Energy Efficient Universal Quantum Optimal Control

Master Thesis

Sebastiaan Fauquenot

Energy Efficient Universal Quantum Optimal Control

Master Thesis

by

Sebastiaan Fauquenot

in partial fulfilment of the requirements for the degree of

Master of Science
in Applied Physics

at the Delft University of Technology,
to be defended publicly on Tuesday May 28, 2024 at 2:00 PM.

Student number:		
Project duration:	October 9, 2023 – May 9, 2024	
Thesis committee:	Dr. rer. nat. Sebastian Feld,	QuTech, TU Delft (supervisor)
	Dr. Ir. Fabio Sebastiano,	QuTech, TU Delft (examiner)
	Dr. Ir. Tim Taminiau,	QuTech, TU Delft (examiner)
	Dr. Ir. Aritra Sarkar,	Q&CE, TU Delft (daily-supervisor)

Abstract

Quantum optimal control is a rapidly growing field with diverse methods and applications [1]. In this work, the possibility of using quantum optimal control techniques to co-optimize the energetic cost and the process fidelity of a quantum unitary gate is investigated. The theoretical definition and quantization of quantum unitary gates, as well as the relationship between the process fidelity and the energetic cost of a quantum unitary gate are explored. Two different quantum optimal control methods to co-optimize both fidelity and energetic cost, i.e., the Gradient Ascent Pulse Engineering method [2] and model-free Deep Reinforcement Learning [3] are investigated. The performance of both quantum optimal control techniques in the presence of noise is probed. We find that the energetic cost of a quantum unitary gate can be quantized by integrating the control pulses and norm of the corresponding Hamiltonian operators over the total time duration of the unitary, and for single qubit gates by calculating the arc length of the quantum unitary gate on the Bloch sphere [4]. A Pareto optimal front between the process fidelity and the energetic cost of a quantum gate is identified, where a lower energetic cost yields an inherently lower process fidelity. A python package called ”EUQOC” (Energy Efficient Universal Quantum Optimal Control) has been created to implement energy optimal quantum gate synthesis, both with the Energy Optimal Gradient Ascent Pulse Engineering (EO-GRAPE) method and by model-free Deep Reinforcement Learning. It is found that the EO-GRAPE method performs better than the reinforcement learning methods, for all noise settings and neural network sizes. For future work, the optimization problem could be translated to the frequency domain to increase the computational efficiency. Furthermore, the relationship between information and energy can be investigated by looking at the complexity of the pulse or the decomposition of the quantum unitary gate.

List of Acronyms

RSA	Rivest–Shamir–Adleman
QEC	Quantum Error Correction
QOC	Quantum Optimal Control
QOCT	Quantum Optimal Control Theory
QGS	Quantum Gate Synthesis
EOQGS	Energy Optimized Quantum Gate Synthesis
VQA	Variational Quantum Algorithm
qubit	Quantum Bit
CNOT	Controlled NOT
CX	Controlled X
CZ	Controlled Z
TDSE	Time Dependent Schrödinger Equation
NISQ	Noisy Intermediate Scale Quantum
FTQC	Fault Tolerant Quantum Computing
ML	Machine Learning
RL	Reinforcement Learning
GRAPE	Gradient Ascent Pulse Engineering
QISA	Quantum Instruction Set Architecture
SU	Special Unitary Group 2
DRAG	Derivative Removal by Adiabatic Gates
GOAT	Gradient Optimization of Analytical Controls
CRAB	Chopped Random Basis Optimization
GA	Genetic Algorithm
DE	Differential Evolution
QAOA	Quantum Approximate Optimization Algorithm
QSL	Quantum Speed Limit
QRLA	Quantum Reinforcement Learning Agent
EO-GRAPE	Energy Optimized Gradient Ascent Pulse Engineering
WS	Warm Start
WOWS	Without Warm Start

Contents

1 Introduction

If you think you understand quantum mechanics, you don't understand quantum mechanics.
- Richard Feynman

1.1. Quantum Computing

In 1982, Richard Feynman introduced the notion of using quantum phenomena for simulating physics on computers [5]. This notion, currently known as Quantum Computing, has seen rapid development in both theoretical and practical aspects since then, and is currently widely researched by both academia and industry. Over the years, theorists have developed new quantum algorithms, with widespread applications, from breaking RSA encryption, to simulating natural phenomena on a quantum dynamical level. At the same time, experimentalists have constructed small-scale quantum computers using various technologies, ranging from superconducting circuits, to trapped ions, and electron spins. While the progress since 1982 has been immense, there are still a multitude of challenges facing quantum computing today. Possibly one of the biggest problems quantum researchers are facing, is decoherence: the loss of quantum coherence due to interactions with the environment. There are various approaches being actively investigated to mitigating decoherence, but can be roughly separated into three categories: Quantum Error Correction, Quantum Error Mitigation and using novel qubit modalities and materials. The first focuses on developing new algorithms and error correction schemes to identify and correct errors. The latter focuses on improving the performance of the computer by either improving hardware, software or both. Quantum Optimal Control focuses on methods to design and implement electromagnetic field configurations that can effectively steer quantum processes at the atomic or molecular scale in the best way possible. In this work we will be using methods from Quantum Optimal Control theory, to improve the performance and energy efficiency of quantum computers.

In the remaining part of this chapter, we will be discussing the relevant quantum mechanics and quantum information theory necessary for understanding both the intricacies and relevance of Quantum Optimal Control. We will first look at the utility and value of quantum computing, followed by some specific examples and use cases of quantum computing. Next, we will cover basic quantum information theory such as superposition, entanglement, teleportation and other applications of quantum computing. Afterwards, we will cover the challenges that quantum hardware is currently facing. Next, we will delve deep into quantum computer operation and control, as well as a motivation of this research. Finally, the research question and sub questions of this thesis are presented. In chapter 2, we will discuss Quantum Optimal Control theory. First we will cover the types of- and specific algorithms used in quantum optimal control. Next, we will dive into optimal gate synthesis, and cover direct, indirect, gradient-based and gradient-free methods. Finally, we will give an overview of the different resources of a quantum computer, and how we aim to build a new set of algorithms that is able to optimize these resources. In chapter 3, we will cover Energy Optimized Quantum Gate Synthesis, and the new methods and algorithms designed in order to achieve this. In chapter 4, we will cover the results of the algorithms and benchmark them against each other on several aspects and a universal set of gates. In the final chapter 5, we will discuss our results and give a conclusion of the thesis accompanied by some recommendations of future research.

1.1.1. The Promise of Quantum Computation

By using special features of quantum mechanics to our advantage in computing, a whole new range of opportunities, algorithms, and applications arise. From algorithms poised to break RSA encryption [6], to simulating the quantum dynamics of molecules to find new medicine or catalysts [7].

Essentially, researchers are looking for certain problems where there's an exponential speedup by quantum algorithms compared to classical counterparts. In Quantum Complexity Theory, these problems are part of the "Bounded-Error Quantum Polynomial Time" class of decision problems, which are solvable by a quantum computer in polynomial time, with an error probability of at most 1/3 for all instances [8], see figure 1.1 below.

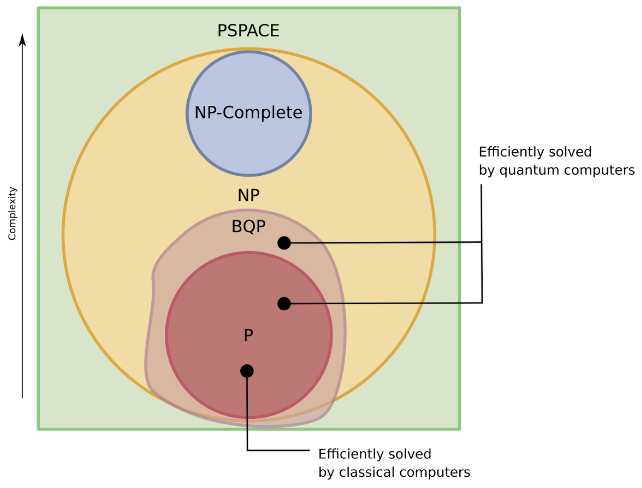

PSPACE
NP-Complete
NP
BQP
P
Complexity
Efficiently solved by quantum computers
Efficiently solved by classical computers

Figure 1.1: Overview and complexity of decision problem classes, including the Bounded-Error Quantum Polynomial Time (BQP) class [9].

Although current quantum hardware is not advanced enough for most valuable applications, there have been some very promising theoretical algorithms proposed and currently still being invented [10], such as Shor's algorithm, Grover's search algorithm, and Variational Quantum Algorithms utilizing a combination of both quantum and classical computing. In figure 2.4, an overview of the currently known applications is given [11]. As one can see, applications range across multiple different problem types and industries, hence the excitement from the industry and business perspective [11].

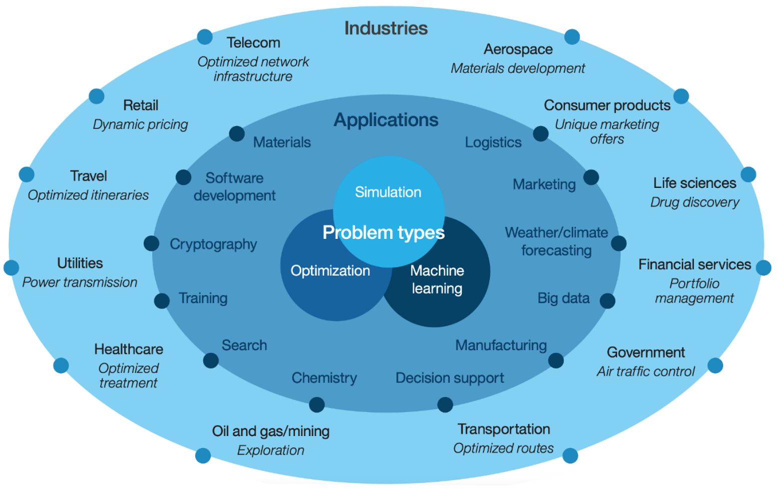


Figure 1.2: Overview of applications in Quantum Computing [11]

1.1.2. Quantum Information

In quantum information, the fundamental unit of computation is called the qubit (Quantum Bit), analogue to the bit in classical computing. In Quantum Mechanics however, the states of a quantum system are represented by state vectors, living in the complex separable Hilbert Space [8]. In reality, a qubit is engineered to be any physical two-level quantum system, for example, the spin of an electron (up- or down spin) and the polarization of a single photon (left- or right-hand circular polarized). In theory, we can describe a state $|0\rangle$ and $|1\rangle$ as:

$$|0\rangle \equiv \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \qquad |1\rangle \equiv \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{1.1}$$

Since qubits are represented as state vectors, they can be in a so-called superposition of states, which is one of the fundamental quantum mechanical phenomena that is being leveraged by quantum computing. We can thus define a single qubit $|\psi\rangle$ as [8]:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \quad \text{where} \quad |\alpha|^2 + |\beta|^2 = 1 \quad \text{and} \quad \alpha, \beta \in \mathfrak{C} \tag{1.2}$$

When qubits are measured, they return a classical bit, thus ”collapsing” the quantum superposition, and projecting its state on either the $|0\rangle$ or the $|1\rangle$ state, with probability $|\alpha|^2$ and $|\beta|^2$, respectively.

Single qubit states can be visualized on the so-called Bloch-Sphere (see figure 1.3), if we define α and β as follows in spherical-coordinates:

$$\alpha = \cos\frac{\theta}{2}, \quad \beta = e^{i\phi}\sin\frac{\theta}{2} \tag{1.3}$$

and thus,

$$|\psi\rangle = \cos\frac{\theta}{2}\,|0\rangle + e^{i\phi}\sin\frac{\theta}{2}\,|1\rangle \tag{1.4}$$

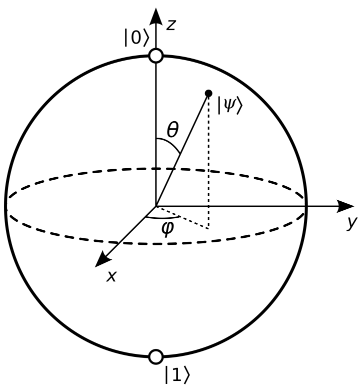


Figure 1.3: Bloch-Sphere representation of single qubit states

The states described above are all so-called ”pure states”. However, in the presence of decoherence or noise, one can put the qubit in a so-called ”mixed state”. A mixed state is a statistical combination of different pure states, essentially shrinking the radius of the Bloch Sphere vector [8]. To describe mixed states, we use the so-called density matrix formalism:

$$\rho = \begin{pmatrix} \rho_{00} & \rho_{01} \\ \rho_{10} & \rho_{11} \end{pmatrix} \tag{1.5}$$

and in the case of a pure state $|\psi\rangle$:

$$\rho = |\psi\rangle\langle\psi| = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \begin{pmatrix} \alpha^* & \beta^* \end{pmatrix} = \begin{pmatrix} |\alpha|^2 & \alpha\beta^* \\ \beta\alpha^* & |\beta|^2 \end{pmatrix} \tag{1.6}$$

The state space of qubits grows by a factor of 2^{N_q}, where N_q is the number of qubits. For instance, if we have two qubits, we can represent their combined state as:

$$|\psi_C\rangle = |\psi_A\rangle \otimes |\psi_B\rangle = \alpha_A\alpha_B|00\rangle + \alpha_A\beta_B|01\rangle + \beta_A\alpha_B|10\rangle + \beta_A\beta_B|11\rangle \tag{1.7}$$

The state above is called a separable state [8], however, not all states are separable. If a state is inseparable, it is by definition an quantum entangled state. Consider for instance the state below:

$$|\psi_C\rangle = \frac{1}{\sqrt{2}}(|0\rangle_A \otimes |1\rangle_B - |1\rangle_A \otimes |0\rangle_B) \tag{1.8}$$

There is no possible way to separate this combined state in $|\psi_A\rangle$ and $|\psi_B\rangle$, and therefore we can say that this combined state $|\psi_C\rangle$ is an entangled state [8]. The measure of entanglement between states is given by the so-called concurrence, defined as:

$$C(|\psi\rangle) = \sqrt{2(1 - \mathrm{Tr}(\rho^2))} \tag{1.9}$$

where,

$$\mathrm{Tr}(\rho) = \sum_{i=1}^{n} \rho_{ii} \quad . \tag{1.10}$$

1.2. Quantum Computer Operation and Control

In order to perform actual quantum algorithms, we need to be able to coherently control and manipulate delicate quantum states. In this section, we will first cover the theory needed in order to understand quantum dynamics and control. Thereafter, we will delve into physical methods used to control quantum states. Finally, we will discuss the up- and downsides of current quantum control methods.

1.2.1. Quantum Logic Gates and Operations

In order to manipulate quantum states, one can apply so-called Quantum Logic Gates, which are represented by $(2^n \times 2^n)$ Unitary Matrices [8]:

$$U^\dagger U = UU^\dagger = I \tag{1.11}$$

Quantum Logic Gates are applied to quantum states by basic matrix multiplication, and map the quantum state onto a new quantum state [8]:

$$U|\psi_1\rangle = |\psi_2\rangle \tag{1.12}$$

$$\begin{pmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} u_{00}\alpha + u_{10}\beta \\ u_{01}\alpha + u_{11}\beta \end{pmatrix} \tag{1.13}$$

The most well known single qubit quantum logic gates are the: Pauli-X (σ_x), Pauli-Y (σ_y), Pauli-Z (σ_z), Hadamard (H), Phase (S), T-Gate ($\pi/8$). For example, the Pauli-X gate can be used to flip the state of a qubit, and the Hadamard gate can be used to put a qubit in a maximum superposition:

$$\sigma_x|0\rangle = |1\rangle, \quad H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \tag{1.14}$$

We can also perform 2-qubit gates, which act on 2 qubits simultaneously. Important 2-qubit gates are the: Controlled-Not (CNOT, CX) Gate, Controlled-Z (CZ) Gate, SWAP Gate. For instance, the CNOT gate flips the target qubit conditionally on the state of the control qubit. If the control qubit is in the $|0\rangle$ state, it will do nothing to the target qubit state, however, if the control qubit is in the $|1\rangle$ state, it will flip the state of the target qubit, i.e.:

$$CNOT|10\rangle = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = |11\rangle \tag{1.15}$$

Next to quantum logic gates, we can also perform quantum measurements of a certain observable. For instance, if we want to measure the expected value of a certain observable $\hat{M}$ of a pure quantum state $|\psi\rangle$, this is given by [8]:

$$\langle \hat{M} \rangle = \langle \psi | \hat{M} | \psi \rangle \tag{1.16}$$

This is the inner product of the complex conjugate of our state $\langle \psi|$ with the observable $\hat{M}$ acting on our state $\hat{M}|\psi\rangle$. Oftentimes in Quantum Computing however, we measure in the computational basis, which is represented by the Pauli-Z (σ_z) operator as observable [8], which is given by:

$$\hat{\sigma}_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{1.17}$$

The probability that a measurement will yield a given result is given by the so-called Born Rule. This rule states that the probability of measuring a certain quantum state is given by the absolute squared of the probability amplitude of a certain quantum state. For instance, if we have the following quantum state $|\psi\rangle$:

$$|\psi\rangle = \sum_i p_i |\lambda_i\rangle \tag{1.18}$$

Then, the probability of measuring the state $|\lambda_i\rangle$, is given by:

$$P(|\lambda_i\rangle) = |\langle \lambda_i | \psi \rangle|^2 = |p_i|^2 \tag{1.19}$$

Quantum states in combination with quantum logic gates and measurements, allow us to construct quantum algorithms, oftentimes represented by so-called Quantum Circuits (see figure 1.4 below).

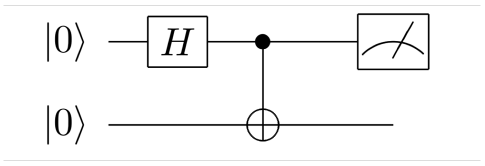


Figure 1.4: Example quantum circuit using two qubits, a Hadamard gate, and a measurement in the computational basis.

In figure 1.4 we can see a very basic Quantum Circuit. We read quantum circuits from left to right (think about time increasing from left to right). On the left hand side, we can see two lines representing our two qubits both in the $|0\rangle$ state. Afterwards, we can see a Hadamard (H) gate being applied on the first qubit, followed by a Controlled-Not (CNOT) gate with the first qubit as control and the second qubit as target. Finally we perform a measurement of the first qubit in the computational (σ_z basis).

1.2.2. Hamiltonian Engineering and Pulse Control

Quantum logic gates are constructed by carefully tuning the Hamiltonian of a quantum system over time. Quantum Systems evolve over time according to the Time-Dependent-Schrödinger-Equation (TDSE) [8]:

$$i\hbar \frac{d}{dt} |\psi(t)\rangle = \hat{H} |\psi(t)\rangle \tag{1.20}$$

Where $\hat{H}$ represents the Hamiltonian of our quantum system. A general solution for this equation is given by:

$$|\psi(t)\rangle = e^{i\hat{H}t/\hbar}|\psi(t_0)\rangle \equiv U(t)|\psi(t_0)\rangle \tag{1.21}$$

Where:

$$U(t) = e^{i\hat{H}t/\hbar} \tag{1.22}$$

As we can see, if we want to perform a specific Unitary (such as a Hadamard or CNOT), we have to apply a certain Hamiltonian on our system. Applying a specific Hamiltonian to our system to achieve a certain Unitary is accomplished through carefully applying certain pulses to our system.

We oftentimes make a distinction between the so-called "Drift" Hamiltonian, and "Control" Hamiltonian. The Drift Hamiltonian is the Hamiltonian of the actual qubit(s), usually consisting of an individual term for the eigen-energy and eigen-states of the qubits, as well as a coupling term between different qubits. The Control Hamiltonian describes the external control fields that can be applied to the qubits.

Let's consider a very simple case of a single qubit without any interaction terms. We can describe the qubit by the following Hamiltonian H_D:

$$H_D = \hbar\omega_0\hat{\sigma}_z \tag{1.23}$$

This qubit with precess about the $\hat{z}$-axis with frequency ω_0. We will also introduce a control Hamiltonian described by:

$$H_C = \hbar\omega_1\hat{\sigma}_x \tag{1.24}$$

If the control field is applied in the $\hat{x}\hat{y}$-plane, and rotates around the $\hat{z}$-axis with frequency ω_{rf}, we can rewrite our total Hamiltonian as [12]:

$$H = \hbar\omega_0\hat{\sigma}_z + \hbar\omega_1(\cos\omega_{rf}t\hat{\sigma}_x + \sin\omega_{rf}t\hat{\sigma}_y) \tag{1.25}$$

The control fields or time-dependent function of the control Hamiltonian operators are referred to as the "control pulses" that one can apply to our quantum system. A control pulse usually has three main parameters: the control field amplitude a_i, the control field frequency ω_i, and the control field phase ϕ_i. We can thus write the control field as a function of these three parameters acting on the control Hamiltonian operators ($\hat{\sigma}_i$):

$$H_k = \sum_i f(a_i, \omega_i, \phi_i)\hat{\sigma}_i, \quad \text{where,} \quad a_i, \omega_i, \phi_i = f(t) \tag{1.26}$$

Adjusting these three parameters over time is thus the definition of pulse control.

1.2.3. Pros and cons of Pulse Control

The use of electromagnetic pulses for the control of qubits has certain up- and downsides associated to it. First, we will dive into some of the pros of using electromagnetic pulses such as flexibility, precision, and fine-grained manipulation of qubits. Thereafter, we will discuss some cons associated to using electromagnetic pulses, such as sensitivity to noise, calibration requirements, and hardware limitations.

First, the benefits of using electromagnetic pulses as qubit controls are discussed [13]. Electromagnetic pulses offer a very high degree of flexibility. As we have seen in previous subsection, we can vary various parameters of the control pulse to implement very specific operations and sequences on qubit states. Furthermore, this flexibility in control over the parameters also makes it possible to achieve very high-fidelity quantum operations. This is crucial to realizing fault tolerance and quantum error correction protocols. This prevision is also what enables us to manipulate and control individual qubits without interaction with the rest of the system. All of this can be done with very high speeds. A quantum operation using electromagnetic pulses can be performed within nanoseconds, well within the decoherence time of qubits [14]. This high speed operation also allows for exploring quantum dynamics on very short timescales, which is essential for several quantum simulation algorithms.

There are also various downsides to using electromagnetic pulses as qubit control, that are often inherent to the nature and physics of the system [15]. First and foremost, electromagnetic pulses are very susceptible to noise, such as electromagnetic interference or fluctuations in the experimental setup. Noise in the electromagnetic pulse will eventually lead to decoherence and therefore errors in the quantum operation [15]. Additionally, in order to accurately implement control pulses, we often need to extensively calibrate parameters such as amplitude, frequency and phase, in order to ensure reliable qubit manipulation. Calibration can obviously be very resource intensive and time consuming, definitely with system sizes increasing. Control pulses need to be generated by classical hardware, such as arbitrary waveform generators (AWG). This classical hardware has limitations in terms of bandwidth, power, and frequency, which limits our range of parameters that we can apply to the qubits [15]. In addition to the limitations of classical hardware, this often also introduces noise due to imperfections and room temperature thermodynamic noise. Finally, electromagnetic pulses are not very adaptable to changing environments. During calibration, the control parameters for the control pulses are fixed, while in reality, the qubits are constantly changing and being affected by external noise sources. Small deviations from the calibrated parameters can lead to significant errors and performance degradation during qubit operations.

1.3. State-of-the-art Hardware and Challenges

The quantum-mechanical phenomena that qubits inherently exhibit are unfortunately also the reason why qubits are very hard to coherently control and use. Current efforts in building qubits, quantum processors, and full stack quantum computers all suffer from the same hardware limitations, noise sources, and decoherence processes. In this subsection, we will give a brief overview of the current state-of-the-art in quantum hardware. First, we will cover the concepts of the Noisy Intermediate-Scale Quantum (NISQ) era, as well as the notion of Fault Tolerance. Afterwards we will discuss the various sources of noise in quantum hardware, address the scalability issues and cover some of the recent advancements made in addressing these challenges.

1.3.1. Noisy Intermediate Scale Quantum versus Fault Tolerance

There are several important performance indicators of quantum hardware, which include, but are not limited to: the number of (physical) qubits, the decoherence time, the single- and 2-qubit error rates, the gate speed, and the qubit connectivity or architecture [16]. The decoherence time of a qubit is the measure of how long a qubit can keep its quantum information: i.e., how long it can stay in a certain quantum state without losing it's coherence to interactions with the environment [8]. Current efforts in building quantum systems are focused on having the highest number of qubits and lowest possible limiting error rates. Optimizing both, i.e., a high number of qubits with a low limiting error rate, is a very challenging and demanding task [16]. However, doing both is a requirement in order to build quantum computers capable of solving actual valuable problems. If the qubit limiting error rate is below the so-called quantum-error-correction (QEC) threshold, the logical error rate of qubits decreases if you increase the size of the system [17]. This is a requirement for reaching so-called "Fault Tolerant Quantum Computing", in which the quantum computer is completely protected from errors by QEC. Most well-known algorithms, such as Shor's algorithm for factoring prime-numbers, lie within the FTQC regime. However, people do still believe that there is an interesting region for quantum computers without error-correction, called the "Noisy Intermediate Quantum" (NISQ) regime. As the name already says, these are in general "Noisy" devices, i.e., with significantly low decoherence times and "Intermediate-Scale", i.e., from hundreds to several thousands of physical qubits [18].

While some research institutes and companies focus on getting below the error correction threshold and then purely focus on increasing the number of qubits, some research institutes and companies are focusing on decreasing the limiting error rate even more at first so we can enter this NISQ era [18]. In figure 1.5 we can see a schematic depiction of the NISQ regime and the FTQC regime as a function of the number of qubits and the limiting error rate. As we can see, the lower the limiting error-rate, the less number of qubits you need to reach fault-tolerance. This is because we need a smaller amount of physical qubits to make one logical qubit if the error rates are smaller.

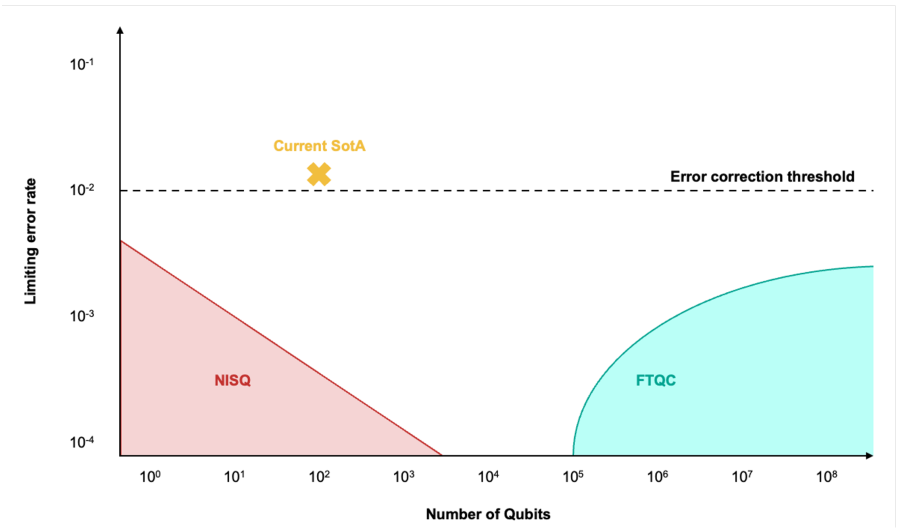


Figure 1.5: Schematic overview of NISQ and FTQC in the context of limiting error rate (either decoherence time or 2-qubit error-rate), and number of (physical) qubits. Red indicates the NISQ era, cyan the FTQC era, and the yellow cross indicates the current state-of-the-art. [19]

As one can see from the shape of the NISQ region in figure 1.5, the more number of qubits there are on a quantum processing unit, the lower limiting error rate we need in order to be in the NISQ region, where applications in the next few years will be explored. If the limiting error rate is too big compared to the amount of qubits, without quantum error correction, one cannot perform any useful calculations anymore, thus leaving the NISQ region [19]. Only when a certain amount of qubits is reached below the error correction threshold, one can reach the fault tolerance regime.

1.3.2. Challenges in Quantum Hardware

In this subsection, we will cover the various sources of noise in quantum hardware and address the scalability issues related to increasing the number of qubits while maintaining low error rates. We will also cover some of the advancements made in addressing these challenges such as error correction codes, and quantum error correction techniques. While we have come a long way since the theoretical description of a quantum computer, we are still far away from having a fault tolerant quantum computing machine. The most prominent challenge that all qubit types are facing is decoherence and noise [15]. Decoherence arises from uncontrolled interactions with the environment the qubits are in. This can originate from noise in our classical control electronics like electromagnetic fluctuations or magnetic field gradients, but also from interactions with two-level systems inside the bulk material [15]. The decoherence and noise is directly correlated to the quality of the materials and fabrication techniques that are used. Current research has shown the importance of well-defined interfaces and material purity on qubit coherence time. Another challenge that quantum hardware is facing is the scalability issue [15]. In order to run useful computations we need a large number of qubits. Scaling up this number of qubits without increasing cross-talk and increasing the error rates poses a significant challenge. In addition, achieving long-range connectivity between qubits via qubit interconnects or electro-optical modulators is a very big challenge. Unwanted interactions between qubits are known as cross talk. This cross talk can lead to unwanted interactions and correlations between qubits, causing errors in the computation. To address the decoherence issue, there is many ongoing research currently being done. The first possible solution could be developing materials and interfaces with reduced loss [20]. Additionally, people are exploring methods such as dynamic decoupling, where qubits are decoupled from dephasing effects from the environment by continuously flipping the qubit's state. These methods are all intended to increase the decoherence time of the qubits, and therefore lower, or mitigate the noise. Opposing the error mitigation strategy, there is also lots of research being done on so-called quantum error correction (QEC), that focuses on developing error correction schemes that are able to detect and correct when errors happen [17]. To address the scalability challenge, researchers are focusing on developing scalable qubit architectures and inter qubit connectivity methods [21]. Additionally, people are also interested in using multiplexed control and readout schemes to reduce the number of control lines needed to address and manipulate the qubits. Techniques to overcome the cross-talk challenges include improving the shielding and isolation methods of the qubits, as well as using custom control sequences to minimize unwanted interactions with other qubits [15].

1.4. Research Question

In this section, we will cover the research question of this work. We will first dive into an overview of the current methods and research that has been done in the field of Quantum Optimal Control. We will highlight different methods, techniques and cover their strengths and limitations in terms of scalability, convergence speed, and applicability to various quantum systems. Next, we will cover the relevance of energy efficiency in quantum computations and why energy efficient control of qubits is relevant. Finally we will discuss how and what optimal control methods we intend to investigate, and provide an outline of the subsequent chapters.

1.4.1. Overview of Current Methods

Future quantum systems are becoming increasingly complex and demanding, and achieving fine-grained control of qubits has become a big challenge in advancing quantum technologies. The collection of techniques focused on achieving high quality control of quantum systems is often referred to as "Quantum Optimal Control Theory", or QOCT. In formal terms, QOCT involves methods to design and implement electromagnetic field configurations that can effectively steer quantum processes at the atomic or molecular scale in the best way possible [1].
We can broadly categorize two main method categories in Quantum Optimal Control: open-loop and closed-loop methods. Open-loop methods rely on using an existing theoretical model of the quantum system and process in question, and accordingly design optimal control pulses based on that theoretical model. Contrarily, closed-loop methods don't require an existing theoretical model of the system, and use experimental feedback loops, with actual measurement data from the quantum system to devise and refine the control pulses. The obvious pitfall of open-loop methods is that they rely on an accurate theoretical representation of the quantum system, while in reality there are always more parameters and factors involved. Closed-loop methods are only dependent on measurement feedback and the quality of those measurements, and also struggle with the rapid timescales of quantum dynamics.

Previous research has explored both open- and closed-loop methods for Quantum Optimal Control Theory. This work ranges from pure analytical methods, such as perturbation expansions, to completely model-free closed-loop methods such as reinforcement learning.[1] provides an extensive overview of all the work that has been done in the field of Quantum Optimal Control Theory.
Besides open-loop and closed-loop methods, there is another distinction to make between analytical techniques and numerical techniques. While analytical approaches introduce less uncertainty and computation effort, they are not scalable to larger quantum systems. When introducing multiple qubit dynamics and noise models, analytical methods quickly become impossible to solve. Numerical methods rely on quantizing the problem into time steps, and using either gradient-based or gradient-free methods to numerically solve the problem at hand. These methods are more scalable to larger systems and are better capable of handling noise models. One of the most used gradient-based numerical methods known today in Quantum Optimal Control is the so-called "Gradient Ascent Pulse Engineering" [2] method, or GRAPE in short.

In recent years, we have also seen growing interest in using Machine Learning techniques to address Quantum Optimal Control problems, in particular one specific branch of machine learning called reinforcement learning (RL). This approach resembles closed loop methods: an agent takes actions to maximize a final reward. If we represent the actions by control pulses, and final reward by final fidelity, we can see how closed loop optimal control methods can be modelled in the reinforcement learning framework. [1].

1.4.2. Relevance of Energy Efficient Quantum Optimal Control

The field of Quantum Thermodynamics has seen a lot of work in recent years, and there is a growing interest in the possibility of achieving quantum advantage through energy efficiency instead of computational power [22]. While Quantum Thermodynamics is seeing an increase in research, little is still known about the energetic cost of a quantum computational process. In [4] they prove an inequality bounding the change of Shannon information encoded in the logical quantum states by quantifying the energetic cost of Hamiltonian gate operations. Subsequently, in [23], they show that optimal control problems ca be solved within the powerful framework on quantum speed limits, and derive state-independent lower bounds on energetic cost. Recent work in Quantum Optimal Control Theory has primarily focused on developing control to carry out quantum processes with the highest fidelity possible. These processes include quantum processes such as state initialization, quantum measurements, and implementing quantum unitary gates. However, in the context of the growing interest in achieving quantum advantage through energy efficiency [22], it seems crucial to investigate the energy efficiency of quantum operations, in particular, unitary quantum gates. Additionally, applying Quantum Optimal Control methods to discover optimal control parameters for energy-efficient quantum unitary gates, while maintaining or bounding fidelity and speed, is of great importance. The impact of optimal energy-efficient quantum unitary gates could stimulate implementations on physical quantum hardware, and provide valuable insights into the energy usage of future quantum computing systems.

1.4.3. Research Question and Overview of Next Chapters

As the relevance of energy efficiency in quantum unitary gates is clear, and as we know that we can potentially use Quantum Optimal Control techniques to reach this goal, we have identified two main research question, and some sub questions for this research:

Research Question 1: *"What is the energetic cost of implementing a quantum unitary gate, and what is the relation between fidelity and the energetic cost?"*

Sub Question 1.1: *"How can we quantify the energetic cost of implementing a quantum unitary gate and relate it to the optimal control pulses?"*

Sub Question 1.2: *"Is there are trade-off between the fidelity and the energetic cost of implementing a quantum unitary gate?"*

Research Question 2: *"What Quantum Optimal Control strategies can we utilize to investigate and co-optimize a quantum unitary gate on both fidelity and energetic cost?"*

Sub Question 2.1: *"How can we modify a gradient-based open-loop Quantum Optimal Control method to co-optimize both fidelity and energetic cost?"*

Sub Question 2.2: *"How can we use a learning based, model-free, closed-loop method to co-optimize both fidelity and energetic cost? "*

Sub Question 2.3: *"How well do both Quantum Optimal Control techniques perform in the presence of noise?"*

In chapter 2, a brief overview of Quantum Optimal Control theory will be presented. The system levels of a quantum computer, and in what part of the quantum computing stack we are operating in will be discussed. Several existing classes and types of algorithms will be presented, narrowing down our focus on Quantum Unitary Gate synthesis. Subsequently, the difference between direct and indirect methods, gradient-based and gradient-free methods, and open-loop versus closed-loop methods are explored. Finally, the different resources of a quantum computation, accompanied by an overview of the existing software packages and tools that we will use in this research are presented. Chapter 3 covers the methods that are used in this research, and how existing algorithms are modified to devise energy-efficient universal quantum unitary gates. Both the cost function and the Pareto optimal front with respect to fidelity, as well as how to implement a quantum simulator and measurement feedback to train our reinforcement learning agent are presented afterwards. In chapter 4 the results for the universal set of gates, for all the methods explored in this work are presented and discussed, as well as the correlation between different ways to describe the energetic cost of quantum unitary gates. In the final chapter 5, the thesis will be concluded and ideas for future work are proposed.

2
Quantum Optimal Control

Everything should be as simple as possible, but no simpler
- Albert Einstein

In this chapter, we will cover Quantum Optimal Control. We will first provide an overview of the Quantum Computing system levels, to create a better understanding of the abstraction levels of a quantum computer, and where Quantum Optimal Control fits in. Subsequently, we will give an overview of the different areas and allied topics of Quantum Optimal Control, as well as an overview of the most well-known algorithms and types of algorithms. Afterwards, we will motivate why we have chosen the Gradient Ascent Pulse Engineering method, and provide a more in-depth review of this specific algorithm. After this, we will cover some of the Quantum Computing Resources, which can be used as inputs to the cost function to be optimized. Finally, we will provide a more concrete overview of the different tools and methods used in this research, such as the programming language, existing software packages, and algorithms.

2.1. The Quantum Computing Stack

In order to successfully run algorithms on qubits, we need to build several essential layers of both hardware and software. The combination of these layers of hardware and software, is often referred to as the Quantum Computing Stack [24].

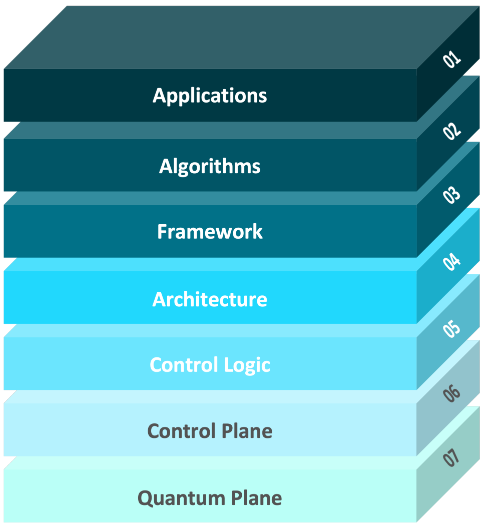


Figure 2.1: Overview of the different layers of Hardware and Software required for Quantum Computing. [25]

In figure 2.1, we can see the various hardware and software layers in the Quantum Computing Stack. The top layer can be regarded as the actual application of the quantum algorithm that one wants to run. Next, we have the algorithm layer, that represents the actual quantum circuit that one needs to run for the specific application. The framework layer can be regarded as a classical compiler. It is able to do quantum gate decomposition (i.e., decomposing gates into the native gate set of a specific type of quantum computer), circuit design, and mapping. Next we have the Architecture layer, or the Quantum Instruction Set Architecture, or QISA in short. The Architecture defines what physical instructions are possible in a specific quantum computer, such as measurement, initialization, and also quantum unitary gates. The next layer is the control logic layer, where the pulse calibration, optimization and also decoding happens. The actual electromagnetic pulses are then addressed to specific qubits through the control plane to mitigate heat and cross talk. Finally the signals arrive at the quantum plane, where the actual qubits reside. The qubits can be based on different technologies, such as superconducting qubits or spin qubits (quantum dots).

As we can see from the figure, the pulses are generated in the Control Logic Layer. However, the actual optimization of a certain pulse to achieve a certain gate with highest precision possible is happening in the compiler stage [24]. In figure 2.2, we can see a more detailed depiction of the quantum compiler stage, where the pulse optimization is highlighted in red.

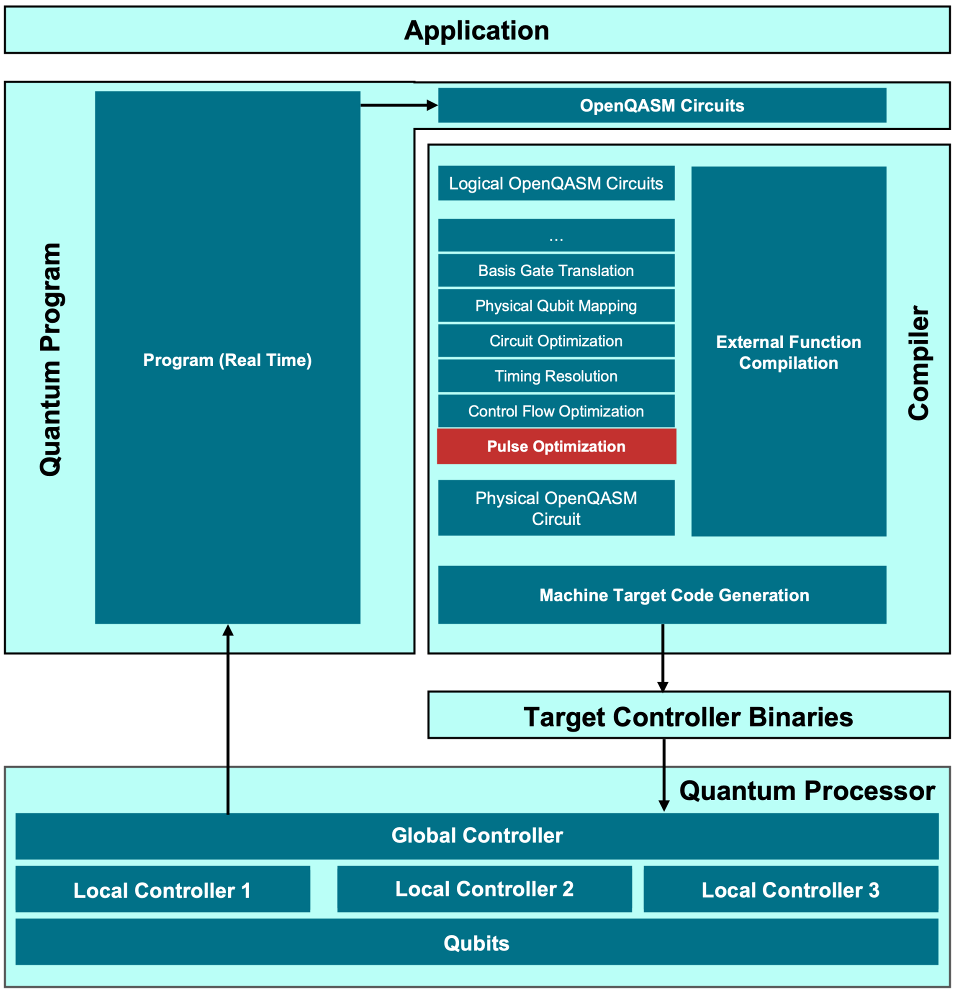


Figure 2.2: Schematic overview of the quantum programming paradigm and quantum compiler layer.

2.2. Overview of Existing Algorithms

There exist many different methods and algorithms to achieve quantum optimal control. In this section, we will give a brief overview of the different classifications we can make between all existing methods, and give an overview of the most well-known algorithms. For a schematic representation of all the methods we will discuss below, please refer to table 2.1.

The first classification that we can make is the difference between analytical and numerical methods [26]. As the name already suggests, the analytical methods use mathematical theory and representations of the quantum system to analytically solve for the optimal pulse. On the other hand, numerical methods leverage the power of discretization and linearization to allow the use of numerical methods and algorithms. The most well known analytical methods are discussed first.

The first example of an analytical method for quantum optimal control, is the so-called Pontryagin's Maximum Principle [27]. It states that any optimal control together with the optimal state trajectory is a 2-point boundary value problem with a maximum condition of the control Hamiltonian. Using this description, one can use a time-varying Lagrangian description and multiplier vector to solve the problem. Another method to analytically devise optimal control pulses is through generalization of adiabatic evolutions. An example of this is the so-called "Derivative Removal by Adiabatic Gates" method, or DRAG in short. If the system is more complex, such as multiple qubits, interactions, or noise systems, we often need to make a perturbative expansion to solve it analytically [28]. Finally, we can use SU(2) Lie Algebra to devise rules on when a quantum system is fully reachable, or controllable [29].

Table 2.1: Overview of some existing methods of Quantum Optimal Control (non exhaustive), including their category based on analytical, numerical, closed loop, open loop, gradient based, and gradient free methods.

QOC Method	**Analytical**	**Numerical**	**Closed Loop**	**Open Loop**	**Gradient Based**	**Gradient Free**
Pontryagin's maximum principle	X			X	X	
DRAG	X			X		X
Perturbative expansion	X			X	X	
SU(2) Lie Algebra	X			X		X
Reinforcement learning		X	X			
Q-Learning		X	X			
msMS-DE		X	X			
Sampling based learning control		X	X			
s-GRAPE		X	X			
b-GRAPE		X	X			
Krotov		X		X	X	
GOAT		X		X	X	
GRAPE		X		X	X	
CRAB		X		X		X
GA		X		X		X
DE		X		X		X

For the numerical methods, we have a few more classifications we can make. The first one being, open-loop and closed-loop methods. Open-loop methods use a model of the system to devise optimal pulses that are in return directly applied to the system, and give a certain output. Closed-loop methods on the other hand use feedback of the system to adjust the pulses accordingly. Let us first discuss some of the most well-known closed loop methods.

The first, and possibly the most popular method, is a sub-class of Machine Learning, called Reinforcement Learning. In Reinforcement Learning, a so-called 'Agent' is allowed to take certain 'Actions', and apply it to the 'Environment'. The environment then outputs a certain 'State', accompanied by a certain 'Reward', based on the action that it took. In quantum optimal control, the 'Action' is the control pulse, the 'Environment' is the quantum system, and the state is the output state of the quantum system after applying that specific control pulse [3]. For a schematic overview please refer to figure 2.3.

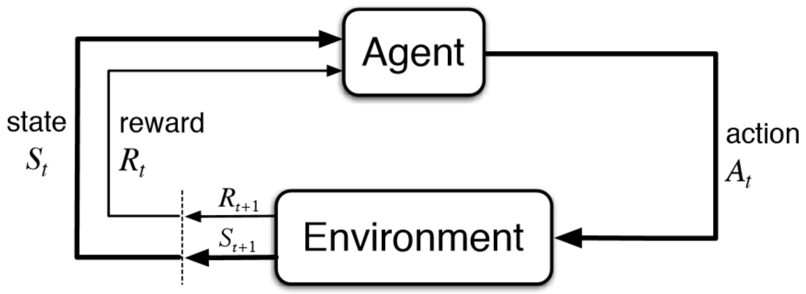


Figure 2.3: Schematic overview of the basic workings of a Reinforcement Learning agent [30]

In the case of a model-free reinforcement learning algorithm, this policy can either be updated by the use of a Neural Network, or by a method called Q-Learning [31]. If we need to increase the robustness in the feedback loop, we could potentially use the Hessian matrix information combined with the closed loop learning based algorithm, called the msMS-DE algorithm [26]. We can also use samples of a quantum system for training purposes, and then evaluate the performance with a test and evaluation phase. This is called sampling based learning control, or SLC in short [32]. Finally, we have hybrid methods that use the gradient-based GRAPE algorithm in combination with reinforcement learning or other machine learning methods, such as s-GRAPE or b-GRAPE [26].

If we look at open loop methods, we can make a distinction between gradient-based, and gradient-free methods. Gradient-based methods rely on calculation local gradients to in return move towards a local optimum, while gradient free methods usually use some form of stochastic search algorithms to reach an optimum [26]. Let us first look at gradient-based methods.

The first method we will discuss is the so-called Krotov method [33]. This method utilizes Lagrange multipliers based on the process fidelity to find optimal pulses for gate synthesis and state-to-state transfer. If representing the control pulse in an analytical form is preferred, we may use the so-called Gradient Optimization of Analytical Controls, or GOAT method [34]. This method uses an educated guess of the shape of the pulse (e.g., Gaussian), to form a coupled system of equations, which can then be solved numerically by forward integration methods, such as the Runge-Kutta method [34]. Potentially the most well-known and widely used method is the so-called Gradient Ascent Pulse Engineering, or GRAPE method [2]. This method uses the discretization of the control operators to iteratively solve for the optimal control pulse. In section 2.3, we will cover the GRAPE algorithm in more detail.

The first gradient free method we will discuss is the Chopped Random Basis Optimization, or CRAB method [35]. This method leverages the fact that optimal solutions could reside in a low dimensional subspace of the total search space. The control sequences are represented as a linear addition of basis functions. Another gradient free method is based on an evolutionary process, called the Genetic Algorithm, or GA algorithm [36]. This method utilizes a set of initial random guesses, and then evolves these methods based on a fitness function. A variation of this method is the so-called Differential Evolution, or DE method [37].

In this thesis, we want to use both an open-loop and a closed-loop method to assess whether it is possible to devise energy efficient optimal control pulses. To this end, we have chosen the Gradient Ascent Pulse Engineering method as the open-loop method, and Deep Reinforcement Learning as the closed loop method. The motivation of this choice will be discussed in the dedicated sections of both methods.

2.2.1. Allied topics

There exist many applications and use cases for quantum optimal control. In figure 2.4, we can see an overview of the most well-known applications, ranging from hardware tailored quantum optimal control, to algorithms and circuit compilation. In this section, we will give a brief overview of all applications and use cases. In chapter 2.2.2 we will motivate why quantum gate synthesis was chosen as a focus area.

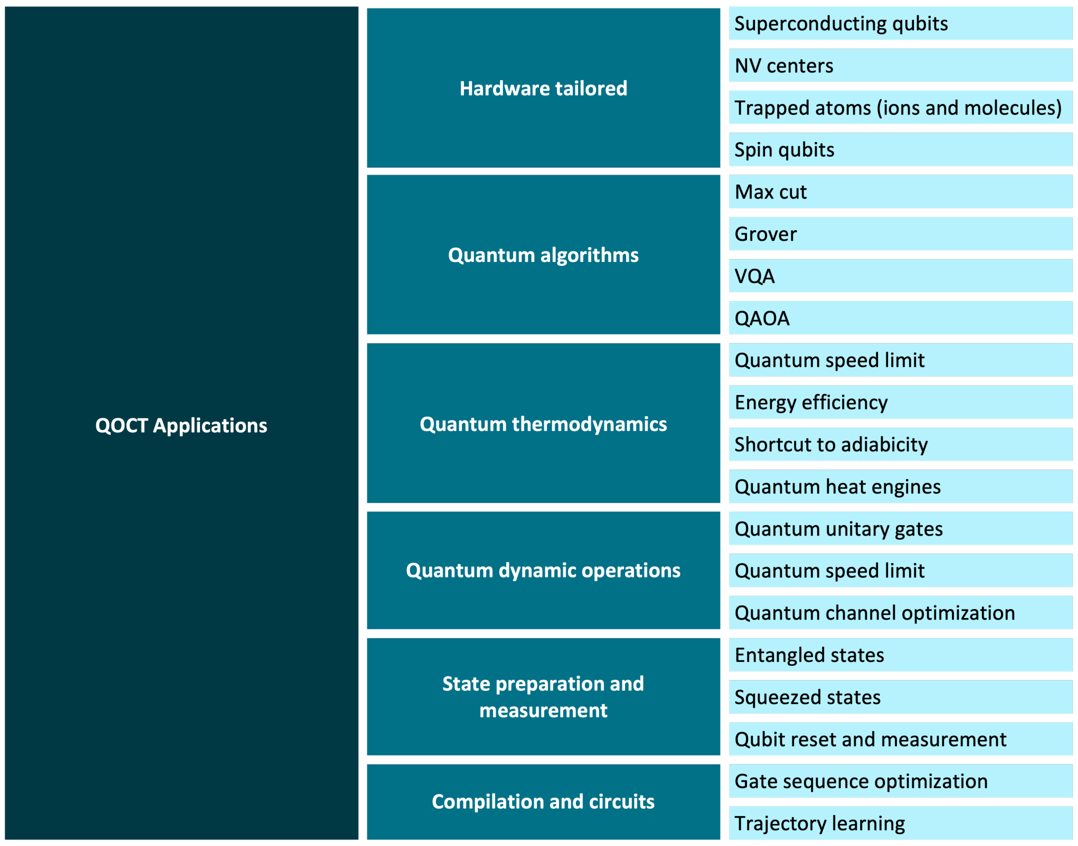


Figure 2.4: Schematic overview of some existing applications of Quantum Optimal Control (non exhaustive)

The first use-case category is Hardware tailored quantum optimal control. These are applications of various quantum optimal control methods, adjusted and specified for a certain qubit technology. A well-known method for superconducting qubits is the DRAG pulse for weakly coupled and harmonic transmon qubits [38]. NV centers are well suited for a wide range of quantum technologies, such as computing, communication, but also sensing. We have seen optimal controlled quantum sensing protocols for NV centers in diamond [39]. Next to superconducting qubits, trapped ions are one of the most mature technologies to date, but suffer from scalability and gate speed issues. Regarding optimal control, we have seen the application of QOC for improving the dissipative preparation of entangled states [40]. Finally, we have so-called spin-qubits, which use the 2-level systems created by electron spins in magnetic fields. For spin-qubits, we have seen quantum optimal control being used for state preparation [41].

The second application category we will cover is quantum algorithms. Here, quantum optimal control techniques are used to optimize the algorithms at a very fine granularity. There are overlapping and similar ideas being used mostly in variational quantum algorithms compared to closed loop optimal control methods. Firstly, QOCT has been used for landscape analysis in the well known optimization problem MaxCut [42]. Secondly, the well known Grover search algorithm can actually be transformed into a optimal control problem, and solved through the analytical method called the Pontryagin's Maximum Principle [43]. Furthermore, quantum optimal control techniques can be used to run variational algorithms, such as VQA [44] and QAOA [45].

The third use-case category is Quantum thermodynamics. Firstly, we can use quantum optimal control methods to find theoretical frameworks for both the quantum speed limit, and energy efficiency [46] [4]. Secondly, work has been done on shortcuts to adiabicity and using quantum control to optimize the operation cycle of quantum heat engines [47] [48].

The next big category is quantum dynamic operations. The first example of this is perhaps the most well known and well studied application of quantum optimal control theory, namely quantum unitary gates. A typical problem in quantum optimal control is to devise pulses that perform a certain quantum unitary gate with the highest precision possible, based on the quantum system at hand [49]. Finding quantum unitary gates at the quantum speed limit is also an application of this [46]. Finally, we have seen the optimization of quantum channels, where quantum information is encoded in a pulse shape of a single photonic qubit, and the readout and driving pulses are devised by quantum optimal control techniques [50].

The fifth category of quantum optimal control is state preparation and measurement, a very important component of any quantum computation. To achieve high fidelity quantum circuits, we need to be able to prepare qubits in certain states, and read out their state with high fidelity. We have seen quantum optimal control being used for the generation and preparation of both entangled states [51], as well as optical squeezed states [52]. Finally, an important and popular use case of quantum optimal control is the resetting and measurement of qubits with the highest possible fidelity [53].

The sixth category is quantum compilation and circuits. Quantum optimal control has been used to optimize gate sequences [54], as well as trajectory learning, to relate the parameter values of a quantum circuit, to the control space, which will give rise to a continuous class of gates.

2.2.2. Gate Synthesis

In this work, we will focus on what the energetic cost is of a quantum unitary gate, and what the trade-off between fidelity and energetic cost is in a quantum unitary gate. We will also look at what quantum optimal control strategies we can utilize to co-optimize a quantum unitary gate on both fidelity and energetic cost. This in essence is a quantum gate synthesis problem.

In a quantum gate synthesis problem, you are given a target unitary operator U_T, corresponding to the quantum gate that you want to achieve. The goal of the quantum gate synthesis problem is then to find a control sequence Ω, that, after a certain time T, implements a unitary gate $U(T)$, with the highest gate fidelity possible. The gate fidelity here is defined as:

$$F_G(U_T, U(T)) = \left| \frac{\langle U_T | U(T) \rangle}{\langle U_T | U_T \rangle} \right|^2 \tag{2.1}$$

Where $\langle U_T | U(T) \rangle$ represents the overlap between two operators, and is defined as the trace between the product of the complex conjugate of U_T and $U(T)$:

$$\langle U_T | U(T) \rangle = \mathrm{Tr}(U_T^\dagger U(T)) \tag{2.2}$$

As we can see, the control sequence that we obtain will be independent of the initial state $|\psi\rangle$, and thus implement a quantum unitary gate.

2.3. Gradient Ascent Pulse Engineering

In this section we will in detail introduce the Gradient Ascent Pulse Engineering (GRAPE) algorithm for quantum unitary gate synthesis [2]. Before we discuss the actual derivation and workings of the algorithm, we first need to introduce how we will define our quantum system, and what variables we will use. We will define and refer to the total time T, discretized into N equal steps of duration $\Delta t = T/N$. During each time step j, the control amplitudes u_k are constant, i.e., during the j^{th} time step, the amplitude of the k^{th} control Hamiltonian is given by $u_k(j)$.

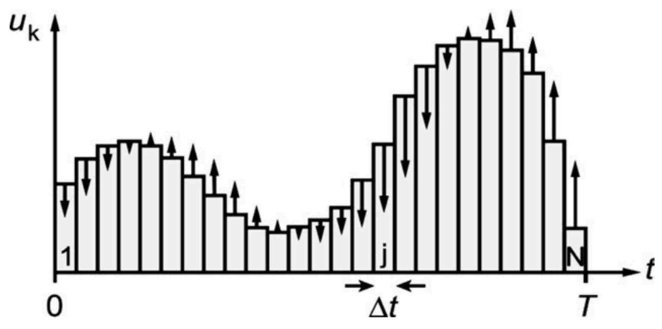


Figure 2.5: Example control parameter $u_k(t)$, consisting of N time steps of duration $\Delta t = T/N$. The vertical arrows indicate the gradient, which represents how the amplitude at each time step should by modified in the next iteration in order to maximize the performance function Φ [2]

The total Hamiltonian of our closed quantum system consists of a constant drift Hamiltonian H_d, and a sum of control Hamiltonians H_k, with amplitude $u_k(t)$. We can thus write our total system Hamiltonian as:

$$\mathcal{H} = H_d + \sum_{k=1}^{m} u_k(t) H_k \tag{2.3}$$

The dynamics of a state $|\psi(t)\rangle$, represented as a vector in a Hilbert space evolves according to the time-dependent Schrödinger equation

$$i\hbar \frac{\partial}{\partial t} |\psi(t)\rangle = \mathcal{H}(t) |\psi(t)\rangle \tag{2.4}$$

Or the Liouville-von Neumann equation if we define the density matrix $\rho(t) = |\psi(t)\rangle \langle\psi(t)|$

$$i\hbar \frac{\partial}{\partial t} \rho(t) = [\mathcal{H}(t), \rho(t)] \tag{2.5}$$

The time-evolution of a quantum state during a time step j is then given by the Unitary operator:

$$U_j = \exp\left\{-i\Delta t \left(H_d + \sum_{k=1}^{m} u_k(j) H_k \right)\right\} \tag{2.6}$$

And the total Unitary after time T is given by:

$$U(t = T) = \prod_{j=0}^{N} U_j = \prod_{j=0}^{N} \exp\left\{-i\Delta t \left(H_d + \sum_{k=1}^{m} u_k(j) H_k \right)\right\} \tag{2.7}$$

To understand the GRAPE algorithm, we also need to introduce the concept of the so-called forward, and backward propagators.

For a specific time slice, the forward and backward propagators are the rest of the unitary after and before this time slice respectively, such that the total unitary is constructed from their composition.

The forward propagator X_j is therefore given by:

$$X_j \equiv U_j \dots U_1 \tag{2.8}$$

Accordingly, the backward propagator P_j can be defined as:

$$P_j \equiv U_{j+1}^\dagger \dots U_N^\dagger \tag{2.9}$$

We then need to define our so-called performance function, that we will need to maximize. In quantum unitary gate synthesis, we can define this performance function as the overlap between the target unitary U_T and the final unitary after time T:

$$\Phi = |\langle U_T | U(T) \rangle|^2 \tag{2.10}$$

The basic idea of the GRAPE algorithm, is to continuously update the control parameters u_k according to the gradient of the performance function Φ with respect to the control parameters u_k. This will be repeated for a certain amount of iterations, or GRAPE-iterations, given by N_G.

We can mathematically define this update rule as follows, using an arbitrary step size ϵ [2]:

$$u_k(j) \rightarrow u_k(j) + \epsilon \frac{\partial \Phi}{\partial u_k(j)} \tag{2.11}$$

Now that we have defined our performance function and updating rule, we can define the actual GRAPE algorithm [2]:

Algorithm 1 Gradient Ascent Pulse Engineering

```
1. Guess initial $u_k(j)$
while $g \leq N_G$ do
    2. Calculate $X_j$, $\forall j \leq N$
    3. Calculate $P_j$, $\forall j \leq N$
    4. Evaluate $\partial\Phi/\partial u_k(j)$ and update $m \times N$ control amplitudes $u_k(j)$ according to equation 2.11
    5. $g = g + 1$
end while
```

The starting initial guess of the GRAPE algorithm can be zero, random, or an educated guess, which might lead to faster convergence.

To evaluate the gradient, we have to derive the derivative of the performance function with respect to the control parameters $u_k(j)$. We can rewrite our performance function in terms of the forward and backward propagator as:

$$\Phi = \langle U_F | U_N \dots U_1 \rangle \langle U_1 \dots U_N | U_F \rangle = \langle P_j | X_j \rangle \langle X_j | P_j \rangle \tag{2.12}$$

By using perturbation theory [2] to the first order in $\partial u_k(j)$, we arrive at:

$$\frac{\partial \Phi}{\partial u_k(j)} = -\langle P_j | X_j \rangle \langle i\Delta t H_k X_j | P_j \rangle - \langle P_j | i\Delta t H_k X_j \rangle \langle X_j | P_j \rangle = -2\mathcal{R}e \left\{ \langle P_j | i\Delta t H_k X_j \rangle \langle X_j | P_j \rangle \right\} \tag{2.13}$$

By using the definition of the gradient, the forward and backward operator, and by updating the control parameters according to equation 2.11, we can arrive at global maxima of the performance function for a specific set of control parameters $u_k(j)$.

2.4. Overview of Quantum Computing Resources

In this section, we will briefly review the various resources of a quantum computation, that could potentially be optimized by quantum optimal control techniques. The three main resources of a quantum computation are: fidelity, energy and time. In the next three subsections, we will introduce how we mathematically and theoretically define these three resources, and how they affect a quantum computation.

2.4.1. Fidelity

Fidelity is a widely used term that applies to many things in quantum computing. In essence it refers to the accuracy of a certain quantum operation or algorithm. Examples include: initialization fidelity, single qubit gate fidelity, 2 qubit gate fidelity, unitary gate fidelity, read-out fidelity, entanglement fidelity and algorithm fidelity. In this research, we will focus on the unitary gate fidelity, to measure the overlap between the target unitary gate, and the final unitary gate after pulse optimization. The unitary gate fidelity between the target unitary U_T and the final unitary $U(T)$ after a certain time T, is given by [8]:

$$F(U_T, U(T)) = \left| \frac{\mathrm{Tr}(U_T^\dagger U(T))}{\mathrm{Tr}(U_T^\dagger U_T)} \right|^2 \tag{2.14}$$

As we can see, if U_T and $U(T)$ are identical, the Fidelity will by 1, or 100 %. If $U(T)$ is completely orthogonal to U_T, the Fidelity will be 0, or 0%. The goal of any unitary gate synthesis quantum optimal control method is thus to maximize this Fidelity as much as possible (as close to one).

2.4.2. Energy

The energy of a quantum state is defined as the expectation value of the Hamiltonian:

$$\langle E \rangle = \langle \psi | \hat{H} | \psi \rangle \tag{2.15}$$

However, the energetic cost of an actual unitary operation is harder to define. In [4], they define the energetic cost as the time integrated norm of the Hamiltonian, over the total duration of the unitary gate:

$$E[U] = \int_0^\tau dt \|\mathcal{H}(t)\| \tag{2.16}$$

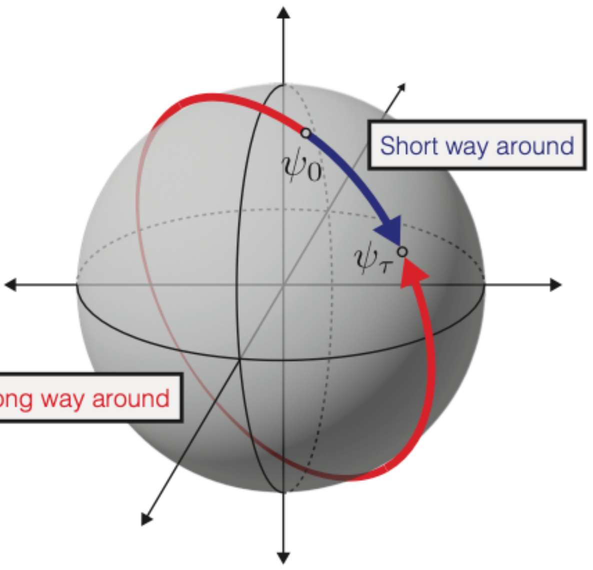


Figure 2.6: Visual interpretation of an energy efficient quantum unitary gate. The two paths both accomplish the same rotation, but one path length is much longer than the other. [4]

In the case of single qubit quantum unitary gates, we can visually interpret it on the Bloch Sphere. The most energy-efficient way to implement a single unitary gate, is given by the geodesic between the two quantum states on the Bloch sphere, i.e., the shorter the path length between the two states, the lower the energetic cost of the quantum unitary gate. In figure 2.6, we can see a visual interpretation of this by the short and long path length between two quantum states on the Bloch sphere. The blue unitary gate in this case has a lower energetic cost than the unitary gate that implements the red path.

In this research, we will investigate both equation 2.16, as well as the path length representation, for computing the energetic cost of quantum unitary gate. Although one cannot visualize multi-qubit Unitary gates on the Bloch sphere, similar geometric arguments hold here as well.

2.4.3. Time

The final resource of a quantum computation is Time. In our case, we are specifically interested in the time it takes to implement a quantum unitary gate. This will be closely related to the energetic cost of the quantum unitary gate, and as we choose a fixed time T for all quantum unitary gates, we leave the Time resources out of scope for this research.

Nevertheless, there is an actual theoretical lower bound on the minimal time a quantum unitary gate will take, which is the so-called ”Quantum Speed Limit”, or QSL in short. The QSL has been bounded by two inequalities: the Mandelstamm-Tamm inequality, and the Margolus-Levitin inequality. The Mandelstamm-Tam inequality states that the minimal time to reach an orthogonal state is given by [4]:

$$\tau \geq \frac{\hbar\pi}{2\Delta E} \tag{2.17}$$

Where ΔE is the standard deviation in the energy of the initial quantum state. The Margolus-Levitin inequality states that the independent bound on the time it takes to reach an orthogonal state is given by:

$$\tau \geq \frac{\hbar\pi}{2(\langle E\rangle - E_0)} \tag{2.18}$$

Here, $\langle E\rangle$ is the expectation value of the energy, defined by equation 2.15, and E_0 is the so-called ground state energy, or the lowest energetic state possible of the quantum state.

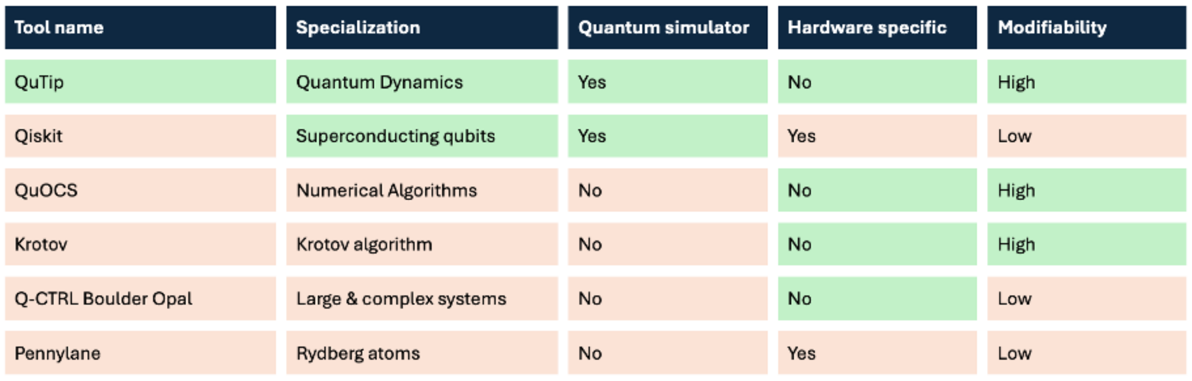
Since the two bounds can be tight, we usually write the quantum speed limit as the maximum of either bound:

$$\tau_{QSL} \equiv \max\left\{\frac{\hbar\pi}{2\Delta E}, \frac{\hbar\pi}{2(\langle E\rangle - E_0)}\right\} \tag{2.19}$$

2.5. Overview of existing tools and methods

Several quantum technology python libraries and packages have dedicated classes and functionalities for and related to quantum optimal control. In this section, we will give a brief overview of some of these existing tools, and will elaborate on what platform we will use during this research.

When selecting what python library to use and build on, we have a couple criteria in mind. Firstly, it has to be hardware agnostic, as we are not focusing on any specific qubit type, but are rather interested in a more general approach. Secondly, the python library should ideally include a quantum simulator class, as we need to have measurement feedback for the closed loop method. Thirdly, it should be easily modifiable, and completely open-source. And lastly, it should ideally not specialize on any specific algorithm, use case, or method.
From our initial exploration, we have identified 6 python libraries that either focus completely on quantum optimal control, or have quantum optimal control features. These libraries include: Xanadu Pennylane [55], Q-CTRL Boulder Opal [56], Krotov [57], QuOCS [58], IBM Qiskit [59], and QuTip [60]. In figure 2.7, we provide an overview of these packages, including how they score on the four selecting criteria.



Tool name	Specialization	Quantum simulator	Hardware specific	Modifiability
QuTip	Quantum Dynamics	Yes	No	High
Qiskit	Superconducting qubits	Yes	Yes	Low
QuOCS	Numerical Algorithms	No	No	High
Krotov	Krotov algorithm	No	No	High
Q-CTRL Boulder Opal	Large & complex systems	No	No	Low
Pennylane	Rydberg atoms	No	Yes	Low

Figure 2.7: Overview of existing python libraries for quantum optimal control, including selected criteria

As one can see from the results of the survey, QuTip [60] scores highest, and therefore we chose to use this package as inspiration of the research. QuTip uses native classes, functions, and other objects that are not very modifiable and integratable with other packages, we therefore chose to re-create our own classes and functions only using NumPy [61] and SciPy [62]. This allows us to make every adjustment possible and have full control over input, output, and inner workings of all functions, classes, and algorithms. For the quantum simulator, the QuTip Processor module is used, which acts as an emulator of a quantum device. This quantum simulator module was used because of its ability to send discrete pulse level signals to individual qubits, for arbitrary control Hamiltonian operators, which is a hard requirement for this research.

For the closed-loop Reinforcement Learning method that we will explore, we will use Google's Keras and TensorFlow framework [63], which is an end-to-end platform for Machine Learning in Python.

Now that an overview of the existing Quantum Optimal Control techniques and applications is given, and a motivation on why certain techniques and applications were chosen, we can dive into Energy Optimal Quantum Gate Synthesis. In the next chapter, the adjustments and additions to the existing GRAPE algorithm will be provided, as well as some theory on reinforcement learning. Afterwards, the implementation of this theory in python will be discussed.

3 Energy Optimal Quantum Gate Synthesis

The task is not to see what has never been seen before,
but to think what has never been thought before
- Erwin Schrödinger

In this chapter, we will introduce the new additions and adjustments that we have made to the Gradient Ascent Pulse Engineering method, as well as using deep reinforcement learning to devise energy optimized control pulses for quantum gate synthesis. We will first cover the theoretical framework by introducing the cost function that we need to optimize, followed by the derivation and expression for the gradient needed to implement the GRAPE algorithm. Finally we will also cover some theoretical concepts of reinforcement learning. After introducing the theoretical framework, we will cover the implementation of these concepts in python by introducing the python package that we have created for this work. We will first give an overview of the code architecture and classes, followed by deep dives into the three main classes of the code, called the Quantum Environment Class, the EO-GRAPE Class, and the QRLA Class.

3.1. Theoretical Framework

In this section we will introduce the theoretical framework and mathematical definitions for energy optimized quantum gate synthesis. Before we address the cost function, we first need to introduce some variables, parameters, and definitions.

Firstly, we will refer to *control pulses* as the set of control parameters that define the amplitude of a control Hamiltonian H_k at each time step j. A certain control parameter $u_k(j)$ is therefore the value of control Hamiltonian H_k for the j^{th} time step. For the theoretical framework, we will assume that the control pulses $u_k(t)$ can be any function of amplitude $a_k(t)$, angular frequency $\omega_k(t)$, and phase $\phi_k(t)$:

$$\vec{u}_k(t) = f\begin{pmatrix} a_k(t) \\ \omega_k(t) \\ \phi_k(t) \end{pmatrix} \tag{3.1}$$

Potentially due to hardware restrictions, one can often not choose any value for the control pulses. Therefore, we will refer to $\mathcal{A}$ as the set of control pulses that are *physically allowed*, or in other words *admissible*:

$$\mathcal{A} = \{\vec{u}_k(t)|\text{admissible protocols}\} \tag{3.2}$$

Put differently, a control pulse is *admissible* if and only if $\vec{u}_k(t) \in \mathcal{A}$.

We will refer to a *cost function* Φ as the function that has to be minimized in the optimization problem. The cost function in our case is a so-called *multi objective cost function*, and therefore is a weighted linear addition of independent cost functions ϕ_i:

$$\Phi = \sum_i w_i \phi_i \tag{3.3}$$

The goal of the optimization problem is thus to find a set of control pulses $\vec{u}_k(t) \in \mathcal{A}$ that minimize the cost function Φ.

3.1.1. Cost function

As stated in equation 3.3, in this research we have chosen to define our cost function as a weighted linear addition of two independent performance metrics, since there are no specific initial arguments to be made as of why this would not be a weighted linear addition. This allows us to take into account both performance metrics, and easily tune the weights to give priority to either fidelity or energetic cost. In our research, the cost function is a weighted linear addition of both the inverse process fidelity, and the energetic cost of a quantum unitary gate:

$$\Phi = w_f \phi_f + w_e \phi_e \tag{3.4}$$

We have defined the process fidelity between a target unitary U_T and the final unitary after time $U(T)$ as:

$$F(U_T, U(T)) = \left| \frac{\mathrm{Tr}(U_T^\dagger U(T))}{\mathrm{Tr}(U_T^\dagger U_T)} \right|^2 \tag{3.5}$$

As the cost function should be minimized, we have defined ϕ_f as the inverse of the process fidelity:

$$\phi_f = 1 - F(U_T, U(T)) \tag{3.6}$$

According to [4], we can define the energetic cost of a certain quantum unitary gate as:

$$C[U(T)] = \int_0^T dt \, ||\mathcal{H}|| \tag{3.7}$$

Where we can define the total system Hamiltonian $\mathcal{H}$ as:

$$\mathcal{H} = H_d + \sum_{k=1}^{m} u_k(t) H_k \tag{3.8}$$

As the time is discretized into N equidistant time steps δt, we can rewrite the Hamiltonian as:

$$\mathcal{H} = \sum_{j=0}^{T} \delta t \left(H_d + \sum_{k=1}^{m} u_k(j) H_k \right) \tag{3.9}$$

By combining equation 3.7 with equation 3.9, we get the following expression for the energetic cost of a quantum unitary gate:

$$C[U(T)] = \sum_{j=0}^{T} \delta t \left|\left| H_d + \sum_{k=1}^{m} u_k(j) H_k \right|\right| \tag{3.10}$$

As the control parameters $u_k(j) \in [-1, +1]$, we arrive at the following expression for the normalized energetic cost of a quantum unitary gate, which is equivalent to the performance metric ϕ_e:

$$\hat{C}[U(T)] = \frac{\sum_{j=0}^{T} \delta t \, ||H_d + \sum_{k=1}^{m} u_k(j) H_k||}{T \, ||H_d + \sum_{k=1}^{m} H_k||} \equiv \phi_e \tag{3.11}$$

The normalization constant will be referred to as:

$$N_e = T \left\| H_d + \sum_{k=1}^{m} H_k \right\| \tag{3.12}$$

The total cost function in this research is the weighted linear addition of ϕ_f and ϕ_e, given by:

$$\Phi = w_f \left(1 - \left| \frac{\mathrm{Tr}(U_T^{\dagger} U(T))}{\mathrm{Tr}(U_T^{\dagger} U_T)} \right|^2 \right) + w_e \left(\frac{1}{N_e} \sum_{j=0}^{T} \delta t \left\| H_d + \sum_{k=1}^{m} u_k(j) H_k \right\| \right) \tag{3.13}$$

3.1.2. Gradient

In order for the cost function to be used in the GRAPE algorithm, the gradient of the cost function with respect to the control pulses $u_k(j)$ has to be derived.

The gradient of Φ with respect to $u_k(j)$ can be written as:

$$\frac{\partial \Phi}{\partial u_k(j)} = w_f \frac{\partial \phi_f}{\partial u_k(j)} + w_e \frac{\partial \phi_e}{\partial u_k(j)} \tag{3.14}$$

From chapter 2.3, we know that the first part reduces to the following expression:

$$\frac{\partial \phi_f}{\partial u_k(j)} = -2\mathcal{R}e \left\{ \langle P_j | i \Delta t H_k X_j \rangle \langle X_j | P_j \rangle \right\} \tag{3.15}$$

The partial derivative of the energetic cost part of the cost function ϕ_e with respect to the control parameters $u_k(j)$ thus remains to be evaluated. The energetic cost part of the cost function can be written as:

$$\phi_e = \frac{1}{N_e} \sum_{j=0}^{T} \delta t \left\| H_d + \sum_{k=1}^{m} u_k(j) H_k \right\| \tag{3.16}$$

Using the identity $||A|| = \sqrt{\mathrm{Tr}\,(A^* A)}$, the expression above can be expanded as follows:

$$\phi_e = \frac{1}{N_e} \sum_{j=0}^{T} \delta t \sqrt{ \mathrm{Tr} \left(H_d^* H_d + \sum_{k=1}^{m} u_k(j)\,(H_d^* H_k + H_k^* H_d) + \sum_{k=1}^{m} \sum_{k'=1}^{m} u_k(j) u_{k'}(j) H_k^* H_{k'} \right) } \tag{3.17}$$

To compute the gradient, the expression can be separated into two parts:

$$\phi_e = f\,(g\,(u_k(j))) = \begin{cases} f\,(g\,(u_k(j))) = \frac{1}{N_e} \sum_{j=0}^{T} \delta t \sqrt{g\,(u_k(j))} \\ g\,(u_k(j)) = \mathrm{Tr}\,(\ldots) \end{cases} \tag{3.18}$$

The gradient can then be expressed in terms of $f(g(u_k(j)))$ and $g(u_k(j))$:

$$\frac{\partial \phi_e}{\partial u_k(j)} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial u_k(j)} \tag{3.19}$$

The first derivative is given by:

$$\frac{\partial f}{\partial g} = \frac{1}{N_e} \sum_{j=0}^{T} \delta t \frac{1}{2\sqrt{g\,(u_k(j))}} \tag{3.20}$$

The second derivative is given by:

$$\frac{\partial g}{\partial u_k(j)} = \mathrm{Tr}\left(\sum_{k=1}^{m} H_d^* H_k + H_k^* H_d\right) + \mathrm{Tr}\left(\sum_{k=1}^{m}\sum_{k'=1}^{m} H_k^* H_{k'} \left(u_k(j) + u_{k'}(j)\right)\right) \tag{3.21}$$

The total gradient of ϕ_e with respect to $u_k(j)$ can thus be written as:

$$\frac{\partial \phi_e}{\partial u_k(j)} = \frac{1}{Ne}\sum_{j=0}^{T} \delta t \frac{\mathrm{Tr}\left(\sum_{k=1}^{m} H_d^* H_k + H_k^* H_d\right) + \mathrm{Tr}\left(\sum_{k=1}^{m}\sum_{k'=1}^{m} H_k^* H_{k'} \left(u_k(j) + u_{k'}(j)\right)\right)}{2\sqrt{\mathrm{Tr}\left(H_d^* H_d + \sum_{k=1}^{m} u_k(j)\left(H_d^* H_k + H_k^* H_d\right) + \sum_{k=1}^{m}\sum_{k'=1}^{m} u_k(j) u_{k'}(j) H_k^* H_{k'}\right)}} \tag{3.22}$$

Combining equation 3.15 with equation 3.22, gives us the total gradient of Φ with respect to the control parameters $u_k(j)$:

$$\frac{\partial \Phi}{\partial u_k(j)} = \begin{pmatrix} -2w_f \\ -w_e \end{pmatrix}\begin{pmatrix} \mathcal{R}e\left\{\langle P_j | i\Delta t H_k X_j\rangle\langle X_j | P_j\rangle\right\} \\ \frac{1}{Ne}\sum_{j=0}^{T} \delta t \frac{\mathrm{Tr}\left(\sum_{k=1}^{m} H_d^* H_k + H_k^* H_d\right) + \mathrm{Tr}\left(\sum_{k=1}^{m}\sum_{k'=1}^{m} H_k^* H_{k'}\left(u_k(j) + u_{k'}(j)\right)\right)}{2\sqrt{\mathrm{Tr}\left(H_d^* H_d + \sum_{k=1}^{m} u_k(j)\left(H_d^* H_k + H_k^* H_d\right) + \sum_{k=1}^{m}\sum_{k'=1}^{m} u_k(j) u_{k'}(j) H_k^* H_{k'}\right)}} \end{pmatrix} \tag{3.23}$$

The expression given by equation 3.23 will be used as the gradient to update the parameters in the GRAPE algorithm, which will be explained in the next subsection.

3.1.3. EO-GRAPE Algorithm

In this subsection, the adjustments and additions to the Gradient Ascent Pulse Engineering algorithm will be introduced. This updated algorithm is referred to as the **E**nergy **O**ptimized **Gr**adient **A**scent **P**ulse **E**ngineering algorithm, or *EO-GRAPE*. At it's core, EO-GRAPE works the same as the original GRAPE algorithm, explained in section 2.3. With the EO-GRAPE algorithm however, a novel updating rule and gradient is used:

$$u_k(j) \rightarrow \epsilon_f \frac{\partial \phi_f}{\partial u_k(j)} + \epsilon_e \frac{\partial \phi_e}{\partial u_k(j)} \tag{3.24}$$

where,

$$\frac{\partial \phi_f}{\partial u_k(j)} = -2w_f \mathcal{R}e\left\{\langle P_j | i\Delta t H_k X_j\rangle\langle X_j | P_j\rangle\right\} \tag{3.25}$$

and,

$$\frac{\partial \phi_e}{\partial u_k(j)} = -\frac{w_e}{Ne}\sum_{j=0}^{T} \delta t \frac{\mathrm{Tr}\left(\sum_{k=1}^{m} H_d^* H_k + H_k^* H_d\right) + \mathrm{Tr}\left(\sum_{k=1}^{m}\sum_{k'=1}^{m} H_k^* H_{k'} \left(u_k(j) + u_{k'}(j)\right)\right)}{2\sqrt{\mathrm{Tr}\left(H_d^* H_d + \sum_{k=1}^{m} u_k(j)\left(H_d^* H_k + H_k^* H_d\right) + \sum_{k=1}^{m}\sum_{k'=1}^{m} u_k(j) u_{k'}(j) H_k^* H_{k'}\right)}} \tag{3.26}$$

As one can see, the expression above is quite complex and requires one to subdivide it into pieces in order to implement it in an algorithm. The code to evaluate the expression contains three for-loops that first loops over all time steps, followed by a loop over k, and subsequently k'. For each iteration, the numerator is calculated first, followed by the denominator. Afterwards, they are divided, and the trace and square root are taken. Finally, the expression is normalized by the normalization factor N_e, and added to the total cost function. This is done for each control Hamiltonian H_k and time step j. In figure 3.1, a schematic drawing of the EO-GRAPE algorithm is given. On the left hand side, the input parameters of the EO-GRAPE algorithm are displayed, including the static (or drift) Hamiltonian H_d, the time-dependent control Hamiltonian H_k, the target quantum unitary gate U_T, and optionally initial values of the control parameters u_0. On the right hand side, the different steps of the algorithm are given. The first step is the dynamic evolution, followed by calculating the cost function. Afterwards, the gradient specified in equation 3.23 is evaluated, and the control parameters are updated according to equation 3.24. After a certain threshold is met, or a number of iterations has been achieved, the algorithm will stop and output the final control parameters.

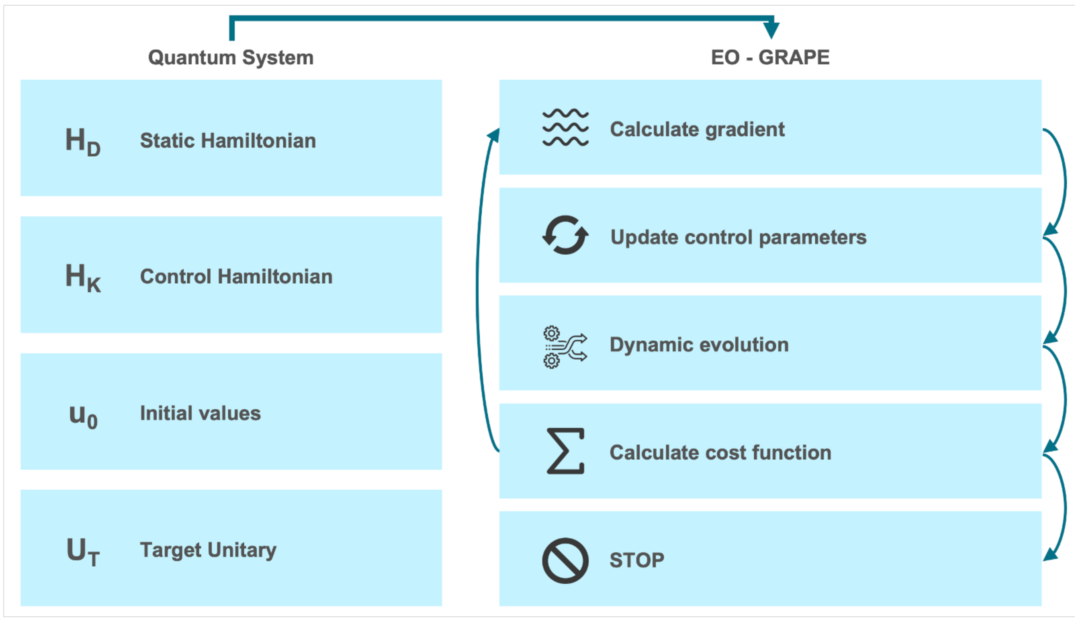


Figure 3.1: Schematic of the EO-GRAPE algorithm, including the parameters of the Quantum System, as well as the different steps in the EO-GRAPE algorithm.

The EO-GRAPE algorithm can thus be formulated as algorithm 2, displayed below.

Algorithm 2 Energy Optimized Gradient Ascent Pulse Engineering

```
1. Guess initial $u_k(j)$
while $g \leq N_G$ do
    2. Calculate $X_j$, $\forall j \leq N$
    3. Calculate $P_j$, $\forall j \leq N$
    4. Evaluate $\partial \phi_f / \partial u_k(j)$
    5. Evaluate $\partial \phi_e / \partial u_k(j)$
    6. Update $m \times N$ control amplitudes $u_k(j)$ according to equation 3.24
    7. $g = g + 1$
end while
```

3.1.4. Reinforcement Learning

In this subsection, the theoretical framework of the reinforcement learning algorithms we have used in this work will be explained. In section 2.2, a more introductory overview of basic reinforcement learning concepts is provided. In this research, two different reinforcement learning agents are utilized. Reinforcement Learning Agent 1, or *RLA-1* in short, interacts with the quantum environment, and is responsible for actually devising the pulses. Reinforcement Learning Agent 2, or *RLA-2* in short, is responsible for approximating pulses generated by the EO-GRAPE algorithm, potentially to be used as a initial policy of *RLA-1*. *RLA-1* will be introduced first, followed by *RLA-2*.

Both Reinforcement Learning agents use the so-called TensorFlow REINFORCE algorithm [63]. First, the agent observes the state that the environment returns, and takes an action based on the policy. The agent sends it's action to the environment and receives a reward based on it's action. The actions and rewards are continuously being observed and registered in the replay buffer. After a certain amount of actions, the policy is updated. How the policy is updated is based on the use of a Deep Feed Forward Neural Network. In figure 3.2, an overview of the different components and steps in the REINFORCE algorithm is shown [64].

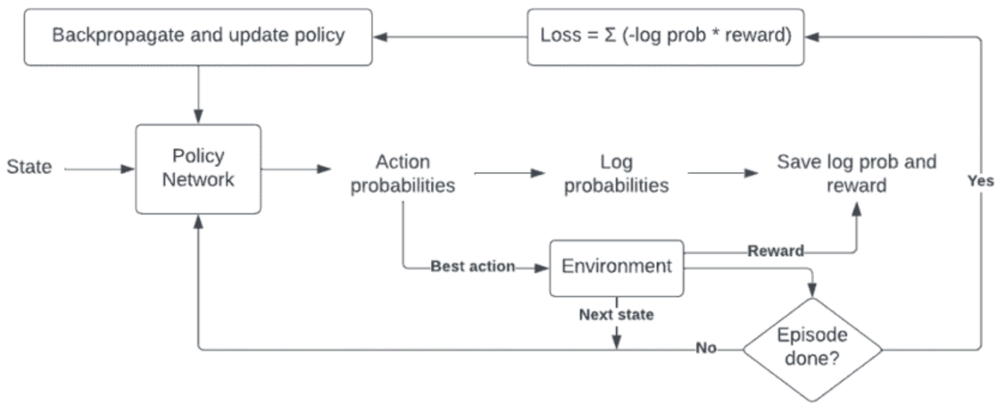


Figure 3.2: Schematic drawing of the basic workings of the TensorFlow REINFORCE agent [64]

With *RLA-1*, the environment is a quantum simulator from QuTip, called the QuTip Processor class [60]. The environment has several input parameters, such as: the number of qubits N_q, the Drift Hamiltonian H_d, the control Hamiltonians H_k, and the decoherence times of each qubits T_1 and T_2. The agent is allowed to take an action, which in this case are the actual control pulses $u_k(j)$, with shape $(\mathrm{len}(H_k) \times \mathrm{len}(t))$. The output of the environment is called the "state", which in this case is the output density matrix ρ_{out} of the quantum system after applying the action. The agent makes a next decision based on the state and the reward, which in our case is defined as:

$$r_{RLA-1} = w_f F(\rho_T, \rho_{out}) + w_e(1 - \hat{C}[U(T)]) \tag{3.27}$$

As one can see, the reward is again a linear weighted addition of both the fidelity between the target output density matrix ρ_T and the output density matrix after applying the action ρ_{out}, and the inverse of the normalized energetic cost of implementing a certain Unitary after time T, $\hat{C}[U(T)]$.

The agent takes decisions based on it's policy function $\pi(A_t, S_t)$, which is constantly being updated by a neural network. With *RLA-1*, a Deep Feed Forward Neural Network with 1 input layer, 3 to 5 hidden layers, and 1 output layer is utilized. In figure 3.3, one can see a schematic of a Deep Feed Forward Neural Network, where we can see the input layer, the hidden layers, and the output layers.

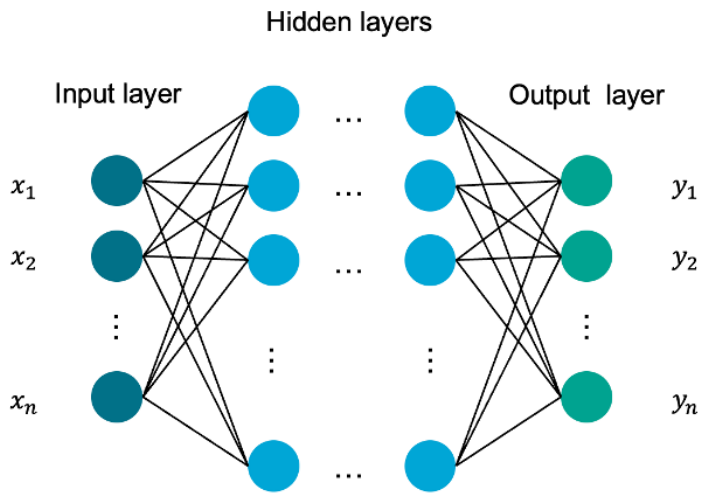


Figure 3.3: Schematic illustration of a Deep Feed Forward Neural Network

The goal of *RLA-2* is to mimic the pulses generated by the EO-GRAPE algorithm, by minimizing the distance between the control pulses of the RL agent, and the control pulses generated by the EO-GRAPE algorithm. The Environment the agent is interacting with is a custom class called the ”GRAPEApproximation” class, that takes in a set of control pulses, and outputs the theoretical Unitary. The environment has input parameters such as the number of qubits N_q, the Drift Hamiltonian H_d, the control Hamiltonians H_k, the number of time steps N_t, the total time T, and the number of EO-GRAPE iterations N_G. The agent is allowed to take an action, which again are the actual control pulses $u_k(j)$, with shape $(\mathrm{len}(H_k) \times \mathrm{len}(t))$. The state that the environment returns to the agent is the theoretical unitary that the control pulses will implement, based on unitary evolution $U(u_k(j))$. The agent will learn based on the reward that the environment returns, which in the case of *RLA-2*, is the distance between the target pulse generated by the EO-GRAPE algorithm, and the action that the agent takes:

$$r_{RLA-2} = -\sum_{j=0}^{T} \left| u_k^{EO-GRAPE}(j) - u_k^{RLA-2}(j) \right|^2 \tag{3.28}$$

For *RLA-2*, again a Deep Feed Forward Neural Network is utilized, with 1 input layer, 3 hidden layers, and 1 output layer, as shown in figure 3.3.

3.2. Code Architecture

In this section, the code architecture, classes, dependencies, and parameters will be discussed. First, an overview of the different classes and functions is provided. Afterwards, each class is discussed in detail, including smaller details like input and output parameters, individual functions, data types, and performance. The software is publicly available as a GitHub repository via the following link: `https://github.com/QML-Group/EO-QCtrl`.

3.2.1. Overview

To investigate our two main research questions, a comprehensive python package was created, called ”Energy Efficient Universal Quantum Optimal Control”, or ”EUQOC” in short. EUQOC contains four main classes, with each having a different objective and dependencies. In this sub-section, we will introduce the four classes and their main functionalities, as well as how they work together to solve energy optimal quantum gate synthesis problems.

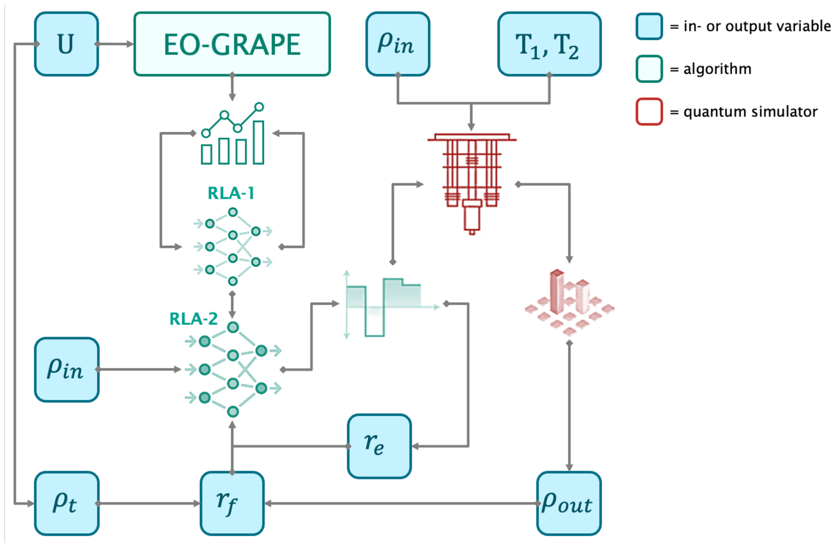


Figure 3.4: A schematic overview of the different classes and how they are co-dependent on each other. The red icons indicate quantum simulators, the blue icons indicate input and output variables of the algorithms or simulators, and the green icons indicate an algorithm.

In figure 3.4, a schematic overview of the four classes working together on a quantum optimal control problem is provided for reference. There are two main quantum environment classes, and two main reinforcement learning classes.

The first quantum environment class is called ***QuantumEnvironment***, which creates an instance of a QuTip processor [60]. As input parameters it takes in the number of qubits N_q, the Drift Hamiltonian H_d, the control Hamiltonians H_k, the decoherence time(s) of the qubit(s) T_1 and T_2, the target quantum unitary gate U_T, the weights associated to fidelity and energy w_f and w_e, the number of time steps N_t, the total gate duration T, and the number of EO-GRAPE iterations N_G. It contains functions to run custom pulses on the simulator, or to run the EO-GRAPE algorithm to calculate optimal control pulses based on the input parameters. It also contains many plotting functions to visualize the control pulses and results of the simulation.

The second quantum environment class is called ***GRAPEApproximation***, which has the sole purpose of interacting with *RLA-2*, to approximate and learn pulses calculated by the EO-GRAPE algorithm. As input parameters, it takes in the number of qubits N_q, the Drift Hamiltonian H_d, the control Hamiltonians H_k, the target unitary U_T, the number of time steps N_t, the total gate duration T, and the number of EO-GRAPE iterations N_G. It contains functions to perform dynamic time evolution of control pulses to calculate the theoretical final quantum unitary gate it implements.

The first reinforcement learning agent class is called ***QuantumRLAgent***, which contains the code to create an instance of *RLA-1*. As input values, it needs a Training and Evaluation environment, which are *QuantumEnvironment* instances, the weights associated to fidelity and energy w_f and w_e, and other reinforcement learning hyper-parameters such as the Neural Network Layers and size, the number of training iterations, the learning rate, etc. When called, the class creates a reinforcement learning agent instance, with options to run training, plot training results, or save and load weight and policy settings.

The second reinforcement learning agent class is called ***GRAPEQRLAgent***, which contains the code to create an instance of *RLA-2*. As input values, it also needs a Training and Evaluation environment, which are *GRAPEApproximation* instances, and other reinforcement learning related hyper-parameters such as the number of training iterations, the learning rate, etc. The agent trains on EO-GRAPE generated pulses, and tries to minimize the distance between the EO-GRAPE generated pulse and the action the agent takes. The policy of the agent is then used to map onto *RLA-1*. The class includes functions to run the training session, to plot training results, and most importantly to save and load the weights of the neurons or policy.

3.2.2. Quantum Environment Class

The Quantum Environment Class is used to create custom instances of a QuTip processor, as well as run the EO-GRAPE algorithm on and plot results. Next to this, the Quantum Environment Class can be used to interact with the Quantum Reinforcement Learning Agent to train and evaluate results on. The Quantum Environment Class therefore serves as the basis for any Energy Optimal Quantum Gate Synthesis experiment. The Quantum Environment Class takes in several attributes, or initial parameters. An overview of the attributes of the Quantum Environment Class is given in table 3.1.

Table 3.1: Overview of the attributes of the Quantum Environment Class, including name and the description of the attributes.

Attribute name	**Description**
n_q	Number of qubits
h_drift	Drift Hamiltonian
h_control	Control Hamiltonian operators
t_1	Relaxation time
t_2	Decoherence time
u_target	Target quantum unitary gate
timesteps	Number of time steps
pulse_duration	Total time duration of the pulse
grape_iterations	Number of EO-GRAPE iterations
w_f	Weight associated to fidelity in the cost function
w_e	Weight associated to energetic cost in the cost function
sweep_noise	Static noise or dynamically increasing noise

The Quantum Environment Class also contains a variety of methods, or functions. Some methods are only intended for interacting with the Quantum Reinforcement Learning Agent, such as the $\_reset$ and $\_step$ methods. Next to this, there exist some methods to calculate fidelity (rewards) and energetic cost (rewards), as well as running the EO-GRAPE algorithm on the attributes specified in calling the class. Finally, there exist many methods for visualizing and plotting results of the experiments, ranging from plotting the gradient as a function of iterations, to plotting the unitary trajectory on the surface of the Bloch sphere. A complete overview of the methods contained in the Quantum Environment Class is given in table 3.2. For the actual code implementing all of the methods described in table 3.2, please refer to the Appendix .

Table 3.2: Overview of the methods of the Quantum Environment Class. Including the method name, description, and what the method returns.

Method name	**Description**	**Returns**
create_environment	Creates instance of QuTip Processor with given attributes	QuTip processor instance
_reset	Resets the quantum system to the initial state	ts.restart instance
_step	Updates the environment according to the given action	ts.transition instance
run_pulses	Run custom set of pulses on environment and return result	Output density matrix
calculate_fidelity_reward	Calculates and returns the fidelity reward of a set of control pulses	Output density matrix, fidelity reward
calculate_energetic_cost	Calculates and returns the energetic cost reward of a set of control pulses	Energetic cost
run_grape_optimization	Runs EO-GRAPE algorithm based on given attributes and returns control pulses	Final control pulses
get_total_arc_length	Calculates the total arc length of the quantum unitary gate on the Bloch Sphere	Arc length
plot_grape_pulses	Plots the pulses generated by the EO-GRAPE algorithm	none
plot_rl_pulses	Plots the pulses generated by a Reinforcement Learning agent	none
plot_tomography	Plots the quantum state tomography of a quantum unitary	none
plot_du	Plots the gradient as a function of EO-GRAPE iterations	none
plot_cost_function	Plots the cost function as a function of EO-GRAPE iterations	none
plot_bloch_sphere_trajectory	Plots the trajectory of the quantum unitary gate on the Bloch sphere	none

3.2.3. EO-GRAPE Approximation Class

The EO-GRAPE Approximation Class is a quantum environment type class intended to interact with the EO-GRAPE Quantum Reinforcement Learning Class, to learn EO-GRAPE pulses and transfer the policy onto the Neural Network of the Quantum Reinforcement Learning Class. The class mainly contains methods for interacting with a reinforcement learning agent, as well as the EO-GRAPE algorithm, and some plotting functions. In table 3.3, an overview of the attributes of the EO-GRAPE Approximation class is provided.

Table 3.3: Overview of the attributes in the EO-GRAPE Approximation class, including the attribute name and description of each attribute.

Attribute name	**Description**
n_q	The number of qubits
h_drift	The Drift Hamiltonian
h_control	The Control Hamiltonian operators
u_target	The target quantum unitary gate
w_f	The weight associated to the fidelity part of the cost function
w_e	The weight associated to the energetic cost part of the cost function
timesteps	The number of time steps
pulse_duration	The total time duration of the control pulse
grape_iterations	The number of EO-GRAPE iterations

The EO-GRAPE Approximation Class contains several methods to interact with a TensorFlow reinforcement learning agent, as well as the EO-GRAPE algorithm to calculate the target/training pulse to approximate. Finally, it contains methods to calculate the square difference between the action and the target pulse as specified by equation 3.28, and methods to perform time evolution of a set of control pulses, given by method *find_sq_diff* and *calc_unitary_and_reward*, respectively. An overview of the methods in the EO-GRAPE Approximation class is provided in table 3.4.

Table 3.4: Overview of the methods of the EO-GRAPE Approximation Class. Including the method name, description and what the method returns

Method name	Description	Returns
_reset	Resets the quantum environment to the initial state	ts.restart instance
_step	Updates the environment according the action	ts.transition instance
run_grape_optimization	Runs the EO-GRAPE algorithm and returns the final control parameters	Final control pulses
calculate_energetic_cost	Calculates the energetic cost of the action	Energetic cost
calc_unitary_and_reward	Performs time evolution of the control pulses and returns the reward	Reward
find_sq_diff	Calculates the square difference between the action and the target pulse	Square difference
plot_grape_pulses	Plots the control pulses	none

3.2.4. Quantum Reinforcement Learning Class

The Quantum Reinforcement Learning Class uses the TensorFlow REINFORCE [63] Reinforcement Learning Agent Framework to interact with a Quantum Environment Class and learn energy optimal quantum gate synthesis. The class requires a training environment and an evaluation environment for training and evaluation purposes. The two environments should be Quantum Environments as specified in section 3.2.2, which contains all the attributes that define the quantum system parameters. Next to the two environments, the Quantum Reinforcement Learning class has several hyper parameter options such as the number of training iterations, the neural network layer sizes, the learning rate, and the replay buffer capacity. Finally, one can provide some optional settings such as an initial policy, whether to increase the noise level during training, and whether to have a low or high noise environment. An overview of the attributes are provided in table 3.5

Table 3.5: Overview of the attributes in the Quantum Reinforcement Learning class, including the attribute name and description of each attribute.

Attribute name	Description
TrainEnvironment	An instance of a Quantum Environment for training purposes
EvaluationEnvironment	An instance of a Quantum Environment for evaluation purposes
num_iterations	The number of training loop iterations
fc_layer_params	The number of nodes per hidden neural network layer
learning_rate	The learning rate of the reinforcement learning agent
replay_buffer_capacity	The size of the replay buffer of the reinforcement learning agent
policy	Optional: the initial policy of the reinforcement learning agent
sweep_noise	Optional: whether or not to increase the noise level during training
noise_level	Optional: low or high noise level

The Quantum Reinforcement Learning class contains several methods. Two methods called *create_network_agent* and *run_training* have the purpose of creating an instance of a TensorFlow REINFORCE agent [63], and running the training cycle of the agent on the given Quantum Environment. Next to these methods, there are several plotting functions to visualize the training and evaluation cycle of the agent. Finally, we have two methods to either save the trained weights of the neural network, or visualize the neural network, called *save_weights* and *show_summary*, respectively. A full overview of the methods contained in the Quantum Reinforcement Learning class is provided in table 3.6.

Table 3.6: Overview of the methods of the Quantum Reinforcement Learning class. Including the method name, description and what the method returns.

Method name	Description	Returns
create_network_agent	Creates an instance of a TensorFlow REINFORCE agent	TensorFlow REINFORCE agent
run_training	Runs the training loop of the reinforcement learning agent	none
get_final_pulse	Returns the final pulse of the reinforcement learning agent	Control pulses
get_highest_fidelity_pulse	Returns the highest fidelity pulse of the reinforcement learning agent	Control pulses
save_weights	Saves the weights of the nodes in the neural network	Neural network weights
show_summary	Prints a summary of the neural network after training	none
plot_fidelity_return_per_episode	Plots the fidelity and return of the agent during training	none
plot_fidelity_energy_reward_per_iteration	Plots the fidelity and energetic cost reward per iteration during training loop	none
plot_final_pulse	Plots the final pulse given by the reinforcement learning agent	none

3.2.5. EO-GRAPE Quantum Reinforcement Learning Class

The EO-GRAPE Quantum Reinforcement Learning class contains attributes and methods to create an instance of a TensorFlow REINFORCE agent that interacts with EO-GRAPE Approximation Classes, as described in section 3.2.3. The sole purpose of the agent is to approximate the control pulses generated by the EO-GRAPE algorithm. The class takes in a training environment and an evaluation environment, as well as some hyper-parameters for the reinforcement learning agent. In table 3.7, a complete overview of the attributes in the EO-GRAPE Approximation Class is provided.

Table 3.7: Overview of the attributes in the EO-GRAPE Quantum Reinforcement Learning class, including the attribute name and description of each attribute.

Attribute name	**Description**
TrainEnvironment	An instance of an EO-GRAPE Approximation Environment for training purposes
EvaluationEnvironment	An instance of an EO-GRAPE Approximation Environment for evaluation purposes
num_iterations	The number of training loop iterations
fc_layer_params	The number of nodes per hidden neural network layer
learning_rate	The learning rate of the reinforcement learning agent
replay_buffer_capacity	The size of the replay buffer of the reinforcement learning agent
policy	Optional: the initial policy of the reinforcement learning agent

The class mainly contains functions to create the reinforcement learning agent and neural network, and to run the training loop on the EO-GRAPE Approximation environment. Similar to the Quantum Reinforcement Learning class, the class also contains functions to save the weights of the trained neural network, as well as some plotting functionalities to visualize the performance of the agent during the training loop. A complete overview of the methods contained in the EO-GRAPE Quantum Reinforcement Learning class is shown in table 3.8.

Table 3.8: Overview of the methods of the EO-GRAPE Quantum Reinforcement Learning class. Including the method name, description and what the method returns

Method name	**Description**	**Returns**
create_network_agent	Creates an instance of a TensorFlow REINFORCE agent	TensorFlow REINFORCE agent
run_training	Runs the training loop of the reinforcement learning agent	none
save_weights	Saves the weights of the nodes in the neural network	Neural network weights
show_summary	Prints a summary of the nodes in the neural network	none
get_final_pulse	Returns the final pulse of the reinforcement learning agent	Control pulses
get_best_pulse	Returns the highest reward pulse of the reinforcement learning agent	Control pulses
plot_reward_per_iteration	Plots the reward as a function of the training loop iterations	none
plot_best_pulse	Plots the highest reward pulse of the reinforcement learning agent	none
plot_final_pulse	Plots the final pulse of the reinforcement learning agent	none

In this chapter, the theoretical framework for the EO-GRAPE algorithm and deep reinforcement learning are introduced. Subsequently, the implementation of these algorithms in python is discussed, where we introduced the ”EUQOC” python package. Using the theoretical framework in combination with the implementation of this in python, one can run experiments to test the theory and benchmark the two Quantum Optimal Control methods. In the next section the results of this thesis will be provided. First the EO-GRAPE performance will be discussed, followed by the reinforcement learning agent performance. Next, the two methods are compared to each other in the presence of increasing noise. Finally, the two methods of evaluating the energetic cost of implementing a quantum unitary gate are correlated.

4 Results

It doesn't matter how beautiful your theory is, it doesn't matter how smart you are. If it doesn't agree with experiment, it's wrong.
- Richard Feynman

In this chapter, the results of the research are presented. First, the results for the EO-GRAPE algorithm will be displayed, including some example pulses, the algorithm convergence, the learning rate optimization, and finally the relation or trade-off between energetic cost and fidelity. After this, the performance of both RL agents will be investigated. Thirdly, the effect of increasing system noise on both methods is shown, as well as the performance with and without warm start, and with varying neural network size. Finally, the results regarding the correlation between the energetic cost and the path length on the Bloch sphere will be presented.

To make the results more readable, we have introduced some abbreviations, which will be introduced below.

For the target quantum unitary gate, the CNOT, Hadamard, T-gate and $R_x(\pi/2)$ gate have been used, which are defined as:

$$\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \quad \text{H} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad \text{T} = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}, \quad R_x(\theta) = \begin{pmatrix} \cos\theta/2 & -i\sin\theta/2 \\ -i\sin\theta/2 & \cos\theta/2 \end{pmatrix} \tag{4.1}$$

For the drift Hamiltonian H_d, two main definitions have been used for the single qubit case H_d^1 and the two-qubit case H_d^2, defined as:

$$H_d^1 = \frac{\hbar\omega_1}{2}\hat{\sigma}_z, \quad H_d^2 = \frac{\hbar\omega_1}{2}\hat{\sigma}_z^{(1)} \otimes \hat{I}_2^{(2)} + \frac{\hbar\omega_2}{2}\hat{I}_2^{(1)} \otimes \hat{\sigma}_z^{(2)} + \hbar J\hat{\sigma}_z^{(1)} \otimes \hat{\sigma}_z^{(2)} \tag{4.2}$$

Where ω_i is the eigen-frequency of qubit i, and J is the coupling strength between qubits.

The total gate time has been fixed to $T = 2\pi$, and the relaxation and decoherence time T_1 and T_2 are defined as multiples of the total gate time T. The abbreviation "*WS*" is used to indicate a "Warm Start" of the reinforcement learning agent by having an initial policy of RLA-2. Similarly, "*WOWS*" is used to indicate "Without Warm Start", i.e., the reinforcement learning agent starts its policy from scratch.

Different Neural Network sizes are used, that are defined as: Neural Network Size 1 = $(200, 100, 50, 30, 10)$, Neural Network Size 2 = $(400, 200, 100, 50, 30, 10)$ and Neural Network Size 3 = $(600, 400, 200, 100, 50, 30, 10)$.

4.1. EO-GRAPE Performance

In this section, the results of the EO-GRAPE algorithm will be introduced. Some optimal pulse examples will be shown first, followed by the convergence of the algorithm. After this the learning rate optimization analysis will be provided. Finally, the trade-off between fidelity and energetic cost will be shown, by sweeping over the weight parameters and plotting their respective values.

4.1.1. Optimal Pulses

In figure 4.1, one can see an example pulse generated by the EO-GRAPE algorithm where the target unitary quantum gate is a CNOT gate, using H_d^2 and three control operators $\{\sigma_x^1, \sigma_x^2, \sigma_x^1\sigma_x^2\}$. As one expects, the target qubit is driven on the frequency of the control qubit to implement a controlled NOT gate. The dark blue line indicates the final values of the control parameters after $N_G = 500$ iterations, and the lighter blue lines indicate the previous iteration values, to show how the algorithm updates the parameters.

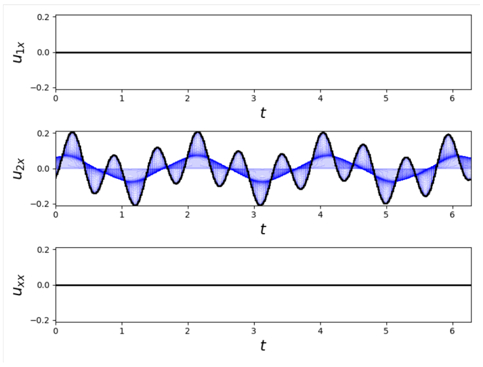
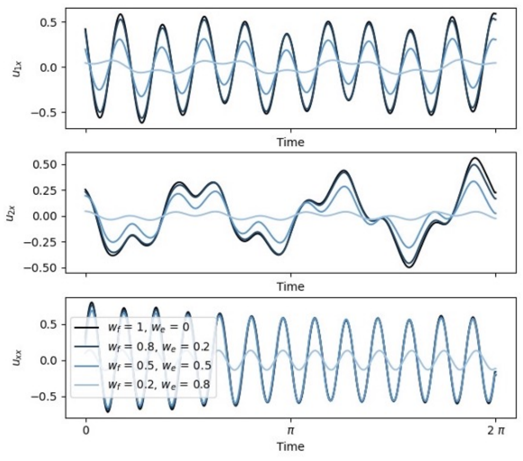
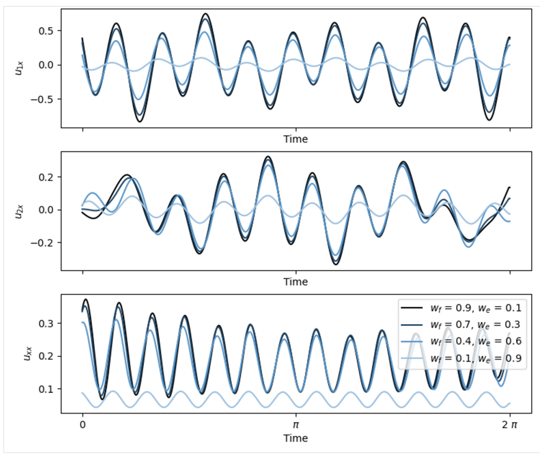


Figure 4.1: Example control pulses (blue) generated by the EO-GRAPE algorithm, implementing a CNOT gate. The final iteration is given by the dark blue line. Parameters: U_T = *CNOT*, $H_d = H_d^2$, $H_k = \{\sigma_x^1, \sigma_x^2, \sigma_x^1\sigma_x^2\}$, $T_1 = \infty$, $T_2 = \infty$, $w_f = 1$, $w_e = 0$, $N_t = 500$, $N_g = 500$

In figure 4.2, the effect of increasing or decreasing the weight associated to fidelity w_f and energetic cost w_e is shown. As one can see, the higher the value of w_e, the lower the amplitude of the control pulses are, to decrease the area, and thus decrease the energetic cost, intuitively matching our expectations. One can also see some interesting other harmonics being introduced when increasing w_e to decrease the energetic cost of the pulses. An overview of the parameters such as U_T, H_d, H_k and others is given in the caption of the figure.



(a)



(b)

Figure 4.2: **(a)** Example control pulses generated by the EO-GRAPE algorithm for different weight settings. **(b)** Example control pulses generated by the EO-GRAPE algorithm for different weight settings. Parameters: U_T = *RAND*, $H_d = H_d^2$, $H_k = \{\sigma_x^1, \sigma_x^2, \sigma_x^1\sigma_x^2\}$, $T_1 = \infty$, $T_2 = \infty$, $w_f = [1, 0.1]$, $w_e = [0, 0.9]$, $N_t = 500$, $N_g = 500$

4.1.2. Convergence

In this section, the convergence of the EO-GRAPE algorithm is investigated, by looking at the gradient and cost function value as a function of EO-GRAPE iteration number. As the gradient is calculated for each time step and for each control line, for each EO-GRAPE iteration, it is inherently impossible to visualize. We therefore look at some other values that indicate a convergence of the algorithm.

In figure 4.3, the average value of the gradient of the maximum value over time of each control line is depicted, for different weight settings w_f and w_e. As one can see, all lines approach to zero as the number of EO-GRAPE iterations increases, indicating a good convergence of the algorithm, along with an indication on how many EO-GRAPE N_G are required for convergence.

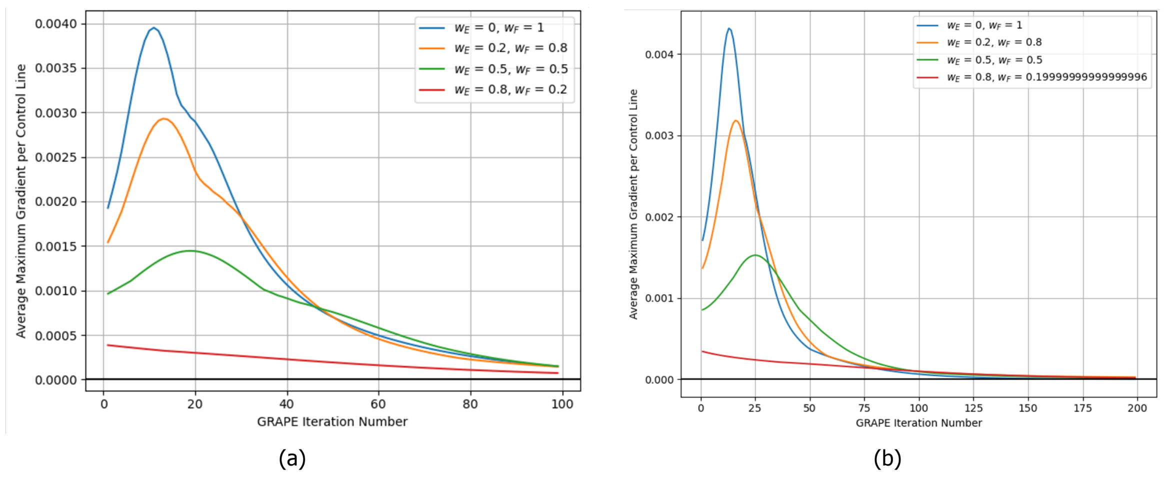


Figure 4.3: **(a)** and **(b)** Averaged maximum gradient of all control lines as a function of EO-GRAPE iteration number, for several different weight settings. Parameters: U_T = *RAND*, $H_d = H_d^2$, $H_k = \{\sigma_x^1, \sigma_x^2, \sigma_x^1\sigma_x^2\}$, $T_1 = \infty$, $T_2 = \infty$, $w_f = [1, 0.2]$, $w_e = [0, 0.8]$, $N_t = 500$, $N_g = 200$

In figure 4.4, the maximum value of the gradient over all time steps for each control line is shown, for a fixed set of weights. Again, one can see in both figures that the gradient approach zero as the number of EO-GRAPE iterations is increased.

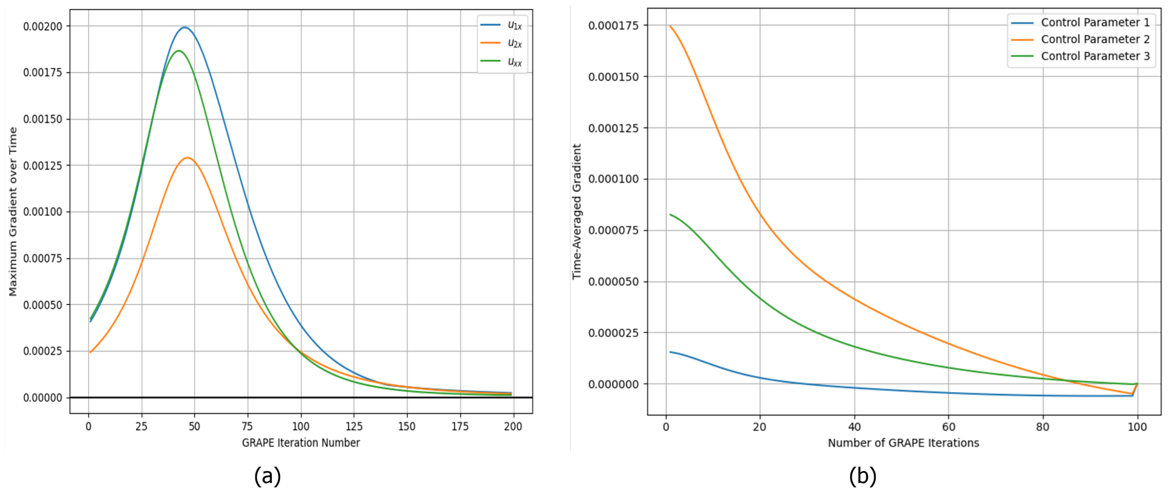


Figure 4.4: **(a)** Maximum gradient over all time steps for each control operator as a function of EO-GRAPE iteration number, using $w_f = 0.5$ and $w_e = 0.5$. **(b)** Maximum gradient over all time steps for each control operator as a function of EO-GRAPE iteration number, using $w_f = 0.2$ and $w_e = 0.8$. Parameters: U_T = *RAND*, $H_d = H_d^2$, $H_k = \{\sigma_x^1, \sigma_x^2, \sigma_x^1\sigma_x^2\}$, $T_1 = \infty$, $T_2 = \infty$, $w_f = 0.5, 0.2$, $w_e = 0.5, 0.8$, $N_t = 500$, $N_g = 200, 100$

Finally, in figure 4.5, we can see the value of the infidelity (orange), normalized energetic cost (green), and total value of the cost function (blue), using $w_f = 0.8$, and $w_e = 0.2$. Again, we can see that the cost function converges to zero as the number of EO-GRAPE iterations is increased. The cost function cannot actually approach zero, because the highest fidelity pulse will always require some energetic cost to be implemented. One can already see that there is an indication of a trade-off between fidelity and energetic cost, which will be explored further in section 4.1.4.

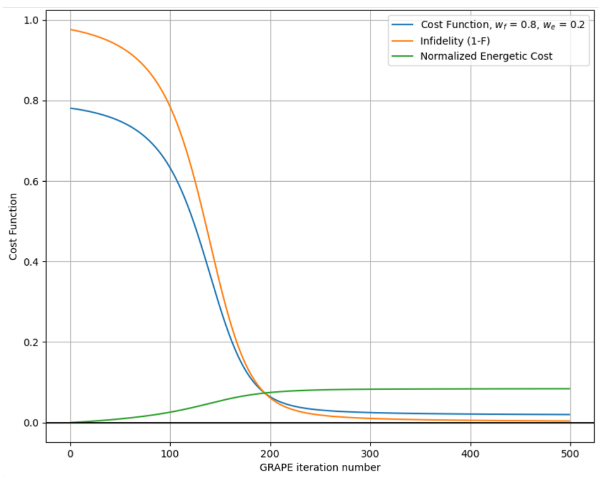


Figure 4.5: Cost function (blue), infidelity (orange) and normalized energetic cost (green) as a function of EO-GRAPE iteration number. Parameters: U_T = *RAND*, $H_d = H_d^2$, $H_k = \{\sigma_x^1, \sigma_x^2, \sigma_x^1\sigma_x^2\}$, $T_1 = \infty$, $T_2 = \infty$, $w_f = 0.8$, $w_e = 0.2$, $N_t = 500$, $N_g = 200$

4.1.3. Learning Rate Optimization

Two important parameters that have a great influence on the eventual convergence and final value of the cost function are the learning rates of fidelity ϵ_f the energetic cost ϵ_e part in the EO-GRAPE updating rule. To ensure that the algorithm is performing in the best possible way, one needs to investigate the influence of these parameters on the final cost function value. Therefore, before investigating the trade-off between fidelity and energetic cost, and benchmarking the EO-GRAPE algorithm against the reinforcement learning methods, the learning rates ϵ_f and ϵ_e have been optimized. In figure 4.6 one can see an example optimization analysis using $w_e = 0.5$ and $w_f = 0.5$. The figure shows the value of the cost function after $N_g = 500$ EO-GRAPE iterations, as a function of the fidelity learning rate ϵ_f and the energy learning rate ϵ_e. The analysis has shown an optimal combination of $\epsilon_f = 1$ and $\epsilon_e = 100$, which we will use as learning rate values for all upcoming experiments.

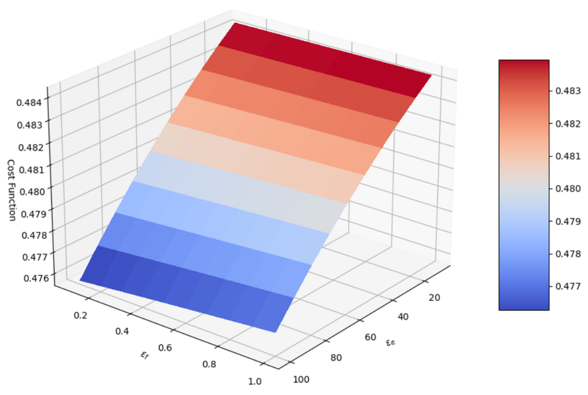


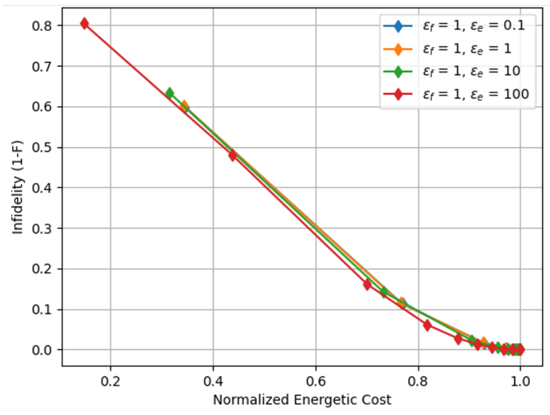
Figure 4.6: Value of the cost function after EO-GRAPE optimization for different settings for the energetic cost learning rate ϵ_e and the fidelity learning rate ϵ_f. Parameters: U_T = *RAND*, $H_d = H_d^2$, $H_k = \{\sigma_x^1, \sigma_x^2, \sigma_x^1\sigma_x^2\}$, $T_1 = \infty$, $T_2 = \infty$, $w_f = 0.5$, $w_e = 0.5$, $N_t = 500$, $N_g = 100$

4.1.4. Fidelity and Energetic Cost Trade-off

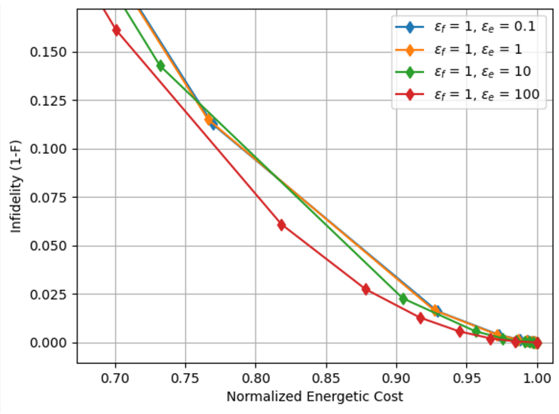
As stated in the research questions of this thesis, we would like to investigate what the relation is between the fidelity of a quantum unitary gate, and the energetic cost needed to implement a quantum unitary gate. If there exists a trade-off between fidelity and energetic cost, we would like to know how severe it is and what shape it has. In this section the results regarding the trade-off between fidelity and energetic cost are discussed.

In figure 4.7, the results of this experiment are depicted. The EO-GRAPE algorithm was run for different set of weights w_f and w_e, and the combination of the value of the fidelity and energetic cost after optimization is stored. These values have been plotted against each other for different values of ϵ_f and ϵ_e. Again one can see that the optimal combination of values is $\epsilon_f = 1$ and $\epsilon_e = 100$, like we concluded from our learning rate optimization analysis. One can clearly see the Pareto front and trade-off between fidelity and energetic cost, where a reduction in energetic cost, directly leads to an increase in infidelity, or decrease in fidelity. This result matches with our intuition that to achieve a lower energetic cost quantum unitary gate, one inherently decreases the area or amplitude of the control pulses, resulting in a lower process fidelity.

Nevertheless, as one can see on the right hand side of the figures, the two are not completely inversely proportional to each other. We can therefore still decrease the energetic cost of a quantum unitary gate by roughly **10 %**, while decreasing the fidelity by roughly **1 %**. Therefore, if one wants to achieve a minimum 2-qubit gate fidelity of 99 %, one could decrease the energetic cost of each quantum unitary gate by 10 %.



(a)



(b)

Figure 4.7: **(a)** Infidelity and energetic cost values for different weight settings and learning rates ϵ_e, showing the trade-off or Pareto front between fidelity and energetic cost. **(b)** Zoomed-in view of the Pareto front between fidelity and energetic cost. Parameters: U_T = *RAND*, $H_d = H_d^2$, $H_k = \{\sigma_x^1, \sigma_x^2, \sigma_x^1\sigma_x^2\}$, $T_1 = \infty$, $T_2 = \infty$, $w_f = [1, 0.1]$, $w_e = [0, 0.9]$, $N_t = 500$, $N_g = 100$

4.2. Reinforcement Learning Performance

In this section the first results of the performance of the reinforcement learning agents are introduced. Firstly the performance of RLA-1, without initial policy will be discussed, and afterwards the results with initial policy from RLA-2 will be introduced.

4.2.1. RLA-1 Performance

In figure 4.8, the fidelity (red), energetic cost (green), and total reward (blue) of RLA-1 for different weight settings are shown, as a function of the training loop iterations (episodes). As one can see from the figures, the agent is able to learn how to modify the control pulses to optimize the reward. Each episode a new random initial state is generated and used for calculating the target density matrix, by applying the target unitary to the initial state. The fidelity is then calculated by the overlap between the output density matrix after applying the action of the agent, and the target density matrix.

When w_e is increased, the energetic cost will be lower, however the fidelity also deteriorates quite rapidly. Next to this the variation in the fidelity also increases when weight of the energetic cost is set higher than the weight of the fidelity. This could be the result of the randomly generated initial state each episode, and could indicate that the agent is not able to actually learn the unitary, but rather a state-to-state transfer. These results motivated us to train RLA-2 on EO-GRAPE generated pulses, and transfer the trained policy onto RLA-1 as an initial policy.

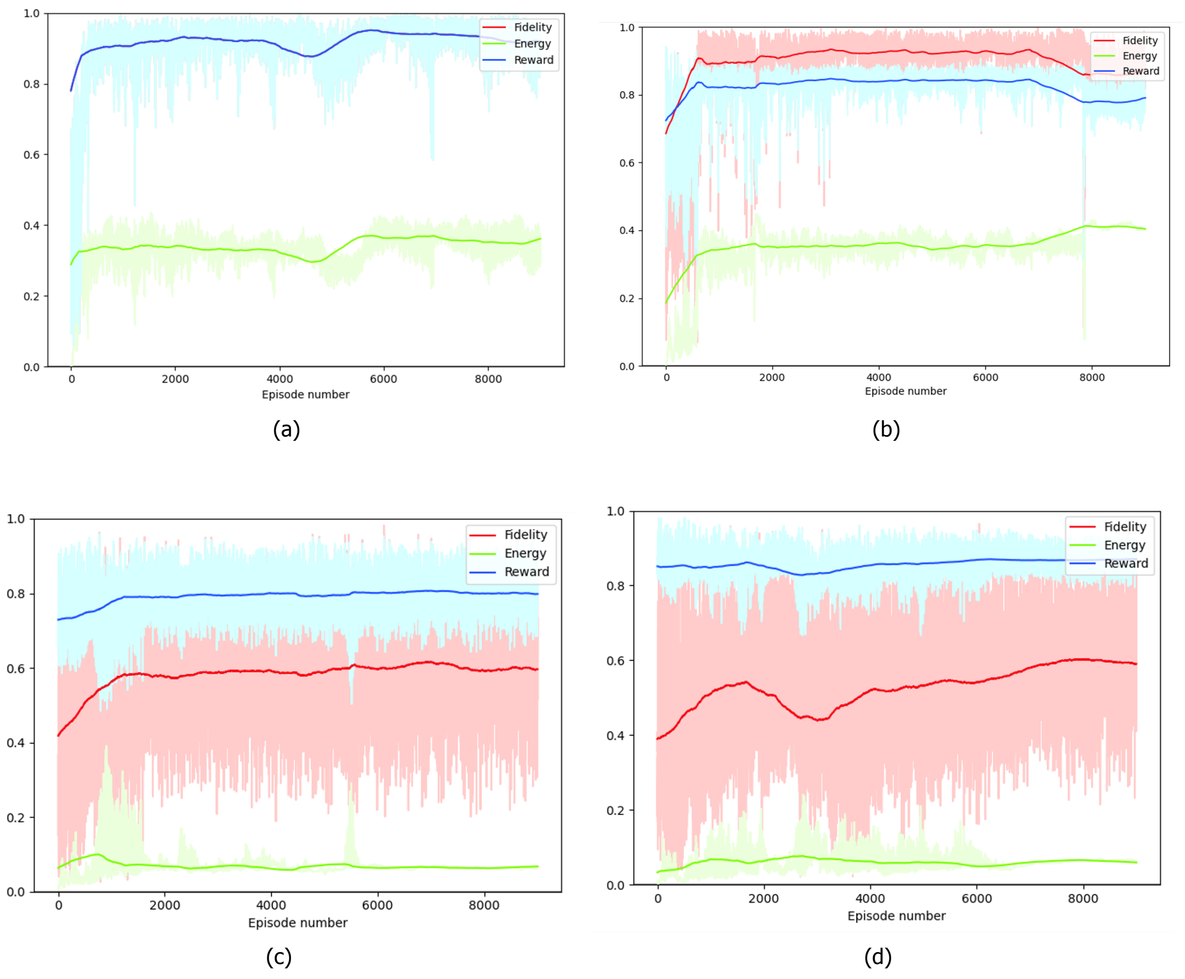


Figure 4.8: Fidelity (red), Energetic Cost (green) and total reward (blue) as a function of training episode number for RLA-1, for different weight settings: **(a)** $w_f = 1$, $w_e = 0$, **(b)** $w_f = 0.7$, $w_e = 0.3$, **(c)** $w_f = 0.4$, $w_e = 0.6$, **(d)** $w_f = 0.2$, $w_e = 0.8$. Parameters: $U_T = CNOT$, $H_d = H_d^2$, $H_k = \{\sigma_x^1, \sigma_x^2, \sigma_x^1\sigma_x^2\}$, $T_1 = \infty$, $T_2 = \infty$, $N_t = 10$, *layer_params* = $(200, 100, 50, 30, 10)$

4.2.2. RLA-2 Performance

In figure 4.9, we show the reward as a function of training loop iteration number (episodes), of RLA-2. The reward is equal to the negative value of the square difference between the target pulse and the action of the agent, given by equation 3.28. The agent therefore tries to minimize the square difference between the action and the target pulse, to optimize the reward. As one can see from figure 4.10, the agent is able to learn the pulse generated by the EO-GRAPE algorithm after roughly 2000 episodes. For all future experiments we therefore use $N_{GA} = 2000$ as the number of Grape Approximation iterations.

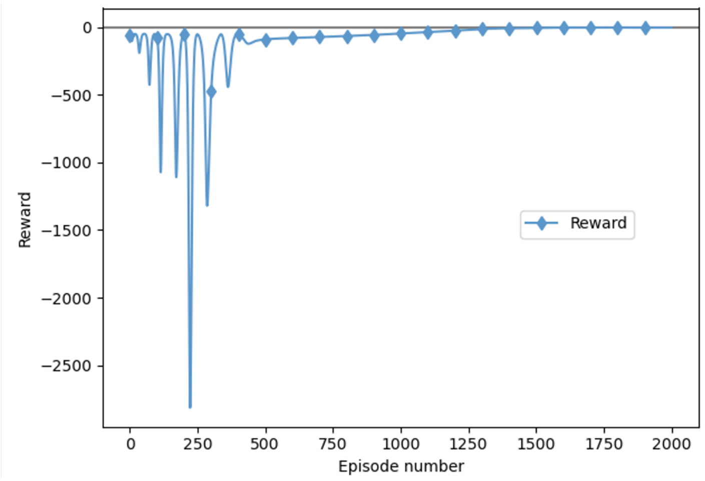


Figure 4.9: Reward of RLA-2 as a function of the training episode number. Parameters: U_T = *CNOT*, $H_d = H_d^2$, $H_k = \{\sigma_x^1, \sigma_x^2, \sigma_x^1\sigma_x^2\}$, $T_1 = 100T$, $T_2 = 100T$, $N_t = 100$, $N_g = 500$, *layer_params* = $(100, 100, 100)$, $N_{QRLA} = 10,000$, $N_{GA} = 2,000$.

If we use the trained policy of RLA-2 as an initial policy for RLA-1, we get quite a different learning curve as compared to starting from scratch. In figure 4.10, the fidelity (red), energetic cost (green), and total reward (blue) as a function of episode number, for RLA-1 with a pre-trained policy is shown. As one can see, the fidelity and energetic cost stay very stable throughout the whole training loop, and there seems to be no further increase in performance. One could therefore question whether the agent is actually still able to learn the system, or is simply replicating pulses that are almost similar and seem to receive a high reward. To investigate this, the performance of both methods is investigated under the influence of increasing noise, or decreasing relaxation T_1 and decoherence T_2 times. The results for these experiments are discussed in the next section.

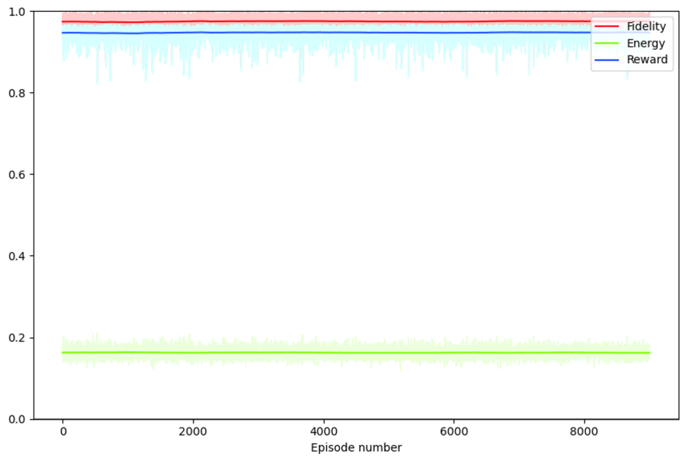


Figure 4.10: Fidelity (red), Energetic Cost (green), and Total Reward (blue), as a function of training episode number, using the trained policy from RLA-2 as initial policy. Parameters: U_T = *CNOT*, $H_d = H_d^2$, $H_k = \{\sigma_x^1, \sigma_x^2, \sigma_x^1\sigma_x^2\}$, $T_1 = 1000T$, $T_2 = 1000T$, $w_f = 0.8$, $w_e = 0.2$, $N_t = 100$, $N_g = 500$, *layer_params* = $(200, 100, 50, 30, 10)$, $N_{QRLA} = 10,000$, $N_{GA} = 2,000$.

4.3. Noise Increase

In this section, the results of the noise increase experiments will be shown and discussed. The individual performance of both the EO-GRAPE algorithm and the reinforcement learning methods will be shown first. Afterwards, the performance of both methods will be benchmarked against each other. Finally, the effect of increasing the neural network size on the performance in the presence of noise will be presented.

4.3.1. Individual Performance

The performance of the EO-GRAPE algorithm while increasing the noise in the system is shown in figure 4.11. We can see that the algorithm is able to achieve high fidelity and low energetic cost throughout most noise settings. From a noise gain setting of $20T$ we can see that the performance decreases quite rapidly.

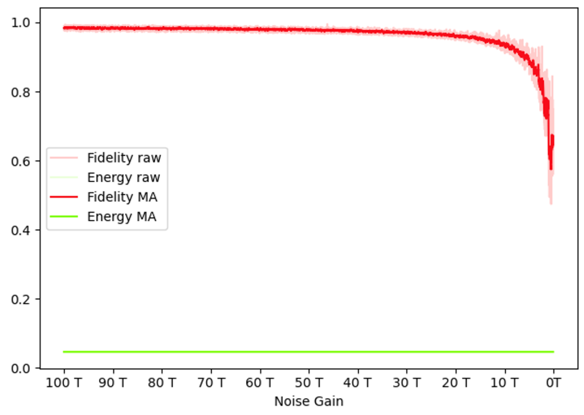


Figure 4.11: Fidelity (red) and Energetic Cost (green) of EO-GRAPE generated control pulses as a function of decreasing decoherence time, or increasing noise level. Parameters: U_T = *Hadamard*, $H_d = H_d^1$, $H_k = \{\sigma_x^1\}$, $T_1 = [100T, 1T]$, $T_2 = [100T, 1T]$, $w_f = 0.7$, $w_e = 0.3$, $N_t = 100$, $N_g = 500$.

In figure 4.12, the performance of the reinforcement learning agent both with and without warm start (by RLA-2) in the presence of increasing noise is given. As we can see, the agent without warm start is able to reach slightly higher fidelity than the agent with warm start, however the energetic cost of the agent with warm start is more stable and lower.

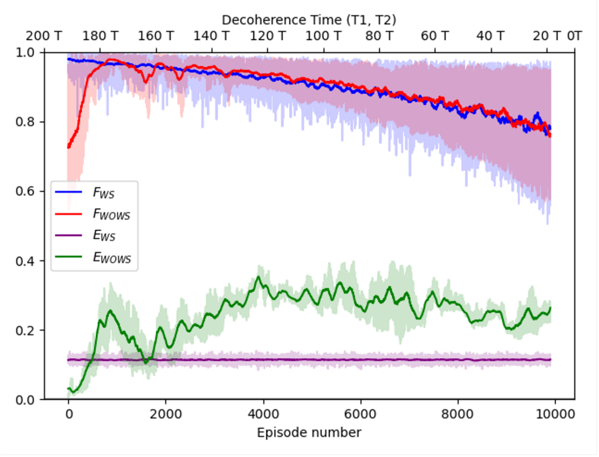


Figure 4.12: Fidelity (blue) and energetic cost (purple) of RL generated control pulses with warm start, and fidelity (red) and energetic cost (green) of RL generated control pulses without warm start, as a function of the training episode number and decreasing decoherence time. Parameters: U_T = *Hadamard*, $H_d = H_d^1$, $H_k = \{\sigma_x^1\}$, $T_1 = [200T, 1T]$, $T_2 = [200T, 1T]$, $w_f = 0.8$, $w_e = 0.2$, $N_t = 100$, $N_g = 500$, *layer_params* = $(200, 100, 50, 30, 10)$, $N_{QRLA} = 10,000$, $N_{GA} = 2,000$.

The EO-GRAPE algorithm calculates control parameters based on the target unitary and the drift and control Hamiltonian that are provided, and thus does not take into account any noise in the system.

It is therefore quite remarkable that the control pulses generated by the EO-GRAPE algorithm are still able to achieve high fidelity and low energetic cost in a highly noisy environment. The reinforcement learning agent without warm start seems to be able to learn the noise characteristics of the system first, and could therefore outperform the reinforcement learning agent with warm start on fidelity. However when we take into account energetic cost as well, it is clear that the reinforcement learning agent with warm start outperforms the agent without a warm start. To investigate how the EO-GRAPE algorithm compares to the reinforcement learning methods, the performance of both will be investigated for a universal set of gates. The results of these experiments will be shown in the next section.

4.3.2. EO-GRAPE versus RL Performance

To compare the EO-GRAPE algorithm performance to both the warm start and without warm start reinforcement learning agents, the fidelity and energetic cost of a universal gate set (CNOT, Hadamard, T) using all three methods has been plotted as a function of increasing noise gain or decreasing relaxation and decoherence time.

In figure 4.13, the results of these experiments are presented. One can see that the EO-GRAPE algorithm outperforms both the warm start and without warm start reinforcement learning agents for all three target unitary gates, and for all noise settings. Interestingly, the variance in the performance of the reinforcement learning agent optimizing a T-gate is much smaller than the variance with the CNOT gate or Hadamard gate as target unitary gate. Next to this, one can see that the reinforcement learning agent with warm start in general has a smaller variance in performance than the reinforcement learning agent without warm start, and that the energetic cost of the pulses generated by the reinforcement learning agent with warm start are in general lower than the energetic cost of the pulses generated by the reinforcement learning agent without warm start.

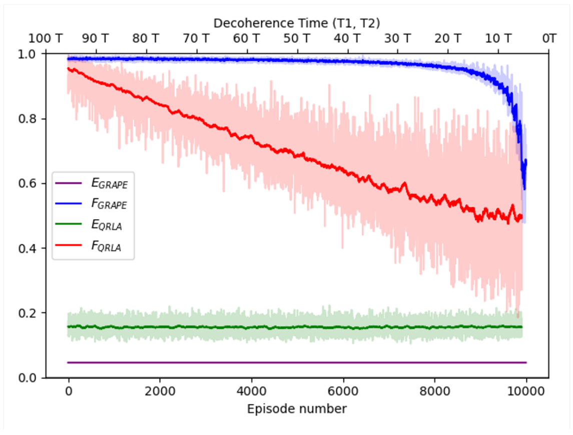
From this figure, we can thus conclude that the reinforcement learning agent is able to learn a universal set of gates co-optimized on fidelity and energetic cost, and additionally that the EO-GRAPE algorithm outperforms both reinforcement learning agents on fidelity and energetic cost, for all noise settings.

4.3.3. Neural Network Size Increase

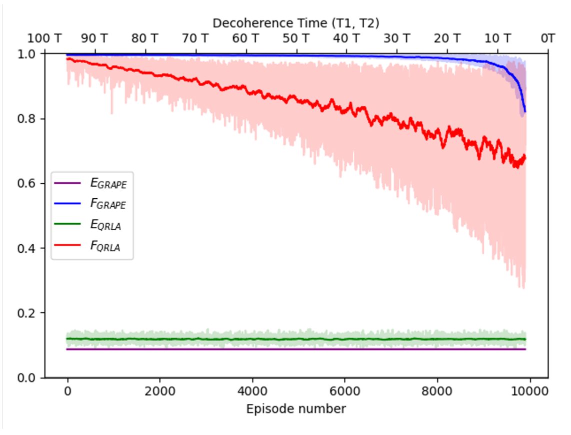
To further explore the performance of the reinforcement learning agent compared to the EO-GRAPE algorithm, we have investigated the effect of the size of the neural network on the performance of the reinforcement learning agent. To make the effect more noticeable, the weight of energetic cost was set to zero $w_e = 0$, and the weights of the fidelity was set to one $w_f = 0$, i.e., the agent is only trying to optimize the pulse on fidelity, and not on energetic cost. The same experiments as in figure 4.13 have been done, where the fidelity of both the EO-GRAPE algorithm and the reinforcement learning agents with and without warm start are plotted as a function of increasing noise or decreasing relaxation and decoherence time. The experiments all use the Hadamard gate as the target quantum unitary gate, and are plotted for three different neural network sizes, with and without warm start, and two different noise settings. The Neural Network Sizes can be found in introduction of chapter 4. In figure 4.14, the results of the experiment are displayed for a low noise setting, i.e., $T_1, T_2 \in [200T, 10T]$, and in figure 4.15, the results are shown for a high noise setting, i.e., $T_1, T_2 \in [10T, 0.1T]$.

In figure 4.14, one can see that in a low noise environment, the EO-GRAPE algorithm outperforms the reinforcement learning agent, both with and without warm start, and for all three neural network sizes. One can also see the increasing the neural network size has little effect on the performance of the reinforcement learning agent, apart from the variance getting slightly smaller.

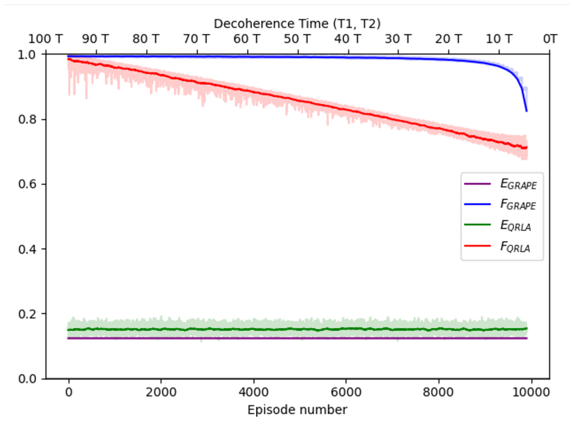
In figure 4.15, one can see that in a high noise environment, the performance of the EO-GRAPE algorithm and the reinforcement learning agents approach each other. However, as the variance is so big, one cannot conclude any significant result from this. Also, the noise setting of $0.1T$, is not physically relevant, and a fidelity of $F = 0.7$ is also not usable in a physical system. We can again see that the effect of the neural network size is small, apart from a slight decrease in the variance of the performance.



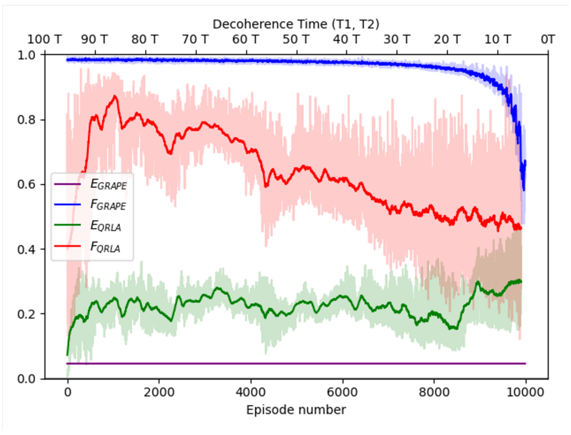
(a)



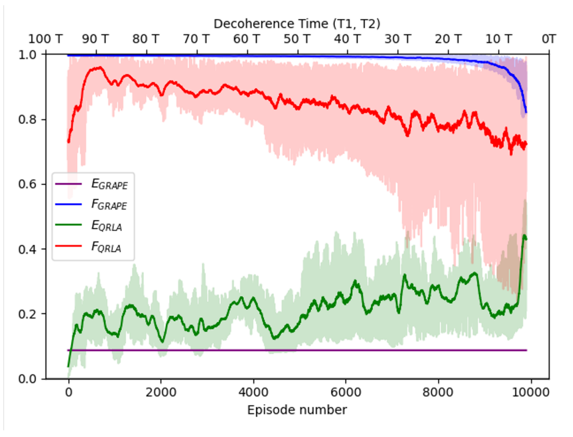
(b)



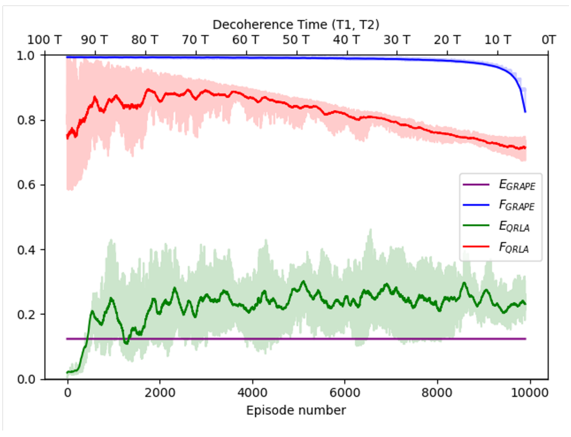
(c)



(d)

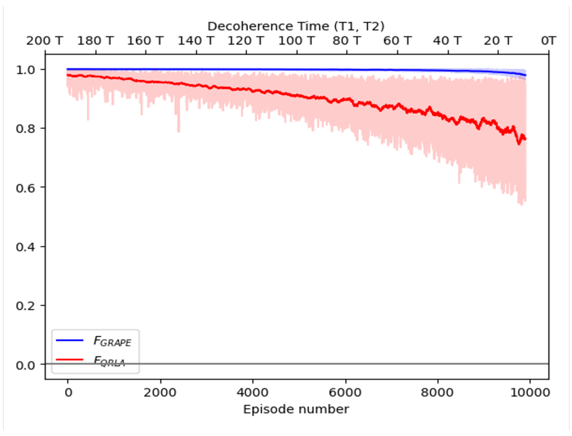


(e)

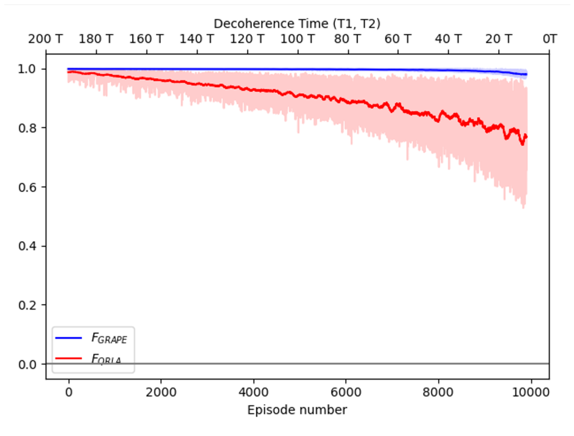


(f)

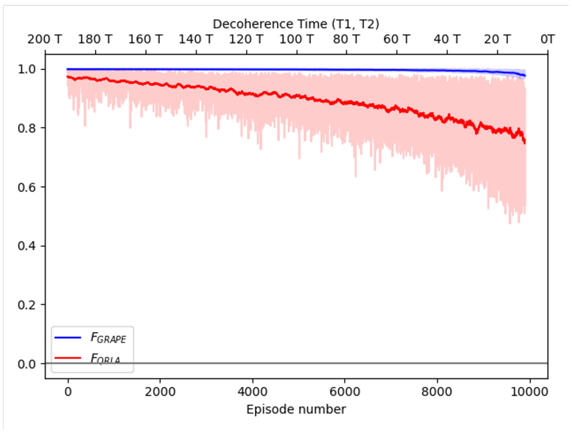
Figure 4.13: Fidelity (blue) and energetic cost (purple) of EO-GRAPE generated control pulses, and fidelity (red) and energetic cost (green) of RL generated control pulses, as a function of the training episode number and decreasing decoherence time, for a universal set of gates, and with and without warm start. **(a)** CNOT gate, with warm start, **(b)** Hadamard gate, with warm start, **(c)** T-gate, with warm start, **(d)** CNOT gate, without warm start, **(e)** Hadamard gate, without warm start, **(f)** T-gate, without warm start. Parameters: U_T = *CNOT, Hadamard, T*, $H_d = H_d^2, H_d^1$, $H_k = \{\sigma_x^1\}, \{\sigma_x^1, \sigma_x^2, \sigma_x^1\sigma_x^2\}$, $T_1 = [100T, 1T]$, $T_2 = [100T, 1T]$, $w_f = 0.8$, $w_e = 0.2$, $N_t = 100$, $N_g = 500$, *layer_params* = $(200, 100, 50, 30, 10)$, $N_{QRLA} = 10{,}000$, $N_{GA} = 2{,}000$.



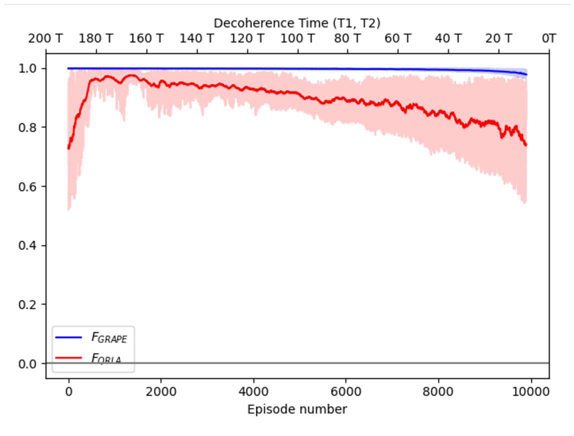
(a)



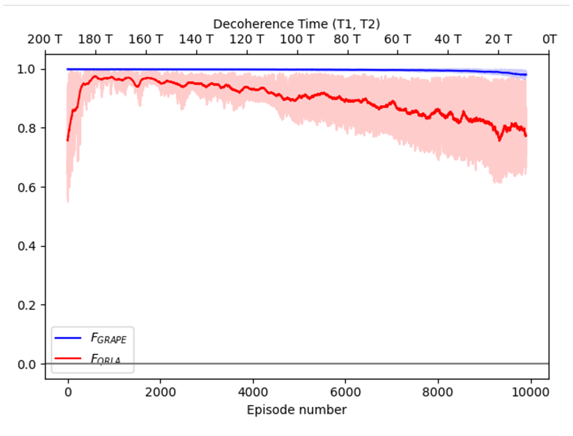
(b)



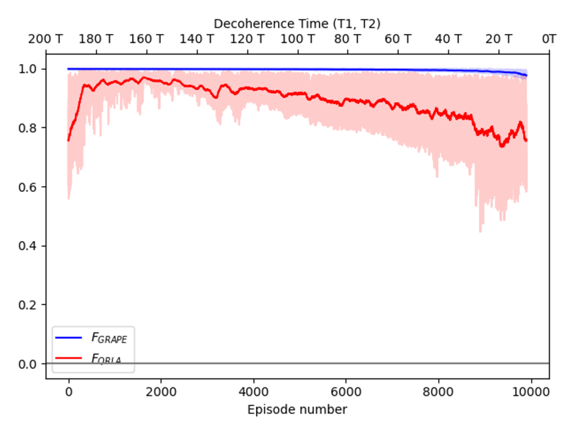
(c)



(d)

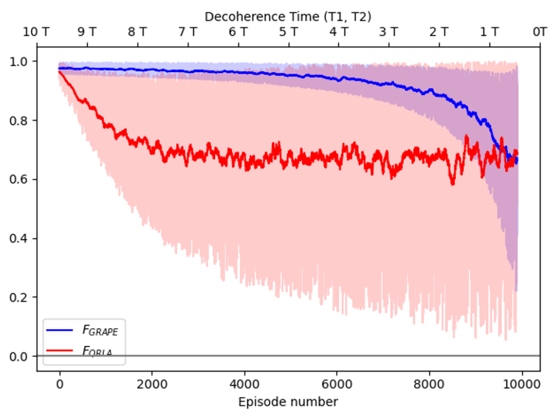


(e)

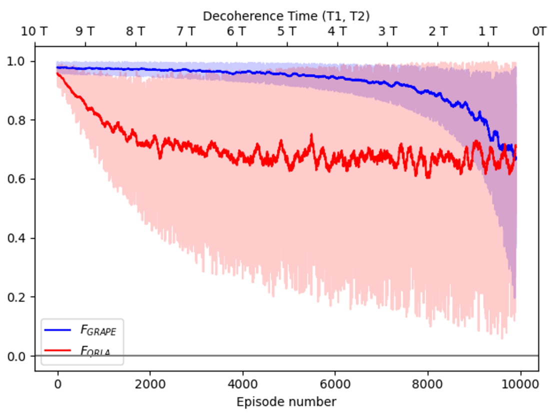


(f)

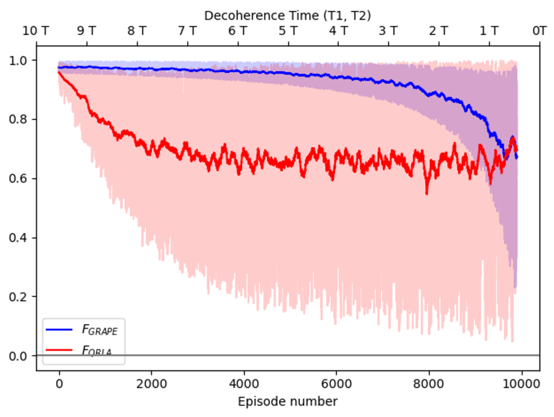
Figure 4.14: EO-GRAPE generated pulses fidelity (blue) and RL generated pulses fidelity (red), as a function of training episode number and decreasing decoherence time (high noise setting), for different neural network sizes and with and without warm start. **(a)** Neural Network Size 1, with warm start, **(b)** Neural Network Size 2, with warm start, **(c)** Neural Network Size 3, with warm start, **(d)** Neural Network Size 1, without warm start, **(e)** Neural Network Size 2, without warm start, **(f)** Neural Network Size 3, without warm start. Parameters: U_T = *Hadamard*, $H_d = H_d^2$, $H_k = \{\sigma_x^1\}$, $T_1 = [200T, 10T]$, $T_2 = [200T, 10T]$, $w_f = 1$, $w_e = 0$, $N_t = 100$, $N_g = 500$, $N_{QRLA} = 10,000$, $N_{GA} = 2,000$.



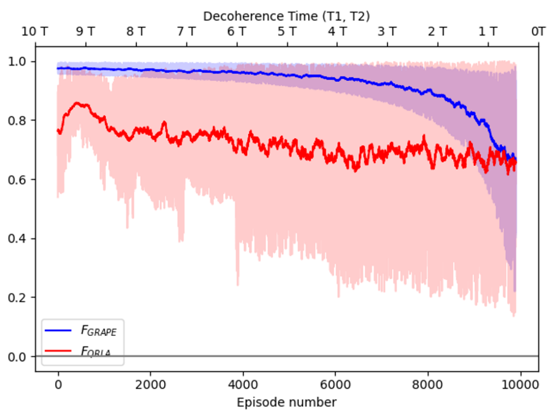
(a)



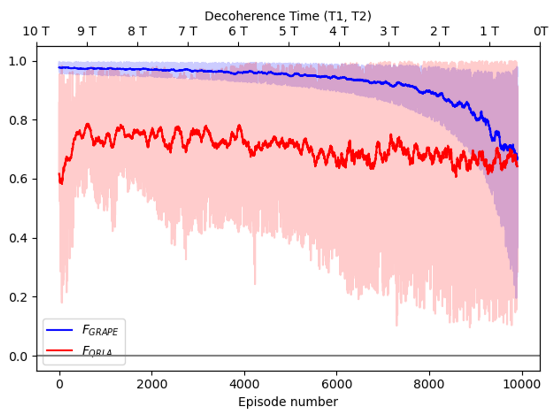
(b)



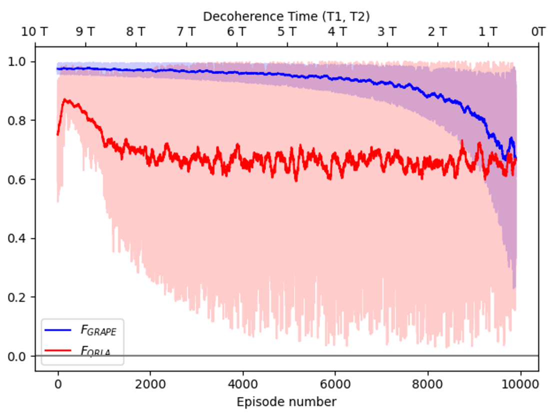
(c)



(d)



(e)



(f)

Figure 4.15: EO-GRAPE generated pulses fidelity (blue) and RL generated pulses fidelity (red), as a function of training episode number and decreasing decoherence time (low noise setting), for different neural network sizes and with and without warm start. **(a)** Neural Network Size 1, with warm start, **(b)** Neural Network Size 2, with warm start, **(c)** Neural Network Size 3, with warm start, **(d)** Neural Network Size 1, without warm start, **(e)** Neural Network Size 2, without warm start, **(f)** Neural Network Size 3, without warm start. Parameters: U_T = *Hadamard*, $H_d = H_d^2$, $H_k = \{\sigma_x^1\}$, $T_1 = [10T, 0.1T]$, $T_2 = [10T, 0.1T]$, $w_f = 1$, $w_e = 0$, $N_t = 100$, $N_g = 500$, $N_{QRLA} = 10,000$, $N_{GA} = 2,000$.

4.4. Bloch Sphere Path Length

In this section, the effect of the control pulses generated by both the EO-GRAPE algorithm and the reinforcement learning agents on the unitary paths on the Bloch sphere are investigated. First the effect of the drift Hamiltonian will be shown for both the EO-GRAPE algorithm and the reinforcement learning agent. Afterwards, the relation between the path length of the unitary path on the Bloch sphere and the energetic cost calculated through equation 2.16 will be investigated.

4.4.1. Example Paths

In figure 4.16, the unitary path on the Bloch sphere by the control pulses generated by the EO-GRAPE algorithm (left) and the reinforcement learning agent (right), are shown. An initial state of $|\psi_i\rangle = |0\rangle$ (green vector) and a target unitary of $U_T = R_x(\pi/2)$, is used, resulting in a target state of $|\psi_T\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle) \equiv |-i\rangle$ (orange vector). The control Hamiltonian operators are $H_k = \{\sigma_x^1, \sigma_y^1\}$, and there is no drift Hamiltonian, $H_d = I_2$. One can see that path induced by the control pulses generated by the EO-GRAPE algorithm are much more smooth and straight than the path induced by the control pulses generated by the reinforcement learning agent. One can nicely see how the reinforcement learning agent learns by reward, as seen by the random walk paths the state vector travels, before eventually arriving at the correct target state.

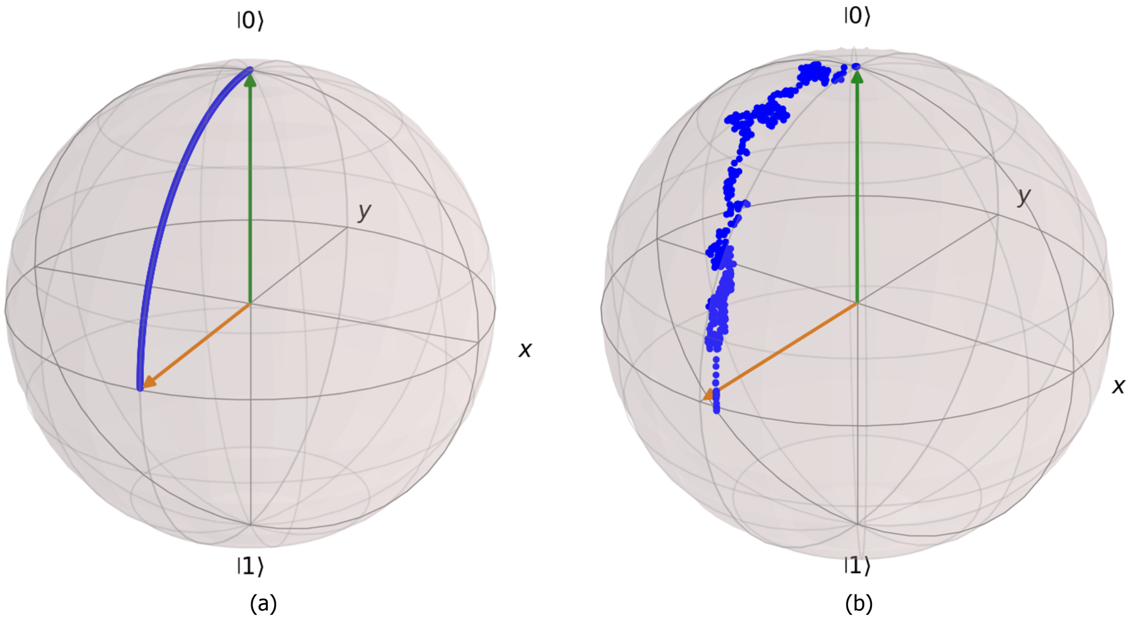


Figure 4.16: Initial state (green vector), target state (orange vector), and path of the quantum unitary (blue) in the absence of a drift Hamiltonian H_d, by **(a)** EO-GRAPE generated pulses and **(b)** RL generated pulses on the Bloch Sphere. Parameters: $U_T = R_x(\pi/2)$, $H_d = I_2$, $H_k = \{\sigma_x^1, \sigma_y^1\}$, $T_1 = 1000T$, $T_2 = 1000T$, $w_f = 1$, $w_e = 0$, $N_t = 500$, $N_g = 500$, $N_{QRLA} = 10,000$, $|\psi_i\rangle = |0\rangle$

In figure 4.17, the unitary path on the Bloch sphere by the control pulses generated by the EO-GRAPE algorithm (left) and the reinforcement learning agent (right) are shown in the presence of a drift Hamiltonian $H_d = \hbar\omega_1/2\hat{\sigma}_z$. The same initial state, target quantum unitary, and control Hamiltonian operators are used. One can see the effect of the drift Hamiltonian, causing the state vector to precess about the $\hat{z}$-axis at the qubit frequency ω_1.

Interestingly, one can see that the path induced by the control pulses generated by the EO-GRAPE algorithm first "overshoots" the target state, and then rotates back up to reach the target state, while the path induced by the control pulses generated by the reinforcement learning agent don't overshoot and arrive at the target state in one time. As mentioned in chapter 3, [4] suggests that the most energy efficient quantum unitary is equivalent to state vector travelling from the initial state to the target state via the geodesic between the two states on the Bloch sphere surface. In this scenario, the path induced by the reinforcement learning agent seems shorter than the path induced by the EO-GRAPE algorithm, suggesting that the energetic cost of the reinforcement learning agent control pulses are lower than the EO-GRAPE generated pulses in this specific scenario. To further investigate this, the relation between the energetic cost and the path length of the unitary path on the Bloch sphere has been explored. The results of these experiments are presented in the next sub section.

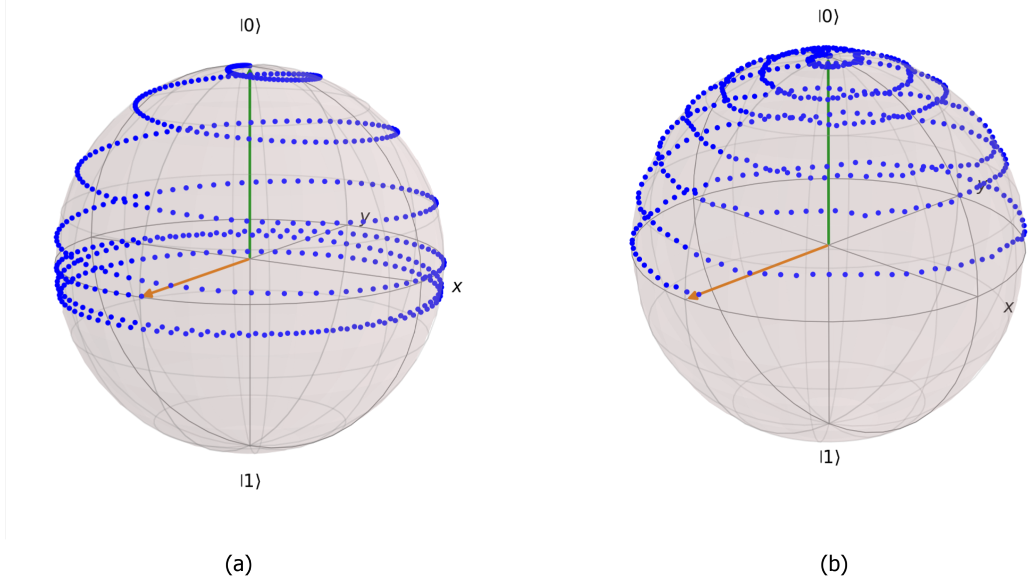


Figure 4.17: Initial state (green vector), target state (orange vector), and path of the quantum unitary (blue) in the presence of a drift Hamiltonian H_d, by **(a)** EO-GRAPE generated pulses and **(b)** RL generated pulses on the Bloch Sphere. Parameters: $U_T = R_x(\pi/2)$, $H_d = H_d^1$, $H_k = \{\sigma_x^1, \sigma_y^1\}$, $T_1 = 1000T$, $T_2 = 1000T$, $w_f = 1$, $w_e = 0$, $N_t = 500$, $N_g = 500$, $N_{QRLA} = 10{,}000$, $|\psi_i\rangle = |0\rangle$

4.4.2. Path Length versus Energetic Cost Correlation

In figure 4.18, the correlation between the energetic cost of the control pulses generated by the EO-GRAPE algorithm and the path length of the unitary path on the Bloch sphere is shown. The weights associated to fidelity w_f and the weights associated to energetic cost w_e are varied, for the same initial state, target unitary gate, and drift and control Hamiltonian operators.

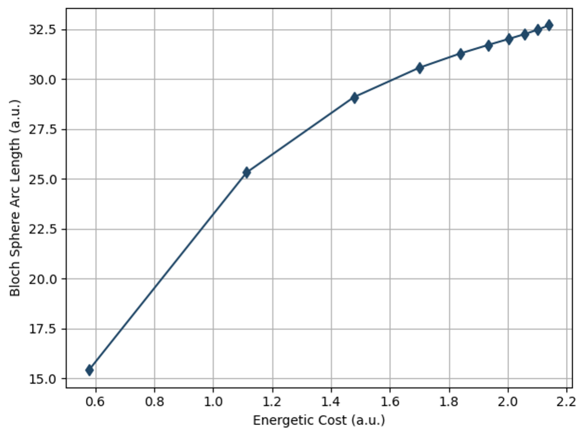


Figure 4.18: Correlation between energetic cost and path length of the path of the unitary on the Bloch Sphere for EO-GRAPE generated pulses. Parameters: $U_T = R_x(\pi/2)$, $H_d = H_d^1$, $H_k = \{\sigma_x^1, \sigma_y^1\}$, $T_1 = 1000T$, $T_2 = 1000T$, $w_f = [1, 0.1]$, $w_e = [0.1, 0.9]$, $N_t = 100$, $N_g = 500$, $|\psi_i\rangle = |0\rangle$

One can see that there is a positive proportional relationship between the energetic cost of the control pulses generated by the EO-GRAPE algorithm, and the path length of the unitary path on the Bloch sphere, confirming the theoretical framework presented by [4]. A higher energetic cost results in a longer path length of the unitary path of the Bloch sphere.

In figure 4.19, the correlation between the path length of the unitary path on the Bloch sphere and the energetic cost of the control pulses generated by the reinforcement learning agent are shown. As the reinforcement learning agent generates more random pulses and resulting in random paths, the error bars from 10 experiment repetitions is included.

The color coding indicates the average fidelity of the 10 experiments for that specific weight setting, again highlighting the trade-off between fidelity and energetic cost. With the control pulses generated by the reinforcement learning agent, we can again see a positive proportional relationship between the two metrics, confirming the theoretical framework presented by [4]. One can also see that in general, the path length of the unitary paths induced by the control pulses generated by the reinforcement learning agent are shorter than the path length of the unitary paths induced by the control pulses generated by the EO-GRAPE algorithm.

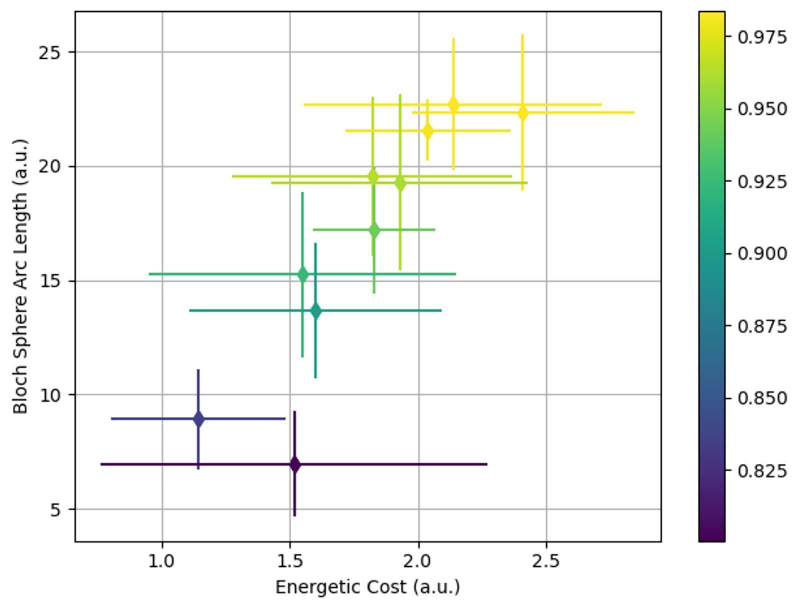


Figure 4.19: Correlation between energetic cost and path length of the path of the unitary on the Bloch Sphere for RL generated pulses, where the color coding indicates the average fidelity of the control pulse. Parameters: $U_T = R_x(\pi/2)$, $H_d = H_d^1$, $H_k = \{\sigma_x^1, \sigma_y^1\}$, $T_1 = 1000T$, $T_2 = 1000T$, $w_f = [1, 0.1]$, $w_e = [0.1, 0.9]$, $N_t = 100$, $N_{QRLA} = 10,000$, $|\psi_i\rangle = |0\rangle$

In figure 4.20, the correlation between the path length of the path on the Bloch sphere and the energetic cost of the control pulses of both EO-GRAPE generated pulses and RL generated pulses are shown (combination of figure 4.18 and 4.19).

One can see that for a weight setting of $w_e = 1$, $w_f = 0$, the pulses generated by the EO-GRAPE algorithm give a lower energetic cost due to the more structured nature of the optimizer. However, for a higher value of the energetic cost, the RL agent is able to find pulses with a much shorter path length than the EO-GRAPE generated pulses, probably because the RL agent is not restricted to keeping the pulse harmonics in shape.

As can be seen from the results presented in this subsection, the theory presented in chapter 2.4.2 agrees with our findings. The shorter the path length on the Bloch sphere, the lower the energetic cost required to implement a quantum unitary gate. Furthermore, the control pulses generated by the EO-GRAPE algorithm are taking some form of accessible geodesics, as described by the theory in section 2.4.2.

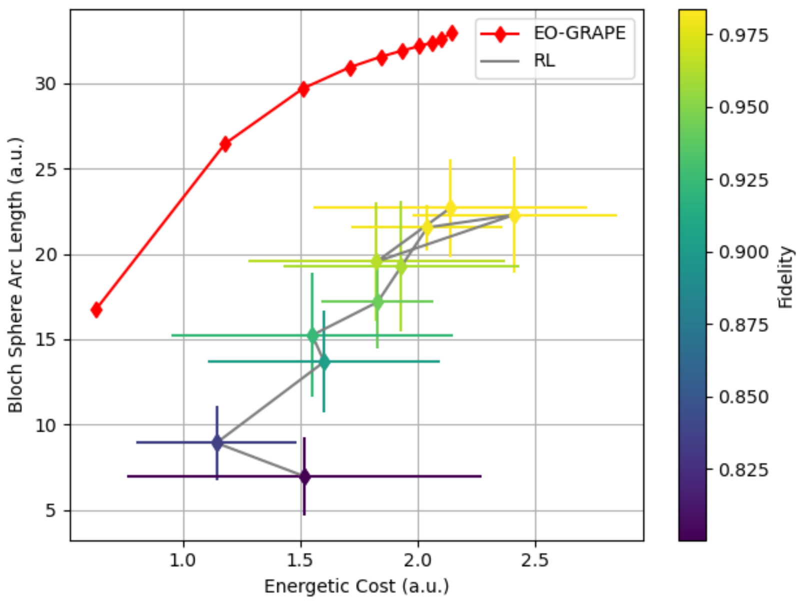


Figure 4.20: Combined plot of EO-GRAPE and RL generated pulses showing the correlation between energetic cost path length of the path of the unitary on the Bloch Sphere, where the color coding indicates the average fidelity of the control pulse. Parameters: $U_T = R_x(\pi/2)$, $H_d = H_d^1$, $H_k = \{\sigma_x^1, \sigma_y^1\}$, $T_1 = 1000T$, $T_2 = 1000T$, $w_f = [1, 0.1]$, $w_e = [0.1, 0.9]$, $N_t = 100$, $N_{QRLA} = 10,000$, $|\psi_i\rangle = |0\rangle$

5
Conclusion and Future Work

I never think of the future, it comes soon enough.
- Albert Einstein

5.1. Conclusion

With quantum processors and algorithms becoming increasingly complex, the need for high fidelity and coherent control of qubits has become monumental. The field of Quantum Optimal Control aims to design and implement electromagnetic field configurations that can effectively steer quantum processes at the atomic or molecular scale in the best way possible. This includes a large variety of methods, such as analytical, numerical, open-loop, closed-loop and gradient based or gradient free methods. Quantum Optimal Control spreads use cases in many other allied topics, such as quantum algorithms, quantum thermodynamics, or compilation and circuits. While there has been a large interest in studying quantum thermodynamics, little is still known about the energetic cost of quantum computational processes. In the context of the growing interest in achieving quantum advantage through energy efficiency, it appears to be crucial to understand energy efficiency in quantum operations, and how to optimize it.

This research therefore focuses on two main research questions: ”What is the energetic cost of a quantum unitary gate, and what is the relation between fidelity and energetic cost?”, and ”What Quantum Optimal Control strategies can we utilize to investigate and co-optimize a quantum unitary gate on both fidelity and energetic cost?”.

During this research we have answered all main and sub research questions. Regarding the main and sub research question 1 and 1.1, we have found that the energetic cost of implementing a quantum unitary gate through discrete pulse level control can be quantified through integrating the norm of the total Hamiltonian required to implement a certain quantum unitary gate over the total gate duration. In addition to this, we have found that this energetic cost positively correlates to the path length of the quantum unitary on the Bloch sphere, supporting the theory that the most energy-efficient way to implement a quantum unitary gate is through the geodesic between two quantum states. Looking at sub question 1.2, we have seen that there is a trade-off between the fidelity and the energetic cost required to implement a quantum unitary gate. We have seen that a decrease in energetic cost of 10 % yields an increase in infidelity of roughly 1 % in the low infidelity range. Regarding sub question 2.1, we developed a novel cost function and gradient for the Gradient Ascent Pulse Engineering Method, allowing for co-optimization of both the fidelity and energetic cost of a quantum unitary gate. This novel algorithm is the Energy Optimized Gradient Ascent Pulse Engineering algorithm, or EO-GRAPE. Next to the gradient-based open-loop quantum optimal control method, we have also investigated a learning-based, model-free, closed-loop method as described in sub question 2.2. A Deep Reinforcement Learning agent was developed to interact with a quantum environment, and to learn control pulses that minimize both the energetic cost as well as the infidelity. Regarding the final sub question 2.3, we have seen that both optimal control methods perform relatively well in low noisy systems.

However, when one decreases the relaxation and decoherence times of the qubits in the system, the EO-GRAPE algorithm outperforms the reinforcement learning agents for all noise settings and neural network sizes.

Theoretical research on how to quantify the energetic cost of a quantum unitary gate, as well as how we can adapt existing methods or create novel methods to co-optimize the energetic cost besides process fidelity has been done. In addition to this, a python software package has been developed called Energy Optimized Universal Quantum Optimal Control, or ”EUQOC”, that contains code to create quantum environments as well as code to run the optimization methods covered in this thesis.

The evaluation of the relation between fidelity and energetic cost, and the performance of both methods has been done by experiments sweeping the weight of the fidelity and the weight of the energetic cost, and by monitoring the fidelity and energetic cost of a universal set of gates as a function of increasing noise, respectively. Next to this, the relation between the path length of the quantum unitary path on the Bloch sphere and the energetic cost of a control pulse sequence has been investigated.

These experiments have shown that there exists a inversely proportional relation between the energetic cost and the infidelity (or error rate) of a quantum unitary gate. We have seen that a decrease in energetic cost of 10 % yield an increase in infidelity of roughly 1 % in the low infidelity range. Next to this, the experiments showed that while the reinforcement learning agent is able to learn how to devise energy efficient control pulses to implement a certain quantum unitary, the control pulses generated by the EO-GRAPE algorithm outperform the control pulses generated by the reinforcement learning agent for all noise settings and neural network sizes. Finally, a positive proportional relation between the path length of the quantum unitary on the Bloch sphere and the energetic cost of a control pulse sequence was observed, suggesting that the notion of energy efficiency and geodesics on the Bloch sphere is correct.

This work has shown that one can co-optimize a quantum unitary gate on energy efficiency as well as fidelity by using Quantum Optimal Control methods. We have seen that there exist a Pareto optimal front between the energetic cost and the fidelity of a quantum unitary gate. Next to this, the benchmarks have shown that the Energy Optimized Gradient Ascent Pulse Engineering method works best for the optimization problem, however it should be noted that after hyper parameter optimization and larger training cycles, it could be that the reinforcement learning agent’s performance dramatically increases. While the pulses generated by the algorithms are not suitable for physical quantum hardware, it does give researchers a framework and toolbox to investigate lower energy configurations of their control pulses and the effect is has on the fidelity of the quantum unitary gate.

In conclusion, the core contributions of this research are summarized below.

- Formulation of the energetic cost of implementing a quantum unitary gate using discrete control pulses
- Formulation of the gradient of the energetic cost with respect to all control parameters
- Development of a modified version of the Gradient Ascent Pulse Engineering algorithm to co-optimize the fidelity and energetic cost of a quantum unitary gate
- Identification of the trade-off between the fidelity and energetic cost of implementing a quantum unitary gate
- Verification of the theory describing the energetic cost of a quantum unitary gate as the path length of a quantum unitary gate on the Bloch sphere
- NumPy implementation of the QuTip Gradient Ascent Pulse Engineering algorithm
- Implementation of the Energy Optimized Gradient Ascent Pulse Engineering algorithm, including the novel gradient evaluation
- Integration of a QuTip Processor quantum simulator back-end and the EO-GRAPE and reinforcement learning agent classes
- Development of a Deep Reinforcement Learning agent able to learn and generate energy optimized control pulses for a universal set of quantum unitary gates
- Development of a Deep Reinforcement Learning agent able to learn and generate pulses based on EO-GRAPE generated control pulses

- Integration and mapping of a Reinforcement Learning agent policy between two different Reinforcement Learning agents
- Development of a Quantum Environment class containing all methods and arguments required for energy optimized quantum optimal control
- Development and investigation of methods to benchmark the performance of the EO-GRAPE algorithm and Deep Reinforcement Learning agents with increasing noise levels
- Investigation of the effect of increasing the neural network size on the Reinforcement Learning agent performance
- Verification of the correlation between the path length of a quantum unitary on the Bloch sphere and the energetic cost of implementing a quantum unitary gate for Reinforcement Learning generated control pulses

5.2. Future Work

In this section, several future work topics that emerged during this research will be introduced and explained. Three topics utilize the framework presented in this research to investigate theoretical applications closely related to this work. Three other topics investigate new methods or adjustments to the quantum optimal control methods introduced in this work to further explore and optimize the performance of these methods.

5.2.1. Information and Energy Relation

Information, entropy and energy are closely related quantities. Landauer's principle states that the minimum energy required by a logic operation will be the temperature times the entropy [65]. The entropy has been established by Claude E. Shannon, given in units of bits. As one can see, the information (bits) and energy of a control sequence are closely related [66]. Therefore a similar investigation as to the one performed in this research between fidelity and energy can be proposed between information and energy. One can investigate the co-optimization of both the information contained in a control sequence, as well as the energy of a control sequence, and see what the relation is between the two, and if it is possible to co-optimize, similar to fidelity and energy. In theory, one could change the fidelity part of the cost-function presented in this work by a measure for information, and repeat the same experiments that have been performed in this work.

5.2.2. Minimum Universal Controllability

Different formulations of the drift and control Hamiltonian operators have been investigated in this research. However, the effect of having multiple control operators has not been thoroughly researched. Minimizing the number of control operators per qubit to universally control the qubits is therefore a critical component in minimizing the energetic cost of a quantum unitary gate. We can therefore ask ourselves, given an N-qubit system, with drift Hamiltonian H_d, how many individual control terms H_k do we need to universally control the quantum system? There exists a theoretical framework to address this problem called the "Lie rank test", that states when an N-qubit system with control Hamiltonian H_d is fully controllable [67]. One could therefore use this theoretical framework to minimize the number of control operators needed per qubit and per drift Hamiltonian, and observe the effect that it has on the energetic cost of quantum unitary gates performed on the qubits.

5.2.3. Frequency Domain Optimization

The algorithms developed in this work generate and train on control pulses that are represented as 2D matrices with dimensions $(\mathrm{len}(H_k), N_t)$. If one has three control operators and 500 time steps, a control pulse already contains 1500 individual parameters that the algorithm and reinforcement learning agents need to adjust. However, as one can see from the pulses generated by the EO-GRAPE algorithm, the pulses can often be decomposed into individual $\sin$ or $\cos$ functions. It is therefore plausible one can transform the control pulses to the frequency domain by applying a Fourier transformation. This would allow one to represent a control pulse with originally 1500 parameters, to just a few amplitude and frequency parameters [68]. This would dramatically increase the computational efficiency of both algorithms, and could also increase the performance of the reinforcement learning agent, as it would automatically apply harmonic pulses instead of random block pulses.

5.2.4. Hyper Parameter Optimization

This work has primarily focused on using reinforcement learning as a method to investigate energy efficiency in quantum unitary gates, and has not focused on the theoretical framework behind reinforcement learning. We do believe however that tremendous steps can be made in the performance of the reinforcement learning agent by optimizing the hyper parameters of the agent based on theoretical or empirical results [69]. We therefore suggest a theoretical study and investigation of the reinforcement learning agent, accompanied by a hyper parameter optimization, to improve the performance of the reinforcement learning agent and compare it to the EO-GRAPE algorithm with increasing noise or decreasing relaxation and decoherence times.

5.2.5. Concept Learning for Universal Gate Set and Random Unitaries

During this research, we have investigated co-optimizing control pulses on both fidelity and energetic cost, for one specific universal gate set: (CNOT, Hadamard, T). However, one could also train a reinforcement learning agent to learn and create other energy optimal pulses that create new universal gate sets from a specific set of hardware restrictions, such as a specific waveform, set of frequencies, or bandwidth [70]. This removes the restriction of specific quantum gate synthesis, and allows the agent to experiment and try other random unitaries that together form a universal gate, and thereby any quantum computation, with a potentially lower energetic cost than using (CNOT, Hadamard, T).

5.2.6. Hardware Restricted Energy Optimized Quantum Gate Synthesis

The hardware restrictions imposed by both the qubit type and control electronics has not been taken into account in this work. However, extending this work by using the EO-GRAPE or RL algorithms to devise energy optimized control pulses for very specific quantum hardware will be very relevant. This work provides the theoretical framework and software tools in order to implement energy optimized quantum gate synthesis. By adjusting the drift Hamiltonian to include more terms and actual parameters based on hardware, and by restricting the EO-GRAPE algorithm, or punishing the RL agent to devise pulses according to the physical limitations of the control electronics, one can implement the theory and software tools provided in this work to devise very specific control pulses for very specific quantum hardware [71].

Appendix

Appendix 1: Quantum Environment Class

```python

class QuantumEnvironment(py_environment.PyEnvironment):

    def __init__(self, n_q, h_drift, h_control, labels, t_1, t_2, u_target, w_f = 1, w_e = 0,
     timesteps = 500, pulse_duration = 2 * np.pi, grape_iterations = 500, n_steps = 1,
    sweep_noise = False):

        """
        QuantumEnvironment Class
        ------------------------

        Create instance of a Quantum Processor with customizable Drift and Control
    Hamiltonian, Relaxation and Decoherence times for Pulse Level control

        Contains EO-GRAPE Algorithm (Energy Optimized Gradient Ascent Pulse Engineering)

        Parameters
        ----------
        n_q : int
            Number of qubits.
        h_drift : Qobj
            Drift Hamiltonian.
        h_control : list of Qobj
            List of Control Hamiltonians.
        t_1 : int
            Relaxation time of the Processor.
        t_2 : int
            Decoherence time of the Processor.
        initial_state : Qobj, optional
            Initial state of the quantum state.
        u_target : array, optional
            Target Unitary Evolution Operator.
        timesteps : int, optional
            Number of timesteps to discretize time. Default is 500.
        pulse_duration : float, optional
            Total pulse duration time. Default is 2pi
        grape_iterations : int, optional
            Number of GRAPE iterations. Default is 500
        """

        self.n_q = n_q
        self.h_drift = h_drift
        self.h_control = h_control
        self.labels = labels
        self.t_1 = t_1
        self.t_2 = t_2
        self.u_target = u_target
        self.timesteps = timesteps
        self.pulse_duration = pulse_duration
        self.grape_iterations = grape_iterations
        self.h_drift_numpy = fc.convert_qutip_to_numpy(h_drift)
        self.h_control_numpy = fc.convert_qutip_list_to_numpy(h_control)
        self.action_shape = (len(h_control), timesteps)
        self.state_shape = (2**(n_q), 2**(n_q))
        self.state_size = 2*(2**(2*n_q))
        self.current_step = 0
        self.reward_counter = 0
        self._episode_ended = False
        self.n_steps = n_steps
        self.w_f = w_f
        self.w_e = w_e
        self.fidelity_list = []
        self.reward_list = []
        self.energy_list = []
        self.sweep_noise = sweep_noise
        self.noise = None

        self.create_environment()

    def create_environment(self):

        """
        Create instance of a Qutip Processor as environment with custom
        Drift and Control Hamiltonians, T1, and T2 times
        """

        timespace = np.linspace(0, self.pulse_duration, self.timesteps)
        simulatortimespace = np.append(timespace, timespace[-1])

        targets = list(range(self.n_q))

        self.environment = Processor(N = self.n_q)

        self.environment.add_drift(self.h_drift, targets = targets)

        if self.sweep_noise == False:
            self.noise = [self.t_1, self.t_2]
            noise = RelaxationNoise(t1 = self.t_1, t2 = self.t_2)
            self.environment.add_noise(noise = noise)

        for operator in self.h_control:
            self.environment.add_control(operator, targets = targets)

        self.environment.set_all_tlist(simulatortimespace)

    def action_spec(self):

        """
        Returns the action spec
        """

        return array_spec.BoundedArraySpec(
            shape = (len(self.h_control) * self.timesteps,),
            dtype = np.float32,
            name = "pulses",
            minimum = -1,
            maximum = 1,
        )

    def observation_spec(self):

        """
        Returns the observation spec
        """

        return array_spec.BoundedArraySpec(
            shape = (2*(2**(2*self.n_q)),),
            dtype = np.float32,
            name = "density matrix",
            minimum = np.zeros(2*(2**(2*self.n_q)), dtype=np.float32),
            maximum = np.ones(2*(2**(2*self.n_q)), dtype = np.float32),
        )

    def _reset(self):
        """
        Resets the environment and returns the first timestep of a new episode
        """
        self.current_step = 0
        self._episode_ended = False
        self.initial_dm = self.initial_state * self.initial_state.dag()

        self.initial_dm_np = fc.convert_qutip_to_numpy(self.initial_dm)
        self.initial_dm_np_re = self.initial_dm_np.real
        self.initial_dm_np_im = self.initial_dm_np.imag
        self.initial_dm_np_re_flat = self.initial_dm_np_re.flatten()
        self.initial_dm_np_im_flat = self.initial_dm_np_im.flatten()
        self.combined_initial_dm = np.ndarray.astype(np.hstack((self.initial_dm_np_re_flat,
    self.initial_dm_np_im_flat)), dtype = np.float32)

        return ts.restart(self.combined_initial_dm)

    def _step(self, action):

        """
        Updates environment according to the action
        """

        action_2d = np.reshape(action, (len(self.h_control), self.timesteps))

        if self._episode_ended:

            return self._reset()

        if self.current_step < self.n_steps:

            next_state, fidelity = self.calculate_fidelity_reward(action_2d)

            energy = self.calculate_energetic_cost(action_2d)
            self.fidelity_list.append(fidelity)
            self.energy_list.append(energy)
            reward = self.w_f * fidelity + self.w_e * (1 - energy)
            self.reward_list.append(reward)

            terminal = False

            if self.current_step == self.n_steps - 1:

                terminal = True

        else:

            terminal = True
            reward = 0
            next_state = 0
        self.current_step += 1
        self.reward_counter += 1

        if terminal:
            self._episode_ended = True
            return ts.termination(next_state, reward)

        else:
            return ts.transition(next_state, reward)

    def run_pulses(self, pulses, plot_pulses = False):

        """
        Send pulses to a specific qutip processor instance and simulate result

        Parameters
        ----------
        pulses : array
            (K x I_G) Array containing amplitudes of operators in Control Hamiltonian.

        Returns
        -------
        result : Result instance of Environment using specified Initial State and Pulse set.
        """

        for i in range(len(pulses[:, 0])):
            self.environment.pulses[i].coeff = pulses[i]

```

```python
        result = self.environment.run_state(init_state = self.initial_state)
        density_matrix = result.states[-1]

        if plot_pulses == True:
            self.environment.plot_pulses()
            plt.show()

        return density_matrix

    def calculate_fidelity_reward(self, pulses, plot_result = False):

        """
        Calculates Fidelity Reward for a specific Qutip result.

        Parameters
        ----------
        Result : Result instance of Environment using specified Initial State and Pulse set.

        Returns
        -------

        combined_dm_sim_np_im_flat : Flattened and combined real and imaginary part of the density matrix after simulation

        r_f : Fidelity Reward for a specific Qutip result, Initial State, and Target Unitary.
        """

        for i in range(len(pulses[:, 0])):
            self.environment.pulses[i].coeff = pulses[i]

        self.result = self.environment.run_state(init_state = self.initial_state)

        dm_sim = self.result.states[-1]

        dm_sim_np = fc.convert_qutip_to_numpy(dm_sim)
        dm_sim_np_re = dm_sim_np.real
        dm_sim_np_im = dm_sim_np.imag
        dm_sim_np_re_flat = dm_sim_np_re.flatten()
        dm_sim_np_im_flat = dm_sim_np_im.flatten()
        combined_dm_re_im_flat = np.ndarray.astype(np.hstack((dm_sim_np_re_flat, dm_sim_np_im_flat)), dtype = np.float32)
        self.dm_target = (Qobj(self.u_target) * self.initial_state) * (Qobj(self.u_target) * self.initial_state).dag()

        r_f = fidelity(dm_sim, self.dm_target)

        if plot_result == True:

            vz.hinton(dm_sim, xlabels = [r'$\vert 00\rangle$', r'$\vert 01\rangle$', r'$\vert 10\rangle$', r'$\vert 11\rangle$'],
              ylabels = [r'$\vert 00\rangle$', r'$\vert 01\rangle$', r'$\vert 10\rangle$', r'$\vert 11\rangle$'])
            plt.show()

        return combined_dm_re_im_flat, r_f

    def calculate_energetic_cost(self, pulses, return_normalized = True):

        """
        Calculate Energetic Cost of certain set of Pulses

        Parameters
        ----------
        pulses : array
            (K x I_G) Array containing amplitudes of operators in Control Hamiltonian.

        Returns
        ---------

        return_value : float
            Energetic cost of the quantum operation
        """

        h_t_norm = []
        stepsize = self.pulse_duration/self.timesteps
        self.max_u_val = 2

        for i in range(self.timesteps - 1):
            h_t = 0
            for j in range(len(self.h_control)):
                h_t += pulses[j, i] * self.h_control_numpy[j]

            h_t_norm.append(np.linalg.norm(h_t))

        energetic_cost = np.sum(h_t_norm) * stepsize
        energetic_cost_normalized = energetic_cost / (self.pulse_duration * self.max_u_val *
    np.linalg.norm(np.sum(self.h_control_numpy)))

        if return_normalized == True:

            return_value = energetic_cost_normalized

        elif return_normalized == False:

            return_value = energetic_cost

        return return_value

    def grape_iteration(self, u, r, J, u_b_list, u_f_list, dt, eps_f, eps_e, w_f, w_e):

        """
        Perform one iteration of the GRAPE algorithm and update control pulse parameters

        Parameters
        ----------

        u : The generated control pulses with shape (iterations, controls, time)

        r : The number of this specific GRAPE iteration

        J : The number of controls in Control Hamiltonian

        u_b_list : Backward propagators of each time (length M)

        u_f_list : Forward propagators of each time (length M)

        dt : Timestep size

        eps_f : Distance to move along the gradient when updating controls for Fidelity

        eps_e : Distance to move along the gradient when updating controls for Energy

        w_f : Weight assigned to Fidelity part of the Cost Function

        w_e : Weight assigned to Energy part of the Cost Function

        Returns
        ---------

        u[r + 1, :, :] : The updated parameters
        """

        du_list = np.zeros((J, self.timesteps))
        max_du_list = np.zeros((J))

        for m in range(self.timesteps - 1):

            P = u_b_list[m] @ self.u_target

            for j in range(J):

                Q = 1j * dt * self.h_control_numpy[j] @ u_f_list[m]

                du_f = -2 * w_f * fc.overlap(P, Q) * fc.overlap(u_f_list[m], P)

                denom = self.h_drift_numpy.conj().T @ self.h_drift_numpy + u[r, j, m] * (self.h_drift_numpy.conj().T @ self.h_control_numpy[j] + self.h_control_numpy[j].conj().T @ self.h_drift_numpy)

                du_e = 0

                for k in range(J):

                    du_e += -1 * dt * w_e * (np.trace(self.h_drift_numpy.conj().T @ self.h_control_numpy[j] + self.h_control_numpy[j].conj().T @ self.h_drift_numpy) + np.trace(self.h_control_numpy[j].conj().T @ self.h_control_numpy[k] * (u[r, j, m] + u[r, k, m])))

                    denom += u[r, j, m] * u[r, k, m] * self.h_control_numpy[j].conj().T @ self.h_control_numpy[k]

                du_e /= (2 * np.trace(denom) ** (1/2))

                du_e_norm = du_e / (self.pulse_duration * (np.linalg.norm(self.h_drift_numpy) + np.linalg.norm(np.sum(self.h_control_numpy))))

                du_t = du_f + du_e_norm

                du_list[j, m] = du_t.real

                max_du_list[j] = np.max(du_list[j])

                u[r + 1, j, m] = u[r, j, m] + eps_f * du_f.real + eps_e * du_e_norm.real

        for j in range(J):
            u[r + 1, j, self.timesteps - 1] = u[r + 1, j, self.timesteps - 2]

        return max_du_list

    def run_grape_optimization(self, w_f, w_e, eps_f, eps_e):

        """
        Runs GRAPE algorithm and returns the control pulses, final unitary, Fidelity, and Energetic Cost for the Hamiltonian operators in H_Control
        so that the unitary U_target is realized

        Parameters
        ----------

        w_f : float
            Weight assigned to Fidelity part of the Cost Function

        w_e : float
            Weight assigned to Energetic Cost part of the Cost Functions

        eps_f : int
            Learning rate for fidelity

        eps_e : int
            Learning rate for energy

        Returns
        --------

        u : Optimized control pulses with dimension (iterations, controls, timesteps)

        u_f_list[-1] : Final unitary based on last GRAPE iteration

        du_max_per_iteration : Array containing the max gradient of each control for all GRAPE iterations

        cost_function : Array containing the value of the cost function for all GRAPE iterations

```

```python
        infidelity_array : Array containing the infidelity for all GRAPE iterations

        energy_array : Array containing the energetic cost for all GRAPE iterations

        """

        self.w_f = w_f

        self.w_e = w_e

        times = np.linspace(0, self.pulse_duration, self.timesteps)

        if eps_f is None:
            eps_f = 0.1 * (self.pulse_duration) /(times[-1])

        if eps_e is None:
            eps_e = 0.1 * (self.pulse_duration) / (times[-1])

        M = len(times)
        J = len(self.h_control_numpy)

        u = np.zeros((self.grape_iterations, J, M))

        self.du_max_per_iteration = np.zeros((self.grape_iterations - 1, J))

        self.cost_function_array = []
        self.infidelity_array = []
        self.energy_array = []

        with alive_bar(self.grape_iterations - 1) as bar:

            for r in range(self.grape_iterations - 1):

                bar()
                dt = times[1] - times[0]

                def _H_idx(idx):
                    return self.h_drift_numpy + sum([u[r, j, idx] * self.h_control_numpy[j] for j in range(J)])

                u_list = [expm(-1j * _H_idx(idx) * dt) for idx in range(M - 1)]

                u_f_list = []
                u_b_list = []

                u_f = np.eye(*(self.u_target.shape))
                u_b = np.eye(*(self.u_target.shape))

                for n in range(M - 1):

                    u_f = u_list[n] @ u_f
                    u_f_list.append(u_f)

                    u_b_list.insert(0, u_b)
                    u_b = u_list[M - 2 - n].T.conj() @ u_b

                self.du_max_per_iteration[r] = self.grape_iteration(u, r, J, u_b_list, u_f_list, dt, eps_f, eps_e, w_f, w_e)

                cost_function = w_f * (1 - fc.Calculate_Fidelity(self.u_target, u_f_list[-1])) + w_e * self.calculate_energetic_cost(u[r])
                self.cost_function_array.append(cost_function)
                self.infidelity_array.append(1 - fc.Calculate_Fidelity(self.u_target, u_f_list[-1]))
                self.energy_array.append(self.calculate_energetic_cost(u[r]))
                self.final_unitary = u_f_list[-1]

        return u[-1]

    def plot_grape_pulses(self, pulses):
        """
        Plot the pulses generated by the EO-GRAPE Algorithm

        Parameters
        ----------

        pulses : Pulses generated by the EO-GRAPE algorithm

        labels : Labels for the operators in h_control
        """

        time = np.linspace(0, self.pulse_duration, self.timesteps)

        colors = ["blue", "orange", "green"]

        fig, ax = plt.subplots(len(self.h_control_numpy))

        if len(self.h_control_numpy) == 1:

            ax.plot(time, pulses[0, :], label = f"{self.labels[0]}", color = colors[0])
            ax.set(xlabel = "Time", ylabel = f"{self.labels[0]}")

        else:

            for i in range(len(self.h_control_numpy)):

                ax[i].plot(time, pulses[i, :], label = f"{self.labels[i]}", color = colors[i])
                ax[i].set(xlabel = "Time", ylabel = f"{self.labels[i]}")

        plt.subplot_tool()
        plt.show()

    def plot_rl_pulses(self, pulses):

        """
        Plot the pulses generated by the QRLAgent

        Parameters
        ----------

        pulses : Pulses generated by the QRLAgent

        labels : Labels for the operators in h_control
        """

        time = np.linspace(0, self.pulse_duration, self.timesteps)

        colors = ['#03080c','#214868', '#5b97ca']

        fig, ax = plt.subplots(len(self.h_control_numpy))

        if len(self.h_control_numpy) == 1:

            ax.axhline(y = 0, color = "grey", ls = "dashed")
            ax.step(time, pulses[0, :], label = f"{self.labels[0]}", color = colors[0])
            ax.set(xlabel = "Time", ylabel = f"{self.labels[0]}")
            ax.legend()

        else:

            for i in range(len(self.h_control_numpy)):

                ax[i].axhline(y = 0, color = "grey", ls = "dashed")
                ax[i].step(time, pulses[i, :], label = f"{self.labels[i]}", color = colors[i])
                ax[i].set(xlabel = "Time", ylabel = f"{self.labels[i]}")
                ax[i].legend()

        fig.suptitle("Final Pulse Generated by the QRLAgent")
        fig.tight_layout()

        plt.subplot_tool()
        plt.show()

    def plot_tomography(self):

        """
        Plot the tomography of the target unitary and the unitary realized by the EO-GRAPE algorithm

        Parameters
        ----------

        final_unitary : The final unitary obtained by the EO-GRAPE algorithm
        """

        op_basis = [[Qobj(identity(2)), Qobj(sigmax()), Qobj(sigmay()), Qobj(sigmaz())]] * 2
        op_label = [["I", "X", "Y", "Z"]] * 2

        u_i_s = to_super(Qobj(self.u_target))
        u_f_s = to_super(Qobj(self.final_unitary))
        chi_1 = qpt(u_i_s, op_basis)
        chi_2 = qpt(u_f_s, op_basis)

        fig_1 = plt.figure(figsize = (6,6))
        fig_1 = qpt_plot_combined(chi_1, op_label, fig=fig_1, threshold=0.001, title = 'Target Unitary Gate ')

        fig_2 = plt.figure(figsize = (6, 6))
        fig_2 = qpt_plot_combined(chi_2, op_label, fig = fig_2, threshold = 0.001, title = 'Final Unitary after Optimization')

        plt.show()

    def plot_du(self):

        """
        Plot the max gradient over all timesteps per control operator as function of GRAPE iterations

        Parameters
        ----------

        du_list : The list containing gradient values from the EO-GRAPE algorithm

        labels : The labels associated to the operators in h_control
        """

        iteration_space = np.linspace(1, self.grape_iterations - 1, self.grape_iterations - 1)

        for i in range(len(self.h_control_numpy)):
            plt.plot(iteration_space, self.du_max_per_iteration[:, i], label = f"{self.labels[i]}")
        plt.axhline(y = 0, color = "black", linestyle = "-")
        plt.xlabel("GRAPE Iteration Number")
        plt.ylabel("Maximum Gradient over Time")
        plt.title("Maximum Gradient over Time vs. GRAPE Iteration Number")
        plt.legend()
        plt.grid()
        plt.show()

    def plot_cost_function(self):

        """
        Plot the value of the cost function as function of GRAPE iterations

        Parameters
        ----------

        cost_fn : The array containing cost function values obtained by the EO−GRAPE
    algorithm

        infidelity : The array containing the infidelity values obtained by the EO−GRAPE
    algorithm

        energy : The array containing the energetic cost values obtained by the EO−GRAPE
    algorithm

        w_f : The weight assigned to the fidelity part of the cost function

        w_e : The weight assigned to the energetic cost part of the cost function
        """

        iteration_space = np.linspace(1, self.grape_iterations − 1, self.grape_iterations −
    1)

        plt.plot(iteration_space, self.cost_function_array, label = f"Cost Function, $w_f$ =
    {self.w_f}, $w_e$ = {self.w_e}")
        plt.plot(iteration_space, self.infidelity_array, label = f"Infidelity (1−F)")
        plt.plot(iteration_space, self.energy_array, label = f"Normalized Energetic Cost")
        plt.axhline(y = 0, color = "black", linestyle = "−")
        plt.xlabel("GRAPE iteration number")
        plt.ylabel("Cost Function")
        plt.title("Cost Function, Infidelity, and Energetic Cost vs. GRAPE Iteration Number")
        plt.legend()
        plt.grid()
        plt.show()

    def convert_dm_to_coordinates(dm):

        a = dm[0, 0]
        b = dm[1, 0]

        x = 2 ∗ b.real
        y = 2 ∗ b.imag
        z = 2 ∗ a − 1

        return x, y, z

    def calc_arc_length(n1, n2):

        return np.arccos(np.dot(n1, n2))

    def get_total_arc_length(self):

        density_matrix_list = fc.convert_qutip_list_to_numpy(self.result.states)

        x_coordinate_list = []
        y_coordinate_list = []
        z_coordinate_list = []

        for i in range(len(density_matrix_list)):
            dm = density_matrix_list[i]
            a = dm[0, 0]
            b = dm[1, 0]
            x = 2 ∗ b.real
            y = 2 ∗ b.imag
            z = 2 ∗ a − 1
            x_coordinate_list.append(x)
            y_coordinate_list.append(y)
            z_coordinate_list.append(z)

        coordinate_matrix = np.vstack((x_coordinate_list, y_coordinate_list,
    z_coordinate_list)).real

        arc_length_list = []

        for i in range(len(coordinate_matrix[0]) − 1):

            argument = np.linalg.norm(np.cross(coordinate_matrix[:, i], coordinate_matrix[:, i + 1]))/(np.dot(coordinate_matrix[:, i], coordinate_matrix[:, i + 1]))

            arclen_tan = np.arctan(argument)

            arc_length_list.append(arclen_tan)

        total_arc_length = np.sum(arc_length_list)

        return total_arc_length

    def plot_bloch_sphere_trajectory(self):

        density_matrix_list = fc.convert_qutip_list_to_numpy(self.result.states)

        x_coordinate_list = []
        y_coordinate_list = []
        z_coordinate_list = []

        for i in range(len(density_matrix_list)):
            dm = density_matrix_list[i]
            a = dm[0, 0]
            b = dm[1, 0]
            x = 2 * b.real
            y = 2 * b.imag
            z = 2 * a - 1
            x_coordinate_list.append(x)
            y_coordinate_list.append(y)
            z_coordinate_list.append(z)

        points = [x_coordinate_list, y_coordinate_list, z_coordinate_list]

        target_state = (Qobj(self.u_target) * self.initial_state)

        bsphere = Bloch()

        bsphere.make_sphere()

        bsphere.add_states(self.initial_state)

        bsphere.add_states(target_state)

        bsphere.add_points(points)

        bsphere.render()

        plt.show()
```

Appendix 2: EO-GRAPE Approximation Class

```
class GRAPEApproximation(py_environment.PyEnvironment):

    def __init__(self, n_q, h_drift, h_control, labels, u_target, w_f = 1, w_e = 0, eps_f =
    1, eps_e = 100, timesteps = 500, pulse_duration = 2 * np.pi, grape_iterations = 500,
    n_steps = 1):

        self.n_q = n_q
        self.h_drift = h_drift
        self.h_control = h_control
        self.labels = labels
        self.u_target = u_target
        self.w_f = w_f
        self.w_e = w_e
        self.eps_f = eps_f
        self.eps_e = eps_e
        self.timesteps = timesteps
        self.pulse_duration = pulse_duration
        self.grape_iterations  = grape_iterations
        self.h_drift_numpy = fc.convert_qutip_to_numpy(h_drift)
        self.h_control_numpy = fc.convert_qutip_list_to_numpy(h_control)
        self.current_step = 0
        self.reward_counter = 0
        self._episode_ended = False
        self.n_steps = n_steps
        self.difference_list = []
        self.reward_list = []

        self.target_pulse = self.run_grape_optimization()

    def action_spec(self):

        return array_spec.BoundedArraySpec(
            shape = (len(self.h_control) * self.timesteps,),
            dtype = np.float32,
            name = "pulse",
            minimum = -10,
            maximum = 10,
        )

    def observation_spec(self):

        return array_spec.BoundedArraySpec(
            shape = (2*(2**(2*self.n_q)),),
            dtype = np.float32,
            name = "unitary",
            minimum = np.zeros(2*(2**(2*self.n_q)), dtype=np.float32),
            maximum = np.ones(2*(2**(2*self.n_q)), dtype = np.float32),
        )

    def _reset(self):

        self.current_step = 0
        self._episode_ended = False
        self.initial_unitary = identity(2**self.n_q)
        self.initial_unitary_np = fc.convert_qutip_to_numpy(self.initial_unitary)
        self.initial_unitary_np_re = self.initial_unitary_np.real
        self.initial_unitary_np_im = self.initial_unitary_np.imag
        self.initial_unitary_np_re_flat = self.initial_unitary_np_re.flatten()
        self.initial_unitary_np_im_flat = self.initial_unitary_np_im.flatten()
        self.combined_initial_unitary = np.ndarray.astype(np.hstack((self.
    initial_unitary_np_re_flat, self.initial_unitary_np_im_flat)), dtype = np.float32)

        return ts.restart(self.combined_initial_unitary)

    def _step(self, action):

        if self._episode_ended:

            return self._reset()
```

```python

        if self.current_step < self.n_steps:

            next_state, reward = self.calc_unitary_and_reward(action)
            self.reward_list.append(reward)

            terminal = False

            if self.current_step == self.n_steps - 1:

                terminal = True

        else:
            terminal = True
            reward = 0
            next_state = 0
        self.current_step += 1
        self.reward_counter += 1

        if terminal:
            self._episode_ended = True
            return ts.termination(next_state, reward)

        else:
            return ts.transition(next_state, reward)

    def grape_iteration(self, u, r, J, u_b_list, u_f_list, dt):

        """
        Perform one iteration of the GRAPE algorithm and update control pulse parameters

        Parameters
        ----------

        u : The generated control pulses with shape (iterations, controls, time)

        r : The number of this specific GRAPE iteration

        J : The number of controls in Control Hamiltonian

        u_b_list : Backward propagators of each time (length M)

        u_f_list : Forward propagators of each time (length M)

        dt : Timestep size

        eps_f : Distance to move along the gradient when updating controls for Fidelity

        eps_e : Distance to move along the gradient when updating controls for Energy

        w_f : Weight assigned to Fidelity part of the Cost Function

        w_e : Weight assigned to Energy part of the Cost Function

        Returns
        --------

        u[r + 1, :, :] : The updated parameters
        """

        du_list = np.zeros((J, self.timesteps))
        max_du_list = np.zeros((J))

        for m in range(self.timesteps - 1):

            P = u_b_list[m] @ self.u_target

            for j in range(J):

                Q = 1j * dt * self.h_control_numpy[j] @ u_f_list[m]

                du_f = -2 * self.w_f * fc.overlap(P, Q) * fc.overlap(u_f_list[m], P)

                denom = self.h_drift_numpy.conj().T @ self.h_drift_numpy + u[r, j, m] * (self.h_drift_numpy.conj().T @ self.h_control_numpy[j] + self.h_control_numpy[j].conj().T @ self.h_drift_numpy)

                du_e = 0

                for k in range(J):

                    du_e += -1 * dt * self.w_e * (np.trace(self.h_drift_numpy.conj().T @ self.h_control_numpy[j] + self.h_control_numpy[j].conj().T @ self.h_drift_numpy) + np.trace(self.h_control_numpy[j].conj().T @ self.h_control_numpy[k] * (u[r, j, m] + u[r, k, m])))

                    denom += u[r, j, m] * u[r, k, m] * self.h_control_numpy[j].conj().T @ self.h_control_numpy[k]

                du_e /= (2 * np.trace(denom) ** (1/2))

                du_e_norm = du_e / (self.pulse_duration * (np.linalg.norm(self.h_drift_numpy) + np.linalg.norm(np.sum(self.h_control_numpy))))

                du_t = du_f + du_e_norm

                du_list[j, m] = du_t.real

                max_du_list[j] = np.max(du_list[j])

                u[r + 1, j, m] = u[r, j, m] + self.eps_f * du_f.real + self.eps_e * du_e_norm.real

        for j in range(J):
            u[r + 1, j, self.timesteps - 1] = u[r + 1, j, self.timesteps - 2]

        return max_du_list

    def run_grape_optimization(self):

        """
        Runs GRAPE algorithm and returns the control pulses, final unitary, Fidelity, and Energetic Cost for the Hamiltonian operators in H_Control
        so that the unitary U_target is realized

        Parameters
        ----------

        w_f : float
            Weight assigned to Fidelity part of the Cost Function

        w_e : float
            Weight assigned to Energetic Cost part of the Cost Functions

        eps_f : int
            Learning rate for fidelity

        eps_e : int
            Learning rate for energy

        Returns
        -------

        u : Optimized control pulses with dimension (iterations, controls, timesteps)

        u_f_list[-1] : Final unitary based on last GRAPE iteration

        du_max_per_iteration : Array containing the max gradient of each control for all GRAPE iterations

        cost_function : Array containing the value of the cost function for all GRAPE iterations

        infidelity_array : Array containing the infidelity for all GRAPE iterations

        energy_array : Array containing the energetic cost for all GRAPE iterations

        """

        times = np.linspace(0, self.pulse_duration, self.timesteps)

        if self.eps_f is None:
            eps_f = 0.1 * (self.pulse_duration) /(times[-1])

        if self.eps_e is None:
            eps_e = 0.1 * (self.pulse_duration) / (times[-1])

        M = len(times)
        J = len(self.h_control_numpy)

        u = np.zeros((self.grape_iterations, J, M))

        self.du_max_per_iteration = np.zeros((self.grape_iterations - 1, J))

        self.cost_function_array = []
        self.infidelity_array = []
        self.energy_array = []

        with alive_bar(self.grape_iterations - 1) as bar:

            for r in range(self.grape_iterations - 1):

                bar()
                dt = times[1] - times[0]

                def _H_idx(idx):
                    return self.h_drift_numpy + sum([u[r, j, idx] * self.h_control_numpy[j] for j in range(J)])

                u_list = [expm(-1j * _H_idx(idx) * dt) for idx in range(M - 1)]

                u_f_list = []
                u_b_list = []

                u_f = np.eye(*(self.u_target.shape))
                u_b = np.eye(*(self.u_target.shape))

                for n in range(M - 1):

                    u_f = u_list[n] @ u_f
                    u_f_list.append(u_f)

                    u_b_list.insert(0, u_b)
                    u_b = u_list[M - 2 - n].T.conj() @ u_b

                self.du_max_per_iteration[r] = self.grape_iteration(u, r, J, u_b_list, u_f_list, dt)

                cost_function = self.w_f * (1 - fc.Calculate_Fidelity(self.u_target, u_f_list[-1])) + self.w_e * self.calculate_energetic_cost(u[r])
                self.cost_function_array.append(cost_function)
                self.infidelity_array.append(1 - fc.Calculate_Fidelity(self.u_target, u_f_list[-1]))
                self.energy_array.append(self.calculate_energetic_cost(u[r]))
                self.final_unitary = u_f_list[-1]

        return u[-1]

    def calculate_energetic_cost(self, pulses, return_normalized = False):

        """
        Calculate Energetic Cost of certain set of Pulses

        Parameters
        ----------
        pulses : array
            (K x I_G) Array containing amplitudes of operators in Control Hamiltonian.

        Returns
        ---------

        return_value : float
            Energetic cost of the quantum operation
        """

        h_t_norm = []
        stepsize = self.pulse_duration/self.timesteps

        for i in range(self.timesteps - 1):
            h_t = 0
            for j in range(len(self.h_control)):
                h_t += pulses[j, i] * self.h_control_numpy[j]

            h_t_norm.append(np.linalg.norm(h_t))

        energetic_cost = np.sum(h_t_norm) * stepsize
        energetic_cost_normalized = energetic_cost / (self.pulse_duration * np.linalg.norm(np
.sum(self.h_control_numpy)))

        if return_normalized == True:

            return_value = energetic_cost_normalized

        elif return_normalized == False:

            return_value = energetic_cost

        return return_value

    def calc_unitary_and_reward(self, action):

        times = np.linspace(0, self.pulse_duration, self.timesteps)

        action_2d = np.reshape(action, (len(self.h_control), self.timesteps))

        def _H_idx(idx):
            return self.h_drift_numpy + sum([action_2d[j, idx] * self.h_control_numpy[j] for
j in range(len(self.h_control_numpy))])

        dt = times[1] - times[0]

        u_list = [expm(-1j * _H_idx(idx) * dt) for idx in range(len(times) - 1)]

        u_f_list = []

        u_f = np.eye(*(self.u_target.shape))

        for n in range(len(times) - 1):

            u_f = u_list[n] @ u_f
            u_f_list.append(u_f)

        final_unitary = u_f_list[-1]
        final_unitary_re = final_unitary.real
        final_unitary_im = final_unitary.imag
        final_unitary_re_flat = final_unitary_re.flatten()
        final_unitary_im_flat = final_unitary_im.flatten()
        combined_final_unitary = np.ndarray.astype(np.hstack((final_unitary_re_flat,
final_unitary_im_flat)), dtype = np.float32)

        reward = self.find_sq_diff(action)

        return combined_final_unitary, reward

    def find_sq_diff(self, action):
```

```python

        target_flat = self.target_pulse.flatten()

        diff = target_flat - action
        diff_sq = np.square(diff)
        tot_diff = -1 * np.sum(diff_sq)

        return tot_diff

    def plot_grape_pulses(self, pulses):

        """
        Plot the pulses generated by the EO-GRAPE Algorithm

        Parameters
        ----------

        pulses : Pulses generated by the EO-GRAPE algorithm

        labels : Labels for the operators in h_control
        """

        time = np.linspace(0, self.pulse_duration, self.timesteps)

        colors = ["blue", "orange", "green"]

        fig, ax = plt.subplots(len(self.h_control_numpy))

        if len(self.h_control_numpy) == 1:

            ax.plot(time, pulses[0, :], label = f"{self.labels[0]}", color = colors[0])
            ax.set(xlabel = "Time", ylabel = f"{self.labels[0]}")

        else:

            for i in range(len(self.h_control_numpy)):

                ax[i].plot(time, pulses[i, :], label = f"{self.labels[i]}", color = colors[i])
                ax[i].set(xlabel = "Time", ylabel = f"{self.labels[i]}")

        plt.subplot_tool()
        plt.show()
```

Appendix 3: Quantum Reinforcement Learning Class

```
class QuantumRLAgent:

    def __init__(self, TrainEnvironment, EvaluationEnvironment, num_iterations, w_f, w_e,
    num_cycles = 1, fc_layer_params = (100, 100, 100), learning_rate = 1e-3,
    collect_episodes_per_iteration = 1, eval_interval = 1, replay_buffer_capacity = 10,
    policy = None, rand_initial_state = True, sweep_noise = False, noise_level = "Low",
    initial_state = basis(4,2)):

        """
        QuantumRLAgent Class
        ---------------------


        Create instance of a reinforcement learning agent interacting with a
    QuantumEnvironment Instance

        Parameters
        ----------

        TrainEnvironment : class
            Quantum Environment Instance

        EvaluationEnvironment : class
            Quantum Environment Instance

        num_iterations : int
            Number of training loop iterations
        """

        self.env_train_py = TrainEnvironment
        self.env_eval_py = EvaluationEnvironment
        self.num_cycles = num_cycles
        self.env_train_py.n_steps = self.num_cycles
        self.env_eval_py.n_steps = self.num_cycles
        self.num_iterations = num_iterations
        self.fc_layer_params = fc_layer_params
        self.learning_rate = learning_rate
        self.collect_episodes_per_iteration = collect_episodes_per_iteration
        self.eval_interval = eval_interval
        self.replay_buffer_capacity = replay_buffer_capacity
        self.policy = policy
        self.rand_initial_state = rand_initial_state
        self.initial_state = initial_state
        self.env_train_py.w_f = w_f
        self.env_eval_py.w_f = w_f
        self.env_train_py.w_e = w_e
        self.env_eval_py.w_e = w_e
        self.sweep_noise = sweep_noise
        self.noise_level = noise_level

        self.create_network_agent(policy = policy)

    def create_network_agent(self, policy = None):

        """
        Create Neural Network and Agent Instance based on Quantum Environment Class
        """

        self.train_env = tf_py_environment.TFPyEnvironment(self.env_train_py)
        self.eval_env = tf_py_environment.TFPyEnvironment(self.env_eval_py)

        self.actor_net = actor_distribution_network.ActorDistributionNetwork(
            self.train_env.observation_spec(),
            self.train_env.action_spec(),
            fc_layer_params = self.fc_layer_params
        )

        self.optimizer = keras.optimizers.Adam(learning_rate = self.learning_rate)

        self.train_step_counter = tf.compat.v2.Variable(0)

        self.tf_agent = reinforce_agent.ReinforceAgent(
            self.train_env.time_step_spec(),
            self.train_env.action_spec(),
            actor_network = self.actor_net,
            optimizer = self.optimizer,
            normalize_returns = True,
            train_step_counter = self.train_step_counter
        )

        self.tf_agent.initialize()

        if policy is None:

            self.eval_policy = self.tf_agent.policy
            self.collect_policy = self.tf_agent.collect_policy

        else:
            self.eval_policy = tf.compat.v2.saved_model.load(policy)
            self.collect_policy = self.tf_agent.collect_policy


        self.replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
            data_spec = self.tf_agent.collect_data_spec,
            batch_size = self.train_env.batch_size,
            max_length = self.replay_buffer_capacity
        )

        self.eval_replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
            data_spec = self.tf_agent.collect_data_spec,
            batch_size = self.eval_env.batch_size,
            max_length = self.num_cycles + 1
        )

        self.avg_return = tf_metrics.AverageReturnMetric()

        self.eval_observers = [self.avg_return, self.eval_replay_buffer.add_batch]
        self.eval_driver = dynamic_episode_driver.DynamicEpisodeDriver(
            self.eval_env,
            self.eval_policy,
            self.eval_observers,
            num_episodes = 1
        )

        self.train_observers = [self.replay_buffer.add_batch]
        self.train_driver = dynamic_episode_driver.DynamicEpisodeDriver(
            self.train_env,
            self.collect_policy,
            self.train_observers,
            num_episodes = self.collect_episodes_per_iteration
        )

        self.tf_agent.train = common.function(self.tf_agent.train)

    def run_training(self, save_episodes = True, clear_buffer = False):

        """
        Starts training on Quantum RL Agent

        Parameters
        ----------

        save_episodes : bool : True
            Saves episodes if set to True

        clear_buffer : bool : False
            Clears buffer each episode if set to True
        """

        noise_low = np.linspace(start = 200, stop = 10, num = self.num_iterations) * self.
    env_eval_py.pulse_duration
```

```
        noise_high = np.linspace(start = 10, stop = 0.01, num = self.num_iterations) * self.
    env_eval_py.pulse_duration

        if self.noise_level == "Low":

            self.noise = noise_low

        elif self.noise_level == "High":

            self.noise = noise_high

        self.return_list = []
        self.episode_list = []
        self.iteration_list = []


        with trange(self.num_iterations, dynamic_ncols = False) as t:

            for i in t:

                if self.rand_initial_state == True:

                    new_initial_state = rand_ket(2**self.env_train_py.n_q)

                else:

                    new_initial_state = self.initial_state

                self.env_train_py.initial_state = new_initial_state
                self.env_eval_py.initial_state = new_initial_state

                if (i % 100 == 0):

                    if self.sweep_noise == True:

                        noise = RelaxationNoise(t1 = self.noise[i], t2 = self.noise[i])
                        self.env_train_py.noise = [self.noise[i], self.noise[i]]
                        self.env_eval_py.noise = [self.noise[i], self.noise[i]]
                        self.env_train_py.environment.add_noise(noise = noise)
                        self.env_eval_py.environment.add_noise(noise = noise)

                t.set_description(f"Episode {i}")

                if clear_buffer:
                    self.replay_buffer.clear()

                final_time_step, policy_state = self.train_driver.run()
                experience = self.replay_buffer.gather_all()
                train_loss = self.tf_agent.train(experience)

                if i % self.eval_interval == 0 or i == self.num_iterations - 1:

                    self.avg_return.reset()
                    final_time_step, policy_state = self.eval_driver.run()

                    self.iteration_list.append(self.tf_agent.train_step_counter.numpy())
                    self.return_list.append(self.avg_return.result().numpy())

                    t.set_postfix({"Return" : self.return_list[-1]})

                    if save_episodes:
                        self.episode_list.append(self.eval_replay_buffer.gather_all())

    def plot_fidelity_return_per_episode(self):

        """
        Plots the average fidelity and return per episode versus number of episode
        """

        avg_eval_reward_per_episode = []

```

```python
        for i in range(self.num_iterations):
            sum_eval = np.sum(self.env_eval_py.fidelity_list[self.num_cycles * i : self.num_cycles + self.num_cycles * i])
            avg_eval_reward_per_episode.append(sum_eval/self.num_cycles)

        fig, ax1 = plt.subplots()
        ax2 = ax1.twinx()
        ax2.axhline(y = 0, color = "grey")
        ax1.plot(self.iteration_list, avg_eval_reward_per_episode, label = "Fidelity", marker = "d", color = '#214868', markevery = 20)
        ax2.plot(self.iteration_list, self.return_list, label = "Return", marker = "d", color = "#5b97ca", markevery = 20)
        ax1.set_ylabel("Fidelity")
        ax2.set_ylabel("Return")
        ax1.set_xlabel("Episode number")
        ax1.legend(loc = (0.7, 0.45))
        ax2.legend(loc = (0.7, 0.55))
        fig.tight_layout()

        plt.show()

    def plot_fidelity_energy_reward_per_iteration(self):

        """
        Plots fidelity per iteration
        """

        self.iteration_space = np.linspace(1, self.num_cycles * self.num_iterations, self.num_cycles * self.num_iterations)
        fig, ax1 = plt.subplots()
        ma_fid = self.moving_average(self.env_eval_py.fidelity_list)
        ma_energy = self.moving_average(self.env_eval_py.energy_list)
        ma_iteration_space = np.arange(len(ma_fid))
        np.save('RL_Fidelity_Noise_RAW', self.env_eval_py.fidelity_list)
        np.save('RL_Energy_Noise_RAW', self.env_eval_py.energy_list)
        np.save('RL_Fidelity_List_MA', ma_fid)
        np.save('RL_Energy_List_MA', ma_energy)
        ax2 = ax1.twiny()
        ax2.axhline(y = 0, color = "grey")
        ax2.set_xticks([0, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000], ['100 T', '90 T', '80 T', '70 T', '60 T', '50 T', '40 T', '30 T', '20 T', '10 T', '0T'])
        ax2.set_xlabel("Decoherence Time (T1, T2)")
        ax1.plot(ma_iteration_space, self.env_eval_py.fidelity_list[:len(ma_fid)], color = '#FFCCCB')
        ax1.plot(ma_iteration_space, self.env_eval_py.energy_list[:len(ma_fid)], color = '#ECFFDC')
        ax1.plot(ma_iteration_space, ma_fid, label = "Fidelity", color = '#F70D1A')
        ax1.plot(ma_iteration_space, ma_energy, label = "Energy", color = '#7CFC00')
        ax1.set_ylim(0.0, 1.0)
        ax1.set_xlabel("Episode number")
        ax1.legend(loc = 'upper right')
        fig.suptitle(f"Fidelity, Energy, and Total Reward per Iteration QRLAgent \n $w_f = {self.env_eval_py.w_f}$, $w_e = {self.env_eval_py.w_e}$")
        fig.tight_layout()
        plt.show()

    def get_final_pulse(self):

        """
        Returns final action of agent
        """

        self.final_val = self.episode_list[-1]
        self.final_pulse = self.final_val.action.numpy()[0, 0, :]
        self.pulse_2d = np.reshape(self.final_pulse, (len(self.env_train_py.h_control), self.env_train_py.timesteps))

        return self.pulse_2d

    def get_highest_fidelity_pulse(self):

        """
        Returns the pulse with the highest fidelity
        """

        self.max_index = self.env_eval_py.fidelity_list.index(max(self.env_eval_py.fidelity_list))
        self.max_val = self.episode_list[self.max_index]
        self.max_pulse = self.max_val.action.numpy()[0, 0, :]
        self.max_pulse_2d = np.reshape(self.max_pulse, (len(self.env_train_py.h_control), self.env_train_py.timesteps))

        return self.max_pulse_2d

    def plot_final_pulse(self):

        """
        Plots the final action generated by the RL agent
        """

        self.timespace = np.linspace(0, self.env_eval_py.pulse_duration, self.env_eval_py.timesteps)

        colors = ['#03080c','#214868', '#5b97ca']

        self.final_val = self.episode_list[-1]
        self.final_pulse = self.final_val.action.numpy()[0, 0, :]

        self.pulse_2d = np.reshape(self.final_pulse, (len(self.env_train_py.h_control), self.env_train_py.timesteps))

        fig, ax = plt.subplots(len(self.env_train_py.h_control))

        if len(self.env_train_py.h_control) == 1:

            ax.axhline(y = 0, color = "grey", ls = "dashed")
            ax.step(self.timespace, self.pulse_2d[0], label = f"{self.env_train_py.labels[0]}", color = f"{colors[0]}")
            ax.set(xlabel = "Time (a.u.)", ylabel = f"{self.env_train_py.labels[0]}")
            ax.legend()

        else:

            for i in range(len(self.env_train_py.h_control)):
                ax[i].axhline(y = 0, color = "grey", ls = "dashed")
                ax[i].step(self.timespace, self.pulse_2d[i], label = f"{self.env_train_py.labels[i]}", color = f"{colors[i]}")
                ax[i].set(xlabel = "Time (a.u.)", ylabel = f"{self.env_train_py.labels[i]}")
                ax[i].legend()

        fig.suptitle("Final Pulse Generated by the QRLAgent")
        fig.tight_layout()

        plt.show()

    def show_summary(self):

        """
        Prints summary of the Actor Network (including number of parameters)
        """

        self.actor_net.summary()

    def save_weights(self, directory):

        my_weights = PolicySaver(self.collect_policy)
        my_weights.save(directory)

    def moving_average(self, a, n = 100):

        ret = np.cumsum(a)
        ret[n:] = ret[n:] - ret[:-n]
        return ret[n - 1:] / n
```

Appendix 4: EO-GRAPE Quantum Reinforcement Learning Class

```
class GRAPEQRLAgent:

    def __init__(self, TrainEnvironment, EvaluationEnvironment, num_iterations, num_cycles =
    1, fc_layer_params = (100, 100, 100), learning_rate = 1e-3,
    collect_episodes_per_iteration = 1, eval_interval = 1, replay_buffer_capacity = 100,
    policy = None):

        """
        GRAPEQRLAgent Class

        Create instance of reinforcement learning agent interacting with a GRAPEApproximation
     environment

        Parameters
        ----------

        TrainEnvironment : class
            GRAPEApproximation Instance

        EvaluationEnvironment : class
            GRAPApproximation Instance

        num_iterations : int
            Number of RL loop iterations
        """

        self.env_train_py = TrainEnvironment
        self.env_eval_py = EvaluationEnvironment
        self.num_cycles = num_cycles
        self.env_train_py.n_steps = self.num_cycles
        self.env_eval_py.n_steps = self.num_cycles
        self.num_iterations = num_iterations
        self.fc_layer_params = fc_layer_params
        self.learning_rate = learning_rate
        self.collect_episodes_per_iteration = collect_episodes_per_iteration
        self.eval_interval = eval_interval
        self.replay_buffer_capacity = replay_buffer_capacity
        self.policy = policy

        self.create_network_agent()

    def create_network_agent(self):

        """
        Create Neural Network and Agent Instance based on GRAPEApproximation class
        """

        self.train_env = tf_py_environment.TFPyEnvironment(self.env_train_py)
        self.eval_env = tf_py_environment.TFPyEnvironment(self.env_eval_py)

        self.actor_net = actor_distribution_network.ActorDistributionNetwork(
            self.train_env.observation_spec(),
            self.train_env.action_spec(),
            fc_layer_params = self.fc_layer_params
        )

        self.optimizer = keras.optimizers.Adam(learning_rate = self.learning_rate)

        self.train_step_counter = tf.compat.v2.Variable(0)

        self.tf_agent = reinforce_agent.ReinforceAgent(
            self.train_env.time_step_spec(),
            self.train_env.action_spec(),
            actor_network = self.actor_net,
            optimizer = self.optimizer,
            normalize_returns = True,
            train_step_counter = self.train_step_counter
        )

        self.tf_agent.initialize()

        if self.policy is None:

            self.eval_policy = self.tf_agent.policy
            self.collect_policy = self.tf_agent.collect_policy

        else:
            self.eval_policy = tf.compat.v2.saved_model.load(self.policy)
            self.collect_policy = self.tf_agent.collect_policy

        self.eval_policy = self.tf_agent.policy
        self.collect_policy = self.tf_agent.collect_policy

        self.replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
            data_spec = self.tf_agent.collect_data_spec,
            batch_size = self.train_env.batch_size,
            max_length = self.replay_buffer_capacity
        )

        self.eval_replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
            data_spec = self.tf_agent.collect_data_spec,
            batch_size = self.eval_env.batch_size,
            max_length = self.num_cycles + 1
        )

        self.avg_return = tf_metrics.AverageReturnMetric()

        self.eval_observers = [self.avg_return, self.eval_replay_buffer.add_batch]
        self.eval_driver = dynamic_episode_driver.DynamicEpisodeDriver(
            self.eval_env,
            self.eval_policy,
            self.eval_observers,
            num_episodes = 1
        )

        self.train_observers = [self.replay_buffer.add_batch]
        self.train_driver = dynamic_episode_driver.DynamicEpisodeDriver(
            self.train_env,
            self.collect_policy,
            self.train_observers,
            num_episodes = self.collect_episodes_per_iteration
        )

        self.tf_agent.train = common.function(self.tf_agent.train)

    def run_training(self, save_episodes = True, clear_buffer = False):

        """
        Starts training on Quantum RL Agent

        Parameters
        ----------

        save_episodes : bool : True
            Saves episodes if set to True

        clear_buffer : bool : False
            Clears buffer each episode if set to True
        """

        self.return_list = []
        self.episode_list = []
        self.iteration_list = []

        with trange(self.num_iterations, dynamic_ncols = False) as t:
            for i in t:

                t.set_description(f"Episode {i}")

                if clear_buffer:
                    self.replay_buffer.clear()

                final_time_step, policy_state = self.train_driver.run()
                experience = self.replay_buffer.gather_all()
                train_loss = self.tf_agent.train(experience)

                if i % self.eval_interval == 0 or i == self.num_iterations - 1:

                    self.avg_return.reset()
                    final_time_step, policy_state = self.eval_driver.run()

                    self.iteration_list.append(self.tf_agent.train_step_counter.numpy())
                    self.return_list.append(self.avg_return.result().numpy())

                    t.set_postfix({"Return" : self.return_list[-1]})

                    if save_episodes:
                        self.episode_list.append(self.eval_replay_buffer.gather_all())

    def plot_reward_per_iteration(self):

        """
        Plots reward per iteration
        """

        self.iteration_space = np.linspace(1, self.num_cycles * self.num_iterations, self.
num_cycles * self.num_iterations)
        fig, ax1 = plt.subplots()
        ax1.axhline(y = 0, color = "grey")
        ax1.plot(self.iteration_space, self.env_eval_py.reward_list, label = "Reward", marker
 = "d", color = '#5b97ca', markevery = 50)
        ax1.set_xlabel("Episode number")
        ax1.set_ylabel("Reward")
        ax1.legend(loc = (0.7, 0.45))
        fig.suptitle("Reward per Iteration GRAPEQRLAgent")
        fig.tight_layout()
        plt.show()

    def get_final_pulse(self):

        """
        Returns final action of agent
        """

        self.final_val = self.episode_list[-1]
        self.final_pulse = self.final_val.action.numpy()[0, 0, :]
        self.pulse_2d = np.reshape(self.final_pulse, (len(self.env_train_py.h_control), self.
env_train_py.timesteps))

        return self.pulse_2d

    def get_best_pulse(self):

        """
        Returns best action of agent
        """

        self.max_index = self.env_eval_py.reward_list.index(max(self.env_eval_py.reward_list)
)
        self.max_val = self.episode_list[self.max_index]
        self.max_pulse = self.max_val.action.numpy()[0, 0, :]
        self.max_pulse_2d = np.reshape(self.max_pulse, (len(self.env_train_py.h_control),
self.env_train_py.timesteps))

        return self.max_pulse_2d

    def plot_best_pulse(self):

        self.max_index = self.env_eval_py.reward_list.index(max(self.env_eval_py.reward_list)
    )
        self.max_val = self.episode_list[self.max_index]
        self.max_pulse = self.max_val.action.numpy()[0, 0, :]
        self.max_pulse_2d = np.reshape(self.max_pulse, (len(self.env_train_py.h_control),
    self.env_train_py.timesteps))

        fig, ax = plt.subplots(len(self.env_train_py.h_control))

        self.timespace = np.linspace(0, self.env_eval_py.pulse_duration, self.env_eval_py.
    timesteps)

        colors = ['#03080c','#214868', '#5b97ca']

        if len(self.env_train_py.h_control) == 1:

            ax.axhline(y = 0, color = "grey", ls = "dashed")
            ax.step(self.timespace, self.max_pulse_2d[0], label = f"{self.env_train_py.labels
    [0]}", color = f"{colors[0]}")
            ax.set(xlabel = "Time (a.u.)", ylabel = f"{self.env_train_py.labels[0]}")
            ax.legend()

        else:

            for i in range(len(self.env_train_py.h_control)):
                ax[i].axhline(y = 0, color = "grey", ls = "dashed")
                ax[i].step(self.timespace, self.max_pulse_2d[i], label = f"{self.env_train_py
    .labels[i]}", color = f"{colors[i]}")
                ax[i].set(xlabel = "Time (a.u.)", ylabel = f"{self.env_train_py.labels[i]}")
                ax[i].legend()

        fig.suptitle("Approximated GRAPE pulse generated by GRAPEQRLAgent")
        fig.tight_layout()

        plt.show()

    def plot_final_pulse(self):

        """
        Plots the final action generated by the RL agent
        """

        self.timespace = np.linspace(0, self.env_eval_py.pulse_duration, self.env_eval_py.
    timesteps)

        colors = ['#03080c','#214868', '#5b97ca']

        self.final_val = self.episode_list[-1]
        self.final_pulse = self.final_val.action.numpy()[0, 0, :]

        self.pulse_2d = np.reshape(self.final_pulse, (len(self.env_train_py.h_control), self.
    env_train_py.timesteps))

        fig, ax = plt.subplots(len(self.env_train_py.h_control))

        if len(self.env_train_py.h_control) == 1:

            ax.axhline(y = 0, color = "grey", ls = "dashed")
            ax.step(self.timespace, self.pulse_2d[0], label = f"{self.env_train_py.labels[0]}
    ", color = f"{colors[0]}")
            ax.set(xlabel = "Time (a.u.)", ylabel = f"{self.env_train_py.labels[0]}")
            ax.legend()

        else:

            for i in range(len(self.env_train_py.h_control)):
                ax[i].axhline(y = 0, color = "grey", ls = "dashed")
                ax[i].step(self.timespace, self.pulse_2d[i], label = f"{self.env_train_py.
    labels[i]}", color = f"{colors[i]}")
                ax[i].set(xlabel = "Time (a.u.)", ylabel = f"{self.env_train_py.labels[i]}")
                ax[i].legend()

        fig.suptitle("Approximated GRAPE pulse generated by GRAPEQRLAgent")
        fig.tight_layout()

        plt.show()

    def show_summary(self):

        """
        Prints summary of the Actor Network (including number of parameters)
        """

        self.actor_net.summary()

    def save_weights(self, directory):

        my_weights = PolicySaver(self.collect_policy)
        my_weights.save(directory)
```

Appendix 5: Miscellaneous Functions

```python
def tensor(a, b):
    """
    Returns tensor product between two matrices
    """

    return np.kron(a, b)

def identity(N):
    """
    Returns Identity Matrix with Dimension 'N'
    """

    return np.identity(N)

def sigmax():
    """
    Returns Pauli-x Matrix
    """

    return np.array([[0,1],
                     [1,0]])

def sigmay():
    """
    Returns Pauli-y Matrix
    """

    return np.array([[0, -1j],
                     [1j, 0]])

def sigmaz():
    """
    Returns Pauli-z Matrix
    """

    return np.array([[1, 0],
                     [0, -1]])

def cnot():
    """
    Returns CNOT Unitary Gate
    """

    return np.array([[1, 0, 0, 0],
                     [0, 1, 0, 0],
                     [0, 0, 0, 1],
                     [0, 0, 1, 0]])

def Generate_Rand_Unitary(N):
    """
    Returns N-Dimenstional Random Unitary
    """

    x = rand_unitary(N)
    y = x.full()
    nparray = np.array(y)

    return nparray

def hadamard():
    """
    Returns Hadamard Gate
    """

    return (1/np.sqrt(2)) * np.array([[1, 1],
                                      [1, -1]])

def t_gate():

    """
    Returns T−Gate
    """

    return np.array([[1, 0],
                     [0, np.exp(−1j * (np.pi/4))]])

def rx_gate(theta):

    """
    Returns X−Rotation gate
    """

    return np.array([[np.cos(theta/2), −1j * np.sin(theta/2)],
                     [−1j * np.sin(theta/2), np.cos(theta/2)]])

def rz_gate(theta):

    return np.array([[np.exp(−1j * theta/2), 0],
                     [0, np.exp(1j * theta/2)]])

def overlap(A, B):
    return np.trace(A.conj().T @ B) / A.shape[0]

def Calculate_Unitary_Scipy(H_Static, H_Control, Control_Pulses, Timesteps, Total_Time):

    """
    Calculates Unitary based on Static Hamiltonian, Control Hamiltonian, and control
    parameters

    Parameters
    −−−−−−−−−−

    H_Static : Static/Drift Hamiltonian Term

    H_Control : Control Hamiltonian containing operators that can be tuned in the Hamiltonian
     via the control fields

    Control_Pulses : The Control Parameters for each term in "H_Control"

    Timesteps : Number of timesteps 'N for time discretization

    Total_Time : Total Unitary Gate Time

    Returns
    −−−−−−−−−−

    Unitary_Total : Unitary Gate based on input parameters

    """

    time = np.linspace(0, Total_Time, Timesteps+1)

    H_Total = 0
    U_Total = []

    for i in range(Timesteps−1):
        dt = time[i+1] − time[i]
        H_Total = H_Static
        for j in range(len(H_Control)):
                H_Total += Control_Pulses[i*len(H_Control) + j] * H_Control[j] # (H_1(t = 0),
     H_2(t=0), H_1(t=1), ...)
        U = expm(−1j*H_Total*dt)
        U_Total.append(U)

    Unitary_Total = np.eye(4,4)
    for x in U_Total:
        Unitary_Total = x @ Unitary_Total

    return Unitary_Total
```

```
def Calculate_Fidelity(U_Target, U):

    """
    Calculate Fidelity Between Target Unitary and other Unitary U

    Parameters
    ----------

    U_Target : Target Unitary Gate

    U: Unitary Gate to Calculate Fidelity of

    Returns
    ----------

    F: Fidelity between U_Target and U

    """

    F = abs(np.trace(U_Target.conj().T @ U)/np.trace(U_Target.conj().T @ U_Target))**2

    return F

def CalculateEnergeticCost(Control_Pulses, H_Static, H_Control, Timesteps, Total_Time,
    Return_Normalized = False):

    """
    Calculate Energetic Cost of certain Unitary

    Parameters
    ----------

    Control_Pulses : The Control Parameters for each term in "H_Control"

    H_Static : Static/Drift Hamiltonian Term

    H_Control : Control Hamiltonoian containing operators that can be tuned in the
    Hamiltonian via the control fields

    Timesteps : Number of timesteps 'N' for time discretization

    Total_Time : Total time of unitary gate

    Returns
    ----------

    EC : Energetic Cost of the Control Pulses based on the static and drift Hamiltonian
    """

    H_T_Norm = []
    stepsize = Total_Time/Timesteps

    for i in range(Timesteps-1):
        H_T = 0

        for j in range(len(H_Control)):
            H_T += Control_Pulses[j, i] * H_Control[j]

        #H_T += H_Static  # Optionally include Static Hamiltonian

        H_T_Norm.append(np.linalg.norm(H_T))

    EC = np.sum(H_T_Norm) * stepsize

    EC_Normalized = EC / (Total_Time * np.linalg.norm(np.sum(H_Control)))

    if Return_Normalized == True:

        Value = EC_Normalized

    elif Return_Normalized == False:
        Value = EC

    return Value

def convert_qutip_to_numpy(operator):

    data = operator.full()
    array = np.array(data)

    return array

def convert_qutip_list_to_numpy(operator_list):

    new_list = []

    for operator in operator_list:
        new_list.append(convert_qutip_to_numpy(operator))

    return new_list
```

Appendix 6: Example Usage of Classes

```
# Initialize Environments
TrainingEnvironment = QuantumEnvironment(number_qubits, h_d, h_c, h_l, t1, t2,
    target_unitary_cnot, 0.2, 0.8, number_of_timesteps, gate_duration,
    number_of_grape_iterations, n_cycles)
EvaluationEnvironment = QuantumEnvironment(number_qubits, h_d, h_c, h_l, t1, t2,
    target_unitary_cnot, 0.2, 0.8, number_of_timesteps, gate_duration,
    number_of_grape_iterations, n_cycles)

TrainingEnvironmentGRAPE = GRAPEApproximation(number_qubits, h_d, h_c, h_l,
    target_unitary_cnot, w_f = 1.0, w_e = 0, timesteps = number_of_timesteps,
    grape_iterations = number_of_grape_iterations)
EvaluationEnvironmentGRAPE = GRAPEApproximation(number_qubits, h_d, h_c, h_l,
    target_unitary_cnot, w_f = 1.0, w_e = 0, timesteps = number_of_timesteps,
    grape_iterations = number_of_grape_iterations)

ApproximationAgent = GRAPEQRLAgent(TrainingEnvironmentGRAPE, EvaluationEnvironmentGRAPE,
    num_iterations_Approx, fc_layer_params = (100, 100, 100), replay_buffer_capacity = 100)

# Run GRAPE Approximation Training Phase and save policy
ApproximationAgent.run_training()
ApproximationAgent.save_weights('Test_Policy_Approx')

# Initialize RLAgent Environment including loaded policy
RLAgent = QuantumRLAgent(TrainingEnvironment, EvaluationEnvironment, num_iterations_RL, w_f =
     0.2, w_e = 0.8, fc_layer_params = (200, 100, 50, 30, 10), replay_buffer_capacity = 10,
    policy = None, rand_initial_state = False)

# Run Trainingq
RLAgent.run_training()
RLAgent.save_weights('Test_Policy_RL')

# Plot the Reward per iteration of the Approximation Agent
ApproximationAgent.plot_reward_per_iteration()

# Plot Best Pulse Generated by the Approximation agent
ApproximationAgent.plot_best_pulse()

# Plot the Pulse Generated by the QRLAgent
RLAgent.plot_final_pulse()

# PLot the Fidelity per iteration of the QRLAgent
RLAgent.plot_fidelity_energy_reward_per_iteration()
```

Appendix 7: Bloch Sphere Arc Length Experiment Example

```
# Define input parameters
drift_hamiltonian = h_d_1_qubit

n_q = number_qubits

control_hamiltonian = h_c_1_qubit

hamiltonian_label = h_l_1_qubit

u_target = fc.rx_gate(np.pi/2)

initial_state = basis(2,0)

gate_duration = 2 * np.pi

number_of_timesteps = 200

t1 = 1000 * gate_duration

t2 = 1000 * gate_duration

num_experiments = 10

weights = [1, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1]

# Simple Bloch Sphere Plot
def bloch_sphere_grape():

    environment = QuantumEnvironment(n_q, drift_hamiltonian, control_hamiltonian,
     hamiltonian_label, t1, t2, u_target, w_f = 1, w_e = 0, timesteps = number_of_timesteps,
     pulse_duration = gate_duration, grape_iterations = 200, n_steps = 1, sweep_noise = False)
    environment.initial_state = initial_state
    grape_pulses = environment.run_grape_optimization(w_f = 1, w_e = 0, eps_f = 1, eps_e =
     100)
    _, f_grape= environment.calculate_fidelity_reward(grape_pulses, plot_result = False)

    environment.plot_grape_pulses(grape_pulses)

    print("Fidelity is:", f_grape)

    environment.plot_bloch_sphere_trajectory()

    total_arc_length = environment.get_total_arc_length()

    print("Total Arc Length is:", total_arc_length)

def bloch_sphere_rl():

    TrainingEnvironment = QuantumEnvironment(n_q, drift_hamiltonian, control_hamiltonian,
     hamiltonian_label, t1, t2, u_target, w_f = 0.5, w_e = 0.5, timesteps =
     number_of_timesteps, pulse_duration = gate_duration, grape_iterations = 200, n_steps = 1,
      sweep_noise = False)
    TrainingEnvironment.initial_state = initial_state
    EvaluationEnvironment = QuantumEnvironment(n_q, drift_hamiltonian, control_hamiltonian,
     hamiltonian_label, t1, t2, u_target, w_f = 0.5, w_e = 0.5, timesteps =
     number_of_timesteps, pulse_duration = gate_duration, grape_iterations = 200, n_steps = 1,
      sweep_noise = False)
    EvaluationEnvironment.initial_state = initial_state
    RLAgent = QuantumRLAgent(TrainingEnvironment, EvaluationEnvironment, num_iterations_RL,
     w_f = 0.5, w_e = 0.5, fc_layer_params = (200, 100, 50, 30, 10), replay_buffer_capacity =
     10, policy = None, rand_initial_state = False)
    RLAgent.initial_state = initial_state
    RLAgent.run_training()

    BestPulse = RLAgent.get_highest_fidelity_pulse()

    _, f_rl = EvaluationEnvironment.calculate_fidelity_reward(BestPulse, plot_result = False)

    print("Fidelity is", f_rl)

    EvaluationEnvironment.plot_bloch_sphere_trajectory()

    arc_length = EvaluationEnvironment.get_total_arc_length()

    print(”Total Arc Length is:” , arc_length)

def correlation_experiment_grape():

    weights = [1, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1]
    arc_length_list = []
    energetic_cost_list = []

    for i in range(len(weights)):

        environment = QuantumEnvironment(n_q, drift_hamiltonian, control_hamiltonian, hamiltonian_label, t1, t2, u_target, w_f = weights[i], w_e = 1 − weights[i], timesteps = number_of_timesteps, pulse_duration = gate_duration, grape_iterations = 200, n_steps = 1, sweep_noise = False)
        environment.initial_state = initial_state
        grape_pulses = environment.run_grape_optimization(w_f = weights[i], w_e = 1 − weights[i], eps_f = 1, eps_e = 100)
        _, f_grape= environment.calculate_fidelity_reward(grape_pulses, plot_result = False)
        total_arc_length = environment.get_total_arc_length()
        energetic_cost = environment.calculate_energetic_cost(grape_pulses, return_normalized = False)
        arc_length_list.append(total_arc_length)
        energetic_cost_list.append(energetic_cost)

    plt.plot(energetic_cost_list, arc_length_list, marker = ’d’, color = ’#214868’)
    plt.xlabel(”Energetic Cost (a.u.)”)
    plt.ylabel(”Bloch Sphere Arc Length (a.u.)”)
    plt.grid()
    plt.tight_layout()
    plt.show()

def correlation_experiment_rl():

    weights = [1, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1]
    arc_length_list = np.zeros((num_experiments, len(weights)))
    energetic_cost_list = np.zeros((num_experiments, len(weights)))
    fidelity_list = np.zeros((num_experiments, len(weights)))

    for i in range(num_experiments):

        for j in range(len(weights)):

            TrainingEnvironment = QuantumEnvironment(n_q, drift_hamiltonian, control_hamiltonian, hamiltonian_label, t1, t2, u_target, w_f = weights[j], w_e = 1 − weights[j], timesteps = number_of_timesteps, pulse_duration = gate_duration, grape_iterations = 200, n_steps = 1, sweep_noise = False)
            TrainingEnvironment.initial_state = initial_state
            EvaluationEnvironment = QuantumEnvironment(n_q, drift_hamiltonian, control_hamiltonian, hamiltonian_label, t1, t2, u_target, w_f = weights[j], w_e = 1−weights[j], timesteps = number_of_timesteps, pulse_duration = gate_duration, grape_iterations = 200, n_steps = 1, sweep_noise = False)
            EvaluationEnvironment.initial_state = initial_state
            RLAgent = QuantumRLAgent(TrainingEnvironment, EvaluationEnvironment, num_iterations_RL, w_f = weights[j], w_e = 1 − weights[j], fc_layer_params = (200, 100, 50, 30, 10), replay_buffer_capacity = 10, policy = None, rand_initial_state = False)
            RLAgent.initial_state = initial_state
            RLAgent.run_training()

            BestPulse = RLAgent.get_final_pulse()

            _, f_rl = EvaluationEnvironment.calculate_fidelity_reward(BestPulse, plot_result = False)
            total_arc_length = EvaluationEnvironment.get_total_arc_length()
            energetic_cost = EvaluationEnvironment.calculate_energetic_cost(BestPulse, return_normalized = False)
            arc_length_list[i, j] = total_arc_length
            energetic_cost_list[i, j] = energetic_cost
            fidelity_list[i, j] = f_rl

    np.save("Arc_Length_Multiple_Experiments.npy", arc_length_list)
    np.save("EC_RL_Multiple_Experiments.npy", energetic_cost_list)
    np.save("F_RL_Multiple_Experiments.npy", fidelity_list)

correlation_experiment_rl()

arc_length_array = np.load("Arc_Length_Multiple_Experiments.npy")
ec_array = np.load("EC_RL_Multiple_Experiments.npy")
f_array = np.load("F_RL_Multiple_Experiments.npy")
mean_arc_length = []
st_dev_arc_length = []
mean_ec = []
st_dev_ec = []
mean_f = []

for i in range(len(weights)):

    m_arc_length = np.mean(arc_length_array[:, i])
    mean_arc_length.append(m_arc_length)
    std_arc_length = np.std(arc_length_array[:, i])
    st_dev_arc_length.append(std_arc_length)
    m_ec = np.mean(ec_array[:, i])
    mean_ec.append(m_ec)
    std_ec = np.std(ec_array[:, i])
    st_dev_ec.append(std_ec)
    m_f = np.mean(f_array[:, i])
    mean_f.append(m_f)

fig, ax = plt.subplots()
scatter = ax.scatter(mean_ec, mean_arc_length, marker = 'd', c = mean_f)
clb = plt.colorbar(scatter)
norm = colors.Normalize(vmin = min(mean_f), vmax = max(mean_f))
mapper = cm.ScalarMappable(norm = norm, cmap = "viridis")
color = np.array([(mapper.to_rgba(v)) for v in mean_f])

for x, y, e_x, e_y, c in zip(mean_ec, mean_arc_length, st_dev_ec, st_dev_arc_length, color):
    plt.scatter(x, y, marker = 'd', color = c)
    plt.errorbar(x, y, xerr = e_x, yerr = e_y, fmt = "d", color = c)

plt.xlabel("Energetic Cost (a.u.)")
plt.ylabel("Bloch Sphere Arc Length (a.u.)")
plt.grid()
plt.tight_layout()
plt.show()
```

Bibliography

[1] C. P. Koch, U. Boscain, T. Calarco, G. Dirr, S. Filipp, S. J. Glaser, R. Kosloff, S. Montangero, T. Schulte-Herbrüggen, D. Sugny, *et al.*, *Quantum optimal control in quantum technologies. strategic report on current status, visions and goals for research in europe,* EPJ Quantum Technology **9**, 19 (2022).

[2] N. Khaneja, T. O. Reiss, C. Kehlet, T. Schulte-Herbrüggen, and S. J. Glaser, *Optimal control of coupled spin dynamics: design of nmr pulse sequences by gradient ascent algorithms.* Journal of magnetic resonance **172 2**, 296 (2005).

[3] M. Y. Niu, S. Boixo, V. Smelyanskiy, and H. Neven, *Universal quantum control through deep reinforcement learning,* (2018), arXiv:1803.01857 [quant-ph] .

[4] S. Deffner, *Energetic cost of hamiltonian quantum gates,* Europhysics Letters **134**, 40002 (2021).

[5] R. P. Feynman, *Simulating physics with computers,* International journal of theoretical physics **21**, 467 (1982).

[6] P. W. Shor, *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer,* SIAM Journal on Computing **26**, 1484–1509 (1997).

[7] C.-F. Chen, H.-Y. Huang, J. Preskill, and L. Zhou, *Local minima in quantum systems,* (2023), arXiv:2309.16596 [quant-ph] .

[8] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition* (Cambridge University Press, 2011).

[9] F. Bellaiche, *An update on quantum computing and complexity classes,* Https://www.quantum-bits.org/?p=2309.

[10] S. Jordan, *Quantum algorithm zoo,* Https://quantumalgorithmzoo.org/.

[11] R. Sutor and H. Higgins, *Taking the quantum leap,* Https://www.ibm.com/thought-leadership/institute-business-value/en-us/report/quantumleap.

[12] B. Terhal, *Notes on the rotating frame,* Quantum Hardware , AP 3292, Winter 2016.

[13] D. Egger and N. Kanazawa, *The benefits of high-resolution pulses for quantum computers,* May 14, 2022.

[14] K. Kubo and H. Goto, *Fast parametric two-qubit gate for highly detuned fixed-frequency superconducting qubits using a double-transmon coupler,* Applied Physics Letters **122** (2023), 10.1063/5.0138699.

[15] P. Krantz, M. Kjaergaard, F. Yan, T. P. Orlando, S. Gustavsson, and W. D. Oliver, *A quantum engineer's guide to superconducting qubits,* Applied Physics Reviews **6** (2019), 10.1063/1.5089550.

[16] C. Coleman, *A guide to quantum computing benchmarking and certification,* August 12, 2022.

[17] R. Acharya, I. Aleiner, R. Allen, T. I. Andersen, M. Ansmann, F. Arute, K. Arya, A. Asfaw, J. Atalaya, R. Babbush, D. Bacon, J. C. Bardin, J. Basso, A. Bengtsson, and S. Boixo, *Suppressing quantum errors by scaling a surface code logical qubit,* (2022), arXiv:2207.06431 [quant-ph] .

[18] J. Preskill, *Quantum computing in the nisq era and beyond,* Quantum **2**, 79 (2018).

[19] J. Kelly, *A preview of bristlecone, google's new quantum processor,* (2018).

[20] C. E. Murray, *Material matters in superconducting qubits,* Materials Science and Engineering: R: Reports **146**, 100646 (2021).

[21] S. Joshi and S. Moazeni, *Scaling up superconducting quantum computers with cryogenic rf-photonics,* (2022), arXiv:2210.15756 [quant-ph] .

[22] QEI, *The quantum energy initiative manifesto,* QEI (2023).

[23] M. Aifer and S. Deffner, *From quantum speed limits to energy-efficient quantum gates,* New Journal of Physics **24**, 055002 (2022).

[24] C. G. Almudever, L. Lao, X. Fu, N. Khammassi, I. Ashraf, D. Iorga, S. Varsamopoulos, C. Eichler, A. Wallraff, L. Geck, A. Kruth, J. Knoch, H. Bluhm, and K. Bertels, *The engineering challenges in quantum computing,* in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017* (2017) pp. 836–845.

[25] D. Shaw, *Quantum software outlook 2022,* Https://www.factbasedinsight.com/quantum-software-outlook-2022/.

[26] T. S. Mahesh, P. Batra, and M. H. Ram, *Quantum optimal control: Practical aspects and diverse methods,* (2022), arXiv:2205.15574 [quant-ph] .

[27] U. Boscain, M. Sigalotti, and D. Sugny, *Introduction to the pontryagin maximum principle for quantum optimal control,* PRX Quantum **2** (2021), 10.1103/prxquantum.2.030203.

[28] A. Ma, A. B. Magann, T.-S. Ho, and H. Rabitz, *Optimal control of coupled quantum systems based on the first-order magnus expansion: Application to multiple dipole-dipole-coupled molecular rotors,* Phys. Rev. A **102**, 013115 (2020).

[29] A. Ibort and J. M. Pérez-Pardo, *Quantum control and representation theory,* Journal of Physics A: Mathematical and Theoretical **42**, 205301 (2009).

[30] S. Bhatt, *Reinforcement learning 101,* Https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292.

[31] D. Dong, J. Tang, Q. Yu, P. Girdhar, and O. Shindi, *Optimal and robust control of quantum systems using reinforcement learning approaches,* UNSW Library (2023).

[32] D. Dong, M. A. Mabrok, I. R. Petersen, B. Qi, C. Chen, and H. Rabitz, *Sampling-based learning control for quantum systems with uncertainties,* IEEE Transactions on Control Systems Technology **23**, 2155–2166 (2015).

[33] M. E. Fonseca, F. F. Fanchini, E. F. de Lima, and L. K. Castelano, *Effectiveness of the krotov method in controlling open quantum systems,* (2023), arXiv:2208.03114 [quant-ph] .

[34] S. Machnes, E. Assémat, D. Tannor, and F. K. Wilhelm, *Tunable, flexible, and efficient optimization of control pulses for practical qubits,* Phys. Rev. Lett. **120**, 150401 (2018).

[35] T. Caneva, T. Calarco, and S. Montangero, *Chopped random-basis quantum optimization,* Physical Review A **84** (2011), 10.1103/physreva.84.022326.

[36] J. Brown, M. Paternostro, and A. Ferraro, *Optimal quantum control via genetic algorithms for quantum state engineering in driven-resonator mediated networks,* Quantum Science and Technology **8**, 025004 (2023).

[37] S. Hu, H. Ma, D. Dong, and C. Chen, *Two-step robust control design of quantum gates via differential evolution,* J. Franklin Inst. **360**, 13972 (2023).

[38] E. S. Matekole, Y.-L. L. Fang, and M. Lin, *Methods and results for quantum optimal pulse control on superconducting qubit systems,* in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (IEEE, 2022).

[39] P. Rembold, N. Oshnik, M. M. Müller, S. Montangero, T. Calarco, and E. Neu, *Introduction to quantum optimal control for quantum sensing with nitrogen-vacancy centers in diamond,* AVS Quantum Science **2** (2020), 10.1116/5.0006785.

[40] V. Nebendahl, H. Häffner, and C. F. Roos, *Optimal control of entangling operations for trapped-ion quantum computing,* Phys. Rev. A **79**, 012312 (2009).

[41] A. Castro, A. García Carrizo, S. Roca, D. Zueco, and F. Luis, *Optimal control of molecular spin qudits,* Phys. Rev. Appl. **17**, 064028 (2022).

[42] Z. Wang, S. Hadfield, Z. Jiang, and E. G. Rieffel, *Quantum approximate optimization algorithm for maxcut: A fermionic view,* Physical Review A **97** (2018), 10.1103/physreva.97.022304.

[43] C. Lin, Y. Wang, G. Kolesov, and U. Kalabić, *Application of pontryagin's minimum principle to grover's quantum search problem,* Physical Review A **100** (2019), 10.1103/physreva.100.022327.

[44] M. Larocca, P. Czarnik, K. Sharma, G. Muraleedharan, P. J. Coles, and M. Cerezo, *Diagnosing barren plateaus with tools from quantum optimal control,* Quantum **6**, 824 (2022).

[45] L. T. Brady, C. L. Baldwin, A. Bapat, Y. Kharkov, and A. V. Gorshkov, *Optimal protocols in quantum annealing and quantum approximate optimization algorithm problems,* Physical Review Letters **126** (2021), 10.1103/physrevlett.126.070505.

[46] T. Caneva, M. Murphy, T. Calarco, R. Fazio, S. Montangero, V. Giovannetti, and G. E. Santoro, *Optimal control at the quantum speed limit,* Phys. Rev. Lett. **103**, 240501 (2009).

[47] Z. Yin, C. Li, J. Allcock, Y. Zheng, X. Gu, M. Dai, S. Zhang, and S. An, *Shortcuts to adiabaticity for open systems in circuit quantum electrodynamics,* Nature Communications **13** (2022), 10.1038/s41467-021-27900-6.

[48] I. Khait, J. Carrasquilla, and D. Segal, *Optimal control of quantum thermal machines using machine learning,* Phys. Rev. Res. **4**, L012029 (2022).

[49] G. Song and A. Klappenecker, *Optimal realizations of controlled unitary gates,* (2002), arXiv:quant-ph/0207157 [quant-ph] .

[50] S. Gherardini, M. M. Müller, S. Montangero, T. Calarco, and F. Caruso, *Information flow and error scaling for fully quantum control,* Phys. Rev. Res. **4**, 023027 (2022).

[51] Y. Lin, J. Gaebler, F. Reiter, T. Tan, R. Bowler, A. Sørensen, D. Leibfried, and D. Wineland, *Dissipative production of a maximally entangled steady state of two quantum bits,* Nature **504** (2013), 10.1038/nature12801.

[52] D. Basilewitsch, C. P. Koch, and D. M. Reich, *Quantum optimal control for mixed state squeezing in cavity optomechanics,* Advanced Quantum Technologies **2**, 1800110 (2019).

[53] D. Basilewitsch, F. Cosco, N. Lo Gullo, M. Möttönen, T. Ala-Nissilä, C. P. Koch, and S. Maniscalco, *Reservoir engineering using quantum optimal control for qubit reset,* New Journal of Physics **21**, 093054 (2019).

[54] A. Litteken, L. M. Seifert, J. D. Chadwick, N. Nottingham, T. Roy, Z. Li, D. Schuster, F. T. Chong, and J. M. Baker, *Dancing the quantum waltz: Compiling three-qubit gates on four level architectures,* in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23 (ACM, 2023).

[55] V. B. et al., *Pennylane: Automatic differentiation of hybrid quantum-classical computations,* (2022), arXiv:1811.04968 [quant-ph] .

[56] Q-CTRL, *Boulder Opal documentation: Designing robust, configurable, parallel gates for large trapped-ion arrays,* /boulder-opal/application-notes/designing-robust-configurable-parallel-gates-for-large-trapped-ion-arrays (2022), [Online; accessed 12-August-2022].

[57] M. H. Goerz, D. Basilewitsch, F. Gago-Encinas, M. G. Krauss, K. P. Horn, D. M. Reich, and C. P. Koch, *Krotov: A Python implementation of Krotov's method for quantum optimal control,* SciPost Phys. **7**, 80 (2019).

[58] M. Rossignolo, T. Reisser, A. Marshall, P. Rembold, A. Pagano, P. J. Vetter, R. S. Said, M. M. Müller, F. Motzoi, T. Calarco, F. Jelezko, and S. Montangero, *QuOCS: The Quantum Optimal Control Suite,* Computer Physics Communications , 108782 (2023).

[59] Qiskit Community, *Qiskit: An open-source framework for quantum computing,* (2017).

[60] J. Johansson, P. Nation, and F. Nori, *Qutip: An open-source python framework for the dynamics of open quantum systems,* Computer Physics Communications **183**, 1760–1772 (2012).

[61] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, *Array programming with NumPy,* Nature **585**, 357 (2020).

[62] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, *SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python,* Nature Methods **17**, 261 (2020).

[63] M. Abadi, *TensorFlow: Large-scale machine learning on heterogeneous systems,* (2015), software available from tensorflow.org.

[64] D. Jayakody, *Reinforce - a quick introduction (with code),* (2023).

[65] T. Van Vu and K. Saito, *Finite-time quantum landauer principle and quantum coherence,* Physical Review Letters **128** (2022), 10.1103/physrevlett.128.010602.

[66] S. Haseli, H. Dolatkhah, S. Salimi, and A. S. Khorashad, *Controlling the entropic uncertainty lower bound in two-qubit systems under decoherence,* Laser Physics Letters **16**, 045207 (2019).

[67] A. Ibort and J. M. Pérez-Pardo, *Quantum control and representation theory,* Journal of Physics A: Mathematical and Theoretical **42**, 205301 (2009).

[68] D. Dong, C.-C. Shu, J. Chen, X. Xing, H. Ma, Y. Guo, and H. Rabitz, *Learning control of quantum systems using frequency-domain optimization algorithms,* IEEE Transactions on Control Systems Technology **29**, 1791–1798 (2021).

[69] T. Eimer, M. Lindauer, and R. Raileanu, *Hyperparameters in reinforcement learning and how to tune them,* (2023), arXiv:2306.01324 [cs.LG] .

[70] M. Abdelhafez, B. Baker, A. Gyenis, P. Mundada, A. A. Houck, D. Schuster, and J. Koch, *Universal gates for protected superconducting qubits using optimal control,* Phys. Rev. A **101**, 022321 (2020).

[71] K. Takase, A. Kawasaki, B. K. Jeong, T. Kashiwazaki, T. Kazama, K. Enbutsu, K. Watanabe, T. Umeki, S. Miki, H. Terai, M. Yabuno, F. China, W. Asavanant, M. Endo, J.-i. Yoshikawa, and A. Furusawa, *Quantum arbitrary waveform generator,* Science Advances **8** (2022), 10.1126/sciadv.add4019.