Effect of Pure Dephasing Quantum Noise in the Quantum Search Algorithm Using Atos Quantum Assembly

Maria Heloísa Fraga da Silva *, Gleydson Fernandes de Jesus *, Clebson Cruz *

Posted Date: 14 May 2024

doi: 10.20944/preprints202405.0930.v1

Keywords: Quantum Computing; Grover's algorithm; Software Development; Quantum Noise

 doi:10.20944/preprints202405.0930.v1

Article

Effect of Pure Dephasing Quantum Noise in the Quantum Search Algorithm Using Atos Quantum Assembly

Maria Heloísa Fraga da Silva [1,2,*], Gleydson Fernandes de Jesus [2,*] and Clebson dos Santos Cruz [1,*]

[1] Grupo de Informação Quântica e Física Estatística, Centro de Ciências Exatas e das Tecnologias, Universidade Federal do Oeste da Bahia—Campus Reitor Edgard Santos. Rua Bertioga, 892, Morada Nobre I, 47810-059 Barreiras, Bahia, Brazil

[2] Latin American Quantum Computing Center, High Performance Computing Center, SENAI CIMATEC. Av. Orlando Gomes, 1845, Piatã, 41650-010 Salvador, Bahia, Brazil.

* Correspondence: maria.fraga@fbter.org.br (M.H.F.d.S.); gleydson.jesus@fieb.org.br (G.F.d.J.); clebson.cruz@ufob.edu.br (C.d.S.C.)

Abstract: Quantum computing is tipped to lead the future of global technological progress. However, the obstacles related to quantum software development are an actual challenge to overcome. In this scenario, this work presents a implementation of the quantum search algorithm in Atos Quantum Assembly Language (AQASM) using the quantum software stack my Quantum Learning Machine (myQLM). We present the creation of a virtual quantum processor whose configurable architecture allows the analysis of induced quantum noise effects on the quantum algorithms. The codes are available throughout the manuscript so that readers can replicate them and apply the methods discussed in this article to solve their own quantum computing projects. The presented results are consistent with theoretical predictions and demonstrate that AQASM and myQLM are powerful tools for building, implementing, and simulating quantum hardwares.

Keywords: quantum computing; Grover's algorithm; software development; AQASM; quantum noise

1. Introduction

The advent of quantum computing is one of the bases of the so-called second quantum revolution [1–4]. The scientific community ceased to be mere observers of the laws of quantum mechanics and is now actively acting to use them in the development of practical quantum devices [1,2]. In this scenario, developing devices that work according to the laws of quantum mechanics is one of the most ambitious goals of the current century. This revolution is expected to be responsible for the major technological breakthroughs of the 21st century, and millions of dollars are already being invested in research into the development of quantum hardware by companies and universities around the world [5,6]. The advent of this technological revolution also marks the onset of the Noisy Intermediate-Scale Quantum (NISQ) era [7–9], characterized by the development of quantum devices with a limited number of qubits and high susceptibility to noise [8]. In this scenario, the NISQ era provides an enticing insight into the capacity of quantum computing to revolutionize fields ranging from cryptography and optimization to drug discovery and materials science [10].

Nevertheless, the realization of this potential depends on successfully overcoming challenging technological obstacles [8]. NISQ devices are inherently prone to errors arising from decoherence, gate imperfections, and environmental noise [11,12]. Moreover, the developing fault-tolerant quantum error correction codes is challenging and requires new methods to reduce the effects of noise on quantum computing [9].

In this context, our work delves into the intricacies of quantum noise and its impact on quantum algorithms. We present a study on the Effect of pure dephasing quantum noise in the quantum search algorithm, employing an implementation of the 4-qubit quantum search algorithm on the Atos Quantum Assembly Language (AQASM) [13,14]. Leveraging the my Quantum Learning Machine (myQLM) quantum software stack [15,16], we provide a practical framework for writing and executing quantum algorithms through a Python interface. Our approach enables researchers to conduct noise-free simulations using the PyLinalg linear algebra simulator via myQLM, all from the convenience of a

 doi:10.20944/preprints202405.0930.v1

standard home computer. Importantly, we make our programming code readily available throughout the text, facilitating reproducibility and enabling others to build upon our work.

However, the implementation of Grover's algorithm, a cornerstone of quantum search, presents challenges due to the necessity for multi-controlled quantum gates. To address this obstacle, we utilize the NoisyQProc simulator, an invaluable resource within the Quantum Learning Machine (QLM) programming development platform [17]. While this simulator is a private version of myQLM and not publicly accessible, it allows for the emulation of real quantum processors with adjustable architectures, enabling comprehensive analysis of quantum noise effects in the quantum search problem.

Our findings underscore the efficacy of AQASM as a versatile tool for building, implementing, and simulating quantum hardware, thus bridging the gap between theory and practice in the NISQ era. By elucidating the challenges and opportunities inherent in quantum computing, we contribute to the ongoing dialogue surrounding the development of practical quantum technologies and pave the way for future advancements in the field.

2. Quantum Search Algorithm

First, let us start with an overview of the four-qubit quantum search algorithm. The search problem is a very common topic in classical computing. Considering an unstructured database with N entries, the problem consists in determining the index of the database entry (x) that satisfies some predefined search criteria $x = y$, where y is the searched element - a brief explanation of the algorithm can be found on supplementary material[18].

Assuming Grover's algorithm for $n = 4$ qubits, one can create a list of $N = 16$ items, represented by each state of the computational basis of a 4 qubits system. Table 1 shows the list of 16 items for which we chose color names randomly arranged. Each item is associated with a number from 0 to 15, as listed in the second column. In the third column, we represent each decimal as its equivalent in binary code. Finally, as Grover's algorithm performs the search on the state vectors, each binary entry is associated with a state vector in the computational basis for 4 qubits, as per the last column.

Table 1. Conversion of items into vectors.

ITEM	DECIMAL	BINARY	VECTOR
Orange	0	0000	$\|0000\rangle$
Magenta	1	0001	$\|0001\rangle$
Yellow	2	0010	$\|0010\rangle$
Violet	3	0011	$\|0011\rangle$
Beige	4	0100	$\|0100\rangle$
Purple	5	0101	$\|0101\rangle$
White	6	0110	$\|0110\rangle$
Pink	7	0111	$\|0111\rangle$
Brown	8	1000	$\|1000\rangle$
Green	9	1001	$\|1001\rangle$
Black	10	1010	$\|1010\rangle$
Cyan	11	1011	$\|1011\rangle$
Salmon	12	1100	$\|1100\rangle$
Gray	13	1101	$\|1101\rangle$
Red	14	1110	$\|1110\rangle$
Blue	15	1111	$\|1111\rangle$

In the following, we present the four steps that characterize Grover's algorithm: Initialization, Oracle, Amplitude Amplification, and Measurement [19], along with the integral code written using AQASM language and myQLM quantum software stack.

First of all, we recommend installing the Jupyter Notebook environment, which is available for free. In order to do this, we recommend installing the free Anaconda platform, which can be downloaded for Windows, macOS and Linux operating systems from the official website [20]. Once installed, the platform's interface on the home page displays applications including Jupyter Notebook.

 doi:10.20944/preprints202405.0930.v1

By clicking on "Launch", the user is automatically directed to the Jupyter environment, where it is possible to create a Python-compatible ipynb file and perform actions explained in the Jupyter documentation [21]. Secondly, the myQLM tool must be installed and it is also available for free using the official guide provided [22]. Once the above installations have been made, one can proceed to write and run the code shown in the boxes below using AQASM, which documentation is available on the official GitHub [23].

Finally, both the Jupyter Notebook file in myQLM regarding the Boxes in Section II for the noise-free simulation and the Jupyter Notebook file in QLM regarding the Boxes in Section III for the noisy simulation are available in a GitHub repository created by the authors for easy access by readers [18].

Once the integrated development environment has been established, the first step is to import the computational tools needed to implement the quantum algorithm in AQASM. Box 1 shows the command cell that imports the computational tools required to implement Grover's algorithm in AQASM.

Box 1: Importing the computational tools

```
from qat.lang.AQASM import Program, CNOT, H,␣
↪Z, X, QRoutine
```

Subsequently, we allocate the amount of quantum and classical bits that will compose our quantum circuit, as shown in Box 2. The same quantity is defined for both since the bits will store the results of the measurements performed on the qubits to identify their final states [19].

Box 2: Allocating qubits and classical bits registers

```
# Creating Grover's program
grover = Program()

# Allocating qubits and classical bits␣
↪registers
Qr = grover.qalloc(4)
Cr = grover.calloc(4)
```

2.1. Initialization

In order to initialize the qubits in a balanced superposition represented by the state $|S\rangle$, one can apply the `Hadamard` gate on all the qubits

$$|S\rangle = H^{\otimes 4}|0000\rangle \, . \tag{1}$$

In terms of coding, Box 3 shows the initialization process of Grover's algorithm.

Box 3: Initializing the qubits in a balanced superposition

```
for i in range(0,4):
    grover.apply(H, Qr[i])
```

2.2. Oracle

In the following, we must build the Oracle that searches for the desired item. There are two main methods used in literature to build the Oracle subroutine: the Boolean and phase inversion methods [19,24,25]. In the boolean technique, the presence of an auxiliary qubit, also known as ancilla, initialized in the $|1\rangle$ state, is necessary. In this scenario, the ancilla is changed only if the input to the

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 May 2024 doi:10.20944/preprints202405.0930.v1

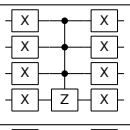
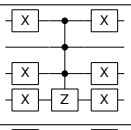
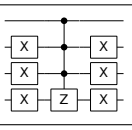
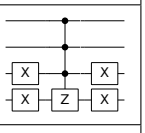
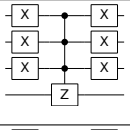
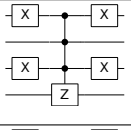
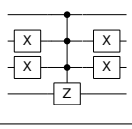
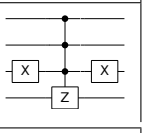
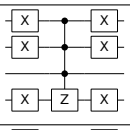
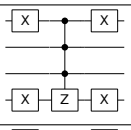
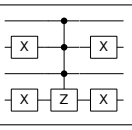
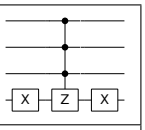
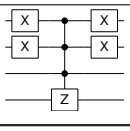
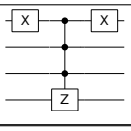
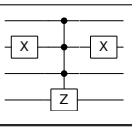
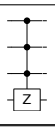
circuit is the sought state. However, this method is analogous to the classical search problem [24,25] and is generally applied to compare the computing power of the quantum superposition principle for quantum computing [25].

Thus, for the purposes of simplicity, we opted for the most straightforward method, the phase inversion method [24,25]. This method excludes the need for an ancilla, and the Oracle's function in this process becomes to identify the sought element in the balanced superposition of the states of the computational base described above and to add to it a negative phase. In this context, the Oracle function can be represented by the unitary operator:

$$O|x\rangle = \begin{cases} -|x\rangle & \text{se } x = y, \\ |x\rangle & \text{se } x \neq y, \end{cases} \tag{2}$$

where $|x\rangle$ is a state of the computational basis. Therefore, O can be defined as a diagonal matrix, which adds a negative phase to the state corresponding to the searched item $|y\rangle$. It is important to highlight that there is an oracle to search for each state vector on the 4-qubit computational basis. Table 2 shows the quantum circuits for all oracles in the 4-qubit Grover's algorithm.

Table 2. Quantum circuit for the Oracle subroutines for each state vector in the 4-qubit computational basis.

STATE	ORACLE	STATE	ORACLE	STATE	ORACLE	STATE	ORACLE
$\|Orange\rangle$		$\|Beige\rangle$		$\|Brown\rangle$		$\|Salmon\rangle$	
$\|Magenta\rangle$		$\|Purple\rangle$		$\|Green\rangle$		$\|Grey\rangle$	
$\|Yellow\rangle$		$\|White\rangle$		$\|Black\rangle$		$\|Red\rangle$	
$\|Violet\rangle$		$\|Pink\rangle$		$\|Cyan\rangle$		$\|Blue\rangle$	

Here we arbitrarily choose to find the item "Blue", associated with decimal 15, which correspondent binary is 1111 and state vector $|1111\rangle$ according to Table 1. The oracle responsible for marking this item is composed solely of the multi-controlled gate Z (MCZ), having qubits 0 to 3 as the controls and qubit 4 as the target, as shown in Box 4.

 doi:10.20944/preprints202405.0930.v1

Box 4: Creating the Oracle subroutine that marks state $|1111\rangle$

```
oracle_1111 = QRoutine()

# Building the multi-qubit controlled Z gate
def MCZ(nb_control):
    return Z.ctrl(nb_control-1)

# Applying the multi-qubit controlled Z gate
oracle_1111.apply(MCZ(4), Qr[0], Qr[1],␣
 ↪Qr[2], Qr[3])
```

However, even having indicated the sought element with a negative phase, the Oracle routine is still insufficient to guarantee that the searched element in the list will be found if we measure our balanced superposition. This is due to the fact that adding the phase in the sought state does not affect the probability distribution of the global state. In this sense, the probability of finding the searched item is $1/N$ (6.25%), which is equivalent to the classical Oracle in a single query in the list. Therefore, it is necessary to amplify the probability of the sought element, increasing the chance of finding it and reducing the probabilities of the other states in the process. This step is performed by the Amplitude Amplification method [24–26].

2.3. Amplitude Amplification

In contrast to the Oracle, the Amplitude Amplification subroutine is the same, regardless of the state representing the item sought. The amplification is done by performing a reflection represented by the unitary operation

$$A = 2|S\rangle\langle S| - \mathbb{I}, \tag{3}$$

increasing the amplitude of the searched item $|y\rangle$ [25]. Box 4 shows the cell that creates the Amplitude Amplification subroutine. First, we apply the `Hadamard` gate to all qubits in the Oracle-modified state. Then gate `X` is used to flip all qubits and we apply the `MCZ` gate. Finally, the process is finished by flipping all qubits again and applying the `Hadamard` gate, as shown in Box 5, obtaining the final state and amplifying the probability of the sought item being found.

Box 5: Creating the Amplitude Amplification subroutine

```
ampl = QRoutine()

for i in range(0,4):
    ampl.apply(H, Qr[i])

for i in range(0,4):
    ampl.apply(X, Qr[i])

# Applying the multi-qubit controlled Z gate
ampl.apply(MCZ(4), Qr[0], Qr[1], Qr[2],␣
 ↪Qr[3])

for i in range(0,4):
    ampl.apply(X, Qr[i])

for i in range(0,4):
    ampl.apply(H, Qr[i])
```

When we reach the final state, Grover's algorithm is completed by measuring the amplified state. The probability of finding the searched item in a single measurement after the Amplitude Amplification

 doi:10.20944/preprints202405.0930.v1

process in the 4-qubit algorithmic probability is 39.0625%, which is an improvement compared to the 6.25%, achieved in the classical algorithm in a single query of the Oracle, but still insufficient to assure that the search will be successful. Thus, in order to achieve the full potential of Grover's algorithm, subroutines Oracle and Amplitude Amplification need to be repeated to maximize the probability of finding the sought state in the measurement of the quantum state [24]. The algorithmic probability of finding the searched item after r repetitions is given by

$$P = \left[\frac{2}{\sqrt{N}}\left(\frac{N-2r}{2N} + \frac{N-r}{N}\right)\right]^2. \tag{4}$$

From Eq. (4), it is possible to verify that $r = \sqrt{N}$ is enough to assure that the user obtains the sought state after the measurement. Figure 1 shows a sketch of the process, with a pictorial representation of the probability amplitudes in the 4-qubits scenario.

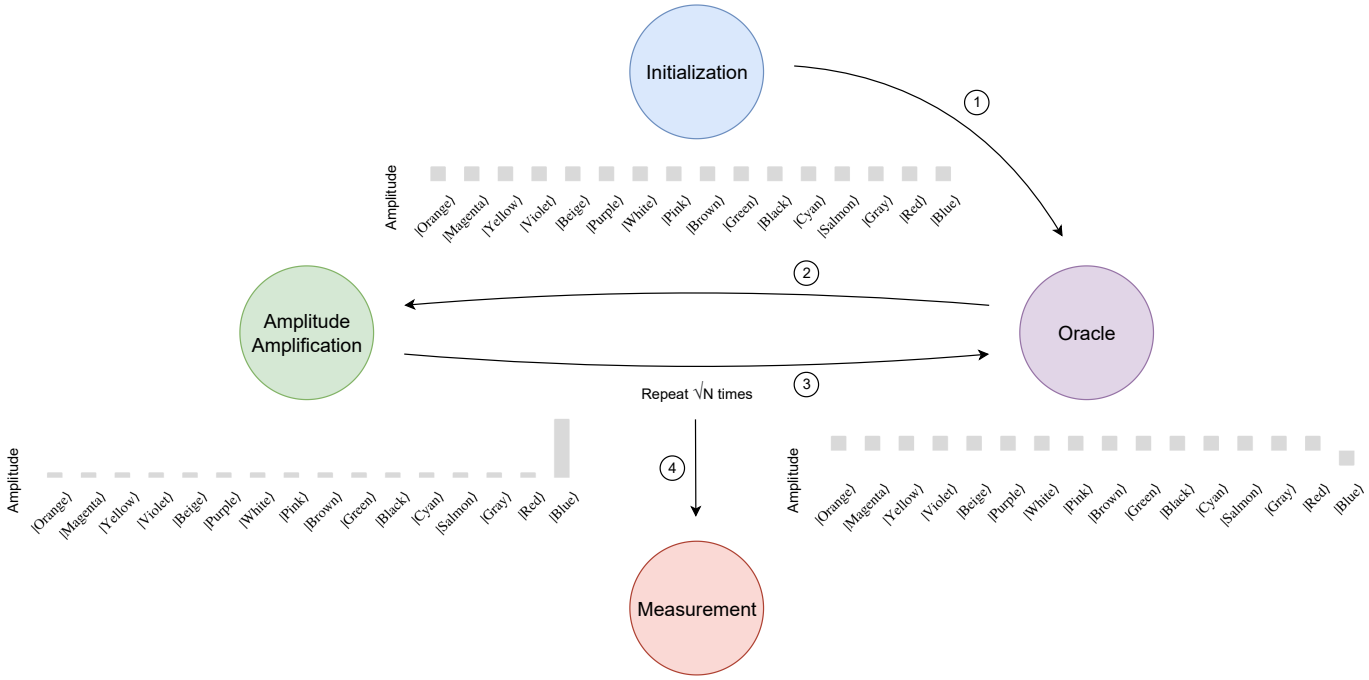


Figure 1. Sketch of the four steps of Grover's algorithm along with the evolution of the probability amplitudes of each element of the 4-qubits computational basis.

Since it is up to the user to inform which item he or she is looking for, it is possible to make available, at the beginning of the code, an interaction instructing him to type the name of the item so that our software applies the corresponding oracle (Box 6). In this context, the conditional statement `if` was used for item 0, `elif` for items 1 to 15, and `else` if the user enters an item that doesn't belong to the list.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 May 2024 doi:10.20944/preprints202405.0930.v1

Box 6: Asking for user input

```
# Dictionary to map items to their oracles
oracle_mapping = {
    "Orange": oracle_0000,
    "Magenta": oracle_0001,
    "Yellow": oracle_0010,
    "Violet": oracle_0011,
    "Beige": oracle_0100,
    "Purple": oracle_0101,
    "White": oracle_0110,
    "Pink": oracle_0111,
    "Brown": oracle_1000,
    "Green": oracle_1001,
    "Black": oracle_1010,
    "Cyan": oracle_1011,
    "Salmon": oracle_1100,
    "Gray": oracle_1101,
    "Red": oracle_1110,
    "Blue": oracle_1111,
}

item = input("Which item do you want to find?
 ↪ Use initial capital letters. ")

# Checks if the item is in the dictionary
if item in oracle_mapping:
    # Obtains the oracle corresponding to␣
 ↪the item
    oracle = oracle_mapping[item]

    # Repeats the subroutines
    for x in range(3):
        grover.apply(oracle, Qr)
        grover.apply(ampl, Qr)
else:
    print("This item is not in the list,␣
 ↪please check the spelling or enter another␣
 ↪option.")
```

At this point, the user would type the input “Blue" implying the application of the subroutines Oracle and Amplitude Amplification. The following output is displayed:

```
Which item do you want to find?  Use initial capital letters.  Blue
```

2.4. Measurement

The fourth and final step of the Algorithm is to perform measurements. Here we will store the final result of each qubit in the corresponding classical bit via the *measure* command, as illustrated in Box 7. Note that it is necessary to transform the program into a circuit before representing it graphically in a drawing, which is laid out in SVG format as in Figure 2.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 May 2024 doi:10.20944/preprints202405.0930.v1

Box 7: Measurements

```
# Measuring the qubits
grover.measure(Qr, Cr)

# Returning a circuit implementing Grover's␣
 ↪program
circuit = grover.to_circ()

# Drawing the circuit
%qatdisplay circuit  --svg
```

We can visualize below the complete circuit of Grover's algorithm for the Blue item search, assembled through the commands entered so far.

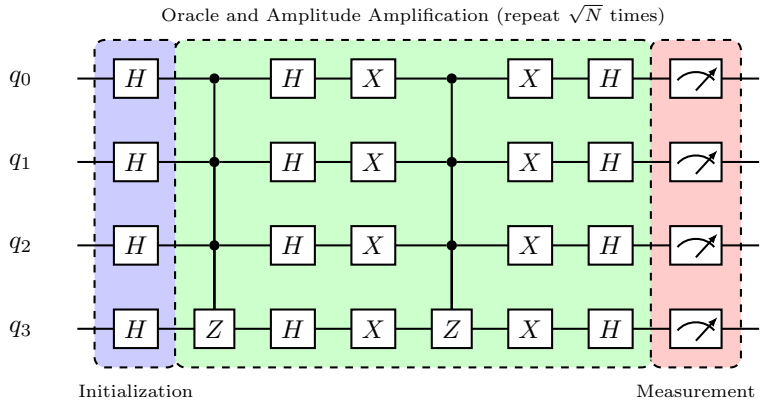


Figure 2. Grover's circuit using myQLM (abbreviated).

Box 8: Simulating and plotting the results

```
# Simulating the results
from qat.qpus import PyLinalg
qpu = PyLinalg()
job = circuit.to_job(nbshots=8192)
result = qpu.submit(job)

# Printing the results
for sample in result:
    print("State %s: probability %s +/- %s"␣
 ↪% (sample.state, sample.probability,␣
 ↪sample.err))
```

Therefore, using the quantum simulator `PyLinalg` made available by Atos and SENAI CIMATEC [27] for data processing, one can turn the search results into histogram format, as shown in Figure 3. Moreover, PyLinalg simulates the execution of quantum circuits on a local processor and returns the counts of each measurement in the final state for a given set of repetitions (or shots) of that circuit in quantity defined by the user. Increasing the number of shots optimizes the exploratory simulation process, bringing it closer to the theoretically predicted result [19]. As shown in Box 8, we performed 2^{13} shots.

 doi:10.20944/preprints202405.0930.v1

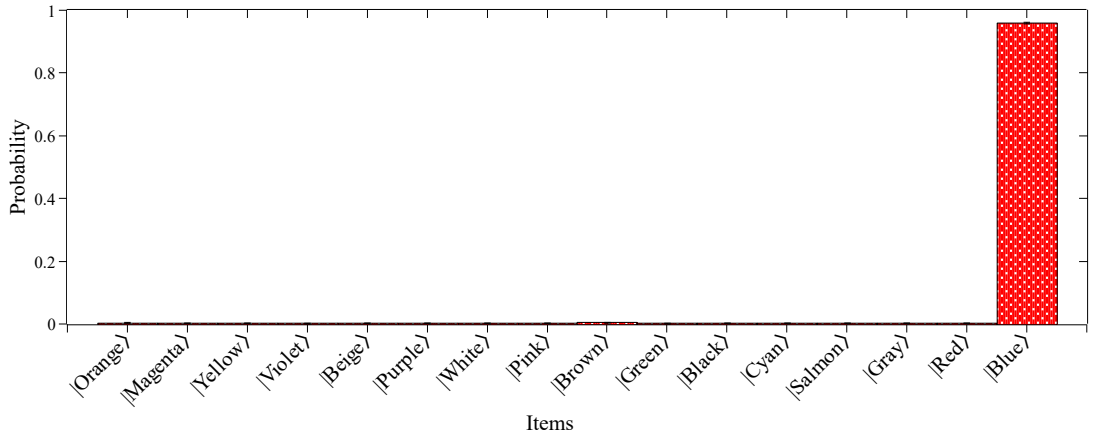


Figure 3. Probability distribution for the 16 items of the database. The searched item is found with 95.86% probability, while the other ones did not reach 0.5% probability.

As can be seen, the searched item was found with 95.86% probability, while the other items in the list shown in Table 1 did not reach 0.5% probability each. This significant increase in probability is related to the application of the repetitions of the oracle and amplitude amplification subroutines, as shown in Eq. (4). This is in contrast to the classical scenario since the probability of finding an item in an unstructured list with N=16 entries, running just one query to the list, is 6.25%. This result highlights the benefit of using quantum features such as superposition to process information. In addition, while classically, the Oracle needs to query the list $N/2$ on average, the quantum algorithm can find the marked item in $\sqrt{N}$ attempts using Grover's amplitude amplification method for solving the search problem. In conclusion, it is shown that the combination of Oracle and Amplification subroutines for developing Grover's algorithm results in a quadratic acceleration of the search problem, demonstrating that quantum computers have a major advantage over conventional computers.

3. Quantum Noise Interference on Grover's Algorithm

Despite its clear advantage, Grover's algorithm requires the application of multi-controlled quantum gates [19], which is an obstacle to implementing its 4-qubit and up versions in some quantum processors architectures presents in the popular IBM Quantum Experience [28] platform, for example. Consequently, there is a barrier to the scalability that would be required to make some algorithms useful and marketable on a large scale [8,29]. On the other hand, the number of qubits that are accessible in the system can be increased to substantially benefit the Grover algorithm. This would result in an exponential rise in the amount of storage space available in the database, which is the location where the searches are carried out. In this context, we carried out an emulation of a genuine quantum processor by making use of myQLM as a means of simulating the interconnectivity amongst the necessary qubits in an effort to overcome this challenge of compatibility.

In addition, the existence of noise that affects the qubits during the implementation of quantum operations or even during idle time is the primary obstacle that prevents present quantum computers from reaching their full potential. In this scenario, Grover's algorithm is one of the main hostages of noise since the quantum advantage of this algorithm can be better leveraged in quantum computers with a large number of qubits, which implies a large amount of noise and requires error correction protocols.

It is essential to note that all measurements conducted on a system are impacted by noise to varying degrees. This is anticipated by quantum physics, which postulates that quantum states collapse instantaneously at the point of measurement. This disturbance is inherent to the theory and cannot be prevented by any measuring technique [30]. Therefore, in order to simulate the quantum noise interference of the quantum processor on Grover's algorithm, we use the quantum computing simulator CIMATEC KUATOMU, an ATOS QLM simulator, located at SENAI CIMATEC's HPC center

 doi:10.20944/preprints202405.0930.v1

in Brazil, to build a simulated quantum hardware topology, since the AQASM noisy section is restricted to QLM users.

3.1. Quantum Hardware Model Simulation

To simulate our quantum hardware model, we need to adjust our code to meet the technical requirement that our hardware supports a maximum arity port of 3, which means that a quantum gate can be applied to a maximum of three qubits simultaneously. For this purpose, we decomposed the `CCCZ` gate into a set formed by `Hadamard` and `CCNOT` gates, as illustrated below. Consequently, it is necessary to introduce an auxiliary qubit for the circuit implementation to decompose this gate.

Therefore, to implement this new circuit, we created a specific 5-qubit topology in order to support the 4-qubit Grover algorithm and the auxiliary qubit of the decomposed `CCCZ` gate. All qubits are linked for the sake of simplicity; as a result, all of them can be concurrently controlled or targeted by controlled quantum operations. Figure 4 shows a sketch of the 5-qubit topology in which the 4-qubit Grover algorithm is implemented.

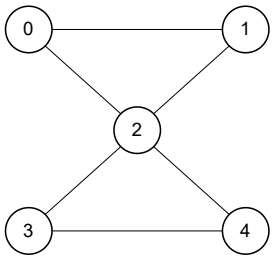


Figure 4. Sketch of the 5-qubit simulated topology. All qubits are coupled for simplicity since they can all be concurrently controlled or targeted by controlled quantum operations.

Similar to how we completed the noise-free simulation in Section II, we needed to import the logic gates we would use, as seen in Box 9.

Box 9: Importing the computational tools

```
from qat.lang.AQASM import Program, H, X,␣
↪CCNOT
```

In this scenario, the allocation of classical bits and quantum bits takes place in the exact same manner as it did in the previous simulation. The fact that we employ one of these qubits as an ancilla necessitates that we now assign 4 classical bits and 5 quantum bits in our system. (see Box 10).

Box 10: Allocating qubits and classical bits registers

```
# Creating Grover's program
grover = Program()

# Allocating qubits and classical bits␣
↪registers
Qr = grover.qalloc(5)
Cr = grover.calloc(4)
```

In order to build the topology connections for the quantum hardware (as shown in Figure 4), we must import the libraries `Topology` and `HardwareSpecs`. Box 11 contains the coding for the list items that represent the connections between the qubits in the idealized topology. The first member of each ordered pair is the control qubit, and the second element is the target one.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 May 2024 doi:10.20944/preprints202405.0930.v1

Box 11: Defining the topology of the simulated quantum hardware

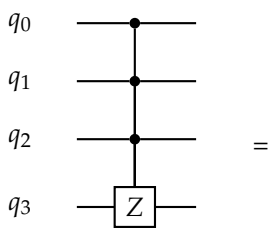
```
from qat.core.wrappers.hardware_specs import Topology, HardwareSpecs

topology = Topology()
for control, target in [(0, 1), (0, 2), (1, 0), (1, 2),
                        (2, 0), (2, 1), (2, 3), (2, 4),
                        (3, 2), (3, 4), (4, 2), (4, 3)
                       ]:
    topology.add_edge(control, target)

hw_specs = HardwareSpecs(nbqbits=n, topology=topology)
```

In Box 12, the initialization of the qubits occurs during the first step of Grover's algorithm by putting them in a balanced superposition. Since qubit 2 is a supplementary qubit in this instance, it is unnecessary to start it in a superposition state.

Box 12: Initializing the qubits in a balanced superposition

```
H(Qr[0])
H(Qr[1])
H(Qr[3])
H(Qr[4])
```

The second and third phases of the quantum search algorithm, implemented on the simulated quantum hardware are shown in Box 13. Initially, we apply the oracle that explicitly looks for the state $|1111\rangle$ by applying the `MCZ` gate decomposed as illustrated in Figure 5 (c). Next, we use amplitude amplification to enhance the probability that the marked object will be discovered while decreasing the probability that other items. These two procedures are repeated $\sqrt{N}$ times, in order to maximize the probability of finding the sought state, in the same way as performed in the noise-free simulation in section II.



(a)

 doi:10.20944/preprints202405.0930.v1

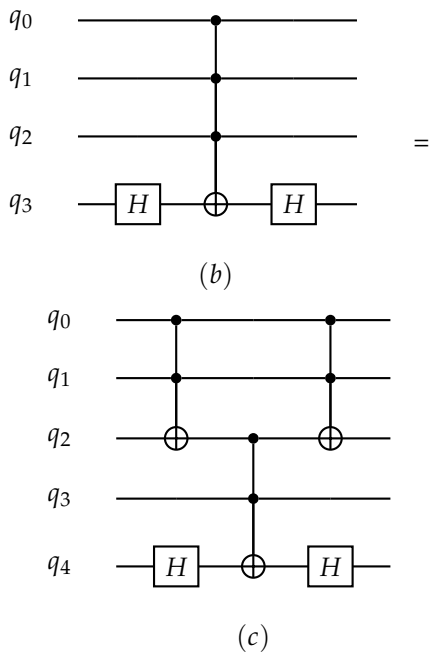


Figure 5. The decomposition of (a) multi-controlled `Z` gate (`CCCZ`) using (b) `Hadamard` and `CCCNOT` gates or (c) `Hadamard` and `CCNOT` gates.

 doi:10.20944/preprints202405.0930.v1

Box 13: Applying the Oracle and Amplitude Amplification subroutine that marks state $|1111\rangle$

```
for i in range(3):
    # Oracle
    H(Qr[4])
    CCNOT(Qr[0], Qr[1], Qr[2])
    CCNOT(Qr[2], Qr[3], Qr[4])
    CCNOT(Qr[0], Qr[1], Qr[2])
    H(Qr[4])

    # Amplitude Amplification
    H(Qr[0])
    H(Qr[1])
    H(Qr[3])
    H(Qr[4])
    X(Qr[0])
    X(Qr[1])
    X(Qr[3])
    X(Qr[4])
    H(Qr[4])
    CCNOT(Qr[0], Qr[1], Qr[2])
    CCNOT(Qr[2], Qr[3], Qr[4])
    CCNOT(Qr[0], Qr[1], Qr[2])
    H(Qr[4])
    X(Qr[0])
    X(Qr[1])
    X(Qr[3])
    X(Qr[4])
    H(Qr[0])
    H(Qr[1])
    H(Qr[3])
    H(Qr[4])
```

After this, we will finish the last step of Grover's algorithm, which consists of taking measurements in accordance with Box 14. It is worth noting that, the execution of the measurement on the auxiliary qubit is completely unnecessary.

Box 14: Measurements

```
grover.measure(Qr[0], Cr[0])
grover.measure(Qr[1], Cr[1])
grover.measure(Qr[3], Cr[2])
grover.measure(Qr[4], Cr[3])

# Returning a circuit implementing Grover's␣
 ↪program
circuit = grover.to_circ()

# Drawing the circuit
%qatdisplay circuit  --svg
```

 doi:10.20944/preprints202405.0930.v1

3.2. Quantum Noise

In the following, we introduce the quantum noise model in the simulated quantum hardware. Generally, to quantify quantum noise in quantum hardware setups, the measurement of two constants is performed: qubit relaxation time T_1 (i.e., *longitudinal relaxation* or *amplitude damping*) and qubit dephasing time T_2 (i.e., *transverse relaxation*) [31–33]. There is also a third parameter called Pure dephasing time which is often the dominant contribution to T_2 [34,35]. This parameter determines how long a system will be able to keep its coherence, whereas amplitude damping offers a model for the physical process of energy decay associated with the application of quantum noise [36] and it is represented by the following equation:

$$T_\phi = \frac{1}{\frac{1}{T_2} - \frac{1}{2T_1}} \tag{5}$$

In this regard, we introduce the Pure Dephasing channel in order to perform the noisy simulations on the quantum processor emulated with the topology described in Figure 4.

In order to perform the noisy simulation, there are some properties that must be specified: the running time of the quantum gates used to build the quantum hardware topology and the operating time for the quantum gates used in building quantum hardware topology. The gate application and relaxation times are properties of each quantum computer, which can be altered once the calibrations are performed. Therefore, we selected these settings not with the objective of recreating the results of a particular system but rather with the only purpose of demonstrating the impact of these noises on the outcomes of the simulation. Consequently, it will be possible to make a comparison between the results of the noisy simulation and the results of the ideal noise-free simulation performed in the previous section.

Box 15 imports the library `DefaultGatesSpecification` and specifies the application timings of the gates used in Grover's algorithm. Therefore, regarding the time needed to complete the procedures, we considered the following gate times in the code: `X` gate → 35.5 ns; `Hadamard` gate → 35.5 ns; `CCNOT` gate → 350 ns, Measurements → 35.5 ns.

Box 15: Defining gate times

```
gate_times = {"X":35.5, "H":35.5, "CCNOT":
 ↪350, "measure":35.5}

from qat.hardware import␣
 ↪DefaultGatesSpecification
gates_spec =␣
 ↪DefaultGatesSpecification(gate_times)
```

Furthermore, in order to apply the Pure Dephasing channel in Box 16, we need to import the library known as `ParametricPureDephasing`. In addition to the noise model, one can define the qubit relaxation times T_1 and T_2, respectively as shown in the second code line of Box 16. Finally, in the last code line we use the equation 5 to define T_ϕ.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 May 2024 doi:10.20944/preprints202405.0930.v1

Box 16: Defining qubit relaxation times

```
from qat.quops import ParametricPureDephasing

T1, T2 = 1000, 1000 # nanosecs

# Pure Dephasing: Lindblad approximation.
PD_Lindblad = ParametricPureDephasing(T_phi␣
 ↪= 1/(1/T2 - 1/(2*T1)))
```

Subsequently, after importing the `HardwareModel` and `NoisyQProc` libraries, we specify the model of the quantum processor that we will use, including the gate specifications and the noise model that will be applied (see Box 17). Note that these parameters were defined by us in Boxes 15 and 16.

Box 17: Importing quantum hardware

```
from qat.hardware import HardwareModel
from qat.qpus import NoisyQProc

hardware_lindblad =␣
 ↪HardwareModel(gates_spec,␣
 ↪idle_noise=[PD_Lindblad])
noisy_qpu =␣
 ↪NoisyQProc(hardware_model=hardware_lindblad)
```

The last step of our code is to simulate Grover's algorithm in the emulated quantum hardware under the Pure Dephasing noise model, as presented in Box 18. First, we need to create a `job` and link it to the quantum circuit created in Box 14 from Grover's program developed in the previous steps, as in the first code line of the following box. The parameter `nbshots` indicates the amount of repetitions used in the circuit, as will be explained later. Finally, these instructions were applied only to qubits 0, 1, 3 and 4 because, as explained before, qubit 2 is an auxiliary one.

Box 18: Simulation

```
job = circuit.to_job(nbshots=8192,␣
 ↪qubits=[0,1,3,4])
result = noisy_qpu.submit(job)
for sample in result:
    print("State %s, probability %s, err␣
 ↪%s"%(sample.state, sample.probability,␣
 ↪sample.err))
```

The simulations were conducted out using the stochastic technique, which requires that the density matrix be interpreted using a probability distribution based on pure states [24]. Using this method, the error scales with $\frac{1}{\sqrt{shots}}$. Thus, we perform the simulation with 2^{13} shots to get proper results.

Figure 6 shows the probability distributions of Grover's algorithm performed in quantum hardware simulated under the Pure Dephasing noise model. Based on the results obtained from this analysis, we can gradually observe the effects of noise in the application of the algorithm by decreasing the dephasing time (T_2). It is worth noting that, although the most probable state is still the searched item $|1111\rangle$, the other states arose with significant probabilities even after the $\sqrt{N}$ repetitions of the Oracle and Amplitude Amplification subroutines, different from the noise-free simulation presented in Figure 3. As expected, the situation becomes worse when we gradually decrease the dephasing

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 May 2024 doi:10.20944/preprints202405.0930.v1

time. The effect of the noise is to reduce the algorithm's accuracy, approximating the probability of the sought state to the other items of the unstructured list.

In order to obtain a landscape of the effect of the Pure Dephasing channel on the sought state, we plot the probability of the sought state as a function of the Pure Dephasing times (T_ϕ) in Figure 7. As can be seen, as T_ϕ increases, the probability of the sought state approximates to a probability limit 95.86%, obtained for the noise-free simulation presented in Figure 3, highlighted in the dashed blue line. On the other hand, the performance of the search algorithm is negatively impacted by decreasing the dephasing time, rapidly decreasing the probability sought state. Therefore, we are convinced that these findings demonstrate that the existence of quantum noise in quantum processors reduces the efficiency of the algorithm, which can lead to inaccurate outcomes when simulating quantum circuits.

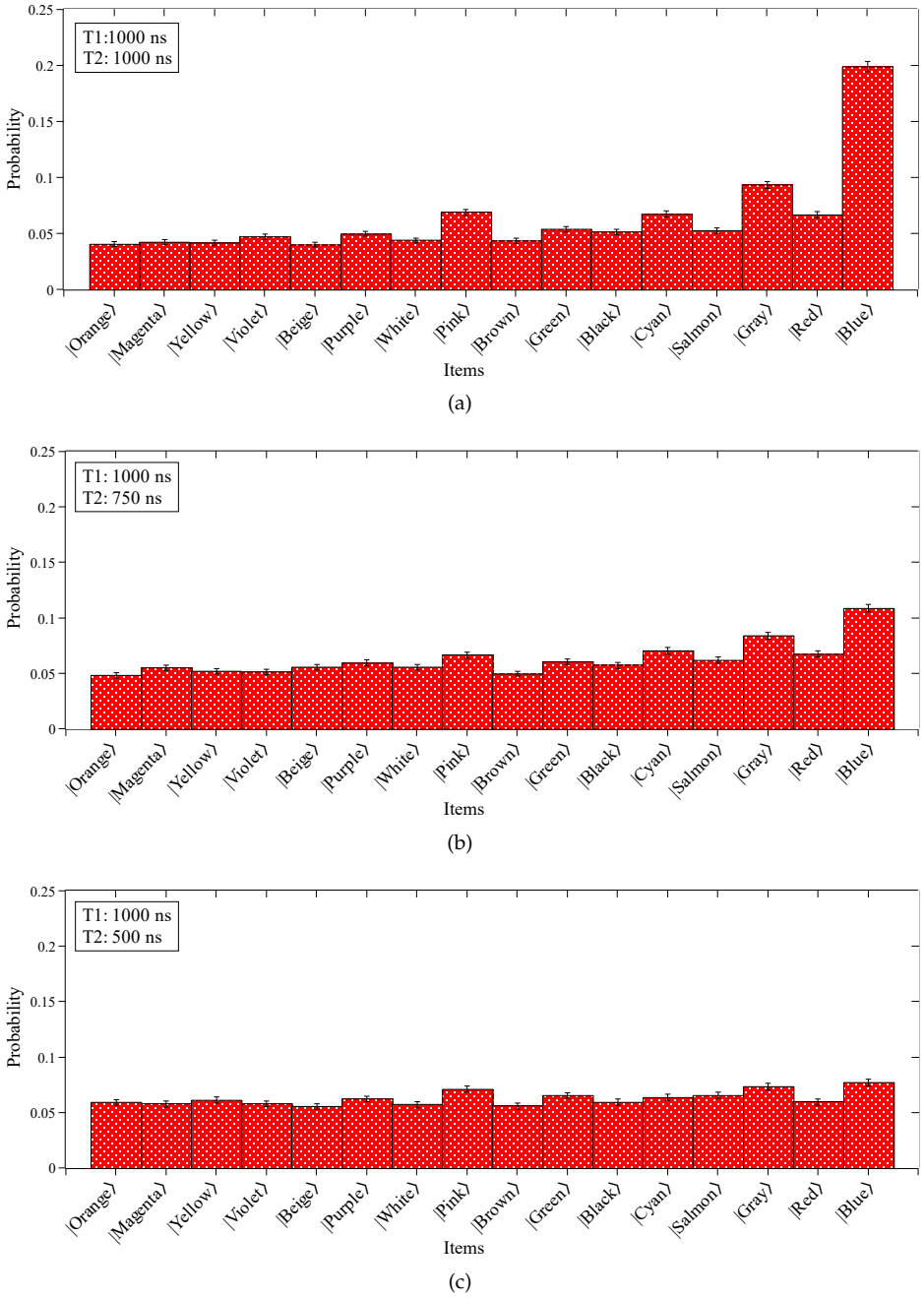


Figure 6. Simulation with noise using T_ϕ when T_1 is equal to 1000 ns and T_2 is equal to (a) 1000 ns, (b) 750 ns, and (c) 500 ns.

 doi:10.20944/preprints202405.0930.v1

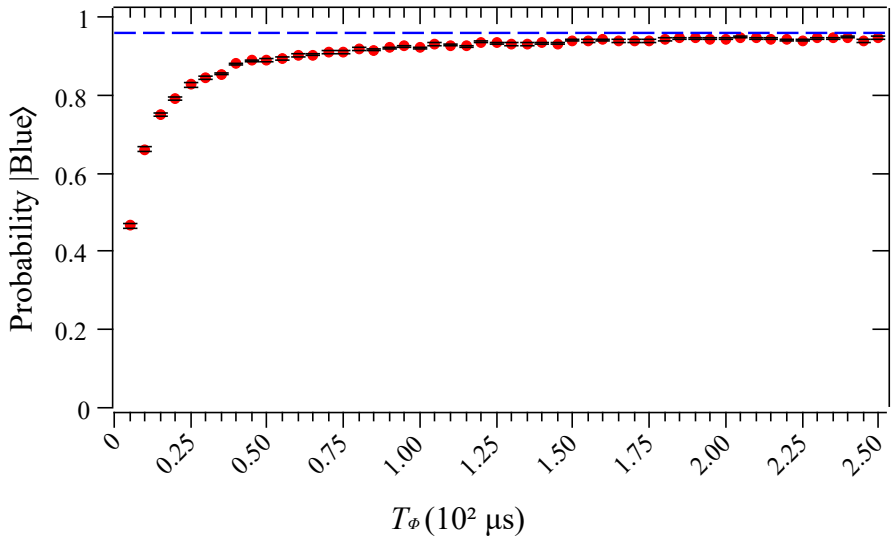


Figure 7. Probability distribution for item $|Blue\rangle$ as a function of Pure Dephasing times, T_ϕ. The dashed blue line highlights the probability of the sought state (95.86%) obtained in the noise-free simulation presented in Figure 3.

4. Conclusion

This paper presents the quantum search problem based on the famous Grover's algorithm associated with the binary encoding of words into quantum states of the 4-qubits computational basis. We highlight the main conditions for developing projects and their implementation in dedicated hardware processors. Our findings are consistent with the theoretical predictions found in the aforementioned literature for the examples that were covered, and they demonstrate that AQASM and myQLM would be practical tools for both the implementation and analysis of quantum algorithms. Moreover, we emulate a genuine quantum processor by configuring a dedicated quantum simulator to conform to the necessary architectural specifications for performing Grover's algorithm. Furthermore, we demonstrate that the existence of quantum noise leads to a progressive decline in the quality of the outcomes produced by a quantum noisy simulation when compared to the result performed in a noise-free setup. Finally, we propose as a potential direction for future study the scaling up of our program to a greater number of qubits, which would result in an exponential increase in our database capacity.

Data Availability Statement: The data that support the findings of this study are openly available in Supplementary Information Grover Paper, a GitHub repository created by the authors [18].

Acknowledgments: The authors thank SENAI CIMATEC for the access to the KUATOMU quantum simulator, which was used in obtaining the quantum noise simulation results. The authors thank the Bahia State Research Support Foundation (FAPESB) for financial support.

Conflicts of Interest: The authors have no conflicts to disclose.

References

1. Ivan H. Deutsch. Harnessing the power of the second quantum revolution. *PRX Quantum*, 1:020101, Nov 2020.
2. Matteo Atzori and Roberta Sessoli. The second quantum revolution: Role and challenges of molecular chemistry. *Journal of the American Chemical Society*, 141(29):11339–11352, 2019.
3. Barbara M Terhal. Quantum supremacy, here we come. *Nature Physics*, 14(6):530–531, 2018.
4. Aram W Harrow and Ashley Montanaro. Quantum computational supremacy. *Nature*, 549(7671):203–209, 2017.
5. Elizabeth Gibney. The quantum gold rush. *Nature*, 574(7776):22–24, 2019.

 doi:10.20944/preprints202405.0930.v1

6. Guido Peterssen. Quantum technology impact: The necessary workforce for developing quantum software. In *QANSWER*, pages 6–22, 2020.
7. Xinbiao Wang, Yuxuan Du, Yong Luo, and Dacheng Tao. Towards understanding the power of quantum kernels in the nisq era. *Quantum*, 5:531, 2021.
8. John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, 2018.
9. Yasunari Suzuki, Suguru Endo, Keisuke Fujii, and Yuuki Tokunaga. Quantum error mitigation as a universal error reduction technique: Applications from the nisq to the fault-tolerant quantum computing eras. *PRX Quantum*, 3:010345, Mar 2022.
10. M. AbuGhanem and Hichem Eleuch. Nisq computers: A path to quantum supremacy. *ArXiv*, abs/2310.01431, 2023.
11. J. A. Montanez-Barrera, M. V. von Spakovsky, Cesar E. Damian Ascencio, and S. Cano-Andrade. Decoherence predictions in a superconducting quantum processor using the steepest-entropy-ascent quantum thermodynamics framework. *Physical Review A*, 2022.
12. S. Dasgupta and T. Humble. Reliability of noisy quantum computing devices. *ArXiv*, abs/2307.06833, 2023.
13. https://myqlm.github.io/02_user_guide/01_write/01_digital_circuit/05_aqasm.html. Accessed: 09/04/2024.
14. https://atos.net/en/2017/press-release/general-press-releases_2017_07_04/atos-launches-highest-performing-quantum-simulator-world. Accessed: 07/04/2022.
15. https://atos.net/en/lp/myqlm. Accessed: 09/04/2024.
16. https://atos.net/en/2019/press-release_2019_05_16/atos-launches-myqlm-to-democratize-quantum-programming-for-researchers-students-and-developers-worldwide. Accessed: 07/04/2022.
17. https://atos.net/en/solutions/quantum-learning-machine. Accessed: 09/04/2024.
18. https://github.com/maheloisaf/Supplementary-Information-Grover-Paper/tree/main. Accessed: 07/04/2022.
19. Gleydson Fernandes de Jesus, Maria Heloísa Fraga da Silva, Teonas Gonçalves Dourado Netto, Lucas Queiroz Galvão, Frankle Gabriel de Oliveira Souza, and Clebson Cruz. Quantum computing: an undergraduate approach using qiskit. *Revista Brasileira de Ensino de Física*, 43:e20210033, 2021.
20. https://www.anaconda.com/download#downloads. Accessed: 09/04/2024.
21. https://docs.jupyter.org/en/latest/. Accessed: 09/04/2024.
22. https://myqlm.github.io/01_getting_started/%3Amyqlm%3A01_install.html. Accessed: 09/04/2024.
23. https://myqlm.github.io/02_user_guide/01_write/01_digital_circuit/05_aqasm.html. Accessed: 09/04/2024.
24. Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.
25. Caroline Figgatt, Dmitri Maslov, Kevin A Landsman, Norbert M Linke, Shantanu Debnath, and Christofer Monroe. Complete 3-qubit grover search on a programmable quantum computer. *Nature communications*, 8(1):1–9, 2017.
26. Jairo Ernesto Castillo, Yesenia Sierra, and Nelson L Cubillos. Classical simulation of grovers quantum algorithm. *Revista Brasileira de Ensino de Física*, 42, 2019.
27. https://www.senaicimatec.com.br/. Accessed: 09/04/2024.
28. IBM Quantum Experience https://quantum-computing.ibm.com. [Accessed: 11-November-2021].
29. Barbara M Terhal. Quantum error correction for quantum memories. *Reviews of Modern Physics*, 87(2):307, 2015.
30. Aashish A Clerk, Michel H Devoret, Steven M Girvin, Florian Marquardt, and Robert J Schoelkopf. Introduction to quantum noise, measurement, and amplification. *Reviews of Modern Physics*, 82(2):1155, 2010.
31. Rahaf Youssef. Measuring and simulating t1 and t2 for qubits. Technical report, Fermi National Accelerator Lab.(FNAL), Batavia, IL (United States), 2020.
32. Brian Rost, Barbara Jones, Mariya Vyushkova, Aaila Ali, Charlotte Cullip, Alexander Vyushkov, and Jarek Nabrzyski. Simulation of thermal relaxation in spin chemistry systems on a quantum computer using inherent qubit decoherence. *arXiv preprint arXiv:2001.00794*, 2020.
33. C. Lü, J.L. Cheng, M.W. Wu, and I.C. da Cunha Lima. Spin relaxation time, spin dephasing time and ensemble spin dephasing time in n-type GaAs quantum wells. *Physics Letters A*, 365(5-6):501–504, jun 2007.
34. JL Skinner and D Hsu. Pure dephasing of a two-level system. *The Journal of Physical Chemistry*, 90(21):4931–4938, 1986.

 doi:10.20944/preprints202405.0930.v1

35. E Ferraro, M Fanciulli, and M De Michielis. Phonon-induced relaxation and decoherence times of the hybrid qubit in silicon quantum dots. *Physical Review B*, 100(3):035310, 2019.
36. Angela Dudley, Michael Nock, Thomas Konrad, Filippus S. Roux, and Andrew Forbes. Amplitude damping of laguerre-gaussian modes. *Opt. Express*, 18(22):22789–22795, Oct 2010.