Realistic simulation in the Qadence SDK

Ethan OBADIA

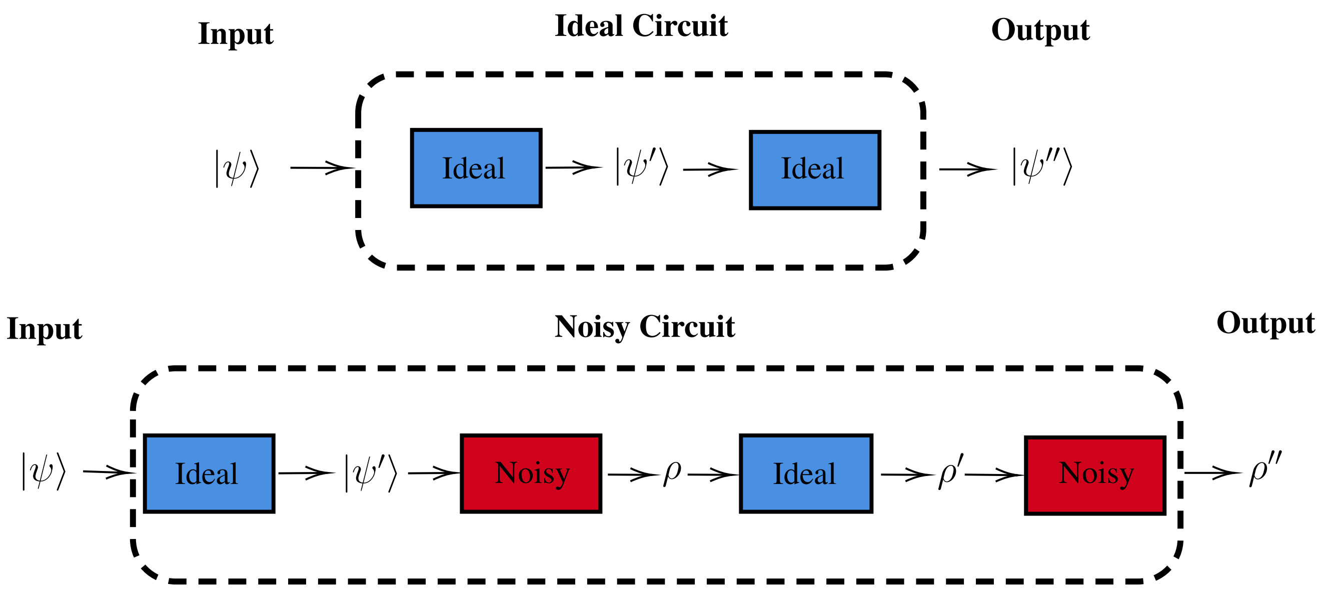


FIGURE 1 – Comparison between the simulation logics of a noiseless and noisy circuit.

Sommaire

Acknowledgments

I would like to extend my sincere gratitude to my supervisor, Dr. Roland Guichard, for giving me this great opportunity. His trust in me greatly contributed to my rapid integration into the team. I am also very grateful for his availability and willingness to share his knowledge on subjects that were new to me. I have learned a lot from him, and this exceptional experience would not have been the same without his support.

I would also like to thank the members of my team who provided invaluable assistance and, in some cases, also supervised me. Special thanks to Dominik Seitz, Gergana Velikova, and João Mouthino, whom I had the opportunity to meet during my trip to Amsterdam. Their support and collaboration were essential to the success of my internship. Finally, I would like to thank my fellow intern, Melvin Mathé, with whom I exchanged ideas about our respective topics.

Chapitre 1

Introduction

As part of the Master's internship in Fundamental and Applied Physics, specializing in Quantum Information, at Sorbonne Université, I had the opportunity to work at Pasqal, specifically in the Quantum Libraries department. Pasqal is a leading company in the field of quantum computing, specializing in the development of quantum computers using neutral atoms as qubits.
This report focuses on the work I carried out during my in-person internship. Although I was physically based in France, most of my colleagues were located abroad, notably in Amsterdam and London. This unique situation allowed me to discover and adopt new online working methods essential for effective collaboration on international research projects.

Quantum computing uses the principles of quantum physics to process information in a fundamentally different way from classical computers, opening the door to new approaches for solving certain types of problems. Unlike classical bits, which represent distinct and orthogonal states — with a bit being either in the state $|0\rangle$ or in the state $|1\rangle$ — quantum computers use qubits (quantum bits) as units of information. Qubits are quantum states that can exist in a superposition of orthogonal states, such as : $|\psi\rangle = a|0\rangle + b|1\rangle$ where $(a, b) \in \mathbb{C}^2$.
Bits are represented by the presence or absence of an electric current, while qubits can be represented in multiple ways. Pasqal has chosen to use neutral atoms [5], where the state $|0\rangle$ corresponds to the ground state and the state $|1\rangle$ to an excited state of an atom, manipulated by energy transitions.

A major challenge in quantum computing is the difficulty of isolating qubits, which are fragile systems, from external disturbances known as **noise**. Noise causes almost inevitable errors in the state of the qubits, thereby affecting their storage, manipulation, or measurement.
Currently, in the so-called NISQ era [10] (Noisy Intermediate-Scale Quantum), where errors are still prevalent and a fault-tolerant quantum computing system [12] remains a future goal, systems are particularly vulnerable to noise.
Unlike classical systems, where bits are represented by robust "on/off" states of transistors, errors are much more frequent and varied in quantum systems. Therefore, since simulations are often conducted under idealized, noise-free conditions, where the system is assumed to be perfectly isolated, it is crucial to develop noisy simulation tools for quantum computing. This will enable a better understanding of the real behavior of quantum systems, particularly when running algorithms, and help devise strategies to manage and correct errors more effectively in the future.

Thus, it is relevant to integrate into Pasqal's existing software, Qadence, the capability to simulate noisy quantum circuits using noisy digital channels. To achieve this, a quantum software developer typically follows a structured approach :

1. They become familiar with the organization of the **stack** (a set of interconnected libraries and software components) and the formalism of the libraries they are working with, including how the various libraries fit together and are coded.
2. They study the specific theory related to the objects they wish to integrate.
3. Based on their knowledge of the stack and the theoretical concepts studied, they translate this theory into concrete functionalities by integrating a prototype.
4. They test the new functionalities and write documentation to enable users to utilize their work safely and effectively.

To describe this approach, this report is organized as follows : first, we will describe the Pasqal stack to understand the libraries we intend to modify. Next, we will introduce the theoretical framework of noisy channels that we aim to use. Then, we will present the algorithm and formalism employed in integrating this new tool. Finally, an application of this tool will be illustrated through Grover's algorithm.

Chapitre 2

Stack Architecture

This chapter presents the architecture of the quantum simulation stack developed by Pasqal, focusing on Qadence and its interactions with different backends, such as Pyqtorch, to simulate quantum circuits. Understanding this architecture is crucial to identifying where modifications are needed for the integration of noisy channels.

Qadence

Pasqal, as a "full-stack company," develops both its hardware based on neutral atoms and its own software to enable simulations. Currently, Pasqal is developing an open-source Software Development Kit (SDK) written in Python, named **Qadence** [11].
In computing, an SDK is a set of tools, libraries, and documentation designed to help developers create applications for a specific platform. It simplifies development by providing pre-built components and code examples. In quantum computing, a quantum SDK enables developers to create, simulate, and run quantum algorithms and circuits, offering tools to interact with qubits and perform measurements. Qiskit [6], Cirq, and PennyLane [1] are well-known examples of quantum SDKs.

Qadence is particularly designed to handle digital-analog quantum circuits. Digital Analog Quantum Computing (DAQC) [9] is a combination of the two most developed paradigms to date : Digital Quantum Computing (DQC), which manipulates qubits using discrete quantum gates, and Analog Quantum Computing (AQC), which utilizes the continuous evolution of quantum systems under the influence of external fields. DQC is universal, meaning it can execute any quantum algorithm, but it suffers from a high error rate because it requires individual or small groups of qubits to be manipulated while isolating them from the rest of the qubit register to avoid interference. In contrast, AQC is not universal but reduces errors by leveraging the natural interactions between qubits.
DAQC takes advantage of the strengths of both approaches by combining digital gates and analog blocks, allowing for the creation of hybrid circuits. Thanks to this flexibility, Qadence allows the construction of fully digital, analog, or hybrid circuits. In this report, we will focus on examples related to fully digital circuits.

Moreover, Qadence is a quantum programming language that acts as a “**front-end language**.” This means that all the gates and blocks we create in Qadence are essentially conceptual objects without intrinsic value. These blocks serve as abstract templates, and by calling the appropriate **backends**, we give them concrete meaning for performing computations.

In the context of quantum computing, a backend refers to the infrastructure and systems that enable the execution of algorithms. The backend can be a classical simulator or a real quantum processor (QPU) accessible via the cloud. Conversely, the frontend is the user interface that allows for the visualization and management of quantum circuits and algorithms.

Qadence primarily has three backends, **Pyqtorch**, **Horqrux**, and **Pulser** [13], which it calls via APIs to execute and simulate the circuits created on its interface, allowing for both classical and quantum simulations.
Pyqtorch and Horqrux are two backends used exclusively for non-quantum simulations, meaning they do not allow connection to a real quantum computer. They are based on PyTorch and JAX, two Python libraries for machine learning that offer different approaches to quantum machine learning applications and quantum circuit parameter optimization.
This report will focus only on Pyqtorch, as it is the backend to which I contributed.
Pulser is a library developed by Pasqal to program pulse sequences for quantum computers based on neutral atoms. The qubits, encoded in the energy levels of atoms, can be manipulated through energy transitions induced by these pulses, allowing for both non-quantum simulations and execution of these sequences on real QPUs.
Pulser is a **low-level** Python library that treats qubits in a physical manner and requires knowledge of atomic physics to manipulate qubits directly using laser pulses. Each operation, such as transitioning a qubit from $|0\rangle$ to $|1\rangle$, requires defining the pulses that induce the energy transitions, reflecting the actual hardware.
On the other hand, Pyqtorch and Horqrux are higher-level libraries that abstract these physical details, treating qubits purely mathematically. In these libraries, qubits and logic gates are manipulated via functions without worrying about their physical realization. For example, an operation like the CNOT gate is directly implemented via a function in Pyqtorch and Horqrux, while in Pulser, it requires an elaborate sequence of pulses, as illustrated in Figure [2.1].

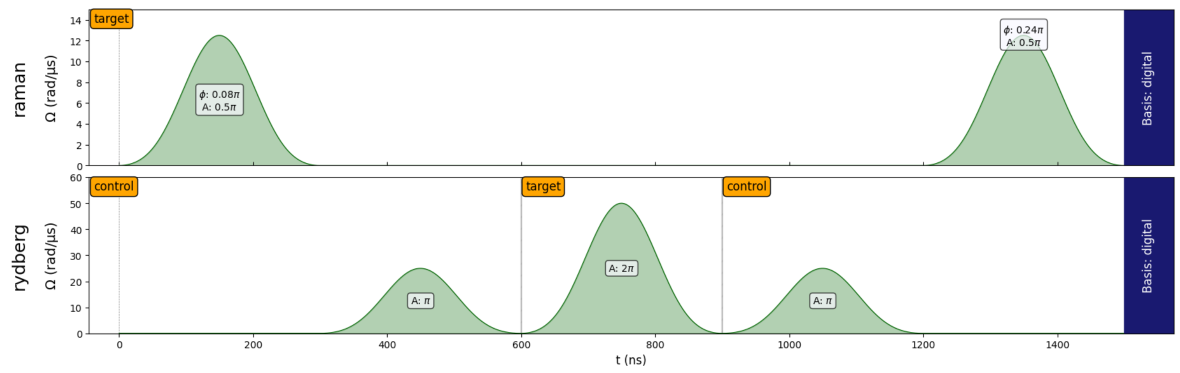


FIGURE 2.1 – Pulse sequence required to implement a CNOT gate, generated by Pulser.

Since Pyqtorch and Horqrux are standalone higher-level backends, they do not directly allow quantum simulations on a QPU. However, Qadence, as a front-end language, can connect to the appropriate backends to perform the necessary computations, enabling developers to focus on the logic of their algorithms without dealing with hardware complexity. This flexibility in backend choice simplifies development, making programming more accessible and intuitive.
In summary, as shown in Figure [2.2], Qadence's architecture is designed to create conceptual blocks that, via APIs, automatically call the appropriate backends. These backends perform the necessary cal-

culations and simulations, thereby giving meaning to the conceptual blocks created by Qadence and meeting the specific needs of the user. If the user does not manually specify a backend, the default will be Pyqtorch.

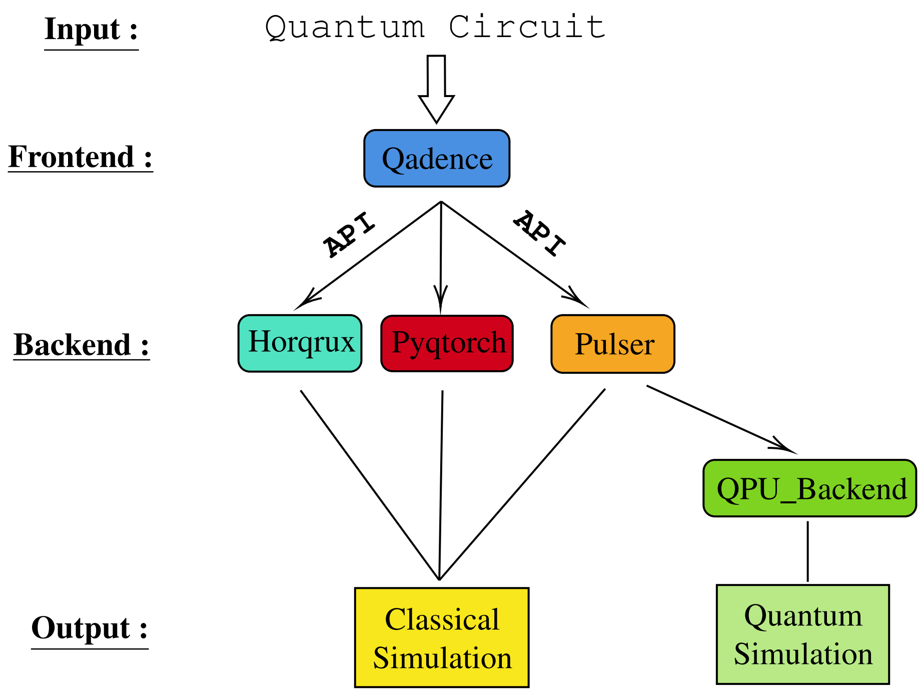


FIGURE 2.2 – Representation of the quantum simulation stack architecture with Qadence.

Pyqtorch

Pyqtorch is a state-vector simulator designed for quantum machine learning and written in PyTorch, an open-source Python machine learning library used for building and training neural networks.
A neural network is a learning model inspired by the functioning of biological neurons, consisting of interconnected layers of nodes (neurons). Each node receives inputs, applies a transformation to them, and transmits the output to the nodes of the next layer. The connections between nodes are weighted, and these weights are adjusted during training to minimize a cost function, which measures the error between the model's predictions and the expected results. In this model, inputs, weights, and outputs are all represented by **tensors**, which are multidimensional data structures similar to arrays. For example, a 2D tensor corresponds to a matrix, and a 1D tensor corresponds to a vector.

In Pyqtorch, two types of objects are distinguished : quantum states and quantum operators, both represented by PyTorch tensors of different sizes. In quantum circuits, operators are gates applied to the input state vectors of the circuit. There are several types of gates : primitive gates and parametric gates, each with its own characteristics and specific uses :

- Primitive gates are basic operations on qubits that do not depend on external parameters. For example, Pauli gates such as the X gate (or NOT), which inverts a qubit's state.
- Parametric gates allow for operations that depend on one or more parameters, providing additional customization for the rotations or transformations performed by the gate. For example, the RX(θ) gate performs a rotation of θ around the X-axis of the Bloch sphere.

Each of these categories can be controlled, where one or more control qubits determine whether the operation is applied to the target qubit. For example, the CNOT (Controlled-NOT) gate is a controlled version of the X gate, which inverts the target qubit's state only if the control qubit is in state $|1\rangle$. The same applies to a CRX(θ) gate.

To leverage PyTorch's features and represent quantum circuits in a differentiable manner, a neural network model is created where the state vectors of the qubits act as the network's inputs, the quantum gates act as neuron layers applying transformations to the states, and the weights correspond to the parameters of the gates when they are parametric. This structure allows for optimizing the parameters of a quantum circuit similarly to the optimization of weights in a classical neural network.
To represent different types of gates as layers, they are coded using classes based on the `torch.nn.Module`, a base class in PyTorch used to define and structure neural networks by encapsulating the layers, operations, and parameters necessary for the network. Thus, to initialize a gate of any type in Pyqtorch, an instance of its class is created with arguments specifying the qubits (target and/or control) and, if necessary, the relevant parameters.
When creating a class that inherits from `torch.nn.Module`, a `forward` method must be defined to specify how the data passes through the module to produce an output. This method, called the **forward pass**, describes the pathway of inputs through the network's transformations. In this context, as illustrated in Figure [2.3], the forward pass corresponds to the application of gate U on the initial state vector $|\psi\rangle$. The `forward` function is thus defined to perform the following computation :

$$|\psi'\rangle = U|\psi\rangle \tag{2.1}$$

Thanks to the ingenious formalism of tensors used in Pyqtorch, this computation can be performed without modifying the operator U, even when it does not apply to the entire system. This allows for classical simulation of the created circuits.

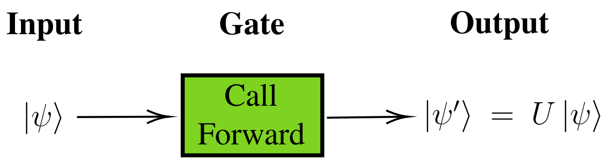


FIGURE 2.3 – Representation of the forward pass of non-noisy gates.

Chapitre 3

Noise Formalism

This chapter presents the formalism required to model errors in a quantum circuit, focusing on the representation of open quantum systems and noisy quantum channels. Understanding these theoretical concepts helps identify the necessary modifications to broaden the range of possible simulations.

Understanding these theoretical concepts makes it possible to identify what changes need to be made to adapt the simulation tools.

Open Quantum System

A physical system can be modeled as the combination of a subsystem that we wish to study and its environment, which corresponds to the surrounding subsystem, as illustrated in Figure [3.1].
To study the evolution of a physical system, two descriptions are possible :

- **Closed system** : The subsystem is considered completely and perfectly isolated from its environment. Interactions between these two subsystems are thus neglected, and the subsystem under study is treated as the total system.
- **Open system** : The subsystem is not assumed to be isolated from its environment. It is therefore necessary to take into account the interactions between the subsystem and its environment.

In general, we start by studying closed systems and then extend the study by considering the system as open, incorporating the disturbances caused by the interactions between the environment and the subsystem under study.

In quantum physics, a closed quantum system is fully represented by a **state vector** [3], which is a vector belonging to a Hilbert space — an abstract mathematical space representing all possible states of a quantum system. State vectors are denoted by kets, such as : $|\psi\rangle \in \mathcal{H}$.
The ket $|\psi\rangle$ encapsulates both the amplitude and the phase associated with the different configurations of the system. When a transformation U is applied to a state vector, it follows equation (2.1).
A state vector also represents a pure state [14], meaning a fully determined quantum state. In other words, it is a state for which we have the maximum possible information about the system, and all quantum properties are precisely defined.

However, this description is not sufficient for studying open quantum systems. When a quantum system interacts with its environment, it is generally no longer possible to describe it solely by a pure state. Indeed, quantum information can be partially exchanged with the environment, resulting in a loss of coherence that leads the system to be in a **mixed state** [8]. In a mixed state, the quantum information is no longer fully contained in a single state vector but is probabilistically distributed among several

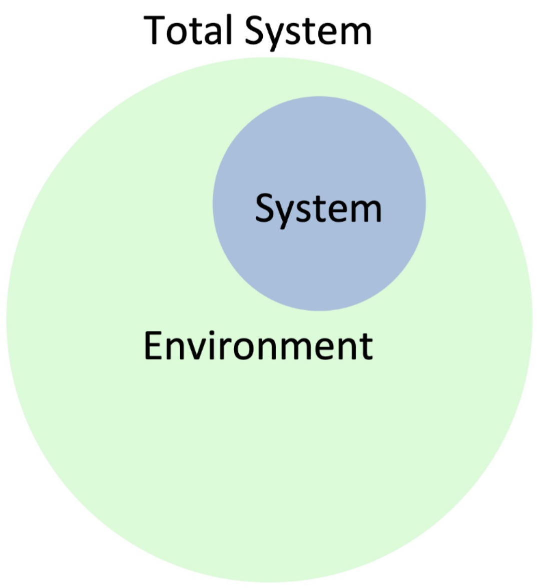


FIGURE 3.1 – Representation of the quantum system, including the subsystem under study and its environment, from [7].

pure states. To handle these more general cases, we consider a probabilistic combination p_i of possible pure states $|\psi_i\rangle$. The system is then described by a **density matrix** [3] $\rho \in \mathcal{L}(\mathcal{H})$ defined as follows :

$$\rho = \sum_i p_i |\psi_i\rangle\langle\psi_i| \tag{3.1}$$

The density matrix formalism is particularly useful for representing mixed states, but it can also be used to represent pure states. In this case, the density matrix reduces to the **projector** on that state, since the sum in equation (3.1) contains only one term, as the system is in a single possible state with certainty. Thus, for a pure state $|\psi\rangle$, the density matrix ρ is written as :

$$\rho = |\psi\rangle\langle\psi| \tag{3.2}$$

From equations (3.1) and (3.2), it is easy to see that the evolution of the density matrix ρ under a unitary transformation U is expressed in this case as :

$$\rho' = U\rho U^\dagger \tag{3.3}$$

Equation (3.3) describes how a density matrix ρ evolves deterministically in the case of a closed quantum system. In this case, the evolution is said to be **unitary** because it is reversible.

Noisy Quantum Channels

For an open quantum system that interacts with its environment, the evolution of the density matrix can no longer be described by a unitary transformation. Interactions with the environment introduce effects of **decoherence** and **dissipation**, making it necessary to use a more general description with a super-operator $S : \rho \to S(\rho)$, often called a **quantum channel**.

Quantum channels must satisfy certain properties, in particular being Completely Positive and Trace-Preserving (CPTP) [2], to maintain the probability distribution of quantum states. According to the Kraus theorem, any super-operator that is CPTP can always be decomposed as follows :

$$S(\rho) = \sum_i K_i \rho K_i^\dagger \tag{3.4}$$

where K_i are the **Kraus operators**, which characterize how the system interacts with the environment. Kraus operators are not unique but must always satisfy the following constraint :

$$\sum_i K_i K_i^\dagger = \mathbb{I}.$$

It is therefore possible to determine the dynamics of an open quantum system if a Kraus decomposition of the applied quantum channel is known.

In quantum simulations, it is often assumed that the system is closed by considering that the qubit register is completely isolated from the rest of the system and external disturbances, called noise. However, in a real quantum processor (QPU), it is impossible to ensure perfect isolation of qubits, which are fragile systems sensitive to noise. This noise causes almost inevitable and irreversible errors.
To model these disturbances in the context of an open system, we use **noisy quantum channels**, which mathematically represent the effects of noise on qubits. In our simulations, we will consider open systems to simulate digital quantum circuits. This involves focusing on digital noisy quantum channels. These channels allow realistic simulation of errors encountered on a QPU and a better understanding of their impact on quantum algorithms.

To perform these realistic simulations, several noisy quantum channels will be integrated into Pyqtorch, each representing a specific type of noise whose Kraus decomposition is known. These channels include the Bit Flip Channel, the Phase Flip Channel, the Depolarizing Channel, the Pauli Channel, the Amplitude Damping Channel, the Phase Damping Channel, and the Generalized Amplitude Damping Channel (for more details on these channels, see Appendix A).

Chapitre 4

Integration Strategy

This chapter outlines the strategy followed for integrating noisy quantum channels into Pyqtorch. Since Pyqtorch is a backend for Qadence, these modifications enable the addition of digital noise to the Qadence interface, thereby expanding its simulation capabilities.

Density Matrix

Pyqtorch was initially designed as a state vector simulator, where the inputs and outputs of quantum circuits are pure states. To integrate noisy quantum channels, it is essential to reconsider how these states are manipulated within the library.

In a typical noisy quantum circuit simulation, as illustrated in Figure [4.1], the circuit begins with a state vector as input. When a noise-free quantum gate is applied, the simulation uses the `forward` method, shown in Figure [2.3], to apply a unitary transformation corresponding to this gate on the state. However, as soon as a noisy gate is encountered, the system transitions from a closed system description to an open system description. At this point, it becomes necessary to switch from a state vector representation to a density matrix representation.

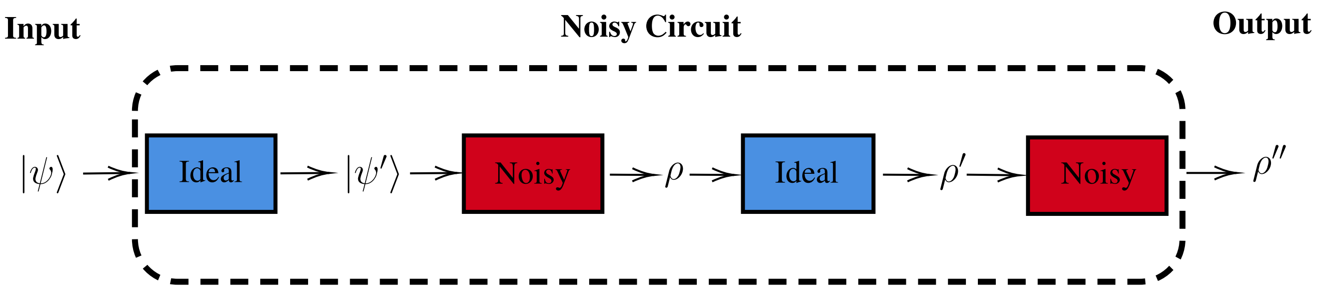


FIGURE 4.1 – Simulation logic representation of a noisy circuit.

Once the density matrix has been transformed by a non-unitary operation, it is impossible to revert to a pure state because the density matrix can now describe a mixed state. This implies that once a noisy gate is applied, the circuit must remain in the density matrix formalism until the end of the simulation.

The first step in integrating noisy gates into Pyqtorch is therefore to enable the library to support density matrices. This requires making all the existing gates, i.e., noise-free gates, compatible with this formalism. One of the major challenges of this extension is that, until now, Pyqtorch represented quantum states solely by vectors, i.e., tensors of specific sizes, while quantum operators were represented separately. Now that the density matrix, which is itself an operator, must be considered as a state representation, it is necessary to adapt Pyqtorch to understand and handle this paradigm shift.

To integrate the density matrix formalism into Pyqtorch, we start by creating a new type, `DensityMatrix`, allowing quantum gates to distinguish whether the input state is a state vector or a density matrix.
For noise-free gates, it is essential to adapt them to support density matrices as input and output an evolved density matrix. As illustrated in Figure [4.2], the `forward` function of ideal gates must be modified to include two distinct paths :

- The first path, already implemented, is applied when the input is a state vector, following the evolution given by equation (2.1).
- The second path is used when the input is a density matrix, following the evolution described by equation (3.3).

To enable the evolution of density matrices in a closed quantum circuit, it is necessary to develop specific functions capable of manipulating these matrices and performing multiplications between operators.
In Pyqtorch, there is already an established formalism where operators are represented by matrices, while state vectors are represented by tensors of specific sizes, designed to allow efficient targeting of qubits on which a quantum gate must be applied. However, for density matrices, which are operators, this targeting method cannot be directly applied.
To perform matrix multiplication, such as the one used in equation (3.3), it is imperative that the matrices have compatible dimensions. To make the multiplication possible, it is necessary to extend the dimension of the quantum gate operator to the entire system.
This extension is achieved by "promoting" the operator U to the full size of the system. This is done by multiplying the operator by identities I on the qubits not affected by the gate, as illustrated by the following equation :

$$U_{prom} = I \otimes \cdots \otimes U \otimes \cdots \otimes I \tag{4.1}$$

Where U_{prom} is the promoted operator, and U is the operator representing the quantum gate applied only to the targeted qubits.

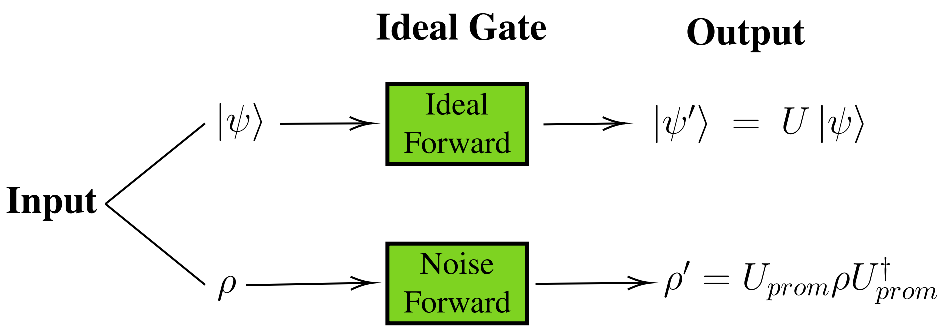

Ideal Gate
Output
Input
$|\psi\rangle$
Ideal Forward
$|\psi'\rangle = U|\psi\rangle$
ρ
Noise Forward
$\rho' = U_{prom}\rho U_{prom}^{\dagger}$

FIGURE 4.2 – Representation of the forward pass of noise-free gates supporting density matrices.

Noisy Quantum Gates

After modifying the existing quantum gates in Pyqtorch to support density matrices in the context of closed systems, we now focus on integrating realistic simulations by introducing noisy gates. To do this, a new class, `Noise`, is created as a subclass of `torch.nn.Module`, similar to the other gates in Pyqtorch (see the complete code in Appendix B).
The `Noise` class is designed to initialize noisy gates with new attributes such as the Kraus operators, which describe the type of noise applied, and the probability associated with the occurrence of this noise. For example, in the case of a `BitFlip` gate, this probability corresponds to the likelihood that the Pauli X gate is applied to the qubit. By creating subclasses of `Noise`, we can define the Kraus operators for each type of noisy gate. This modular approach allows the integration of predefined noisy gates and the design of custom gates from a given set of Kraus operators.
Unlike parametric gates, noisy gates do not have parameters to optimize. They are registered as `buffers`, so although they are represented as layers in the neural network, their weights are not adjusted during training.
The `forward` function of the `Noise` class can receive an input that is either a state vector $|\psi\rangle$ or a density matrix ρ. If the input is a pure state vector, it is converted into a density matrix following equation (3.2) to match the formalism used in an open system. Then, `forward` determines the evolution dictated by equation (3.4). Figure [4.3] illustrates this process.

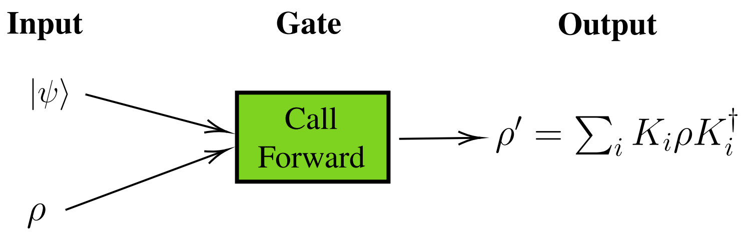


FIGURE 4.3 – Representation of the forward pass of noisy gates

To enable a smoother integration of noisy gates in Pyqtorch, a new syntax has been adopted. Although it is possible to initialize noisy gates individually in a circuit, the preferred method is to treat these gates as **noise protocols** associated with ideal gates.
In this approach, a new attribute named `noise` is added to ideal gates. It is initialized by default to `None`, indicating that no noise is applied and that the gate remains ideal. However, this attribute can also be set as an instance of a new class `NoiseProtocol`, which defines a specific noise protocol. This class allows the definition of the noise type, the associated probability, and the qubits on which the noise should be applied. For more complex configurations, the `noise` attribute can also be a dictionary containing multiple instances of `NoiseProtocol`, allowing multiple types of noise to be associated with a single gate. This is demonstrated in the example below :

```python
bitflip_noise = Noisy_protocols(protocol=Noisy_protocols.BITFLIP, options=
    {"error_probability": 0.2})
depolarizing_noise = Noisy_protocols(protocol= Noisy_protocols.DEPOLARIZING
    , options= {"error_probability": 0.4, "target": 0})
noises = {
    "bitflip": bitflip_noise,
    "depolarizing": depolarizing_noise
}
x_noisy = X(target=1, noise=noises)
```

This noise management is more realistic, as on a QPU, an error often repeats for the same type of gate. This approach links an ideal gate to its noisy version in a single command, faithfully reflecting real behaviors.
To integrate this new logic, it is necessary to modify the `forward` function of the existing ideal gates to support the noise attribute and handle the applied noise protocol. Algorithm [1] describes the final version of this function in pseudocode.

Algorithm 1 Ideal forward function after the noise integration

```
1:  **Input :** Initial state : $|\psi\rangle$ or $\rho$
2:  **Output :** Evolved state : state_evolved
3:
4:  if Noise is present then
5:      if State is a pure state (not a density matrix) then
6:          Convert State to density matrix : $\rho = |\psi\rangle \langle\psi|$
7:      end if
8:      Apply the ideal gate to the state : $\rho = U_{prom}\rho U_{prom}^{\dagger}$
9:      if Noise is a dictionary (multiple noises) then
10:         for each noise gate in Noise do
11:             Apply the forward pass of the noise gate to $\rho$
12:             Update $\rho$ from noise gate
13:         end for
14:         Return $\rho$
15:     else
16:         Apply the forward pass of the single noise gate to $\rho$
17:         Return $\rho$
18:     end if
19: else
20:     if State is a density matrix then
21:         Compute the evolved state : $\rho = U\rho U^{\dagger}$
22:         Return $\rho$
23:     else
24:         Compute the evolved state : $|\psi\rangle = U |\psi\rangle$
25:         Return $|\psi\rangle$
26:     end if
27: end if
```

Chapitre 5

Application to Grover's Algorithm

This chapter demonstrates the application of the functionalities integrated into Pyqtorch to enable realistic simulations. Specifically, we perform noisy simulations of Grover's algorithm and compare the results obtained with ideal simulations, thereby contributing to the development of the documentation for these libraries.

Theory

Grover's algorithm [4] is a quantum algorithm designed to accelerate the search for an element in an unstructured database. In a classical context, searching for an element in a database of size N requires examining an average of $N/2$ elements to ensure a match, equivalent to $O(N)$ searches. Grover's algorithm, by exploiting the unique properties of quantum physics, such as superposition and interference, reduces the number of evaluations required to just $O(\sqrt{N})$. With this quadratic speedup, the algorithm reaches a solution much faster than classical methods.

In this algorithm, each element of the database is represented by a qubit in the computational basis. For a database of size $N = 2^n$, n qubits are required for encoding.

The different steps followed during the execution of Grover's algorithm are :

1. **Initial Superposition :** The algorithm begins by applying Hadamard gates H to each qubit in the initial state $|0\rangle$, creating a quantum superposition where all possible states are represented with equal probability amplitudes.
2. **Application of the Quantum Oracle :** The oracle identifies the target state by inverting its phase, changing it from $+1$ to -1, without affecting the other states.
3. **Amplitude Amplification :** The algorithm applies a series of operations to amplify the amplitude of the state marked by the oracle and reduce those of the others. This process is repeated about $O(\sqrt{N})$ times to maximize the probability of measuring the target state.
4. **Final Measurement :** Once the optimal number of iterations is reached, a measurement is performed on the quantum state. Due to the amplification steps, it is highly likely that the measurement will reveal the state corresponding to the desired solution.

We now focus on the specific search for the data element corresponding to the qubit state $|011\rangle$. To do this, following the steps presented above, we construct the quantum circuit represented in Figure [5.1].

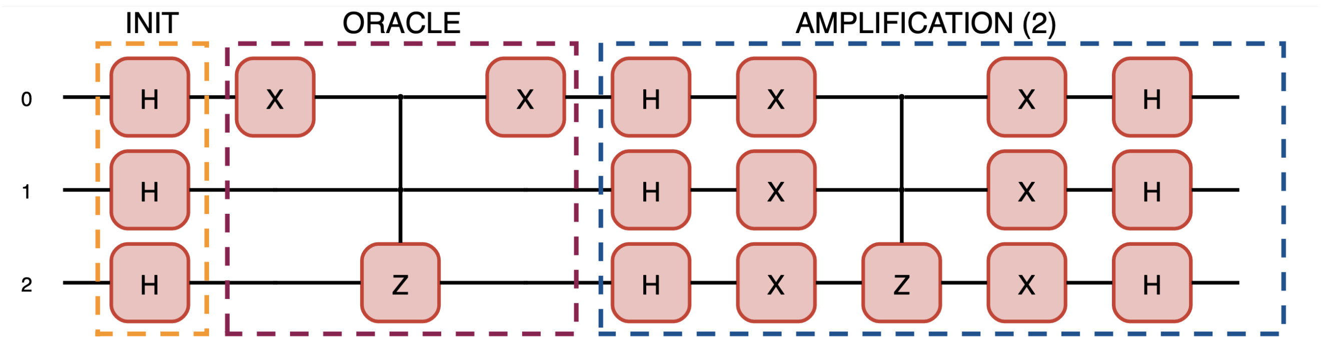


FIGURE 5.1 – Representation generated by Qadence of Grover's algorithm circuit for searching the state $|011\rangle$ (see Appendix C for more details).

The circuit performs two repetitions of amplitude amplification. This decision is related to the fact that, with three qubits, the database contains $2^3 = 8$ states. Therefore, the optimal number of iterations to maximize the probability of finding the target state is on the order of $\sqrt{2^3} = \sqrt{8}$, or about two iterations.

Simulations

After integrating new functionalities into Pyqtorch to handle density matrices, it is essential to validate each newly created function through unit tests. To do this, tests have been developed using the **Pytest** library to verify that certain specific assertions are met, thereby ensuring the reliability of the code.
In addition to unit tests, it is necessary to ensure the consistency of the results obtained by the overall implementation. In this context, a simulation of Grover's circuit, presented in Figure [5.1], was carried out. This simulation compares the results obtained from density matrices provided as input with those produced from state vectors, both methods being expected to yield identical results under ideal conditions.

To simulate Grover's algorithm, a `sampling` method is used to handle the probabilistic nature of the measurements. For all simulations presented in this report, 20,000 samples were chosen, a number that offers a compromise between complexity and uncertainty. Figure [5.2] presents a comparison of the samples obtained via simulation with a state vector and those obtained with density matrices. The similar results confirmed by this figure demonstrate that Pyqtorch is capable of effectively handling density matrices.

As part of the contribution to the documentation of Qadence and Pyqtorch, we present the characteristics of the newly implemented noisy channels. For this purpose, we imagine a scenario of realistic

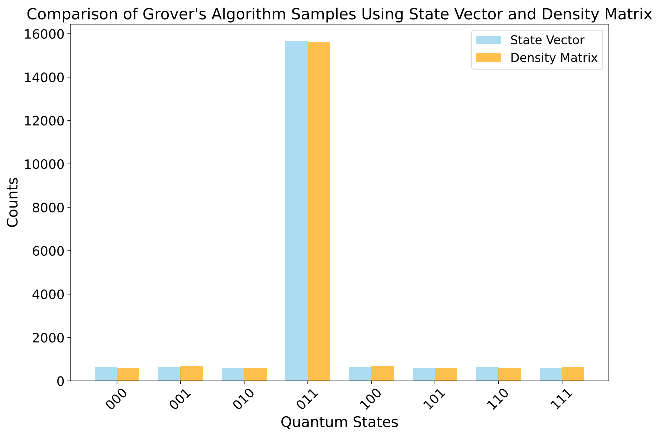


FIGURE 5.2 – Comparison of samples from Grover's algorithm using state vectors and density matrices.

simulations of the circuit presented in Figure [5.1], where the QPU has a low fidelity on the X gate, meaning it is not executed ideally. To model this imperfection, different noise protocols were applied specifically to all the X gates in the circuit.
The simulation results are presented in the form of histograms, where each bar represents the occurrence of measured states at the end of each sample, as a function of the error probability applied at the start of the simulation. This probability gradually increases, starting from the ideal case ($p = 0$) to the completely noisy case ($p = 1$).

We start by studying the BitFlip channel, which models the error where the state of a qubit is flipped by applying, with a certain probability p, the Pauli X operator. This operation is illustrated in Figure [5.3].

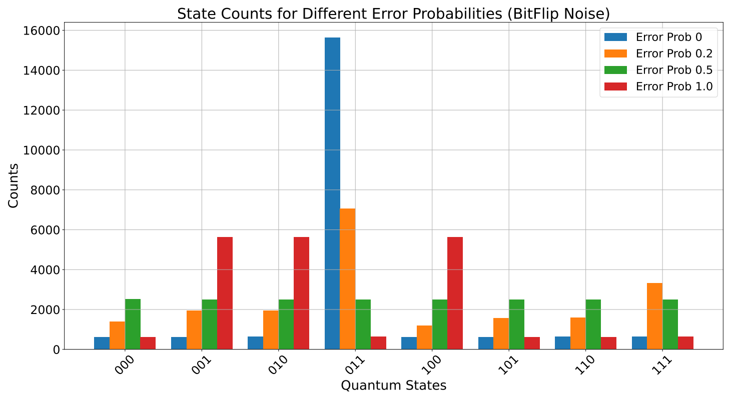


FIGURE 5.3 – Study of the characteristics of the BitFlip channel in the context of Grover's algorithm simulation to determine the state $|011\rangle$.

It can be observed that in the limiting case where the error probability reaches 1, meaning the BitFlip channel consistently applies the qubit's state inversion, the original circuit is completely altered. At this probability, each initially applied X gate is immediately canceled out by another noise-induced X gate, effectively applying the identity ($XX^\dagger = I$).
Under these conditions, the circuit's initialization step is unaffected by noise since it does not contain

any X gates. However, during the oracle step, only the CZ gate remains, resulting in incorrect marking of the target state. Indeed, instead of marking the expected state, the oracle modifies the phase of state $|111\rangle$, generating the following state :

$$|\psi_{\text{mark}}\rangle = \frac{1}{\sqrt{8}} \left(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle - |111\rangle \right).$$

The amplitude amplification step, not targeting the correct state, enhances the amplitudes of the undesired states $|001\rangle$, $|010\rangle$, and $|100\rangle$, producing the output state :

$$|\psi_{\text{out}}\rangle = \frac{1}{4\sqrt{2}} \left(|000\rangle + 3|001\rangle + 3|010\rangle + |011\rangle + 3|100\rangle + |101\rangle + |110\rangle - |111\rangle \right).$$

However, the case $p = 1$ is unrealistic, as, in practice, gates are used with a certain fidelity. A gate with fidelity below 50% would not be selected to execute an algorithm, as it would cause too many errors. Thus, gates with fidelity around 0.8, corresponding to an error probability of 0.2, are preferred. Within this range, it is observed that up to probability $p = 0.5$, the target state $|011\rangle$ is less frequently observed, indicating that the effect of BitFlip noise gradually disrupts the circuit's ability to preserve this target state. As p approaches 0.5, each qubit has an equal chance of being in the state $|0\rangle$ or $|1\rangle$. At this stage, the system reaches a **maximally mixed state**, where all configurations of quantum states are observed with equal probability. This situation reflects a total loss of coherence and information about the system's initial state, as the mixing is such that no particular state is favored over the others.

Next, we examine the PhaseFlip channel, which models the error where the phase of a qubit is flipped by applying, with a certain probability p, the Pauli Z operator. This operation is illustrated in Figure [5.4].

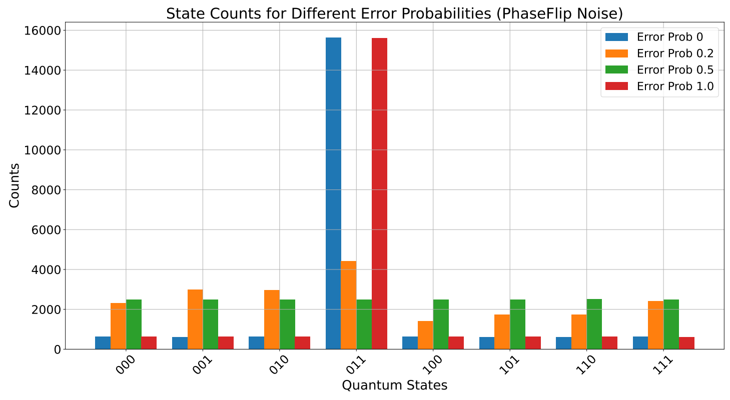


FIGURE 5.4 – Study of the characteristics of the PhaseFlip Noise channel in the context of Grover's algorithm simulation to determine the state $|011\rangle$.

Like the BitFlip channel, the PhaseFlip channel leads to a maximally mixed state when the error probability reaches $p = 0.5$. However, PhaseFlip tends more quickly toward this total mixture state. This phenomenon is explained by the fact that this noise does not affect the amplitude of the states like BitFlip does but rather modifies their phase by introducing a sign difference.
Grover's algorithm's oracle, which operates by marking the target state by changing its phase, is particularly sensitive to PhaseFlip-type noise. This phase alteration significantly disrupts the oracle's marking mechanism, making it ineffective. Consequently, the probability of measuring the target state is much

more affected with PhaseFlip than with BitFlip.

Furthermore, the same behavior is observed between the ideal case and the fully noisy case. Indeed, when $p = 1$, each qubit undergoes an even number of phase flips, mutually canceling out the effects of each phase flip. As a result, these inversions have no net impact on the gates present in the circuit.

Figure [5.5] shows the study of the circuit for a Depolarizing noise protocol. This channel models a situation where a qubit is randomly mixed by applying the three Pauli operators (X, Y, and Z).

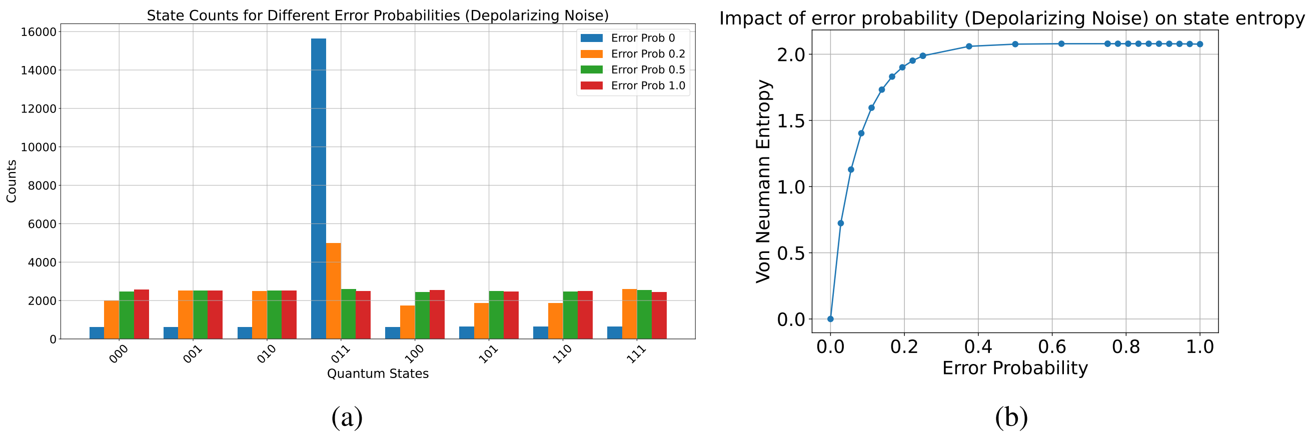


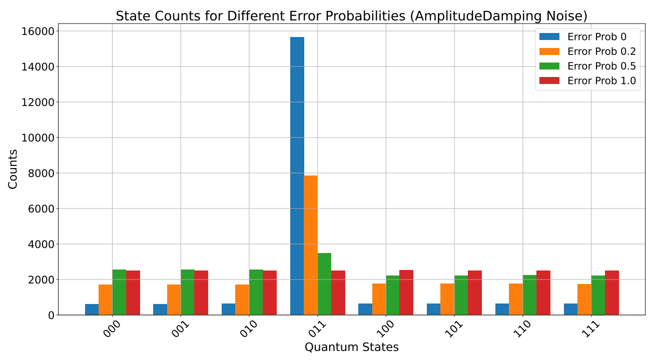
FIGURE 5.5 – Study of the characteristics of the Depolarizing Noise channel in the context of Grover's algorithm simulation to determine the state $|011\rangle$.

To explain the rapid convergence to the maximally mixed state in Figure [5.5a], one can refer to the evolution of the von Neumann entropy of the output state, represented in Figure [5.5b]. This measure helps understand how the state of the quantum system is affected by Depolarizing noise, particularly when the error probability varies.
The von Neumann entropy, denoted as $S(\rho)$, is defined for a density matrix ρ representing the quantum state of the system by the expression :

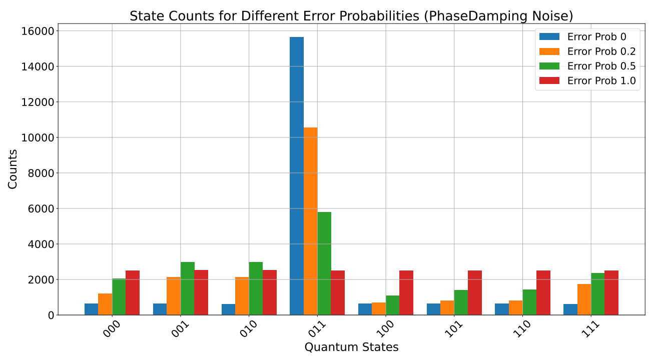
$$S(\rho) = -\mathrm{Tr}(\rho \log \rho) \tag{5.1}$$

This entropy quantifies the degree of uncertainty or mixing of a quantum state. It is zero for a pure state and becomes positive for a mixed state, increasing with the level of mixing of quantum states.
As shown in Figure [5.5b], starting from $p = 0.25$, the Depolarizing channel already drives the quantum system to a maximally mixed state, where the von Neumann entropy quickly reaches its maximum. This suggests that the goal of this noise channel is to maximize coherence loss as soon as the noise probability exceeds a relatively low threshold. Indeed, Figure [5.5a] shows that as soon as p reaches 0.2, the distribution of measured states becomes nearly uniform, indicating that the system has lost all useful quantum information long before the noise probability reaches equiprobability, as was the case for the previously presented channels. This demonstrates that the Depolarizing channel is particularly created to model decoherence.

Finally, we focus on Damping-type channels that model dissipation phenomena, represented in Figure [5.6].
The results obtained with the BitFlip and PhaseFlip channels differ significantly from those of the Damping-type channels due to the very nature of these quantum noises. The Flip-type channels, such as



(a)



(b)

FIGURE 5.6 – Study of the characteristics of the AmplitudeDamping and PhaseDamping channels in the context of Grover's algorithm simulation to determine the state $|011\rangle$.

BitFlip and PhaseFlip, model discrete and **reversible** errors that do not result in an irreversible loss of information, as their effects can be undone with an appropriate probability. In contrast, Damping-type channels, like PhaseDamping and AmplitudeDamping, represent processes of **irreversible decoherence and dissipation**, where the loss of coherence due to interactions with the environment cannot be restored.

These differences prevent the appearance of a minimum at probability $p = 0.5$ in Figure [5.6], unlike what is observed in Figures [5.3] and [5.4].
In this case, coherence loss occurs in a continuous and cumulative manner as the noise is applied, making it impossible to restore the information simply by reversing the noise process. Thus, instead of reaching a minimum at $p = 0.5$, the von Neumann entropy continues to increase until it reaches its maximum when the noise probability reaches 1. This irreversible nature of the Damping channels makes them much more destructive for quantum information, as each interaction with the environment results in a loss of coherence that cannot be recovered. This characteristic also makes them more realistic for modeling real quantum interactions, such as those involving qubits represented by atoms, where decoherence and dissipation effects naturally occur due to constant interactions with the environment.

Ultimately, these simulations highlight the definitions and characteristics of the different noise channels implemented to facilitate their understanding and use for Qadence and Pyqtorch users.

Chapitre 6

Conclusion

This report provides a detailed description of the method used for integrating noisy quantum gates into Qadence and Pyqtorch.

To integrate noisy quantum gates into Qadence, it was necessary to create and modify its established conceptual block structure and establish new API connections with its backend.
In Pyqtorch, it required expanding and modifying the design of this library to support the formalism of open quantum systems and, consequently, the density matrix formalism. This process involved developing an integration prototype based on a solid understanding of the theoretical framework and the functioning of the software stack.

This approach allowed us, through realistic simulations of Grover's algorithm, to test the reliability of the implementation and understand the properties of the predefined quantum channels during integration. This work will serve as a basis for the documentation of Pyqtorch and Qadence regarding noisy simulations. Currently, the code for the figures generated as part of this project is available in a GitHub repository at the following address : Qadence Digital Noise.

The new functionalities have expanded the range of simulation possibilities offered. These developments open up new avenues for research and simulation for users.

Finally, future improvements could be considered to optimize the implemented code, particularly regarding the multiplication of operators. Currently, this operation relies on promoting operators to the full size of the system, as presented in equation (4.1). This approach generates large tensors that are costly in terms of manipulation and memory. A possible improvement would be to develop a method that allows operators to be applied directly to the relevant qubits. Although the simulations performed have shown smooth execution, this optimization would facilitate the simulation of circuits involving a large number of qubits.

Bibliographie

[1] Ville BERGHOLM et al. *PennyLane : Automatic differentiation of hybrid quantum-classical computations*. arXiv :1811.04968 [physics, physics :quant-ph]. Juill. 2022. DOI : 10.48550/arXiv.1811.04968. URL : http://arxiv.org/abs/1811.04968.

[2] Man-Duen CHOI. “Completely positive linear maps on complex matrices”. In : *Linear Algebra and its Applications* 10.3 (juin 1975), p. 285-290. ISSN : 0024-3795. DOI : 10.1016/0024-3795(75)90075-0. URL : https://www.sciencedirect.com/science/article/pii/0024379575900750.

[3] Claude COHEN-TANNOUDJI, Bernard DIU et Franck LALOË. *Mécanique Quantique - Tome 1 : Nouvelle édition*. EDP Sciences. ISBN : 978-2-7598-2288-1. DOI : 10.1051/978-2-7598-2288-1.

[4] Lov K. GROVER. *A fast quantum mechanical algorithm for database search*. arXiv :quant-ph/9605043. Nov. 1996. DOI : 10.48550/arXiv.quant-ph/9605043. URL : http://arxiv.org/abs/quant-ph/9605043.

[5] Loic HENRIET et al. “Quantum computing with neutral atoms”. In : *Quantum* 4 (sept. 2020). arXiv :2006.12326 [quant-ph], p. 327. ISSN : 2521-327X. DOI : 10.22331/q-2020-09-21-327. URL : http://arxiv.org/abs/2006.12326.

[6] Ali JAVADI-ABHARI et al. *Quantum computing with Qiskit*. arXiv :2405.08810 [quant-ph]. Juin 2024. DOI : 10.48550/arXiv.2405.08810. URL : http://arxiv.org/abs/2405.08810.

[7] Daniel MANZANO. “A short introduction to the Lindblad Master Equation”. In : *AIP Advances* 10.2 (fév. 2020). arXiv :1906.04478 [cond-mat, physics :quant-ph], p. 025106. ISSN : 2158-3226. DOI : 10.1063/1.5115323. URL : http://arxiv.org/abs/1906.04478.

[8] Michael A. NIELSEN et Isaac L. CHUANG. *Quantum Computation and Quantum Information : 10th Anniversary Edition*. 1re éd. Cambridge University Press. ISBN : 978-1-107-00217-3 978-0-511-97666-7. DOI : 10.1017/CBO9780511976667.

[9] Adrian PARRA-RODRIGUEZ et al. “Digital-Analog Quantum Computation”. In : *Physical Review A* 101.2 (fév. 2020). arXiv :1812.03637 [cond-mat, physics :quant-ph], p. 022305. ISSN : 2469-9926, 2469-9934. DOI : 10.1103/PhysRevA.101.022305. URL : http://arxiv.org/abs/1812.03637.

[10] John PRESKILL. “Quantum Computing in the NISQ era and beyond”. In : *Quantum* 2 (août 2018). arXiv :1801.00862 [cond-mat, physics :quant-ph], p. 79. ISSN : 2521-327X. DOI : 10.22331/q-2018-08-06-79. URL : http://arxiv.org/abs/1801.00862.

[11] Dominik SEITZ et al. *Qadence : a differentiable interface for digital-analog programs*. arXiv :2401.09915 [quant-ph]. Jan. 2024. DOI : 10.48550/arXiv.2401.09915. URL : http://arxiv.org/abs/2401.09915.

[12] Peter W. SHOR. *Fault-tolerant quantum computation*. arXiv :quant-ph/9605011. Mars 1997. DOI : 10.48550/arXiv.quant-ph/9605011. URL : http://arxiv.org/abs/quant-ph/9605011.

[13] Henrique SILVÉRIO et al. “Pulser : An open-source package for the design of pulse sequences in programmable neutral-atom arrays”. In : *Quantum* 6 (jan. 2022). arXiv :2104.15044 [quant-ph], p. 629. ISSN : 2521-327X. DOI : 10.22331/q-2022-01-24-629. URL : http://arxiv.org/abs/2104.15044.

[14] Mark M. WILDE. *From Classical to Quantum Shannon Theory*. DOI : 10.1017/9781316809976.001. URL : http://arxiv.org/abs/1106.1445.

Appendices

Annexe A

This appendix presents the different noisy quantum channels [8] that have been integrated into Pyqtorch to perform realistic simulations of quantum circuits. Each channel models a specific type of quantum noise whose Kraus decomposition is known, allowing for the analysis and correction of noise effects on quantum systems. The implemented channels are as follows :

— **Bit Flip Channel :** This channel models an error where the state of a qubit is flipped from $|0\rangle$ to $|1\rangle$ or from $|1\rangle$ to $|0\rangle$ with probability p. Mathematically, it is defined by :

$$\textbf{BitFlip}(\rho) = (1-p)\rho + pX\rho X^\dagger,$$

where X is the Pauli-X operator (NOT gate), representing a flip of the qubit state.

— **Phase Flip Channel :** This channel models an error where the phase of a qubit is flipped, meaning that the state $|0\rangle$ remains unchanged while the state $|1\rangle$ acquires a phase of -1. This process occurs with probability p. It is defined by :

$$\textbf{PhaseFlip}(\rho) = (1-p)\rho + pZ\rho Z^\dagger,$$

where Z is the Pauli-Z operator, representing a phase flip.

— **Depolarizing Channel :** This channel models a situation where the state of a qubit is randomly mixed with the three Pauli operators (X, Y, Z), with probability p, resulting in a total loss of quantum information. It is defined by :

$$\textbf{Depolarizing}(\rho) = (1-p)\rho + \frac{p}{3}(X\rho X^\dagger + Y\rho Y^\dagger + Z\rho Z^\dagger),$$

where X, Y, and Z are the Pauli operators.

— **Pauli Channel :** This channel generalizes the depolarizing channel by allowing distinct probabilities p_x, p_y, and p_z for applying the Pauli operators X, Y, and Z. It is defined by :

$$\textbf{PauliChannel}(\rho) = (1-p_x-p_y-p_z)\rho + p_xX\rho X^\dagger + p_yY\rho Y^\dagger + p_zZ\rho Z^\dagger.$$

— **Amplitude Damping Channel :** This channel models the energy loss of a qubit to its environment, such as in the relaxation process to the ground state. It is defined by :

$$\textbf{AmplitudeDamping}(\rho) = K_0\rho K_0^\dagger + K_1\rho K_1^\dagger,$$

where the Kraus operators K_0 and K_1 are given by :

$$K_0 = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1-\gamma} \end{pmatrix}, \quad K_1 = \begin{pmatrix} 0 & \sqrt{\gamma} \\ 0 & 0 \end{pmatrix},$$

with γ representing the probability of energy dissipation.

— **Phase Damping Channel :** This channel models the loss of phase coherence without energy loss, often associated with phase decoherence. It is defined by :

$$\textbf{PhaseDamping}(\rho) = K_0 \rho K_0^\dagger + K_1 \rho K_1^\dagger,$$

where the Kraus operators K_0 and K_1 are given by :

$$K_0 = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1-\gamma} \end{pmatrix}, \quad K_1 = \begin{pmatrix} 0 & 0 \\ 0 & \sqrt{\gamma} \end{pmatrix},$$

with γ representing the probability of phase coherence loss.

— **Generalized Amplitude Damping Channel :** This channel models the energy exchange of a qubit at non-zero temperature, where it can both gain and lose energy in interaction with its environment. It is defined by :

$$\textbf{GeneralizedAmplitudeDamping}(\rho) = K_0 \rho K_0^\dagger + K_1 \rho K_1^\dagger + K_2 \rho K_2^\dagger + K_3 \rho K_3^\dagger,$$

with the Kraus operators :

$$\begin{cases} K_0 = \sqrt{p} \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1-\gamma} \end{pmatrix}, & K_1 = \sqrt{p} \begin{pmatrix} 0 & \sqrt{\gamma} \\ 0 & 0 \end{pmatrix}, \\ K_2 = \sqrt{1-p} \begin{pmatrix} \sqrt{1-\gamma} & 0 \\ 0 & 1 \end{pmatrix}, & K_3 = \sqrt{1-p} \begin{pmatrix} 0 & 0 \\ \sqrt{\gamma} & 0 \end{pmatrix}, \end{cases}$$

where p represents the probability of energy dissipation and γ the probability of energy transfer between the qubit and the environment.

Annexe B

This appendix presents the code of the Noise class, used to integrate noisy quantum gates into Pyqtorch. This class inherits from torch.nn.Module and allows the application of noise channels using Kraus operators. It is designed to be modular and flexible, facilitating the addition of new noisy gates and their management in simulations. It is important to note that Pyqtorch is constantly evolving, and this version may change over time to adapt to new requirements and improvements.

```python
class Noise(torch.nn.Module):
    def __init__(
        self,
        kraus: list[Tensor],
        target: int,
        error_probability: tuple[float, ...] | float,
    ) -> None:
        super().__init__()
        self.target: int = target
        self.qubit_support: tuple[int, ...] = (self.target,)
        for index, tensor in enumerate(kraus):
            self.register_buffer(f"kraus_{index}", tensor)
        self._device: torch.device = kraus[0].device
        self.probabilities: tuple[float, ...] | float = error_probability

    def extra_repr(self) -> str:
        return f"qubit_support = {self.qubit_support}, error_probability =
            {self.probabilities}"

    @property
    def kraus_operators(self) -> list[Tensor]:
        return [getattr(self, f"kraus_{i}") for i in range(len(self.
            _buffers))]

    def unitary(self, values: dict[str, Tensor] | Tensor = dict()) -> list[
        Tensor]:
        # Since PyQ expects tensor.Size = [2**n_qubits, 2**n_qubits,
            batch_size].
        return [kraus_op.unsqueeze(2) for kraus_op in self.kraus_operators]

    def forward(
        self, state: Tensor, values: dict[str, Tensor] | Tensor = dict()
    ) -> Tensor:
        """
        Applies a noisy quantum channel on the input state.
        The evolution is represented as a sum of Kraus operators:
        .. math::
            S(\\rho) = \\sum_i K_i \\rho K_i^\\dagger,

        Each Kraus operator in the `kraus_list` is applied to the input
           state, and the result
        is accumulated to obtain the evolved state.

        Args:
            state (Tensor): Input quantum state represented as a tensor.

        Returns:
            Tensor: Quantum state as a density matrix after evolution.

        Raises:
            TypeError: If the input `state` or `kraus_list` is not a Tensor
                .
        """
        n_qubits = int(log2(state.size(1)))
        rho_evols: list[Tensor] = []
        for kraus in self.tensor(values, n_qubits):
            rho_evol: Tensor = apply_density_mat(kraus, state)
            rho_evols.append(rho_evol)
        rho_final: Tensor = torch.stack(rho_evols, dim=0)
        rho_final = torch.sum(rho_final, dim=0)
        return rho_final

    @property
    def device(self) -> torch.device:
        return self._device

    def to(self, device: torch.device) -> Noise:
        super().to(device)
        self._device = device
        return self

    def tensor(self, values: dict[str, Tensor] = {}, n_qubits: int = 1) ->
       list[Tensor]:
        blockmats = self.unitary(values)
        mats = []
        for blockmat in blockmats:
            if n_qubits == 1:
                mats.append(blockmat)
            else:
                full_sup = tuple(i for i in range(n_qubits))
                support = tuple(sorted(self.qubit_support))
                mat = (
                    IMAT.clone().to(self.device).unsqueeze(2)
                    if support[0] != full_sup[0]
                    else blockmat
                )
                for i in full_sup[1:]:
                    if i == support[0]:
                        other = blockmat
                        mat = torch.kron(mat.contiguous(), other.contiguous
                            ())
                    elif i not in support:
                        other = IMAT.clone().to(self.device).unsqueeze(2)
                        mat = torch.kron(mat.contiguous(), other.contiguous
                            ())
                mats.append(mat)
        return mats
```

The main functions of this class are described below :

- `__init__` : Initializes the `Noise` class with the necessary parameters to define the type of noise :
  - `kraus` : List of Kraus operators that characterize the noise.
  - `target` : The target qubit on which the noise is applied.
  - `error_probability` : The error probability associated with the noise.
- `forward` : Computes the evolution of the quantum state under the effect of noise. The method successively applies each Kraus operator to the input state and accumulates the results to obtain the final evolved state, see Figure [4.3].
- `to` : Transfers the gate to another computing device (e.g., from CPU to GPU) while retaining the parameters and Kraus operators.
- `tensor` : Generates matrices for the Kraus operators, adapted to the full size of the quantum system. It allows the extension of operators to the system scale using Kronecker products.

After presenting the `Noise` class, here is an example of a subclass that pre-defines the Bit Flip channel. This subclass illustrates how to define a specific type of noise using the Kraus operators described in Appendix A.

```
class BitFlip(Noise):
"""
Initialize the BitFlip gate.

The bit flip channel is defined as:

.. math::
    \\rho \\Rightarrow (1-p) \\rho + p X \\rho X^{\\dagger}

Args:
    target (int): The index of the qubit being affected by the noise.
    error_probability (float): The probability of a bit flip error.
```

```python

    Raises:
        TypeError: If the error_probability value is not a float.
    """

    def __init__(
        self,
        target: int,
        error_probability: float,
    ):
        if error_probability > 1.0 or error_probability < 0.0:
            raise ValueError("The error_probability value is not a correct
               probability")
        K0: Tensor = sqrt(1.0 - error_probability) * IMAT
        K1: Tensor = sqrt(error_probability) * XMAT
        kraus_bitflip: list[Tensor] = [K0, K1]
        super().__init__(kraus_bitflip, target, error_probability)
```

The `__init__` method of the `BitFlip` class first checks that the error probability is within the valid range $[0,1]$. Then, it defines the corresponding Kraus operators based on this probability (see Appendix A). Finally, it calls the constructor of the parent class `Noise` to initialize the instance with the Kraus operators, the target qubit, and the error probability.

Annexe C

This appendix explains the operation of the Grover circuit illustrated in Figure [5.1], used to search for the state $|011\rangle$. The following steps explain the circuit's progression and show how the algorithm achieves this goal.

We initialize the input state $|\psi_0\rangle = |000\rangle$ by applying Hadamard gates (H) to each of the 3 qubits :

$$|\psi_1\rangle = \frac{1}{\sqrt{8}}\left(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle\right)$$

By applying the oracle, we mark with a negative phase the state $|011\rangle$ we want to find :

1. Apply a NOT gate (X) on the first qubit to the state $|\psi_1\rangle$:

$$|\psi_{2a}\rangle = \frac{1}{\sqrt{8}}\left(|100\rangle + |101\rangle + |110\rangle + |111\rangle + |000\rangle + |001\rangle + |010\rangle + |011\rangle\right)$$

2. Apply a double controlled-Z gate (CZ) between the first and second qubits (control qubits) and the third qubit (target) :

$$|\psi_{2b}\rangle = \frac{1}{\sqrt{8}}\left(|100\rangle + |101\rangle + |110\rangle - |111\rangle + |000\rangle + |001\rangle + |010\rangle + |011\rangle\right)$$

3. Apply a NOT gate (X) again on the first qubit :

$$|\psi_{2c}\rangle = \frac{1}{\sqrt{8}}\left(|000\rangle + |001\rangle + |010\rangle - |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle\right)$$

We apply the amplitude amplification by performing two iterations of reflection relative to the mean of the amplitudes :

1. Apply a Hadamard gate to each qubit in the system :

$$|\psi_{3a}\rangle = \frac{1}{4}\left(3|000\rangle + |001\rangle + |010\rangle - |011\rangle - |100\rangle + |101\rangle + |110\rangle - |111\rangle\right)$$

2. Apply an X gate to each qubit in the system :

$$|\psi_{3b}\rangle = \frac{1}{4}\left(-|000\rangle + |001\rangle + |010\rangle - |011\rangle - |100\rangle + |101\rangle + |110\rangle + 3|111\rangle\right).$$

3. Apply a double controlled-Z gate (CZ) between the first and second qubits (control qubits) and the third qubit (target) :

$$|\psi_{3c}\rangle = \frac{1}{4}\left(-|000\rangle + |001\rangle + |010\rangle - |011\rangle - |100\rangle + |101\rangle + |110\rangle - 3|111\rangle\right).$$

4. Apply a NOT gate (X) again on the first qubit :

$$|\psi_{3d}\rangle = \frac{1}{4}\left(-3|000\rangle + |001\rangle + |010\rangle - |011\rangle - |100\rangle + |101\rangle + |110\rangle - |111\rangle\right).$$

5. Apply another Hadamard gate (H) on the first qubit :

$$|\psi_{3e}\rangle = \frac{1}{4\sqrt{2}}\left(-|000\rangle - |001\rangle - |010\rangle - 5|011\rangle - |100\rangle - |101\rangle - |110\rangle - |111\rangle\right).$$

From the output state $|\psi_{3e}\rangle$, the measurement of the desired state $|011\rangle$ is five times more likely than that of the other states.