CIRCUIT SIMULATION USING QUTIP

Thesis

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

PHYSICS

by

BHARTI CHANDNAVAT MEENA

(2160010)

Under the guidance of

Dr. Deepak Vaid

PHYSICS DEPARTMENT

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA

SURATHKAL, MANGALORE -575025

July 2023

Bharti.

DECLARATION

I, hereby declare that the report of P.G Project Work entitled "Circuit Simulation Using Qutip" which is being submitted to the National Institute of Technology Karnataka, Surathkal, in partial fulfillment of the requirements for the award of the Degree of Master of Science in the Department of Physics, is a bonafide report of the work carried out by me. The material contained in this report has not been submitted to any University or Institution for the award of any degree.

Bharti.

NITK SURATHKAL
July 2023

BHARTI CHANDNAVAT MEENA
216PH010

CERTIFICATE

This is to certify that the P.G. Project Work Report entitled **"Circuit Simulation Using Qutip"** submitted by **Bharti Chandnavat Meena, (Roll Number: 216PH010)** as the record of the work carried out by him, is accepted as the **P.G. Project Work Report Submission** in partial fulfillment of the requirements for the award of the degree of **Master of Science** in the **Department of Physics**.

External Guide	**Internal Guide**
Dr. A. V. Narasimhadhan	**Dr Deepak Vaid**
(Name and signature with date and seal)	(Name and signature with date and seal)

Chairperson- Dr. Ajith K. M

(Signature with date and seal)

ACKNOWLEDGEMENT

I would like to express my sincere thanks to my project advisor Dr. Deepak Vaid, Department of Physics, NITK, Surathkal for his guidance, and instruction for the valuable project opportunity that he provided for me. I would like to thank our project coordinator Dr. Deepak Vaid, Department of Physics, NITK Surathkal for his kind support. Especially, I would like to thank my parents for their support and for having faith in me. I used Grammarly to make my English writing mistake-free.

Bharti

Place: NITK SURATHKAL
Date: July 2023

BHARTI CHANDNAVAT MEENA
216PH010

Abstract

Circuit **S**imulations **U**sing **Q**uTip (CSUQ) is widely used in quantum information processing. They were first introduced for the study of the impact of noise on quantum circuits, particularly in the context of quantum information processing. Starting with some basics of Quantum Computation, a quantum circuit has been constructed in QuTip and also find their unitaries and states with their probabilities in QuTip with the help of circuit simulation, and introduced noisy pulses in a quantum circuit. Some of the basic quantum algorithms have been run in QISKIT. "Quantum Toolbox in Python" (QuTip) is exactly the Python package that helps to understand quantum systems and also IBM provides software called Quantum Information software kit (QISKIT) that provides an interface between the quantum computer and users with the help of which one can compute any program source code on a real quantum computer. We simulate noisy quantum circuits at the pulse level using QuTip. Through the examination of individual pulses and the use of QuTiP's solvers for quantum dynamics and control optimization, our recently released tools in the qutip-qip package enable the simulation of quantum circuits. On simulated processors, we build quantum circuits by applying control pulses to a target Hamiltonian that depicts the development of physical qubits. Based on the physical model, we include various types of noise, such as density-matrix dynamics or Monte Carlo quantum trajectories. Users can simulate noise in control pulses and define environment-induced decoherence at the processor level. To highlight our strategy, we explain how the Deutsch-Jozsa algorithm is built and run using control optimization methods on spin-chain and superconducting-qubit-based processors.

Bharti.

Contents

Chapter 1

Introduction

This report is organized into several sections. This chapter 1 discusses a general Introduction of its subsections and briefs the basics of Quantum computation. we introduce the mathematical preliminaries for working with QISKIT such as section 1.1 Qubits which are the building blocks of Quantum computer, section 1.2 how to define multi particle quantum state, section 1.3 Quantum gates which are used to make quantum circuits in a quantum computer and then we defined entanglement section 1.4 and the formation of bell states. Using all these building blocks, at last, we introduced quantum teleportation circuit section 1.5. In further sections, we discussed some of the Quantum algorithms. In section 1.6 we discussed Deutsch Josza Algorithm. In section 1.7 we introduced the quantum circuit in section 1.7 and find its unitaries in subsection 1.8.2 that run the circuit in subsection 2.1.1 with the help of QuTip. Introduction of pulse in chapter 3 and noise in section 3.1 in a quantum system. The processor in the **qutip-qip** module simulates quantum circuits at the level of time evolution based on the open system solver. You may think of the processor, on which the quantum circuit will be constructed, as an emulator of a quantum device. The QuTiP open-time evolution solvers are used to simulate the evolution after the circuit is first assembled into a Hamiltonian model and noisy dynamics are included. We use a 3-qubit Deutsch-Jozsa algorithm simulation on a chain of spin qubits as an example to show how our system works. We'll go through this example and provide a brief overview of the procedure and all the major elements. We start by defining a Deutsch-Jozsa circuit and then start the simulation at the gate level. The underlying physical system, which is a subclass of processors, is then selected as the spin chain model. We supply the σ_x driving strength of 0.25MHz and the qubit count. The default setting is used for the other parameters, including the interaction strength. By defining the coherence times (T_1andT_2), which we explain below, the decoherence noise may also be added. By initializing it with the hardware parameters, this processor creates a Hamiltonian model for a spin chain system, including the drift and control Hamiltonians. The model class represents the Hamiltonian model, which is kept as an attribute of the initialized processor. The processor can also save simulation configurations, such as whether to interpolate the pulse coefficients using a cubic spline. Although not directly included in the model, these configurations might be crucial for the pulse-level simulation. Now, we use the **LinearSpinChain.load-circuit** method to send the circuit to the CPU. The processor will first break down the circuit's gates into native gates that may be used immediately on the designated hardware model. Then, using the **gatecompiler** designed for a particular model, each gate in the circuit is mapped to the control coefficients and driving Hamiltonians.

The potential for running multiple pulses simultaneously is investigated using a scheduler. To run the simulation of the processor, we start by generating and saving a detailed description of the circuit's behavior at the pulse level. This description allows us to simulate how the circuit evolves over time. In order to simulate this temporal evolution, we use the run-state method. First, we create a Lindblad model, which takes into account any defined noise models (although in this case, there are none). The Lindblad model helps us account for any possible external influences on the circuit. Now, we can specify various solver parameters to customize the simulation. For example, we can define a time sequence called **tlist** that determines at which intervals we want to capture the intermediate results of the simulation. We can also specify measurement observables using **"e-ops"**, which are quantities we are interested in measuring during the simulation. Additionally, we can provide options to further customize the solver's behavior. By running the simulation with these parameters, we obtain a result object. This object contains information about the final state of the circuit, and intermediate states at different time points, and we defined the expected values of the observables. This allows us to analyze and understand the behavior of the processor throughout the simulation.

Overall, the simulation process involves generating a pulse-level description of the circuit, constructing a Lindblad model, specifying solver parameters, and running the simulation to obtain a result object that provides valuable insights into the circuit's temporal evolution. This makes it possible to get all of the data that QuTiP's solvers offer.

Appendix A includes code for Quantum teleportation circuit in QISKIT.
Appendix B includes Python code for Deutsch Josza Algorithm in QISKIT.
Appendix C includes code for the plotting Quantum circuit in QuTip.
Appendix D includes code for Operator-label circuit simulation in QuTip.
Appendix E includes code for models in QuTip.
Appendix F includes code for noisy pulse simulation in QuTip.

1.1 Qubits

A classical computer operates on strings of bits. A quantum computer, on the other hand, works with qubits. We treat qubits as abstract mathematical objects. As a classical bit has a state – either 0 or 1 – a qubit also has two possible states $|0\rangle$ or $|\uparrow\rangle$ and $|1\rangle$ or $|\downarrow\rangle$

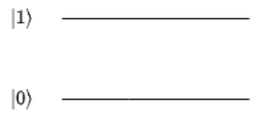


Figure 1.1: Quantum Two level system: Qubit

So a qubit is a two-level system spanned by two states $|0\rangle, |1\rangle$ or $|\uparrow\rangle, |\downarrow\rangle$. Or we can say that the states are spanned by n-bit strings $|x_1x_2x_3...x_{ni}\rangle$ with $x_i \epsilon$ 0, 1. It is also possible to form linear combinations of states, often called superpositions:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

where α and β are complex numbers. Hence we can say that the state of a qubit is a vector in a two-dimensional complex vector space and the states $|0\rangle$ and $|1\rangle$ are known as computational basis states and form an orthonormal basis for this vector space. A quantum computer state can be in any superposition of these basis states.[26]

1.2 Multiple particle Quantum States

We use the tensor product to describe Multi-particle states:

$$|a\rangle \otimes |b\rangle = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} \otimes \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} a_1 b_1 \\ a_2 b_2 \\ a_2 b_1 \\ a_2 b_2 \end{pmatrix}$$

Example: system A is in state $|1\rangle_A$ and system B is in state $|0\rangle_B$

So the total biparticle state is —$10\rangle_{AB} = |1\rangle_A \otimes |0\rangle_B = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} \otimes \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} a_1 b_1 \\ a_2 b_2 \\ a_2 b_1 \\ a_2 b_2 \end{pmatrix}$

Remark:
States of this form are called uncorrelated, but there are also bi-particle states that cannot be written as $|\psi\rangle_A \otimes |\psi\rangle_B$ These states are correlated and sometimes even entangled, For example:

$$\begin{aligned} |\psi\rangle_{AB} &= \frac{1}{\sqrt{2}} \left(|00\rangle_{AB} + |11\rangle_{AB} \right) \\ &\neq |\psi_A\rangle \otimes |\psi\rangle_B \\ &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \end{aligned}$$

1.3 Quantum Gates

A quantum computer is formed from a *quantum circuit* comprising wires and elementary quantum gates to carry around and manipulate quantum information, similar to how a conventional computer is built from an electrical circuit containing wires and logic gates. As quantum theory is unitary, quantum gates are represented by unitary matrices.[22]

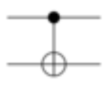
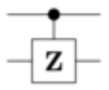
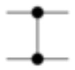
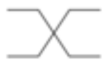
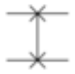
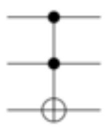
Operator	Gate(s)		Matrix
Pauli-X (X)	$-\boxed{\mathbf{X}}-$	$-\oplus-$	$\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$
Pauli-Y (Y)	$-\boxed{\mathbf{Y}}-$		$\begin{bmatrix}0 & -i\\ i & 0\end{bmatrix}$
Pauli-Z (Z)	$-\boxed{\mathbf{Z}}-$		$\begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}$
Hadamard (H)	$-\boxed{\mathbf{H}}-$		$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$
Phase (S, P)	$-\boxed{\mathbf{S}}-$		$\begin{bmatrix}1 & 0\\ 0 & i\end{bmatrix}$
$\pi/8$ (T)	$-\boxed{\mathbf{T}}-$		$\begin{bmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{bmatrix}$
Controlled Not (CNOT, CX)			$\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{bmatrix}$
Controlled Z (CZ)			$\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & -1\end{bmatrix}$
SWAP			$\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$
Toffoli (CCNOT, CCX, TOFF)			$\begin{bmatrix}1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&1\\ 0&0&0&0&0&0&1&0\end{bmatrix}$

Figure 1.2: Quantum Logic Gates

Real Dirac notation,

$$U = \begin{pmatrix} U_{00} & U_{01}\\ U_{10} & U_{11}\end{pmatrix}$$

$U = U_{00}|0\rangle\langle 0| + U_{01}|0\rangle\langle 1| + U_{10}|1\rangle\langle 0| + U_{11}|1\rangle\langle 1|$

1.3.1 Single Qubit gates

1. Pauli-X Gate

   Also known as classical NOT gate

$$\begin{aligned}\sigma_x &= \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix} = |0\rangle\langle 1| + |1\rangle\langle 0|\\ \sigma_x|0\rangle &= \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}\begin{pmatrix}1\\ 0\end{pmatrix} = \begin{pmatrix}0\\ 1\end{pmatrix} = |1\rangle \\ \sigma_x|0\rangle &= |1\rangle\\ \sigma_x|1\rangle &= |0\rangle\end{aligned} \tag{1.1}$$

   Pauli-X gate flips the bit and thus is called a bit flip gate. It represents rotation around the x-axis by angle π
   since $|+\rangle$ and $|-\rangle$ states lie on the x-axis hence there will be no effect of bit flip on them.

2. Pauli-Z gate

$$\begin{aligned}\sigma_x &= \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix} = |0\rangle\langle 0| - |1\rangle\langle 1|\\ \sigma_x|+\rangle &= \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}\frac{1}{\sqrt{2}}\begin{pmatrix}1\\ 1\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\ -1\end{pmatrix} = |1\rangle \\ \sigma_z|+\rangle &= |-\rangle\\ \sigma_z|-\rangle &= |+\rangle\end{aligned} \tag{1.2}$$

   Pauli-Z gate flips the state $|+\rangle$ to $|-\rangle$ and vice-versa and thus is called a phase flip gate. It represents rotation around the z-axis by angle π

3. Pauli-Y gate

$$\sigma_y = \begin{pmatrix}0 & -i\\ i & 0\end{pmatrix} = i\sigma_x\sigma_z \tag{1.3}$$

Pauli-Y gate is the combination of bit flip and phase flip. It represents rotation around the y-axis by angle π

$$\sigma_i^2 = I = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{1.4}$$

σ_x,σ_y,σ_z are Pauli Matrices
Together with identity, they form the basis of 2×2 matrices

4. Hadamard Gate

$$\begin{aligned} H &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \\ &= |0\rangle\langle 0| + |0\rangle\langle 1| + |1\rangle\langle 0| - |1\rangle\langle 1| \\ H|0\rangle &= |+\rangle \\ H|1\rangle &= |-\rangle \end{aligned} \tag{1.5}$$

This gate is used to create superposition states. It is also used to change between x and z basis

5. S Gate

$$\begin{aligned} S &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \\ &= |0\rangle\langle 0| + |0\rangle\langle 1| + |1\rangle\langle 0| - |1\rangle\langle 1| \\ S|+\rangle &= |+i\rangle \\ S|-\rangle &= |-i\rangle \end{aligned} \tag{1.6}$$

S gate adds 90 to the phase ϕ. S.H. is applied to change from a z to y basis.

1.3.2 Multi qubit gates

1. CNOT Gate
Acts on a pair of qubits, with one acting as 'control' and the other as 'target'. Performs a NOT on the target whenever the control is $|1\rangle$. If the control qubits are in a superposition state, this gate creates entanglement.

2. Toffoli Gate
   $\rightarrow$ Double Controlled-Not
   It has two control qubits and one target.
   Applies NOT to target only when both controls are in state $|1\rangle$
   Toffoli gate with Hadamard Gate is a universal gate set for quantum.

3. SWAP Gate
   Swaps the state of two qubits.

4. Identity Gate
   Is actually the absence of a gate. It ensures nothing is applied to a qubit for one unit of gate time.

Refer Figure 1.2 for more Quantum gates and their corresponding matrix notation.

1.4 Entanglement

If a pure state $|\psi\rangle_{AB}$ on systems A, B, cannot be written as $|\psi\rangle_A \otimes |\phi_B\rangle$ then it is entangled.

1.4.1 Bell States

There are four Bell states that are maximally entangled and build an orthogonal basis.

$$\begin{aligned}
|\psi^{00}\rangle &= \frac{1}{\sqrt{2}}\left(|00\rangle + |11\rangle\right) \\
|\psi^{01}\rangle &= \frac{1}{\sqrt{2}}\left(|01\rangle + |10\rangle\right) \\
|\psi^{10}\rangle &= \frac{1}{\sqrt{2}}\left(|00\rangle - |11\rangle\right) \\
|\psi^{11}\rangle &= \frac{1}{\sqrt{2}}\left(|01\rangle - |10\rangle\right)
\end{aligned} \tag{1.7}$$

In general, we can write

$$|\psi^{ij}\rangle = \left(I \otimes \sigma_x^j \sigma_z^i\right) |\psi^{00}\rangle$$

Creation of Bell States

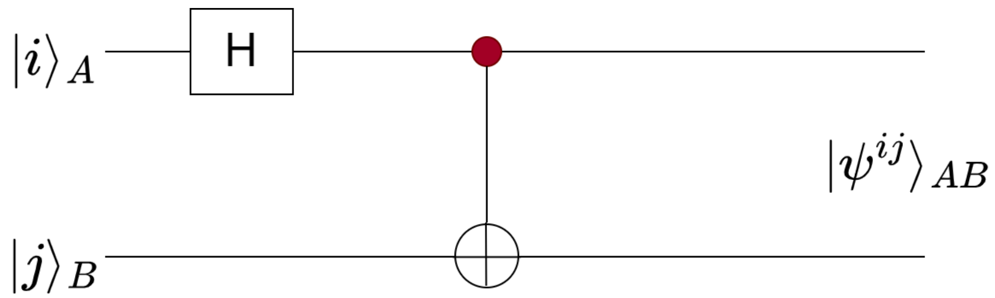


Figure 1.3: Creation of bell states

Initial state	H_A	$CNOT_{AB}$
$\vert\psi^{00}\rangle$	$\left(\frac{1}{\sqrt{2}}\vert 00\rangle + \vert 10\rangle\right)$	$\left(\frac{1}{\sqrt{2}}\vert 00\rangle + \vert 11\rangle\right)$
$\vert\psi^{00}\rangle$	$\left(\frac{1}{\sqrt{2}}\vert 01\rangle + \vert 11\rangle\right)$	$\left(\frac{1}{\sqrt{2}}\vert 01\rangle + \vert 10\rangle\right)$
$\vert\psi^{00}\rangle$	$\left(\frac{1}{\sqrt{2}}\vert 00\rangle - \vert 10\rangle\right)$	$\left(\frac{1}{\sqrt{2}}\vert 00\rangle - \vert 11\rangle\right)$
$\vert\psi^{00}\rangle$	$\left(\frac{1}{\sqrt{2}}\vert 01\rangle - \vert 11\rangle\right)$	$\left(\frac{1}{\sqrt{2}}\vert 01\rangle - \vert 10\rangle\right)$

Bell States creation code:

This is the code of creation of the bell state. Which is help to understand the construction of the bell state in qiskit.

```
import numpy as np
from qiskit import *
from qiskit.quantum_info import Statevector
from qiskit.visualization import plot_state_qsphere
qreg = QuantumRegister(2, 'q')
qc=QuantumCircuit(qreg)
qc.reset(qreg[0])
qc.reset(qreg[1])
qc.h(0)
qc.cx(0,1)
qc.x(0)
qc.z(0)
sv=Statevector.from_instruction(qc).data
print(sv)
plot_state_qsphere(sv)
```

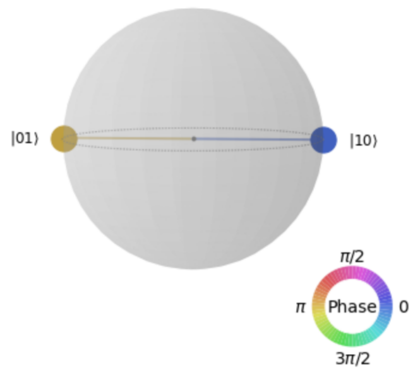


Figure 1.4: Formation of Bell states on Q-sphere

1.5 Teleportation

Quantum teleportation is a technique for moving quantum states around, even in the absence of a quantum communications channel linking the sender of the quantum state to the recipient.

PROBLEM:

Alice and Bob met long ago and they generated an EPR pair, each taking one qubit of the EPR pair when they separated. Alice wants to send her unknown state to Bob. She can only send him classical information.[29][22]

They both share the maximally entangled state.[27]

$$|\psi^{00}\rangle_{AB} = \frac{1}{\sqrt{2}}\left[|00\rangle_{AB} + |11\rangle_{AB}\right]$$

SOLUTION: The initial state of the total system

$$|\phi\rangle_s \otimes |\psi^{00}\rangle_{AB} = \frac{1}{\sqrt{2}}\left(\alpha|011\rangle_{SAB} + \beta|100\rangle_{SAB} + \beta|111\rangle_{SAB}\right)$$

$$
\begin{aligned}
&= \frac{1}{2\sqrt{2}}[(|00\rangle_{SA} + |11\rangle_{SA}) \otimes (\alpha|0\rangle_B + \beta|1\rangle_B) \\
&\quad +(|01\rangle_{SA} + |110\rangle_{SA}) \otimes (\alpha|1\rangle_B + \beta|0\rangle_B) \\
&\quad +(|00\rangle_{SA} - |11\rangle_{SA}) \otimes (\alpha|0\rangle_B - \beta|1\rangle_B) \\
&\quad +(|01\rangle_{SA} - |10\rangle_{SA}) \otimes (\alpha|1\rangle_B - \beta|0\rangle_B)] \\
&= \frac{1}{\sqrt{2}}[(|\psi^{00}\rangle_{SA} \otimes |\phi\rangle_B \\
&\quad +(|\psi^{01}\rangle_{SA} \otimes \sigma_x|\phi\rangle_B \\
&\quad +(|\psi^{10}\rangle_{SA} \otimes \sigma_z|\phi\rangle_B \\
&\quad +(|\psi^{11}\rangle_{SA} \otimes \sigma_x\sigma_z|\phi\rangle_B]
\end{aligned}
$$

PROTOCOL

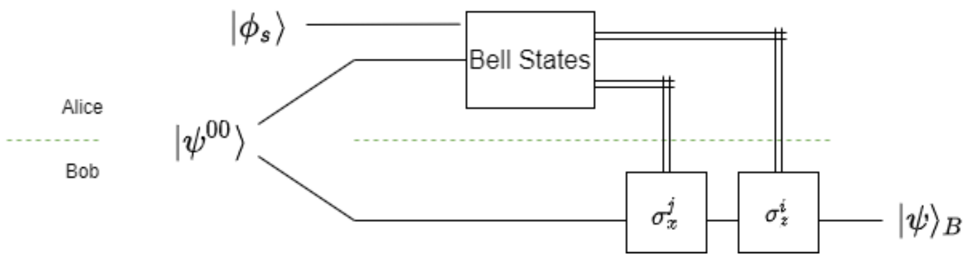


Figure 1.5: Teleportation protocol

1. Alice measures S and A on a bell basis.

Alice measures state	$\rightarrow$	corresponding Bob's state
$|\psi^{00}\rangle$		$|\phi\rangle_B$
$|\psi^{01}\rangle$		$\sigma_x|\phi\rangle_B$
$|\psi^{10}\rangle$		$\sigma_z|\phi\rangle_B$
$|\psi^{11}\rangle$		$\sigma_x\sigma_z|\phi\rangle_B$

2. She sends her classical output i, j to Bob.
3. Bob applies $\sigma_z^i\sigma_x^j$ to his qubit and gets $|phi\rangle$.
4. Bob applies $\implies$ Bob's final state.

		i	j	Bob's applies	Bob's final state
$\|\psi^{00}\rangle$	$\|\phi\rangle_B$	0	0	I	$\|\phi\rangle_B$
$\|\psi^{01}\rangle$	$\sigma_x\|\phi\rangle_B$	0	1	σ_x	$\|\phi\rangle_B$
$\|\psi^{10}\rangle$	$\sigma_z\|\phi\rangle_B$	1	0	σ_z	$\|\phi\rangle_B$
$\|\psi^{11}\rangle$	$\sigma_x\sigma_z\|\phi\rangle_B$	1	1	$\sigma_z\sigma_x$	$\|\phi\rangle_B$

NOTE:
Alice's state collapses during the measurement, so she does not have the initial state $|\phi\rangle$ anymore.
This is expected due to the no-cloning theorem, as she cannot copy her state, but just sends her state to Bob when destroying her own.
States gets teleported .

Refer Appendix A to see how to Implement a Teleportation circuit in QISKIT.

Quantum Energy Teleportation: The Quantum Energy Teleportation (QET) quantum algorithm utilized in this study is publicly available and offers quantum circuit implementations of QET as well as real-time data on the most recent machine attributes. The results and QET methods can be replicated by anybody using the quantum circuits given in this study. Anyone can validate the QET protocol with real-time access to all the quantum computer's characteristics, regardless of whether they own a quantum device. Any system capable of QET can use the techniques presented in this study. It is crucial to make clear that in QET, only conventional information, not energy is sent. The communication subsystem between the sender (Alice) and receiver (Bob) during the QET procedure is not activated by energy carriers in the system. Compared to the time it takes for natural heat to be generated, the time scale of energy conveyance by QET is substantially shorter. By carrying out actions on his local system based on the conventional information Alice communicated, Bob can draw energy from the system. Bob gets energy much more quickly (at the speed of light) than it can be normally conveyed from Alice to Bob is referred to as teleportation or energy transfer. A many-body quantum system serves as the intermediary in the description of QET as a universal method of extracting quantum energy. Measurements on a quantum many-body system's ground state are one example of a non-trivial local operation that produces excited states and raises the energy expectation value. The experimental equipment is what increases energy.

Entanglement is a property of the ground states of quantum many-body systems that results in local quantum fluctuations in the overall ground state. The ground state entanglement is destroyed and energy E_A is introduced into the entire system when the quantum state at subsystem A is locally measured. Although the introduced energy initially remains in the area of subsystem A, actions on A alone are unable to recover it since data about E_A is also stored in distant regions due to pre-existing entanglement. Energy extraction is made possible by QET by combining Local Operations and Classical Communication with conditional operations. Due to mid-circuit measurement, the time development of the post-measurement state is non-unitary. QET is predicated on the universal fact of measurement and the generic character of the many-body system's ground state entanglement. It has been demonstrated that QET can replicate many-body phase diagrams and phase structures by teleporting energy to Bob's local system[11, 9]. This shows that simple Local Operations and Classical Communication can detect global features like symmetry, topology, and long-range correlation, which is advantageous for quantum computer research in physics. The simplest QET model proposed is the main subject of the work. The creation of a quantum circuit that works with actual quantum computers and networks is one of the goals of this article. depicts the whole configuration of the quantum circuits used for QET, which have also been expanded to long-range and large-scale quantum networks. The usage of 2 qubits and a maximum circuit depth of 6 shows that QET can be implemented on modern quantum computers[10, 12].
So Quantum Energy Teleportation (QET) is also possible like quantum state teleportation (QST).[13]

1.6 Deutsch Jozsa Algorithm

To decide some property of the oracle (constant or balanced) using the minimum number of queries.
On a classical computer, such an oracle is given by a function: $f : (0, 1)^n \to (0, 1)^n$
On a Quantum computer, the oracle must be reversible.

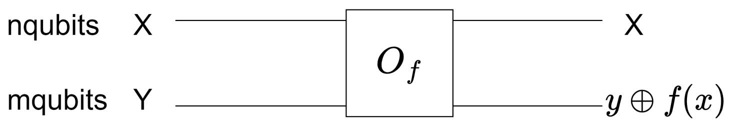


Figure 1.6: Reversible oracle

SOLUTION:

1. Let x and y are in state $|0\rangle$.

$$|x\rangle = |0\rangle, |y\rangle = |0\rangle$$
$$i.e.|\psi_{in}\rangle = |x\rangle|y\rangle = |0\rangle|0\rangle$$
$$|\psi_{out} = |x\rangle|y + f(x)\rangle = |0\rangle|0 \oplus f(0)\rangle = |0\rangle f(0)$$

2. Let the state of y is changed from $|0\rangle$ to $|1\rangle$.

$$|x\rangle = |0\rangle, |y\rangle = |1\rangle$$
$$i.e.|\psi_{in}\rangle = |x\rangle|y\rangle = |0\rangle|1\rangle$$
$$|\psi_{out} = |x\rangle|y + f(x)\rangle = |0\rangle|1 \oplus f(0)\rangle = \begin{cases} |0\rangle|1\rangle, & \text{when} f(0) = 0 \\ |0\rangle|0\rangle, & \text{when} f(0) = 1 \end{cases}$$

3. Now let y be in a superposition state.

$$|\psi_{in}\rangle = |x\rangle|y\rangle = |0\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right)$$
$$|\psi_{out}\rangle = |0\rangle \left(\frac{|0 \oplus f(0)\rangle - |1 \oplus f(0)\rangle}{\sqrt{2}}\right) = \begin{cases} |0\rangle\frac{|0\rangle - |1\rangle}{\sqrt{2}}, & \text{when} f(0) = 0 \\ -|0\rangle\frac{|0\rangle - |1\rangle}{\sqrt{2}}, & \text{when} f(0) = 2 \end{cases}$$

$$|\psi_{out}\rangle = (-1)^{f(0)}|0\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) \tag{1.8}$$

Equation 1.8 known as Deutsch Algorithm.
U_f also called Bit flip oracle can be seen as a unitary that performs the map,

$$U_f|x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle$$

for$f:(0,1)^n \rightarrow (0,1),$ we can construct U_f.

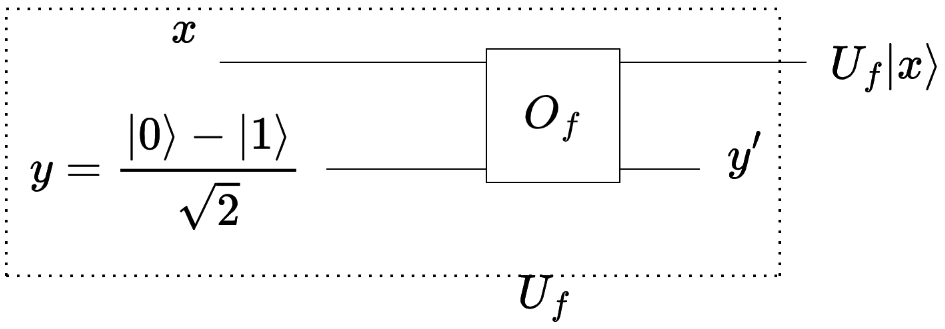


Figure 1.7: Deutsch Josza oracle

$$\begin{aligned}
|0\rangle\left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) &\rightarrow (-1)^{f(0)}|0\rangle\left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right)\\
|0\rangle\left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) &\rightarrow (-1)^{f(0)}|1\rangle\left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right)\\
U_f|x\rangle &= (-1)^{f(x)}|x\rangle \text{ Independent of } \ |y\rangle \qquad (1.9)
\end{aligned}$$

4. When both Qubits are in a superposition state:

$$\begin{aligned}
|\psi_{in}\rangle &= \left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right)\left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right)\\
&= \frac{1}{\sqrt{2}}\left(|0\rangle\left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) + |1\rangle\left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right)\right)
\end{aligned}$$

$$\begin{aligned}
|\psi_{out}\rangle &= \frac{1}{\sqrt{2}}\left((-1)^{f(0)}|0\rangle\left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) + (-1)^{f(1)}|1\rangle\left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right)\right)\\
&= \frac{1}{\sqrt{2}}\left((-1)^{f(0)} + (-1)^{f(1)}|1\rangle\right)\left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right)
\end{aligned}$$

$$\left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right)\left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) = \begin{cases} \left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right)\left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right); & \text{when} f(0) = f(1)\\ \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right)\left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right); & \text{when} f(0) \neq f(1)\end{cases}$$

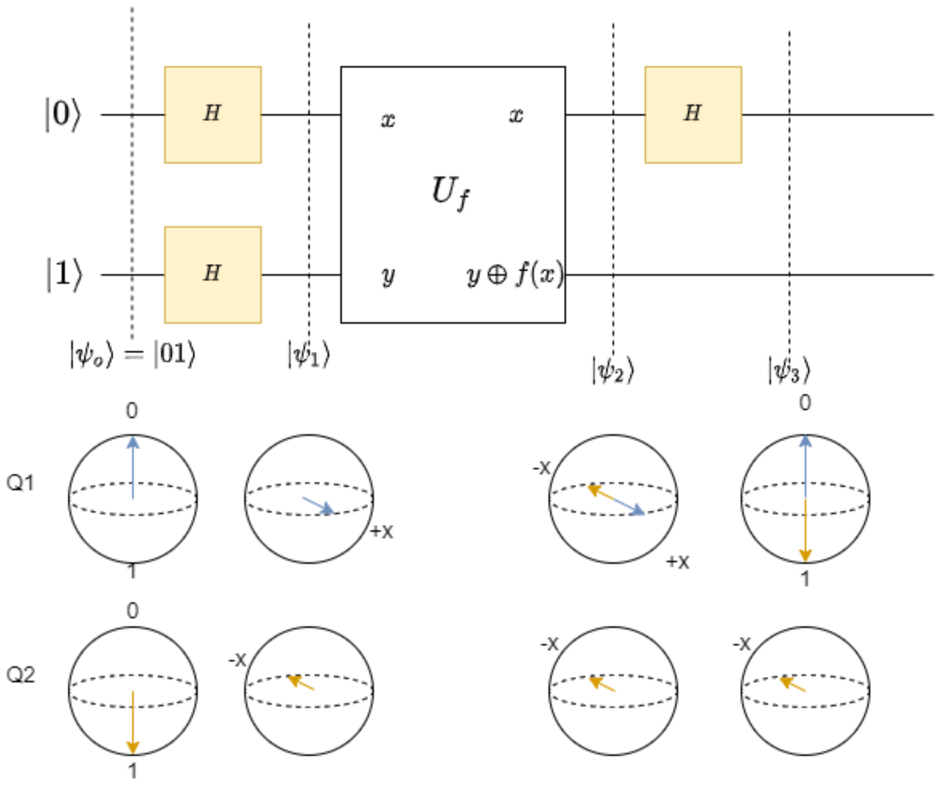


Figure 1.8: Deutsch Jozsa circuit for 2 qubits

From Figure 1.8 you can tell if f(0)=f(1) by checking the first bit at one go. Classical algorithm would require two checks.

N Qubit GENERALIZATION: DEUTSCH JOZSA ALGORITHM

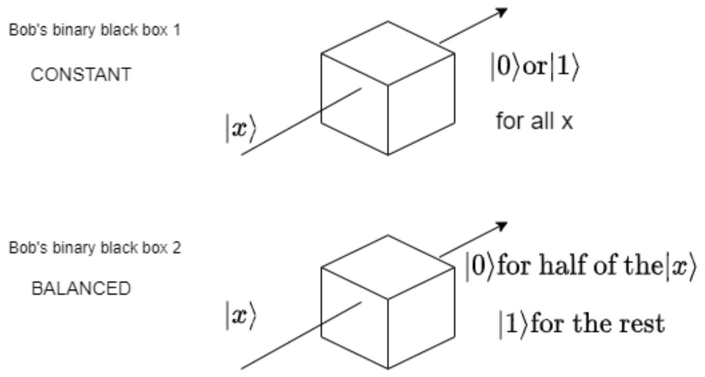


Figure 1.9: Deutsch Josza oracle

Alice can send bob a bunch of numbers from 0 to n and a bit to write the answer. Bob promises to use any one of the black boxes for all the numbers: Doesn't tell which one. Bob sends the bit back to Alice.
Alice needs to determine which one did he use? (with a minimum no. of trials)

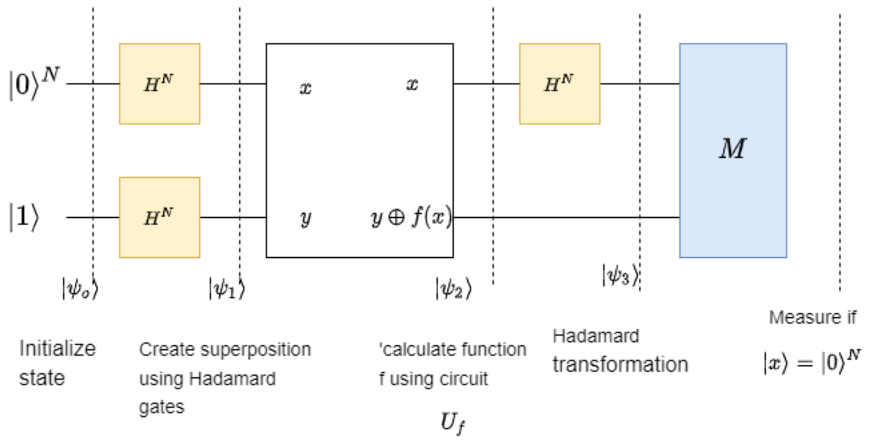


Figure 1.10: Deutsch Jozsa circuit

After measurement if $|x\rangle$ are 0 $\implies$ oracle is Constant;
If $|x\rangle$ are 1 $\implies$ oracle is Balanced.

Refer Appendix B to see the code of the Deutsch Jozsa Algorithm in Qiskit.

1.7 Quantum Circuit

A quantum circuit is the most common model for quantum computing, just like a classical circuit, in quantum information.
We can draw a quantum circuit consisting of qubits and quantum gates, just like we can draw a classical circuit consisting of bits and logic gates, in which a computation is a sequence of resisters, quantum gates, and measurements.

1.7.1 Registers

The register consists of multiple qubits (qubits and classical bits). It's the quantum analog of the classical processor register. The number of qubit registers in the circuit is specified by argument N, and the argument **num-cbit** specifies the number of classical bits available for measurement and control.
Qubit register: The qubit in quantum computing are conceptually grouped together in the qubit register. Each qubit has an index within this register, index counting starting at index 0 and counting up by 1. For example, we have a system with 5 qubits, resultant we get a qubit register that has a width of 5 and is indexed by 0, 1, 2, 3, 4.
Each qubit can be addressed in qubit operations by using the qubit index.

1.7.2 Gates

Each quantum gate is stored as a class object called **Gate** that contains details like the gate name, the target qubits, and the arguments. A classical bit can also have gates controlled by providing the register number with the argument

```
|classical-controls|
```

.

1.7.3 Measurements

Measurement is essential to express any quantity of object (qubits and cbits). Measurements on an individual qubit (the circuit's middle, and end) can also be carried out. Each measurement is saved as a class object called **Measurement** with parameters such as targets, the target qubit on which the measurement will be carried out, and **classical-store**, the index of the classical resister which stores the measurement result.[30]

In QuTip (Quantum Tool in Python), we used **QubitCircuit** to represent a quantum circuit.

DEMONSTRATION

```
from qutip_qip.circuit import QubitCircuit
from qutip_qip.operations import Gate
from qutip import tensor, basis

qc = QubitCircuit(N=2, num_cbits=1)
swap_gate = Gate(name="SWAP", targets=[0, 1])
qc.add_gate(swap_gate)

# measurement gate
qc.add_measurement("M0", targets=[1], classical_store=0)
qc.add_gate("CNOT", controls=0, targets=1)

# classically controlled gate
qc.add_gate("X", targets=0, classical_controls=[0]) # classically controlled gate
qc.add_gate(swap_gate)

print(qc.gates)
```

circuit gets demonstrated.

1.8 Unitary Matrix

A square matrix of complex numbers is said to be unitary if B is a unitary matrix $B^\dagger$ is its complex transpose then the following equation is satisfied.

$$B^\dagger B = BB^\dagger = I \tag{1.10}$$

Statement: The product of the unitary matrix and the conjugate transpose of a unitary matrix is equal to the Identity Matrix.
The dimension of the unitary matrix is $n \times n$.

1.8.1 Properties

1. The unitary matrix is an invertible matrix.
2. The inverse of a unitary matrix is another unitary matrix.
3. A matrix is unitary, if and only if its transpose is unitary.
4. A matrix is unitary if its rows are orthonormal, and the columns are orthonormal.
5. The unitary matrices can also be non-square matrices but have orthonormal columns and rows.
6. The unitary matrix is a non-singular matrix.
7. Two matrices A and B are unitary then product AB and BA are also unitary.
8. The sum or difference of two unitary matrices is also a unitary matrix.

1.8.2 Unitary of Demonstrated Circuit

In QuTip, to find the unitaries of the circuit there are some useful functions, which are associated with the circuit object. A method to return a list of the unitaries **QubitCircuit.propagators()**, which is associated with the sequence of the gates in the circuit. Hence, the unitaries are expanded to the full dimension of the circuit. Also, another argument can be registered to achieve the unitaries in their original dimension.[19]

The argument is called **expand=False**, which is specified to the

```
|QubitCircuit.propagators()|
```

.

1.9 Gates

In QuTip, **QubitCircuit** has a primitive method called **QubitCircuit.resolve-gate()** to decompose the primitive gate into elementary gate sets, such as CNOT or SWAP gate with single qubit gates [RX, RY, and RZ(rotation around x,y, and z axis respectively). This approach isn't entirely optimal, though. The likelihood is high that combining quantum gates will further lower the depth of the circuit. Prior to adding the measurements to the circuit, the gate resolution must be completed. Therefore, this method is not fully advanced. It is possible that the depth of the circuit can be further reduced by merging the quantum gate. It is required that the gate resolution is carried out before the measurements to the circuits are added.
Custom Gates: QuTip is allowed to define its own gate (customized gate).
: To define a customized gate, **qutip.Qobj** is a key step to returning a gate function and is saved in the attribute **user-gates**.

1.9.1 Plotting a Quantum Circuit

As we described above, a demonstrated circuit can be plotted directly using a **Q circuit** (quantum circuit drawing application and implemented directly into QuTiP) library.
ISWAP Gate is a **Customized gate**.
It is a SWAP gate with an additional phase states $|01\rangle$ and $|10\rangle$.

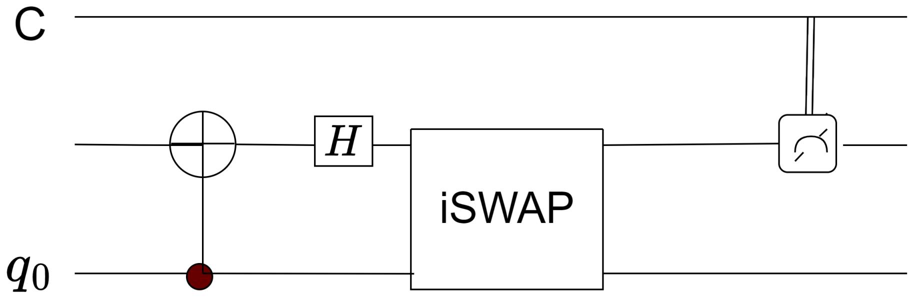


Figure 1.11: Quantum circuit

Refer Appendix C to see how to plot a quantum circuit in QuTip.

Chapter 2

oprator-level circuit simulation

This is the first method of circuit simulation. A unitary application on the input states through matrix product is used by this method. This method simulates circuits exactly in a deterministic manner. This is achieved through the

`|CircuitSimulator|`

.

2.1 W-STATE

W-states are a class of multi-qubit states. In a W-state, precisely one of the qubits is on, and the rest are off. A W-state for an n-qubit system is described by[5]

$$|W_n\rangle = \sum \frac{1}{\sqrt{n}} |(2^i)_n\rangle \tag{2.1}$$

$$= \frac{1}{\sqrt{n}}|00\ldots01\rangle + \frac{1}{\sqrt{n}}|0\ldots010\rangle + \cdots + \frac{1}{\sqrt{n}}|010\ldots0\rangle + \frac{1}{\sqrt{n}}|10\ldots00\rangle \tag{2.2}$$

Bell state is a W-state on two qubits.
W state for Three qubits ($n = 3$):

$$|W_3\rangle = \frac{1}{\sqrt{n}}[|0\rangle_1 \otimes (|01\rangle_{23} + |10\rangle_{23}) + |1\rangle_1 \otimes |00\rangle_{23}] \tag{2.3}$$

$$|W_3\rangle = \frac{1}{\sqrt{n}}[|001\rangle + |010\rangle + |100\rangle] \tag{2.4}$$

2.1.1 Run a Quantum Circuit:

Let's start with a simple circuit which we use to demonstrate a few examples of circuit simulation. we take a circuit from **OpenQASM** because QuTip supports the importation and exportation of the quantum circuit in the OpenQASM2 W-state format through the function **read-qasm()** and **save-qasm()**. We demonstrated this by using the w-state generation circuit.
OpenQASM 2 : OpenQASM (Open Quantum Assembly Language) is an intermediate representation for spaces, examples, and tools.
W-state is one of the useful open qasm examples; **W-state.qasm**: Generating a 3-qubit W-state using Toffoli gate.
The example can not necessarily be executed on any existing hardware. We can still run a circuit in a circuit simulator.[16]
This circuit prepares the W-stat :

$$|W_3\rangle = \frac{1}{\sqrt{n}}[|001\rangle + |010\rangle + |100\rangle] \tag{2.5}$$

This way the simplest way to carry out state evolution through a quantum circuit is to be provided an input state to the method called.

```
|QubitCircuit.run()|
```

Now run the circuit without applied measurement gates to get W-state.
As we see without measurement gates, the state returned is indeed the required W-state. Therefore, getting all the possible outcomes with associated probabilities can be carried out by circuit evolution in a deterministic manner.
Such that, it is returned all the possible state outputs along with their probabilities. we are adding measurement gates into the circuit. After applying measurement gates to get all the possible output states along with their associate probabilities of observing the outputs, can be used the function

```
|QubitCircuit.run-statistics()|
```

.

```
result = qc.run_statistics(state=tensor(basis(2, 0), basis(2, 0), basis(2, 0)))
states = result.get_final_states()
probabilities = result.get_probabilities()
```

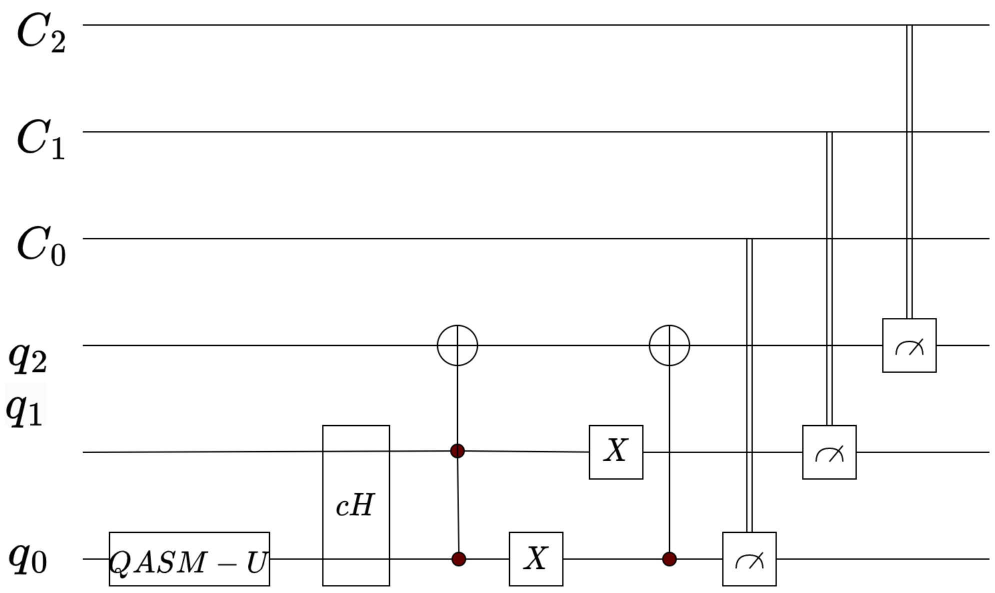


Figure 2.1: w-state circuit

In the above cell, the function is returned by an object called **result**, which contains the output states. And then another argument **get-results()** can be used to obtain the possible states with probabilities. Already the state created by the circuit; is the W-state. As we can see, we observe the W states $|001\rangle$, $|010\rangle$, and $|100\rangle$ with equal probability.

Result

we observed the states and their probabilities.

States	Probability
$\|001\rangle$	0.33333
$\|010\rangle$	0.33333
$\|100\rangle$	0.33333

We observed W-state with their equal probability.
Refer Appendix D to see the code in qutip.

2.2 Circuit Simulator

We have two different ways to simulate the action of quantum circuits using QuTiP:

- The first method uses unitary application through matrix products on the input states. This method simulates circuits precisely in a deterministic manner. This is achieved through the class **CircuitSimulator**. A short guide to exact simulation can be found at Gate-level circuit simulation.

- The second method of circuit simulation uses driving Hamiltonians with the ability to simulate circuits in the presence of noise. This is achieved by the various classes in **device**. A short guide to processors for QIP simulation can be found at Pulse-level circuit simulation.

The functions **QubitCircuit.run()** and **QubitCircuit.run-statistics()** make use of the **CircuitSimulator** which enable exact simulation with more granular options. And the simulator object takes a quantum circuit as an argument. It can optionally be supplied with an initial state. Here, We have two modes in which the exact simulator can function. The default mode is the **"state-vector-simulator"** mode. In this mode, the state evolution proceeds to maintain the ket-state during the computation. For each measurement gate, one of the possible outcomes is chosen in a probabilistic manner, and computation proceeds. We continue with our previous circuit for demonstration.
This initializes the simulator object and carries out any pre-computation requirement. And Here, we have two ways to carry out the state evolution with the simulator. The first way is to use the **CircuitSimulator.run()** and the second way **CircuitSimulator.run-statistics()** functions just like above (with the **CircuitSimulator** class).

The **CircuitSimulator** class is enabled to step through the quantum circuit. This only executes one gate in the circuit, and allow is for a better understanding of how the state evolution takes place. The method steps through both the gates and the measurements.

2.2.1 Precomputing the unitary

In QuTip, the **CircuitSimulator** class is initialized such that the circuit evolution is conducted by applying each unitary to the state interactively. However, by setting the argument **precompute-unitary=True**, And **CircuitSimulator** is precomputing the product of the unitaries between the measurements.
The **sim.ops** argument stores all the circuit operations which are applied during the state evolution. As we observed, all the unitaries of the circuit are compressed into a single unitary product with precompute optimization enabled. This is more efficient if it is run in the same circuit in multiple initial states. However, as the number of qubits increases, then this will consume more and more memory and it's become complicated.
Result:
Unitaries

$$\begin{bmatrix} 0 & 0.577 & 0 & -0.577 & 0 & 0.408 & 0 & -0.408 \\ 0.577 & 0 & -0.577 & 0 & 0.408 & 0 & -0.408 & 0 \\ 0.577 & 0 & 0.577 & 0 & 0.408 & 0 & 0.408 & 0 \\ 0 & 0.577 & 0 & 0.577 & 0 & 0.408 & 0 & 0.408 \\ 0.577 & 0 & 0 & 0 & -0.816 & 0 & 0 & 0 \\ 0 & 0.577 & 0 & 0 & 0 & -0.816 & 0 & 0 \\ 0 & 0 & 0.577 & 0 & 0 & 0 & -0.816 & 0 \\ 0 & 0 & 0 & 0.577 & 0 & 0 & 0 & -0.816 \end{bmatrix}$$

Table of measurement, target, and classical store

Measurement	Target	Classical-store
$M0$	[0]	0
$M1$	[1]	1
$M2$	[2]	2

Refer Appendix D to see the code in qutip.

2.3 Density Matrix

A density matrix is a matrix in quantum mechanics that describes the quantum state of a physical system. It is used to calculate the probabilities of outcomes of

measurements on the system using the Born rule. It is a generalization of state vectors and wavefunctions: while those can only represent pure states, then density matrices can also represent mixed states. which occur when a system's preparation is not fully known, deals with a statistical ensemble of possible preparations, or when describing an entangled system without describing the combined state.
We compose the density matrix as the outer product of a state with itself

$$\rho = |\psi\rangle\langle\psi| \tag{2.6}$$

it is a pure state density matrix.

For mixed state

$$\rho = \sum_k p_k |\psi_k\rangle\langle\psi_k| \tag{2.7}$$

The purity of a quantum state $(\psi\rangle)$ is defined as the 'Trace' of the squared density matrix(ρ^2).
Purity $= Tr[\rho^2]$
where Trace = sum of the diagonal elements in the matrix.

2.3.1 Open Quantum System

In many areas of modern physics, it is critical to simulate the dynamics of open quantum systems, where a quantum system interacts with its environment. The study of decoherence, optical response theory, and quantum rate theory are all built on this principle. The generalized quantum master equation (GQME) formalism offers a flexible and all-encompassing strategy to solve this issue. The GQME offers an effective method for modeling and comprehending the behavior of open quantum systems, allowing researchers to investigate and evaluate a variety of phenomena in these disciplines[17]. In the realm of quantum systems, perfect isolation is rare, which means that a purely unitary evolution is often an approximation. By concentrating on the system's Hilbert space and utilizing superoperators that produce non-unitary dynamics (i.e., for an open system), we can either reduce computing overhead or expand the Hilbert space to account for potential interactions with the environment. The Lindblad master equation is a common tool for describing the evolution of an open quantum system. Both differential equation solvers like **qutip.mesolve** and

qutip.mcsolve, which use Monte Carlo sampling of quantum trajectories, can be used to solve this equation. For pulse-level circuit simulators, both solvers act as the simulation back ends[15]. These solvers provide effective simulations of quantum dynamics in open systems. They can include more general Lindbladians that take into consideration time-dependent rates, as well as noise models based on Born-Markov Secular (BMS) approximations. These noise models are typically strong and adaptable enough to reflect the primary impacts of ambient noise, making them appropriate for describing different kinds of hardware implementations.[28][3]

2.3.2 Density Matrix simulation

In QuTip, by default, the state evolution is carried out in the **"state-vector-simulator"** mode (specified by the mode argument) as we described before. In the **"density-matrix-simulator"** mode, the input state can be either a ket or a density matrix. If it is a ket, so it is converted into a density matrix before the evolution is carried out. Unlike the **"state-vector-simulator"** mode, upon measurement, the state does not collapse to one of the post-measurement states. Rather, now the new state is the density matrix representing the ensemble of post-measurement states. In this situation, we measure the qubits only and forget about all the results.
To demonstrate the matrix, we consider the original W-state circuit which is followed by measurement only on the first qubit.
Result:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.33333257 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.33333257 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.33333486 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

It is a mixed state.
We got a mixed-state density matrix.
The **qutip.mesolve** function is a tool that can be used to solve a wide range of open quantum dynamics. The density matrix of the quantum system, $\rho(t)$, evolves over

time under the influence of a mathematical operator known as the superoperator L and can be expressed by an equation of the form

$$\frac{\partial \rho(t)}{\partial t} = L\rho(t) \tag{2.8}$$

When utilizing the **qutip.mesolve** function, we have the option of either providing the entire superoperator L or splitting the dynamics into the Hamiltonian and noise terms, which are represented as a set of collapse operators (c-ops) and associated rates. Then, behind the scenes, the function will successfully resolve Equation 2.8 for us.

Equation 2.8's structure is highly adaptable and may take into account a variety of situations. Going beyond the conventional Born-Markov and secular approximations typically used in quantum physics, it enables the incorporation of time-dependent rates and collapse operators. To put it simply, **qutip.mesolve** is a helpful tool that aids in the solution of difficult equations defining the behavior of open quantum systems. It offers freedom in determining the dynamics, enabling simulations that go beyond simple approximations and are more realistic and accurate.

One of the easiest ways to simulate the behavior of a quantum system is by using a Lindblad master equation. This type of equation is commonly used when studying quantum circuits with multiple qubits that undergo relaxation and dephasing. Let's consider an example of a quantum circuit with N qubits. The Lindblad master equation for this system describes how the density matrix of the qubits evolves over time. It takes into account the effects of relaxation (where the qubits lose their quantum information to the environment) and dephasing (where the qubits experience random phase fluctuations). By using the Lindblad master equation, we can simulate the dynamics of the quantum circuit and observe how the density matrix changes over time. This approach allows us to study the effects of relaxation and dephasing on the quantum states of the qubits and gain insights into their behavior in realistic scenarios.

Overall, the Lindblad master equation provides a straightforward and effective method to model and analyze quantum systems, such as quantum circuits, in the presence of relaxation and dephasing. It enables researchers to understand and predict the behavior of these systems more accurately, facilitating advancements in quantum computing and related fields. Lindblad master equation:

$$\frac{\partial \rho(t)}{\partial t} = -i[H(t), \rho(t)] + \sum_{j=0}^{N-1} \lambda_j D[\sigma_j^-]\rho(t) + \sum_{j=0}^{N-1} \frac{\lambda_j^D}{2} D[\sigma_j^z]\rho(t) \tag{2.9}$$

where H is Hamiltonian, λ_j is the relaxation rate of qubit j, λ_j^D the pure dephasing rate of qubit j,

$$D[\Gamma_n]X = \Gamma_n X \Gamma_n^\dagger - \frac{1}{2}\Gamma_n^\dagger \Gamma_n X - \frac{1}{2} X \Gamma_n^\dagger \Gamma_n \tag{2.10}$$

is the Lindblad dissipator. For a generic jump operator, Γ_n acting on a density matrix X, and σ_j^α are Pauli operators, with α = x, y, z.

We can replicate the combined effects of pulse-level control and noise in a quantum system using the previously outlined technique. This method takes into account both the coherent Hamiltonian component, which represents the system's regulated processes, and the impact of noise on the system's behavior. It's crucial to remember that switching to the system's density matrix description results in a quadratic increase in memory size. As a result, the amount of memory needed for the simulation expands quickly as the system's dimensions and qubit count do as well. This could provide a problem for some simulations, particularly when working with bigger quantum systems.

The Monte-Carlo quantum trajectory solver is a different method that can be utilized to get around this restriction. The **"qutip.mcsolve"** function is one example of this solver's implementation. Instead of directly computing the density matrix, this solver simulates the evolution of the quantum system using a probabilistic technique. This considerably lowers the memory overhead needed for the simulation. In simpler terms, the **"qutip.mcsolve"** function can be used if the memory capacity becomes a limiting factor for a simulation because of the quadratic rise in the density matrix description. This Monte-Carlo quantum trajectory solver provides a useful method for larger-scale simulations, allowing simulations of the behavior of quantum systems with pulse-level control and noise to be performed with less memory usage.

Refer Appendix D to see the simulation code of the density matrix in QuTip.

2.3.3 Quantum Monte-Carlo trajectory

A large group of computational techniques known as "Monte Carlo methods" rely on repeated random sampling to produce numerical results. The fundamental idea is to harness randomness to find solutions to issues that could theoretically be deterministic. The dynamics of quantum systems are studied computationally via Monte Carlo trajectory simulations. The Monte Carlo trajectory approach tracks the individual pathways of quantum systems as they develop over time, in contrast to conventional methods that rely on the solution of differential equations or the manipulation of density matrices. Monte Carlo sampling with quantum trajectories is an alternate method for simulating quantum systems. This approach introduces a stochastic term using pseudo-random sampling and incorporates noise effects into a useful non-Hermitian Hamiltonian. The system is continuously subjected to the effective Hamiltonian's application, which incorporates the pertinent parts of Equation 2.9 involving Lindblad dissipators.

This Monte Carlo sampling approach allows researchers to efficiently capture the impact of noise on the dynamics of the quantum system. By taking into consideration the dissipative effects brought on by interactions with the environment, the technique offers a mechanism to replicate the behavior of open quantum systems. This method enables researchers to learn more about the behavior of the system by statistical sampling and analysis, which is especially helpful when investigating complicated systems where it might be difficult to find analytical answers.[15]

$$H_{eff} = H(t) - \frac{i}{2}\sum_{n}\Gamma_n^{\dagger}\Gamma_n \tag{2.11}$$

This is accomplished by comparing a randomly chosen value to the unnormalized wavefunction's norm. A quantum jump happens if the generated number is bigger than usual, keeping the wave function suitably normalized. The simulation mimics the inherently probabilistic nature of quantum systems by including this stochastic component. Random number comparisons can be used to determine whether quantum jumps—which reflect interactions with the environment or measurement events—occur. The wavefunction's validity is maintained by renormalization during the simulation. It is computationally viable to examine the non-deterministic properties of quantum systems using this method, which also offers a statistical understanding of phenomena like quantum measurements and decoherence effects.

$$|\psi(t+\delta t)\rangle = \frac{\Gamma_n|\psi(t+\delta t\rangle}{\sqrt{\langle\psi(t)|\Gamma_n^\dagger\Gamma_n|\psi(t)\rangle}} \tag{2.12}$$

The density matrix master equation solution has disadvantages compared to the quantum trajectory approach. By using the Hilbert space's dimension rather than its square, the computational complexity is reduced. Simulations are now more effective thanks to this enhancement. Additionally, the quantum trajectory technique allows for the simulation of specific trajectories, which sheds light on processes that average statistics would hide. It reveals minute subtleties that averaged dynamics could overlook. The quantity of trajectories required to assess a mean path with a minimal standard deviation trades off, though. Fortunately, concurrent computation of these trajectories using multi-core computing platforms is possible. The quantum trajectory approach's simulation tool, QuTiP, makes use of Python's multiprocessing module to increase computing performance.[25][15][24]

Chapter 3

Pulse

Pulse is a signal.
A signal in a quantum circuit carries out data and information about the state of the circuit. The circuit's behavior can be understood by analyzing the signals produced. The signal when flows only in one direction, which is either positive or negative, such a signal is termed a Unidirectional signal that's called a pulse signal.
A Pulse shape is formed by a rapid or sudden transient change from a baseline value to a higher or lower level value, which returns to the same baseline value after a certain time period. Such a signal can be termed a Pulse Signal.

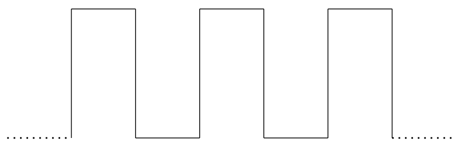


Figure 3.1: Pulse

A Pulse signal is a unidirectional, non-sinusoidal signal which is similar to a square signal but it is not symmetrical like a square wave. A series of continuous pulse signals is simply called a pulse train. A train of pulses indicates a sudden high level and a sudden low-level transition from a baseline level which can be understood as ON/OFF respectively. A signal in a circuit carries both data and knowledge about the state of the circuit Pulse is a precise electromagnetic signal that is used to manipulate the qubits.

$Differenttypesofpulses:$

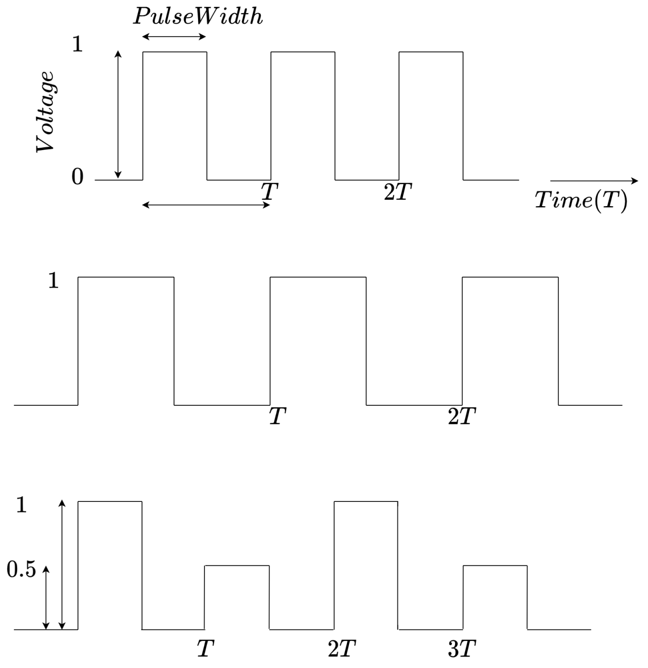


Figure 3.2: Different types of pulses

These pulses are typically used to control the interactions between qubits and perform quantum gates, which are the basic building blocks of quantum algorithms. Pulses are an essential component of quantum control systems, as they allowed for precise control over the quantum state of the qubits and enable the implementation of complex quantum algorithms.
It enables the user to determine the precise time dynamics of an experiment in a quantum system. It is particularly effective for strategies that reduce errors.[27]

3.1 Noisy Pulse

The noise pulses are unwanted electromagnetic signals, that can be impacted the behavior of qubits and compromised the accuracy of quantum circuits. These noise pulses can arise from various sources, such as electromagnetic interference from nearby electronic devices, temperature fluctuations, and quantum measurement noise. The quantum coherence of the qubits, which is necessary for executing accurate quantum

computations, can be harmed by noise pulses and faults in quantum gates. Thus, reducing and reducing the impact of noise pulses is a significant difficulty in quantum computing and researchers are actively working on developing techniques to reduce their impact on quantum systems.[23]
Noise in quantum circuits can have a significant impact on accuracy. Unlike classical bits that can only have a value of 0 or 1, quantum bits exist in a state known as a superposition which allows them to be in multiple states at the same time. This makes quantum systems much more sensitive to their environment and any disturbances can cause decoherence, or the loss of coherence of the quantum state, which results in errors in the computation. Noise can also affect the operation of quantum gates, which are the building blocks of quantum circuits, leading to inaccuracies in the results of quantum computations. This makes it challenging to build practical quantum computers that can perform large-scale quantum computations with high accuracy. To overcome these challenges, error correction techniques have been developed to mitigate the effects of noise in quantum circuits.

3.2 Theory

Quantum theory provides a detailed description of the physical quantum hardware that a circuit is implemented. The quantum system's temporal evolution serves to describe the dynamics of the system that implements a unitary gate in a quantum circuit. We think about unitary time evolution and open quantum dynamics for standalone or open quantum systems. By resolving the principal equation or by sampling Monte Carlo trajectories, the latter can be simulated. Here, we provide a brief description of both those techniques and the QuTiP solvers that correspond to them.

In a closed quantum system, the behavior of the system is determined by the Hamiltonian and the starting state. The Hamiltonian represents the energy of the system and contains information about its interactions and properties. To control and manipulate the quantum system, the Hamiltonian is divided into two parts: the non-controllable drift, represented by H_d, which accounts for natural evolution and changes over time, and the controllable terms, merged together as H_c, which allow us to influence the system's behavior. By combining these parts, we obtain the complete system Hamiltonian, which plays a crucial role in governing the dynamics of the quantum system.

$$H(t) = H_c(t) + H_d(t) \tag{3.1}$$

$$H(t) = \sum_k c_k(t) H_k + H_d(t) \tag{3.2}$$

where the time-dependent control coefficients, represented by $C_k(t)$, are used to steer the quantum system and achieve the desired unitary gates. These coefficients allow us to manipulate the system's evolution in time and guide it toward specific quantum operations. On the other hand, the terms in the Hamiltonian, denoted by H_k, describe the effects of physical controls that have been implemented on the system. These physical controls can include external fields, electromagnetic pulses, or other mechanisms that influence the behavior of the quantum system. By combining the control coefficients and the physical control terms in the Hamiltonian, we can shape and direct the dynamics of the quantum system to accomplish the desired operations and achieve the desired outcomes.

The unitary transformation applied to the quantum system, which is guided by the Hamiltonian, is found by solving the Schrodinger operator equation.

$$i\hbar \frac{\partial U(t)}{\partial t} = H(t)U(t) \tag{3.3}$$

A pulse-level description of the circuit, which includes the control Hamiltonian, is obtained by selecting the appropriate H(t) that can implement the desired unitaries or quantum circuit operations. The control coefficients $C_k(t)$ parametrization plays an important role in determining the behavior of the solver. These coefficients can be calculated automatically using control optimization methods, which will be discussed later, or derived from theoretical models.

3.3 Processor

The pulse-level simulation procedure is handled by the processor class. Connecting several components, it serves as the primary API interface. We offer a few already built-in processors that include compiler routines and Hamiltonian models. They differ mostly in the method used to determine the control pulse for a quantum circuit, which gives rise to various sub-classes.

There are two methods to determine the control pulses for a quantum system. The first method known as ModelProcessor, is more experiment-focused and depends on physical models. The control Hamiltonians required to build particular gates are already known and saved in the processor model characteristics when utilizing a ModelProcessor. This strategy is appropriate when it is possible to combine the control pulses necessary to create the needed quantum circuits in a well-established manner. Different models have already been created, including the spin chain, Cavity QED, and circuit QED, each with predefined driving Hamiltonians stored in their respective Model objects.

The second method, called OptPulseProcessor, is based on the optimal control module in QuTiP. One simply needs to define the Hamiltonians that are available in their quantum system when using this method. After that, the processor use algorithms to find the best control pulses for achieving the desired unitary development. This method is especially helpful when it is necessary to employ optimization techniques in order to establish the precise control pulses required for the desired quantum operations.

3.3.1 Optimal Pulse Processor

The **OptPulseProcessor** is a tool that chooses the best control pulses for a particular system by using the **optimize-pulse-unitary()** function from the optimal control module. The system's Hamiltonian is made up of two parts: a drift part and a control part. Only the control portion of the Hamiltonian is altered and optimized during the optimization phase to provide the required unitary transformation. Unitary time-dependent transformation

$$U(\Delta t) = exp(i.\Delta t[H_d + \sum_k u_k H_k]) \tag{3.4}$$

The **OptPulseProcessor** seeks to identify the control pulses that maximize the desired transformation while minimizing undesirable effects by employing optimization

methods. To customize the control pulses for attaining particular targets, this optimization procedure takes into account a variety of variables, such as limitations and user-defined objectives. We can modify and direct the dynamics of quantum systems towards desired outcomes using the **OptPulseProcessor**, which offers a practical and effective method for optimizing control pulses for quantum systems. It speeds the optimization process and aids in the discovery of ideal control approaches in a variety of applications, including quantum information processing and quantum technologies, by concentrating simply on the control portion of the Hamiltonian. We must pass the required parameters as keyword arguments to the **OptPulseProcessor's load-circuit()** function in order to make the discovery of optimal pulses easier. The overall evolution time **(evo-time)** and the number of time slots **(num-tslots)** for each gate in the quantum circuit are typically the two crucial inputs. Keyword arguments can be used to specify additional parameters. We can use the **optimize-pulse-unitary()** method to get a list of the choices that are available. Additionally, individual gate-specific parameter customization is allowed within the circuit, providing flexibility and fine-tuning during the optimization process.[2]
Refer Appendix E to see the code.

3.4 Model

The physical qubits must be accurately modeled in pulse-level simulations. In this approach, a processor-based instance of the Model class serves as a representation of the physical model. Important details about the particular quantum hardware being mimicked are kept in the Model object. This contains information on both the available control Hamiltonians that may be changed and the drift Hamiltonian, which symbolizes the features of the system that are out of our control. The Model object also contains details on possible sources of noise in the system. It is developed as a subclass of the Model class to define a particular physical model, such as a **SpinChainModel**, which offers a thorough and unique representation of the quantum hardware under simulation. This strategy guarantees that the physical properties and behavior of the quantum system's physical qubits are faithfully reflected in the pulse-level simulations. As in the sample at the start of this section, when initializing a particular Processor, a Model object is automatically produced for user convenience. It is feasible to gather crucial systemic information. Calling a function that returns a tuple with the Hamiltonian as a qutip will do this Qobj and the target qubits' indexes are two quantum objects. These target qubits stand in for the particular qubits that the control operations can be used on. By using the existing control Hamiltonians, it is now simpler to analyze and modify the system with this knowledge. These control Hamiltonians allow for accurate control and manipulation of the quantum system during simulations since they correlate to the operations that may be carried out on the target qubits. which the model object will access. Be aware that regardless of the internal implementation, a model can typically be correctly identified by the processor if the method **Model.get-control(label)** delivers the results in the proper format. We define it differently, for example. This will be useful in situations when it would be tedious to enumerate all the possible target qubit combinations, such as in an all-to-all coupled system using ions or neutral atoms.

When using models, it is possible to replicate actual qubits and their interactions more accurately by taking into consideration things like leakage levels and resonator-induced coupling. Predefined models implemented as subclasses of the **Model** class serve as an example of this increased realism. These include the fixed-frequency superconducting qubit model, the qubits-resonator model, and the spin chain model. The spin chain model represents a system with qubits arranged in a chain-like form, enabling the investigation of qubit interactions and group behavior. Resonators are used to pair qubits and facilitate information transfer between them, and they are a component of the qubits-resonator model. This model enables the study of qubit-resonator interactions and how they affect system dynamics. The fixed-frequency superconducting qubit model concentrates on superconducting qubits with set frequencies and simulates the key features of these kinds of qubits. Custom Hamiltonian models are also definable as subclasses of the preset models, allowing for the construction of unique representations of particular quantum hardware. This adaptability enables scientists to modify the simulation to take into account the specifics of their experimental setup or theoretical model. We learn more about the specific features of various quantum hardware combinations by using these predefined and bespoke models. In order to analyze the behavior of quantum systems in the real world and create efficient control schemes for quantum information processing and other quantum technologies, it is imperative to have this insight.

3.4.1 Spin Chain model

The quantum computing models **LinearSpinChain** and **CircularSpinChain** are based on the idea of a spin chain. These simulations represent the behavior of qubits placed in either a linear or circular pattern. Operations like σ_x, σ_z, and combinations like $\sigma_x \sigma_x + \sigma_y \sigma_y$ are all part of the control Hamiltonians in these models. The **OptPulseProcessor** uses ISWAP or SQRTISWAP as two-qubit gates to process gates in these models by first breaking down the needed gate into a universal gate set. In order to be applied to nearby qubits in the spin chain, these universal gates are then resolved into quantum gate sequences. In order to implement the desired gate, the processor determines the best control pulses by calculating the pulse coefficients that correspond to these quantum gates. We can successfully simulate and optimize the behavior of qubits in a spin chain configuration using these models and the decomposition method. As a result, it is possible to investigate different quantum gate operations and create the best control pulses for achieving desired gate operations in real-world quantum computing systems.[18] In quantum information processing, the **SpinChainModel** is a preset model that depicts a system of spin qubits with a spin-exchange interaction. One of the most basic kinds of interactions between adjacent qubits is this one, called the exchange interaction. The spin qubits are placed in a one-dimensional chain architecture called the **SpinChainModel**, which can have either open ends or closed ends. Only adjacent qubits in the chain of qubits interact with one another. In this model, σ_j^x and σ_j^z, which stand for operations on the x and z axes of the Bloch sphere, respectively, are used to describe the single-qubit control Hamiltonians.

$$H = \sum_{j=0}^{N-1} \Omega_j^x(t)\sigma_j^x + \Omega_j^z(t)\sigma_j^z + \sum_{j=0}^{N-2} g_j(t)(\sigma_j^x \sigma_{j+1}^x + \sigma_j^y \sigma_{j+1}^y) \tag{3.5}$$

The exchange Hamiltonian, which defines the interchange of spin states between neighboring qubits, is $\sigma_j^x \sigma_j^x + \sigma_j^y \sigma_j^y$, (j is index) and it realizes the exchange interaction. The exchange interaction and the control Hamiltonians are both a part of the overall Hamiltonian of the **SpinChainModel**. The time-dependent control coefficients, Ω_x, Ω_z, and g, represent a sum of elements. N is the number of qubits in the chain. We can simulate and investigate the dynamics of spin qubits in a one-dimensional chain while taking into account both the exchange interaction between neighboring qubits

and single-qubit control operations by using the **SpinChainModel**. This model is a useful resource for researching quantum information processing and creating spin-based quantum system control systems.
Refer Appendix E to see the code.

Qubit-resonator model: Qubit interactions are sometimes realized in experimental setups by connecting them with a quantum bus or resonator. The qubit-resonator model illustrates a system with a single resonator and several linked qubits. The resonator is only occasionally stimulated and mostly serves as a conduit for creating entanglement because the coupling strength between the qubits and the resonator is purposefully maintained low. provide quantitative requirements for the Tavis-Cummings model's viability and generate a more comprehensive Hamiltonian description that takes into account the photons' cascaded interactions with all subsequent atoms. demonstrate how our model's predictions can be qualitatively and quantitatively different from those derived using the Tavis-Cummings model[1]. The single-qubit control Hamiltonians in the qubit-resonator model are σ_x and σ_y, which represent operations on the x and y axes of the Bloch sphere. The Tavis-Cummings Hamiltonian, which includes the annihilation and creation operators of the resonator $(a, a^\dagger)$ and the lowering and raising operators of each qubit (σ_j^-, σ_j^+), describes the dynamics between the resonator and the qubits.
The equation

$$\alpha \sum_j a^\dagger \sigma_j^- + a\sigma_j^+ \tag{3.6}$$

where the sum is taken over all qubits, gives the interaction. Depending on the particular physical implementation, the control of the qubit-resonator coupling is determined. Depending on the experimental setup and the required operations to be carried out, it may include single-qubit control or multi-qubit control,

$$H = \sum_{j=0}^{N-1} \Omega_j^x(t)\sigma_j^x + \Omega_j^y(t)\sigma_j^y + g_j(t)(a^\dagger \sigma_j^- + a\sigma_j^+) \tag{3.7}$$

where j is an index parameter. The Hamiltonian of the resonator is shortened to finite values in the computational simulation.[7][6]

3.4.2 Superconducting qubit Model

Superconducting qubits : The construction of quantum computers using superconducting qubits is one of the most promising strategies. Leading contenders in the quest to create a quantum computer that can perform computations that are beyond the capabilities of current supercomputers are superconducting qubits. The noisy intermediate-scale quantum (NISQ) technological period, where non-error-corrected qubits are used to build quantum simulations and quantum algorithms, has been used to demonstrate prototype algorithms using superconducting qubits. This method also shows promise for the longer-term objective of developing larger-scale error-corrected quantum computers thanks to the recent demonstrations of numerous high-fidelity, two-qubit gates as well as operations on logical qubits in extensible superconducting qubit systems.

A three-level system is used in the SCQubits model to replicate a qubit's behaviour. The ground state and the first excited state are the only two states that make up the qubit subspace. Leakage, a phenomenon that happens when the population of the qubit subspace spills into other energy levels during single-qubit operations, is taken into account by this formulation. Two orthogonal quadrature operators, $a + a^\dagger$ and $i(a - a^\dagger)$, are used to operate the qubit. To correspond with the three-level qubit representation, these operators truncate the single-qubit control to a three-level operator. The superconducting qubits of the SCQubits model are placed in a one-dimensional framework, just as in the Spin Chain model. Only neighbouring qubits can interact with one another, making it possible to investigate interactions and systemic behaviour. A Cross Resonant pulse is used in the default implementation to realise the interaction between qubits, which makes it easier for information to flow between nearby qubits.[7] The SCQubits model helps us to better understand the behaviour and features of superconducting qubits by taking into consideration crucial elements like leakage and interaction between nearby qubits. The performance of superconducting qubit systems in applications involving quantum information processing can be understood and improved with the use of this information. Let's look at the cross-resonance interaction example, which is frequently used in fixed-frequency superconducting qubits.[**?**] For this interaction, the Hamiltonian can be expressed as follows

$$H = H_d + \sum_{j=0}^{N-1} \Omega_j^x (a^\dagger + a) + \Omega_j^y i(a^\dagger - a) + \sum_{j=0}^{N-2} \Omega^{cr1} \sigma_j^z \sigma_{j+1}^x + \Omega_j^{cr2} \sigma_j^x \sigma_{j+1}^z \tag{3.8}$$

where H_d represents the drift Hamiltonian

$$H_d = \sum_{i=0}^{N-1} \frac{\alpha_i}{2} a_i^\dagger a_i^\dagger a_i a_i \tag{3.9}$$

defined by the anharmonicity α_i of the second excited state. Based on the qubit-resonator detuning and coupling strength, the coefficients Ω^{cr1} and Ω^{cr2} are calculated. The simulation greatly shrinks the Hilbert space by using this efficient Hamiltonian, which enables the use of more qubits in the simulation. This benefit of reduced complexity comes from the ability for us to model larger systems and thoroughly investigate the effects of noise. It is also possible to create a CNOT gate utilizing this kind of interaction with the addition of single-qubit gates. One of the main benefits of the framework is the ability to choose the amount of complexity during the modeling process. The simulation is flexible enough for researchers to modify it to fit the particular needs of their experiment or study, making it an effective tool for studying quantum systems and replicating diverse noise effects.[14]
Refer Appendix E to see the code.

3.4.3 Compiler

In quantum computing, a compiler is necessary to transform a quantum circuit into pulse-level controls that can be used by the quantum hardware. The compiler is implemented in the framework as an instance of the **GateCompiler** class. There are various steps in this process. The compiler begins by breaking down each quantum gate in the circuit into native gates that the hardware can implement, such as x and y-axis rotations and the CNOT gate. In models involving chained or superconducting qubits, where the gate involves disconnected qubits, SWAP gates are added to create the necessary connectivity. For this reason, only one-dimensional chain architectures are currently supported. The compiler then converts each decomposed quantum gate into a description of a pulse-level control. It determines the ideal pulse length and strength required to perform the gate successfully while taking into account the hardware parameters specified in the Hamiltonian model. Continuous pulses can also have their shape set. A pulse scheduler is used to optimize the operation of many quantum gates. It investigates the potential for running these gates concurrently while accounting for temporal restrictions and available resources. By utilizing the parallel characteristics of the quantum gear, this scheduling procedure maximizes efficiency. Time-dependent pulse coefficients, or $c_j(t)$, are produced by the compiler for each control Hamiltonian, or H_j. These coefficients are stored in the processor and contain all the information required to implement the quantum circuit. Two NumPy arrays, one for the control amplitude and the other for the related time sequence, are used to represent each coefficient, $c_j(t)$. A cubic spline interpolation is used to approximate the coefficient for continuous pulses. Performance is enhanced by this method because it makes use of QuTiP's compiled Cython code. The framework comes with the relevant compilers and preset physical models that were covered before in this part. When calling the **Processor.load-circuit** method, these compilers are used. Let's use the three-qubit Deutsch-Jozsa algorithm as an example. the three models on which this algorithm's pulses were compiled. The fact that the same circuit can be compiled into utterly different pulse-level controllers is clear from these charts, demonstrating the compiler's adaptability to various physical models.

To swap the first two qubits in the spin chain model, SWAP gates are placed before and after the first CNOT gate. The CNOT is divided into two iSWAP gates and additional single-qubit corrections, whereas the SWAP gate is divided into three iSWAP gates. It is necessary to break down both the Hadamard and two-qubit gates into their component native gates (ISWAP and rotation on the x and z axes). varied gate times are produced by the square pulses in the compiled coefficients and the varied control coefficients on σ_z and σ_x. The constructed pulses for the superconducting-qubit processor have a Gaussian form. Given that the second excited level is only marginally adjusted away from the qubit transition energy, this is critical for superconducting qubits. Leakage to the non-computational subspace is typically avoided by a smooth pulse. The zeroth and first qubits are switched using SWAP gates, which are added in a manner similar to the spin chain. One SWAP gate is combined with three CNOT gates. We give the program the identical control Hamiltonian model for the spin chain model that we used for the optimal control model. All controls are active (non-zero pulse amplitude) for the majority of the execution period in the compiled optimal signals. We see that each optimized pulse is different for similar gates on various qubits (such as Hadamard), proving that the optimized solution is not the only one and that one may impose additional constraints, such as adaptations for the particular hardware. This makes it possible to optimize circuits elsewhere before sending them to QuTiP for pulse-level modeling.

3.4.4 Scheduler

One crucial component of the compilation is the scheduling of a circuit. Without it, the gates would have to be activated one at a time, leaving many qubits idle during circuit execution, which would lengthen execution time and lower fidelity. The scheduler is employed in the framework following the computation of each gate's control coefficient. It uses a scheduling algorithm to determine when each gate will open while maintaining the accuracy of the outcome. In order to save execution time, one can also decide whether permutation among commuting gates is permitted. The scheduler used here assumes that the connection in the given circuit matches the hardware at this point and disregards the hardware design. In predefined processors, the scheduler launches automatically when a circuit is loaded, necessitating no user intervention. Here, we give two examples of how to use the Scheduler class directly to help explain the scheduling method. A scheduler for quantum gates must take the commutation connection between the gates into account, unlike a scheduler for conventional gates. To reduce the execution time, two CNOT gates that are controlled by the same qubit but act on distinct target qubits can be switched. By building a dependency graph with commuting gates clustered in the same cycle, this flexibility is investigated. Within the cycle, the gate with the highest priority is carried out first. When the gates have various execution periods, however, optimizing the scheduling of commuting gates becomes more difficult. The scheduler is made simpler in the current design by leaving out more sophisticated methods that could further optimize the execution of commuting gates. These methods exist, but because gates have different execution speeds, they add complexity. However, the fundamental scheduling method built into the framework still takes gate commutation into account and aids in increasing the overall effectiveness of the quantum circuit execution.[8]

Scheduling optimizes the operation of quantum gates inside a circuit, which is essential to the compilation process. Without scheduling, the gates would be carried out in a sequential manner, resulting in idle qubits and longer execution times, both of which can lower fidelity. The scheduler is used in the structure after the control coefficients for each gate have been calculated. It uses a scheduling algorithm to decide when each gate should open while guaranteeing the accuracy of the final result. It's important to note that the scheduler built into the system assumes that the hardware connection of the circuit matches the hardware at this point and does not take the hardware design into account. In predefined processors, the scheduler immediately starts up when a circuit is loaded, requiring no user input. This expedites the compilation process and makes it easier for users to execute circuits.[20]

Optimal control: The optimal control technique in QuTiP is a different approach to finding optimized control pulses in addition to employing compilers with predefined gate-to-pulse mappings. This approach uses open-loop quantum control theory to find the optimal pulses for any input control Hamiltonians[4]. The approach creates control pulse settings for individual gates, sequences, and complete circuits through multivariable optimization of the control function $c_j(t)$. These pulses are made to be as accurate as possible with the planned evolution that the circuit has indicated. The **OptPulseProcessor** class in the qutip-qip interface is used to access the optimal control module in QuTiP. Users must first declare the control Hamiltonians that describe the system's physical controls in order to apply the algorithm. One way to do this is by giving an instance of the Model class, such as the **SpinChainModel**. Each quantum gate is enlarged to a unitary operation acting on the whole Hilbert space once the quantum circuit has been loaded, and this unitary operation is then supplied to the optimum control algorithm as the intended target. The method returns the driving pulses for the system, which are then combined to mimic the entire circuit with the associated physical control sequences.

3.4.5 Noise

The system's noise module makes it possible to include control and decoherence noise in simulations of open quantum systems according to Lindblad's description. Compared to the gate-based simulator we previously examined, this method offers a more straightforward and practical means of integrating noise. The current approach allows for the introduction of noise at different simulation levels, allowing users to concentrate on the dynamics of the dominating noise sources while resolving other noise sources, such as single-qubit relaxation, as collapse operators to increase computing efficiency. Users can dedicate CPU resources to the noise kinds that are most pertinent to their particular research or application thanks to this flexibility. The Noise class in the structure characterizes variations between the real physical dynamics and the compiled description in addition to compensating for flaws in the Hamiltonian model and circuit compilation. It accepts as input the compiled pulse-level circuit description and adds noise components to it, allowing the definition of noise that is correlated with the computed pulses. Three distinct noise models are already included in the system we will go into more detail on each of them later.

Noisy Hamiltonian Model: The basic weaknesses of the simulated quantum system may be included in the Model class, which describes the Hamiltonian model. As a result, these flaws could prevent the ideal pulse from fully realizing the desired unitary gate. Therefore, noise is inevitably injected into the simulation while creating a realistic Hamiltonian model. As an example, the physical qubit is represented as a multi-level system in the superconducting-qubit processor model. Population leakage from the qubit subspace may result from the presence of weak detuning between the qubit transition frequency and the second excitation level. The system becomes noisy as a result of this leakage. Another illustration is the always-on ZZ type cross-talk, which develops as a result of the interaction of physical qubits at higher levels. The superconducting qubit model has this kind of cross-talk as well, which adds to the simulation's noise.[21]

Control noise: Control noise explains problems in quantum system manipulation such as frequency drift or pulse amplitude deviations. The control coefficient cj(t) being impacted by the noise of random amplitude is an illustration of this. We replicate cross-talk-induced decoherence between two nearby ion trap qubits in order to show control noise. We design a two-qubit processor with the second qubit tuned a little differently from the first qubit. The initial qubit is subjected to a series of -pulses and random phases. We add noise so that the second qubit is also impacted by the same pulse. Detuning prevents this pulse from flipping the second qubit; instead, it causes it to display a diffusive behavior, which lowers the average fidelity of the second qubit in comparison to its original value. We simulate the results and reproduce them on our two-qubit processor. We simulate the Hamiltonian defined by

$$H = \Omega(t)(\sigma_0^x + \lambda\sigma_1^x) + \delta\sigma_1^z \tag{3.10}$$

where λ denotes the ratio of the cross-talk pulse's amplitudes and Ω is Rabi frequency. "**Rabi frequency:** When an atom is exposed to an oscillating electromagnetic field, the probability amplitudes of two energy levels change at a rate known as the Rabi frequency. The amplitude (not the strength) of the electromagnetic field and the Transition Dipole Moment between the two levels are directly connected. In a two-level system, population transfer between them will take place at the Rabi frequency if the incident light matches the energy difference between the levels. However, the population transfer will take place at a generalized Rabi frequency if the input light is only slightly out of resonance. It is crucial to remember that the Rabi frequency is a semiclassical idea since it treats the electromagnetic field as a continuous wave while treating the atom as having quantized energy levels." Our simulation shows the same fidelity decline curve as the results of the experiment. It's important to remember that, while the experimental results might include other types of noise, our simulation only considers the effect of cross-talk. We can detect and pinpoint the contributions of different noise sources thanks to these simulations.

Lindblad noise: When a quantum system interacts with its surroundings, such as a thermal bath, Lindblad noise, which results in information loss, happens. Non-unitary dynamics are produced when it is simulated using collapse operators. Decoherence, which is characterized by coherence times T_1 and T_2, which stand for energy loss and dephasing, respectively, is the most prevalent type of Lindblad noise. The Processor is given the parameters t1 and t2 and the relevant operators are automatically constructed to conveniently replicate this noise. A list of values, each corresponding to a different qubit, or a single value for every qubit are both acceptable for these parameters. The definition of the operator for T_1 is $\frac{a}{\sqrt{T_1}}$, where "a" denotes the destruction operator. The operator is specified as $a^{\dagger}a\sqrt{\frac{2}{T_2}}$, where T_2 is the pure dephasing time determined by the formula 1/T2 = 1/T2 1/(2T1). The destruction operator "a" is reduced to a two-level operator consistent in the case of qubits, which are two-level systems. Constant T_1 and T_2 values can be directly supplied when the Processor is initialized. DecoherenceNoise allows for the definition of unique collapse operators, including time-dependent ones. For instance, the code below uses **qutip.sigmam()** to define a collapse operator and increases linearly over time. It is important to note that the fitted decay does not begin at 1 because $\frac{\pi}{2}$ pulses are modeled as physical processes. This serves as an example of how state preparation mistakes can be incorporated into simulations. [3][17]

3.4.6 Controlled Pulse

The circuit is broken down into pulse-level controllers $c_j(t)H_j$ in this simulation framework, and the evolution of the physical qubits is simulated. The internal dynamics representation of the qutip-qip workflow facilitates the definition of custom pulse-dependent noise and aids in the comprehension of the simulation process. An instance of the class Pulse represents a control pulse and the noise that goes along with it. It is saved as an initialized Pulse object when an ideal control is constructed and sent back to the processor. A π-pulse is defined in the given code using the Hamiltonian's σ_x term, which flips the designated qubit (in this case, the zeroth qubit). The variable **tlist** specifies that the pulse will be administered for the duration of π. The parameters **coeff** and **tlist** describe the control coefficient c(t). The dynamics of a given control term can be found by combining the initialized Pulse object with the given Hamiltonian and target qubits. A Pulse can define different noise kinds, such as Lindblad or control noise, after being initialized with the optimum control. The **add-control-noise** technique is used to add a noisy Hamiltonian as control noise in one example. A Hamiltonian term called σ_z that can represent a frequency drift is introduced to the sample code. In a similar vein, collapse operators that require a particular control pulse can be introduced by using the **Pulse.add-lindblad-noise** method. Continuous functions can also be used to represent control pulses and noise in addition to steady pulses. In these circumstances, the continuous pulse coefficient is interpolated using a cubic spline using the **tlist** and **coeff** parameters as NumPy arrays. This makes it possible to use the QuTiP solvers' compiled Cython version, which performs better than using a Python function for the coefficient. The interpolation method is specified at the initialization of the Pulse object using the spline-kind="cubic" keyword argument. The Processor's interpolation method can also be specified using the same signature.

By creating a quantum gate specifically for the measurement procedure, measurements in quantum systems can be tailored. The resulting density matrix can then be used to derive the measurement's statistics. Additionally, by defining a new subclass of Noise, new types of noise can be added. This subclass adds noisy dynamics to the compiled ideal Pulse, enabling the simulation of more intricate and realistic settings.

Researchers can investigate the effects of various noise sources on quantum systems and acquire a greater understanding of their behavior by expanding the capabilities of the simulation structure.
Refer Appendix F to see the code

3.5 Conclusion

First of all, we introduced quantum computation. Then we created a quantum circuit with the help of quantum gates and measurements and find out their unitaries, and states with their probabilities with the help of circuit simulation using Qutip. Introduced pulse and noise and their effect on a quantum circuit. We gave a structure for the pulse-level simulation of quantum circuits, allowing the study of noisy quantum devices on classical computers. The addition of new tools for simulating controls in noisy quantum circuits improves the capabilities of the solvers and quantum circuit model in QuTiP. We have supplied predefined quantum hardware models, compilation and scheduling procedures, and noise models that can be modified to concentrate computing resources on important physical dynamics during noise analysis. The framework's modular structure enables integration with a range of hardware models, gate breakdown methods, and optimization strategies. It is especially suitable for simulating processors based on quantum error-correcting techniques and bosonic models. State creation and measurement can be represented as specialized gates to emulate them as noisy physical processes. Benefits of pulse-level simulation include rapid experimental result verification, the creation of quantum algorithms (such as variational quantum algorithms), and testing compilation and scheduling systems with accurate noise models. It also makes it possible to explore non-Markovian types of noise, quantum error correction codes, quantum damping methods, and noise measurement in model devices.

Chapter 4

Future Scope

The simulation of quantum dynamics following the master equation for various hardware models and simulation of the noisy quantum circuit at pulse-label to make it noise free (boost their accuracy). QuTiP's future work will focus on giving performance improvements. The capabilities of the framework-supported quantum control optimization algorithms are intended to be improved by planned advances in both qutip and qutip-qip. There are chances to incorporate the GOAT(Gradient Optimization Algorithm) method, allowing for efficient use of QuTiP's solvers for circuit control optimization, notably for modeling universal gate operations and quantum machine learning. The quantum open-source ecosystem's integration with other software frameworks is another area of exploration. In order to connect qutip-qip, attempts have been made to standardize the quantum intermediate representation of quantum circuits. An attractive area for research and development is the application of qutip-qip in the context of open quantum hardware. The framework may be used in research labs using various quantum technologies as a foundation for API interconnection between simulation and hardware control. This creates opportunities for cooperation and experimentation with different hardware platforms, which will accelerate developments in the area of quantum computing.

Bibliography

[1] Martin Blaha, Aisling Johnson, Arno Rauschenbeutel, and Jürgen Volz. Beyond the tavis-cummings model: Revisiting cavity QED with ensembles of quantum emitters. *Physical Review A*, 105(1), jan 2022.

[2] Gabriel Blaj, C. Kenney, Angelo Dragone, G. Carini, S. Herrmann, Philip Hart, Astrid Tomada, Jason Koglin, Gunther Haller, Sébastien Boutet, Marc Messerschmidt, G. Williams, Matthieu Chollet, Georgi Dakovski, S. Nelson, Jack Pines, Sanghoon Song, and J. Thayer. Optimal pulse processing, pile-up decomposition, and applications of silicon drift detectors at lcls. *IEEE Transactions on Nuclear Science*, PP:1–1, 10 2017.

[3] Heinz-Peter Breuer and Francesco Petruccione. *The Theory of Open Quantum Systems*. 01 2006.

[4] D. D'Alessandro. *Introduction to Quantum Control and Dynamics*. Chapman & Hall/CRC Applied Mathematics & Nonlinear Science. Taylor & Francis, 2007.

[5] Firat Diker. Deterministic construction of arbitrary w states with quadratically increasing number of two-qubit gates, 2016.

[6] Fu-Quan Dou and Fang-Mei Yang. Superconducting transmon qubit-resonator quantum battery. *Physical Review A*, 107(2), feb 2023.

[7] Pol Forn-Diaz. *Superconducting Qubits and Quantum Resonators*. PhD thesis, 09 2010.

[8] Gian Giacomo Guerreschi and Jongsoo Park. Two-step approach to scheduling quantum circuits. *Quantum Science and Technology*, 3(4):045003, jul 2018.

[9] Masahiro Hotta. Energy entanglement relation for quantum energy teleportation. *Physics Letters A*, 374(34):3416–3421, jul 2010.

[10] Masahiro Hotta. Quantum energy teleportation: An introductory review, 2011.

[11] Kazuki Ikeda. Criticality of quantum energy teleportation at phase transition points in quantum field theory. *Phys. Rev. D*, 107:L071502, Apr 2023.

[12] Kazuki Ikeda. Long-range quantum energy teleportation and distribution on a hyperbolic quantum network, 2023.

[13] Kazuki Ikeda. Realization of quantum energy teleportation on superconducting quantum hardware, 2023.

[14] Morten Kjaergaard, Mollie E. Schwartz, Jochen Braumüller, Philip Krantz, Joel I.-J. Wang, Simon Gustavsson, and William D. Oliver. Superconducting qubits: Current state of play. *Annual Review of Condensed Matter Physics*, 11(1):369–395, 2020.

[15] M. Kritsotakis and I. K. Kominis. Retrodictive derivation of the radical-ion-pair master equation and monte carlo simulation with single-molecule quantum trajectories. *Phys. Rev. E*, 90:042719, Oct 2014.

[16] Boxi Li, Shahnawaz Ahmed, Sidhant Saraogi, Neill Lambert, Franco Nori, Alexander Pitchford, and Nathan Shammah. Pulse-level noisy quantum circuits with QuTiP. *Quantum*, 6:630, jan 2022.

[17] Daniel A. Lidar. Lecture notes on the theory of open quantum systems, 2020.

[18] Daniel Loss and David P. DiVincenzo. Quantum computation with quantum dots. *Phys. Rev. A*, 57:120–126, Jan 1998.

[19] Albert Messiah. *Quantum mechanics.* Courier Corporation, 2014.

[20] Tzvetan Metodi, Darshan Thaker, Andrew Cross, Frederic Chong, and Isaak Chuang. Scheduling physical operations in a quantum information processor. *Proceedings of SPIE - The International Society for Optical Engineering*, 6244, 05 2006.

[21] Pranav Mundada, Gengyan Zhang, Thomas Hazard, and Andrew Houck. Suppression of qubit crosstalk in a tunable coupling superconducting circuit. *Phys. Rev. Appl.*, 12:054023, Nov 2019.

[22] Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.

[23] Hendra I. Nurdin. *Quantum Stochastic Processes and the Modelling of Quantum Noise*, pages 1808–1815. Springer International Publishing, Cham, 2021.

[24] Jason F. Ralph, Simon Maskell, and Kurt Jacobs. Multiparameter estimation along quantum trajectories with sequential monte carlo methods. *Phys. Rev. A*, 96:052306, Nov 2017.

[25] J. Schachenmayer, A. Pikovski, and A. M. Rey. Many-body quantum spin dynamics with monte carlo trajectories on a discrete phase space. *Phys. Rev. X*, 5:011022, Feb 2015.

[26] Barak Shoshany. ”thinking quantum”: Lectures on quantum theory, 2018.

[27] A tA v, MD SAJID ANIS, Abby-Mitchell, Héctor Abraham, AduOffei, Rochisha Agarwal, Gabriele Agliardi, Merav Aharoni, Vishnu Ajith, Ismail Yunus Akhalwaya, Gadi Aleksandrowicz, Thomas Alexander, Matthew Amy, Sashwat Anagolum, Anthony-Gandon, and Israel F. Čepulkovskis. Qiskit: An open-source framework for quantum computing, 2021.

[28] Yuchen Wang, Ellen Mulvihill, Zixuan Hu, Ningyi Lyu, Saurabh Shivpuje, Yudan Liu, Micheline B. Soley, Eitan Geva, Victor S. Batista, and Sabre Kais. Simulation of open quantum system dynamics based on the generalized quantum master equation on quantum computing devices, 2022.

[29] Wikipedia contributors. Quantum teleportation — Wikipedia, the free encyclopedia, 2023. [Online; accessed 5-February-2023].

[30] Howard M Wiseman and Gerard J Milburn. *Quantum measurement and control.* Cambridge university press, 2009.

Appendix A

Teleportation circuit in QISKIT

Initializing the state to be teleported

First, create a quantum circuit that creates the state $\sqrt{0.70}|o\rangle + \sqrt{0.30}|1\rangle$

```
import numpy as np
from qiskit import IBMQ, Aer
from qiskit.quantum_info import Operator
from qiskit import QuantumCircuit, execute
from qiskit.visualization import plot_histogram
from qiskit.tools.jupyter import *

def initialize_qubit(given_circuit, qubit_index):
    import numpy as np
    given_circuit.initialize([np.sqrt(0.70), np.sqrt(0.30)],
    qubit_index)
    return given_circuit

#create entanglement between Alice's and Bob's qubits.

def entangle_qubits(given_circuit, qubit_Alice, qubit_Bob):
    given_circuit.h(qubit_Alice)
    given_circuit.cx(qubit_Alice, qubit_Bob)
    return given_circuit
```

```
#do a Bell measurement of Alice's qubits.

def bell_meas_Alice_qubits(given_circuit, qubit1_Alice,
qubit2_Alice, clbit1_Alice, clbit2_Alice):
    given_circuit.cx(qubit1_Alice, qubit2_Alice)
    given_circuit.h(qubit1_Alice)
    given_circuit.barrier()
    given_circuit.measure(qubit1_Alice, clbit1_Alice)
    given_circuit.measure(qubit2_Alice, clbit2_Alice)
    return given_circuit
```

Finally, we apply controlled operations on Bob's qubit.

1. an **X gate** is applied on Bob's qubit if the measurement outcome of Alice's second qubit, clbit2_Alice, is 1.
2. a **Z gate** is applied on Bob's qubit if the measurement outcome of Alice's first qubit, clbit1_Alice, is 1.

```
def controlled_ops_Bob_qubit(given_circuit, qubit_Bob, clbit1_Alice,
clbit2_Alice):
    given_circuit.x(qubit_Bob).c_if(clbit2_Alice, 1)
    given_circuit.z(qubit_Bob).c_if(clbit1_Alice, 1)
    return given_circuit

#imports
from qiskit import QuantumRegister, ClassicalRegister
#set up the qubits and classical bits
all_qubits_Alice = QuantumRegister(2)
all_qubits_Bob = QuantumRegister(1)
creg1_Alice = ClassicalRegister(1)
creg2_Alice = ClassicalRegister(1)


#QUANTUM TELEPORTATION circuit here
```

```
# Initialize
mycircuit = QuantumCircuit(all_qubits_Alice, all_qubits_Bob,
creg1_Alice, creg2_Alice)
initialize_qubit(mycircuit, 0)
mycircuit.barrier()
#Entangle
entangle_qubits(mycircuit, 1, 2)
mycircuit.barrier()
#Do a Bell measurement
bell_meas_Alice_qubits(mycircuit, all_qubits_Alice[0],
all_qubits_Alice[1], creg1_Alice, creg2_Alice)
mycircuit.barrier()
# Apply classically controlled quantum gates
controlled_ops_Bob_qubit(mycircuit, all_qubits_Bob[0], creg1_Alice,
creg2_Alice)

#Look at the complete circuit
mycircuit.draw(output='mpl')
```

Out[14]:

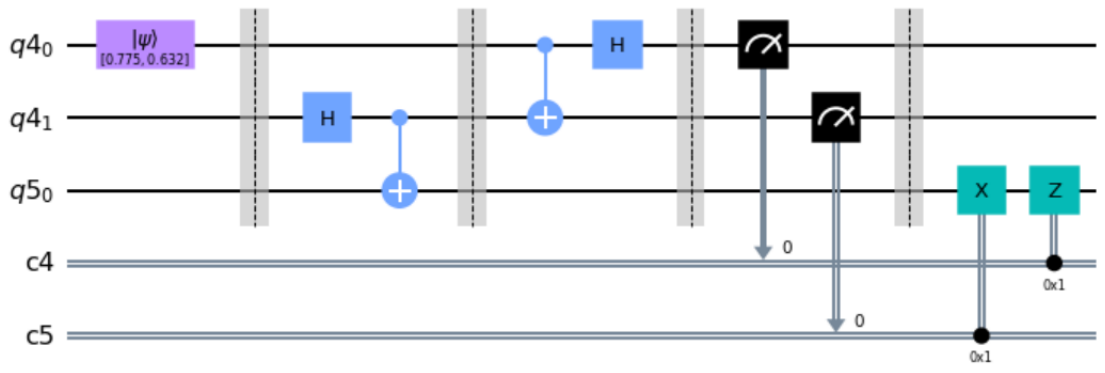


Figure A.1: Teleportation circuit

Appendix B

Deutsh Josza Algorithm in QISKIT

```
import numpy as np
from qiskit import IBMQ, BasicAer
from qiskit import QuantumCircuit, execute
from qiskit.visualization import plot_histogram
from qiskit.tools.jupyter import *

def dj_oracle(case,n):
    oracle_qc=QuantumCircuit(n+1)
    if case=="bal":
        for qubit in range(n):
            oracle_qc.cx(qubit,n) then last qubit=0
    if case=="cons":
        out=np.random.randint(2)
        if out==1:
            oracle_qc.x(n)


    oracle_gate=oracle_qc.to_gate()
    oracle_gate.name="oracle"
```

```python
    return oracle_gate

#DEUTSCH JOZSA ALGORITHM
def dj_algorithm(n,case='random'):
    dj_ckt=QuantumCircuit(n+1,n)
    for qubit in range(n):
        dj_ckt.h(qubit)

    dj_ckt.x(n)
    dj_ckt.h(n)
    if case=='random':
        random=np.random.randint(2)
        if random==0:
            case="cons"
        else:
            case="bal"

    oracle=dj_oracle(case,n)
    dj_ckt.append(oracle,range(n+1))

    for i in range(n):
        dj_ckt.h(i)
        dj_ckt.measure(i,i)
    return dj_ckt
n=4
dj_ckt=dj_algorithm(n)
dj_ckt.draw()
```

Out[39]:

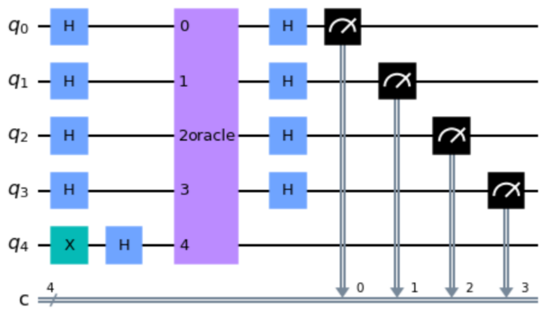


Figure B.1: DJ circuit for n=4 qbits

TO CHECK RESULT

```
backend=BasicAer.get_backend('qasm_simulator')
shots=2084
sj_ckt=dj_algorithm(n,'bal')
results=execute(dj_ckt,backend=backend,shots=shots).result()
answer=results.get_counts()
plot_histogram(answer)
```

Out[37]:

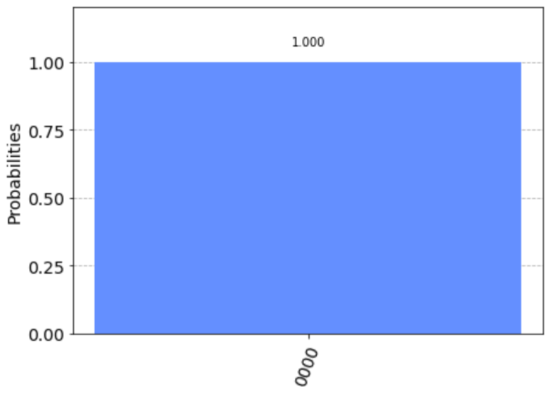


Figure B.2: result

Appendix C

Plotting a Quantum Circuit in QuTip

Demonstration of the quantum circuit with the help of gate and registers First, create a quantum circuit with the help of available gates and a register. Then we will add a useful function which is called **QubitCircuit.propagators()**. That will give us unitaries associated with the sequence of gates or full dimensions.

```
import numpy as np
from qutip_qip.circuit import QubitCircuit
from qutip_qip.operations import Gate
from qutip import tensor ,basis

qc = QubitCircuit(N=2, num_cbits=1)
swap_gate = Gate(name="SWAP", targets=[0,1])

qc.add_gate(swap_gate)
qc.add_measurement("M0",targets=[1],classical_store=0)

qc.add_gate("CNOT", controls=0, targets=1)
qc.add_gate("X", targets=0, classical_controls=[0])
qc.add_gate(swap_gate)
```

```
print(qc.gates)
```

```
U_list=qc.propagators(ignore_measurement=True)
print(U_list)

# Another argument to get unitaries (same thing)
U_list=qc.propagators(expand=False, ignore_measurement=True)
print(U_list)
```

Plotting a Quantum Circuit:

```
from qutip_qip.circuit import QubitCircuit
from qutip_qip.operations import Gate

# create quantum circuit
qc = QubitCircuit(2, num_cbits=1)
qc.add_gate("CNOT", controls=0, targets=1)
qc.add_gate("SNOT", targets=1)
qc.add_gate("ISWAP", targets=[0,1])
qc.add_measurement("M0", targets=1, classical_store=0)
# plot quantum circuit
qc.png()
```

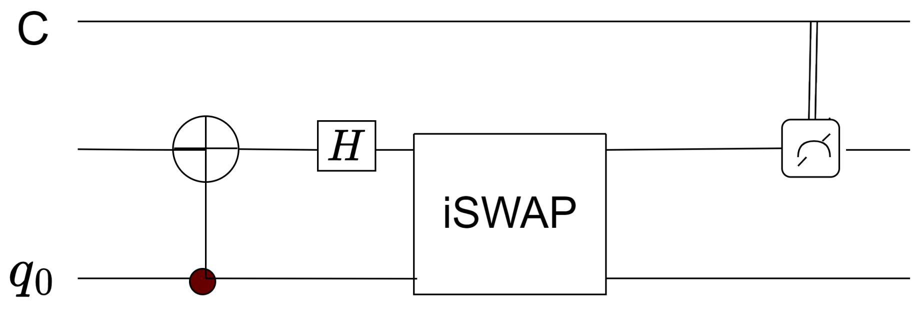


Figure C.1: Quantum circuit

Appendix D

Run a quantum circuit (OLCS)

First, create a W-state quantum circuit.

```python
from qutip_qip.circuit import QubitCircuit
from qutip_qip.operations import (Gate, controlled_gate, hadamard_transform)
from matplotlib import pyplot as plot
from qutip import tensor, basis

def controlled_hadamard():
    # Controlled Hadamard
    return controlled_gate(hadamard_transform(1), 2, control=0, target=1, control_val

qc = QubitCircuit(N=3, num_cbits=3)
qc.user_gates = {"cH": controlled_hadamard}
qc.add_gate("QASMU", targets=[0], arg_value=[1.91063, 0, 0])
qc.add_gate("cH", targets=[0,1])
qc.add_gate("TOFFOLI", targets=[2], controls=[0, 1])
qc.add_gate("X", targets=[0])
qc.add_gate("X", targets=[1])
qc.add_gate("CNOT", targets=[1], controls=0)

zero_state = tensor(basis(2, 0), basis(2, 0), basis(2, 0))
result = qc.run(state=zero_state)
```

```
wstate = result

print(wstate)

qc.add_measurement("M0", targets=[0], classical_store=0)
qc.add_measurement("M1", targets=[1], classical_store=1)
qc.add_measurement("M2", targets=[2], classical_store=2)

result = qc.run_statistics(state=tensor(basis(2, 0), basis(2, 0), basis(2, 0)))
states = result.get_final_states()
probabilities = result.get_probabilities()

for state, probability in zip(states, probabilities):
    print("State:\n{}\nwith probability {}".format(state, probability))
```

Refer subsection 2.1.1 to see the result.

```
# Circuit Simulator
from qutip_qip.circuit import CircuitSimulator

sim = CircuitSimulator(qc, state=zero_state)

print(sim.step())

# Precomputing
sim = CircuitSimulator(qc, precompute_unitary=True)

print(sim.ops)
```

Refer subsection 2.2.1 to see the result.

Density Matrix Simulation

```
qc = QubitCircuit(N=3, num_cbits=3)
qc.user_gates = {"cH": controlled_hadamard}
qc.add_gate("QASMU", targets=[0], arg_value=[1.91063, 0, 0])
qc.add_gate("cH", targets=[0,1])

qc.add_gate("TOFFOLI", targets=[2], controls=[0, 1])
qc.add_gate("X", targets=[0])

qc.add_gate("X", targets=[1])
qc.add_gate("CNOT", targets=[1], controls=0)

qc.add_measurement("M0", targets=[0], classical_store=0)
qc.add_measurement("M0", targets=[1], classical_store=0)
qc.add_measurement("M0", targets=[2], classical_store=0)

sim = CircuitSimulator(qc, mode="density_matrix_simulator")

print(sim.run(zero_state).get_final_states()[0])
```

Refer section 2.3 to see the result.

Appendix E

Deutsh Josza Algorithm in QuTip

```
import numpy as np
from qutip import basis
from qutip_qip.circuit import QubitCircuit
from qutip_qip.device import LinearSpinChain

# Define a circuit
qc = QubitCircuit(3)
qc.add_gate("X", targets=2)

qc.add_gate("SNOT", targets=0)
qc.add_gate("SNOT", targets=1)
qc.add_gate("SNOT", targets=2)

# Oracle function f(x)
qc.add_gate("CNOT", controls=0, targets=2)
qc.add_gate("CNOT", controls=1, targets=2)

qc.add_gate("SNOT", targets=0)
qc.add_gate("SNOT", targets=1)

# Run gate-level simulation
init_state = basis([2,2,2], [0,0,0])
```

```
ideal_result = qc.run(init_state)

# Run pulse-level simulation
processor = LinearSpinChain(num_qubits=3, sx=0.25, t2=30)
processor.load_circuit(qc)
tlist = np.linspace(0, 20, 300)
result = processor.run_state(init_state, tlist=tlist)
```

Spin Chain Model

```
# Deutsch-Jozsa algorithm
from qutip_qip.circuit import QubitCircuit

qc = QubitCircuit(3)
qc.add_gate("X", targets=2)

qc.add_gate("SNOT", targets=0)
qc.add_gate("SNOT", targets=1)
qc.add_gate("SNOT", targets=2)

# Oracle function f(x)
qc.add_gate("CNOT", controls=0, targets=2)
qc.add_gate("CNOT", controls=1, targets=2)

qc.add_gate("SNOT", targets=0)
qc.add_gate("SNOT", targets=1)

from qutip_qip.device import LinearSpinChain
spinchain_processor = LinearSpinChain(num_qubits=3, t2=30)  # T2 = 30
spinchain_processor.load_circuit(qc)
fig, ax = spinchain_processor.plot_pulses(figsize=(8, 5))
fig.show()
```

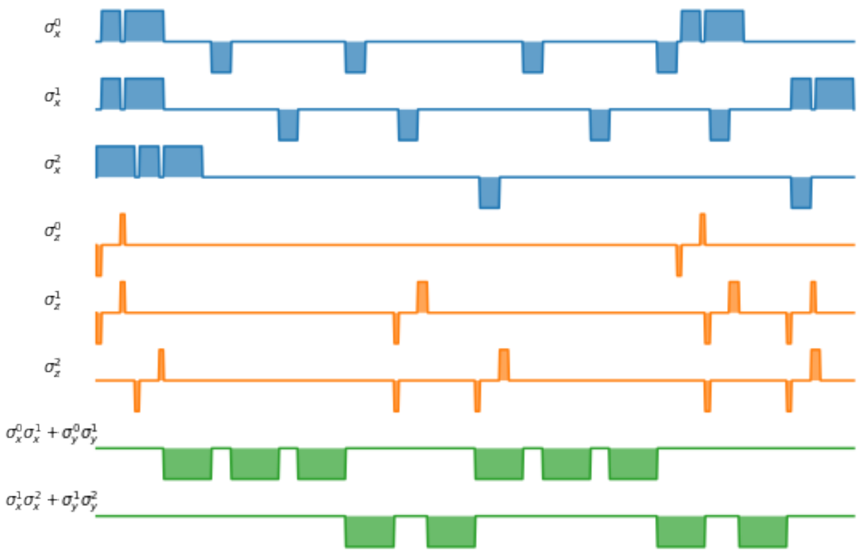


Figure E.1: Result

Superconducting qubits Model

```
from qutip_qip.device import SCQubits

scqubits_processor = SCQubits(num_qubits=3)
scqubits_processor.load_circuit(qc)
fig, ax = scqubits_processor.plot_pulses(figsize=(8, 5))

fig.show()
```

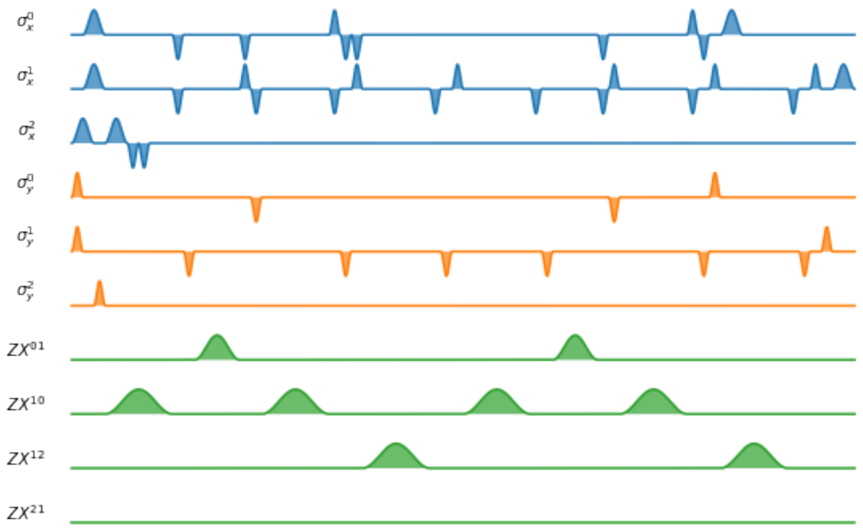


Figure E.2: Result

Optimal Pulse Processor

We are applying optpulseProcessor to get control pulses.

```
from qutip_qip.device import OptPulseProcessor, SpinChainModel

setting_args = {"SNOT": {"num_tslots": 6, "evo_time": 2},
                "X": {"num_tslots": 1, "evo_time": 0.5},
                "CNOT": {"num_tslots": 12, "evo_time": 5}}
opt_processor = OptPulseProcessor(
    num_qubits=3, model=SpinChainModel(3, setup="linear"))
opt_processor.load_circuit(  # Provide parameters for the algorithm
    qc, setting_args=setting_args, merge_gates=False,
    verbose=True, amp_ubound=5, amp_lbound=0)

fig, ax = opt_processor.plot_pulses(figsize=(8, 5))
fig.show()
```

Result:

Gate	Final Fidelity Error	Final Gradient Normal	**Terminated due to**	**No. of iterations**
0	$9.185e-12$	0.00084	$Goalachieved$	8
1	$1.581e-05$	0.0025	$functionconverged$	226
2	0.0137	0.0128	$functionconverged$	247
3	$1.397e-07$	0.00017	$functionconverged$	445
4	$1.572e-08$	0.00014	$functionconverged$	119
5	$1.105e-08$	0.00011	$functionconverged$	111
6	0.039	0.069	$functionconverged$	196
7	0.012	0.023	$functionconverged$	355

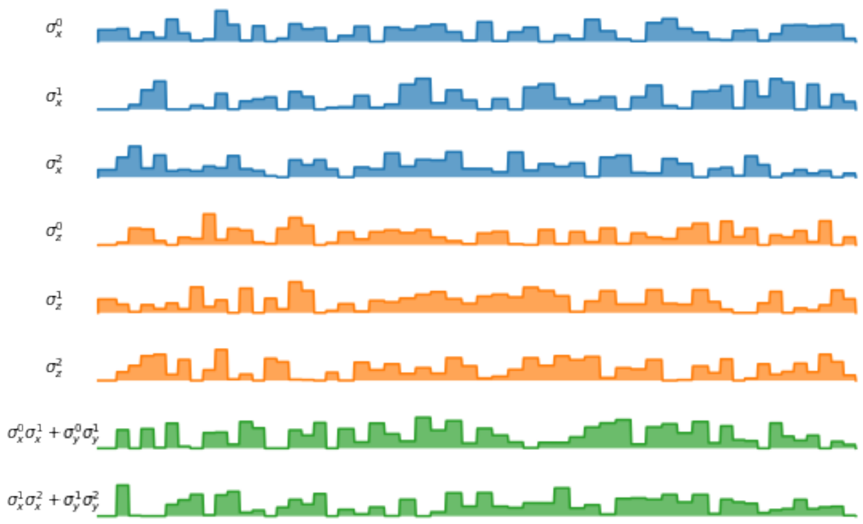


Figure E.3: controlled pulse

Pulse

```
from qutip_qip.compiler import GateCompiler

compiler = GateCompiler()
coeff, tlist = compiler.generate_pulse_shape(
    "hann", 1000, maximum=2., area=1.)
fig, ax = plot.subplots(figsize=(4,2))
ax.plot(tlist, coeff)
ax.set_xlabel("Time")
ax.set_ylabel()

fig.show()
```

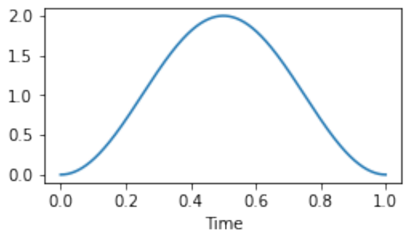


Figure E.4: Pulse Shape

Appendix F

Simulation of noisy pulse

First, Demonstrates a little Gaussian noise on the pulse amplitude. For visualization, we plot the noisy pulse intensity instead of the state fidelity.

```
from qutip import sigmaz, sigmay
from qutip_qip.device import Processor
from qutip_qip.noise import RandomNoise

# add control Hamiltonians
processor = Processor(1)
processor.add_control(sigmaz(), targets=0, label="sz")

# define pulse coefficients and tlist for all pulses
processor.set_coeffs({"sz": np.array([0.3, 0.5, 0. ])})
processor.set_tlist(np.array([0., np.pi/2., 2*np.pi/2, 3*np.pi/2]))

# define noise, loc and scale are keyword arguments for np.random.normal
gaussnoise = RandomNoise(
            dt=0.01, rand_gen=np.random.normal, loc=0.00, scale=0.02)
processor.add_noise(gaussnoise)

# Plot the ideal pulse
fig1, axis1 = processor.plot_pulses(
    title="Original control amplitude", figsize=(5,3),
    use_control_latex=False)

# Plot the noisy pulse
qobjevo, _ = processor.get_qobjevo(noisy=True)
noisy_coeff = qobjevo.to_list()[1][1] + qobjevo.to_list()[2][1]
fig2, axis2 = processor.plot_pulses(
    title="Noisy control amplitude", figsize=(5,3),
    use_control_latex=False)
axis2[0].step(qobjevo.tlist, noisy_coeff)
```

Result:

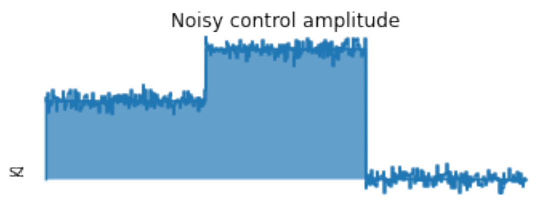


Figure F.1: Pulse with noise

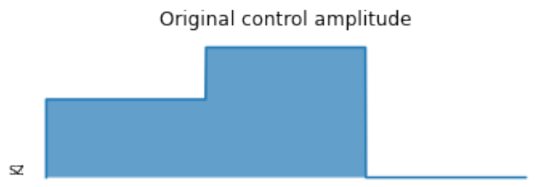


Figure F.2: Pulse without noise

We got the controlled noisy pulse.