. Hayes. Is quantum search practical? *Computing in Science Engineering*, 7(3):62–70, May 2005.
- [15] C. P. Williams. *Explorations in Quantum Computing*. Texts in Computer Science. Springer, 2nd edition, 2011.

- [16] C. Zalka. Using grover's quantum algorithm for searching actual databases. *Physical Review A*, 62:52305, 10 2000.