Doctoral Dissertation Academic Year 2026

Data-Efficient Characterization & Calibration Framework for Quantum Processors

Graduate School of Media and Governance
Keio University

Poramet Pathumsoot

© 2026

Poramet Pathumsoot

ORCID: 0000-0003-2996-480X

All rights reserved.

Abstract

Quantum processors are expected to speed up various applications, including, but not limited to, chemistry, finance, and secure communication, on the premise that the device's error rate is well below the quantum error correction threshold. Thus, the infrastructure provider must perform routine characterization and calibration (C&C) to maintain the device's quality. The user cannot access the device during the C&C routine, which is not ideal but necessary for both the user and the provider. To maximize device availability, the provider must minimize the time required for the C&C routine. This thesis introduces a modular C&C framework designed to reduce system downtime, implemented as an open-source Python library named Inspeqtor. To enable broad data efficiency, the framework establishes compatible abstractions for Bayesian Optimal Experimental Design (BOED) tailored for general-purpose models. Alongside this capability, Inspeqtor provides first-class support for a Graybox characterization method adaptable to multiple qubit implementations. Through statistical analysis of this Graybox method, we derive a performance bound showing that, while the expected minimum of the empirical error depends on the number of measurement shots, the predictive performance bound on average gate fidelity remains independent of the shot count due to the cancellation of data variance. This reveals that once the model's test error approaches this shot-dependent variance, it can accurately predict average gate fidelity, suggesting that strategic balancing of shot count and sample size can be exploited to train highly accurate predictive models. Furthermore, we develop a Probabilistic Graybox Model (PGM) to enable the integration of Graybox characterization with BOED. We validate this approach via a numerical simulation, demonstrating that the PGM maintains high accuracy even under stochastic noise and with limited training sample sizes. By building each module directly around the system's mathematical representation, Inspeqtor bridges the gap between software implementation and mathematical abstraction. This functional, composable approach narrows the domain-knowledge gap for infrastructure engineers and quantum researchers, widening the path to highly available quantum computing platforms.

Keywords: Characterization, Calibration, Optimal Control, Quantum Device.

Acknowledgement

A decade has passed since I decided to walk down this road, and it would not have been possible without my academic advisors. First, I would like to express my gratitude to Rodney Van Meter for these opportunities to fulfill one of my dreams. Rod has completely changed how I shape the idea into reality. He taught me what it takes to develop a creation that is not just work, but reliable and substantial. I want to thank my undergraduate and master's advisor, Sujin Suwanna, for preparing me and setting the stage for me to step up. I would also like to thank Areeya Chantasri, Michal Hajdušek, Shota Nagayama and Kusumoto Hiroyuki for their help and encouragement throughout the research that enabled me to reach this milestone. I would like to thank Andrew Todd for teaching me how to properly develop and maintain software.

I would like to thank my thesis committee, Rodney Van Meter, Michal Hajdušek, Naoki Yamamoto, Jin Mitsugi, and Tatsuro Shimizu, for their valuable advice and comments in shaping this dissertation.

I am deeply thankful to the Yoshida Scholarship Foundation and Keio University, along with their dedicated staff, for granting me the YKK Leader 21 scholarship. Their generosity allowed me to devote myself fully to my research, free from the burden of financial worries.

To Naphan Benchasattabuse, Satoh Ryosuke, Kento Samuel Soon, Monet Tokuyama, Marii Koyama, Natchapol Patamawisut, Sara Metwalli, Sittha Jeamburaset, and everyone in AQUA, I am very grateful. You made my life in Japan brighter, lighter, and filled with laughter. To my friends in Thailand—Sgt. Oxford, Prin, Nicha, Supawit, Pachraporn, Arin, Ratnamon, Nutsajee, Somjutar, Sudarat, Chinnakit, Krit, Kitthapak, Kriengkrai, Chanisara, Poramat, Tarudee, Thanaphorn and so many more—thank you for sharing your time, your joy, and most importantly, your listening ears with me.

None of this would have been possible without my mother, Pranee Pathumsoot. The unwavering support of my family gives me the courage to keep moving forward with warmth in my heart.

I am thankful for everyone I encountered along the way. Some passed by to help me temper my dao heart, while some stayed to offer their heartfelt support. I have been moved by the kindness shown through your actions, and I am endlessly grateful to those who stood by me throughout this journey. Thank you all for your patience with me. I know I can be a handful 😘

Published and Preprint Contents and Contributions

Portions of this dissertation are reproduced from the following publications and preprints.

Pathumsoot, P., Matsuo, T., Satoh, T., Hajdušek, M., Suwanna, S., & Van Meter, R. (2020). Modeling of measurement-based quantum network coding on a superconducting quantum processor. Physical Review A, 101(5), 052301. https://doi.org/10.1103/PhysRevA.101.052301

Pathumsoot, P., Chantasri, A., Hajdušek, M., & Van Meter, R. (2026). Graybox characterization and calibration with finite-shot estimation on superconducting-qubit experiments. New Journal of Physics. https://doi.org/10.1088/1367-2630/ae6df2

Pathumsoot, P., Hajdušek, M., & Van Meter, R. (2025). Probabilistic Graybox Characterization of Quantum Devices with Bayesian Neural Networks (arXiv:2509.24232). arXiv. https://doi.org/10.48550/arXiv.2509.24232

N.B. Currently under consideration at Transaction on Quantum Engineering.

Contents

Abstract III
Acknowledgement IV
Published and Preprint Contents and Contributions V
1. Introduction 2
    1.1. The promises of quantum devices 2
    1.2. The Challenges 2
    1.3. Related works 5
    1.4. How to read this dissertation 7
2. Background: Before the Journey Begins 9
    2.1. Quantum Information 9
        2.1.1. Quantum State 9
        2.1.2. Quantum Operation 11
        2.1.3. From Hamiltonian to Final Quantum State 13
        2.1.4. Noise in the quantum system 15
        2.1.5. Fidelity 17
    2.2. Characterization & Calibration (C&C) 18
        2.2.1. Calibration 18
            2.2.1.1. Closed-loop Calibration 19
            2.2.1.2. Open-loop Calibration 21
            2.2.1.3. Hybrid Approach Calibration 21
        2.2.2. Characterization 22
    2.3. C&C Framework 23
    2.4. The notion of “Data Efficiency” 24
3. Theoretical methodologies 27
    3.1. C&C: Statistical Machine Learning 28
        3.1.1. Characterization using Statistical Model 30
        3.1.2. Graybox Characterization Method 33
        3.1.3. What we can learn from SML approach 36
            3.1.3.1. Decomposition of MSE 37
            3.1.3.2. Performance bound 40
        3.1.4. Numerical and Experimental Characterization using Statistical Graybox Model 45
            3.1.4.1. Experimental procedure 45
            3.1.4.2. Numerical Analysis 48
            3.1.4.3. Characterization and Calibration for Superconducting Qubit .. 52
    3.2. C&C: Probabilistic Machine Learning 57

3.2.1. Predictive model, but probabilistic 58
3.2.2. Characterization using a Probabilistic Model 59
3.2.3. Probabilistic Graybox model with Bayesian Neural Network 61
3.3. Data-Efficient approaches 67
3.3.1. Data-variance 67
3.3.2. One-step Optimal BOED 69
3.3.3. Sequential-BOED 69
4. Beyond a Single Processor 71
4.1. Entanglement Swapping Based Approach 72
4.2. Measurement-based Quantum Network Coding 74
4.3. Simulating Resource Distribution on a Superconducting Qubit Device 75
4.4. C&C Ways to Improve the Experiment 79
5. Design and Implementation of Inspeqtor 83
5.1. Structure of `inspeqtor` 87
5.2. Why JAX 88
5.3. Control 91
5.4. Data 96
5.4.1. Experiment Config 98
5.4.2. Experiment Data 99
5.4.3. Operator, State, and Expectation Value 101
5.4.4. Data flow 102
5.5. Physics 104
5.6. Optimize 109
5.6.1. Bayesian Optimization. 112
5.6.2. Control calibration process 113
5.7. Models 114
5.7.1. Predictive model interface 115
5.7.2. Loss function interface 116
5.7.3. Train model 116
5.7.4. Serialization and Deserialization 118
5.7.5. Characterization process 118
5.8. Probabilistic 119
5.8.1. Predictive model 120
5.8.2. Model Inferenece 122
5.8.3. Characterization process 123
5.8.4. Distance measures of probability distribution 124
5.9. Bayesian Optimal Experimental Design 125
5.10. Utils 127

6. Numerical Experiments with Inspeqtor 129
6.1. Noisy Quantum Device 129
6.2. Statistical Approach 133
6.3. Probabilistic Approach 139
6.4. Statistical vs. Probabilistic 147
6.5. BOED approach 151
7. Next Steps Forward 163
7.1. Captain's Log: The Journey So Far 163
7.2. To boldly go where no one has gone before 165
Bibliography 168
A Detailed Derivation 185
A.1 Data variance: An alternative derivation 185
A.2 The performance upper bound in terms of the data variance 187
A.3 Rotating Frame 189
B List of Papers and Presentations 191
B.1 Peer-Reviewed Journals 191
B.2 Conference Papers (Peer-Reviewed) 191
B.3 Manuscripts under Review and In Preparation 192
Glossary 193

Chapter 1. Introduction

1.1. The promises of quantum devices

Quantum devices with all-time peak performance are extremely important for achieving computational power beyond what classical devices can provide [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]. Under the hood, a quantum computer harnesses properties of objects that obey the laws of quantum mechanics, such as superposition, interference, and entanglement, as its processing units, opening up multiple possibilities. One of the possiblilities of using quantum computer is to break cryptosystems, including, RSA protocol [1], [2] drive the needs of secure quantum communication protocols [13], [14], [15], [16] that do not rely on the mathematical assumption that certain problem is hard to solve within a reasoable time. However, by using the same underlying physics that governs nature, a quantum computer has a natural application.

"*Nature isn't classical, dammit, and if you want to make a simulation of nature, you'd better make it quantum mechanical, and by golly it's a wonderful problem, because it doesn't look so easy.* [17]"

Richard Feynman

One of the most anticipated applications is the use of a quantum computer for Hamiltonian simulation, which will aid researchers in understanding the chemical mechanism and potentially enable the development of even more efficient synthesis methodologies. For instance, understanding the mechanism of nitrogen fixation at FeMoCo (Iron-Molybdenum Cofactor) may lead to cheaper fertilizer production [11], [18]. Nonetheless, the road ahead is still full of challenges.

1.2. The Challenges

To gain this power, there are two main problems that we have to solve first. The first is to achieve a peak performance quantum device. The second is to maintain the peak performance. The first problem may be solved by using quantum optimal control. Through quantum optimal control, we aim to determine the control of a quantum system so that it maximizes performance metrics, such as minimizing the distance between a target state and the constructed state as quickly as possible under given constraints. However, due to temporal drifting of the noise [19], [20], [21], [22], [23], [24], we have to perform a routine calibration, which solves the second problem. Yet, this protocol may induce another problem. Since calibration requires downtime, the device's availability for actual computation is consequently reduced. Thus, it is also extremely important to reduce the downtime for device calibration as much as possible.

What if we use error correction? You might ask... Similar to the well-established computer system, a quantum computer will eventually use fault-tolerant quantum error correction (FTQEC) protocol [25], [26], [27], [28], [29] for its logical operations instead of the raw physical operations. Nevertheless, FTQEC relies on an assumption that the underlying physical operations operate below a noise threshold [29]. Thus, calibration of physical-level operations is a requirement even in the era of FTQEC. Moreover, since the physical operation quality may degrade over time due to the temporal noise, routine calibration is unavoidable. The need for routine calibration of physical-level operations is not limited to devices such as quantum computers but also applies to small quantum devices like quantum repeaters [30], [31], [32], [33], which are essential for long-range communication [34], [35], [36] using quantum protocols [13], [14], [15], [16], [34] in the early generations [37].

Reducing maintenance downtime improves device availability. However, it is also important to minimize the amount of quantum data required for calibration, since high operational costs can make quantum device calibration financially devastating. Therefore, efficient calibration of operations at the physical level addresses the problem at its source.

The next question is how to calibrate and characterize a quantum device efficiently?... In fact, achieving this leads to several follow-up questions and challenges to be addressed.

Challenge number one: There are multiple possible physical realizations (qubit modalities/platforms) of the quantum device. As of the time this dissertation is written, the superconducting platform [29], [38], [39] is prominent due to our ability to engineer the behavior of these artificial qubits to some extent. Other choices include a trapped-ion platform [40], [41], [42], [43], [44], [45], which offers a long lifetime, and a photonic platform [46], [47], [48], [49], which is suitable for long-distance communication. Each platform has its own physical model and requires different infrastructure to operate and, consequently, maintain. Thus, calibration of each platform leads to specialized software and hardware implementation [50], [51]. Fortunately, it is possible to apply the theory of quantum optimal control to each platform, since they are quantum systems nonetheless. Regardless of the platform, we can use an appropriate abstraction level to calibrate for optimal control.

Challenge number two: The abstraction requires a mathematical model of the device. Consequently, we have to estimate accurate model parameters from the system. The process of constructing a model from a device is called "characterization". Device characterization typically involves experiments on the actual hardware, from which we can extract model parameters. Once the model is constructed, we can perform the optimal control using the model in parallel since it is a local replica (offline) of the quantum

device. We refer the calibration approach using model of the system as the model-based calibration. Nonetheless, depending on the model of choice, the characterization process may also turn out to be another costly operation on the quantum device. Alternatively, we can directly calibrate a quantum device using a model-free approach, thereby skipping characterization and allowing the optimizer to interact with the device rather than with the replica model. Consequently, we must sequentially calibrate the device for each quantum operation required for the universal gateset. Moreover, if the blackbox optimization algorithm is not carefully designed, it could lead to a costly closed-loop (online) calibration process.

As a challenger, I aim to attack these challenges by proposing a framework for quantum device characterization and calibration that focuses on utilizing experimental data efficiently. In this dissertation, I present Inspeqtor, a framework and a Python library that implements it. The library implements reusable building components that enable the composition of a complete routine maintenance system tailored to the quantum device. Consider a flow chart in Figure 1.1. The maintenance system may consist of characterization and calibration, in which the calibration interacts with a local replica of the subsystem (model-based calibration), or may rely solely on closed-loop calibration, in which the optimizer interacts directly with the quantum system (model-free calibration). The library offers components that implement flexible characterization techniques, such as the Graybox characterization method [52], [53], which allows users to integrate their prior knowledge of the device, or adopt fully Blackbox or physics-based methods. To reduce interaction with the device, the library offers an estimator of expected information gain given the experiment's design through Bayesian Optimal Experimental Design (BOED) [54], [55], allowing users to devise a data-collection strategy based on the metric. For calibration, the library implements Bayesian optimization, a powerful method where evaluating a cost function is expensive. On the other hand, I use JAX as the numerical backend, allowing users to use Optax, which offers a range of gradient-based optimizers. We prioritize compatibility, readability, and testability across the component designs.

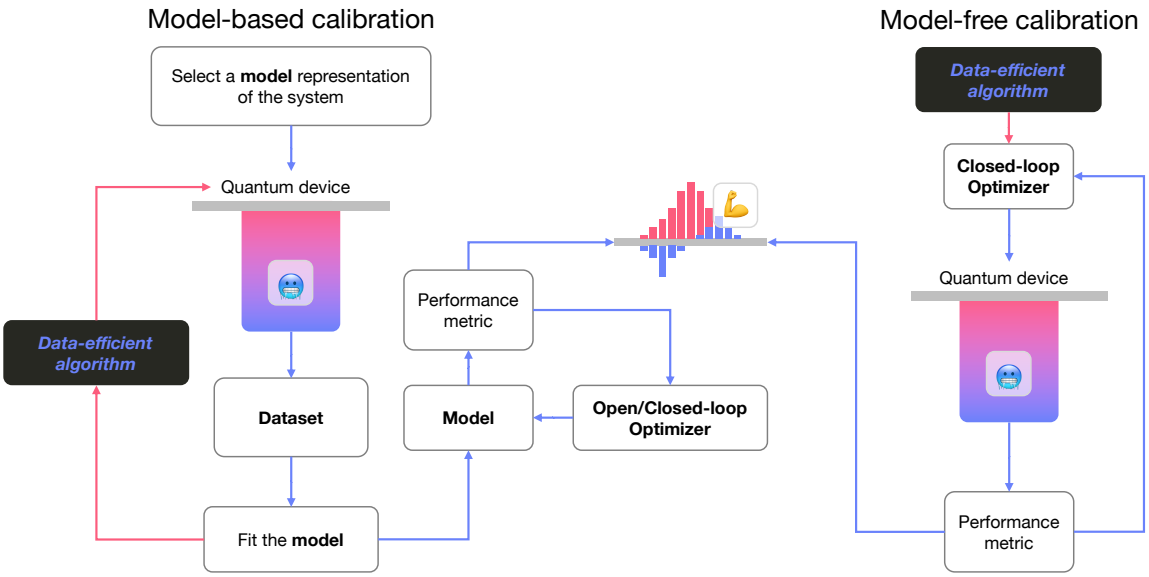


Figure 1.1: A basic illustration of the model-based and model-free calibration workflow, whose goal is to obtain optimal control. With the model-based approach, we start by selecting a model representing the system. We then collect a dataset to characterize the model. At this stage, it is possible to close the loop by introducing a data-efficient algorithm into the workflow, which can potentially reduce the amount of data needed from the system for characterization. We can then optimize for the optimal control by interacting with the model offline. For the model-free approach, we may instead start by selecting a closed-loop (blackbox) optimizer and optimize for the optimal control by interacting with the system directly.

We expect to support even more characterization and calibration techniques, including, but not limited to, reinforcement learning-based algorithms, circuit-level-based algorithms, in the near future. Nonetheless, our current capacities open up several paths for the implementation of the maintenance algorithm and system.

1.3. Related works

In this section, I will discuss some key related works. I will discuss other related existing characterization and calibration methods in more detail in Section 2.2. A direct quantitative comparison between these methods is difficult, as it depends on many factors, such as the choice of control, assumptions, the system model, and the target model parameters. Nonetheless, to minimize QPU downtime and C&C usage, we can discuss several aspects.

Boulder Opal [56] is a characterization and calibration software in the form of a Python library interface to the cloud computing resource of Q-CTRL. The software offers

an efficient data-collection algorithm based on a Bayesian particle filter that selects the pulse envelope (experimental design) that minimizes a cost function based on the anticipated posterior covariance. The research [57] accompanies the software, primarily focusing on selecting the pulse envelope that minimizes the posterior sample covariance in some sense. In that regard, the algorithm seems to limit the possible experimental design to the possible form of pulse. Nonetheless, I expect that the generalization to other forms or combinations of control is possible. On the other hand, BOED does not enforce the choice of control action. In fact, BOED is a general-purpose framework for optimal experimental design applicable across many situations. More importantly, Boulder Opal is a closed-source commercial software, as of the time of writing this dissertation, that would require payment for heavy computation. Inspeqtor is open-source software that performs computation locally. It is especially useful when we need to perform C&C across multiple QPUs, such as in a quantum repeater network, without waiting for a single endpoint.

In [58], the author introduces a technique for efficient characterization, calibration, and correction for quantum information processors. They show how one can perform Gate Set Tomography and Robust Phase Estimation using a Kalman filter by making an approximation for fast parameter estimation. Once the model parameters are identified, the control is determined by using a Linear, Quadratic, Gaussian (LQG) control framework. This approach is particularly useful when the computational cost of classical computation is lower than that of the particle filter and BOED approaches. However, it is necessary to verify that the LQG assumptions hold for the technique to operate properly. When estimating physical parameters, if their distribution does not necessarily follow a Gaussian distribution, a general approach is needed. This work also does not consider an optimal experimental design when performing characterization. From a software perspective, this work presents a methodology and does not include a library for direct use. Similarly to [59], which uses BOED with a normalizing flow to estimate the posterior distribution and inform the experimental design, although the source code is available, it is not a library ready to use.

In the end, these techniques are, in fact, a form of Bayesian inference. Even though it is a rigorous way to update our knowledge of the hidden variable upon a new observation, the computation is generally expensive. Techniques such as the Kalman filter, particle filter, variational inference (primarily used in this dissertation), and normalizing flows are used to aid computation, each with its own advantages. Inspeqtor is a framework that provides building blocks for a complete solution. We did not impose a limit on the methodologies in each stage. The initial workflows and techniques implemented in Inspeqtor serve as a starter pack, with straightforward extensions and customization.

1.4. How to read this dissertation

TL;DR
At the beginning of some chapters, we use the **TL;DR** box to summarize the section. The aim is to convey the messages of the section as fast as possible for the impatient reader.

This dissertation aims to explain and demonstrate the data-efficient characterization of quantum device framework packaging into `inspeqtor`. The design decisions and examples are discussed throughout the dissertation. Thus, the actual `python` code snippet is presented instead of pseudo-code.

This dissertation is structured as follows,

📚 **Chapter 2. Background: Before the Journey Begins** In this chapter, we review quantum mechanics, traditional quantum device characterization and calibration, and the previous works on efficiently performing characterization and calibration on quantum devices.

🗺️ **Chapter 3. Theoretical methodologies** In preparation for efficiently characterizing the quantum device, we analyze the statistical properties of the (statistical) predictive model (a.k.a., replica model) to identify that it is possible to characterize an optimal model with realistic measurable quantities from a real device. We then perform numerical and experimental study to verify our analytical results. We then construct a probabilistic predictive model from the statistical version, which is compatible with the BOED framework.

🥶 **Chapter 4. Beyond a Single Processor** We will discuss some of the motivativing use cases of the characterization and calibration framework in a system beyond a single quantum processor such as in quantum network.

✨ **Chapter 5. Design and Implementation of Inspeqtor** Inspeqtor is both the name of the framework and the library implementing the framework. In this chapter, we discuss the core idea, design, and how we implement it as a Python library.

🚀 **Chapter 6. Numerical Experiments with Inspeqtor** Built on top of the solid foundation, we demonstrate the characterization experiments using `inspeqtor`. Using the qubit characterization task, we first apply the basic statistical Graybox method, then the probabilistic enhancement. We then demonstrate the framework's potential by performing data-efficient characterization.

🌙 **Chapter 7. Next Steps Forward** We finally conclude and discuss our results thus far, and outline possible directions in this chapter.

Chapter 2. Background: Before the Journey Begins

TL;DR
Venturing into unknown territory without the necessary knowledge is unwise. Thus, we first review the foundations on which we will build. In Section 2.1, we review the basic mathematical formalism used in quantum information, along with its connection to the practical use of current quantum computers. In Section 2.2, we then discuss how to make a quantum computer operate properly in practice. Finally, we review the existing C&C frameworks in Section 2.3, as well as the notion of "data-efficient" that we adopt, and existing approaches in Section 2.4.

2.1. Quantum Information

2.1.1. Quantum State

In a nutshell, quantum information [60], [61] is stored in a **quantum state**. In a digital quantum computer, a qubit is a two-level quantum system that serves as the smallest unit of memory. Similar to a classical bit, this two-level system can be in the states zero and one. Mathematically, the quantum states corresponding to zero and one are represented by the vectors,

$$0 \rightarrow \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \text{and } 1 \rightarrow \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{1}$$

In a more compact representation, we use bra-ket notation to represent a quantum state. Thus, the quantum states representing bit zero and bit one are

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle, \text{and } \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle. \tag{2}$$

However, a quantum state is not necessarily in either the zero or one state. It can be a linear combination or **superposition** of the zero and one bases (states). Thus, the state of a single qubit is defined as

$$|\psi\rangle = \begin{bmatrix} a \\ b \end{bmatrix} = a\begin{bmatrix} 1 \\ 0 \end{bmatrix} + b\begin{bmatrix} 0 \\ 1 \end{bmatrix} = a|0\rangle + b|1\rangle; a, b \in \mathbb{C}, \tag{3}$$

where $|a|^2 + |b|^2 = 1$. The quantum state in Equation 3 is a linear combination of the computational basis states (zero and one), which span the vector space $\mathbb{C}^2$. However, there are other possible bases that also span the vector space $\mathbb{C}^2$. For example,

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \text{and } |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle). \tag{4}$$

One way to extract information from a qubit is to measure it projectively. Projective **measurement** collapses the superposition state and causes the quantum state to be randomly projected onto either the basis state, with probabilities $|a|^2$ and $|b|^2$, respectively.

We will discuss the measurement of a quantum state in more detail shortly. First, we introduce a more general mathematical representation of a quantum state. Ket notation, however, can describe only a pure quantum state in a closed system, that is, one with no interaction with the environment. In an open system, the quantum state interacts with the environment, and a more general description is given by the density matrix,

$$\rho = \sum_i p_i |\psi_i\rangle\langle\psi_i|, \tag{5}$$

where p_i is the classical probability associated with the pure state $|\psi_i\rangle\langle\psi_i|$ and the bra notation $\langle\psi| = |\psi\rangle^\dagger$ denotes the complex conjugate transpose of the pure quantum state $|\psi\rangle$. Consequently, $|\psi_i\rangle\langle\psi_i|$ is the outer product of the vectors $|\psi\rangle$ and $\langle\psi|$, which forms a matrix. The basis states must span the vector space, and the probabilities must satisfy $\sum_i p_i = 1$.

Projective measurement of the mixed state also cause the quantum state collapse, yielding a classical bit measurement result (readout) with the following probabilities:

$$\Pr(i) = \mathrm{Tr}(\Pi_i \rho), \tag{6}$$

where $\mathrm{Tr}(A)$ is a trace of a matrix A, and Π_i is a projector operator corresponding to the readout i. For example, if we measure the qubit in the computational basis, the projectors corresponding to the 0 and 1 are,

$$\Pi_0 = |0\rangle\langle 0|, \text{and } \Pi_1 = |1\rangle\langle 1|. \tag{7}$$

Depending on the readout, the mixed state ρ changes to the collapsed state ρ', as follows,

$$\rho \to \rho' = \frac{\Pi_i \rho \Pi_i}{\mathrm{Tr}(\Pi_i \rho)}. \tag{8}$$

In many applications, we often need to extract information from the quantum state more than just the measurement result. For example, we might want to extract the expectation value of some quantum observable $\hat{O}$, which is a Hermitian operator. The expectation value of the observable $\hat{O}$ with respect to the quantum state ρ can be calculated as

$$\left\langle \hat{O} \right\rangle = \mathrm{Tr}\left[\hat{O}\rho\right]. \tag{9}$$

To represent a system of more than one qubit, we must expand our space. Let us consider a system of two qubits ρ_1 and ρ_2, prepared separately. The total quantum state ρ_{12} of the two-qubit system can be written as the tensor product ($\otimes$) as follows,

$$\rho_{12} = \rho_1 \otimes \rho_2, \tag{10}$$

or in the case of pure states, where $\rho_i = |\psi\rangle\langle\psi|_i$,

$$|\psi\rangle_{12} = |\psi\rangle_1 \otimes |\psi\rangle_2. \tag{11}$$

The quantum state in Equation 10 is a product state. We usually omit the tensor product symbol and write it as $|\psi\rangle_1|\psi\rangle_2$ for simplicity. However, there are quantum states that cannot be written as a product state, which are called **entangled** states. For example, the Bell state

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle). \tag{12}$$

Entanglement is a unique property of quantum mechanics with no classical counterpart. It plays a crucial role in quantum computation and quantum communication. In computation, entanglement can be used to detect errors in quantum error correction or to provide a resource for measurement-based quantum computation. For example, quantum teleportation allows an unknown quantum state to be transmitted from one location to another using a pair of entangled qubits and classical communication, though unfortunately not faster than light. Communication is secure because of the **no-cloning theorem**, which states that an unknown quantum state cannot be copied.

We can verify that there is no hidden local variable that produces the effect of entanglement using the Clauser-Horne-Shimony-Holt (CHSH) test [62]. The test checks whether a quantum system, such as a Bell pair, produces expectation values for selected observables that violate the CHSH inequality,

$$S = \langle AB\rangle - \langle AB'\rangle + \langle A'B\rangle + \langle A'B'\rangle \leqslant 2, \tag{13}$$

where A, A', B, B' are observables. If the inequality is violated, it verifies that no hidden local variable is at play in the experiment. A maximally entangled state maximally violates this inequality with $S = 2\sqrt{2}$.

2.1.2. Quantum Operation

At a high level, the quantum circuit abstraction is a way to express and reason about the operations of a quantum computer. This abstraction is a direct analogy to a classical circuit, which is a building block of operation in a classical computer. The quantum circuit is composed of quantum logic gates, as is the classical circuit. We can think of the quantum circuit as a set of instructions that manipulate the quantum state. A quantum

logic gate can be represented mathematically by a unitary matrix. To perform an arbitrary computation on a quantum computer, a universal gateset is needed. There are several choices of the universal gateset. The elementary operations required for a universal gate are typically chosen from single and two-qubit gates.

Examples of single-qubit gates are

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, H = \frac{1}{2}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \tag{14}$$

which are 3 Pauli gates and a Hadamard gate, respectively. The Pauli X gate, also known as the NOT gate, flips the qubit's state. The Z gate, also known as the phase-flip gate, flips the phase of the quantum state. The Y gate combines the effects of the X and Z gates, flipping both the state and the phase of the quantum state. The Hadamard gate can create and destroy superposition. For example, applying the Hadamard gate to the zero state creates the superposition state $|+\rangle$ as follows,

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle. \tag{15}$$

Applying the Hadamard gate to the $|+\rangle$ state destroys via **interference** the superposition and returns it to the zero state,

$$\begin{aligned} H|+\rangle &= \frac{1}{\sqrt{2}}(H|0\rangle + H|1\rangle)) && (16.1)\\ &= \frac{1}{\sqrt{2}}\left(\left(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\right) + \left(\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right)\right) && (16.2)\\ &= \frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}(2|0\rangle)\right) && (16.3)\\ &= |0\rangle. && (16.4) \end{aligned}$$

Moreover, the Hadamard gate can be used to change the measurement basis. We also have parameterized single-qubit gates, such as the rotation gate $R_{X(\theta)} = \exp\left(-i\theta\frac{X}{2}\right)$, which rotates the quantum state around the x-axis of the Bloch sphere by an angle θ. In a matrix form, the rotation gate can be written as

$$R_X(\theta) = \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -i\sin\left(\frac{\theta}{2}\right) \\ -i\sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{bmatrix}. \tag{17}$$

For the two-qubit gate, the prime example would be the controlled-X gate,

$$\mathrm{CX} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{18}$$

The controlled-X (CNOT) gate applies an X gate to the target qubit when the control qubit is in the state $|1\rangle$, and leaves the target qubit unchanged when the control qubit is in the state $|0\rangle$. The CNOT gate is a fundamental two-qubit gate that can both create and remove entanglement between qubits. For example, applying the CNOT gate to the state $|+\rangle|0\rangle$ produces the Bell state $|\Phi^+\rangle$. Applying the CNOT gate to the Bell state $|\Phi^+\rangle$ removes the entanglement and returns the system to the state $|+\rangle|0\rangle$.

Regardless of which unitary operator represents the quantum gate, the transformation of the quantum state can be calculated by applying that operator to the state. The transformation of the quantum state by the quantum gate $\hat{U}$ can be calculated by matrix multiplication as follows,

$$|\psi_f\rangle = \hat{U}|\psi_i\rangle, \tag{19}$$

for the pure state case. In the case of density matrix formalism, the transformation can be calculated as,

$$\rho_f = \hat{U}\rho_i\hat{U}^\dagger. \tag{20}$$

However, in the presence of an environment, a unitary operator cannot fully describe a possible transformation caused by noise. Instead of limiting ourselves to ideal operation, we can also incorporate noise in the calculation. In that case, we refer to this transformation as a quantum channel. Mathematically, a quantum channel could be noisy. In a simple representation, we introduce a set of Kraus operators $\{K_0, ..., K_i\}$ that satisfies $\sum_i K_i^* K_i = I$, the quantum channel transforming state ρ is

$$\rho_f \to \Phi(\rho) = \sum_i K_i \rho K_i^*. \tag{21}$$

2.1.3. From Hamiltonian to Final Quantum State

To achieve finer control over the quantum state, we need to understand the underlying dynamics of the quantum system. At a lower level than the quantum circuit model, the dynamics of the quantum state are governed by a differential equation (DE). In a closed system, the dynamic of the quantum state is governed by the Schrödinger equation, and the Hamiltonian $\hat{H}$ is the operator that describes the total energy of the system. For instance, in a single-qubit system, the ground state and the excited state of the qubit are

separated by an energy gap $E = \hbar\omega_q$ where ω_q is the frequency energy splitting of the qubit. The Hamiltonian may be written as

$$\hat{H} = \frac{\hbar\omega_q}{2}\hat{\sigma}_z. \tag{22}$$

where $\hat{\sigma}_z$ is the Pauli Z operator. We note that we will also denote the Pauli operators as $\hat{\sigma}_x, \hat{\sigma}_y, \hat{\sigma}_z$ and may use them interchangeably. To manipulate the quantum state $|\psi(t)\rangle$, i.e., apply quantum gates, we can add a control term to the Hamiltonian. For example, in the case of a superconducting qubit, the control term is a microwave pulse that can be represented as

$$\hat{H}(\mathbf{\Theta}, t) = \frac{\hbar\omega_q}{2}\hat{\sigma}_z + \Omega_d s(\mathbf{\Theta}, t)\hat{\sigma}_x, \tag{23}$$

where Ω_d is the control strength, and $s(\mathbf{\Theta}, t)$ is the control signal, which is a function of the control parameters $\mathbf{\Theta}$ and time t. Thus, given a Hamiltonian describing the system, $\hat{H}(t)$ (where we drop the explicit dependence on the control parameters for simplicity), the quantum state $|\psi(t)\rangle$ can be calculated by solving the Schrödinger equation in the form,

$$i\hbar \frac{\partial}{\partial t}|\psi(t)\rangle = \hat{H}(t)|\psi(t)\rangle. \tag{24}$$

Alternatively, in this dissertation, the unitary operator $\hat{U}(t)$ is often preferred and can be found by solving

$$i\hbar \frac{\partial}{\partial t}\hat{U}(t) = \hat{H}(t)\hat{U}(t). \tag{25}$$

In the quantum circuit abstraction, the unitary gate transforms the quantum state instantaneously. However, in the physical implementation, the transformation of the quantum state is not instantaneous, and the quantum state evolves through time. Generally, the solution is given as a time-ordered operator

$$\hat{U}(t) = \mathcal{T}_+ \exp\left\{-i\int_0^t \hat{H}_{\text{total}}(s)ds\right\}. \tag{26}$$

The time ordering operator $\mathcal{T}_+$ is needed because the Hamiltonian at different times may not commute. A closed-form solution is hard to find; we typically need to use numerical methods to solve it, which usually leads to an approximate solution. The unitary operator $\hat{U}(t)$ at the final time t_f corresponds to the quantum gate that we want to implement. We can then apply the unitary operator to the initial quantum state to find the final quantum state, as shown in Equation 20.

In an open system, a different differential equation is needed to describe its dynamics. For instance, a Lindblad master equation is needed to solve for the quantum state in the form of a density matrix $\rho(t)$. The master equation [63], [64] is a differential equation of the following form,

$$\frac{\partial}{\partial t}\rho(t) = -\left(\frac{i}{\hbar}\right)\left[\hat{H}_{\text{ideal}}(t), \rho(t)\right] + \sum_i \left(\hat{C}_i \rho(t) \hat{C}_i^\dagger - \frac{1}{2}\left\{\hat{C}_i^\dagger \hat{C}_i, \rho(t)\right\}\right). \tag{27}$$

The collapse operator $\hat{C}_i$ describes noise, such as amplitude damping and dephasing.

2.1.4. Noise in the quantum system

Noise refers to the undesired factors that affect the dynamics of a system. In the quantum regime, there are several ways to quantify system noise. Regardless of the actual physical process that generates the noise, we can model it in the form we are interested in, which might be easier to deal with.

One of the most fundamental types of noise is Pauli noise. The noise in this system is **characterized** in terms of a quantum channel where the Kraus operators $\hat{K}$ are a set of Pauli matrices and the Identity matrix. More formally, a Pauli channel is as follows,

$$\rho_f \to \Phi(\rho) = p_0\rho + p_1\hat{X}\rho\hat{X}^\dagger + p_2\hat{Y}\rho\hat{Y}^\dagger + p_3\hat{Z}\rho\hat{Z}^\dagger \tag{28}$$

where $\{p_0, p_1, p_2, p_3\}$ are probabilities that the operators $\left\{\hat{I}, \hat{X}, \hat{Y}, \hat{Z}\right\}$ are applied on the quantum state ρ.

Modeling the noise as a Pauli channel is simple, yet powerful. Multiple error management techniques assume that system noise can be modeled as a Pauli channel—for example, quantum error correction and entanglement purification. The analysis using this model often performs at the circuit level. For instance, the noise is modeled as applying a Pauli gate randomly with some probability, allowing for a simulation of a large-scale quantum system with noise. This modelling approach does not specify the underlying physical process responsible for the Pauli noise, but models the system's effective noise as a Pauli channel.

One can model the noise at a lower level closer to physical implementation, in constrast to the Pauli channel that can be used regardless of the physical realization of the qubit. At this level, detailed implementation of the experiment can be also taken into the consideration. For instance, we can start from the mathematical description of the dynamics of the quantum system, i.e., the Hamiltonian, the noise can be inserted and modeled at that level. Consequently, the solution might depends on the noise model, which can be used for further analysis, e.g., an optimal control problem. However, the

analytical solution of an arbitrary noise model is generally hard, if not impossible, to solve for. Thus, there is a need for a numerical method.

Nonetheless, **detuning** is the noise model for which the solution of the system can be solved analytically in the simple case. For example, consider the following transmon qubit Hamiltonian with a control term as in Equation 23. In `qiskit` [65], the signal is defined as a functional of a function of control envelope $h(\mathbf{\Theta}, t)$ and drive frequency ω_d and phase φ as,

$$s(\mathbf{\Theta}, t) = \mathrm{Re}\{h(\mathbf{\Theta}, t)e^{i(2\pi\omega_d t+\varphi)}\}. \tag{29}$$

Detuning can be modeled as a deviation of the drive frequency ω_d from the qubit frequency ω_q. Thus, the detuning can be represented as a coefficient $\delta = \omega_d - \omega_q$. Detuning can affect the system's dynamics in a nontrivial way. The noise can be mitigated using control calibration techniques such as Derivative Removal by Adiabatic Gate (DRAG). Furthermore, one can use a specialized optimal control solution to mitigate the noise [66].

In the quantum open system, the noise is not limited to the unwanted unitary transformation, i.e., unwanted energy transfer within the system, but also the system-environment energy transfer [67], [68], [69]. In the Bloch-Redfield model, the noise may be characterized by two quantities: (1) a longitudinal relaxation rate,

$$\Gamma_1 = \frac{1}{T_1}, \tag{30}$$

and (2) a transverse relaxation rate,

$$\Gamma_2 = \frac{1}{T_2} = \frac{\Gamma_1}{2} + \Gamma_\varphi, \tag{31}$$

where Γ_φ is the pure dephasing rate. The effect of the noise on the quantum state in Equation 3 can be modeled in the Bloch-Redfield density matrix as,

$$\rho_{\mathrm{BR}} = \begin{pmatrix} 1 + (|a|^2 - 1)e^{-\Gamma_1 t} & ab^* e^{i\delta\omega t} e^{-\Gamma_2 t} \\ a^* b e^{-i\delta\omega t} e^{-\Gamma_2 t} & |b|^2\ e^{-\Gamma_1 t} \end{pmatrix}. \tag{32}$$

The system can also be affected by **stochastic** noise, such as colored noise, which is characterized by a power spectral density (PSD). This type of noise is common in solid-state qubits, particularly 1/f noise, a low-frequency noise source [70] that may lead to pure dephasing if it exists along the Z-axis of the qubit Hamiltonian. Stochastic noise can also exist along other axes. In general, we may classify noise as **additive**, which is independent of the control signal, or **multiplicative**, which depends on the control signal [71], [72]. This type of noise can be characterized and mitigated by specialized control such as Dynamical Decoupling (DD) pulse sequences [68], [73], [74].

In a qubit system realized from the energy levels of an atom, the state of the qubit is defined by two of its energy levels, e.g., the ground state can be defined as $|0\rangle$, and the first excited state as $|1\rangle$. The physical realization of control relies on the precise specific frequency. However, in the case that the frequency of the control field is deviated from the perfect value, the quantum state might transfer to the undesired level. This noise is called **leakage**. A specialized technique that can mitigate leakage noise is the DRAG pulse [75], [76], [77].

Crosstalk is a type of noise in which the operation of one qubit affects another. This type of noise occurs in systems where qubits are connected to each other through some interaction [78], [79]. For example, in a silicon spin qubit system [80], the Rabi frequency of a driven qubit changes when the drive of a neighboring qubit is turned on. For superconducting qubits, crosstalk can be caused by always-on coupling between qubits or by frequency collisions [81]. Techniques to mitigate crosstalk include careful calibration of the controls [82] and architectural design of the system [81].

In the experimental setup, the measurement result from the quantum system is binary. However, a misclassification can occur, flipping the bit $0 \to 1$ and $1 \to 0$. This can be modeled as a probability of classical bit-flip error. Several error-mitigation techniques [83], [84], [85] are proposed to mitigate measurement errors.

2.1.5. Fidelity

In quantum information, we are often interested in how close a quantum state is to another. The measure of the overlap between two quantum states is state fidelity. The state fidelity between quantum state ρ and σ is

$$F(\rho, \sigma) = \mathrm{Tr}\left[\sqrt{\sqrt{\rho}\sigma\sqrt{\rho}}\right]^2. \tag{33}$$

In the case where $\rho = \sigma$, the state fidelity $F(\rho, \sigma) = 1$, indicating that these two states are the same. In the case where $\rho = |\psi_\rho\rangle\langle\psi_\rho|$ is a pure state, then $|\psi_\rho\rangle\langle\psi_\rho|^2 = |\psi_\rho\rangle\langle\psi_\rho|$ and the expression above can reduce as,

$$F(\rho, \sigma) = \mathrm{Tr}\left[\sqrt{|\psi_\rho\rangle\langle\psi_\rho|\sigma|\psi_\rho\rangle\langle\psi_\rho|}\right]^2; \tag{34.1}$$

$$= \langle\rho_\psi|\sigma|\rho_\psi\rangle\,\mathrm{Tr}[|\psi_\rho\rangle\langle\psi_\rho|], \tag{34.2}$$

$$= \langle\rho_\psi|\sigma|\rho_\psi\rangle, \tag{34.3}$$

$$F(|\psi_\rho\rangle, \sigma) = \langle\rho_\psi|\sigma|\rho_\psi\rangle \tag{34.4}$$

In the case where $\rho = |\psi_\rho\rangle\langle\psi_\rho|$ and $\sigma = |\psi_\sigma\rangle\langle\psi_\sigma|$ are pure states, then state fidelity reduces to

$$F\big(|\psi_\rho\rangle, |\psi_\sigma\rangle\big) = |\langle\psi_\rho|\psi_\sigma\rangle|^2 \ . \tag{35}$$

Fidelity is not limited to the quantum state; we can also measure the closeness between quantum channels. The process fidelity measuring the closeness of quantum channel $\hat{U}$ and $\mathcal{E}$ is

$$F_{\text{process}}\big(\hat{U}, \mathcal{E}\big) = \text{Tr}\,\frac{\left[S_{\hat{U}}^\dagger S_\mathcal{E}\right]}{d^2}, \tag{36}$$

where $S_{\hat{U}}$ and $S_\mathcal{E}$ are a superoperator representation of operator $\hat{U}$ and $\mathcal{E}$ respectively, and d is the number of basis states of the channel. Process fidelity is especially useful when one needs to measure the quality of a quantum channel in an experiment.

In the context of control calibration, we might be interested in the low-level control of a quantum device that maximizes an **Average Gate Fidelity (AGF)** of target quantum gate and quantum channel. Letting $\hat{U}$ be target a unitary gate and the quantum channel $\mathcal{E}$ realized in experiment then AGF is defined as

$$\bar{F}\big(\mathcal{E}, \hat{U}\big) = \int d\psi \langle\psi|\hat{U}^\dagger \mathcal{E}(|\psi\rangle\langle\psi|)\hat{U}|\psi\rangle. \tag{37}$$

We can write AGF in a form that is easier to collect in the real experiment. From the Equation. 19 in [86], we set $d = 2$ in a single qubit case, and chose to measure the qubit in Pauli bases, then AGF can be written as

$$\bar{F}\big(\mathcal{E}, \hat{U}\big) = \frac{1}{2} + \frac{1}{12}\sum_{j=1}^{3}\sum_{k=0}^{1}\sum_{m=1}^{3} \alpha_{j,k} b_{m,j}\,\text{Tr}\left[\hat{P}_m \mathcal{E}\big(\rho_{j,k}\big)\right], \tag{38}$$

where $\hat{P}_j, \hat{P}_m \in \left\{\hat{X}, \hat{Y}, \hat{Z}\right\}$. $|\psi\rangle\langle\psi|_{j,k}$ are the basis states, $|+\rangle, |-\rangle, |i\rangle, |-i\rangle, |0\rangle, |1\rangle$. The index $k \in \{0, 1\}$ corresponds $\{+1, -1\}$ eigenvalues. The coefficients are $\alpha_{j,0} = 1, \alpha_{j,1} = -1$ for any j Pauli observable and $b_{m,j} = \frac{1}{2}\,\text{Tr}\left[\hat{U}\hat{P}_j^\dagger\hat{U}^\dagger\hat{P}_m\right]$.

2.2. Characterization & Calibration (C&C)

2.2.1. Calibration

Multiple physical realizations of quantum devices are available nowadays. Currently, there is no clear winner in building a quantum computer, in the sense that with CMOS, a transistor is the main way to build a bit in a traditional computer. Regardless of the implementation choice, the physical operation of that platform used to realize quantum operation is most likely to be imperfect due to noise. For example, in the superconducting qubit platform [68], the quantum state is controlled by a microwave pulse aimed at the qubit. Not just any control pulse can control the qubit, but it must be a pulse at a specific

frequency. Furthermore, to rotate the state from the initial to the final position, control parameters such as duration, amplitude, and pulse axis play an important role.

In theory, the control parameters may be analytically calculated. For example, in the noiseless case of a superconducting single-qubit control, we can select the envelope of the control and the parameters such that the integral over the envelope function is π to produce a Pauli-X gate. But in practice, due to the noise, the theoretical ideal set of control parameters might not work perfectly as expected. In general, errors can be compensated for with careful adjustment of the control parameters. This process of adjusting the control to achieve some objective is referred to as control **calibration**. This process can be categorized by the need to interact with the target device to be calibrated [87]. First, a **closed-loop** approach in which the algorithm decides based on feedback from the target device, eliminating the need for interaction. Second is the **open-loop** approach, where the interaction with the target device is not needed. Instead, the algorithm will query a mathematical representation, a model of the target device. The advantages and disadvantages of these two approaches will be discussed in Section 2.2.1.1 and Section 2.2.1.2. The calibration approaches may be categorized into **model-free** approaches, where we make (almost) no assumptions about the target system, and **model-based** approaches, where we use a (partial) model of the target system's mechanism and noise. Previous studies have also attempted to get the best of both worlds, which we called the hybrid approach and discuss in Section 2.2.1.3. In any case, the objective of the calibration is to minimize or maximize some cost function. Typically, those cost functions are defined as a function of fidelity, which is the quantity that indicates the closeness to the target quantum state or operation. Further extension might include minimizing the duration of the control as an additional objective.

As mentioned, the open-loop approach does not require direct interaction with the target device, but instead interacts with its mathematical representation. The construction of such a representation, however, introduces additional complications. A mathematical **model** of the target device is not easily obtained, especially if accuracy is required. One of the more straightforward ways to construct the model is to define a parameterized function that represents the dynamics of the system, including noise. Experiments can then be performed to estimate the **system parameters**. This process of estimating the system parameters is referred to as **characterization** and will be discussed in Section 2.2.2.

2.2.1.1. Closed-loop Calibration

The most alluring feature of the closed-loop approach is the lack of need for extensive prior knowledge of the system. We refer to this type of algorithm as a model-free algorithm, such as reinforcement learning (RL) [88], [89], [90], [91], Bayesian optimization [92], [93], [94], Nelder-Mead (standard Blackbox optimization algorithm). These

algorithms rely on different techniques; however, the goal remains to optimize the parameters that maximize or minimize a cost function. We will now discuss the mainstream approaches in closed-loop calibration.

Reinforcement learning is a type of machine learning that trains the neural network policy of agents to perform tasks through trial and error. The algorithm allows agents to take a sequence of actions according to a policy while observing the environment's state and the reward for each action. At the end of the sequence, the policy is updated to maximize the reward in the next sequence. RL has been widely used in robotics applications.

In the context of control calibration of a quantum device, each element of RL may be defined as follows. The environment is a target device, including noisy conditions. The action is an allowed control of the target device; for example, Baum et al. [90] defined the action as the amplitude at each time step of the control sequence. The reward is the value of the fidelity, which may be penalized by the duration of the control. In other words, the agent is trained to perform control to maximize the fidelity as fast as possible.

The model-free nature of RL can eventually become a disadvantage if care is not taken. As the space of action grows, the number of interactions with the target device would become an unbearable cost. This is when the concepts of exploration and exploitation become crucial, i.e., whether to explore an unknown territory or fine-tune the known to the fullest. The right balance must be struck, otherwise, the cost of exploration will go further up than the risk of finding a better region. On the other hand, the right answer might reside right next to the explored space, awaiting just a tiny tweak to discover it. In the calibration task, we have a choice to choose a simple control with a tiny, limited search space or a far larger space that allows for fine-grained control. Constraints can be made to limit the possibilities, mostly due to the limitations of the target device, making it possibly easier for the RL agent to find the right answer faster.

Bayesian Optimization is typically employed for problems where the cost function is costly to evaluate. The concept is to model the given cost function as a Gaussian process. Upon observing new data, the model is updated. Then, the acquisition function decides based on a metric, such as expected improvement evaluated using a Gaussian model, the next point to evaluate. The steps are repeated until the budget runs out or terminal conditions are met. The simplified concept of Bayesian optimization is to evaluate the cost function at a point with some uncertainty, which may contain a critical point to be optimized; thus, it is well-suited for a task with a limited budget. At a high level, the acquisition function decides where to evaluate; this somewhat aligns with the concepts of exploration and exploitation in RL. One of the biggest disadvantages of Bayesian optimization is that it is not good at a large number of parameters.

In essence, the closed-loop, or model-free, calibration approach is an instance of an optimization problem in which the cost function is a black box that can only be evaluated by observing the output for a given input. Thus, standard black-box optimization algorithms can be used for the calibration task. Such algorithms include, but are not limited to, Nelder-Mead, COBYLA, Powell's method, and evolutionary algorithms [95]. For more details, we refer the reader to Chapter 7 of M. J. Kochenderfer and T. A. Wheeler [96].

2.2.1.2. Open-loop Calibration

One of the greatest strengths of calibration via an open-loop approach is the ability to leverage the gradient direction toward optimal control parameters. This is possible because the target system is modeled as a parametrized function. Different techniques can then be used to calculate the gradient of the function represented by the target device. One of the first gradient-based algorithms for optimal control problems for quantum devices is Gradient Ascent Pulse Engineering (GRAPE) [97]. Many of its variations have been developed and reviewed in [98].

Another gradient-based approach is Gradient Optimization of Analytic conTrols (GOAT) [99]. These two approaches differ from each other in the method of gradient computation of system evolution with respect to some parameters.

With recent developments of deep neural networks, the computation of gradients on neural networks can be done efficiently. Hence, there is a study that uses a recently proposed architecture, Physics-Informed Neural Network (PINN), to study the problem of quantum optimal control [100].

Nonetheless, the open-loop approach requires characterization of the quantum device. Moreover, its performance depends heavily on the accuracy of the model, which in turn depends on the accuracy of the model parameters. Calibrating with mismatched model parameters may be viewed as equivalent to calibrating a different device. Thus, careful system characterization is crucial for the open-loop approach.

2.2.1.3. Hybrid Approach Calibration

Recent studies have explored the hybridization of open and closed loops. One of the approaches is to begin with open-loop optimization to produce a sub-optimal solution, then employ closed-loop optimization to enhance the solution further [99], [101], [102]. Another approach is to repurpose the data from characterization experiments to get a relatively good initial state, and iteratively update with data from experiments, such as those done in [103], by using the concept of Bayesian Optimization. It is also worth mentioning that R.-B. Wu, B. Chu, D. H. Owens, and H. Rabitz [104] uses experimental data to help in the calculation of gradient-based optimization. However, doing so requires

that the experiment be executed upon request from the optimizer. Again, these studies did not focus on signaling the need for the recalibration.

2.2.2. Characterization

Characterization is the process that determines the system or model parameters. Thus, to characterize a device is to first identify the model or the form of a parametrized function that could describe the behavior of the system. Then, experiments have to be done to determine the parameters of the model. Another way to describe characterization is the process of estimating a mathematical object, i.e., "quantum channel", that describes how the system evolves.

We will categorize characterization into two main types depending on the targeting properties. First is channel characterization, where the entire quantum channel is fully described by a mathematical object. Second is parameter characterization, which targets a particular parameter of the aforementioned parametrized model of the system.

For the channel characterization, the most straightforward one is process tomography, which is also the most resource-intensive to perform as the size of the system grows larger [60], [105]. For instance, standard process tomography requires estimating $d^4 - d^2$ independent real parameters, where $d = 2^n$ and n is the number of qubits in the system. Thus, the algorithm scales as $O(16^n)$ with the number of qubits. In an experimental realization, we need to estimate the expectation values of $4^n 3^n$ combinations, depending on the choice of basis [106]. To reduce the number of experimental combinations required, we refer to techniques such as those in [106]. Nonetheless, the method provides complete information of the targeted quantum channel, though distinguishing state-preparation, measurement error (SPAM) from the channel is not considered.

A more advanced tomography method is Gateset tomography [107], which provides a characterization of gate-set and is self-consistent (i.e., taking into account the SPAM error). The former method is also one of the ingredients for the closed-loop optimization, as fidelity can be computed from the quantum channel reconstructed from the process tomography. It is not necessary to perform process tomography if only fidelity is required [86], [108], thereby avoiding the potentially excessive number of measurements that grows exponentially with the number of qubits. Another recent approach models the system such that it takes into account the environment interaction with the system, then encodes the noise into an abstract mathematical function that can be estimated using machine learning techniques [52], the *Graybox* method. I refer to this class of characterization approaches as *parametrized channel characterization*. Note that the parameters here are not the parameters defined in the system model, but rather parameters that describe its corresponding quantum channel.

Parameter characterization assumes an explicit mathematical form for the system model, such as the detuning in the Hamiltonian of the device discussed in Section 2.1.4. Thus, the system parameters have to be estimated directly with specialized methods. For the superconducting qubit that is subject to leakage noise, Wood et al. [109] and Chen et al. [110] proposed methods to characterize the parameters. In a system where interactions between memories are not well controlled, crosstalk can occur. Crosstalk is the process where another memory affects another unintentionally in a destructive way and is being studied in [79]. One of the noises in a charged system is colored noise that is described by the power spectrum of the noise. This can be characterized using methods proposed in [71], [111]. Tornow et al. [112] show that with restless measurement, the characterization process can be faster.

There are several studies about adaptive experiment characterization of quantum devices. Sarra et al. [59] use Bayesian Optimal Experiment Design to estimate system Hamiltonians. Lennon et al. [113] use neural networks and information gain approximation techniques for adaptive measurement, selecting points of maximal information gain to characterize quantum dots, improving measurement efficiency by a factor of four compared to grid search. Fiderer et al. [114] use reinforcement learning to minimize Bayes risk for qubit frequency estimation tasks. Note that these studies focus on the characterization task only.

2.3. C&C Framework

With regard to the existing C&C framework, `C3` [115] is an integrated tool-set to perform control calibration and device characterization using both open-loop and closed-loop techniques. However, it uses a standard optimizer for its algorithms, and it does not focus on being a data-efficient framework. One recent framework, `qibocal` [116], focuses on being **platform-agnostic**, i.e., API remains the same regardless of the target quantum device platform. However, it again does not focus on data utilization efficiency. While QCtrl's Boulder Opal [56] is production-ready software with a study about a data-efficient characterization approach [57], it is a closed-source package. Furthermore, the computation has to be done in the cloud, which may result in delays when high-demand requests arrive simultaneously and may incur undesirable costs if not managed properly. JAXQuantum [117] is a toolkit for quantum hardware design, simulation, and control. According to its documentation, the library is built using JAX. It focuses on the superconducting qubit platform and aims to be a drop-in replacement for QuTiP [63].

2.4. The notion of "Data Efficiency"

The notion of "data-efficient" may be used differently in the literature. For example, Yanjun Ji, M. Roth, D. A. Kreplin, I. Polian, and F. K. Wilhelm [118] repurpose data from previous executions, such as algorithms or benchmarking, to construct a circuit-size-independent predictive model specifically for that family of algorithms, such as QAOA, and VQE. They can then use the predictive model, for instance, the device- and application-aware circuit compilation. In our context, "data-efficient" is an efficient usage of experimental data, i.e., from current data at hand, we want to determine the next batch of experimental data that will help us characterize the device faster. In other words, we want to create a dataset with a minimal sample size that can be used to construct an accurate predictive model.

The concept is indeed possible; that is, we want to perform the experiment where the model is the most uncertain. **Bayesian Optimal Experimental Design (BOED)** [55], [119] is a framework that allows us to bring the concept to reality. Mathematically, we are interested in the posterior model $p(y \mid \theta, d)$ construction, where y is observed data, θ is the latent variable that we seek to infer with as little data as possible, and d is the design of the experiment. The **Expected Information Gain (EIG)** (nats for logalithm with natural base and bits for base 2) of θ with experiment with design d can be defined as the expected difference between the entropy of the prior and the posterior

$$\mathrm{EIG}(d) \triangleq \mathbb{E}_{p(y|d)}[H[p(\theta)] - H[p(\theta|y,d)]] \tag{39}$$

However, it is in general intractable to obtain a closed-form expression for EIG. There are a number of EIG estimators, each with its own tradeoff. We will estimate EIG by using a variational marginal (VM) estimator to estimate EIG [55]. This method is preferred, since θ may be high-dimensional. However, it is a biased estimator if the variational distribution used in the VM estimator does not contain the target distribution. However, it provides a fast convergence. The variational marginal EIG is given by,

$$\mathrm{EIG}(d) \leqslant \mathcal{U}_{\mathrm{marg}}(d) \triangleq \mathbb{E}_{p(y,\theta \mid d)}\left[\log \frac{p(y|\theta,d)}{q_m(y|d)}\right] \tag{40.1}$$

$$\approx \hat{\mu}(d) \tag{40.2}$$

$$\triangleq \frac{1}{N}\sum_{n=1}^{N} \log \frac{p(y_n \mid \theta_n, d)}{q_m(y_n \mid d)}. \tag{40.3}$$

Consequently, in order to estimate EIG using approach propose by A. Foster *et al.* [55], we have to be able to compute the **probabilistic** predictive model (posterior distribution) $p(y|\theta,d)$ and the marginal distribution $q(y|d)$. We can estimate the posterior distribution

in a number of ways. For instance, using Bayesian Neural Network (BNN) [120], [121], [122], normalizing flow [123], [124], and particle filter.

After we estimate the EIG for the given experiment design, we have to decide which experiments to perform to gather data from the device. One simple thing we can do is to perform an experiment where the EIG is maximized. This approach is one-step optimal from an information-theoretic point of view [55]. That is, it does not take into account the future experiment. This simple strategy might not maximize the total EIG of the sequential experiments. Furthermore, this approach requires two-step optimization, which are (1) estimation of the EIG and (2) searching for an experimental design that maximizes EIG. To reduce the computational challenge, A. Foster, M. Jankowiak, M. O'Meara, Y. W. Teh, and T. Rainforth [125] proposes a single-step approach. Later, L. Sarra and F. Marquardt [59] study the approach in [55], [126] to achieve BOED in the context of quantum device characterization. They use normalizing flow to estimate the posterior distribution of the system and the optimal design. Interestingly, some normalizing flow models can be interpreted as a BNN [124].

As discussed earlier, experimenting with a design that maximizes the EIG at the current step does not necessarily maximize the total EIG of the sequential experiments setting. To tackle the problem of sequential experiment design [54], recent studies have proposed the use of Deep Neural Network (DNN) to predict the next experimental design [127], [128]. The DNN is trained by minimizing the lower bound of the total EIG. However, this approach requires significant computational power during DNN training. While at the deployment time, the next design can be efficiently determined. The concept of using DNN to make a decision on what is the next experiment design to perform in the context of quantum characterization is explored in [114], but they did not use the result presented in [127], [128]; instead, they optimize the DNN using Bayes risk (their source code is not easy to use.)

As an alternative to the EIG, T. M. Stace *et al.* [57] proposes the OBSID algorithm for performing a sequential experiment which build in to Boulder Opal [56]. They used a particle filter technique to sample from the posterior distribution. They experiment with designs that minimize cost functions based on anticipated posterior covariance and modified Fisher information. There are some disadvantages in using Fisher information compared to EIG [54]. For example, since the Fisher Information Matrix is a matrix, one has to optimize the summary statistics rather than fully characterize the statistics as EIG. R. S. Gupta, C. L. Edmunds, A. R. Milne, C. Hempel, and M. J. Biercuk [129] also utilize particle filter based method to schedule an efficient sequential experiment and spatial noise mapping.

Nonetheless, care must be taken, as using too few data points (samples) or employing a poorly chosen control parameterization may lead to structural unidentifiability problems [57], [130], [131], [132], [133], in which the model parameters cannot be uniquely determined.

J. P. Marceaux [58] proposes a characterization, calibration, and correction technique for QPU. This work applies a Linear-Quadratic-Gaussian (LQG) control framework to a QPU. The framework assumes a linear system subject to additive Gaussian noise and aims to produce a control that minimizes a quadratic cost function. The authors show how one can use a Kalman filter to perform characterization, such as Gate Set tomography and Robust Phase Estimation, which are self-consistent characterization protocols (i.e., accounting for state preparation and measurement errors). The technique allows us to update the system's state as new observations arrive. With the updated state, we can then update the control using an LQG control law. This work doe I expect this technique to be fast. However, we must verify the LQG assumptions for the technique to operate successfully.

Chapter 3. Theoretical methodologies

TL;DR
In this chapter, I establish the notation for C&C used throughout the rest of this dissertation. I categorize the characterization into statistical and probabilistic approaches and discuss them in Section 3.1 and Section 3.2, using the Graybox characterization method as a toy model. In Section 3.1.4, we perform numerical and experimental studies to verify our analytical results. I then discuss the feasibility of data-efficient approaches that could be used within this framework in Section 3.3.

In a nutshell, it is resource-intensive to achieve optimal control of the target device, whether using an open-loop or a closed-loop approach. In the closed-loop approach, the balance between exploration and exploitation must be chosen, and it is important to keep in mind that this happens on the real device. With the open-loop approach, the accuracy of the characterized model is prominent. Thus, the experimenter's need to characterize the target device remains crucial and potentially costly. A characterized model can be used beyond the optimal control problem; further information can be extracted, and the interaction cost with the target device can be reduced by first interacting with the model in the early stage of development.

Goal

"Data-efficiency" is of utmost importance, meaning we aim to reduce the interaction with the device as much as possible while maintaining accurate characterization of the device.

Throughout this chapter, we will discuss how we can approach the problem and the ways forward to data efficiency, starting from the characterization of a quantum device using statistical machine learning and then moving forward to the probabilistic machine learning approach. Building upon the solid foundation of the statistical version of the model, we then upgrade it with uncertainty quantification capability with the probabilistic version. Finally, the model can be used with BOED, the principled framework for efficient utilization of data [55]

The field of machine learning studies how to model a system of interest without explicitly programming it to make predictions by inferring from observed data. In this dissertation, we would like to categorize the field into statistical and probabilistic approaches. In our context, we are interested in a regression program since we are dealing with the problem of quantum device characterization to construct a **predictive model**.

For the Statistical Machine Learning (SML), we consider a model that learns by minimizing an objective function computed from the observed data, while for Probabilistic Machine Learning (PML), we infer the predictive model from the observed data by using Bayesian inference.

SML is a mature field that has found success across many areas. Compared to PML, it is a faster, more efficient way to construct a predictive model. However, it does not explicitly quantify prediction uncertainty, which is crucial in many applications, such as self-driving cars and medical applications. On the other hand, the probabilistic approach is a mathematical formulation of uncertainty [134]. It is also a common language that we can use to connect with other fields [135]. Moreover, Empirical Risk Minimization (ERM) is equivalent to Maximum a Posteriori (MAP), which gives a natural connection between SML and PML. Nonetheless, we can learn from SML's approach and apply the insights gained to implement the PML approach, which we might view as a natural way to use SML as a prior for the PML. For example, we can use SML as a baseline reference for the PML implementation.

3.1. C&C: Statistical Machine Learning

By itself, SML does not provide a tool that can be used with BOED out of the box, but it is certain to pave the way toward an elegant tool. Also, it is simple, easy to understand, and computationally inexpensive approach for quantum device characterization. Thus, from the development point of view, starting our journey here would be considered a solid step along this path.

In the literature [136], the term **predictive model** is referred to a function that given $\boldsymbol{x}$, predicts output y, formally $y = f(\boldsymbol{x})$, that is constructed from some dataset. In our context, to characterize a quantum device is to construct a **predictive model** from the data obtained from an experiment. From the information point of view, we extract the information of the system via experiment, encode it to a model, then we can predict the output of the target device using the model instead of performing a new experiment on the target device. I will refer to the procedure of estimating a predictive model from dataset by the word train / fit / estimate / construct. Given a large enough dataset, we can fit the model with high accuracy. However, if this requires too much overhead, it becomes impractical. Thus, characterizing quantum device by exploiting our knowledge of quantum mechanics and the system to reduce the resources needed to construct an expensive model is preferred.

One of the simplest ways of characterizing a quantum device is to **assume** a functional form of the device to characterize. The **functional form** of the quantum device may consist of multiple functions. In the context of modeling the system in quantum

mechanics, it is convenient to model the system in terms of a Hamiltonian. In quantum mechanics, a Hamiltonian is the mathematical object that describes the dynamics of the quantum system and its interaction with the environment in the case of an open systems. In our context of characterization, we model the system with a functional form of a Hamiltonian. That is the mathematical form consisting of Hermitian matrices, and the functions that indicate the finer detail. The function may be time dependent, $\hat{H}_1(t)$, or independent, $\hat{H}_0$, or both. Thus, a general form of Hamiltonian functional form $\hat{H}(t)$ is

$$\hat{H}(t) = \hat{H}_0 + \hat{H}_1(t) \tag{41}$$

The functional form above leaves the dependence on the system parameters implicit. In practice, the function may depend on a lot of system parameters, thus, in this dissertation, I would like to be explicit when the parameters are the focus in order to avoid symbol clutter.

Now, I will describe the functions that make up the total Hamiltonian, starting with the most important on. Consider a vector of control parameters $\mathbf{\Theta}$, where the control function $s(\mathbf{\Theta}, t)$ is a time-dependent function. For example, in a superconducting platform, the control function may be a function of the signal which is a function of the pulse envelope itself. The next one is a function of noise that describes how noise interacts with the system. The noise function itself might be a functional of the control function. The function can be chosen freely, given that the total Hamiltonian still remains Hermitian. For example, the time-dependent Hamiltonian might be a controllable envelope acting on the Pauli-X axis $\hat{H}_1(t) = s(\mathbf{\Theta}, t)\hat{\sigma}_X$.

Modeling the device at the level of Hamiltonian is one of the possible ways. However, the Hamiltonian is not the observed object in the experimental setting. The Hamiltonian generates an evolution operator, which can be obtained by solving an appropriate differential equation, either, either the Schrödinger equation in the case of a closed system or beyond like the master equation in the case of open systems. We have two choices of the final product to solve for, (1) the final quantum state ρ_f from ρ_0 and (2) the propagator / unitary $\hat{U}$. Furthermore, since quantum information requires extraction via measurement, the observable $\hat{O}$ has to be chosen. The expectation value of the observable is then

$$\langle \hat{O} \rangle = \mathrm{Tr}\left[\hat{O}\rho_f\right] = \mathrm{Tr}\left[\hat{O}\hat{U}\rho_0\hat{U}^\dagger\right]. \tag{42}$$

The choice between solving for final state or propagator typically depends on the object of interest. For instance, if the main focus is the expectation value of an observable given multiple initial states, computing the Propagator is more efficient since we only have to solve the differential equation once. Computing hte expectation value using final states would require solving the differential equation multiple times for each initial state.

The expectation value expression allows us to be one step closer to the realistic setting. The dependence on the control parameter is implicit for the propagator and final state in the expression. However, it also implies that the expectation value is a function of the control parameters. This opens the path for us to model the target device at the level of a higher order functional form, i.e. a function of control parameters that returns the expectation value. In general, we can write the predictive model $\check{f}$ of the expectation value $\langle \hat{O} \rangle$ given control parameter $\mathbf{\Theta}$ as

$$\langle \hat{O} \rangle \approx \check{f}\left(\mathbf{\Theta}, \rho_0, \hat{O}\right). \tag{43}$$

The next step can be to either directly implement the predictive model as a full Blackbox such as by using a Deep Neural Network or imposing prior knowledge about the system.

3.1.1. Characterization using Statistical Model

Training a predictive model requires an input x and an output y, also referred to as feature and label in ML literature, respectively. Each pair of input and output is called a sample. A dataset is formed of N samples. The input may be a vector of real numbers. In the case of complex input, the real and imaginary part can be separated and stored as real numbers. The sample can have multiple outputs too. This is common, for example in the image generation task.

In the context of modeling a quantum system, we may model the input as the control parameters $\mathbf{\Theta}$, where it can be a vector, i.e., multiple input features. The output label may be modeled as the expectation value calculated from the measurement readout of the quantum device. Since measuring a quantum device only yields a binary outcome as a result each time, multiple executions of the same input have to be carried out to obtain an ensemble of binary outcomes. Those binary values are then used to calculate the expectation value. Formally, each of the n binary outcomes $b \in \{0, 1\}^n$ is mapped to its corresponding eigenvalue values $e \in \{1, -1\}^n$ and the expectation value of the observable $\hat{O}$ for a given initial state ρ_0 is calculated by

$$\mathbb{E}[\hat{O}]_{\rho_0} = \frac{1}{n} \sum_{i}^{n} e_i. \tag{44}$$

So far, I have not specified the size of the quantum system. Note that the observable and initial state are of arbitrary size. Here the size refers to the dimension of the Hilbert space, not to be confused with sample size. For the sake of clarity, I will refer to the size of the Hilbert space in term of the number of qubits, otherwise it will be specified explicitly. The number of qubits of the initial state and observable depends on the characterization method and the device. For example, to characterize a single qubit system, the initial state and observable result in a 2×2 matrix. For a two-qubit system, we are dealing with a $4 \times$

4 matrix. The size of the Hilbert space grows exponentially with the number of qubits. However, the characterization and calibration rely on classical computers, whose memory cannot keep up with the quantum system. Thus, an efficient characterization method is necessary. Capturing the interplay between qubit interactions and system-environment nteractions without having to computer unrealisticaly large matrices is essential.

Each sample in the dataset defined above provides complete information about the system for the channel controlled by the given control parameters. In other words, a quantum channel can be reconstructed using each sample, given that the initial state and observable pairs are chosen sufficiently, e.g. process tomography. It is not necessary for every characterization method to collect aforementioned dataset. For example, a predictive model that has explicit mathematical form requires specialized experiments to estimate system parameters, for example a model used in a clear-box approach proposed in [137]. The explicit model characterization approach has an advantage of being able to characterize with less data consumption, but the flexibility is limited.

There are varieties of regression models that can be used. The go-to choice for the task when beginning with less prior knowledge is the family of DNN models. Intuition and nature of data then can be acquired by researchers to inforce a more informed model. Particularly in our case, quantum mechanics is known. The dynamics of a system can be solved explicitly given that the Hamiltonian of the system is fully known. The differential equation can be solved by using numerical solver of estimated by using other technique such as Physics-Informed Neural Network (PINN). One can also model the system with known part and parametrize the unknown part. One example, Graybox characterization method [52], [53] will be the primary case study to be discussed in Section 3.1.2.

In the case that the quantity of interest is derived expectation value of an observable, we can define a model using a model that predicts expectation value. For example, fidelity can be calculated by expectation value, whether it is state fidelity, process fidelity, or average gate fidelity. In the case of a model that predicts a mathematical object prior to expectation value of observable such as quantum state, or unitary, we may define a model using the model predicts the expectation value. The central pivotal role of expectation value is because it is the quantity that is estimated directly from the binary measurement results. While a binary outcome is closer to the low level implementation, it is a random variable by nature. However, modeling a deterministic outcome would be more straightforward. Thus, expectation value as an output label is reasonable choice.

In the statistical machine learning paradigm, there are several methods to fit the predictive model. However, the most prominent one is Empirical Risk Minimization (ERM), that is used to find the model parameters θ^* that minimize a loss function l, averaging over the dataset of size N. Formally, the loss function is defined as follows,

$$\mathcal{L}(\theta) = \sum_{i}^{N} \ell(x_i, y_i; \theta), \tag{45}$$

and we want to optimize for the model parameters by solving a minimization problem as follows,

$$\theta^* = \arg\min_{\theta \in \Theta} \mathcal{L}(\theta). \tag{46}$$

The word **empirical** indicates that this is a value calculated from the experimental dataset.

The most generic loss function for the regression model training is perhaps a mean squared error (MSE) of the output label function. In the case of the expectation value of observable $\mathbb{E}\left[\hat{O}\right]_{\rho_0}$, the loss function $\ell\left(\boldsymbol{\Theta}, \langle\hat{O}\rangle_{\rho_0}; \theta\right)$ can be defined as

$$\ell\left(\boldsymbol{\Theta}, \langle\hat{O}\rangle_{\rho_0}; \theta\right) = \left(\mathbb{E}[\hat{O}]_{\rho_0} - \check{f}\left(\boldsymbol{\Theta}, \rho_0, \hat{O}; \theta\right)\right)^2. \tag{47}$$

An MSE-based loss function is a function that outputs the value in the unit of square of the output label. The loss function is the formed from the average taken over the experimental settings of initial state ρ_0 and observable $\hat{O}$ that provide sufficient information for system characterization,

$$\mathcal{L}_{\text{SE[E]}} = \frac{1}{K} \sum_{\hat{O}, \rho_0} \left(\mathbb{E}[\hat{O}]_{\rho_0} - \check{f}\left(\boldsymbol{\Theta}, \rho_0, \hat{O}\right)\right)^2, \tag{48}$$

where K is the number of initial states and observables combinations. While another loss function is the absolute error based function of the form,

$$\ell\left(\boldsymbol{\Theta}, \mathbb{E}[\hat{O}]_{\rho_0}; \theta\right) = \left|\mathbb{E}[\hat{O}]_{\rho_0} - \check{f}\left(\boldsymbol{\Theta}, \rho_0, \hat{O}\right)\right|. \tag{49}$$

The AE-based loss function outputs the value of the same unit as the label.

In fact, the choice of loss function is rather arbitrary. It depends on the use case of the predictive model. For example, if only the fidelity is the quantity of interest, one may define loss function in term of fidelity directly.

Nonetheless, choosing on MSE-based loss function does has advantages for analysis. I will discuss the benefits in Section 3.1.3. After loss function selection, the model can be fitted using any suitable optimization algorithm. Implementation details will be discussed in Chapter 5.

After the model training, assessing its performance is the next step. This also includes finding the model that performs best. Different initial model parameters typically yield different models. Furthermore, the complexity of the model can also affect the perfor-

mance as well as speed and resources needed for training. For Deep Neural Network model, regardless of model architecture, the number of trainable parameters contributes to the resources needed for model training. Also, during the prediction phase, a large model would require more time to evaluate. Usually, the problem of selecting the most suitable model that performs well at a reasonable size is tackled by **hyperparameter tuning**, a.k.a. **model selection**. This is the blackbox optimization over the space of possible models that can predict a given dataset. More detail will be discussed in Chapter 5, but the discussion here is a preview of Section 3.1.3.

3.1.2. Graybox Characterization Method

In this section, I will discuss the main characterization method that will be used throughout this dissertation, the Graybox characterization method. First, the motivations behind the choice of the approach will be listed. Second, the mathematical formulation of the method will be reviewed. The architecture of model that can be used with the method will be discussed.

The Graybox characterization method is chosen for several reasons. (1) The method allows for a partially implicit model of the system from its mathematical formulation. The ideal Hamiltonian must be explicit. However, the noise can be implicit and represented by a regression model instead. This is a very powerful method as the explicit noise model can be hard to identify in an experimental setting. (2) The control function can be chosen. The freedom of control allows for model feature engineering, applying specialized control methodologies, and exploiting the prior knowledge of the platform. Moreover, only the control related to the application of interest can be selected, reducing the control space to be explored. For example, for Quantum Noise Spectroscopy (QNS), CPMG pulse is the logical choice. (3) It is a great example of a characterization method that combines both explicit and implicit solutions. The hybridization shows the potential and flexibility of the studies done in this dissertation for other approaches that may be fully mathematical implicit or explicit.

Before proceeding to the mathematical formulation of the Graybox method, I would like to remark on the fundamental concept of the Graybox first.

*Graybox model consist of a known explicit part, namely **Whitebox**, and an unknown implicit part, namely **Blackbox**. The former can be calculated by using knowledge from physics, where the later can be learned by statistical learning methods such as a Deep Neural Network.*

Now we are going to review the foundation that we will build upon [52]. Note that, however, this is not the only way to model a Graybox characterization method. Let us first

consider a total Hamiltonian $\hat{H}_{\text{total}}$ of the system that is the sum of an Ideal Hamiltonian $\hat{H}_{\text{ideal}}$ and a noise Hamiltonian $\hat{H}_{\text{noise}}$ of the form,

$$\hat{H}_{\text{total}}(\boldsymbol{\Theta}, t) = \hat{H}_{\text{ideal}}(\boldsymbol{\Theta}, t) + \hat{H}_{\text{noise}}(\boldsymbol{\Theta}, t) \tag{50}$$

The time and control dependent form is not a requirement, as it can be time and control independent. I keep the time-dependent form for its generality, and drop the explicit dependent on control for the sake of cleanness. The propagator of the total Hamiltonian is a time-ordered propagator obtained by solving Schrödinger equation $\hat{U}(t) = \mathcal{T}_+ \exp\Big\{-i \int_0^t \hat{H}_{\text{total}}(s) ds\Big\}$

The first step of the Graybox approach is to transform the total Hamiltonian to the interaction picture by using the time-ordered propagator $\hat{U}_{\text{ideal}}(t) = \mathcal{T}_+ \exp\Big\{-i \int_0^t \hat{H}_{\text{ideal}}(s) ds\Big\}$. The total Hamiltonian in the new frame becomes

$$\hat{H}_I(t) = \hat{U}^\dagger_{\text{ideal}}(t) \hat{H}_{\text{noise}} \hat{U}_{\text{ideal}}(t). \tag{51}$$

Similarly, the time-ordered propagator is $\hat{U}_I(t)$. The total propagator in the Schrödinger equation is $\hat{U}(T) = \hat{U}_{\text{ideal}}(T)\hat{U}_I(T)$. The Graybox method rearranges this expression via the ideal Propagator acting on the interaction picture Propagator,

$$\hat{U}_{I,R}(T) = \mathcal{T}_+ \exp\left\{-i \int_0^T \hat{U}_{\text{ideal}}(T) \hat{H}_{I(s)} \hat{U}^\dagger_{\text{ideal}}(T) ds\right\}. \tag{52}$$

Thus, the total propagator becomes

$$\hat{U}(T) = \hat{U}_{I,R}(T) \hat{U}_{\text{ideal}}(T). \tag{53}$$

The expectation value becomes

$$\langle \hat{O} \rangle_{\rho_0} = \text{Tr}\Big[\hat{U}_{I,R}(T) \hat{U}_{\text{ideal}}(T) \rho_0 \hat{U}^\dagger_{\text{ideal}}(T) \hat{U}^\dagger_{I,R}(T) \hat{O}\Big], \tag{54}$$

using the property of trace, we can define an operator containing the noise as

$$\hat{V}_O(T) = \langle \hat{O}^{-1} \hat{U}^\dagger_{I,R}(T) \hat{O} \hat{U}_{I,R}(T) \rangle_c, \tag{55}$$

resulting in the expectation value of observable of the form

$$\langle \hat{O} \rangle_{\rho_0} = \text{Tr}\Big[\hat{V}_O(T) \hat{U}_{\text{ideal}}(T) \rho_0 \hat{U}^\dagger_{\text{ideal}}(T) \hat{O}\Big]. \tag{56}$$

Observe that the expectation value above isolates the noise operator from the ideal operator. We can further simplify the expression by defining a noisy observable $\widehat{W}_O$ by noticing that the Pauli observable is constant over the classical expectation value,

$$\widehat{W}_O(T) = \langle \hat{U}^\dagger_{I,R}(T) \hat{O} \hat{U}_{I,R}(T) \rangle. \tag{57}$$

The noisy observable is a Hermitian matrix that is dependent on both ideal and noisy operators. With the noisy observable, the expectation value of the observable can be written as

$$\langle \hat{O} \rangle_{\rho_0} = \mathrm{Tr}\left[\widehat{W}_O(T)\hat{U}_{\text{ideal}}(T)\rho_0\hat{U}^{\dagger}_{\text{ideal}}(T)\right]. \tag{58}$$

The Graybox model is then defined over the noise and ideal operators. The ideal evolution which is explicitly known is a **Whitebox**, while, the implicit unknown part is a **Blackbox**.

As discussed in Section 2.1.3, the closed-form solution to the Schrödinger equation is not generally solvable via analytical methods, except for simple cases. Thus, numerical methods are needed for the calculation of the solution. The implementation of the ODE solver will be discussed later in Chapter 5. The main ways to obtain a solution include numerical ODE solver, Trotterization, or Physics Informed Neural Network (PINN). Regardless of the numerical method used, accuracy of the Whitebox is also important as the accuracy of Whitebox plays affect the accuracy of the Graybox predictive model.

Formally, **Whitebox** is the component in the **Graybox** that has the explicit mathematical procedure. For example, the procedure of calculating the propagator and then transform it to measurable quantity can also be considered **Whitebox** [53], [138].

The **Blackbox** component is the implicit part that estimates the unknown part. The implicit functional form to be estimated must be identified first to be able to chose the appropriate statistical learning method. The function should be part of the overall functional form of the Graybox. From the mathematical formulation above, the Blackbox is the noisy observable as a function of control parameters $\boldsymbol{\Theta}$,

$$\widehat{W}_O(\boldsymbol{\Theta}; T) = \langle \hat{U}^{\dagger}_{I,R}(\boldsymbol{\Theta}; T)\hat{O}\hat{U}_{I,R}(\boldsymbol{\Theta}; T)\rangle \tag{59.1}$$

$$= \tilde{\mathscr{O}}(\boldsymbol{\Theta}). \tag{59.2}$$

Here, the dependence on control parameters comes from the fact that $\hat{U}_{I,R}(\boldsymbol{\Theta}; T)$ depends on $\hat{U}_{\text{ideal}}(\boldsymbol{\Theta})$ which is a function of ideal Hamiltonian. Thus, the output of the Blackbox is a Hermitian matrix of the same size as the Hilbert space of the system, e.g. a 2×2 matrix for a single qubit system. If the statistical learning method of choice is Neural Network, one can define the output layer of the model to have $2k$ neurons, where k is the number of elements of the Hermitian matrix. The factor of 2 is for the real and imaginary components of each matrix entry. However, one can also model the output layer to be parameters of a parametrized Hermitian matrix, the number of output neurons can be decreased in this manner.

In this dissertation, the choice of Deep Neural Network as Blackbox is the main focus as it can be easily promoted to a BNN. The details will be discussed in Section 3.2. In

previous works, the architecture of Blackbox is identified based on the prior knowledge of the problem. For instance, if the input is a sequence-like object, a Recurrent Neural Network (RNN) is used, for its capability to deal with time series information. One example is the control that is applied in sequence, where the accumulated consequence is expected to be captured by the RNN architecture [52], [138]. In Ref. [52], they model a simulated device by using the LSTM (one variation of RNN) as the Blackbox to map control parameters to the noise operator defined in Equation 55. Then the Whitebox maps the results and the system information to the expectation values. In Ref. [138], the authors model the reconfigurable photonic circuit by using GRU (one variation of RNN architecture) as a Blackbox to model the map between time series of control voltage to the interaction Hamiltonian. Then the Whitebox transform the interaction Hamiltonian to power loss during the measurement. In the case of simpler control, A. Youssry *et al.* model the quantum photonic circuit by using a simple Multi-Layer Perceptron (MLP) as the Blackbox to map the control to the static Hamiltonian. Then Whitebox transforms the Hamiltonian to the probability of observing measurement results.

There are multiple possible choices of Blackbox and Whitebox depending on how we model the physical system. Each choice have its own advantages. For example, Blackbox can model the operator defined in Equation 59 as well instead of Equation 55. Modeling Equation 59 results in a simpler computation procedure than Equation 55. Another example is to model Equation 52, which is even more simpler. Moreoever, modeling the Equation 52 has the additional advantage that we can simultanously characterize SPAM noise. Furthermore, to completely eliminate the heavy calculation of solving the Schrödinger equaiton, we can also model Equation 53 which is the total unitray directly.

Combining both boxes, the Graybox model is successfully constructed. The Whitebox does not need the training phase as it can calculate the output from input directly from the mathematical procedure. In contrast, the Blackbox needed the training phase. The performance or the accuracy of the Graybox model can be evaluated by testing on the unseen experimental data. In ML practice, dataset would be split to training, validating, and testing subsets. Usually, the testing dataset is used for model to evaluate the test loss after the model selection and represent the expected accuracy of the model against the actual evaluation time.

3.1.3. What we can learn from SML approach

In this section, we include parts of the analysis found in our prior work [139][1]. The loss function basically told us how well this model did what we trained it to do. For the loss function averaging over the dataset based on MSE of the expectation value of observables

[1]We acknowledge the use of IBM Quantum services for this work. The views expressed are those of the authors, and do not reflect the official policy or position of IBM or the IBM Quantum team.

$\mathcal{L}_{\text{SE[E]}}$, the model is trained to predict the expectation value of the observable as closely as possible to the actual value produced from the real device. However, the value of $\mathcal{L}_{\text{SE[E]}}$ after the model training does not have a self-contained meaningful physical interpretation. For example, in the image classification task, the accuracy can be easily interpreted by counting the correct predictions and dividing the number by the total number of elements in the test dataset, the model will have an associated performance quantity, e.g. 73.4%. We know that the lower $\mathcal{L}_{\text{SE[E]}}$, the better, but at what particular value can the model be used in practical usage?

One of the practical applications of predictive models is to be used in control calibration by the open-loop approach. The usage of a predictive model to produce high performance physical gates is important, because quantum error correction protocols require physical gates to perform with error rate below some noise threshold, otherwise the logical error rate cannot be improved. Thus, the predictive model is expected to predict at high accuracy and precision. Thus, the prediction error in term of gate performance from actual experiment serves as a better performance metric.

One can define a loss function that is based on fidelity, which can be interpreted directly as a closeness of prediction and experiment as a scale of fidelity from zero to one. For instance, the expectation values can be used to calculate AGF. Thus, the loss function defined based on the AGF predicted by the model $\bar{F}_{\text{model}}$ and the experimental AGF $\bar{F}_{\text{exp}}$ with respect to $\hat{U}_{\text{ideal}}(\boldsymbol{\Theta})$ can be defined as,

$$\mathcal{L}_{\text{AE[F]}} = \left|\bar{F}_{\text{model}} - \bar{F}_{\text{exp}}\right|. \tag{60}$$

This loss function allows us to interpret the model as it trained to predict the expectation values that produces the AGF that matches the value calculated from expectation values from experiment.

3.1.3.1. Decomposition of MSE

One of the challenges of training machine learning is determining when to stop model selection, i.e., determining if this model with this model parameters is good enough for practical application. In the following, we will analyze the model trainability. Let us consider quantum experiments composed of qubits where only binary measurement outcomes are observed, e.g., $+1$ and -1 for qubit measurements. The expectation value given an initial state and a set of control actions, $\langle \hat{O} \rangle_{\boldsymbol{\Theta}}$, can be estimated by taking an average of the measurement outcomes over n executions of the same circuit. We refer to n as the number of shots or simply shots and denote the n-shot estimate of the expectation value by $\mathbb{E}_n \left[\hat{O}\right]_{\boldsymbol{\Theta}}$. The finite number of shots is one of the sources of uncertainty in the experimental setting, causing the model in Equation 43 to be inaccurate. Typically, one

would need to minimize a loss function as low as possible; however, with the finite-shot training data, there might be limits on the accuracy of the trained model and its predictive performance. In this section, we will answer the question of what one can expect in model training given a finite-shot dataset.

In order to search for an optimal predictive model for Equation 43, we consider a standard choice for the loss function in machine learning for regression tasks, which is the square error (SE). The model $\check{f}(\boldsymbol{\Theta}) = \langle \hat{O} \rangle^{\text{model}}_{\rho_0,\boldsymbol{\Theta}}$ (noting that we have dropped the dependency on the initial state ρ_0 and observable $\hat{O}$ for notational simplicity) should predict the expectation value as close as possible to the exact (infinite-shot) expectation value $\langle \hat{O} \rangle^{\text{exp}}_{\rho_0,\boldsymbol{\Theta}}$ by learning from the experimentally obtained (finite-shot) expectation value, $\mathbb{E}_n \left[\hat{O}\right]^{\text{exp}}_{\rho_0,\boldsymbol{\Theta}}$, for a given control $\boldsymbol{\Theta}$. For the SE loss function, we write the loss as a squared difference between the two quantities,

$$\mathcal{L}_{\text{SE[E]}} \equiv \frac{1}{M} \sum_{k,\rho_0} \left(\langle \hat{O} \rangle_{\text{model}_{\rho_0,\boldsymbol{\Theta}}} - \mathbb{E}_n \left[\hat{O}_k\right]^{\text{exp}}_{\rho_0,\boldsymbol{\Theta}} \right)^2 \tag{61}$$

where the purpose of the sum is to average over a set of M different combinations of observables $\hat{O}_k$ as well as initial states ρ_0. We use the subscript SE[E] to indicate that the loss is an average squared error of the expectation values of the quantum observable, where the mean here refers to the average over the combinations of initial states and observables. To approximate the lowest possible value of $\mathcal{L}_{\text{SE[E]}}$ given the finite-shot data, we use the decomposition of the square error [140], applying to an example of the predictive model similar to $\check{f}(\boldsymbol{\Theta})$ in Equation 43. Let us consider the input X as a random variable sampled from a probability distribution $X \sim p_{X(x)}$, and the output Y as a random variable sampled from a probability distribution $Y \sim p_{Y|X}(y|x)$. The predictor $\check{f}(x) = \check{f}(x;\mathcal{D})$ (not necessary) trained on the fixed size dataset $\mathcal{D} = \{(X,Y)\}$ and kept fixed should minimize a mean square error given by

$$\mathbb{E}_{x,y}\left[\left(y - \check{f}(x)\right)^2\right] = \mathbb{E}_x\left[\left(\check{f}(x) - \mathbb{E}[y|x]\right)^2\right] + \mathbb{E}_x[\text{Var}(y|x)]. \tag{62}$$

The expectations in Equation 62 are with respect to all possible values of input and output. Note that both terms are positive. Equation 62 is minimized when the predictor is optimal, in other words it predicts the expected value of the output for a given input $\check{f}(x) = \mathbb{E}[y|x]$, reducing the first to zero. However, the last term, the expectation of conditional variance of the distribution of Y given the input value of $X = x$ over input space x, is considered an irreducible error as it is inherited in the observed data Y. Thus, the expected conditional variance is the lower bound of the square error, $\mathbb{E}_{x,y}\left[\left(y - \check{f}(x)\right)^2\right]$, achievable by the optimal predictor.

In our context of Equation 43, our model $\check{f}(\boldsymbol{\Theta})$ should minimize $\mathcal{L}_{\mathrm{SE[E]}}$ in Equation 61 where the observed data is the finite-shot expectation value, i.e., $Y = \mathbb{E}_n\left[\hat{O}\right]^{\mathrm{exp}}_{\rho_0,\check{\boldsymbol{\Theta}}}$. Therefore, we can apply the decomposition in Equation 62, replacing x with $\boldsymbol{\Theta}$, $\check{f}(x)$ with $\langle\hat{O}\rangle^{\mathrm{model}}_{\rho_0,\boldsymbol{\Theta}}$, taking the average over all possible initial states and observables. We find that the conditional variance in Equation 62 is given by

$$\mathrm{Var}(y|x) \mapsto \mathrm{Var}\left(\mathbb{E}_n\left[\hat{O}\right]^{\mathrm{exp}}_{\rho_0,\boldsymbol{\Theta}} \mid \boldsymbol{\Theta}\right). \tag{63}$$

The expectation over the control variable should give us an approximated lower bound for the loss in Equation 61. The conditional variance $\mathrm{Var}(y|x)$ could be extracted directly from the experimental data. Given particular control parameters, observables, and initial states, a large number of experiments need to be executed so that one can approximate the exact conditional variance. Then, the sample variance should be averaged over all combinations of initial states and observables. However, this method is costly as it requires additional experiments just to estimate the conditional variance. We instead derive an analytical expression of the conditional variance using the variance of a quantum observable based on Bernoulli distribution. The variance of an estimation of an observable $\hat{O}$ from n measurement shots is

$$\mathrm{Var}\left(\mathbb{E}_n\left[\hat{O}\right]^{\mathrm{exp}}_{\rho_0,\boldsymbol{\Theta}} \mid \boldsymbol{\Theta}\right) = \frac{1}{n}\left(1 - \langle\hat{O}\rangle^2_{\rho_0,\boldsymbol{\Theta}}\right) \tag{64}$$

given the control variables $\boldsymbol{\Theta}$ and the initial state ρ_0 fixed when evaluating the expectation value.

We then calculate the bound for our single qubit case, where we consider a set of observables $\hat{O} \in \left\{\hat{X}, \hat{Y}, \hat{Z}\right\}$ which fully characterize a quantum state and a set of pure initial states $\rho_0 \in \{|+\rangle\langle+|, |-\rangle\langle-|, |i\rangle\langle i|, |-i\rangle\langle -i|, |0\rangle\langle 0|, |1\rangle\langle 1|\}$. Then, assuming that the control $\boldsymbol{\Theta}$ only produces unitary operations and all initial states are pure states, we can apply $\langle X\rangle^2_{\rho_0,\boldsymbol{\Theta}} + \langle Y\rangle^2_{\rho_0,\boldsymbol{\Theta}} + \langle Z\rangle^2_{\rho_0,\boldsymbol{\Theta}} = 1$ and obtain the lower bound of an expected loss of Equation 61 as

$$\mathbb{E}_{x,y}\left[\mathcal{L}_{\mathrm{SE[E]}}\right] \geqslant \mathbb{E}_x\left[\frac{1}{18n}\sum_{\rho_0}\sum_{k}\left(1 - \langle\hat{O}_k\rangle^2_{\rho_0,\boldsymbol{\Theta}}\right)\right], \tag{65.1}$$

$$= \mathbb{E}_x\left[\frac{1}{18n}\sum_{\rho_0} 2\right] = \frac{2}{3n}. \tag{65.2}$$

We refer to the average of the conditional variances (irreducible errors), which is independent of the choice of control action, as *data variance*. We note that the result does not depend on the number of initial states ρ_0 taken in the averaging process.

We now discuss how to use the data variance in practice. Since the expected loss on the LHS of Equation 62 is the expectation with respect to joint distribution $p_{X,Y}(x,y) = p_{Y|X}(y|x)p_{X(x)}$, we can estimate it by averaging the squared error over samples from the distribution $p_{X,Y}(x,y)$. Since the x, y samples will form a dataset, we denote $\mathbb{E}_{\{X,Y\}}[\cdot] \to \mathbb{E}_{\text{dataset}}[\cdot]$. In the subsequent calculations, the samples can be taken from the training or testing dataset. Then we can calculate the LHS of Equation 62 by averaging the squared error over the dataset using the predictive model. Consequently, the optimality of the predictor, i.e., first term on the RHS of Equation 62, can be calculated from the empirical value of expected loss, i.e. LHS of Equation 62, subtracted by the data variance. Thus, the data variance, which is an analytical expression of the irreducible error, would provide additional information, complementing the empirical result in the analysis of the predictive model. For instance, we could design an optimization stopping criterion in the predictive model construction process based on the empirical optimality of the predictor.

With Equation 65, we know that the expected loss should be at most reduced to $\frac{2}{3n}$, which also decreases as the number of shots n increases. In practice, we construct a predictive model by minimizing Equation 62 over the training dataset $\mathbb{E}_{\text{train}}\left[\mathcal{L}_{\text{SE[E]}}\right]$. However, care must be taken when selecting a model based on expected training loss. For example, in the case of using a neural network as a part of a predictive model, it may be possible that the model is large enough to overfit the training dataset, but the expected loss approaching $\frac{2}{3n}$ results in selecting a non-optimal model. Thus, it is considered good practice to evaluate the expected loss using (unseen) testing dataset $\mathbb{E}_{\text{test}}\left[\mathcal{L}_{\text{SE[E]}}\right]$. Therefore, we expect that selecting a model that predicts the expected loss over the training and testing datasets simultaneously, which also approximately approaches the data variance, is a first guiding step in model selection.

3.1.3.2. Performance bound

Let us start with the definition of AGF between an ideal unitary $\hat{U}$ parametrized by control variable $\boldsymbol{\Theta}$ (not shown explicitly) and a map $\mathcal{E}$. Let $\rho = |\psi\rangle\langle\psi|$ be an arbitrary pure state, an average of state fidelity between $\hat{U}^\dagger\ |\psi\rangle\langle\psi|\ \hat{U}$ and $\mathcal{E}(|\psi\rangle\langle\psi|)$ integrating over all possible $|\psi\rangle$, which is

$$\bar{F}\left(\mathcal{E}, \hat{U}\right) = \int d\psi \langle\psi \mid \hat{U}^\dagger \mathcal{E}(|\psi\rangle\langle\psi|)\hat{U}\ |\psi\rangle, \tag{66}$$

as defnined in Equation 37. Typically, we are interested in the case where the map $\mathcal{E}$ is an experimental realization of $\hat{U}$. For the single qubit case, the system dimension is $d = 2$, and the AGF can be written [86] with a state decomposition on the Pauli matrices as

$$\bar{F}\left(\mathcal{E},\hat{U}\right) = \frac{1}{2} + \frac{1}{12}\sum_{j=1}^{3}\sum_{k=0}^{1}\alpha_{j,k}\,\mathrm{Tr}\left[\hat{U}P_j^{\dagger}\hat{U}^{\dagger}\mathcal{E}\left(\rho_{j,k}\right)\right] \tag{67.1}$$

$$= \frac{1}{2} + \frac{1}{12}\sum_{j,m=1}^{3}\sum_{k=0}^{1}\alpha_{j,k}\beta_{m,j}\,\mathrm{Tr}\left[\hat{P}_m\mathcal{E}\left(\rho_{j,k}\right)\right] \tag{67.2}$$

$$= \frac{1}{2} + \frac{1}{12}\sum_{j,m=1}^{3}\sum_{k=0}^{1}\alpha_{j,k}\beta_{m,j}\langle\hat{O}\rangle_{j,k,m} \tag{67.3}$$

where indices j,m indicate different Pauli observables, i.e., $\hat{P}_j,\hat{P}_m \in \left\{\hat{X},\hat{Y},\hat{Z}\right\}$, and $\rho_{j,k} \in \{|+\rangle\langle+|,|-\rangle\langle-|,|i\rangle\langle i|,|-i\rangle\langle-i|,|0\rangle\langle 0|,|1\rangle\langle 1|\}$. In this notation, j indicates the Pauli observable and k indicates whether the state is a +1 or a −1 eigenstate of that observable. For example, $\rho_{2,1} = |-i\rangle\langle-i|$. For any Pauli observable $\hat{P}_j$, we have $\alpha_{j,0} = 1$ and $\alpha_{j,1} = -1$. From Equation 67.1, we substitute $\hat{U}P_j^{\dagger}\hat{U}^{\dagger}$ with its Pauli basis expansion, resulting in Equation 67.2 where $\beta_{m,j} = \frac{1}{2}\,\mathrm{Tr}\left[\hat{U}\hat{P}_j^{\dagger}\hat{U}^{\dagger}\hat{P}_m\right]$. This expansion allows us to estimate the AGF between any ideal (qubit) unitary map and its experimental realization by measuring the expectation values of at most 18 combinations of 6 initial states and 3 Pauli observables. Notice that $\mathrm{Tr}\left[\hat{P}_m\mathcal{E}\left(\rho_{j,k}\right)\right]$ is the exact experimental expectation value, so we denote it as $\langle\hat{O}\rangle_{j,k,m}$ in Equation 67.3.

Since the predictive model $\check{f}(\boldsymbol{\Theta})$ can predict the expectation values of a given control parameter, one can calculate the AGF from the prediction. The quantity of interest is the accuracy of AGF prediction, $\bar{F}_{\text{model}}$, to the exact value of the experimental realization, $\bar{F}_{\text{exact}}$. Here, we note that $\bar{F}_{\text{exact}}$ is not a finite-shot estimation of AGF measurable in experimental realization, but is the infinite-shot (exact) value. The difference between them is obtained using Equation 67.2,

$$\bar{F}_{\text{model}} - \bar{F}_{\text{exact}} = \frac{1}{12}\sum_{j,m=1}^{3}\sum_{k=0}^{1}\alpha_{j,k}\beta_{m,j}\left(\langle\hat{O}\rangle_{j,k,m}^{\text{model}} - \langle\hat{O}\rangle_{j,k,m}^{\text{exact}}\right). \tag{68}$$

We now show that the AGF difference in Equation 68 and the MSE in Equation 61 are related through the Cauchy-Schwarz inequality,

$$\left(\sum_i u_i v_i\right)^2 \leqslant \left(\sum_i u_i^2\right)\left(\sum_i v_i^2\right). \tag{69}$$

Let us start by defining functions

$$u_i = \langle\hat{O}\rangle_i^{\text{model}} - \langle\hat{O}\rangle_i^{\text{exact}} \equiv \langle\hat{O}\rangle_{j,k,m}^{\text{model}} - \langle\hat{O}\rangle_{j,k,m}^{\text{exact}}, \tag{70}$$

$$v_i \equiv \alpha_{j,k}\beta_{m,j}, \tag{71}$$

where the indices j, k, m are combined into the single index i, which leads to writing Equation 68 as

$$\bar{F}_{\text{model}} - \bar{F}_{\text{exact}} = \frac{1}{12}\sum_i v_i u_i. \tag{72}$$

Note that we use the Cauchy-Schwarz inequality for real numbers, so we have to show that u_i and v_i are real numbers. Since u_i is defined in terms of the expectation value of an observable, it must be a real number. For v_i, we already know that $\alpha_{j,k} \in \{-1, 1\}$, so we have to show that $\beta_{m,j}$ is a real number. We consider $\beta^*_{m,j}$ as follows,

$$\left(\frac{1}{2}\operatorname{Tr}\left[\hat{U}\hat{P}_j^\dagger\hat{U}^\dagger\hat{P}_m\right]\right)^* = \frac{1}{2}\operatorname{Tr}\left[\left(\hat{U}\hat{P}_j^\dagger\hat{U}^\dagger\hat{P}_m\right)^\dagger\right], \tag{73.1}$$

$$= \frac{1}{2}\operatorname{Tr}\left[\hat{U}\hat{P}_j^\dagger\hat{U}^\dagger\hat{P}_m\right], \tag{73.2}$$

$$\beta^*_{m,j} = \beta_{m,j}, \tag{73.3}$$

where we use the cyclic property of the trace, and use the fact that $\hat{P}_m = \hat{P}_m^\dagger$ and $\hat{P}_j = \hat{P}_j^\dagger$ in Equation 73.1. From Equation 73.3, we show that $\beta_{m,j}$ must be a real number. Therefore, v_i must be a real number. Now, we will express Equation 61 in terms of u_i. We denote the experimental data $\mathbb{E}_n\left[\hat{O}_k\right]^{\text{exp}}_{\rho_0,\boldsymbol{\Theta}}$ with $\mathbb{E}_i^{\text{exp}}$ for notation brevity,

$$\mathcal{L}_{\text{SE[E]}} = \frac{1}{18}\sum_i \left(\langle\hat{O}\rangle_i^{\text{model}} - \mathbb{E}_i^{\text{exp}}\right)^2. \tag{74}$$

We then express Equation 74 in terms of u_i as follows,

$$\mathcal{L}_{\text{SE[E]}} = \frac{1}{18}\sum_i \left(\langle\hat{O}\rangle_i^{\text{model}} - \langle\hat{O}\rangle_i^{\text{exact}} + \langle\hat{O}\rangle_i^{\text{exact}} - \mathbb{E}_i^{\text{exp}}\right)^2, \tag{75.1}$$

$$= \frac{1}{18}\sum_i \left(u_i - \varepsilon_i\right)^2, \tag{75.2}$$

$$= \frac{1}{18}\sum_i u_i^2 - \frac{1}{9}\sum_i u_i\varepsilon_i + \frac{1}{18}\sum_i \varepsilon_i^2. \tag{75.3}$$

In Equation 75.1, we inserted zero in the forms of the exact experimental expectation value $\langle\hat{O}\rangle_i^{\text{exact}}$. Then, in Equation 75.2, we defined $\varepsilon_i = \mathbb{E}_i^{\text{exp}} - \langle\hat{O}\rangle_i^{\text{exact}}$ and the remaining terms are u_i as we defined in Equation 70. Lastly, we expanded the expression in Equation 75.3. We then, rearrange for $\sum_i u_i^2$ as follows,

$$\sum_i u_i^2 = 18\mathcal{L}_{\text{SE[E]}} + \sum_i (2u_i\varepsilon_i - \varepsilon_i^2). \tag{76}$$

For the summation $\sum_i v_i^2$, we can look at the properties of $\alpha_{j,k}$ and $\beta_{m,j}$. Since we know that $\alpha_{j,k}$ has two possible values, $\alpha_{j,0} = 1$ and $\alpha_{j,1} = -1$, and that $\beta_{m,j}$ does not depend on the index k, we can write

$$\sum_i v_i^2 = \sum_{j,k,m} \left(\alpha_{j,k}\beta_{m,j}\right)^2 = 2\sum_{j,m} \left(\beta_{m,j}\right)^2. \tag{77}$$

This quantity can be simplified even further. Note that $\beta_{m,j} = \frac{1}{2}\operatorname{Tr}\left[\hat{U}\hat{P}_j^\dagger\hat{U}^\dagger\hat{P}_m\right]$, where $\hat{P}_j$ and $\hat{P}_m$ are Pauli matrices. We express $\hat{P}_j^\dagger = \hat{P}_j = \rho_{j,+} - \rho_{j,-}$ in terms of its $+1$ and -1 eigenvectors. When the initial states ρ_+ and ρ_- evolve under the same unitary, their expectation values when measuring the same Pauli observable will have the same magnitude but opposite sign, $\operatorname{Tr}\left[\hat{U}\rho_{j,+}\hat{U}^\dagger\hat{P}_m\right] = -\operatorname{Tr}\left[\hat{U}\rho_{j,-}\hat{U}^\dagger\hat{P}_m\right]$ We can therefore write

$$\beta_{m,j} = \operatorname{Tr}\left[\hat{U}\rho_{j,+}\hat{U}^\dagger\hat{P}_m\right] = \langle\hat{P}_m\rangle_{\rho_{j,+}}. \tag{78}$$

Note that the quantity we want is the summation $\sum_{j,m}\left(\beta_{m,j}\right)^2$, which, given Equation 78, we see that the summation with the index m leads to the exact form of purity of a state $\hat{U}\rho_{j,+}\hat{U}^\dagger$. Given that we know $\rho_{j,+}$ is a (pure) eigenstate, we are sure that the purity is unity. That is, the summation of index m will be 1. Since index j is the sum over initial states, this means that the summation with the index j gives a multiplicative factor of 3, leading to

$$\sum_i v_i^2 = 2\sum_{j,m}\left(\beta_{m,j}\right)^2 = 2\sum_{j,m}\langle\hat{P}_m\rangle^2_{\rho_{j,+}} = 6. \tag{79}$$

Substituting the results of u_i and v_i summation into Equation 69, we arrive at the following inequality,

$$\left(\bar{F}_{\text{model}} - \bar{F}_{\text{exact}}\right)^2 \leqslant \frac{6}{144}\left(18\mathcal{L}_{\text{SE[E]}} + \sum_i (2u_i\varepsilon_i - \varepsilon_i^2)\right). \tag{80}$$

To remove the random variable ε_i, we compute the expected value of the inequality. Notice that, u_i and v_i do not depend on ε, so $\bar{F}_{\text{model}} - \bar{F}_{\text{exact}}$ is constant with respect to $\mathbb{E}_{y|x}$. We use the fact that $\mathbb{E}_\varepsilon[\varepsilon_i] = 0$, and notice that $\mathbb{E}_{y|x}[\varepsilon_i^2] = \operatorname{Var}\left(\mathbb{E}_n\left[\hat{O}\right]^{\text{exp}}_{\rho_0,\boldsymbol{\Theta}} \mid \boldsymbol{\Theta}\right)$ is the conditional variance with the result from Equation 65 to obtain the following bound,

$$\left(\bar{F}_{\text{model}} - \bar{F}_{\text{exact}}\right)^2 \leqslant \frac{3}{4}\left(\mathbb{E}_{y|x}\left[\mathcal{L}_{\text{SE[E]}}\right] - \frac{2}{3n}\right). \tag{81}$$

Furthermore, we can connect Equation 62 to the absolute difference between the AGF predicted by model and exact experimental value by considering an expected value of

inequality in Equation 81 with respect to x and using the Jensen inequality for a random variable $\left|\bar{F}_{\text{model}} - \bar{F}_{\text{exact}}\right|$ with respect to x. We then have

$$\mathbb{E}_x[\left|\bar{F}_{\text{model}} - \bar{F}_{\text{exact}}\right|] \leqslant \sqrt{\frac{3}{4}\left(\mathbb{E}_{x,y}\left[\mathcal{L}_{\text{SE[E]}}\right] - \frac{2}{3n}\right)}, \tag{82}$$

where the inequality becomes the bound of the expected difference of AGF over different inputs in terms of the expected loss. With the decomposition of $\mathcal{L}_{\text{SE[E]}}$, the upper bound of the AGF difference, Equation 82, will depend only on the optimality term $\Lambda = \mathbb{E}_x\left[\left(\check{f}(x) - \mathbb{E}[y|x]\right)^2\right]$, as the terms depending on n are canceled out. Thus, we obtain an alternative form of the bound in Equation 82

$$\mathbb{E}_x(\left|\bar{F}_{\text{model}} - \bar{F}_{\text{exact}}\right|) \leqslant \sqrt{\left(\frac{3}{4}\right)\Lambda}. \tag{83}$$

We plot the LHS of Equation 83 in terms of the optimality in Figure 3.1, showing the required optimality needed to achieve the desired expected difference of AGF. Note that we can estimate the optimality Λ using the testing dataset discussed in previous section. This result suggests that a high-precision quantum gate can be reliably produced by using a predictive model trained on a few-shot dataset. In the best case scenario, with an optimal predictive model $\mathbb{E}_{x,y}\left[\mathcal{L}_{\text{SE[E]}}\right] = \frac{2}{3n}$, the upper bound on the RHS becomes zero, so the model can predict the exact experimental AGF. We would like to highlight that the LHS of Equation 82 is the difference between the model prediction and the exact experimental value, and is not the difference between the model prediction and the finite-shot experimental value. These analytically derived bounds in Equation 65, and Equation 83 confirm the observed behavior in [141].

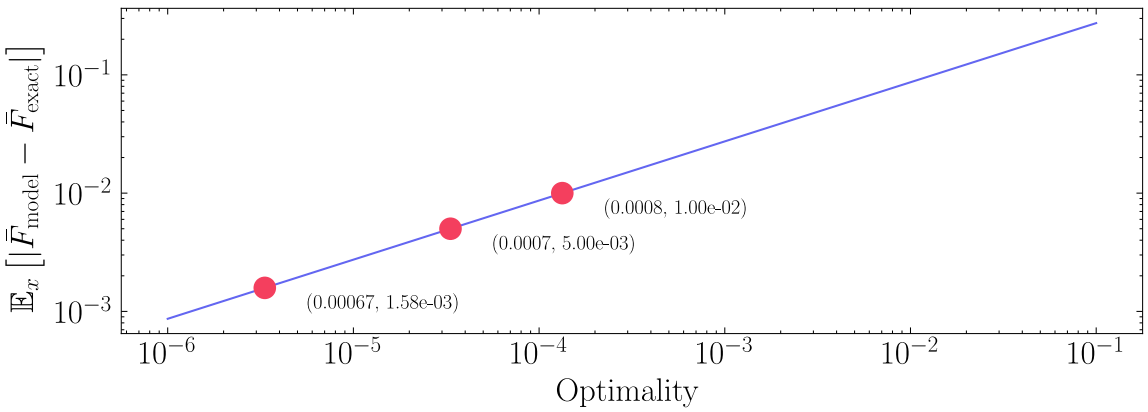


Figure 3.1: The figure shows a plot of the performance bound in terms of optimality. The x-coordinate of annotated tuples is the expected $\mathcal{L}_{\text{SE[E]}}$, while the y-coordinate is the corresponding performance bound. This figure is used in [139].

3.1.4. Numerical and Experimental Characterization using Statistical Graybox Model

In this section, we explain and discuss a device characterization and control calibration using the Graybox method. We start by discussing the experimental procedure, independent of quantum device specification in Section 3.1.4.1. We then use a numerical study to support our analytical results Equation 65 and Equation 82 in Section 3.1.4.2. Using Graybox method, we characterize a transmon qubit, and use the predictive model to calibrate the $\sqrt{X}$ gate. We focus on the $\sqrt{X}$ gate because a sequence of $\sqrt{X}$ and Virtual-Z gates can produce a universal gateset [142]. Next, we experimentally characterize a real superconducting quantum computer, calibrate control variables for the $\sqrt{X}$ gate, and benchmark the results compared to our model prediction in Section 3.1.4.3. This section is adapted from the results from our previous work [139].

3.1.4.1. Experimental procedure

Before proceeding to our numerical and experimental results, we outline our characterization procedure, which includes data acquisition, preprocessing, model training, and hyperparameter tuning. We show the high-level illustration of the experimental procedure in Figure 3.2.

S1 The designing configuration phase consists of four requirements to be identified before performing an experiment.

1. The specifications of qubit(s) in the quantum device are needed for composing the Whitebox, while the Blackbox will characterize the noise. Qudit extension is also possible. For example, for the superconducting platform, the energy splitting of the system can be specified in the Whitebox.
2. A functional form of control (or the pulse sequence in the superconducting qubit example) and bounds on the parameters are needed. As discussed previously, these specifications depend on the device of interest.
3. Number of samples, m, is the size of the dataset used to train the model. Each sample consists of the control variables (input or feature in ML literature) and 18 combinations of expectation values (output or label in ML literature) for a single qubit case. The combination originates from six initial states corresponding to eigenvectors of three Pauli observables $|0\rangle, |1\rangle, |+\rangle, |-\rangle, |+i\rangle, |-i\rangle$ and three Pauli observables.
4. Number of shots, n, that define model training and prediction accuracy limitations.

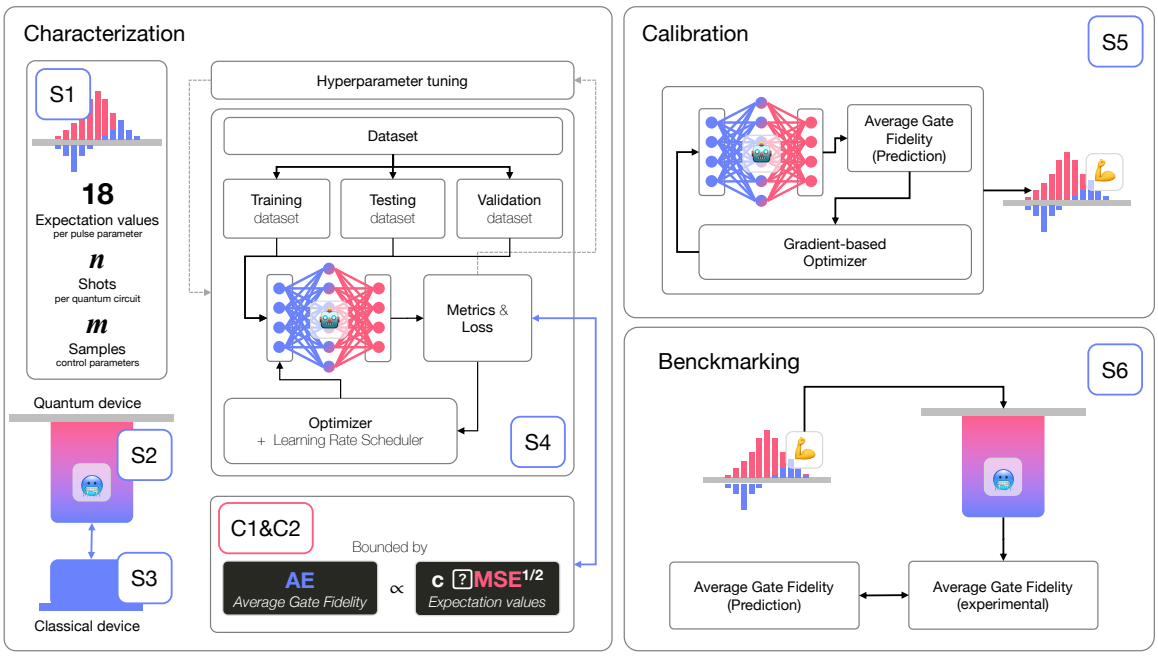


Figure 3.2: Overview of the characterization, calibration, and benchmark pipeline for a quantum device using the Graybox approach and our contribution to the approach with analysis tooling. (S1) The experiment configuration and dataset must be defined before the characterization. (S2) Data collected from the experiment or simulation. (S3) Data preprocessing, which includes pre-calculation of ideal evolutions given control variables. (S4) Model training is performed with the dataset, and hyperparameters are tuned. (C1&C2) We derive a bound on the expected MSE of expectation values achievable by the Graybox model, informing us when to stop model selection. We derive a bound on the expected absolute error of the average gate fidelity between the predicted and exact experimental values. (S5) The trained model is then used as a characterization of quantum device, i.e., it is used as a clone of the device on a local classical computer, where an open-loop control calibration is performed to acquire optimal control variables. (S6) Benchmarking of the optimized control variables is performed on the target device. This figure is used in [139].

Once the experiment configuration is defined, the next step is **S2** n a real device or a simulation with or without noise.

After the data collection is complete, **S3** preprocessing data step involves pre-calculating any quantities used repeatedly in the training step. In our case, it is the ideal unitary corresponding to the control variables. Although not required, it is a recommended step. In our case, we pre-calculate the ideal unitary to speed up the computation.

With the data and information gathered, we characterize the device into a form of predictive model in **S4** model selection} step. At this step, we are searching for the model

parameters and architecture, learning algorithm (the optimizer and its parameters), the suitable sample size, and other hyperparameters, which minimize the validation loss as much as possible. We implement the Whitebox by solving the Schrödinger equation for the ideal unitary operator given the control variables. The architecture of the Blackbox consists of three groups of Multi-Layer Perceptron (MLP) layers. The first group is the Shared layer (MLP of 2 layers), which is shared for all Pauli observables. After processing by the Shared layer, Pauli (MLP of two layers) and Hermitian (MLP of a single layer) groups will independently transform the result from the Shared layer and return parameters for Equation 59 of each Pauli observable. The outputs of the Whitebox and Blackbox parts are combined to produce final predictions of expectation values of all 18 combinations of Pauli observables and initial states. The details of the model architecture and how different parts of the algorithm are connected are shown in Figure 3.3.

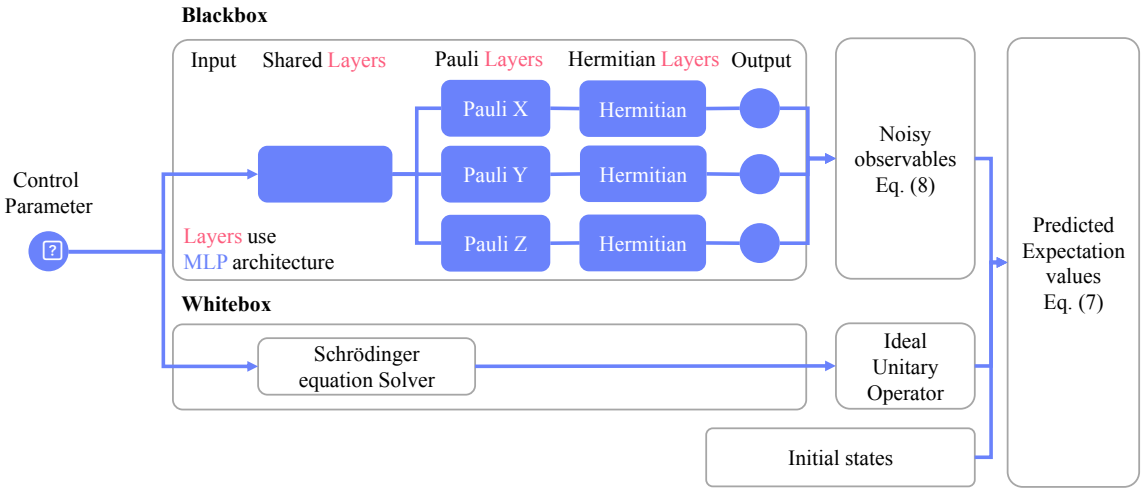


Figure 3.3: The architecture of the Graybox predictive model, consisting of the Whitebox and the Blackbox parts. The Blackbox model consists of three groups of layers: Shared, Pauli, and Hermitian layers. These layers all use the MLP architecture. The Shared layer transforms the vector of input parameters ($\boldsymbol{\Theta}$) into an intermediate vector. Each of the three Pauli layers then transforms the Shared layer's output independently. Finally, the Hermitian layer transforms the output of the Pauli layer into the parameters of the Hermitian matrix corresponding to Equation 59 for each Pauli observable. The Whitebox model is implemented by solving Schrödinger equations for ideal unitary evolution given the control parameters. Outputs of the Whitebox and Blackbox parts are combined to produce final predictions of expectation values for all 18 combinations of Pauli observables and initial states. This figure is used in [139].

After the model selection, we perform **S5** model calibration} to optimize the control variables to a target quantum gate. The optimization is performed by using a gradient-based optimization algorithm. We use the `adam` optimizer with the `warmup-cosine-decay-scheduler` learning rate schedule for the active period of 1,000 iterations. This schedule is practically the same as the schedule used in training the model but is set to be active for 1,000 iterations. The bounds of control variables should be the same as those used to produce control variables in the dataset. We perform **S6** benchmarking} to evaluate the performance of optimized control against the target quantum gate on the real device using the AGF defined in Equation 67.3.

3.1.4.2. Numerical Analysis

To verify our analysis in Section 3.1.3.1 and Section 3.1.3.2, we numerically simulate how the Graybox approach can be used for characterization and calibration for the $\sqrt{X}$ gate. We build our model based on the IBM-Q's transmon qubit device, with a qubit frequency ω_q, driven by a microwave field, resulting in an ideal qubit Hamiltonian given by [65]

$$\hat{H}(\mathbf{\Theta}, t) = \frac{2\pi\omega_q}{2}\hat{\sigma}_z + 2\pi\Omega_d s(\mathbf{\Theta}, t)\hat{\sigma}_x. \tag{84}$$

The first term describes the bare qubit Hamiltonian, while the second term represents the microwave drive at strength Ω_d. The signal function has the following form,

$$s(\mathbf{\Theta}, t) = \text{Re}\left\{h(\mathbf{\Theta}, t)e^{i(2\pi\omega_d t+\varphi)}\right\}, \tag{85}$$

where ω_d and φ are the microwave carrier frequency, which is typically set to be on resonance with the qubit, and its phase, respectively. The signal envelope is described by a function $h(\mathbf{\Theta}, t)$. The phase φ is typically used in producing a Virtual-Z gate [142], though, in our case, we set $\varphi = 0$ for simplicity.

In our numerical simulation, we test the Graybox performance by considering a scenario with a drift (or mismatch) in the qubit frequency. The real value of the qubit frequency denoted by $\omega_q' = 5.0005$ GHz is shifted from the value assumed by an experimenter, $\omega_q = 5.0$ GHz, by 0.5 MHz. This way, to simulate the real qubit's dynamics, we use Equation 84 with ω_q' for the qubit frequency, but $\omega_d = 5.0$ GHz for the microwave drive, reflecting the experimenter's wrong assumption about the resonance condition. For the drive strength, we choose $\Omega_d = 0.1$GHz, which is similar to the value that we obtain from the IBM-Q's device in the next section.

The envelope function $h(\mathbf{\Theta}, t)$ can in principle be arbitrary. However, there are limitations and constraints on its form from the actual implementation that one should consider. In our case, to mimic the experimental settings, the envelop function should be sampled with finite resolution, where the IBM Quantum device has a resolution of dt =

0.22 ns in time, and the amplitude cannot exceed the value of 1. Moreover, as the target qubit is a transmon qubit, population leakage to higher energy levels can be problematic, especially when a qubit is driven by a short pulse control [68]. The way to get around this is to use the DRAG pulse technique [75] to help mitigate the leakage problem. This involves adding a derivative of the original pulse envelope, $\frac{\partial}{\partial t} f(\mathbf{\Theta}, t)$, in the imaginary component, resulting in the control pulse envelope of the form,

$$h(\mathbf{\Theta}, t) = f(\mathbf{\Theta}, t) + i\gamma \frac{\partial}{\partial t} f(\mathbf{\Theta}, t), \tag{86}$$

where γ is the DRAG coefficient [68]. In this numerical simulation, we choose $f(\mathbf{\Theta}, t)$ to be a single Gaussian function for simplicity,

$$f(\mathbf{\Theta}, t) = \frac{A_m \theta}{2\pi} \exp\left(-\frac{\left(t - \frac{T}{2}\right)^2}{2\sigma^2} \right), \tag{87}$$

where the center of the Gaussian is at the middle of the pulse $t = \frac{T}{2}$, and the amplitude as well as the Gaussian width are chosen such that

$$\sigma = \left(A_m \sqrt{2\pi} \Omega_d \ \mathrm{dt} \right)^{-1} \tag{88}$$

is satisfied. This condition is such that the area under the envelope (pulse area) gives the angle of rotation θ around the X-axis of the qubit's Bloch sphere, given that $A_m = 0.5$ is chosen. Therefore, the set of control variables (the parameter space used in training the predictive model) in this case, $\mathbf{\Theta} = [\theta, \gamma]^\top$, consists of the rotation angle θ and the DRAG coefficient γ.

Let us make an important note here on why the DRAG technique is needed in implementing the Graybox approach on the current IBM Quantum device. The leakage problem is a very special type of noise that does not preserve even the Hilbert-space dimension of the quantum system of interest, e.g., a qubit (two levels) should be treated as a qutrit (three levels) when the leakage occurs. Therefore, if we were to include the leakage noise in the Graybox approach, we would need to model the system in a Hilbert space with higher levels, e.g, a qutrit or a qudit, depending on how bad the leakage was, and we would need to also measure observables in the same Hilbert space for data training. However, at the time of our experiments, we could only access the measured observables in the qubit's space. Thus, in this work, we restricted our analysis to the two-level system in the numerical simulation and used the DRAG technique to suppress the leakage as effectively as possible.

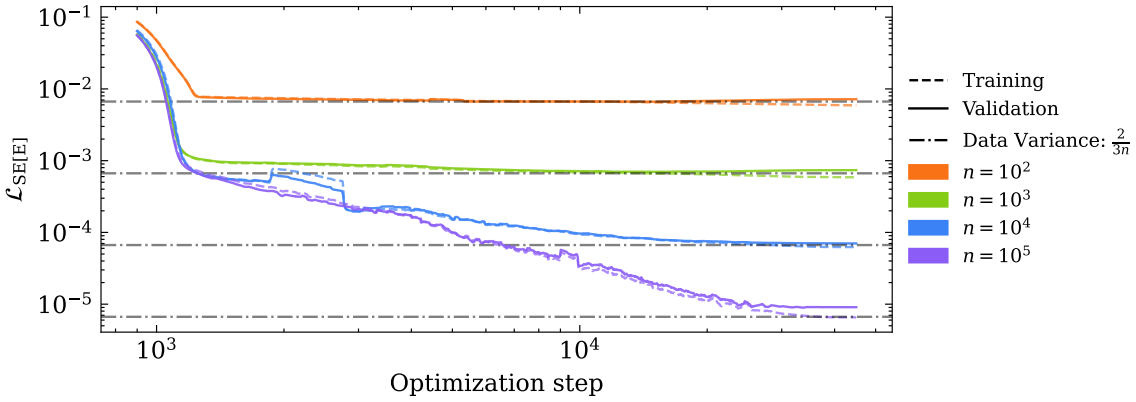


Figure 3.4: The moving average of the training and validation losses for each optimization step for each value of n with a window size of 100. The horizontal dashed lines are the data variance. This figure is used in [139].

We numerically generate the qubit's dynamics with the Hamiltonian Equation 84 using 10^3 different values of control variables, which are sampled uniformly from $\theta \in [0, 2\pi)$ and $\gamma \in [0, 10)$. For each dynamics, we simulate the measurement process with 18 sets (a combination of 6 different initial states and 3 measured Pauli observables as mentioned in Section 3.1.3.2 with a variable n number of shots used in estimating the expectation values. To analyze the performance with varying numbers of shots, we choose $n \in \{10^2, 10^3, 10^4, 10^5\}$. The dataset is then split into a training and a validation dataset, with 90:10 ratio. In the model training step, we use Equation 61 as a learning objective. We did not perform hyperparameter tuning in this numerical study, and so we fixed the number of output features of each layer of the model to a constant, resulting in a total of 3,435 trainable parameters. We refer to Appendix C of our work [139] for more details on the model selection process.

We show the results of the training and validation losses for different values of n in Figure 3.4, where the losses converge faster and reach the the data variance bound of $\frac{2}{3n}$ easier in the case of a lower number of shots n. Observe that some of the empirical train losses converge to a value slightly lower than the ideal data variance. We suspect that this is a result of the model overfitting to the training dataset since the validation losses are higher than the training losses. Moreover, it is also possible that the sample size used to estimate the empirical loss is not large enough, resulting in an inaccurate estimation of the actual value. To estimate the expected loss in Equation 65, we generate a testing dataset consisting of 1,000 samples and use them to estimate the data variance in Equation 65 (See also the histogram of $\mathcal{L}_{\mathrm{SE[E]}}$ the testing dataset in Figure 8 in Appendix C of our work [139]. We observe that the distributions of the test loss have a relatively high variance, suggesting a necessity to use a sufficiently large sample size to estimate

the empirical value. In Table 3.1, we list the average of test loss and its standard deviation, the empirical optimality (which is the optimality calculated from the test loss), and the RHS of the bound in Equation 82.

Shots	Loss	Λ	Bound (RHS)
10^2	$7.6 \pm 3.1 \times 10^{-3}$	9.4×10^{-4}	2.7×10^{-2}
10^3	$7.5 \pm 2.8 \times 10^{-4}$	8.7×10^{-5}	8.1×10^{-3}
10^4	$7.2 \pm 2.9 \times 10^{-5}$	5.0×10^{-6}	1.9×10^{-3}
10^5	$7.8 \pm 6.7 \times 10^{-6}$	1.1×10^{-6}	9.2×10^{-4}

Table 3.1: For each n, we list the average $\mathcal{L}_{\mathrm{SE[E]}}$ loss of testing dataset consist of 1000 sample. From the test loss, we estimate the empirical optimality Λ, and then use it to calculate the RHS of the bound in Equation 82. This table is used in [139].

We perform the control calibration and evaluate the AGF for each predictive model trained with an n-shot dataset. We perform control calibration by minimizing the average gate infidelity $1 - \bar{F}\left(\mathcal{E}, \hat{G}\right)$ to find a control variable for the $\sqrt{x}$ gate. We compare the values of exact and predicted AGF, calculate their difference, and list the results in Table 3.2. We denote the column of exact AGF calculated using the noisy simulator as Exact, and denote the column of predicted AGF by Model. Observe that the absolute error between exact and predicted AGF decreases as n increases, which follows the same behavior of the estimated upper bound in Table 3.1. However, the bound in Equation 82 is not tight as the bound (RHS) in Table 3.1 is multiple orders of magnitude higher than the absolute error in Table 3.2.

To investigate the effect of system parameters drifting, we execute the optimized control variables on the device with $\omega_q = 5.00075$ GHz, which further drifted from the drive frequency. In Table 3.2, we list the values of the exact AGF calculated using the drifted noisy simulator in the column denoted Drifted. We observe significant reductions in prediction accuracy, indicating that the Graybox model's predictive performance is not robust to changes in system parameters. Thus, it is necessary to perform a routine characterization to update the model. Nonetheless, our numerical results suggest that, with accurate system parameters, the Graybox model can be utilized in gradient-based optimization to produce a quantum gate with reliable accuracy.

From Equation 83, we can see that it is theoretically possible to train a predictive model that achieves an optimal performance regardless of the number of shots n. However, we do not attempt to train the optimal predictive model as heavy hyperparameter tuning might be required. Although the numerical result suggests that with the higher value of n,

the absolute error becomes smaller, a single order of magnitude of accuracy improvement may not be worth the price of multiple orders of magnitude required for the number of shots n.

Shots	Exact	Model	Absolute Error	Drifted
10^2	0.99861	0.99921	5.91912×10^{-4}	0.99475
10^3	0.99896	0.99903	7.20100×10^{-5}	0.99544
10^4	0.99993	0.99992	1.83735×10^{-5}	0.99811
10^5	0.99993	0.99993	3.63394×10^{-8}	0.99811

Table 3.2: Comparison of exact and predicted AGF for varying shot counts. The absolute error is the absolute difference between the exact and predicted AGF. We also list the AGF using the same optimized control variable but executed on the device with qubit frequency $\omega_q = 5.00075$ GHz instead of $\omega_q = 5.0005$ GHz (which produces the value in the exact column) in the drifted column. This table is used in [139].

3.1.4.3. Characterization and Calibration for Superconducting Qubit

For the experimental realization, we choose to work with superconducting transmon qubits from IBM Quantum [143], [144], specifically the qubit labelled `q2` of the 127-qubit `ibm_kawasaki` system with the qubit frequency $\omega_q = 5.081$ GHz. It is important to note that, in addition to the qubit's Hamiltonian and the microwave drive as described in Equation 84 and Equation 85, the readout processes and the qubit's state dynamics on the IBM Quantum device are implemented in a rotating frame with respect to the bare qubit frequency ω_q (which is on resonance with the microwave frequency $\omega_q = \omega_d$). Therefore, given the above information, we can now assign the ideal Hamiltonian, $\hat{H}_{\text{ideal}}(\mathbf{\Theta}, t)$, for the Graybox approach to be the transmon qubit's Hamiltonian in the rotating frame given by

$$\hat{H}_{\text{ideal}}(\mathbf{\Theta}, t) = (2\pi\Omega_d)s(\mathbf{\Theta}, t)\left[\cos\left(2\pi\omega_q t\right)\hat{\sigma}_x - \sin\left(2\pi\omega_q t\right)\hat{\sigma}_y\right], \tag{89}$$

where the bare qubit Hamiltonian is no longer present and the term with $\hat{\sigma}_x$ in Equation 84 now becomes a combination of $\hat{\sigma}_x$ and $\hat{\sigma}_y$. For this experiment, the microwave drive strength is $\Omega_d = 0.103$ GHz [65].

For the microwave drive, we use the same form of the signal function $s(\mathbf{\Theta}, t)$ as in Equation 85 and the (discrete) temporal pulse resolution of $\mathrm{dt} = 0.22$ ns. As before, we aim to characterize and calibrate the $\sqrt{X}$ gate on the superconducting qubit device. In order to identify the control variables $\mathbf{\Theta}$ for the Graybox approach, we consider a *super-*

position of multiple Gaussian pulses so that the search space is larger and more complete than a single Gaussian pulse as used in the numerical simulation Section 3.1.4.2. The concept of optimizing over the basis of pulses is also used in [145], [77]. We first define an unnormalized Gaussian function $g(\sigma, c, t) = \exp\left(-\frac{(t-c)^2}{2\sigma^2}\right)$, parametrized by a standard deviation σ and a peak time c. Then, we can define a lifted and truncated pulse for a single Gaussian envelope as[2]

$$\mathcal{G}(A, \sigma, t; c) \equiv A\frac{g(\sigma, c, t) - g(\sigma, c, 0)}{1 - g(\sigma, c, 0)}, \tag{90}$$

such that the first dt pulse value (at $t = 0$) starts at zero and the peak value at $t = c$ is the maximum at $A \in [0, 1)$. To explore different variations of pulses, let us define a superposition of Gaussian envelopes as a sum of i-pulse trains, where $i = 1, 2, ..., k$. We denote a set of $\zeta = \{\sigma_{i,j}\}$ and $\boldsymbol{A} = \{A_{i,j}\}$ for the deviation and amplitude of the i^{th} order and j^{th} pulse of that order. We choose $k = 4$ so that the total number of parameters to be optimized is reasonable. This results in the envelope function,

$$v(\boldsymbol{A}, \zeta, t; \boldsymbol{c}) = \sum_{i=1}^{4} \frac{1}{2^i} \left(\sum_{j=1}^{i} \mathcal{G}_{i,j}(A_{i,j}, \sigma_{i,j}, t; c_{i,j}) \right). \tag{91}$$

Since we aim to optimize the $\sqrt{X}$ gate, the range of ζ and $\boldsymbol{A}$ is chosen such that it contains control variables of $\sqrt{X}$ gate in a noiseless condition. For the standard deviation, we choose different constraints depending on the order i, i.e., $\sigma_i \in [a_i, b_i)$ where $[a_i, b_i)$ are chosen to be $[7, 9)$, $[5, 7)$, $[3, 5)$, and $[1, 3)$ for $i = 1, 2, 3, 4$ respectively. The peak position c of the basis envelopes depends on the order, i, and the term, j. For each i^{th} order, each j^{th} Gaussian pulse evenly places itself within the total pulse length of $T = 80$ dt ≈ 17.78 ns.

Finally, as mentioned in Section 3.1.4.2, we also need to use the DRAG technique to mitigate the transmon's leakage to high energy levels. We therefore plug-in the superposition of Gaussian envelopes into Equation 86 and get the final envelope function for the microwave drive,

$$h(\boldsymbol{\Theta}, t) = v(\boldsymbol{A}, \zeta, t; \mathbf{c}) + i\gamma\frac{\partial v(\boldsymbol{A}, \zeta, t; \mathbf{c})}{\partial t}, \tag{92}$$

where the set of control variables is now given by $\boldsymbol{\Theta} = \{\boldsymbol{A}, \zeta, \gamma\}$, with the addition DRAG coefficient $\gamma \in [-2, 2)$. In the data collection process, similar to the numerical simulation, we sample 1000 sets of control parameters, each of which requires 18 expectation value estimations of 3 observables with 6 different initial states. This results in a total of 18×10^3 quantum circuits to be executed. For each of the quantum circuits, we use $n = 3 \times 10^3$

[2]See https://quantum.cloud.ibm.com/docs/en/api/qiskit/1.2/qiskit.pulse.library.Gaussian for more details about the pulse envelope.

as the number of shots. The reason we chose 1000 samples is so that the whole experiment can fit within the limitation of 1 session (8 hours) allowed by our access to the device at the time when the experiments were performed.

For the *model selection* step, instead of fixing the hyperparameter for model training, we perform hyperparameter tuning for the number of output features of each MLP layer of the model. We randomly split the 1000 datasets into training, validation, and testing sets, each containing 800, 100, and 100 samples, respectively. Although we found that the validation set was not necessary for adjusting the learning rate or stopping training early in our experiments, we still decided to keep it in our process. We did this to ensure our approach remains comprehensive, general, and adaptable for other scenarios where the validation set may be essential. The average of $\mathcal{L}_{\mathrm{SE[E]}}$ of the testing dataset is optimized by the hyperparameter tuner.

There are two choices of loss functions considered in this study, the MSE of expectation values, $\mathcal{L}_{\mathrm{SE[E]}}$ in Equation 61, and the AE of average gate fidelities (AGF), $\mathcal{L}_{\mathrm{AE[F]}}$ in Equation 60. These two choices arise from the discussion in Section 3.1.3.1 and Section 3.1.3.2. We denote the models using the corresponding loss functions by $\check{f}_{\mathrm{SE[E]}}$ and $\check{f}_{\mathrm{AE[F]}}$, respectively.

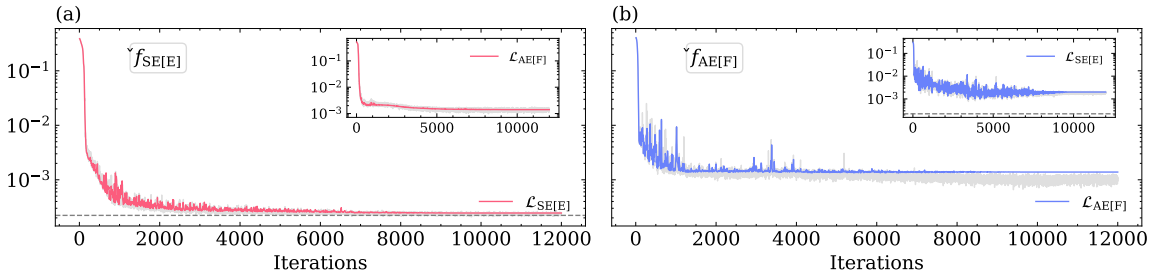
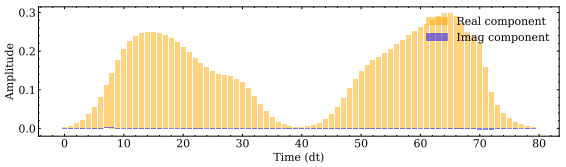


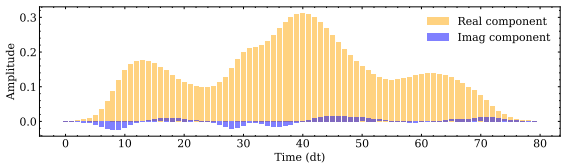
Figure 3.5: The plots of metrics over 12,000 training iterations, where the upper and lower are plots of $\check{f}_{\mathrm{SE[E]}}$ and $\check{f}_{\mathrm{AE[F]}}$, respectively. In (a) and (b), the vertical axis of the main plot is $\mathcal{L}_{\mathrm{SE[E]}}$ ($\mathcal{L}_{\mathrm{AE[F]}}$), while the vertical axis of the inset plot is $\mathcal{L}_{\mathrm{AE[F]}}$ ($\mathcal{L}_{\mathrm{SE[E]}}$). The colored line is the average value over the testing dataset, while the gray line is the average over a batch of the training dataset.

After the model selection process, $\check{f}_{\mathrm{SE[E]}}$ has 4,036 trainable parameters and $\check{f}_{\mathrm{AE[F]}}$ has 11,115 trainable parameters. Figure 3.5 shows the metrics of both models during the training phase. We plot the value of the loss function using a testing dataset with colored lines and the value of the loss function using a training dataset with gray lines. In the plots with $\mathcal{L}_{\mathrm{SE[E]}}$ as the loss function, the gray dashed lines represent the data variance of $\frac{2}{3n}$, which is the minimum value of expected $\mathcal{L}_{\mathrm{SE[E]}}$. We observe in Figure 3.5(a) that

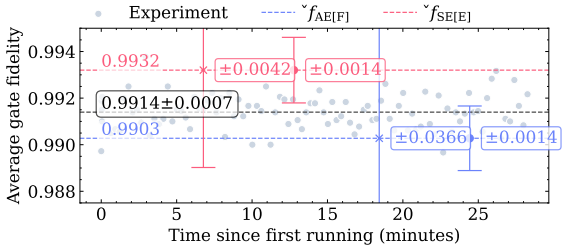
the value of $\mathcal{L}_{\mathrm{SE[E]}}$ approaches the expected minimum value. In contrast, this is not the case for $\check{f}_{\mathrm{AE[F]}}$, as shown in the inset of Figure 3.5(b), as expected because minimizing $\mathcal{L}_{\mathrm{AE[F]}}$ does not imply minimizing $\mathcal{L}_{\mathrm{SE[E]}}$. Thus, the model trained using $\mathcal{L}_{\mathrm{AE[F]}}$ is only good for the AGF prediction. That is, the expectation value predicted by $\check{f}_{\mathrm{AE[F]}}$ is not reliable. Consequently, the function of expectation values using prediction from $\check{f}_{\mathrm{AE[F]}}$ rather than AGF will be inaccurate. Furthermore, $\check{f}_{\mathrm{AE[F]}}$ is overfitting the training dataset as the gray line of $\mathcal{L}_{\mathrm{AE[F]}}$ converged to a value lower than the colored line, representing the average value of the testing dataset.



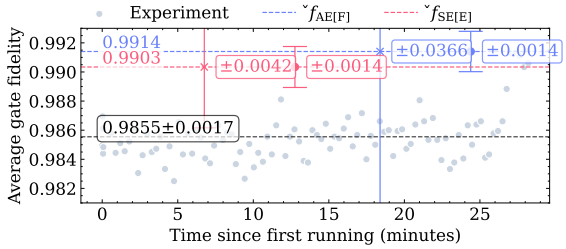
(a) Pulse sequence using $\boldsymbol{\Theta}_{\mathrm{SE[E]}}$



(b) Pulse sequence using $\boldsymbol{\Theta}_{\mathrm{AE[F]}}$



(c) Experimental data using $\boldsymbol{\Theta}_{\mathrm{SE[E]}}$



(d) Experimental data using $\boldsymbol{\Theta}_{\mathrm{AE[F]}}$

Figure 3.6: Average gate fidelities from the benchmarking experiment performed on `ibm_kawasaki` by the pulse sequence optimized using each model, each in the inset plot. The scatter plot on the left shows the 100 AGF against the time experiment executed on the device. We plot the sample mean of the data point as a dashed gray line and annotate the sample mean and sample standard deviation as black text. On the right is the histogram of AGF data points on the left. The dashed blue line is an AGF predicted by $\check{f}_{\mathrm{AE[F]}}$. The dashed red line is an AGF predicted by $\check{f}_{\mathrm{SE[E]}}$. We plot two types of error bars. The first type of error bar associated with the circle points uses the expected test $\mathcal{L}_{\mathrm{AE[F]}}$. The second type of error bar associated with the cross points is calculated using Equation 82 with the test expected $\mathcal{L}_{\mathrm{SE[E]}}$.

To evaluate our model's predictability and demonstrate the (S5) control calibration using the Graybox characterization approach, we optimized a control variable of $\sqrt{X}$ gate. We use the average gate infidelity $1 - \bar{F}\left(\mathcal{E}, \sqrt{X}\right)$ as the cost function. We denote the controls that optimize the cost $\mathcal{C}$ using the trained model $\check{f}_{\mathrm{SE[E]}}$ and $\check{f}_{\mathrm{AE[F]}}$ by $\boldsymbol{\Theta}_{\mathrm{SE[E]}}$ and $\boldsymbol{\Theta}_{\mathrm{AE[F]}}$,

respectively. The corresponding optimized pulse envelope of each model is plotted in Figure 6a and Figure 6b. (S6) Using the control pulses, we then performed direct AGF estimation repeatedly 100 times consecutively on `ibm_kawasaki` for each control variable. We note that there were time gaps between when the characterization experiment finished and the benchmarking experiment started. For $\boldsymbol{\Theta}_{\mathrm{AE[F]}}$ and $\boldsymbol{\Theta}_{\mathrm{SE[E]}}$, the gaps were approximately 2 hours 38 minutes and 3 hours 12 minutes, respectively.

The plots of AGF of both models are shown in Figure 6c and Figure 6d. The uncertainties are calculated as follows. (1) For the error bar associated with the circle mark, we use an average of $\mathcal{L}_{\mathrm{AE[F]}}$ over the testing dataset, denoted by $\mathbb{E}_{\mathrm{test}}\left[\mathcal{L}_{\mathrm{AE[F]}}\right]$, which is the uncertainty based on the expected performance of the model when predicting AGF. We used the expected value over the testing dataset to represent the model's performance on unseen data. We refer to this uncertainty as an expected performance. (2) For the error bar associated with the cross mark, we substitute the expected test $\mathcal{L}_{\mathrm{SE[E]}}$ of each model in Equation 82. We refer to this uncertainty as an upper bound. (3) For the experimental result, we use the standard deviation of experimental data.

From Figure 3.6, we observe that the sample mean of the experimental AGF deviates from the predicted AGF for both models. Additionally, the experimental AGF values fall outside the expected performance range, particularly for $\boldsymbol{\Theta}_{\mathrm{AE[F]}}$, where even the upper bound fails to encompass the experimental data distribution. One possible explanation is parameter drift or noise over time, given that the dataset was collected long before the AGF estimates. As noted in Section 3.1.4.2, the Graybox model's predictive accuracy is not robust against the system parameter drift. Figure 6d shows a shift in experimental AGF data points near the end of the benchmark experiment for $\check{f}_{\mathrm{AE[F]}}$, further indicating the system drift. However, in the subsequent benchmark for $\check{f}_{\mathrm{SE[E]}}$, a significant drift was not observed. Furthermore, $\check{f}_{\mathrm{AE[F]}}$ overfits the training dataset, as is evident in Figure 3.5, which may undermine its predictive performance on unseen data.

Based on our numerical and experimental results, we anticipate that successful practical application of the Graybox characterization method requires attention to the following factors.

Sufficient sample size: As shown in Equation 83, although the model could be trained with a low number of shots n, the sample size for training the Blackbox component must still be sufficiently large. To determine a suitable sample size, for a fixed choice of model architecture, one can perform hyperparameter tuning using the dataset available. A sufficiently large sample size is determined by the closeness of the training and testing MSE values, i.e., $\mathbb{E}_{\mathrm{train}}\left[\mathcal{L}_{\mathrm{SE[E]}}\right] \approx \mathbb{E}_{\mathrm{test}}\left[\mathcal{L}_{\mathrm{SE[E]}}\right]$ and that both MSEs converger to the data variance (as a lower bound of the expected loss defined in Equation 65). Interestingly, in the probabilistic paradigm, the statistical analysis suggests that the model's predictive

performance relies on its ability to predict the hidden expected value (the expectation value without the finite-shot effect), which implies that datasets with high-shot estimates of expectation values should help the model be more accurate. Results from statistical and probabilistic paradigms suggest that balancing between the shot and the sample may further improve model performance and the model selection process. For example, since we are approximating a function of control that outputs the expectation value, and the choice of the control makes the function "well-behaved" (e.g., interpolatable, continuous, and no sudden jump), we may use fewer samples to construct the dataset, while each sample may have more shots to increase the accuracy and precision of expectation value estimation.

Parameter drifting: As system's parameters may drift over time, it is important to minimize the time between characterization, calibration, and implementation of calibrated controls. We note that the problem of parameter drift is not specific to the Graybox approach, but rather affects accuracy of any noise characterisation-calibration techniques. In order for the Graybox method to be effective, we require that the noise drift timescale be equal or longer than the time taken by the Graybox processing time. In such cases, one can then perform periodic performance check over time, to track any changes in the system's parameters. For the Graybox approach, this can be done by collecting data to compute the test MSE of expectation values and use Equation 82 to estimate the upper bound of the AGF difference. This upper bound can be used as a simple "threshold" indicator for the quantum gate's performance maintenance.

Leakage problem on superconducting qubits: From our discussion in Section 3.1.4.2, the leakage problem in the transmon qubit, i.e., population leakage to higher energy levels, could be managed through the DRAG (pulse) technique, while keeping only a two-level qubit's model. However, if somehow the leakage information was available or the observables in larger Hilbert spaces could be measured for model training, then the Graybox approach could be modified to include higher levels, such as a qutrit or a qudit.

Other possible improvement: On real devices, state preparation and measurement (SPAM) errors can also have effects on the predictive model. Explicitly modifying the Graybox model to address SPAM in experimental data could be advantageous, especially when SPAM errors are significant.

3.2. C&C: Probabilistic Machine Learning

In this section, we will explore probabilistic machine learning. The most appealing feature is that it provides a natural way to quantify uncertainty of prediction. With uncertainty quantification associate to the prediction, i.e. predicting the distribution of the output instead of point estimation, a derived quantity would also be a distribution, i.e. has the

associate uncertainty. Moreover, uncertainty quantification can also be used to make an informed decision. That is, we will now know how certain the model is of its prediction. With statistical learning method, the model cannot say "I'm not sure", or "I don't know", but with probabilistic learning method, the model can. This is possible as probabilistic learning method constructs a predictive model using Bayesian Inference. Thus, instead of training to find model parameter, it will perform "**inference**" for model parameter instead. At the higher level, probabilistic learning method is to find a probabilistic model that produce a distribution as close as a distribution that generate observed data. The ability say how certain it is to its prediction, allows us to perform the experiment where we are uncertain the most.

We will take the next step by defining a predictive model in the probabilistic way, and point out the differences from the statistical predictive model. Then, we will review how does model construction in PML work. As an example, we discuss how we transform statistic Graybox model to the probabilistic Graybox model using Bayesian Neural Network (BNN). Finally, we will discuss what we can learn from using PML to characterize the quantum device.

3.2.1. Predictive model, but probabilistic

The raw measurement result of quantum device is stochastic by nature. Statistical predictive model aims to predict expectation value of an observable given some initial state and control parameters. The expectation value is deterministic in the ideal case, i.e. without noise and infinite number of measurement result. However, with finite-shots constraint in the realistic setting, the expectation value is stochastic, and can be viewed as a random variable. Consequently, it becomes a limiting factor of the expected minimum MSE loss that the statistical model can achieve. On the other hand, a probabilistic predictive model, or simply probabilistic model predicts distribution of the output data instead. Thus, it naturally fits well with the nature of the experiment data.

To model the probabilistic model is to model the physical process that generates our observed data. This requires prior knowledge (domain specific knowledge), so that we can model our transformation as close as possible to the actual process. For instance, part of the process might be a probabilistic event that depends on other probabilistic events. Consequently, we have to assign a particular type of distribution to the random variable, and set priors on the distribution parameters. For example, if we are trying to model the weather in Thailand during the day, we may model the observed temperature as a Gaussian distribution based on our experience, and we might want to set the initial parameter (prior) at 30 Celsius instead of 5 Celsius.

I will start the modeling with the output of the observed data first as we are fairly confident in our prior on the measurement process. There are two possible ways to model the output of the probabilistic model for quantum device characterization. (1) Predicting expectation value of observable by modeling it as a Gaussian distribution

$$y \sim \mathcal{N}\left(\langle \hat{O} \rangle_{\rho_0}, \sqrt{\mathrm{Var}\left(\hat{O}\right)_n}\right) \tag{93}$$

where n is the number of shots used to estimate the expectation value. (2) Predicting the raw measurement result as a Bernoulli Distribution,

$$y \sim \mathrm{Bern}\left(\frac{1 + \langle \hat{O} \rangle_{\rho_0}}{2}\right). \tag{94}$$

The former is a direct analogy to the statistical version, while the latter is one step closer to the actual physical process. Note that the expectation value as a prior does not restrict a posterior to be the ideal value, but it served as our belief of the whereabouts of the true value, in case of the presence of the noise. We note that the modelling in Equation 94 works for a single qubit case only.

3.2.2. Characterization using a Probabilistic Model

We now discuss the conventions using in the PML paradigm. The output label is typically referred to as our observable or **observed data**. The trainable parameter (model parameter) that was trained for in the SML paradigm is referred to as a **latent variable**.

The predictive models defined in Equation 93 and Equation 94 require different observed data. I will refer to them as the Gaussian-based model and the Bernoulli-based model respectively. The Gaussian-based model required observed data to be the finite-shots expectation value of observable. The Bernoulli-based model required observed data to be the binary measurement result from device. Both observed data types can be converted into each other in the single qubit case. In fact, the finite-shots expectation value is a compact representation of binary results. Since we do not need a temporal order of binary data, bit $b \in \{0, 1\}$, that used to estimate finite shots expectation value, we can recover the number of each bit n_b by multiplying the number of shots n by the expectation value $\mathbb{E}[\hat{O}]_{\rho_0}$ and using the constraint $n = n_0 + n_1$. Formally the finite-shot expectation value can be calculated with

$$\mathbb{E}[\hat{O}]_{\rho_0} = \frac{1}{n}(n_0 + n_1), \tag{95}$$

and then by substituting $n_1 = n - n_0$, we finally get

$$n_0 = \frac{n}{2}\Big(\mathbb{E}[\hat{O}]_{\rho_0} + 1\Big). \tag{96}$$

For ease of calculation, we can store n_0 and n_1 instead of $\mathbb{E}[\hat{O}]_{\rho_0}$.

Regardless of the model choice, we will now discuss how we can use Bayesian Inference to infer the latent variables and make a prediction, following the process described in the review in [122], [136]. Suppose that our dataset is generated from a likelihood $p(\mathcal{D}, w)$ where w are some parameters. From Bayes' rule, the **posterior distribution** is

$$p(w, \mathcal{D}) = p(w)\frac{p(\mathcal{D}|w)}{p(\mathcal{D})}, \tag{97}$$

where $p(w)$ is a prior distribution of w and $\mathcal{D}$ called *evidence* or *marginal likelihood*. During the predicting phase of a new observation $\boldsymbol{y}^*$ given an input $\boldsymbol{x}^*$, the (posterior) predictive model is then

$$p(\boldsymbol{y}^*|\ \boldsymbol{x}^*, \mathcal{D}) = \int p(\boldsymbol{y}^*|\ \boldsymbol{x}^*, w)p(w|\mathcal{D})dw. \tag{98}$$

In words, we start by inferring for the posterior distribution of model parameters first, then we use it to make the **posterior predictive distribution**, i.e. producing the distribution of the prediction.

The problem now lies in the calculation of the posterior. In PML literature, the evidence term is the term that is mostly impossible to obtain a closed-form in general, i.e. **intractable**. There are several methods that are developed to approximate the posterior. In this dissertation, we consider a **Variational Inference (VI)** method for its computational performance.

In VI, we introduce a tractable variational posterior distribution $q_\theta(w)$ to approximate the true posterior. That is, we have to perform optimization for variational parameters θ to find $p(w|\ \mathcal{D}) \approx q_{\theta(w)}$. Objectively, we have to minimize the Kullback-Leibler (KL) divergence between $p(w|\mathcal{D})$ and $q(w|\ \theta)$ which is

$$\mathrm{KL}(q_\theta \| \ p(w|\mathcal{D})) = \int q_\theta(w) \log q_\theta \frac{w}{p(w|\mathcal{D})} dw, \tag{99}$$

or in the tractable form,

$$\mathrm{KL}(q_\theta \| \ p(w|\mathcal{D})) = \underbrace{-\int q_\theta(w) \log p(w) \frac{p(\mathcal{D}|w)}{q_{\theta(w)}} dw}_{\text{ELBO}} + \log p(\mathcal{D}). \tag{100}$$

Since the log of evidence does not depend on variational parameters, maximizing the Evidence Lower Bound (ELBO) is equivalent to minimizing KL divergence. Therefore, in practice, we optimize for θ by minimizing the negative ELBO,

$$\mathrm{ELBO}(\theta) = -\ \mathrm{KL}(q_\theta \| p) + \log p(\mathcal{D}). \tag{101}$$

In practice, it is harder to perform inference with the Gaussian-based model. This is because the expectation value of an observable is constrained within the range of $[-1, 1]$. At the edges, i.e. $\langle \hat{O} \rangle_{\rho_0} \in \{-1, 1\}$, the variances become 0, where our current implementation using `numpyro` cannot handle it properly.

3.2.3. Probabilistic Graybox model with Bayesian Neural Network

The transition of the statistical version of the Graybox model in Section 3.1.2 to a probabilistic version, Probabilistic Graybox Model (PGM), is straightforward to implement. The stochastic element within the Graybox model is the Blackbox part where we can view the model parameters as random variables. Otherwise, Whitebox is a deterministic transformation. Thus, we need to implement a Probabilistic Blackbox model (PBM) that predicts a distribution rather than a point prediction.

Depending on the chosen statistical learning method, the PBM implementation may need to be significantly modified. However, for DNN, a statistical Blackbox model (SBM), we can directly convert it to BNN. The key difference between BNN and a statistical NN is that the trainable parameters are distributions rather than point values. We have the freedom to choose the distribution type (which can be mixed), which can directly reflect our prior knowledge of the parameters. Normal distributions with zero mean and variance of unity are the generic choice $\mathcal{N}(0, \mathbb{I})$. BNN performs prediction by sampling from the posterior distribution of model parameters; the model's output is then calculated from the parameter samples using standard matrix multiplication. The ensemble of outputs from multiple samplings is then the distribution of model outputs. We illustrate the Blackbox part of the Graybox model in Figure 3.7.

Promoting DNN to BNN seems like an upgrade at first glance; however, in practice, BNN might not necessarily always perform better than DNN. From [121], authors state that BNN can learn a small sample size without overfitting. If true, it would be highly beneficial for quantum device characterization, since collecting data points in an experiment is expensive.

It is worth considering the situation where PGM might offer an advantage over Statistic Graybox Model (SGM). Let us consider the situation where there is a stochastic noise hidden in the system. For example, colored noise is caused by some unknown Power Spectral Density (PSD). This type of noise exists in various qubit platforms [146], such

as superconducting qubits, trapped ions, and nuclear spins. Typically, researchers use specialized methods [71], [73], [146], [147], [148] to handle colored noise. Regardless of the source of the stochastic noise, we focus our attention on the effect of the stochastic noise on the statistics of the finite-shot expectation value. We refer to our work [149] for more detailed analysis. Nonetheless, we review the results again in this dissertation.

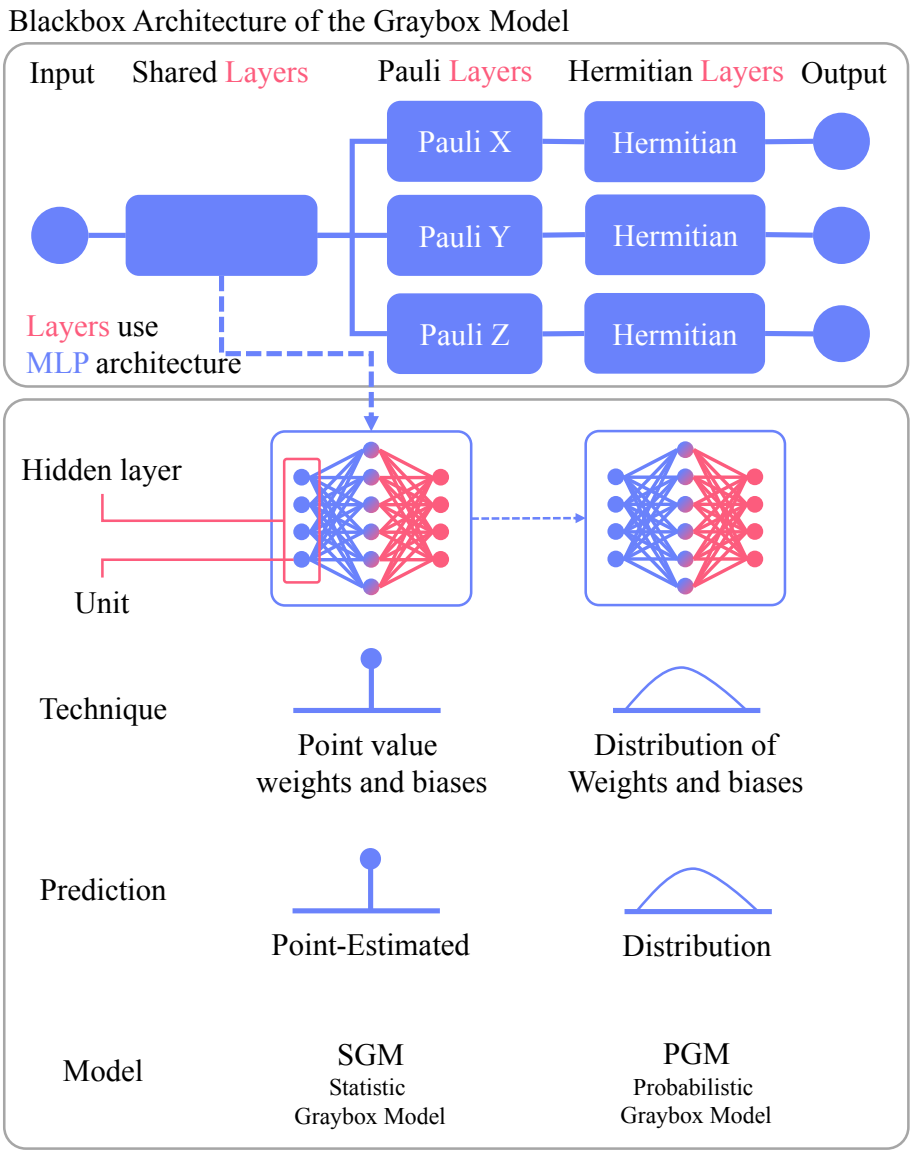


Figure 3.7: An illustration of the Blackbox part of the SGM and PGM. In the upper part of this figure, the Blackbox receives an input, which is the control parameters, and then processes it through the Shared Layers. Then we feed the results to the Pauli layers independently. For each Pauli Layer, the transformed result is sent to its corresponding Hermitian Layer, which transforms the input into the parameters that parametrize the Hermitian operator. In the lower part of this figure, we highlight the differences between the statistical and probabilistic models. This figure is used in [149].

We are interested in estimating the intermediate expectation value $\langle \hat{O} \rangle_{\rho_0}$ as defined in Equation 42. However, due to the nature of the quantum system, we can only observe a

binary measurement outcome, which can be mapped to the eigenvalue $e \in \{-1, 1\}$. The eigenvalue e_i is a random variable,

$$e_i \sim \text{Bern}\left(\frac{1 + \langle \hat{O} \rangle_{\rho_0}}{2}\right), \tag{102}$$

which depends on the value of $\langle \hat{O} \rangle_{\rho_0}$. We note that we keep the control parameters $\boldsymbol{\Theta}$ fixed and thus drop the explicit dependence in the following analysis. We then estimate $\langle \hat{O} \rangle_{\rho_0}$ using the estimator defined in Equation 44. With stochastic noise, the intermediate expectation value becomes a random variable rather than a fixed value. To analyze the effect of random intermediate expectation values, we first assume the expected value and variance of the intermediate expectation value to be,

$$\mathbb{E}\left[\langle \hat{O} \rangle_{\rho_0}\right] = \mu_0, \text{Var}\left(\langle \hat{O} \rangle_{\rho_0}\right) = \sigma_0^2. \tag{103}$$

Now, we can calculate the expected value of the estimator by

$$\mathbb{E}\left[\mathbb{E}\left[\hat{O}\right](\rho_0)\right] = \frac{1}{n}\sum_i^n \mathbb{E}[e_i]. \tag{104}$$

Since e_i is a random variable that depends on another random variable, we have to use the law of the total expectation as follows,

$$\mathbb{E}[e_i] = \mathbb{E}\left[\mathbb{E}\left[e_i \mid \langle \hat{O} \rangle_{\rho_0}\right]\right]. \tag{105}$$

The expected value of eigenvalue e_i given that the intermediate expectation value is fixed to $\langle \hat{O} \rangle_{\rho_0}$, is

$$\mathbb{E}\left[e_i \mid \langle \hat{O} \rangle_{\rho_0}\right] = (+1)\left(\frac{1 + \langle \hat{O} \rangle_{\rho_0}}{2}\right) + (-1)\left(\frac{1 - \langle \hat{O} \rangle_{\rho_0}}{2}\right) = \langle \hat{O} \rangle_{\rho_0}. \tag{106}$$

Thus, substituting back into the total expectation yields,

$$\mathbb{E}[e_i] = \mathbb{E}\left[\langle \hat{O} \rangle_{\rho_0}\right] = \mu_0, \tag{107}$$

by the assumption of the intermediate expectation value. Therefore, the expected value of the estimator is

$$\mathbb{E}\left[\mathbb{E}\left[\hat{O}\right](\rho_0)\right] = \frac{1}{n} n\mu_0 = \mu_0. \tag{108}$$

Now, let us consider the variance of the estimator, which is

$$\mathrm{Var}\Big(\mathbb{E}\Big[\hat{O}\Big](\rho_0)\Big) = \mathrm{Var}\left(\frac{1}{n}\sum_i^n e_i\right) = \frac{1}{n^2}\sum_i^n \mathrm{Var}(e_i). \tag{109}$$

So, we have to calculate the variance of the eigenvalue. Again, we have to calculate the variance using the law of total variance given by

$$\mathrm{Var}(e_i) = \mathbb{E}\Big[\mathrm{Var}\Big(e_i|\langle\hat{O}\rangle_{\rho_0}\Big)\Big] + \mathrm{Var}\Big(\mathbb{E}\Big[e_i|\ \langle\hat{O}\rangle_{\rho_0}\Big]\Big). \tag{110}$$

Considering the first term, the conditional variance is

$$\mathrm{Var}\Big(e_i|\langle\hat{O}\rangle_{\rho_0}\Big) = \mathbb{E}\Big[\Big(e_i - \mathbb{E}\Big[e_i \mid \langle\hat{O}\rangle_{\rho_0}\Big]\Big)^2 \mid \langle\hat{O}\rangle_{\rho_0}\Big] \tag{111.1}$$

$$= \mathbb{E}\Big[\Big(e_i - \langle\hat{O}\rangle_{\rho_0}\Big)^2 \mid \langle\hat{O}\rangle_{\rho_0}\Big] \tag{111.2}$$

$$= \mathbb{E}\Big[e_i^2 - 2e_i\langle\hat{O}\rangle_{\rho_0} + \langle\hat{O}\rangle_{\rho_0}^2 \mid \langle\hat{O}\rangle_{\rho_0}\Big] \tag{111.3}$$

$$= \mathbb{E}\Big[e_i^2 \mid \langle\hat{O}\rangle_{\rho_0}\Big] - \langle\hat{O}\rangle_{\rho_0}^2. \tag{111.4}$$

In the last equation, the conditional expected value of a random variable given a fixed value of itself should be itself. Now consider the first term of the last equation,

$$\mathbb{E}\Big[e_i^2 \mid \langle\hat{O}\rangle_{\rho_0}\Big] = (+1)^2\left(\frac{1+\langle\hat{O}\rangle_{\rho_0}}{2}\right) + (-1)^2\left(\frac{1-\langle\hat{O}\rangle_{\rho_0}}{2}\right) = 1. \tag{112}$$

So

$$\mathrm{Var}\Big(e_i|\langle\hat{O}\rangle_{\rho_0}\Big) = 1 - \langle\hat{O}\rangle_{\rho_0}^2. \tag{113}$$

The above equation is not surprising, since it is a variance of a quantum observable given that the observable is Hermitian. Back to the first term of total variance, we have

$$\mathbb{E}\Big[\mathrm{Var}\Big(e_i|\langle\hat{O}\rangle_{\rho_0}\Big)\Big] = 1 - \mathbb{E}\Big[\langle\hat{O}\rangle_{\rho_0}^2\Big]. \tag{114}$$

For the second term, we can use the result from Equation 106. Combining the first and second terms together, the total variance is

$$\mathrm{Var}(e_i) = 1 - \langle\hat{O}\rangle_{\rho_0}^2 + \mathrm{Var}\Big(\langle\hat{O}\rangle_{\rho_0}\Big). \tag{115}$$

From the definition of variance of a random variable X, we have

$$\mathrm{Var}(X) = \mathbb{E}[X^2] - (\mathbb{E}[X])^2. \tag{116}$$

Thus, the total variance can be simplified as,

$$\mathrm{Var}(e_i) = 1 + \Big(-\mathbb{E}\Big[\langle\hat{O}\rangle_{\rho_0}\Big]^2\Big) = 1 - \mu_0^2 \tag{117}$$

Finally, the variance of the estimator is

$$\mathrm{Var}\left(\mathbb{E}\left[\hat{O}\right](\rho_0)\right) = \frac{1}{n^2} n(1-\mu_0^2) = \frac{1}{n}(1-\mu_0^2). \tag{118}$$

This is similar to the variance of the estimator in the deterministic case, where it depends only on the mean of the intermediate expectation value and is independent of the variance. The result suggests that we cannot infer the variance of the intermediate expectation value from the finite-shot expectation value. In a similar manner, modeling the intermediate with a probability distribution does not necessarily provide an advantage over a simple point estimation.

Let us consider the types of uncertainties and their effects in our setting. Typically, we can categorize uncertainty into model (epistemic) uncertainty and data (aleatoric) uncertainty arising from noise in the observed data, represented by the distribution of observed data. In a probabilistic model, the model uncertainty is accounted for by the posterior predictive distribution [122]. In the statistical model, we can address this uncertainty using techniques such as ensemble learning. Since we know from the analysis that the distribution of the observed data depends only on the hidden expected value, we can quantify data uncertainty by using the model's intermediate expectation value to produce a distribution of the finite-shot expectation value. In other words, the predicted finite-shot expectation value distribution will take the same form as the observed distribution. In the case that the model produces an accurate prediction (predicting the expectation value equal to the hidden expected value in the case of SGM, and predicting a distribution where the expected value is equal to the hidden expected value in the case of PGM), regardless of the model uncertainty, the model will accurately quantify the data uncertainty. Consequently, if we use the model to calibrate the quantum gate, the hidden distributions on the device produced by the optimized control parameters will have the same expected values as the observed distributions of the finite-shot expectation value.

We have numerically shown in [149] and will reproduce the result in Chapter 6 that PGM can capture the distribution of the observed data better than SGM, with a 1.9-fold improvement. Furthermore, we use PGM and SGM to estimate the control parameter for the $\sqrt{X}$ gate. We observed that the solution from PGM is closer to the global optimum than the solution from SGM. Furthermore, a probabilistic model is required for the BOED utilization as well. We note that the Bernoulli-based model only works for a single-qubit case. For two-qubit or more cases, we may have to reformulate the PGM. One of the possible ways is to use a binary associated with the parity of the measurement outcome instead.

3.3. Data-Efficient approaches

TL;DR
The data-efficient approach generally involves quantifying the current (prior) knowledge of the system, estimating some utility value, such as expected information gain, based on the prior knowledge, and collecting data based on the estimated value. The steps are repeatedly performed until some conditions are met.

From the discussion in Section 2.4, there are a number of ways to reduce the consumption of data in the model characterization (not necessarily specific to a quantum system). Here, we are going to discuss them in the context of quantum device characterization and a possible uses of the Equation 65 to reduce the data consumption.

3.3.1. Data-variance

Now, we are discussing how we can use data variance and performance bound to improve and potentially speed up our characterization.

Iterative data-collecting: Without the data variance in Equation 65, we typically need to collect more data than enough for the model, such that we can use model selection to select the most performant model. With Equation 65, we can assess the predictive model's optimality. In the case of a statistical predictive model, we can use its prediction directly. In the case of a probabilistic model, we have to use the prediction's mean as the representation. Either way, we can perform experiments on the dataset in sequence. For each iteration, we use all of the data collected so far to characterize the model. Then, using the testing dataset, we can estimate the model's optimality, at which point we can stop data collection when certain thresholds are reached. For instance, consider the noise model and dataset in Chapter 6; we can stop SGM model characterization at 200 samples rather than 900, as shown in Figure 3.8. The simplicity of this method allows us to quickly assess the model's quality with less computational effort.

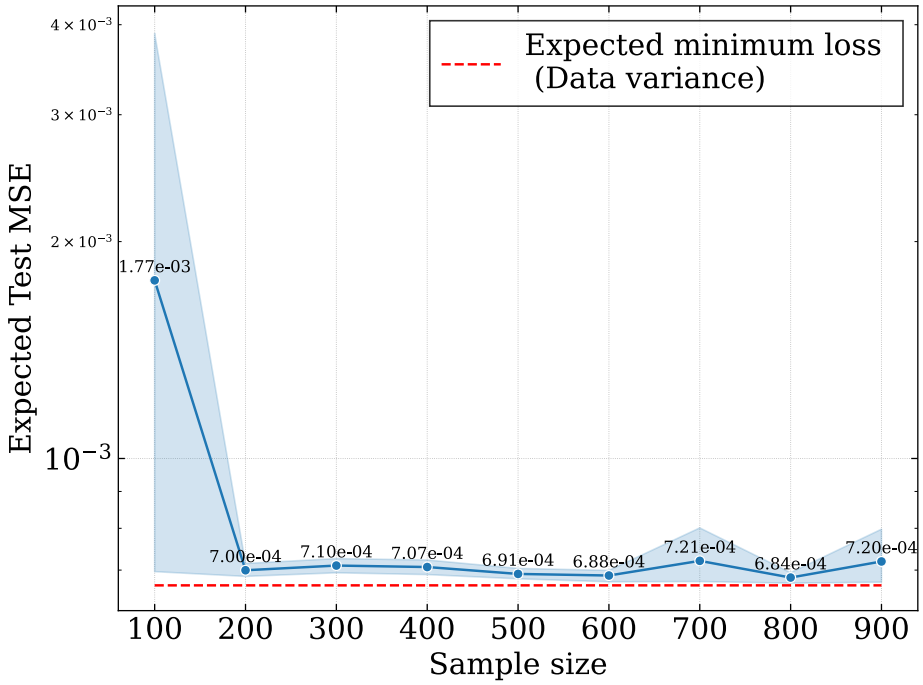


Figure 3.8: We can iteratively collect a small set of samples and characterize the model, so we collect just enough data from the device rather than too much. For example, we can stop data collection at 200 samples rather than 900, since the expected test MSE is already close to the data variance in this case.

Early stopping mechanism: During a model selection process, we typically choose a model based on its performance on the validation or testing dataset and (optionally) a simpler model. If the model candidate does not perform well, e.g., shows an early sign of overfitting, we can terminate the computation early to save resources. With data variance, we can design an early stopping mechanism based on the expected test mean squared error. For example, we can choose the model in which both the testing and training expected mean squared errors of expectation values are close to the data variance. Moreover, if the trial already meets the criteria, we can stop the training effort early, since further training is unlikely to yield substantial improvement. Otherwise, we can terminate the model training early. The benefit of this method is that we can choose a model that predicts close to the system rather than selecting the best candidate.

Moreover, from Equation 65, we can estimate the quantity by sampling the control parameters x_1 first, then performing experiments multiple times with the control parameters to collect the sample y from the conditional distribution $p(y|x_1)$. We then repeat this with $x_2, ..., x_m$ to approximate the joint distribution $p(x, y)$. With this method, we form the specialized dataset such that we can also benchmark the model performance

using probability distribution measures, such as Jensen-Shannon Divergence (JSD), in addition to the optimality of the model.

3.3.2. One-step Optimal BOED

As discussed in Section 2.4, we can calculate EIG using estimators proposed in [55] and simply select the control parameters that maximize the EIG as the next design for the experiment. This procedure is referred to as one-step optimal in the information-theoretic perspective.

In [55], the authors demonstrated the example use case of EIG with the memory capacity experiment. The concept is, however, general. In our context, we can also apply the use of EIG as well. Thus, to estimate the EIG, we need a probabilistic model of the quantum system. Fortunately, we have constructed and discussed such a probabilistic model in Section 3.2. There are wide classes of probabilistic models that can model the quantum system. For instance, from the parametrized Hamiltonian, we can identify some or all of the system parameters (qubit frequency, detuning, etc.) as random variables. We then apply the BOED to identify those system parameters. Alternatively, we can also consider the probabilistic Graybox model where we promote the DNN to BNN. The model parameters of the BNN, which are the weights of the model, are the random variables. So, we apply the BOED to identify those weights. Either way, we expect that with the BOED approach, we can identify the target parameters with fewer experiments than the traditional approaches. We will use the one-step optimal approach for the early implementation of Inspeqtor.

3.3.3. Sequential-BOED

We have discussed from the results presented in previous works that one-step BOED does not necessarily yield the highest total EIG in the sequential experiments setting. Successive works proposed the sequential-BOED approaches, which aim to maximize total EIG rather than a single-step EIG. However, intensive precomputation is required.

To handle the computationally intensive requirement of the schemes developed for sequential-BOED [127], [128], we can strategize the selection of experimental designs based on one-step EIG. For demonstration purposes, we will consider one of the possible strategies in Section 6.5, namely, a **subspace strategy** that divides the control parameters space into multiple subspaces, then selects the designs that maximize the EIG within the subspace. The strategy is motivated by the fact that we want to perform inference for BNN (the Blackbox part of the Graybox model). Hence, we want to retain some balance (to not fully bias the selection to EIG) in our dataset.

It is also possible to use Inspeqtor to perform sequential-BOED as proposed in [127], [128], however, we do not explore them numerically in this dissertation since they require too much computational resources.

Chapter 4. Beyond a Single Processor

TL;DR
In this chapter, we discuss the necessity of a quantum network and how characterization and calibration can help improve the system's performance. Specifically, how can we use a method such as the Graybox characterization method in the quantum network?

So far, we have discussed the quantum device by considering only qubits within the same processing unit. However, in the future, much like traditional computers, quantum computers would likely have to operate together. To be more specific, communication between remote quantum devices is necessary, whether for executing a distributed algorithm [2], [150], [151], [152], building a better interferometric telescopes [153], or for simple quantum information transfer [154]. To establish the connection, two quantum devices must share a system of two entangled qubits in a specific state, such as a Bell state. With the Bell state, quantum information can be transferred via quantum teleportation, and a two-qubit gate between qubits on different chips can be realized, for example, via a teleported CNOT gate [155], [156]. Thus, the core idea of a quantum network is that we can share a "resource" between remote devices.

Even if the core idea is simple, as in the traditional Internet, there are numerous challenges to be addressed before we can achieve a practical, working quantum internet. The challenges include, but are not limited to, path routing, noise handling, resource management, and networking protocols. In this chapter, we will discuss resource distribution approaches, from entanglement swapping-based to measurement-based approaches, which offer some of the solutions to the challenges. We also discuss emulating the quantum network protocol on a superconducting qubit device. We then discuss how we could use `inspeqtor` and the Graybox characterization method to improve the experimental results and how we could apply them in a more realistic setting.

4.1. Entanglement Swapping Based Approach

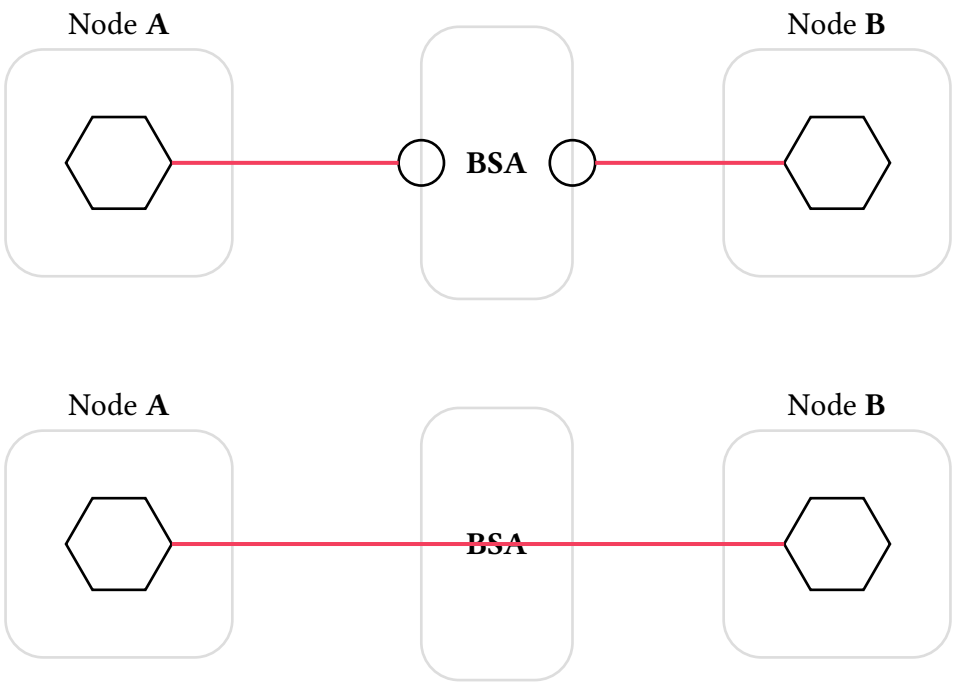


Figure 4.1: Basic entanglement swapping with BSA positions in between Node A and Node B. In the upper part, Nodes A and B send their photonic qubits to the BSA to perform a Bell measurement. In the lower part, after the Bell measurement and correction operation, the matter qubits on both sides become a Bell pair.

Handling the logistics of entangled resource distribution by car, plane, or even ship is obviously inefficient. Thus, the optimal way is to distribute them via a photon, which is another possible physical realization of a qubit. Let us consider the schematic illustrated in Figure 4.1. The idea is that we have both devices create a Bell pair from a matter qubit and a photonic qubit, then guide the photonic qubits of each device through the medium (fiber or free-space), and then measure both of them using a Bell State Analyzer (BSA), which will mathematically measure both photon qubits in the Bell basis. After that, the measurement results should be sent to end devices to perform correction operations based on the measured results. With the correction operations, the matter qubit in each device will form a new Bell state, ready to be used in a desired application. This simple protocol to distribute the entangled resource might be usable over short distances. However, it will become impractical over long distances due to photon loss. Thus, a specialized quantum device, a quantum repeater, is needed to shorten the distance that the photonic qubit has to travel.

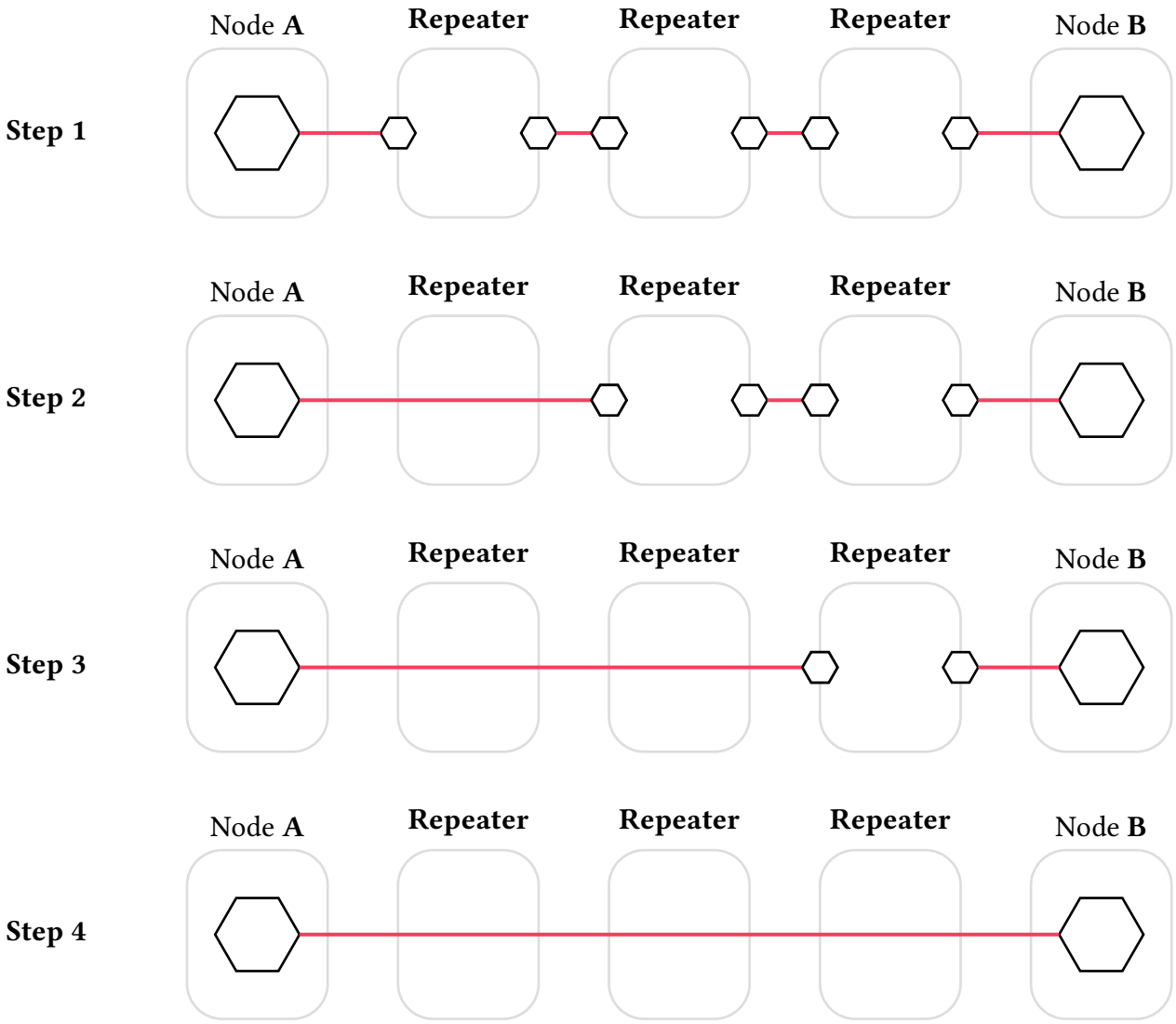


Figure 4.2: Sequential entanglement swapping across multiple quantum repeaters. Step 1: Start with the assumption that adjacent nodes share a matter-matter Bell pair between them. Sequential entanglement swapping is then performed from left to right in Steps 2-4.

To enable even longer-distance connections, we consider the following setup, as illustrated in Figure 4.2. We set up the quantum repeaters between the end quantum devices that need to share an entangled resource, forming a linear chain. Next, we share Bell pairs between adjacent nodes as in Step 1. Then, we sequentially measure both qubits in the repeater in the Bell basis and perform the correction operation in the same manner as the previous setup. We will end up with the Bell pair shared between the end devices in Step 4.

4.2. Measurement-based Quantum Network Coding

We have mainly discussed the resource distribution via entanglement swapping. However, to distribute Bell pairs to quantum devices, we can also use a graph state (which is also an entangled state). As the name suggests, a graph state can be visualized as a graph in which vertices are qubits and edges represent connections between them. Instead of entanglement swapping, we measure the qubit (node) in an appropriate Pauli basis, thereby transforming the graph's topology. This scheme is especially useful when network resources are congested. This approach is called Measurement-based Quantum Network Coding (MQNC) [157].

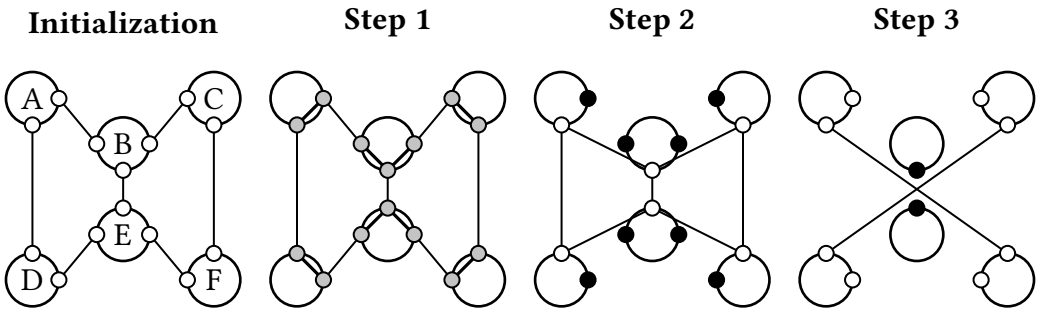


Figure 4.3: Transformation of graph state connectivity via X-measurement. We adapt this figure from [158].

For example, consider the network topology in the initialization step of Figure 4.3. There are six network nodes in the graph, where each node has qubits that are entangled with other qubits (as a Bell pair) at the other nodes, as represented by lines. If A wishes to connect with F and C wishes to connect with D, they cannot achieve simultaneous connections via entanglement swapping. However, it is possible with MQNC. In this case, the connection (edge) is a graph state rather than a Bell pair. Note that we can transform a two-qubit graph state into a Bell pair with a single qubit operation and vice versa. We start by connecting the subsystem into a larger graph state with a controlled-Z gate within the nodes, as illustrated in Step 1 of Figure 4.3. In Step 2, we measure the black qubits in the Y basis and perform correction operations as needed. Finally, by measuring black qubits in Step 3 and performing correction, we are left with two separated two-qubit graph states as desired.

4.3. Simulating Resource Distribution on a Superconducting Qubit Device

We now compare the entanglement swapping-based and MQNC-based approaches by simulating the resource distribution on the superconducting qubit device, IBM Q 20 Tokyo, which was done in our previous work [158][3]. Simulating the full network topology, as illustrated in the initialization step of Figure 4.3, requires the use of SWAP gates because 14 qubits are needed and the available direct connections do not match the network topology. Thus, we consider the sub-problem of 6-qubit topology instead. Namely, we chose qubits 0, 1, 5, 6, 10, and 11, as illustrated in Figure 4.4, for the experiments. With the choice of qubits, we can simulate Step 3 in Figure 4.3 without the need for the SWAP gate. The goal is then to make qubit 0 entangle with qubit 11 and qubit 1 entangle with qubit 10.

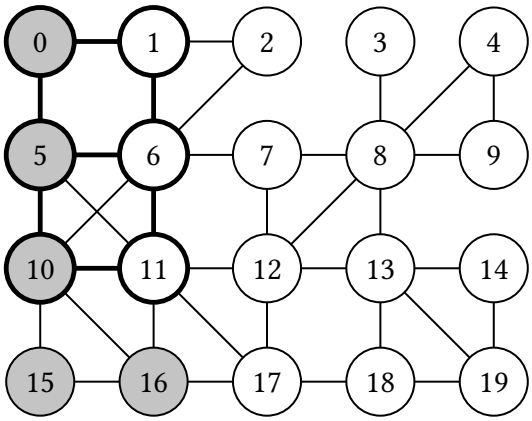


Figure 4.4: Topology of the IBM Q 20 Tokyo device used in the simulation experiments. We adapt this figure from [158].

For the entanglement swapping approach, we first create two Bell pairs, {0, 5} and {6, 11}, then perform entanglement swapping by measuring {5, 6} in the Bell basis. We then start from scratch (or at least reinitialize the qubit ({5, 6}), and do the same as before, but with {1, 6} and {5, 10}. If we perform a correction operation, the qubits {0, 11} and {1, 10} will become a Bell pair ($|\Phi^+\rangle$). However, we skip the correction operation by using a post-selection with a measurement result of "0", thereby avoiding the need to perform the correction, which may reduce the quality of the Bell pair. Finally, we performed state tomography with the post-selected Bell pair and found that the state fidelity $F(\rho, |\Phi^+\rangle) \approx 0.76$ for qubit {0, 11} and $F(\rho, |\Phi^+\rangle) \approx 0.66$ for qubit {1, 10}.

[3]We acknowledge the use of IBM Quantum services for this work. The views expressed are those of the authors, and do not reflect the official policy or position of IBM or the IBM Quantum team.

For the MQNC approach, we start from Step 3 in Figure 4.3. In this case, we must begin by creating a graph state for a 6-qubit system. First, we initialize the qubits in a $|+\rangle$ state by applying a Hadamard gate on all qubits. Second, we apply a controlled-Z gate on the required connections to create the graph's edges. Now, with the graph state, we measure qubits {5, 6} in the X-basis (by applying a Hadamard gate on the qubit before measuring in the Z-basis). Ideally, qubits {0, 11} and {1, 10} will become a two-qubit graph state ($|G_2\rangle$) with a correction operation. Similarly to the entanglement swapping approach, we perform the state-tomography with the post-selection ("1") and found that $F(\rho, |G_2\rangle) \approx 0.57$ for qubits {0, 11} and $F(\rho, |G_2\rangle) \approx 0.58$ for qubits {1, 10} (Note again that, $|G_2\rangle$ can be transform to $|\Phi^+\rangle$ by applying Hadamard gate on either qubit). Ideally, the density matrix of a two-qubit cluster state can be visualized in Figure 4.5(a). The reconstructed density matrices for both pairs are shown in Figure 4.5(c) and (d). The quality reduction observed when using MQNC is not unexpected, as MQNC requires 7 two-qubit gates to create the 6-qubit graph state, whereas the entanglement-swapping approach requires only 3 two-qubit gates throughout the protocol. Figure 4.5(b) show a reconstructed density matrix of a two-qubit cluster state between two adjacency qubits {0, 5}, which is visually closer to the ideal case.

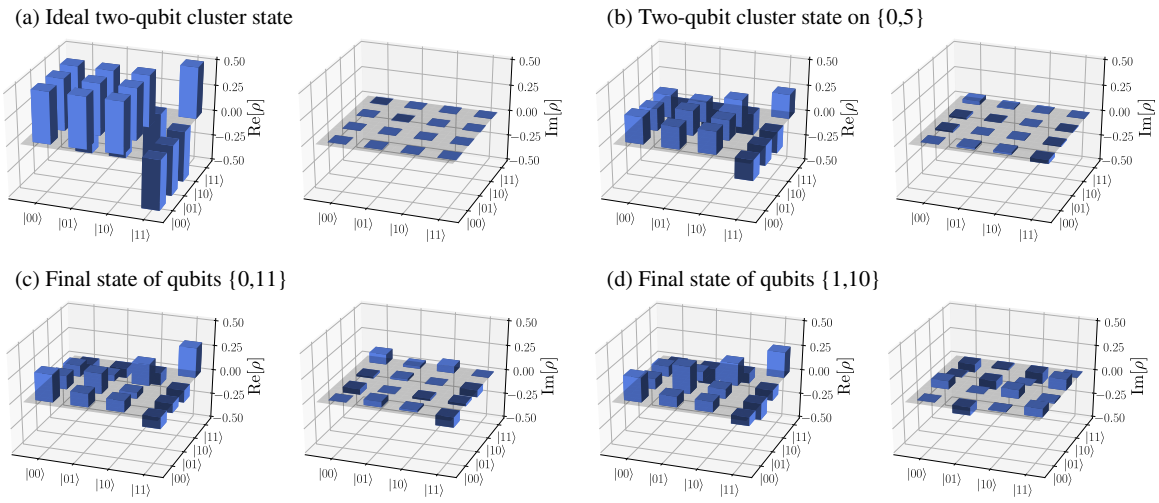


Figure 4.5: Visualization of density matrices of a two-qubit cluster state. For each panel, we plot the amplitude of each matrix element, separately for the real and imaginary parts, as a bar. This figure is reproduced from [158].

To evaluate the quality of generated entanglement, we calculate the concurrence $C(\rho)$ [159] of the quantum state ρ as follows,

$$C(\rho) = \max\{0, \lambda_1 - \lambda_2 - \lambda_3 - \lambda_4\}, \tag{119}$$

where $\{\lambda_1, \lambda_2, \lambda_3, \lambda_4\}$ are the eigenvalues of $R(\rho) = \sqrt{\sqrt{\rho}\tilde{\rho}\sqrt{\rho}}$ and $\tilde{\rho} = (Y \otimes Y)\rho^*(Y \otimes Y)$ in the decreasing order. We find that $C(\rho) = 0.253 \pm 0.018$ for qubit pair $\{0, 11\}$ and $C(\rho) = 0.312 \pm 0.021$ for qubit pair $\{1, 10\}$. Furthermore, we evaluate whether the generated two-qubit state is non-local correlated and can be used to produce certified random numbers that are secure against adversaries limited by the no-signaling principle [160] by evaluating whether the CHSH inequality is violated. The CHSH inequality is

$$S = \langle AB \rangle - \langle AB' \rangle + \langle A'B \rangle + \langle A'B' \rangle \leqslant 2, \tag{120}$$

where in our setting $A = X, A' = Z, B = H, B' = ZHZ$. We list S of each pair in the Table 4.1 where we repeat the experiment for 8 trials. From the Table 4.1, we see that $S < 2$, indicating no CHSH violation. To analyze the results, we simulate the network protocol.

	pair	S Value
Entangled pairs	$\{0, 11\}$	1.165 ± 0.04
	$\{1, 10\}$	1.235 ± 0.04
Separable pairs	$\{0, 10\}$	-0.405 ± 0.06
	$\{0, 1\}$	-0.149 ± 0.03
	$\{10, 11\}$	-0.326 ± 0.04
	$\{1, 11\}$	-0.084 ± 0.06

Table 4.1: S values of each pair using IBM Q 20 Tokyo with 8 trials. Pairs $\{0, 11\}$ and $\{1, 10\}$ show higher S values compared to the separable pairs. This table is reproduced from [158].

To simplify the analysis, we assume the following. The relaxation error, such as T_1 and T_2, is negligible as the timescale of the experiment is shorter than the relaxation time of the qubits. We assume ideal measurement. We model a single-qubit gate as a depolarizing channel $\mathcal{E}_G(\rho)$ corresponding to the quantum gate G as follows,

$$\mathcal{E}_G(\rho) = (1 - \varepsilon)G\rho G^\dagger + \varepsilon\frac{I}{2}, \tag{121}$$

where ε is an adjustable error rate in the simulation. We also model the quantum state of two-qubit cluster state $|G_2\rangle\langle G_2|$, by a rotated Werner state ρ^W [161] as follows,

$$\rho^W = \frac{4F - 1}{3}|G_2\rangle\langle G_2| + \frac{1 - F}{3}I, \tag{122}$$

where $F = F(\rho^W, |G_2\rangle)$ is a fidelity of the Werner state and the ideal two-qubit cluster state. Finally, we model the two-qubit gate error as a single-qubit depolarizing channel, as defined in Equation 121, applied independently to the control and target qubits. The black dashed line in the upper plot of Figure 4.6 shows that the S value is linearly increasing as the fidelity of ρ^W in Equation 122 increases. Using the assumptions discussed above, our simulated results also agree with the model defined in Equation 122 as the blue and orange lines align with the black dashed line. We also plot the S values from the experimental data taken from IBM Q 20 Tokyo as a green triangle and red rectangle. The experimental data closely match the simulated and theoretical model. Thus, from the model, the experimental data have an error rate $\varepsilon \approx 0.031 \pm 0.001$. We then plot S and fidelity as a function of ε in the lower plot of Figure 4.6. Our numerical results suggest that the error rate has to be less than $\varepsilon_{\text{crit}} \approx 0.012$ for the experiment to violate the CHSH inequality.

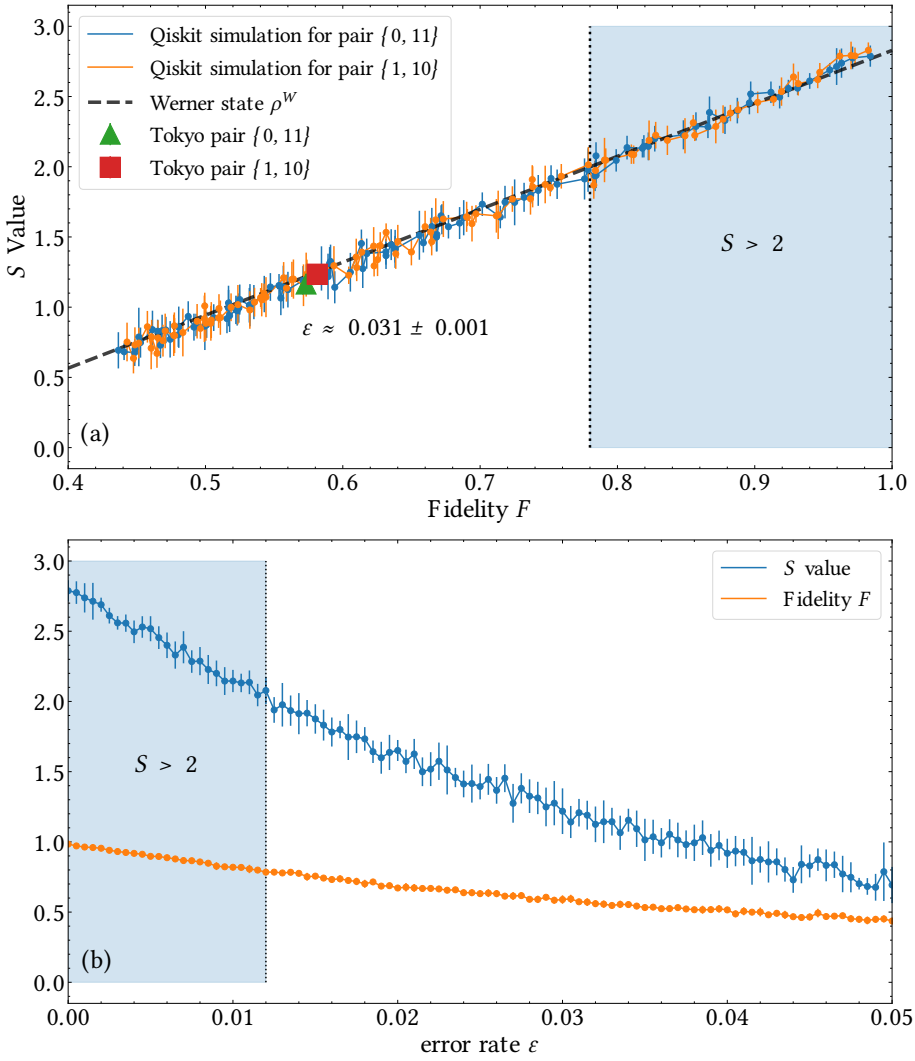


Figure 4.6: In this figure, we are interested in the relation between the error rate ε of the model that produce the two-qubit cluster state and the corresponding S value. The upper plot show the S values in terms of fidelity F. The black dashed line is calculated using Equation 122. The blue and the orange lines are simulated result by varying error rate ε. The lower plot show the S values and fidelity F in terms of error rate ε. This figure is reproduced from [158].

4.4. C&C Ways to Improve the Experiment

In a realistic network scenario, devices are likely to be maintained by an automated system. It is the system designer's responsibility to build a system that ensures high availability and observability (the ability to diagnose and monitor the system). As we observed in the results described above, we now discuss how to improve the simulation

experiment through quantum device characterization and calibration. The improvement goal aims to increase the fidelity of the produced resource (whether it is a Bell pair or a graph state). Let us build our improvement strategy based on the same general setting as in our previous result [158], and extend it to the spatially distant, multiple-processor setup later. We assume that the device has fixed (possibly known or unknown) parameters that limit its quality.

Possible sources of error include initialization, quantum gate errors, decoherence, leakage, measurement errors, and platform-specific errors. In our superconducting qubit setting, the decoherence should be negligible, since the operations should be faster than the T_1 and T_2 times. Leakage may occur when using a custom fast gate without the DRAG technique. However, in our case, it should also be safe to assume that the quantum gates used are carefully accounted for leakage errors (this should be the case for the pre-built quantum gates provided by the IBM Q system). In our discussion, we limited our sources of error to just quantum gate errors, whereas, we can mitigate measurement error by classical processing [83], [84], [85].

The single-qubit gates needed for the network operation under consideration are the Hadamard and Pauli-X gates. For the two-qubit gate, there are controlled-Z and CNOT gates. However, we can use either a controlled-Z or a CNOT gate, since we can decompose one into the other using single-qubit gates. We can improve the fidelity of the quantum gate in two ways.

1. **With access to pulse-level control:** This level of access provides the flexible, and fine-grained control over the operation [66], [94]. We can apply the Graybox characterization method, in which we can freely choose the form of the pulse envelope and the control parameters for single-qubit and two-qubit gates, to all qubits and all connections. Using these Graybox models, we can then calibrate for the quantum gates required for the operations. Furthermore, we can store the predictive model for observability purposes.
2. **With access to a parametrized quantum gate:** Although limited, we can increase the fidelity of the target quantum gate by optimizing it from a parametrized quantum gate. For example, we can optimize the angle of rotation θ of the $R_X(\theta)$ gate such that it produces $\sqrt{X}$ or the Pauli-X gate. We can also do the same for the two-qubit gate. If there are constraints on the two-qubit gate parameterization, we can use a method such as Variational Quantum Gate Optimization [82]. The method constructs the target two-qubit gate by optimizing for the rotation angle of parametrized single-qubit gates applied before and after the fixed two-qubit gate. With the parameterization ready, we can now deploy the Graybox characterization method to produce the predictive models for control calibration.

From these levels of access, we can construct predictive models for the subsystem and use them to calibrate high-fidelity quantum gates. We want to remark that it is not necessary to use the Graybox model for this purpose; for instance, one can also use closed-loop optimization methods. Nonetheless, the Graybox method is designed to capture a variety of noise in the system without an explicit noise model, which is very useful in even more realistic situations. We can also use results in Section 3.1 and Section 3.2 to (periodically) evaluate the accuracy of the Graybox model, and decide on the routine maintenance.

Extending the experiment beyond the single processor and superconducting qubit to spatially separated processors introduces additional complications. One major difference is the challenges of establishing the long-range entanglement. This is where the quantum repeater comes into play, connecting to the adjacent nodes/processors.

A quantum repeater relies on a photonic qubit to establish entanglement over a long distance. Nonetheless, once the connection is created, the same principle still applies as discussed previously in the single processor case, as the entanglement resource is mostly stored in the matter qubit (trapped-ion, superconducting qubit, and other) for an extended period of time. Note that there is an all-photonic quantum repeater architecture [162] which establishes an end-to-end connection without storing intermediate connections with matter qubits. However, recent works explored the feasibility of using both architectures together [33]. Ultimately, it depends on the application at the end nodes whether transferring entanglement to a matter qubit is required. Nevertheless, it is also possible to apply quantum control to the photonic device as done in previous works [53], [137], [138], [163].

In practice, we have to move from the basic quantum repeater architecture to a more advanced architecture. To handle noise, three generations of quantum repeater architectures have been proposed [37]. Each consecutive generation is more sophisticated than its predecessor, constrained by technological requirements. Specifically, the 1st generation (1G) uses entanglement purification to enhance the fidelity of the generated Bell pair. This technique uses a two-qubit gate and (possibly, depending on the specific purification technique) a measurement in the X basis to perform the enhanced operation. 2G uses heralded entanglement swapping as in 1G but at the logical level with quantum error correction (QEC). In particular, 3G uses an even more advanced QEC, which might require up to 100 physical qubits to encode a single logical qubit, and also assumes photon reception with high probability. Although it is also possible to hybridize these repeater generations together [164], and may improve performance in some circumstances, these strategies still rely on the same physical building blocks as in a well-separated generation. Thus, it underscores the importance of quantum control in the quantum network, as in

the computational task, we need a physical gate to perform well below the noise threshold for the QEC protocol to work properly.

As discussed above, the availability is especially important as a single "down" quantum repeater can easily become a network bottleneck. To deliver a high-availability quantum network, the calibration period must be minimized. We thus need an efficient tool for characterization and calibration tasks, which `inspeqtor` aims to be.

Chapter 5. Design and Implementation of Inspeqtor

TL;DR
In this chapter, we will discuss the design and implementation of each module provided by Inspeqtor. We will start with the structure of the library and the choice of numerical framework, then dive into each module, specifically its specifications, how it can be used with other modules, and examples.

In this chapter, the implementation details of the `inspeqtor` package will be discussed. The following are the design decisions that drive the development of `inspeqtor`.

1. Functional style programming: The function should be pure, i.e., it should have no side effects (it does not mutate non-local variables), and the same input always produces the same output. This is to avoid unintended actions triggered by the class's methods since the input might be hidden away when calling the method.
2. Platform-Agnostic: We aim to support a range of quantum device implementations. This goal also motivates the choice of a functional programming style, which offloads the responsibility for mastering and understanding classes and proper inheritance to the users, and lets them just define compatible functions and plug them in for direct use without worry.
3. Utility-based: We aim to provide utility functions for developers to use with their favorite packages.
4. First-class support for the Predictive model: With an automatic differentiation numerical framework, we can calculate the gradient of a function without having to use a specialized formulation. Thus, we design our framework to revolve around the predictive model as defined in Equation 43.

`inspeqtor` aims to be a library that provides an opinionated way to store characterization datasets and utility functions for characterizing and benchmarking the predictive model. The documentation and source code can be found in [165].

Quantum device maintenance routines differ depending on the hardware specifications, for instance, the physical realization of the qubit. Thus, there are multiple possible characterization and calibration pipelines. To design a library that provides reusable code for maintenance engineers, we consider abstracting the pipelines to enable greater flexibility in practical adaptation. Let us consider the sequence diagrams for quantum device characterization and control calibration.

Assume we want to use an open-loop approach to produce quantum gates. One simple maintenance pipeline consists of the characterization, calibration, and operation phases.

We illustrate the pipeline using a sequence diagram presented in Figure 5.1. The choice to use open-loop optimization requires characterizing the device to construct a model. For example, characterizing the device in its Hamiltonian representation, then the model is used to calibrate the control that maximizes some objective function [97], [99]. Or in the form of a (possibly differentiable) predictive model, which we can employ a gradient-based optimizer to find the optimal control [52], which is the main method we use in this dissertation. Finally, the optimized gates are deployed to the device and used in operation. Depending on the physical system, the engineer has to routinely benchmark the performance of the quantum gate since the drift in system parameters might cause the operation fidelity to drop. If the qualities drop below a threshold, characterization and calibration processes are restarted.

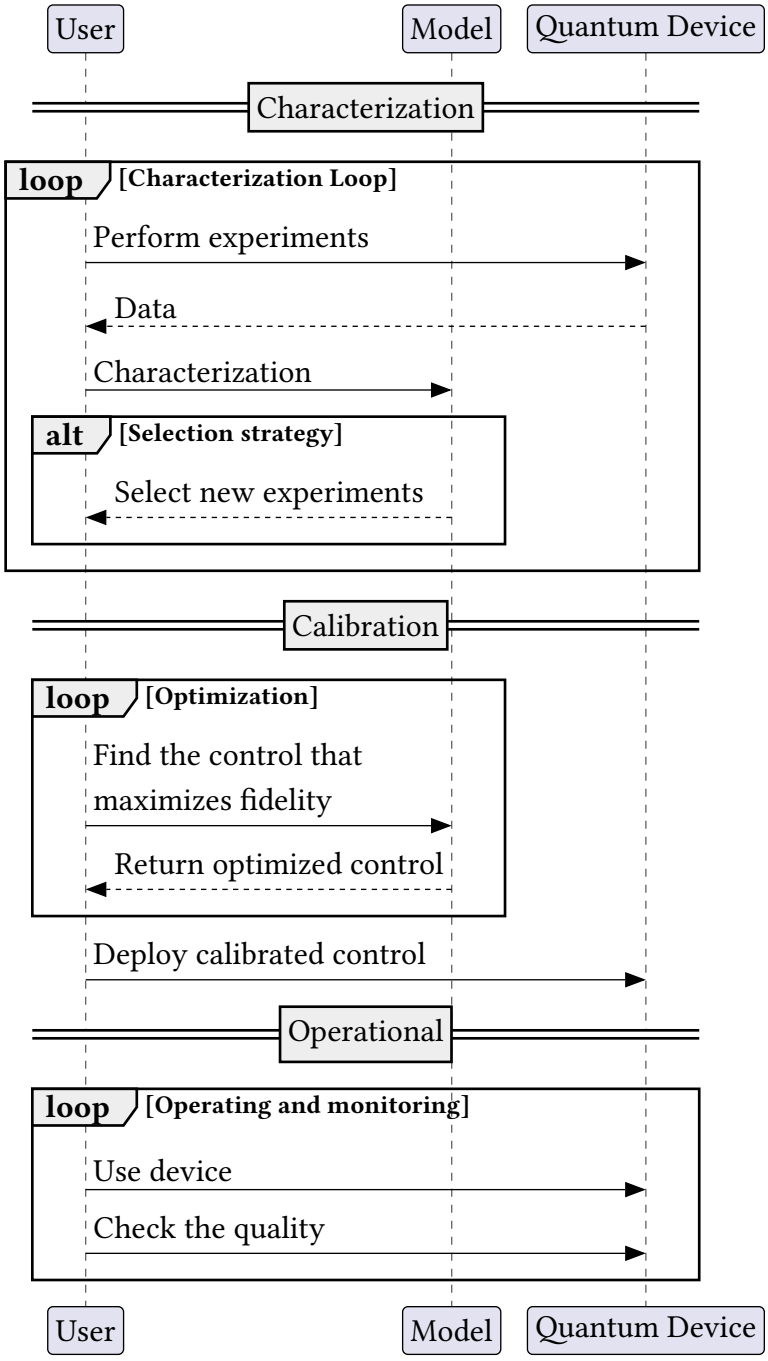


Figure 5.1: Quantum device maintenance pipeline with characterization, calibration, and operational sequence using an open-loop optimization approach.

Moreover, during characterization, data-gathering experiments may be performed in batches or in a feedback loop. As discussed in Section 3.3, it is possible to select experimental designs rather than randomly selecting to improve data utilization. Note that we do not assume a choice of predictive model.

Alternatively, device calibration using a closed-loop approach does not require extensive device characterization. The device priors necessary for the closed-loop approach are gathered and prepared, and the optimizer is deployed close to the device, as illustrated in Figure 5.2.

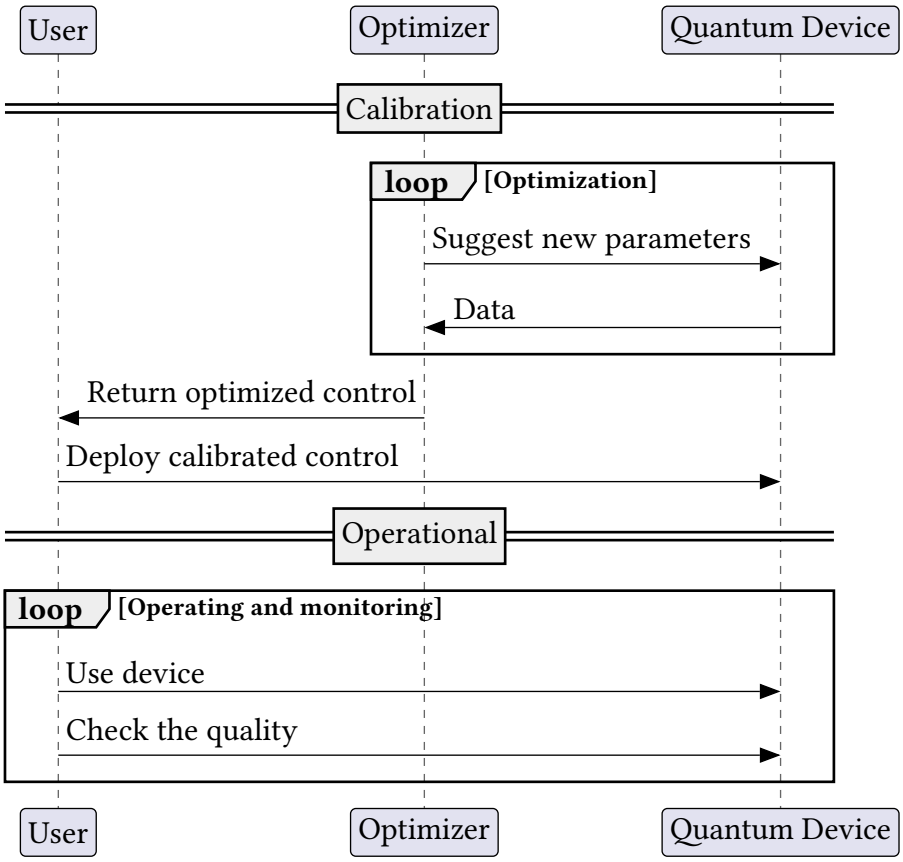


Figure 5.2: Quantum device maintenance pipeline with characterization, calibration, and operational sequence using a closed-loop optimization approach.

With the guidelines from the abstract pipeline, we can now discuss the implementation of `inspeqtor`. First, we discuss the choice of the numerical backend JAX that we used, and then discuss the essential modules, and then finish with a brief discussion of other modules.

Users can install Inspeqtor from PyPI using their preferred package manager. For example, you can install Inspeqtor with `pip` as follows:

```sh
pip install inspeqtor
```

The source code and documentation for Inspeqtor are available at https://github.com/ParametPat/inspeqtor and https://parametpat.github.io/inspeqtor/.

5.1. Structure of `inspeqtor`

By design, `inspeqtor` [4] library is a collection of modules that works together via data structures representing physical entities. To maintain a platform-agnostic design, we model the physical entities following the mathematical representation rather than object with attributes and methods. Thus, in the following sections, we will sometime discuss the design of the function by introducing the function signature base on its mathematical representation, while its actual implementation may include additional arguments which can be static. For instance, let us consider a time-dependent Hamiltonian of a system controllable by a vector of n control parameters $\mathbf{\Theta}$ denoted $\hat{H}(\mathbf{\Theta}, t)$. In our framework, we represent $\hat{H}(\mathbf{\Theta}, t)$ with function $\mathcal{H}$ with the following signature,

$$\mathcal{H}(\mathbf{\Theta} : \mathbb{R}^n \times t : \mathbb{R}) \to \text{Hermitian}(N, \mathbb{C}). \tag{123}$$

The function returns a Hermitian matrix with the appropriate dimension. In the implementation, the user may construct the function $\mathcal{H}$ using `control` . The Hamiltonian function $\mathcal{H}$ is then used to produce a solver function $\mathcal{S}$ that solves the system differential equation given an initial state $\rho_0 \in \text{Hermitian}(N, \mathbb{C})$ with the following signature,

$$\mathcal{S}\left(\mathbf{\Theta} : \mathbb{R}^n \times \rho_0 : \text{Hermitian}(N \times \mathbb{C}) \times \mathcal{H} \times \vec{t} : \mathbb{R}^{|t|}\right) \to \rho_f : \text{Hermitian}(N, \mathbb{C}) \tag{124}$$

The function $\mathcal{S}$ is given in `physics` . Then, the user can use functions in `utils` to compose a function that returns a finite-shot expectation value $\mathbb{E}\left[\hat{O}\right]$ with the following signature,

$$\mathbb{E}(\mathbf{\Theta} : \mathbb{R}^n) \to \mathbb{E}\left[\hat{O}\right] : \mathbb{R}. \tag{125}$$

We organize `inspeqtor` into multiple modules that provide functionality for composing a pipeline to characterize and calibrate the quantum device. To make it intuitive for the user, each module aims to provide functions related to its name. The platform-agnostic functions are exported under the `module` namespace, while the predefined functions for platform-specific or common use cases are exported under the `module.library` namespace. In the case of `models` , we define additional namespaces to organize functions under their respective `submodule`. We recommended use the following import signnature,

[4] We will sometimes highlight the `package` , `module` , `function` , `class` , `method` , `constant` , and `type` by color to improve the readability. We will use the non-highlight typeset to refer to external, argument, or built-in Python libraries. Sometimes, we will also write a full path to access the function or simply refer to it with just its name

```python
import inspeqtor as sq
```

User can then access submodules, for example, by using `sq.control`. We will use this import signature for the rest of the dissertation.

5.2. Why JAX

TL;DR

We chose JAX for the:

1. Automatic differentiation framework that makes it hard to make a mistake.
2. Reproducibility by providing a `key` for each stochastic function calling instead of a global seed.
3. Performance with Just-In-Time compilation.

JAX is a library for numerical computation that is similar to NumPy but also provides automatic differentiation and Just-In-Time (JIT) compilation for high-performance machine learning research [166]. The numerical framework provided by JAX allows the community to build a machine learning library around JAX easily without having to use specialized `array` and pseudo-random number Generator (PRNG) instances. JAX makes the use of a functional programming style easy and natural, because JAX handles automatic differentiation and PRNG explicitly. We will discuss them in the following.

In research, reproducibility is a must. We used JAX mainly for its PRNG. Conventionally, a Python user who uses NumPy would be familiar with setting the seed globally for reproducibility. In modern NumPy, a pseudo-random number `Generator` instance must be created and passed around whenever a random number is needed. Now, to perform machine learning, we have to choose a suitable library; the primary choices are PyTorch [167] and TensorFlow [168], both with rich, mature communities. They handle PRNG internally. Consequently, ensuring reproducibility becomes complex throughout the entire pipeline.

To generate a random number with JAX, we must provide a `key`, which JAX uses to ensure reproducibility. The `key` can be created with the following code,

```python
key = jax.random.key(0)
```

and the random number can be sampled by,

```python
sample = jax.random.uniform(key, shape=(10, ))
```

If the same key is used for the same function, the sampled random number will be the same. We can also derive a new `key` based on the old key by

```python
key, subkey = jax.random.split(key)
```

In fact, the preferred way to use the PRNG capability of JAX is to split the `key` first and use the `subkey` to generate a random number. For example, consider the following code snippet for recommended usage of PRNG:

```python
# Initialize master key
key = jax.random.key(0)
# Split the key into subkey
key, subkey = jax.random.split(key)
# Use the subkey to sample the random number
sample = jax.random.uniform(subkey, shape=(10, ))
```

This design decision allows users to define a stochastic function as a pure function.

Another alluring feature of JAX is its automatic differentiation API. `jax.grad` and `jax.value_and_grad` are functions that can be used to evaluate the gradient of a function. They have the following call signature,

```python
f = lambda x: x**2
f_prime = jax.grad(f)
grad_value = f_prime(jnp.array([0.]))
# Or with the value
value_and_grad = jax.value_and_grad(f)(jnp.array([0.]))
```

Note that these functions return a function. In our opinion, this design of the function-based API of JAX is opposed to the method-based approach of PyTorch and the need for a context manager in TensorFlow, which makes it easier for users to make mistakes. Here is how to calculate the gradient using TensorFlow, PyTorch, and JAX.

In [ ]:

```python
import tensorflow as tf

# Gradient Calculation
x = tf.Variable(3.0)
with tf.GradientTape() as tape: # Nasty context manager
    y = x**2 + 3*x + 2
grad = tape.gradient(y, x)
print("TensorFlow Gradient:", grad.numpy())
```

```

# Random Number Generation with Seed
tf.random.set_seed(42)
rand_tf = tf.random.normal(shape=(2, 3))
print("TensorFlow Random:\n", rand_tf.numpy())
```

```txt
TensorFlow Gradient: 9.0
TensorFlow Random:
 [[ 0.3274685 -0.8426258  0.3194337]
 [-1.4075519 -2.3880599 -1.0392479]]
```

In [1]:

```python
import torch

# Gradient Calculation
x = torch.tensor(3.0, requires_grad=True)
y = x**2 + 3*x + 2
y.backward() # Nasty call `backward` to calculate gradient
print("PyTorch Gradient:", x.grad.item())

# Random Number Generation with Seed
torch.manual_seed(42)
rand_torch = torch.randn(2, 3)
print("PyTorch Random:\n", rand_torch.numpy())
```

```txt
PyTorch Gradient: 9.0
PyTorch Random:
 [[ 0.33669037  0.1288094   0.23446237]
 [ 0.23033303 -1.1228564  -0.18632829]]
```

In [3]:

```python
import jax

# Gradient Calculation
def func(x):
    return x**2 + 3*x + 2
grad_func = jax.grad(func)
print("JAX Gradient:", grad_func(3.0))
```

```
8
9   # Random Number Generation with Seed
10  key = jax.random.key(42)
11  rand_jax = jax.random.normal(key, (2, 3))
12  print("JAX Random:\n", rand_jax)
```

```txt
1  JAX Gradient: 9.0
2  JAX Random:
3   [[-0.02830462  0.46713185  0.29570296]
4   [ 0.15354592 -0.12403282  0.21692315]]
```

Furthermore, JAX provides automatic vectorization, allowing the user to define a function that computes only a single sample. Then, the user can use `jax.vmap` to perform batch calculation. For example, the following snippet is possible,

```python
1  def matrix_multiplication(x, y):
2    # Assume that `x` and `y` is n by n matrix
3    assert len(x.shape) == 2 and y.shape = x.shape
4    return x @ y
5
6  batch_matrix_multiplication = jax.vmap(matrix_multiplication)
7
8  x_batch, y_batch = jnp.ones((10, 2, 2)), jnp.ones((10, 2, 2))
9  z_batch = batch_matrix_multiplication(x_batch, y_batch)
```

This design allows users to easily create functions without worrying about the batch dimension.

The benefits of the functional style of JAX will become apparent as we continue our discussion on the API design of `inspeqtor` .

5.3. Control

TL;DR

The control module assumes that the control is a function of at least time; other parameters are referred to as control parameters. Moreover, to support the different arbitrary shapes of control composing together, the user can define their atomic `Control` classes, and compose them together via a single `ControlSequence` object.

Specification

1. Represent the control function of the quantum device.
2. Be able to convert the vector of the control parameter to the vector control simulable by `physics` 's solver.
3. The user can sample the control parameter from the control instance.
4. Provide a conversion between a structured data type and an array type for readable serialization and use by machine learning frameworks.

The control of the quantum device depends on the system's physical realization. To model the possibility of control, we implement a base class, `BaseControl` , intended for developers to inherit from it in their own implementations. Then, the defined control, denoted `Control` , can be used via a unified interface, implemented via `ControlSequence` . Within the sequence, the developer can mix different types of `Control` .

As an illustrative example, in the superconducting qubit platform, a classical control field has a finite resolution of Δt, we consider a Gaussian envelope control with a max amplitude of A_m, drive strength of Ω, a standard deviation of $\sigma = \frac{\sqrt{2\pi}}{A_m \cdot 2\pi \cdot \Omega \cdot \mathrm{dt}}$, total area of θ and the amplitude of $A = \frac{\theta}{2\pi \cdot \Omega \cdot \mathrm{dt}}$. The envelope is defined as

$$h(\theta, t) = \frac{A}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(t - T/2)^2}{2\sigma^2}\right). \tag{126}$$

In this example, the only control parameter is θ. Control parameters are structured with `ParametersDictType` , simply a Python dictionary with a key as the control parameter's name corresponding to the parameter's value, which is a value with a type of `float` or `jnp.ndarray` of shape `shape = ()`. To inherit the `BaseControl` , one has to implement the `get_bounds` and `get_envelope` methods. The `get_bounds` method is a function with no input arguments that returns a tuple of lower (of `ParametersDictType` type) and upper bound (of `ParametersDictType` type) of the control parameters. This is the bound that will be called by a unified `sample_params` function to sample from a uniform distribution defined with control bounds and return a sample of control parameters with the same structure as the lower and upper bounds. Then the `get_envelope` function receives the control sample and returns the envelope function, which is a function of time. The mathematical specification of the `get_envelope` function is

$$\text{get_envelope } (\boldsymbol{\Theta} : \mathbb{R}^n) \to (t : \mathbb{R} \to h(\boldsymbol{\Theta}, t) : \mathbb{C}) \tag{127}$$

From the Gaussian envelop example, a control class may be defined as follows,

```python
@dataclass
class GaussianPulse(sq.control.BaseControl):
    center: int
    qubit_drive_strength: float
    dt: float
    max_amp: float = 1.

    theta_range: tuple[float, float] = (0.0, 2 * jnp.pi)

    def __post_init__(self):
        # This is the correction factor that will cancel the factor in the front of hamiltonian
        self.correction = 2 * jnp.pi * self.qubit_drive_strength * self.dt

        # The standard derivation of Gaussian pulse is keep fixed for the given max_amp
        self.sigma = jnp.sqrt(2 * jnp.pi) / (self.max_amp * self.correction)

    def get_bounds(
        self,
    ) -> tuple[sq.control.ParametersDictType, sq.control.ParametersDictType]:
        lower: sq.control.ParametersDictType = {"theta": self.theta_range[0]}
        upper: sq.control.ParametersDictType = {"theta": self.theta_range[1]}
        return lower, upper

    def get_envelope(
        self, params: sq.control.ParametersDictType
    ) -> typing.Callable[..., typing.Any]:
        # The area of Gaussian to be rotate to,
        area = (
            params["theta"] / self.correction
        )  # NOTE: Choice of area is arbitrary e.g. pi pulse

        return sq.control.library.gaussian_envelope(
            amp=area, center=self.center, sigma=self.sigma
```

```
34         )
```

The `ControlSequence` can be instantiated as,

```python
ctrl_seq = sq.control.ControlSequence(
    {"gaussian": GaussianPulse(center=160, qubit_drive_strength=0.1,
    dt=0.1)},
    total_dt=320,
)
```

Listing 5.1: The syntax of how to define `ControlSequence` with `GaussianPulse` envelope.

Note that the control sequence receives a dict of control classes. Internally, the total envelope is the sum of each control class, and the bound is a dict of the bounds of each control class. Now, the sample of the control sequence would look like this,

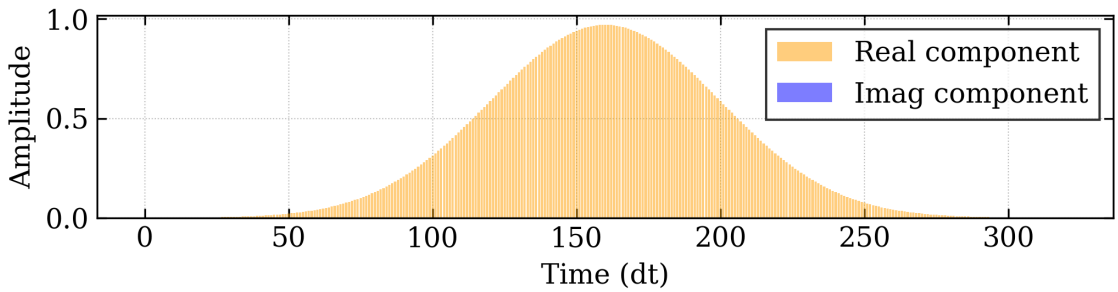
In [4]:
```python
param = ctrl_seq.sample_params(jax.random.key(0))
param
```

Out[4]:
```
{'gaussian': {'theta': Array(6.09436301, dtype=float64)}}
```

We can visualize the envelope of the control sequence as follows,

In [5]:
```python
t_eval = jnp.linspace(0, 320, 320 + 1)
waveform = jax.vmap(ctrl_seq.get_envelope(param))(t_eval)

fig, ax = plt.subplots(figsize=(7, 2))
sq.utils.plot_control_envelope(waveform, t_eval, ax)
```



Furthermore, we can also perform a batch sampling using `jax.vmap` as follows,

In [6]:
```python
params = jax.vmap(ctrl_seq.sample_params)(jax.random.split(jax.random.key(0), 10))
params
```

Out[6]:
```
{'gaussian': {'theta': Array([[0.99921654, 3.02388569, 5.67605486, 3.61496239, 3.08012169,
         3.99586552, 2.9066407 , 3.76043431, 1.56156992,
         3.46160823],      dtype=float64)}}
```

Then, the sample can be used for experimentation and model selection. The data structure of control parameters as a dict of dict is suitable for storing on disk and for readability. We will discuss how to store the control parameters later in Section 5.4. However, this structure might not be optimal for predictive model training. Thus, we also provide generic convenient function `ravel_unravel_fn` , which returns `ravel_fn` and `unravel_fn` to transform the `dict[ParametersDictType]` to `jnp.ndarray` and vice versa. Let us see the following example,

In [7]:
```python
ravel_fn, unravel_fn = sq.control.ravel_unravel_fn(ctrl_seq.get_structure())
# Check that ruravel_fn undo the ravel_fn
chex.assert_trees_all_close(jax.vmap(unravel_fn)(jax.vmap(ravel_fn)(params)), params)
# In a format that is suitable for machine learning task.
jax.vmap(ravel_fn)(params)
```

Out[7]:
```
 Array([[0.99921654],
        [3.02388569],
        [5.67605486],
        [3.61496239],
        [3.08012169],
        [3.99586552],
        [2.9066407 ],
        [3.76043431],
        [1.56156992],
        [3.46160823]], dtype=float64)
```

We decided to implement a control module based on the class object because we needed to standardize how to structure the control parameters and, consequently, how to properly save and load them on disk for later use. It also serves as a way to automatically validate the user at the time of instantiation, helping users detect potential bugs before proceeding. Below is the code snippet for saving and loading the control to and from the file system.

In [8]:

```python
import tempfile
from pathlib import Path

with tempfile.TemporaryDirectory() as temp:
    path = Path(temp)
    path.mkdir(parents=True, exist_ok=True)

    ctrl_seq.to_file(path)
    # For custom control, we have to provide the custom class
    control_reader = sq.control.construct_control_sequence_reader(
        controls=[GaussianPulse]
    )
    # Read the control sequence from path
    reread_ctrl_seq = control_reader(path)

reread_ctrl_seq
```

Out[8]:

```
ControlSequence(controls={'gaussian': GaussianPulse(center=160,
qubit_drive_strength=0.1, dt=0.1, max_amp=1.0,
theta_range=[0.0, 6.283185307179586])}, total_dt=320,
structure=[('gaussian', 'theta')])
```

5.4. Data

TL;DR
This module provides interface between in-memory and file storage systems for the characterization data. We provide a way for users to enforce a consistent data schema using type hinting and store enough data for predictive model construction.

In `inspeqtor`, we prefer to keep the data structure flat and cross-platform. Thus, we avoid using `pickle` to store the data locally and instead use JSON. The next design choice is to keep the data structure flat, i.e., avoid nested structure. So that when we read the

data from disk using Python, we only have to deal with relatively shallow nested dicts. Moreover, to prevent inconsistent key naming, we prefer using `dataclass` or `namedtuple` when possible. We are also using Python's type hint feature. They allow developers to enjoy the benefit of auto-completion (IntelliSense), resulting in a better developer experience. In this section, we will discuss the details of the data entities implemented in Inspeqtor. First, we list the specifications for the `data` .

> 📋 **Specification**
>
> 1. Store data in a cross-platform and human-readable format
> 2. Be the interface between in-memory and the file system
> 3. Enforce schema on the data and perform validation
> 4. Storing experimental data in three parts:
>   1. Experimental Configuration: the experimental settings, such as device specifications and the experiment date.
>   2. Parameters table: Table that each row is the control parameters Θ
>   3. Observed data table: The observations associated with the control parameters.

Sometimes we need to store the specifications of a quantum device, especially when a predictive model using a closed-form Hamiltonian is required. The fundamental unit of data of the quantum device is the qubit. Thus, we provide a `dataclass` for storing information about a qubit, which is `QubitInformation` . The instance can be initialized using the following code snippet:

In [9]:

```python
qubit_info = sq.data.QubitInformation(
    unit="GHz",
    qubit_idx=0,
    anharmonicity=-0.2,
    frequency=5.0,
    drive_strength=0.1,
)
qubit_info
```

Out[9]:

```
QubitInformation(unit='GHz', qubit_idx=0, anharmonicity=-0.2,
frequency=5.0, drive_strength=0.1, date='2026-06-01 01:52:29')
```

Note that the package's initial design is based on the use case of a superconducting device. However, the data class can be easily generalized. Auto-completion makes it easy to access the properties of the qubit. Furthermore, the class also implements methods to convert from and to `dict` object `qubit_info.to_dict()`.

5.4.1. Experiment Config

Initial design of the `inspeqtor` revolves around the Graybox characterization method. Consequently, we designed the data schema to support storing data for an arbitrary control sequence `ControlSequence` as columns of control parameters and their corresponding expectation values. Turns out that it is a good design choice. Since characterization often involves sending a specialized control sequence and storing the observations for post-processing. Especially, the characterization for control calibration, where only a few qubit system data have to be stored.

In [10]:

```python
order = [sq.data.ExpectationValue("0", "Z")]
config = sq.data.ExperimentConfiguration(
    qubits=[qubit_info],
    expectation_values_order=order,
    parameter_structure=ctrl_seq.get_structure(),
    backend_name="simulator",
    shots=1000,
    EXPERIMENT_IDENTIFIER="0001",
    EXPERIMENT_TAGS=["decoherence", "target characterization"],
    description="A target T1 experiment",
    device_cycle_time_ns=0.1,
    sequence_duration_dt=320,
    sample_size=201,
)

print(config)
```

```txt
============================================================
EXPERIMENT CONFIGURATION
============================================================
Identifier: 0001
Backend: simulator
Date: 2026-06-01 01:52:29
Description: A target T1 experiment
```

```
8
9   Shots: 1,000
10  Sample Size: 201
11  Device Cycle Time: 0.1000 ns
12  Sequence Duration: 320 dt
13
14  Qubits: 1
15    - QubitInformation(unit='GHz', qubit_idx=0, anharmonicity=-0.2,
      frequency=5.0, drive_strength=0.1, date='2026-06-01 01:52:29')
16
17  Expectation Values: 1
18    (States: {'0'})
19    (Observables: {'Z'})
20
21  Parameter Structure: [('gaussian', 'theta')]
22  Tags: decoherence, target characterization
23  ============================================================
```

5.4.2. Experiment Data

The central data entity is `ExperimentalData` . This object is responsible for organizing the experimental data. It provides the save and load functionality. It stores experimental data as `polars.DataFrame` [169] objects and separately stores `ExperimentConfig` to keep track of how the data is produced. It saves the data in a CSV format to the disk and loads it back to a ready-to-use format.

It requires `parameter_dataframe`, `observed_dataframe`, and `ExperimentConfig` for the instantiation. The `parameter_dataframe` and `observed_dataframe` are `polars.DataFrame` objects where each row corresponds to a `parameter_id`. The following code snippet demonstrates the intended usage,

In [11]:

```python
import polars as pl
import numpy as np
from flax.traverse_util import flatten_dict

param_df = pl.DataFrame(
    jax.tree.map(lambda x: np.array(x), flatten_dict(params,
    sep="/"))
```

```
).with_row_index("parameter_id")

# We will just random the expectation values for this example
expectation_values = jax.random.uniform(
    jax.random.key(0), shape=(10, 1), minval=-1, maxval=1
)

obs_df = pl.DataFrame(
    jax.tree.map(
        lambda x: np.array(x),
        flatten_dict(
            sq.utils.dictorization(expectation_values.T, order),
            sep="/",
        ),
    )
).with_row_index("parameter_id")

exp_data = sq.data.ExperimentalData(
    config,
    observed_dataframe=obs_df,
    parameter_dataframe=param_df,
)
exp_data.observed_dataframe.join(exp_data.parameter_dataframe,
on="parameter_id")
```

Out[11]:

txt

```
shape: (10, 3)
┌──────────────┬─────────────┬──────────────────┐
│ parameter_id ┆ 0/Z         ┆ gaussian/theta   │
│ ---          ┆ ---         ┆ ---              │
│ u32          ┆ f64         ┆ f64              │
╞══════════════╪═════════════╪══════════════════╡
│ 0            ┆ -0.163086   ┆ 0.999217         │
│ 1            ┆ -0.567409   ┆ 3.023886         │
│ 2            ┆ 0.930643    ┆ 5.676055         │
│ 3            ┆ 0.149001    ┆ 3.614962         │
│ 4            ┆ 0.064453    ┆ 3.080122         │
│ 5            ┆ -0.29019    ┆ 3.995866         │
│ 6            ┆ 0.766018    ┆ 2.906641         │
```

```
14 | 7            ┆ 0.265121  ┆ 3.760434       |
15 | 8            ┆ 0.0676    ┆ 1.56157        |
16 | 9            ┆ -0.616692 ┆ 3.461608       |
17 └──────────────┴───────────┴────────────────┘
```

Note that the user can add additional columns to the object.

Note that the characterization we are interested in applies only to a small system. Not the randomized benchmarking kind. The initial development of the package revolves around the Graybox characterization method, but the concept can be generalized. For the characterization experiment of specific model parameters, such as T_1 and T_2 time, experimental results can also be formatted and saved using `ExperimentalData` object.

User can save and load the `ExperimentalData` from and to a folder directly using `save_to_folder` and `from_folder` functions.

5.4.3. Operator, State, and Expectation Value

Throughout the characterization and calibration process, the user will often have to repeatedly access the observable $\hat{O}$ and the initial state ρ_0, both in their string and matrix representations. `ExpectationValue` is an object that standardizes the way to interact with the mathematical definition of expectation value of quantum observable $\langle \hat{O} \rangle_{\rho_0}$ in `inspeqtor` . User can access string representation via `initial_state` and `observable`.

In [12]:
```python
expval = sq.data.ExpectationValue("0+", "ZX")
expval
```

Out[12]:
```txt
ExpectationValue(initial_state='0+', observable='ZX')
```

To get the matrix representation of the initial state and observable, we use the functional approach as follows:

In [13]:
```python
sq.data.get_initial_state(expval.initial_state, dm=True)
```

Out[13]:
```txt
Array([[0.5+0.j, 0.5+0.j, 0. +0.j, 0. +0.j],
       [0.5+0.j, 0.5+0.j, 0. +0.j, 0. +0.j],
       [0. +0.j, 0. +0.j, 0. +0.j, 0. +0.j],
```

```
4         [0. +0.j, 0. +0.j, 0. +0.j, 0. +0.j]], dtype=complex128)
```

In [14]:
```
1 sq.data.get_observable_operator(expval.observable)
```
python

Out[14]:
```
1 Array([[ 0.+0.j,  1.+0.j,  0.+0.j,  0.+0.j],
2        [ 1.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
3        [ 0.+0.j,  0.+0.j, -0.+0.j, -1.+0.j],
4        [ 0.+0.j,  0.+0.j, -1.+0.j, -0.+0.j]], dtype=complex128)
```
txt

Each token in the string of the initial state and the observable represents a common eigenvector of the Pauli operators.

We expect the user to interact with the expectation values via a predefined list of `ExpectationValue`, defined in `default_expectation_values_order`. The list is a default order of the expectation values that are used throughout `inspeqtor`, since we always have to loop over the combinations of the expectation values. For a system of `n` qubits, we can generate the list of order using `get_complete_expectation_values(n)`.

5.4.4. Data flow

With the `control` and `data` ready, let us illustrate the flow of data using `inspeqtor` in more detail in a simple case. Consider the characterization pipeline in which we perform the characterization experiment using a random strategy, i.e., without an experimental design component. In Figure 5.3, we illustrate how users can use the functions and data entities provided by `inspeqtor` throughout the experiments, from start to end, with data stored in the storage.

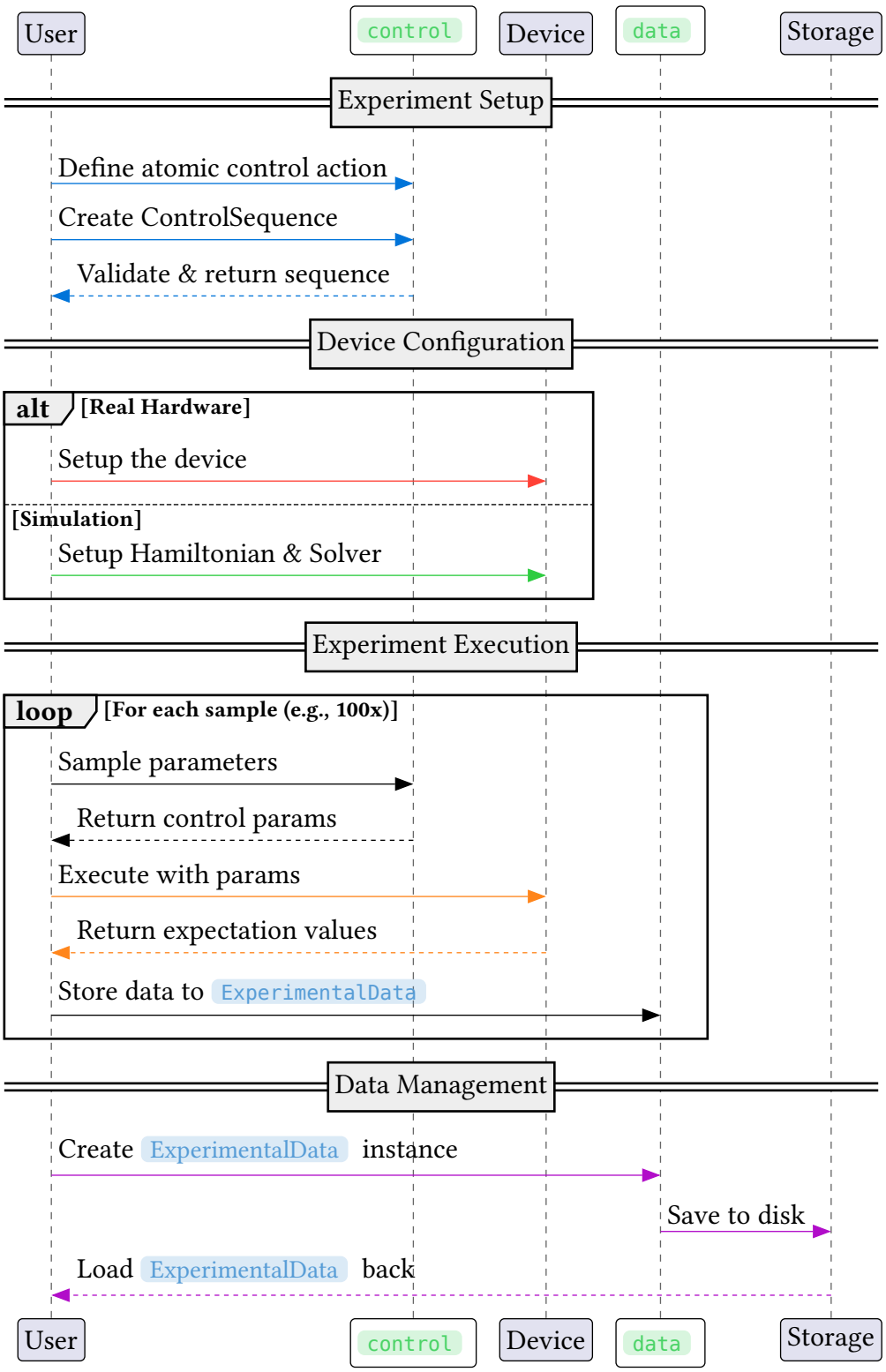


Figure 5.3: This sequence diagram shows the flow of interaction between the user, `inspeqtor`, quantum device and storage.

5.5. Physics

TL;DR

The `physics` module is a collection of intense physics-related functions. This module includes unitary solvers based on ODE and the Trotterization method. It has utility functions such as an automatic rotating transformation function given Hamiltonian and frame, and functions to calculate fidelity metrics.

The main usages of the `physics` module can be divided into two purposes. (1) The first usage is to calculate the unitary propagator given the Hamiltonian and control. The overview flow of the `physics` module is illustrated in Figure 5.4. The second usage is to calculate physical quantities regularly used in quantum information processing, such as the fidelity and the state tomography.

Specification

1. Solve the Schrödinger and Lindblad master equations given the system Hamiltonian and collapse operators.
2. Support physical related functions, e.g., AGF, process fidelity, state fidelity, and tomography.

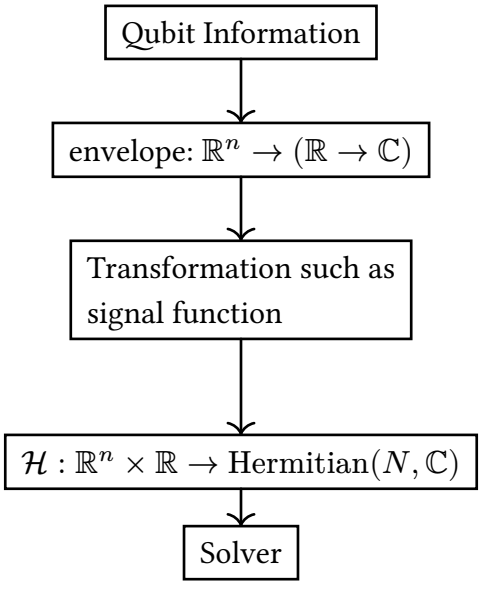


Figure 5.4: Overview of usage flow of `physics` module for unitary calculation.

The `physics` module provides unitary solver functions based on the ODE solver and the Trotterization method. The solver function $\tilde{U}(\boldsymbol{\Theta}; T)$ aims to solve for unitary $\hat{U}(\boldsymbol{\Theta}, T)$ given the control parameters $\boldsymbol{\Theta}$ at the final time T as,

$$\tilde{U}(\boldsymbol{\Theta}; T) \approx \hat{U}(\boldsymbol{\Theta}, T) = \mathcal{T}_{+} \exp\left\{-i \int_0^T \hat{H}_{\text{total}}(\boldsymbol{\Theta}, s) ds\right\}. \tag{128}$$

Usually, the final time T is fixed, so a unitary solver is used as a function of control parameters only. Most of the utility functions that provide a ready-to-use solver also return a solver as a function of control parameters only. However, as we adopt a functional programming paradigm, the solver is not necessarily only a function of control parameters. Moreover, we also provide a `lindblad_solver` which solves the Lindblad master equation as given in Equation 27.

For the ODE solver, we use `diffrax` [170], a general-purpose package for differential equation solvers. This package is part of the JAX ecosystem. The solution's accuracy can be controlled with the `rtol` and `atol` arguments. For the trotterization-based solver, we only use the first-order approximation, where the step size can be set to adjust the approximation quality. The solvers accept a Hamiltonian function with the same argument signature. The Hamiltonian function is a function of control parameters and time,

$$\mathcal{H}(\boldsymbol{\Theta} : \mathbb{R}^n \times t : \mathbb{R}) \to \text{Hermitian}(N, \mathbb{C}). \tag{129}$$

The Lindblad master equation solver also accepts the list of collapse operators $\{C_i\}_i$ which have the same dimension as the Hamiltonian $\mathcal{H}$.

We provide a common Hamiltonian of a single qubit, such as the transmon qubit model. We also provide a rotating transmon qubit model that analytically rotates the transmon qubit in the predefined frame. However, suppose the user needs to transform the Hamiltonian to another frame. In that case, we also provide an automatic transformation function, `auto_rotating_frame_hamiltonian`, that applies a unitary transformation to the given Hamiltonian using the given frame. The expected usage of the function is as follows,

```python
hamiltonian = partial(
    sq.physics.library.transmon_hamiltonian,
    qubit_info=qubit_info,
    signal=sq.physics.make_signal_fn(
        ctrl_seq.get_envelope, drive_frequency=qubit_info.frequency,
        dt=0.1
    ),
)
```

```
frame = 0.73 * sq.utils.Z
hamiltonian = sq.physics.auto_rotating_frame_hamiltonian(hamiltonian,
frame)
```

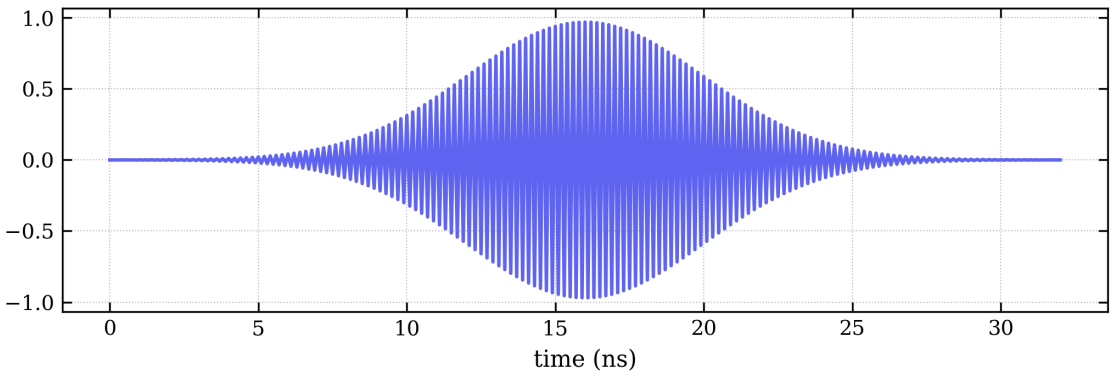
The function returns a new function whose output is transformed by the given frame, and returns it.

In the experimental realization, the envelope function returned by `ControlSequence` may not be used directly in the Hamiltonian, but may need to be transformed to a form that matches the experimental instrument. For example, in the superconducting qubit platform, the envelope function of the microwave pulse has to be transformed into a signal function and then physically send to the qubit. `inspeqtor` provides signal functions that accept different structures of control parameters. We implement a simple signal function. Below, we demonstrate how to use it using the Gaussian envelope defined previously.

In [16]:

```python
signal_fn = sq.physics.make_signal_fn(
    ctrl_seq.get_envelope, drive_frequency=qubit_info.frequency,
    dt=0.1
)

signal = jax.vmap(signal_fn, in_axes=(None, 0))(param, t_eval * 0.1)
fig, ax = plt.subplots(figsize=(7, 2.5))
ax.plot(t_eval * 0.1, signal, color=sq.utils.colors.blue)
_ = ax.set_xlabel("time (ns)")
```



The code snippet demonstrates the expected setup of the Hamiltonian, which is presented below.

In [18]:

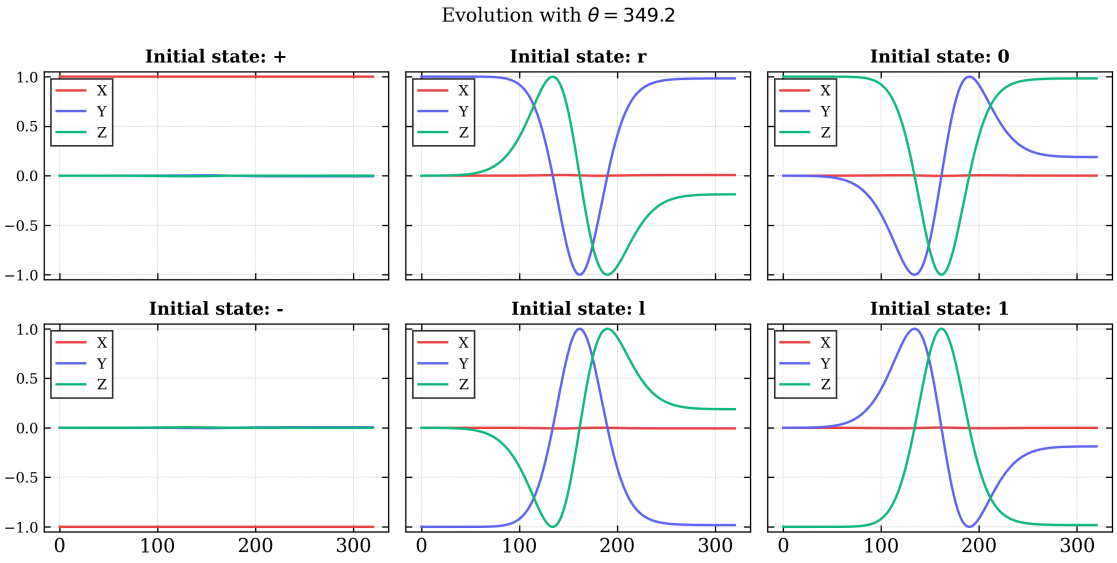
```python
hamiltonian = partial(
    sq.physics.library.rotating_transmon_hamiltonian,
    qubit_info=qubit_info,
    signal=signal_fn,
)

unitary = sq.physics.solver(
    param,
    t_eval * 0.1,
    hamiltonian,
    jnp.eye(2, dtype=jnp.complex_),
    t0=0.0,
    t1=320 * 0.1,
)

order = sq.utils.default_expectation_values_order
expvals = sq.utils.calculate_expectation_values(unitary, order)

fig, axes = sq.utils.plot_expectation_values(
    sq.utils.dictorization(expvals.T, order),
    rf"Evolution with $\theta = {jnp.rad2deg(param['gaussian']
    ['theta']).item():.1f}$",
)

# Increase x-axis font sizes
for ax in fig.axes:
    ax.set_xlabel(ax.get_xlabel(), fontsize=14)
    ax.tick_params(axis='x', labelsize=12)
```



Various functions are available in the inspeqtor for the quality assessment-related functions in quantum experiments. Currently, we support the calculation of the single-qubit case [65][5]. Here is the list of the functions and their description:

`gate_fidelity` : Calculate the fidelity between two unitary operators defined as,

$$F_{\text{gate}}\left(\hat{U},\hat{V}\right)=\frac{\text{tr}\left[\hat{U}^{\dagger}\hat{V}\right]}{\sqrt{\text{tr}\left[\hat{U}^{\dagger}\hat{U}\right]\text{tr}\left[\hat{V}^{\dagger}\hat{V}\right]}} \tag{130}$$

`process_fidelity` : Calculate the fidelity of the target superoperator using Equation 36.

`avg_gate_fidelity_from_superop` : Calculate the average gate fidelity from given superoperator as follows,

$$\bar{F}\left(\hat{U},\mathcal{E}\right)=\frac{2F_{\text{process}}\left(\hat{U},\mathcal{E}\right)+1}{3}, \tag{131}$$

where $F_{\text{process}}\left(\hat{U},\mathcal{E}\right)$ is the process fidelity defined in Equation 36

`to_superop` : Convert a given unitary operator into its superoperator representation using

$$S_{\hat{U}}=\hat{U}^{*}\otimes\hat{U}, \tag{132}$$

which is a tensor product of the operator's conjugate and itself.

[5] I would like to thank the `forest-benchmarking` [105] for initially inspiring us during the first few prototypes and for serving as a mathematical reference.

`state_tomography` : Calculate the process tomography using a simple decomposition of a quantum state of n-qubit system in terms of the expectation value of observable as

$$\rho = \frac{1}{2^n} \sum_i \langle \hat{O}_i \rangle \hat{O}_i, \tag{133}$$

where the index i iterates over the tensor product of Pauli Matrix span the system of qubit.

For the direct average gate fidelity estimation, we design the atomic functions to be composed and a function that can be used repeatedly in the control calibration routine as follows:

`direct_AFG_estimation` : This function calculates the AGF as defined in Equation 38. Its first argument is the result returned from `direct_AFG_estimation_coefficients` , which only depends on the reference unitary operator. While the second argument is the array of expectation values of observables in the same order as `default_expectation_values_order` .

`direct_AGF_estimation_fn` : For convenient usage, the user can also use this function to generate a single argument function (array of expectation values) that will calculate the AGF against the given reference unitary operator. The following is the intended usage

```python
sx_agf_fn = sq.physics.direct_AGF_estimation_fn(sq.constant.SX)
agf = sx_agf_fn(expvals) # expvals.shape == (18,)
 # agf.shape == (1,)
agfs = jax.vmap(sx_agf_fn)(batched_expvals) # expvals.shape == (100, 18,)
 # agfs.shape == (100,)
```

where the user can use `jax.vmap` with `sx_agf_fn` in a cleaner way.

5.6. Optimize

TL;DR

This module implements optimization functions. The `minimize` that returns parameters that minimize a loss function with a gradient-based optimizer with optional upper and lower bounds. The `stochastic_minimize` returns parameters that minimize a stochastic loss function with a gradient-based optimizer, with optional upper and lower bounds.

We separate the optimization baked into `inspeqtor` into two types based on their main purposes. (1) The `minimize` function is a gradient-based optimization function with optional parameter bounds. This function is mainly used for control calibration. (2) The `train_model` function is a single-line function for model training with dataset splitting into training, validating, and testing datasets. Under the hood, we use `optax` [171] as a main optimization library. The specification of `optimize` is as follows.

Specification

1. Support gradient-based optimization algorithm.
2. Support minimization of stochastic function.
3. Support bounded and unbounded optimization.
4. Support Blackbox optimization.

Let us start with the `minimize` function. We design `minimize` to be a simple function with minimal requirements for calling. Our objective is to find parameters $\vec{x}$ that minimize some scalar-output objective function $f(\vec{x})$, mathematically express as,

$$\vec{x}^* = \arg\min f(\vec{x}). \tag{134}$$

User can define their own `loss_fn` (loss function $f(\vec{x})$) which accepts only a single argument of the parameter of type `pytree`. We also did not assume an internal implementation of the `loss_fn`, aside from using JAX for numerical computation. For example, consider a simple function $f(x) = (x-4)^2$, which we wish to find x that minimizes $f(x)$. Below is a code snippet to achieve the purpose.

In [1]:
```python
import jax.numpy as jnp
import inspeqtor as sq
```

In [2]:
```python
def loss_fn(param) -> tuple[jnp.ndarray, dict]:
    """The loss function should accept only a single argument
       which is a pytree of parameters that should minimize some
       objective

    Args:
        param (Pytree): a pytree of parameters that should minimize
        some objective
```

```
7
8       Returns:
9           tuple[jnp.ndarray, dict]: Tuple of scalar output to be
            minimized and dict of arbitrary object.
10      """
11      # Objective function to be minimized.
12      objective = (param["x"] - 4) ** 2
13      aux = {"logging": objective}
14      return objective, aux
```

In [3]:

```python
# Initial guess of the parameters
init_params = {"x": jnp.array(1.0)}
# Optimized iterations
ITERATIONS = 10000
# Optax optimizer
optimizer = sq.optimize.get_default_optimizer(ITERATIONS)
# Lower and upper bound as pytree with the same structure as
init_params
lower = {"x": jnp.array(-5.0)}
upper = {"x": jnp.array(5.0)}
# Minimize the objective defined in `loss_fn`
params, history = sq.optimize.minimize(
    init_params,
    loss_fn,
    optimizer=optimizer,
    lower=lower,
    upper=upper,
    maxiter=ITERATIONS,
)
```

In [18]:

```python
params
```

Out[18]:

```txt
{'x': Array(3.99999179, dtype=float64)}
```

With this pattern, we can compose the `loss_fn` using a predictive model that produces expectation values and `direct_AGF_estimation_fn` to calculate the scalar-output function

of AGF for the minimizer to find the control parameters that minimize the function. For example, the code snippet below shows how we can leverage the utility functions provided by `inspeqtor` to quickly compose a program to calibrate quantum control.

```python
sx_agf = sq.physics.direct_AGF_estimation_fn(sq.constant.SX)
ctrl_seq = ...
predictive_fn = ... # the predictive model.

def loss_fn(params: jnp.ndarray) -> tuple[jnp.ndarray, typing.Any]:
    AGF = sx_agf(predictive_fn(params))
    # Compute the cost.
    return (1 - AGF) ** 2, {"AGF": AGF}

init_params = ctrl_seq.sample_params(jax.random.key(0))
lower, upper = ctrl_seq.get_bounds()
ravel_fn , _ = (
 sq.control.ravel_unravel_fn(ctrl_seq)
)

res = sq.optimize.minimize(
 ravel_fn(init_params),
 jax.jit(loss_fn),
 sq.optimize.get_default_optimizer(1000),
 ravel_fn(lower),
 ravel_fn(upper),
)
```

For the `stochastic_minimize` , the loss function receives two arguments. The first argument is the function parameters. The second parameter is a random key.

5.6.1. Bayesian Optimization.

We also support an optimization of a Blackbox function, i.e., non-gradient optimization via Bayesian Optimization. By default, the flow will try to maximize the function. We use `gpjax` [172] for Gaussian Process computation, and we built on the algorithms in `bayex` [173].

5.6.2. Control calibration process

Let us consider a simple open-loop control calibration pipeline using `inspeqtor`. Assume for a moment that we can construct a predictive model using `data` and `models`. In Figure 5.5, we illustrate the sequence diagram of how a user can use `optimize` in the open-loop control calibration.

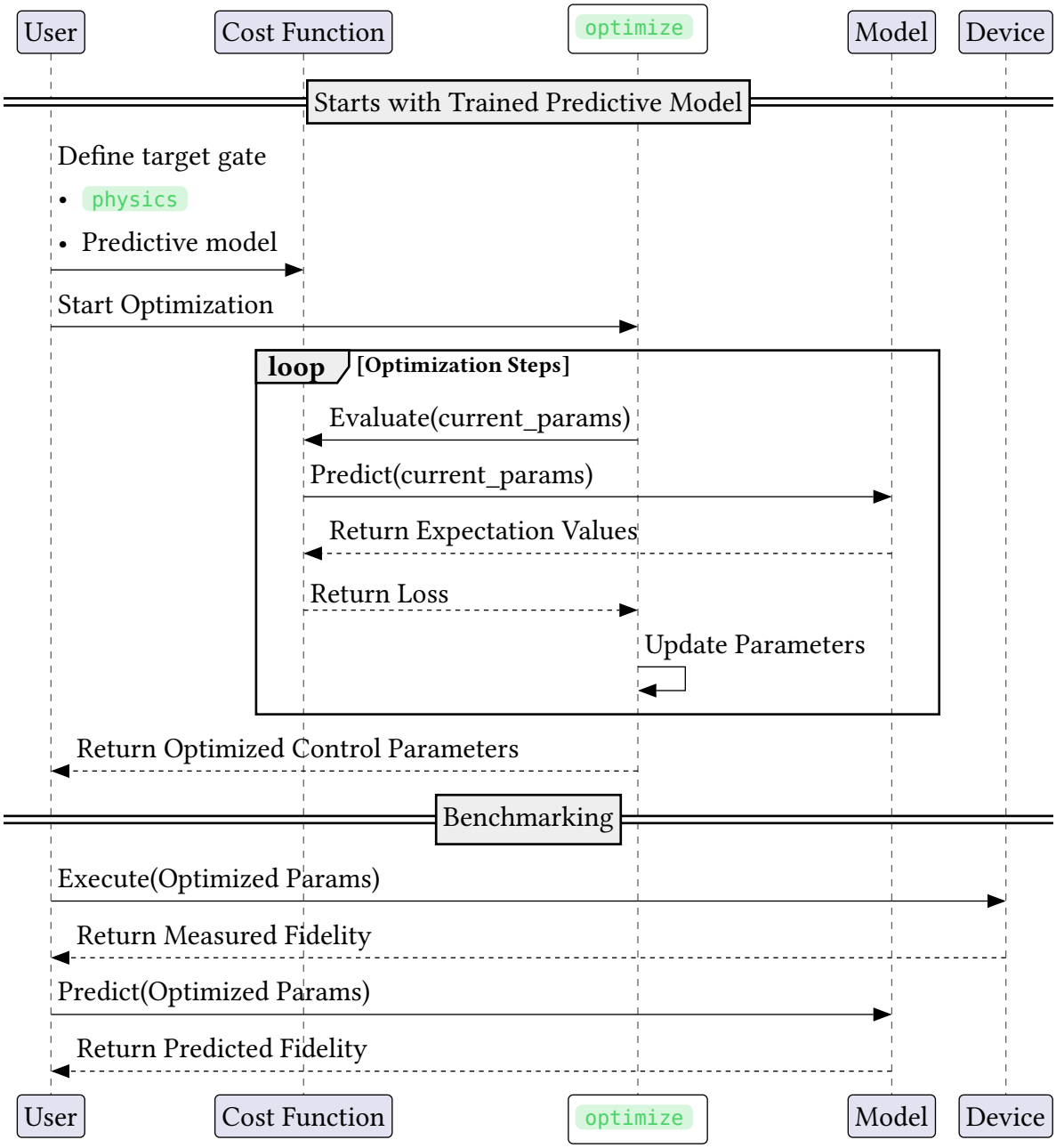


Figure 5.5: A simple open-loop approach control calibration sequence diagram using `optimize`.

5.7. Models

TL;DR
This module contains the implementation of the Blackbox model, along with the elementary functions for creating a custom model in both statistical and probabilistic paradigms. Currently, we support Blackbox model implementation using Linen and NNX of Flax out of the box. With our unified abstractions, we implement `make_loss_fn` and `create_step`, which make up for a common training loop with customizability. The interface `*_predictive_fn` for the SGM model allows for a utility function such as `make_predictive_resampling_model`, which is an SGM-based model-agnostic function. We also provide a common `ModelData` dataclass for saving and loading model parameters.

Our probabilistic sub-module provides a compatibility layer to convert a statistical model to a probabilistic model, as well as components to create a custom neural network. However, we will discuss it in a Section 5.8 as it deserves a self-contained presentation.

The `models` module provides component functions to implement the blackbox part of the Graybox characterization method and beyond. The `library` sub-module serves as an example of how to implement a custom model that is compatible with the rest of the `inspeqtor` package. We implement models in the module using Flax, a neural network package in the JAX ecosystem [174]. We support both Linen and NNX API of Flax in the sub-module with `linen` and `nnx` . In this section, we will discuss design decisions, such as the interfaces of the functions and models, rather than focusing on the actual code implementation, since it is likely to change in the future. Below are the specifications of the module that we aim to follow.

📋 **Specification**

1. Be able to construct a Predictive model in the form of Equation 43.
2. Equipped with a Learning algorithm and logging ability.
3. Be able to save and load predictive models to and from the file system in a human-readable format without having to perform learning from scratch.

Currently, we provide support to different machine learning API via a submodule. However, the names of the models and functions are essentially the same. However, the usage details may differ, but the functionality remains similar.

5.7.1. Predictive model interface

From Equation 43, mathematically, the predictive model is a function of control parameters that return expectation values. For each sub-module, we implement `make_predictive_fn` to make a predictive function signature that mirrors the Equation 43, but with the addition of unitary operators corresponding to the control parameters as an argument. Because the user can pre-calculate the unitary operators and supply them manually, the user can also create the predictive function with only control parameters as arguments by defining a new function that combines our predictive function and `solver` from elsewhere. The key argument required for `make_predictive_fn` is `adapter_fn`, which decouples the specific implementation of the machine learning model from the interface of the predictive function.

The adapter function transforms the model's output into expectation values. The adapter function is designed to be machine learning-agnostic, i.e., it is the same regardless of the machine learning library used. In Table 5.1, we listed a pair of adapter functions and the model.

Adapter function	Model
`observable_to_expvals` • Parametrized operator in Equation 57.	`WoModel`
`unitary_to_expvals` • Parametrized operator in Equation 53.	`UnitaryModel`
`toggling_unitary_to_expvals` • Parametrized operator in Equation 52. • SPAM can be handle with `ignore_spam` argument.	`UnitaryModel`
`toggling_unitary_with_spam_to_expvals` • Parametrized operator in Equation 52.	`UnitarySPAMModel`

Table 5.1: List of the `adapter_fn` currently supported in `inspeqtor` and the corresponding name of the model.

Using the adapter function, we can construct the specified predictive function. In the case where the user needs to use the statistical model as a stochastic model, we provide a transform function `sq.models.probabilistic.make_predictive_resampling_model` that

uses the statistical model's predicted expectation values as probabilities to sample from, producing a finite-shot expectation value. This function is useful when users want to create a predictive model that mimics the behavior of a real device or to quantify prediction uncertainty via a Monte Carlo method.

5.7.2. Loss function interface

With the selected `adapter_fn`, the user can prepare a predefined loss function using `make_loss_fn` . The returned loss function is compatible with `create_step` . We predefine the metric function, such as `sq.models.mse` , that implements the Mean Squared Error loss function. The metric function can then pass to `evaluate_fn` argument of the `make_loss_fn` . The user can define a custom metric function with the following signature,

$$\text{metric } \left(\langle \hat{O} \rangle^{\text{model}} : \mathbb{R}^{n \times m} \times \mathbb{E}\left[\hat{O}\right] : \mathbb{R}^{n \times m} \times U : \text{Unitary}(N, \mathbb{C})^n\right) \to \mathcal{L} : \mathbb{R}. \quad (135)$$

The first and second arguments are the predicted and experimental expectation values for m combinations of size-n samples. The third argument is the ideal unitary operators corresponding to each sample.

5.7.3. Train model

Next, we discuss the `train_model` function. At a high level, this function will train the predictive model as follows. (1) It first initializes the `flax` model and `optax` optimizer. (2) Next, it creates the the `train_step` and `eval_step` functions, which are responsible for updating model parameters based on the gradient of the loss function and evaluating the loss function without updating the model parameters using `create_step` . (3) Then, it will loop over the training dataset with a batch size equal to the size of the validation dataset and log the metrics at each step. (4) Then, at the end of the epoch (i.e., the last batch of the training dataset), it will evaluate the validating and testing datasets at once and log the metrics for both datasets, and also call each user's supplemented callback functions. (5) Finally, the function then returns the model parameters, the state of an optimizer, and the log as a list of instances of `HistoryEntryV3` . The first and second returned values depend on the submodule implementation. In general, the `train_model` has the following call signature.

```python
def train_model(
    # Random key
    key: jnp. ndarray,
    # Data
```

```
5       train_data: DataBundled,
6       val_data: DataBundled,
7       test_data: DataBundled,
8       # Model to be used for training
9       model: nn.Module,
10      optimizer: optax.GradientTransformation,
11      # Loss function to be used
12      loss_fn: typing.Callable,
13      # Callbacks to be used
14      callbacks: list[typing.Callable] = [],
15      # Number of epochs
16      NUM_EPOCH: int = 1_000,
17      # Optional state
18      **kwargs,
19  ) -> [Unknown, Unknown, list[HistoryEntryV3]]:
```

The first argument is a random key from JAX, which is useful when reproducibility is required. The same key would yield the same optimized result. The 2nd to 4th arguments are the training, validating, and testing datasets, respectively. We use the `DataBundled` instance for the dataset to provide a typesafe interface for the user to handle the dataset required for model training using this function. The `optimizer` argument is the Optax optimizer. The `**kwargs` input arguments depend on the submodule implementation. We provide a callback pattern for the user to define a custom behavior for data logging. Each callback in the list will be called after the validation and test datasets are evaluated. We expect the callback to accept the model parameters, optimization state, and history entries. User can perform a custom logic based on the current state of the model, in addition to evaluating everything in the `loss_fn`. One use is to save the model's intermediate state during training. For example, with the `linen` module, the user can then later resume the training on the latest checkpoint instead of starting from scratch by providing `model_params` and `opt_state` arguments and adjusting the `NUM_EPOCH` argument accordingly.

When the user needs to define a custom training logic, we also expose the utility functions we use to compose `train_model` , namely `sq.model.create_step` and `sq.utils.dataloader` which is a Python generator. The `create_step` creates training (update optimization state) and evaluates (just evaluate loss function) step. The `dataloader` is a Python generator that randomly shuffles the supply dataset and yields only a batch per iteration. Let us take a look at the simple code snippet of how to use `dataloader`.

```python
for (step, batch_idx, is_last_batch, epoch_idx), (
 batch_p,
 batch_u,
 batch_ex,
    ) in dataloader(
        (train_p, train_u, train_ex),
        batch_size=BATCH_SIZE,
        num_epochs=NUM_EPOCH,
        key=loader_key,
    ):
    pass
```

We assume that the first argument is a tuple of arrays. The length of the tuple can be arbitrary, and the second tuple yielded from the generator will be a tuple of the same length as the first argument, but each of the arrays will be a mini-batch instead of a full-length array. For the first tuple yielded from the generator, we provide (1) a current optimization step, the current batch index, a boolean indicating whether the current step is the last batch in the epoch, and the epoch index. Furthermore, the user can specify the batch size, the number of epochs, and the random key.

5.7.4. Serialization and Deserialization

We provide a simple dataclass `sq.models.ModelData` to save and load model parameters from disk to memory. A user can store a nested dict with keys and values of generic Python types (e.g., `dict`, `list`, `float`, `int`, and `str`) and `jnp.ndarray`. `ModelData` is compatible with a probabilistic predictive model, which will be discussed later in Section 5.8. Typically, we expect users to store the information necessary to reinstantiate the model without retraining. Currently, only model configuration and model parameters are two distinctive attributes that the `ModelData` stores. The user can also check `ModelData` equality with a simple equality check: `model_1 == model_2`, which checks both configuration and model parameters.

5.7.5. Characterization process

With the `models` ready, we illustrate how users can use our library to build predictive models. Consider a simple model training loop without the hyperparameter tuning, we show a sequence diagram for the pipeline in Figure 5.6. The pipeline can be used for hyperparameter tuning in a straightforward manner. It is up to the user to design the library for the hyperparameter tuning task, for example, Ray Tune [175], and Optuna [176].

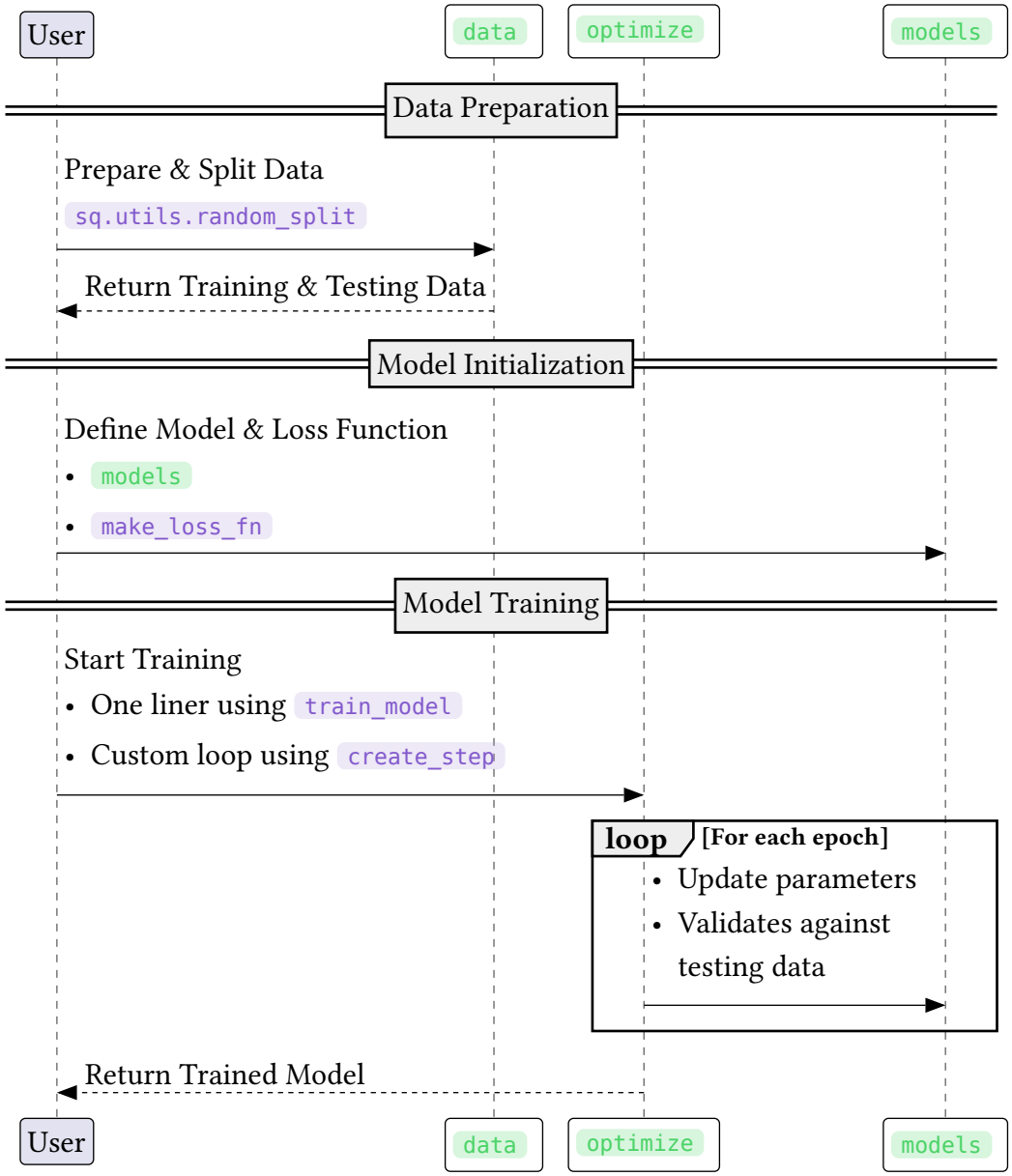


Figure 5.6: Sequence diagram for simple model training pipeline using `models` .

5.8. Probabilistic

TL;DR
Probabilistic module is a collection of probabilistic model implementations relying on the probabilistic programming NumPyro.

In this section, we will introduce the concept of a probabilistic programming. Currently, we rely on NumPyro [177], [178] for probabilistic programming. However, the

concept introduced here should be independent of the choice of library. We will focus on how to use probabilistic programming in a quantum device characterization. In particular, we will discuss the implementation of the probabilistic predictive model, which involves implementing BNN. Let us consider the rough specifications of the `probabilistic` as follows.

📋 **Specification**

1. Be able to construct a Predictive model in the form of Equation 98.
2. Equipped with an Inference algorithm and logging ability.
3. Be able to save and load predictive models to and from the file system in a human-readable format without having to perform inference from scratch.
4. Be able to set up the model with a prior distribution.

The specification above is similar to the specification of `models` with the addition of being able to set up a model with a prior distribution, and furthermore, we require a probabilistic predictive model instead of a point predictive model in Equation 43.

5.8.1. Predictive model

The posterior predictive distribution defined in Equation 98 can be viewed as a probabilistic predictive model or simply a predictive model in a probability machine learning paradigm. In our case, achieving the probabilistic predictive model means replacing parts of the predictive model with probability distributions. For example, consider a predictive model constructed solely from a parametric model of the system. We can assume a distribution for one or more system parameters. Then we can perform Bayesian inference to infer the system parameters. Alternatively, we can consider a system modelled by a neural network, as was done in the Graybox characterization method. We then promote a neural network model to the BNN.

There are multiple options to transform DNN to BNN, e.g., using functions in `numpyro.contrib.module`. In the `inspeqtor`, we provide multiple methods for defining a probabilistic predictive model using the Graybox characterization method. First, the user can convert Flax models using `make_flax_probabilistic_graybox_model` to transform compatible Flax models to the probabilistic Graybox model. Consider the code snippet below.

```python
from inspeqtor.models.probabilistic import make_flax_probabilistic_graybox_model

base_model = ... # Flax model
adapter_fn = ... # Transformation function
flax_module = ... # Function to promote DNN to BNN

graybox_model = make_flax_probabilistic_graybox_model(
    name= "graybox",
    base_model=base_model,
    adapter_fn=adapter_fn,
    prior=sq.probabilistic.dist.Normal(0, 1),
    flax_module=flax_module,
)
```

Listing 5.2: Code snippet for transforming compatible `flax` moodel to BNN.

From Listing 5.2, there are three arguments that the user has to specify appropriately, which are `base_model`, `adapter_fn`, and `flax_module`. The `flax_module` argument lets the user choose whether to transform the DNN to DNN or simply make the statistical module inferable with the SVI method. This is useful in the case where the user wants to train the statistical model by minimizing the negative ELBO instead of minimizing the MSE loss. User has to choose the appropriate `flax_module` depending on which sub-module is chosen, i.e., `linen` or `nnx` .

Alternatively, the user can define a custom BNN from scratch by using our neural network library implementation. Our implementation of `dense_layer` is the building block of the MLP model. The layer uses a specialized matrix multiplication algorithm which handle operations for vectorization of the probabilistic model using NumPyro. The ability to support vectorization by NumPyro is necessary for BOED, e.g., the algorithm proposed in [127].

To take advantage of using probabilistic machine learning, the predictive function in `probabilistic` predicts a binary measurement value by default. We separate the layer of graybox prediction and the complete predictive function because user do not has to worry about the correct usage of observable in NumPyro, making debugging easier. Because we will automatically use the expectation values predicted by the user-supplied predictive model as the probabilities for sampling from the Bernoulli distributions, as defined in Equation 94.

```python
model = sq.models.probabilistic.make_probabilistic_model(
    predictive_model=graybox_model,
 shots: int = 1,
 block_graybox: bool = False,
 separate_observables: bool = False,
 log_expectation_values: bool = False,
)
```

The user can customize the probabilistic model's behavior. For example, a user can specify the number of prediction shots of a binary measurement result, similar to what happens in the real device. This is useful when the model will be used in other probabilistic model. Furthermore, we might want to use the model within another model without training it; we can set `block_graybox` to `True` to hide the model from the optimizer.

5.8.2. Model Inferenece

As discussed in Section 3.2.2, we consider variational inference as the main method for inference. The requirements for SVI are (1) the probabilistic model which we discussed earlier, and (2) the variational distribution which we will discuss in the following.

The variational distribution is referred to as a guide in NumPyro. We also adopt the term for consistency. At the high level, the guide is a function that parameterizes the distributions in the probabilistic model. These parameters are called variational parameters. The variational distributions in the guide do not necessarily match those in the model. For instance, they can be simpler so that it is easier to optimize. Next, we sample from the variational distributions and produce the observations (samples from the target distribution). SVI is basically optimized for a set of variational parameters that produce the observations that minimize the negative ELBO.

We can automatically generate the guide corresponding to the probabilistic model using the `autoguide` module provided by NumPyro. Alternatively, for the BNN inference, we also implement our own automatic guide generation functions that accept an advanced variational parameter initialization strategy. We also implement a robust default initialization strategy that avoids numerical instability during optimization for the BNN. Moreover, our guide provides the option to "block the sample site," allowing it to be used with its model in other probabilistic models.

For the inference loop, we did not implement a single-line call function to perform the SVI, since NumPyro already provides a built-in method for this. Instead, we implement components for inferencing loop building, which extended from NumPyro to perform both update and non-update calls separately. These functions are similar to the implementation for elementary training functions in the `models` .

For both approaches to defining predictive models (from DNN and from scratch), we support setting prior distributions. Setting the prior distribution to the posterior distribution is similar to transfer learning, where parts of the statistical model's weights are initialized from a pre-trained model. However, with prior setting, instead of point-wise transfer, the distribution of the pre-trained weights is transferred.

5.8.3. Characterization process

Similarly to the sequence diagram presented in models , we illustrate how users can use our `probabilistic` to construct the posterior predictive model. Consider a simple model training loop without the hyperparameter tuning. We show a sequence diagram for the pipeline in Figure 5.7.

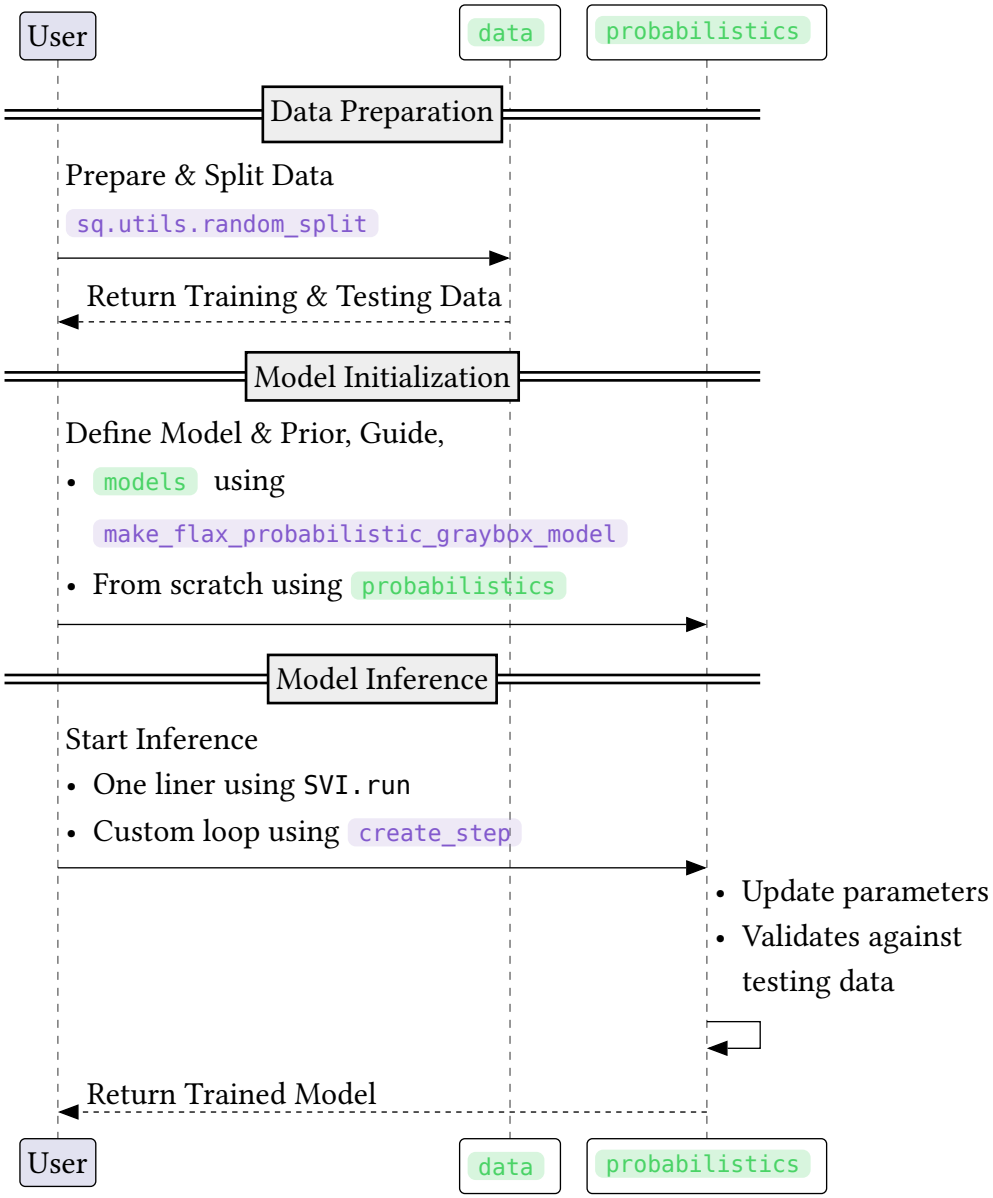


Figure 5.7: Sequence diagram for simple model training pipeline using `probabilstic` .

5.8.4. Distance measures of probability distribution

The module also provides utility functions to calculate the distance measure of the probability distribution. The KL-divergence between two probability density functions p and q we use in calculate is given by

$$D_{\mathrm{KL}}(p \parallel q) = \sum p \log\left(\frac{p}{q}\right). \tag{136}$$

Currently, we support the calculation of KL divergence from the probability mass function (PMF) `kl_divergence` . Alternatively, since the D_{KL} can be numerically unstable, we implement `safe_kl_divergence` which safely converts infinity to zero. We support the calculation from PMF since the data to be calculated are samples. Thus, the binning strategy is needed to approximate the PMF.

KL divergence is asymmetric in its argument, i.e., $D_{\mathrm{KL}}(p \parallel q) \neq D_{\mathrm{KL}}(q \parallel p)$. An alternative measure of distance for the probability distribution is the JSD. The JSD between two probability density functions p and q is given by

$$D_{\mathrm{JSD}}(p \parallel q) = \frac{1}{2}(D_{\mathrm{KL}}(p \parallel m) + D_{\mathrm{KL}}(q \parallel m)); m = \frac{p+q}{2}. \tag{137}$$

For the sake of convenience, we implement a utility function to calculate the JSD from samples of the two distributions `jensenshannon_divergence_from_sample` .

5.9. Bayesian Optimal Experimental Design

TL;DR
This module provides the capability to calculate EIG using the method in [55]. The current implementation at the time of writing this dissertation only support one-step BOED approach for data-efficient characterization.

From the mathematical point of view we discussed in Section 3.3, the concept of BOED revolves around the calculation of EIG and making a decision using EIG. We implement the variational marginal estimator of EIG proposed in [55].

Specification

1. Given experimental designs, probabilistic model, be able to estimate EIG

We implement a single function call `estimate_eig` to estimate the EIG for the given experimental designs. The function signature is presented in the code snippet below.

```python
def estimate_eig(
    key: jnp.ndarray,
    model: typing.Callable,
    marginal_guide: typing.Callable,
    design: jnp.ndarray,
    *args,
```

```
7       optimizer: optax.GradientTransformation,
8       num_optimization_steps: int,
9       observation_labels: list[str],
10      target_labels: list[str],
11      num_particles: tuple[int, int] | int,
12      final_num_particles: tuple[int, int] | int | None = None,
13      loss_fn: typing.Callable = marginal_loss,
14      callbacks: list[] = [],
15  ) -> tuple[jnp.ndarray, dict[str, typing.Any]]:
```

The user can supply the estimator in the form of a loss function `loss_fn`, passed as an argument of `estimate_eig` . Although we currently only support the variational marginal estimator, we expect the function to be usable with other estimators. The difference between our implementation and the `pyro.contrib.oed` is that we also support the supplement of the additional positional arguments `*args` to the probabilistic model `model`. In our case, we required this functionality for the unitary operators corresponding to the control parameters (`design`). The user needs to specify the observed values by using the label `observation_labels` for the estimator to calculate the EIG. The `target_labels` refer to the parameters that we would like to characterize. However, in the case of the variational marginal estimator, they did not use the target labels in EIG calculation. For logging purposes, we use callback patterns rather than returning the list of metrics for each step. Because the array can be large and consume a lot of computational memory, the user can selectively log the metrics as appropriate.

We note that estimating EIG is computationally intensive. For the computational complexity of the `estimate_eig` using the variational marginal estimator in Equation 40, we have to loop over T training steps to train the variational marginal guide. For each step, we have to evaluate the model and marginal guide for N particles. After the training, we evaluate the EIG by using M particles. Thus, the total computation complexity is $\mathcal{O}(TN + M)$. Especially when the probabilistic program involves solving a system of differential equations. The current workaround is to precompute the ideal unitary operators $\hat{U}_{\text{ideal}}(\boldsymbol{\Theta})$, given the control parameters $\boldsymbol{\Theta}$. We apply this technique to the Graybox Characterization method, in which the Blackbox component is fast to evaluate.

Alternatively, we expect to support for other optimal experimental design (OED) approaches. For example, it may be worth investigating the feasibility of approximating EIG using Bayesian D-optimality. To perform this approximation, we may need to approximate our system as a Gaussian linear model [179]. Quantum systems are obviously nonlinear; however, it may be possible to assume that the model input and output are approximately Gaussian. This assumption may hold for the distribution of the finite-shot

expectation value of a quantum observable, at least in the single-qubit case. We also expect to integrate BayesFlow [180], which offers an amortized approach to Bayesian inference and is expected to be faster than the current approach.

5.10. Utils

TL;DR
The utils module contains utility functions and classes useful for data transformations, visualization, common patterns, etc.

`inspeqtor` aims to be the opinionated characterization and calibration framework. We realize that different hardware has different requirements. Instead of dictating every aspect of device characterization and calibration within a single pipeline, we defined an abstraction of the pipeline and implemented useful functions for users to compose freely. The abstractions and pipelines we defined serve as checkpoints that ensure code implementations are reusable, reproducible, and consistent.

In the `utils` , we define several functions and classes for data transformation, visualization, and shortcuts to the common use cases. In Table 5.2, we list some of the examples of the functionality exposed via `utils` .

Goodies	Description
`default_expectation_values_order`	Provides a list of `ExpectationValue` for a single qubit system. The order of the list is what `inspeqtor` usually uses in iteration.
`finite_shot_quantum_device`	A function that returns finite-shot expectation value given number of shots.
`get_measurement_probability`	Calculate the probability of measuring each projector of tensor product of Pauli operators
`count_bits`	Count the integer representation of bits and return as a dictionary.
`prepare_data`	Prepare data from the given `ExperimentalData` , `ControlSeqeunce` , and `whitebox` , and return the instance of `LoadedData`
`plot_control_envelope`	Plot the complex waveform on the given Matplotlib axes
`plot_expectation_values`	Plot the trajectories of the given unitary operators in terms of the expectation values; currently supports only a single qubit unitary operators.

Table 5.2: Descriptions of the auxiliary modules in `inspeqtor` . Note that the details may change in the future. We encourage anyone interested in using the package to check out the document.

Chapter 6. Numerical Experiments with Inspeqtor

TL;DR
This chapter will demonstrate how Inspeqtor can help researchers characterize and calibrate a quantum device. Specifically we will review the settings and results considered in [149] and discuss them in more detail. Thus, the content in this chapter is adapted from our previous work [149]. First, we will introduce the noisy quantum device models that we will use in this chapter. Second, we will characterize the quantum device with the statistical Graybox model and show how we can naively quantify prediction uncertainty. Then, we will use the predictive models to calibrate for a target quantum gate. Next, we will use the Probabilistic Graybox model to characterize the quantum device and calibrate the quantum gate. We will go further beyond our previous work [149] by using BOED to characterize the quantum device in a data-efficient manner. We will compare and analyze the data usage of BOED -based and random strategies. The complete code used to produced the results in this chapter is available in the following URL: https://github.com/PorametPat/thesis-code/tree/main/chapter-5-strategies-v4.

6.1. Noisy Quantum Device

A quantum device is expected to be a large system of qubits. Complete characterization of the system in a classical representation is inefficient, if not impossible. For large-scale characterization, only a few system properties are of interest, and the information will be used to improve the system's performance. In the context of characterizing the device for the predictive model, we only need to consider a subsystem of the device, i.e., up to a two-qubit system. In this chapter, we will consider a single-qubit quantum device for demonstration purposes.

Let us start with the noise model of our noisy superconducting qubit, subject to stochastic colored noise and a shift along the Pauli-X axis, which we would like to characterize. We simulate the stochastic process of controlling a quantum device using the following physical model. A system consisting of a single qubit with frequency ω_q is driven by a time-dependent noisy control signal $s'(\mathbf{\Theta}, t)$ at strength Ω and driving frequency ω_d. In this example, the driving frequency of the control signal is set to $\omega_d = \omega_q$. The Hamiltonian of the qubit in the rotating frame with respect to the qubit frequency is defined as

$$\hat{H}_{\text{rot}} = 2\pi\Omega \cdot s'(\mathbf{\Theta}, t) \cdot \big(\cos\big(2\pi\omega_q t\big)\sigma_x - \sin\big(2\pi\omega_q t\big)\sigma_y\big). \tag{138}$$

In practical scenarios, multiple sources of noise affect the system's dynamics. For simplicity and interpretability in the numerical study, this case is considered where $\hat{H}_{\text{rot}}$ is perturbed by $\hat{H}_{\text{noise}} = \Delta\sigma_x$ in addition to stochastic noise, resulting in the total Hamiltonian,

$$\hat{H}_{\text{total}} = \hat{H}_{\text{rot}} + \hat{H}_{\text{noise}}. \tag{139}$$

The noise induces over-rotation in the system, which can be mitigated by applying an under-rotation control.

The stochastic noise caused by the colored noise is modeled by the noisy signal $s'(\mathbf{\Theta}, t)$. Let us start with an ideal control signal $s(\mathbf{\Theta}, t)$, which is given by

$$s(\mathbf{\Theta}, t) = \text{Re}\{h(\mathbf{\Theta}, t)e^{i(2\pi\omega_d t+\varphi)}\}. \tag{140}$$

In this context, $h(\mathbf{\Theta}, t)$ represents the control envelope, ω_d denotes the driving frequency, and φ is a controllable phase. A single control parameter, $\mathbf{\Theta} = \theta$, is used to determine the area under a Gaussian envelope with a fixed total duration of $T = 320$ dt[6], a maximum amplitude of $A_m = 0.5$ (dimensionless), a standard deviation $\sigma = \frac{\sqrt{2\pi}}{A_m 2\pi\Omega \text{ dt}}$, and an amplitude $A = \frac{\theta}{2\pi\Omega \text{ dt}}$. This results in the control envelope $h(\theta, t) = \frac{A}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(t-\frac{T}{2})^2}{2\sigma^2}\right)$, defined in the time step unit $\text{dt} = \frac{2}{9}$ ns.

[6]In the real-world scenario, given an envelope function, the device has to generate a pulse from a finite timestep. The time unit dt is a device sampling resolution time.

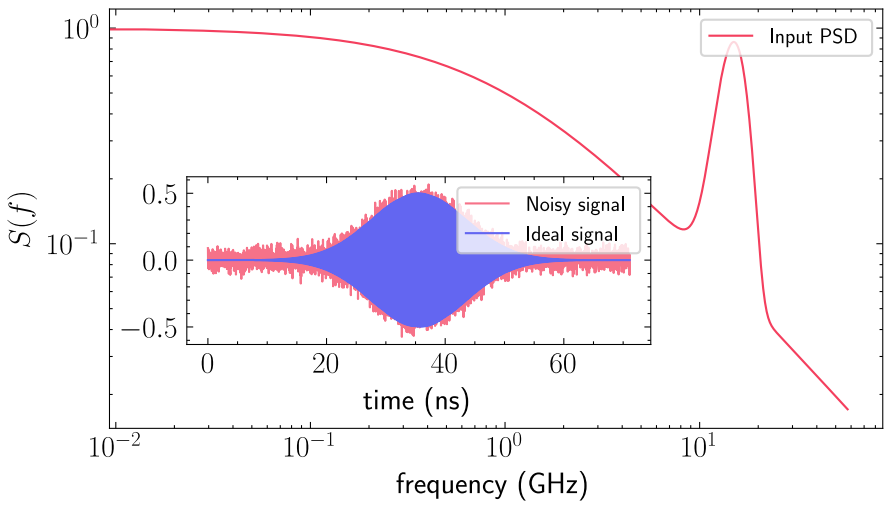


Figure 6.1: The plot shows the Power Spectrum Density (PSD) used to create colored noise with $\delta = 0.01$ as defined in Equation 141. The inset shows the ideal signal $s(2\pi, t)$ in blue and the noisy signal in red. The noisy signal is made by adding the ideal signal and the noise from the PSD in the time domain. This figure is used in [149].

A Power Spectrum Density (PSD) $S(f)$ is a function of frequency f with units of Hz and given as

$$S(f) = \frac{1}{f+1} + 0.8 \cdot \exp\left(-\frac{(f-15)^2}{10}\right), \tag{141}$$

similar to the study in [52]. To simulate noisy evolution, we sample the noise $n(t)$ from a given PSD (positive frequency only) at each time step using the method in [181]. We note that we evaluate the final time step at $T - \frac{T}{\mathrm{nt}}$, where nt is the number of Trotterization steps. We plot the PSD of Equation 141 in Figure 6.1. The total signal sent to the system is the sum of a clean signal and a noise sample $s'(\mathbf{\Theta}, t) = s(\mathbf{\Theta}, t) + \delta n(t)$ where δ controls the noise level. We show an example of a noisy signal in the inset of Figure 6.1 with $\delta = 0.01$. We note that, with the scaled noise, the estimated PSD would be a scaled version of the original PSD defined in Figure 6.1. In this work, we are interested in the regime where correlation time τ_c of the noise is much shorter than the envelope duration T. We assume that there is no correlation between consecutive experimental cycles (shots). In practice, this requires waiting longer than the correlation time between collecting two shots.

Simulating the actual process is computationally very intensive. Thus, we approximate the process by using a resampling technique. Instead of calculating each trajectory for each shot, we approximate the distribution of intermediate expectation values with

their samples, then sample with replacement from the ensemble to approximate the measurement process. We illustrate the simulation process in Figure 6.2.

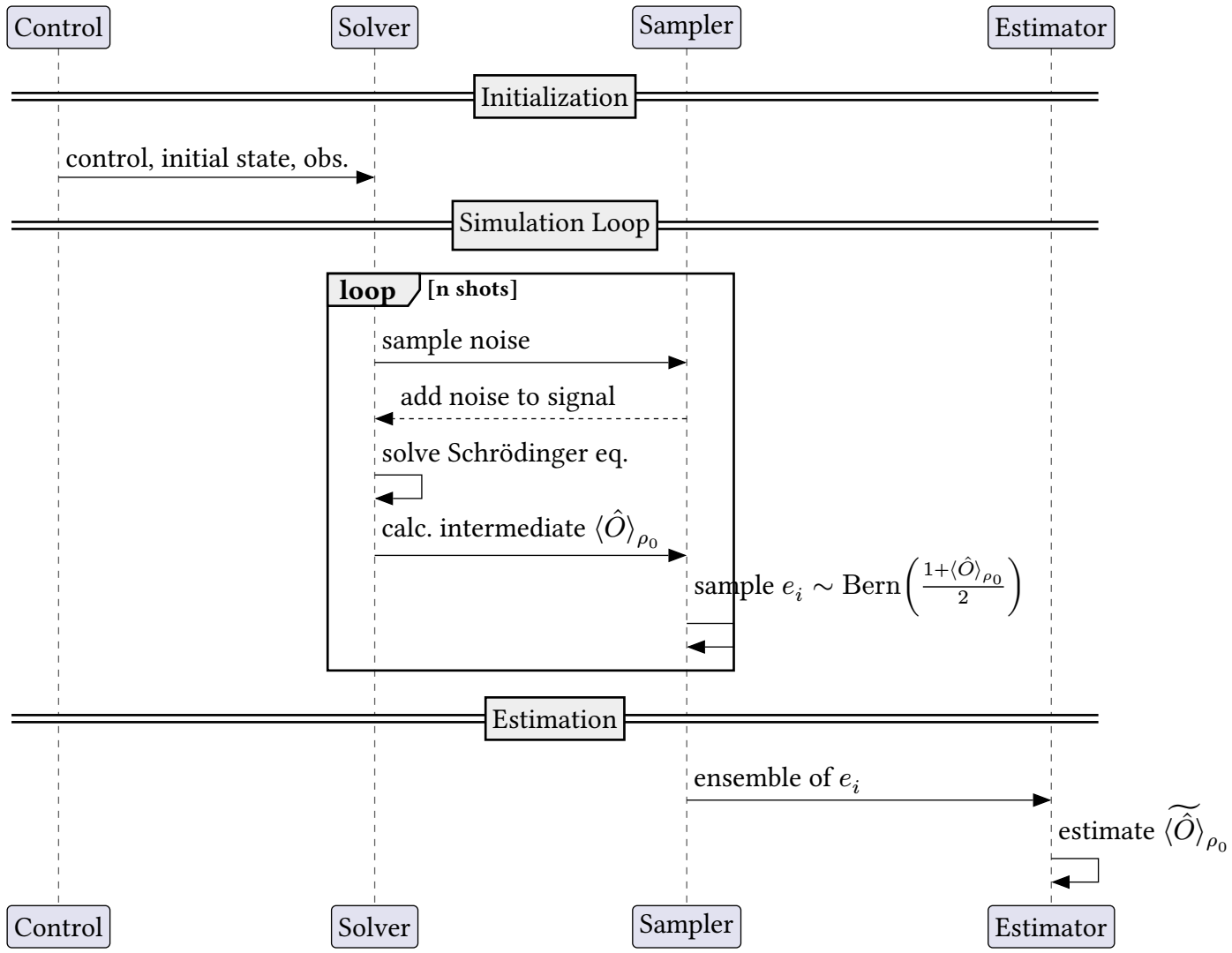


Figure 6.2: The sequence diagram illustrates the process of producing the finite-shot expectation value from power spectral density (PSD) noise. Rather than solving the ordinary differential equation (ODE) for each bit e_i, an ensemble is generated to approximate the distribution. Subsequently, resampling with replacement from this distribution is performed to estimate the finite-shot expectation value.

The dataset consists of 1,000 samples. We use $n = 1,000$ shots to estimate the finite-shot expectation value.

6.2. Statistical Approach

To begin the characterization using `inspeqtor` , we start by specifying the path to the experimental data saved using our framework.

We also specify the path where we will save the model using the following code snippet.

```python
EXPERIMENT_ID = "0002-fixed"
data_path = pathlib.Path(f"./data/PSD/{EXPERIMENT_ID}")
# Ensure that path to dataset exists
assert data_path.exists()

# # Setup model path
model_path = pathlib.Path(f"model/PSD/{EXPERIMENT_ID}/SGM")
model_path.mkdir(parents=True, exist_ok=True)
```

We then load the data, prepare it by pre-calculating the unitary operators, and split it into training and test subsets. To make it easier to pass around, we bundle them using the `DataBundled` object.

```python
loaded_data = sq.data.load_data_from_path(
    data_path,
    hamiltonian_spec=sq.physics.library.HamiltonianSpec(
        method=sq.physics.library.WhiteboxStrategy.TROTTER
        if TROTTERIZATION
        else sq.physics.library.WhiteboxStrategy.ODE,
        trotter_steps=TROTTER_STEPS,
    ),
)


key = jax.random.key(0)
key, random_split_key, train_key, prediction_key =
jax.random.split(key, 4)
(
    train_control_parameters,
    train_unitaries,
    train_expectation_values,
    test_control_parameters,
    test_unitaries,
```

```
    test_expectation_values,
) = sq.utils.random_split(
    random_split_key,
    int(loaded_data.control_parameters.shape[0] * 0.1),  # Test size
    loaded_data.control_parameters,
    loaded_data.unitaries,
    loaded_data.observed_values,
)

shots = loaded_data.experiment_data.config.shots

train_data = sq.data.DataBundled(
    control_params=ml.feature_map(train_control_parameters),
    unitaries=train_unitaries,
    observables=train_expectation_values,
)

test_data = sq.data.DataBundled(
    control_params=ml.feature_map(test_control_parameters),
    unitaries=test_unitaries,
    observables=test_expectation_values,
)
```

The model used in this study is the statistical version of the Graybox characterization method with the same architecture defined in Figure 6.7. We use the Hermitian model for this particular case. We can use the following code snippet to initialize the model.

```python
model = sq.models.library.linen.WoModel(
    shared_layers=[5],
    pauli_layers=[5],
)
```

Now, we can estimate the model parameters (i.e., train the model) using the datasets that we have prepared as follows.

In [13]:

```python
from tqdm.notebook import tqdm  # Use tqdm.notebook for a nice bar in Jupyter

NUM_EPOCH = 1000
```

```
optimizer = sq.optimize.get_default_optimizer(8 * NUM_EPOCH)

key = jax.random.key(0)
key, init_key, loader_key, train_key = jax.random.split(key, 4)

BATCH_SIZE = test_data.control_params.shape[0]

loss_fn = sq.models.library.linen.make_loss_fn(
    adapter_fn=sq.models.adapter.observable_to_expvals,
    model=model,
    evaluate_fn=lambda x, y, z: sq.models.shared.mse(x, y),
)

pbar = tqdm(total=NUM_EPOCH, desc="Training Progress")


# --- 2. Define the Callback Function to Update tqdm ---
def callback_fn(
    model_params, opt_state, history_entry:
    list[sq.models.shared.HistoryEntryV3]
):
    """
    This callback is called after each epoch. It updates the tqdm
    progress bar
    with the latest loss metrics.
    """
    # The history contains train, validation, and test entries in
    order
    train_entry = history_entry[-3]
    val_entry = history_entry[-2]
    test_entry = history_entry[-1]

    # Update the postfix of the bar to display the latest loss
    values
    pbar.set_postfix(
        {
            "Train Loss": f"{train_entry.loss:.5f}",
            "Val Loss": f"{val_entry.loss:.5f}",
            "Test Loss": f"{test_entry.loss:.5f}",
```

```
        }
    )

    # Increment the progress bar by 1 step (for one completed epoch)
    pbar.update(1)


start_time = time.time()
model_params, _, histories = sq.models.library.linen.train_model(
    train_key,
    train_data,
    test_data,
    test_data,
    model,
    optimizer,
    loss_fn,
    callbacks=[callback_fn],
    NUM_EPOCH=NUM_EPOCH,
)
stop_time = time.time()
print(stop_time - start_time)
```

```
Training Progress:   0%|          | 0/1000 [00:00<?, ?it/s]
```
txt

```
13.394061088562012
```
txt

As a demonstration, we define the callback to visualize the progress bar during the training. The user can define custom callback functions and add them to the list of callback function arguments to perform operations, e.g., logging metrics and inspecting model performance. After the model training, we can save the model to the file system as follows.

```python
model_state = sq.models.shared.ModelData(
    config={
        "shared_layers": model.shared_layers,
        "pauli_layers": model.pauli_layers,
    },
    params=model_params,
)
```

```
8
9 model_state.to_file(model_path / "model.json")
```

In the current implementation of the library, we save the configuration and the model parameters. The user is responsible for initializing the model object.

```python
model_state = sq.models.shared.ModelData.from_file(model_path / "model.json")
loaded_model = sq.models.library.linen.WoModel(**model_state.config)
```

This interface is the design decision that enables us to implement machine-learning framework-agnostic I/O operations.

We now use our model to calibrate for the $\sqrt{X}$ gate using the `optimize` module. We first define a helper function for the predictive model using the following snippet.

```python
predictive_fn = sq.models.library.linen.make_predictive_fn(
    sq.models.adapter.observable_to_expvals,
    loaded_model,
    model_state.params,
)
```

In the following code snippet, we define the cost function that computes $(1 - \mathrm{AGF})^2$ and also returns the additional information given a control parameter θ. We also define a callback function to track progress, as done during the characterization stage.

In [18]:

```python
from tqdm.notebook import tqdm  # Use tqdm.notebook for a nice bar in Jupyter

sx_agf_fn = sq.physics.direct_AGF_estimation_fn(sq.utils.SX)

maxiter = 1000
pbar = tqdm(total=maxiter, desc="Optimal Control Progress")

def callback_fn(step_idx, aux):
    # Update the postfix of the bar to display the latest loss values
    pbar.set_postfix({"AGF": aux["AGF"]})
    # Increment the progress bar by 1 step (for one completed epoch)
```

```
    pbar.update(1)

def loss_fn(params: jnp.ndarray) -> tuple[jnp.ndarray, typing.Any]:
    # Calculate the unitary using whitebox
    unitary = jax.jit(loaded_data.whitebox)(params)[-1]

    predicted_expvals = predictive_fn(ml.feature_map(params),
    unitary)

    agfs = sx_agf_fn(predicted_expvals)

    # Compute the cost.
    loss = (1 - agfs) ** 2
    return loss, {"AGF": agfs.mean(), "loss": loss}

init_params = {'gaussian': {'theta': jnp.array(jnp.pi / 2)}}
lower, upper = loaded_data.control_sequence.get_bounds()
l2a_fn, _ =
sq.control.ravel_unravel_fn(loaded_data.control_sequence.get_structu

start_time = time.time()
res = sq.optimize.minimize(
    l2a_fn(init_params),
    jax.jit(loss_fn),
    sq.optimize.get_default_optimizer(maxiter),
    l2a_fn(lower),
    l2a_fn(upper),
    maxiter=maxiter,
    callbacks=[callback_fn],
)
stop_time = time.time()
print(stop_time - start_time, "second")
```

```
Optimal Control Progress:   0%|          | 0/1000 [00:00<?, ?
it/s]
```
txt

```
103.62508273124695 second
```
txt

Finally, we get the control parameter from SGM with the following AGF,

In [19]:

```python
print("The optimized control parameters: ", res[-1][-1]['params'].item())
print("With AGF of ",res[-1][-1]['AGF'].item())
```

```txt
The optimized control parameters:  1.435742511701283
With AGF of  0.9999900253745828
```

6.3. Probabilistic Approach

Similarly to the SGM case, we use the same path to the data and define the path to save the PGM model. However, we will use a Bernoulli-based model PGM, so we need an additional step to preprocess the data as follows,

```python
train_binaries = sq.utils.eigenvalue_to_binary(
    sq.utils.expectation_value_to_eigenvalue(train_expectation_values, shots)
)
train_binaries = jnp.swapaxes(jnp.swapaxes(train_binaries, 1, 2), 0, 1)

test_binaries = sq.utils.eigenvalue_to_binary(
    sq.utils.expectation_value_to_eigenvalue(test_expectation_values, shots)
)

test_binaries = jnp.swapaxes(jnp.swapaxes(test_binaries, 1, 2), 0, 1)

assert train_binaries.shape == (shots, train_pulse_parameters.shape[0], 18)
assert test_binaries.shape == (shots, test_pulse_parameters.shape[0], 18)

train_data = sq.data.DataBundled(
    control_params=ml.feature_map(train_pulse_parameters),
    unitaries=train_unitaries,
    observables=train_binaries,
    aux=train_expectation_values,
)
```

```
21
22 test_data = sq.data.DataBundled(
23     control_params=ml.feature_map(test_pulse_parameters),
24     unitaries=test_unitaries,
25     observables=test_binaries,
26     aux=test_expectation_values,
27 )
```

In the PGM case, we use the base model with the same architecture as SGM. We use the `probabilistic` module to upgrade the statistical model to a probabilistic version using the following code snippet.

```python
hidden_sizes = [[5], [5]]

base_model = sq.models.library.linen.WoModel(
    shared_layers=hidden_sizes[0], pauli_layers=hidden_sizes[1]
)

graybox_model = sq.models.probabilistic.make_flax_probabilistic_graybox_model(
    name="graybox",
    base_model=base_model,
    adapter_fn=sq.models.probabilistic.observable_to_expvals,
    prior=dist.Normal(0, 1),
)
model = sq.models.probabilistic.make_probabilistic_model(
    predictive_model=graybox_model,
)
```

With the model ready, we now estimate the posterior distribution of PGM's model parameters using the SVI method with the following code.

In [6]:

```python
guide = sq.models.probabilistic.auto_diagonal_normal_guide(
    model, train_data.control_params, train_data.unitaries,
    train_data.observables
)

iterations = 10_000
```

```
optimizer = sq.optimize.get_default_optimizer(iterations)

svi = SVI(
    model=model,
    guide=guide,
    optim=numpyro.optim.optax_to_numpyro(optimizer),
    loss=TraceMeanField_ELBO(num_particles=1),
)

assert train_binaries.shape == (shots,
train_pulse_parameters.shape[0], 18)

svi_state = svi.init(
    rng_key=jax.random.key(0),
    control_parameters=train_data.control_params,
    unitaries=train_data.unitaries,
    observables=train_data.observables,
)

update_fn = sq.models.probabilistic.make_update_fn(
    svi,
    control_parameters=train_data.control_params,
    unitaries=train_data.unitaries,
    observables=train_data.observables,
)

eval_fn = sq.models.probabilistic.make_evaluate_fn(
    svi,
    control_parameters=test_data.control_params,
    unitaries=test_data.unitaries,
    observables=test_data.observables,
)

start_time = time.time()
eval_losses = []
losses = []
for i in alive_it(range(iterations), force_tty=True):
    svi_state, loss = jax.jit(update_fn)(svi_state)
```

```
    eval_loss = jax.jit(eval_fn)(svi_state)
    losses.append(loss)
    eval_losses.append(eval_loss)
stop_time = time.time()

svi_result = sq.models.probabilistic.SVIRunResult(
    svi.get_params(svi_state), svi_state, jnp.stack(losses),
    jnp.stack(eval_losses)
)

print(stop_time - start_time, " second")
```

```
|████████████████████████████████████████| 10000/10000 [100%] in 4:38.7 (35.88/s
278.76187801361084  second
```
txt

After model training, we recommend saving the probabilistic model using the following method.

```python
result = sq.models.shared.ModelData(
    params=svi_result.params,
    config={
        "shots": shots,
        "model": "WoBased",
        "model_config": {"hidden_sizes": hidden_sizes},
    },
)

result.to_file(PGM_model_path / "model.json")
```
python

For inference, we can load the model back from the file system as follows.

```python
shots = loaded_data.experiment_data.config.shots

characterized_result = sq.models.probabilistic.SVIResult.from_file(
    PGM_model_path / "model.json"
)

# Initialize the base model first
```
python

```
base_model = sq.models.library.linen.WoModel(
    shared_layers=characterized_result.config["model_config"]["hidden_sizes"][0],
    pauli_layers=characterized_result.config["model_config"]["hidden_sizes"][1],
)

# Create graybox model
graybox_model = sq.models.probabilistic.make_flax_probabilistic_graybox_model(
    name="graybox",
    base_model=base_model,
    adapter_fn=sq.models.probabilistic.observable_to_expvals,
    prior=dist.Normal(0, 1),
)

# Define blank model for guide construction
model = sq.models.probabilistic.make_probabilistic_model(
    predictive_model=graybox_model,
)

# initialize guide
guide = sq.models.probabilistic.auto_diagonal_normal_guide(
    model,
    ml.feature_map(loaded_data.control_parameters),
    loaded_data.unitaries,
    jnp.zeros(shape=(shots, loaded_data.control_parameters.shape[0], 18)),
)

# Construct predictive function for optimal control problem
predictive_fn = Predictive(
    model=sq.models.probabilistic.make_probabilistic_model(
        predictive_model=graybox_model,
        log_expectation_values=True,
    ),
    guide=guide,
    params=characterized_result.params,
    num_samples=1000,
```

```
43 )
```

For the probabilistic model, we will use Maximum Likelihood Estimation (MLE) instead of a gradient-based algorithm, as was done in the SGM case. The core idea is to infer the control parameter that produces binary distributions as close as possible to those produced by the target gate.

To make the MLE problem inferable with the SVI method, we define a probabilistic program as follows.

```python
def make_optimal_control_problem_v3(whitebox_fn, predictive_fn):
    def optimal_control_model(expected_outcomes: jnp.ndarray | None = None):
        control_params = numpyro.param("control_params", jnp.array([jnp.pi / 2]))
        unitary_f = whitebox_fn(control_params)[-1, :, :]

        assert isinstance(control_params, jnp.ndarray)
        assert control_params.shape == (1,)

        with numpyro.handlers.block():
            with numpyro.handlers.seed():
                key = numpyro.prng_key()
                expvals = predictive_fn(key, ml.feature_map(control_params), unitary_f)[
                    "expectation_values"
                ]

        return numpyro.sample(
            "obs",
            dist.BernoulliProbs(
                probs=sq.models.probabilistic.expectation_value_to_prob(expvals)
                # type: ignore
            ),
            obs=expected_outcomes,
        )

    return optimal_control_model

```

```
26
27  # Guide is empty since we are directly optimizing the control
    parameters in the model
28  def optimal_control_guide(expected_outcomes):
29      pass
```

We can now infer the control parameter θ using the following code snippet.

In [13]:

```python
expected_means = sq.utils.calculate_expectation_values(sq.utils.SX)

target_binaries = sq.utils.eigenvalue_to_binary(
    sq.utils.expectation_value_to_eigenvalue(expected_means, shots)
)
target_binaries = jnp.swapaxes(target_binaries, 0, 1)

optimal_control_model = make_optimal_control_problem_v3(
    whitebox_fn=loaded_data.whitebox, predictive_fn=predictive_fn
)

tx = sq.optimize.get_default_optimizer(8000)

svi = SVI(
    model=optimal_control_model,
    guide=optimal_control_guide,
    optim=numpyro.optim.optax_to_numpyro(tx),
    loss=TraceMeanField_ELBO(),
)

svi_state = svi.init(
    rng_key=jax.random.key(0),
    expected_outcomes=target_binaries,
)

update_fn = sq.models.probabilistic.make_update_fn(
    svi,
    expected_outcomes=target_binaries,
)
```

```
30
31 start_time = time.time()
32 control_losses = []
33 for i in alive_it(range(1500), force_tty=True):
34     svi_state, loss = jax.jit(update_fn)(svi_state)
35     control_losses.append(loss)
36 stop_time = time.time()
37
38 res = sq.models.probabilistic.SVIRunResult(
39     svi.get_params(svi_state),
40     svi_state,
41     jnp.stack(control_losses),
42     jnp.stack(control_losses),
43 )
44
45 print(stop_time - start_time, " second")
```

```
1 |████████████████████████████████████████| 1500/1500 [100%] in
  5:10.2 (4.83/s)                                                    txt
2 310.19844698905945  second
```

Finally, we get the control parameter from PGM with the following AGF,

In [14]:

```
1  optimal_control_params = res.params["control_params"]          python
2  sx_agf_fn = sq.physics.direct_AGF_estimation_fn(sq.utils.SX)
3  predicted_expvals = predictive_fn(
4      jax.random.key(0),
5      ml.feature_map(optimal_control_params),
6      loaded_data.whitebox(optimal_control_params)[-1],
7  )["expectation_values"]
8  agf = sx_agf_fn(predicted_expvals.mean(axis=0))
9  assert isinstance(optimal_control_params, jnp.ndarray)
10 print("The optimized control parameters: ",
   optimal_control_params.item())
11 print("With AGF of ", agf.item())
```

```
1 The optimized control parameters:  1.4287474650432057          txt
2 With AGF of  0.99994119127529
```

6.4. Statistical vs. Probabilistic

In this section, we present the characterization and calibration results obtained using SGM and PGM.

We first examine the behavior of the quantum device under both noiseless and noisy conditions. The control parameter is defined in the interval $\theta \in (0, 2\pi)$. We plot the distribution AGF of the $\sqrt{X}$ gate, calculated from finite-shot expectation values, using color variation in Figure 6.3. The distribution is visualized by plotting samples with different colors. For each control parameter, we compute the median of the samples. The absolute difference between each sample and the median determines the color assigned to that sample. Samples with lower absolute differences from the median are represented with darker colors, indicating proximity to the median. Conversely, higher values are shown with lighter colors, reflecting greater deviation from the median. Figure 6.3(a) presents the distributions generated by a noiseless device, aside from finite-shot noise. We observe that AGF concentrates at the value 1 when the control parameter is near $\theta = \frac{\pi}{2}$, as anticipated. The vertical gray line indicates the control parameter at $\theta = \frac{\pi}{2}$. In the presence of noise, as shown in Figure 6.3(b), the distributions of AGF are shifted. This specific choice of noise model and control action enables identification of the global optimal solution and its associated distribution.

We now examine the predictive performance of the constructed predictive model. We plot the distribution of AGF predicted by SGM and PGM in Figure 6.3(c) and Figure 6.3(d), respectively. Spotting visual distinction between the distributions predicted by each backend is challenging. To quantitatively compare the similarity of the two distributions, we use the Jensen-Shannon Divergence (JSD) distance [182]. JSD is selected over the KL divergence due to its numerical stability. JSD is bounded within $[0, \ln(2)]$ [182], where lower values indicate greater similarity between distributions. JSD is also symmetric in its arguments. Because the data consist of empirical samples, a binning strategy is employed for the calculation. The JSD for each control parameter is plotted in Figure 6.4, using the same interval as in Figure 6.3, with SGM shown in red and PGM in blue. The median of AGF simulated by the device is also plotted as a dashed gray line. Both predictive models exhibit poor performance near the control parameter value that yields the near-optimal AGF value. This outcome results from the choice of the $\sqrt{X}$ gate. Since the expectation values that maximize the AGF of the $\sqrt{X}$ gate are $\{-1, 0, 1\}$, predictive models face difficulty in predicting a distribution with near-zero variance. Nevertheless, PGM demonstrates substantially better performance than SGM, particularly at the optimal control parameter.

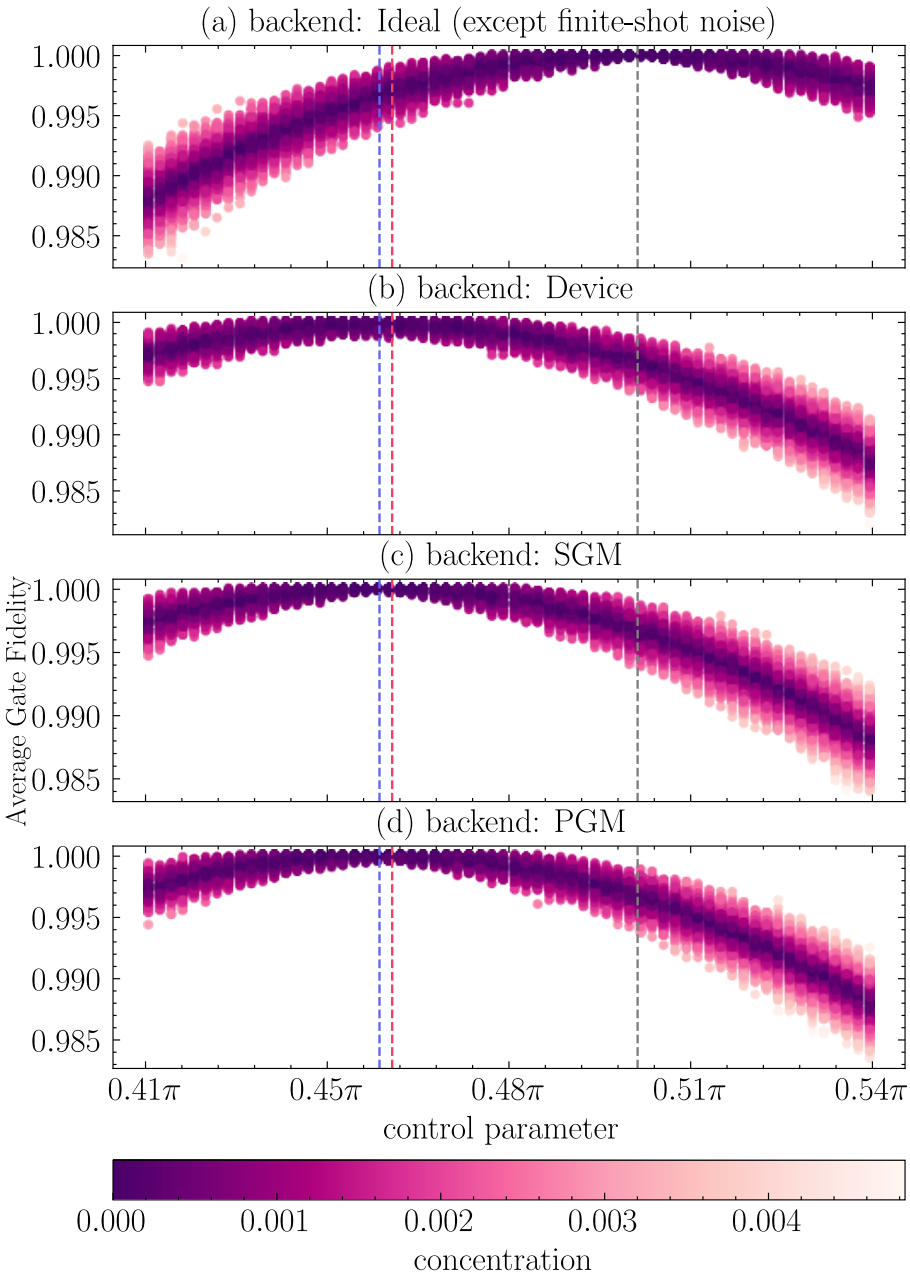


Figure 6.3: The distribution of the AGF of the $\sqrt{X}$ gate predicted by different backends. Each distribution contains 1,000 samples for a given control parameter. Each sample is assigned a concentration value equal to its absolute difference from the median and is colored accordingly. Figure (a) shows the AGF for the noiseless simulator with only finite-shot estimation noise. Figure (b) shows the AGF for the true backend (simulated), modeled with colored noise, detuning noise, and finite-shot noise. Figures (c) and (d) show the distributions predicted by SGM and PGM, respectively. The vertical dashed lines mark the optimized control parameters from SGM (red), PGM (blue), and $\theta = \frac{\pi}{2}$ (gray), the expected optimal solution without noise. We reproduce this figure in [149].

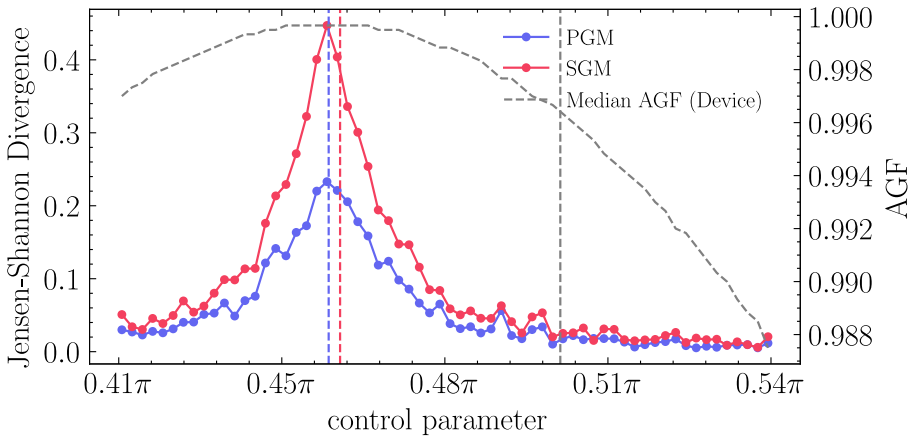
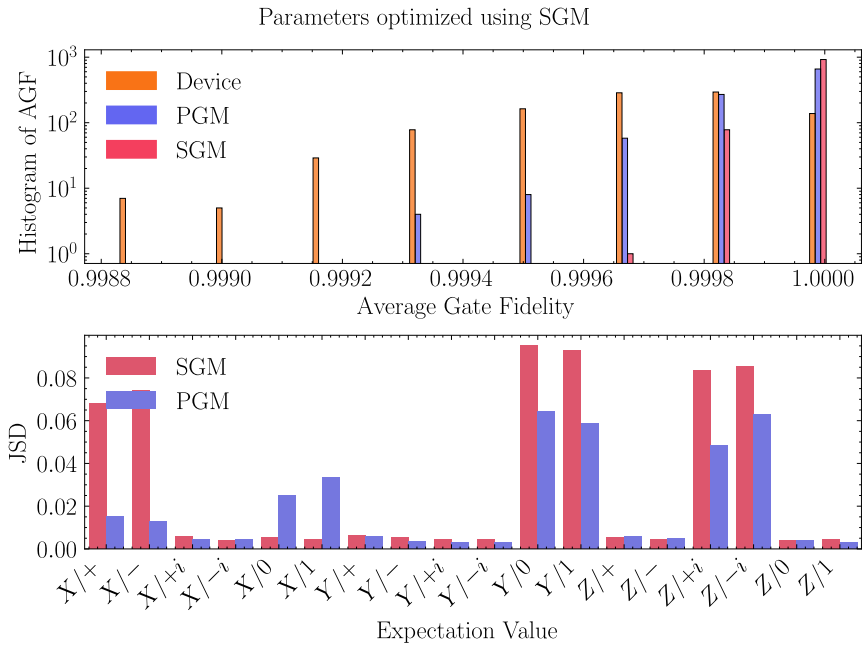
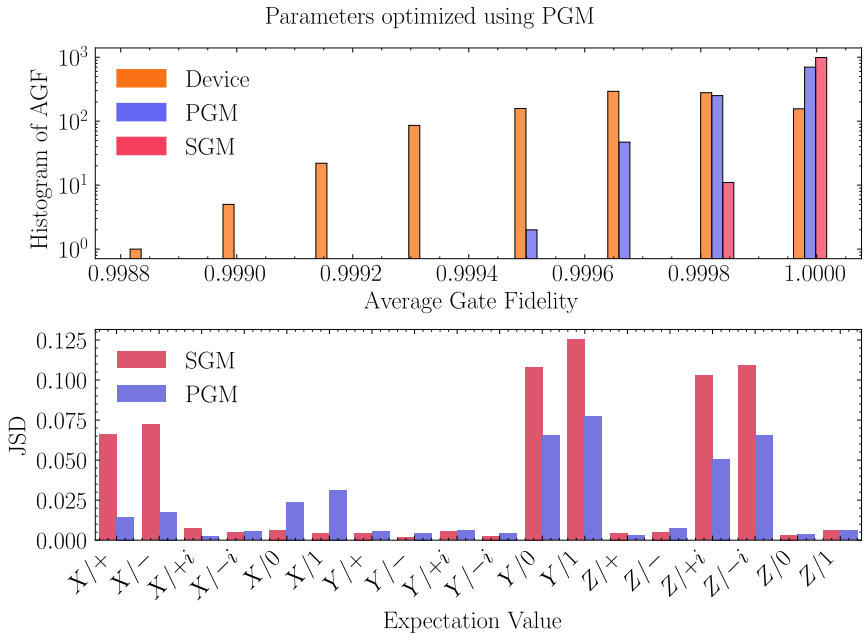


Figure 6.4: Prediction performance of SGM and PGM compare to the true distributions. On the left y-axis, we plot the JSD divergence (the lower the better) of distributions predicted by SGM (Figure 6.3.(c)) in red and PGM (Figure 6.3.(d)) in blue compare to the true distributions (Figure 6.3.(b)). On the right y-axis, we plot the median of AGF of true distribution for a reference as a gray line. We can see that the prediction performance of both models drops when the control parameters approach the optimal solution. The vertical dashed lines are (1) optimized control parameter optimized using SGM in red, (2) optimized control parameter optimized using PGM in blue, and $\theta = \frac{\pi}{2}$ in gray which is the expected optimal solution without noise. We reproduce this figure in [149].

We also plot the optimized control parameters θ^*_{SGM} and θ^*_{PGM} as dashed vertical lines in red and blue respectively in Figure 6.3 and Figure 6.4. The distributions shown in Figure 6.3 are difficult to analyze, particularly at the optimal control parameter value. We plot the distributions of the optimized control parameters in Figure 6.5. In Figure 5a, we generate samples using θ^*_{SGM} by using the device, SGM, and PGM and plot the histogram. We also plot the JSD values for the device-SGM and device-PGM pairs as red and blue bars, respectively. Similarly to Figure 5a, we use θ^*_{PGM} to plot the results in Figure 5b. We also list the relevant statistics of the control calibration in Table 6.1.



(a) θ^*_{SGM}



(b) θ^*_{PGM}

Figure 6.5: On the top panels of each sub figure are plot of the histogram of 1,000 realizations of AGF of $\sqrt{X}$ gate calculation using ground truth (target) simulator and prediction from SGM and PGM. On lower panels, we plot the bar compare the JSD between the distribution of each expectation value predict by both predictive model compare to the distribution of the device. We reproduce this figure in [149].

We can see that PGM predicts AGF's distribution closer to the device than SGM does. However, characterization and calibration using SGM are faster than inference for PGM. Although the SGM's JSD value is not as good as PGM's, the distributions follow the same trend, with important features, such as the optimal point, already close to the true one. This observation suggests that the calibration stage of SGM can be improved, for example, by training for a longer time or by adjusting the hyperparameter that controls the learning algorithm. Moreover, since this is a local replica of the qubit, we can calibrate the gate in parallel using more processors. In fact, characterization and calibration can be done in parallel if classical computational resources are available. Thus, it comes down to the developer's decision whether SGM or PGM is suitable for their use cases.

Control	Parameter	Value
	θ^*	1.435743
	$\mathbb{E}[\text{AGF}]$ by True	0.9996848
	$\mathbb{E}[\text{AGF}]$ by SGM	0.9999867
SGM	$\mathbb{E}[\text{AGF}]$ by PGM	0.9999290
	$D_{\text{JSD(SGM \mid True)}}$	0.3894
	$D_{\text{JSD(PGM \mid True)}}$	0.2120
	Ratio	**1.8366**
	θ^*	1.428747
	$\mathbb{E}[\text{AGF}]$ by True	0.9996950
	$\mathbb{E}[\text{AGF}]$ by SGM	0.9999982
PGM	$\mathbb{E}[\text{AGF}]$ by PGM	0.9999415
	$D_{\text{JSD(SGM \mid True)}}$	0.4419
	$D_{\text{JSD(PGM \mid True)}}$	0.2330
	Ratio	**1.8967**

Table 6.1: Statistic relevant to the control calibration experiments. We reproduce this table from [149].

6.5. BOED approach

Now, we are going to demonstrate the advantage of using a probabilistic model, which is the ability to use it for the calculation of EIG and use the information in device characterization. We are going to show how to use the model in sequential experiments and compare its performance to the approach without EIG.

The study setting is generally the same as the approaches discussed previously, with a modification to the data acquisition step. Previously, we uniformly sampled the control parameters to construct the dataset and experimented on the device in a single batch. However, in this study, we will perform experiments and acquire data in multiple batches sequentially. For each additional batch, we then perform characterization and benchmark the model. Next, we run the subsequent experiments on a new batch of data using the strategy (policy). The steps are repeatedly performed until some conditions are met. The strategies we are interested in are the following.

1. **Random strategy**: This is a strategy that does not utilize EIG for experimental design at all and chooses the experimental design randomly. This strategy is equivalent to the approach discussed in Section 6.3. We illustrate the strategy's flow in Figure 6.6.

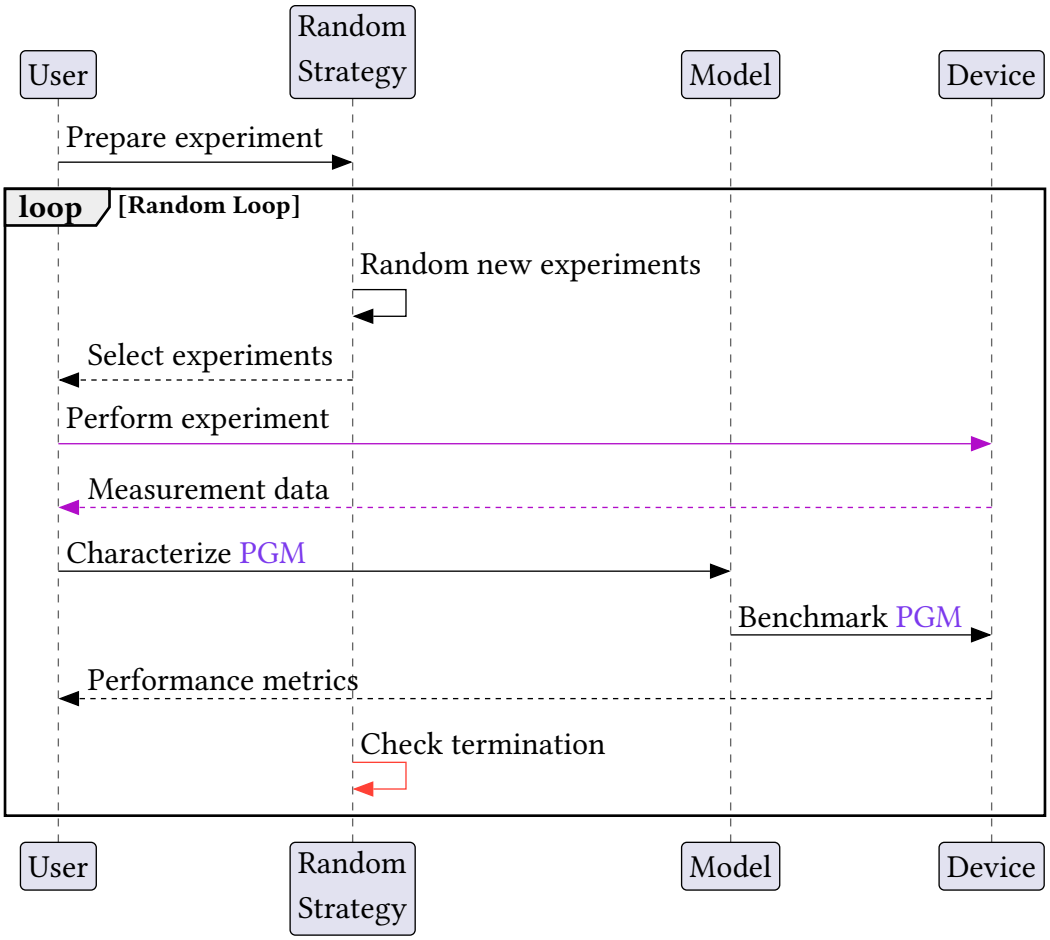


Figure 6.6: The sequence diagram of the random strategy.

2. **Subspace strategy**: This strategy divides the control parameters space into multiple subspaces and then selects the designs that maximize EIG within the subspace. Note that this strategy represents one of the BOED's approaches. Since we are performing inference for a regression predictive model (BNN), we want our model to generalize to unseen data without being overconfident. Thus, instead of choosing designs that maximize EIG globally, which would bias our dataset heavily, we chose designs that maximize EIG within their own subspace. We illustrate the strategy's flow in Figure 6.7.

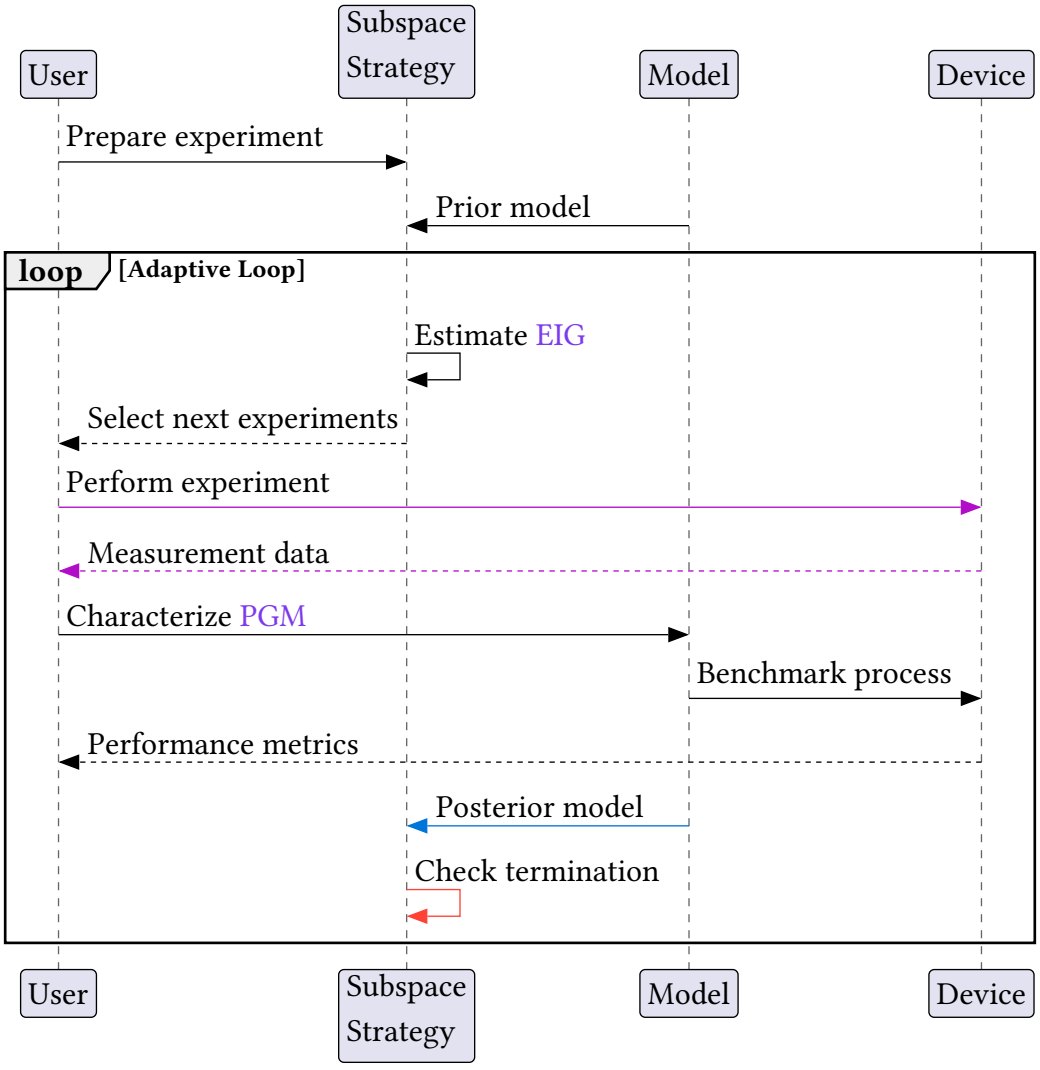


Figure 6.7: The sequence diagram of the subspace strategy.

3. **Alternative strategy**: Using an insight from subspace and random strategy, we also consider the alternative strategy, which first selects the control using the

subspace strategy, then uses random for the next round. This strategy is motivated by the observation that a subspace strategy can select a good initial batch of controls, thereby boosting the model's performance relative to a random strategy. However, after the long sequence, the random sequence can catch up. Thus, it is worth considering the best of both worlds.

Since BNN can learn from a small dataset, we focus on a few sample datasets rather than the `sample_size = 1000` used in the previous approaches. Here, we set the maximum sample size to `50` samples. In 10 sequential experiments, we collect five samples each time.

For the subspace strategy, we illustrate how we select the next experimental designs based on the EIG in Figure 6.8. The plot is one of the realizations of the sequential experiments of the subspace strategy. We plot the histogram of selected control parameters across the strategies in Figure 6.9 for all 10 trajectories (realizations).

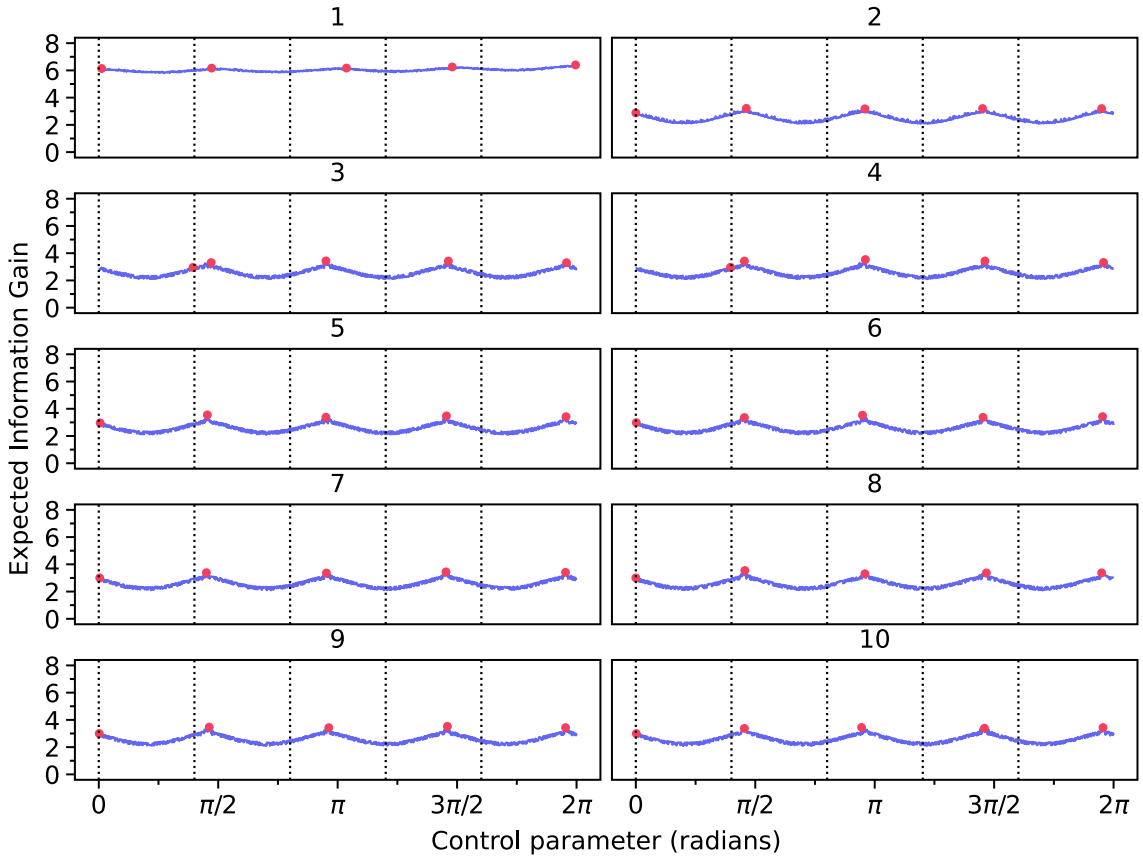


Figure 6.8: This figure shows the estimated EIG (y-axis) given control parameter (x-axis) over the 10 rounds of the sequential experiment of an example realization of the subspace strategy. The red points are the selected control parameters for the experiment at the current round. We can see that the strategy selects the data points that maximize the EIG in its own subspace.

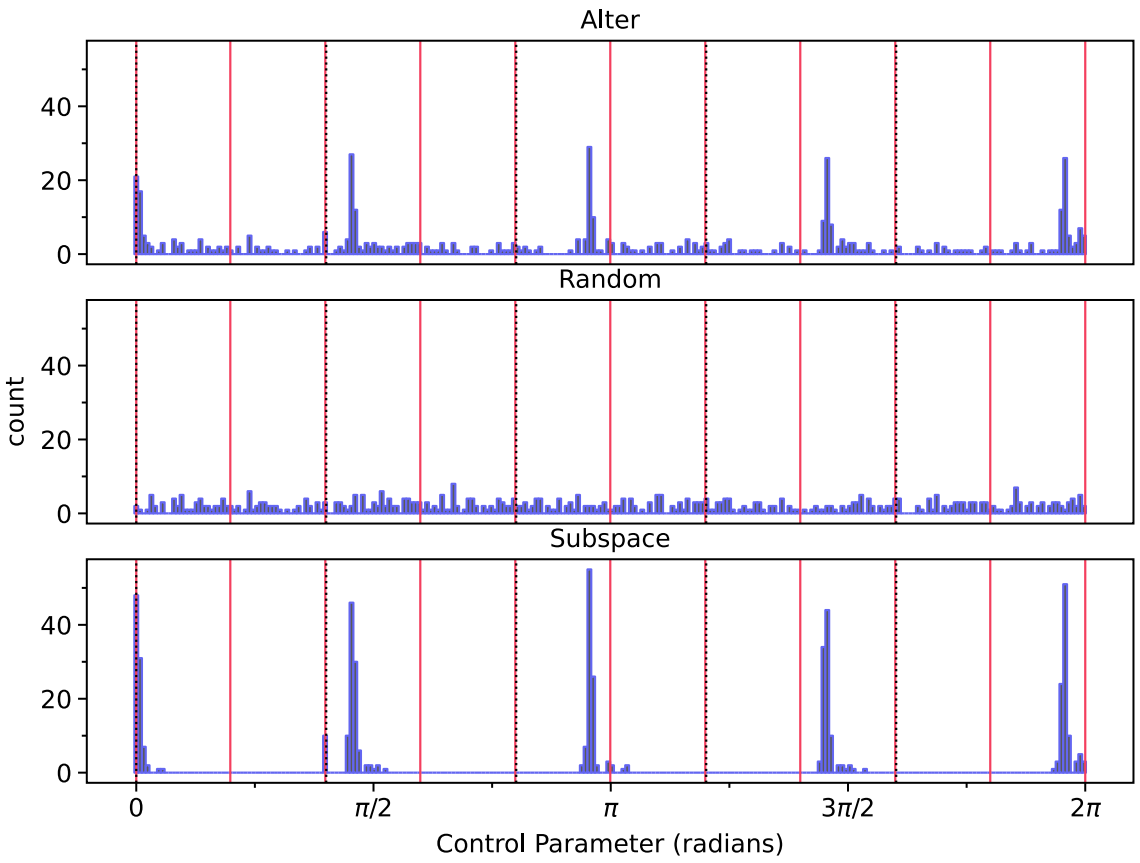


Figure 6.9: The histogram of the control parameters selected by the alternative, subspace, and random strategies across 10 trajectories. The dotted lines divide the control parameters into subspaces. The red lines correspond to the selected test control parameters.

We repeatedly run 10 sequential experiments to compare performance statistics. We test across multiple control parameters, as discussed in Section 3.3.1. The figure summarizes the benchmark results comparing the JSD of the model trained on datasets generated by the random and subspace strategies, as presented in Figure 6.10 and Figure 6.11 with a log-scale on the Y-axis. With the distributions of the test target and predicted control parameter, we also plot the expected test MSE $\mathbb{E}_{\text{test}}\left[\mathcal{L}_{\text{SE[E]}}\right]$ calculated using the mean of the predicted distribution as a representative prediction with the test distribution in Figure 6.12. We also list the average $\mathbb{E}_{\text{test}}\left[\mathcal{L}_{\text{SE[E]}}\right]$ over the control parameters of each strategy at different characterization rounds in Table 6.2. We chose test control parameters $\theta = \{0.0^\circ, 35.7^\circ, 71.7^\circ, 107.7^\circ, 143.8^\circ, 179.8^\circ, 215.9^\circ, 251.9^\circ, 287.9^\circ, 324.0^\circ, 360.0^\circ\}$.

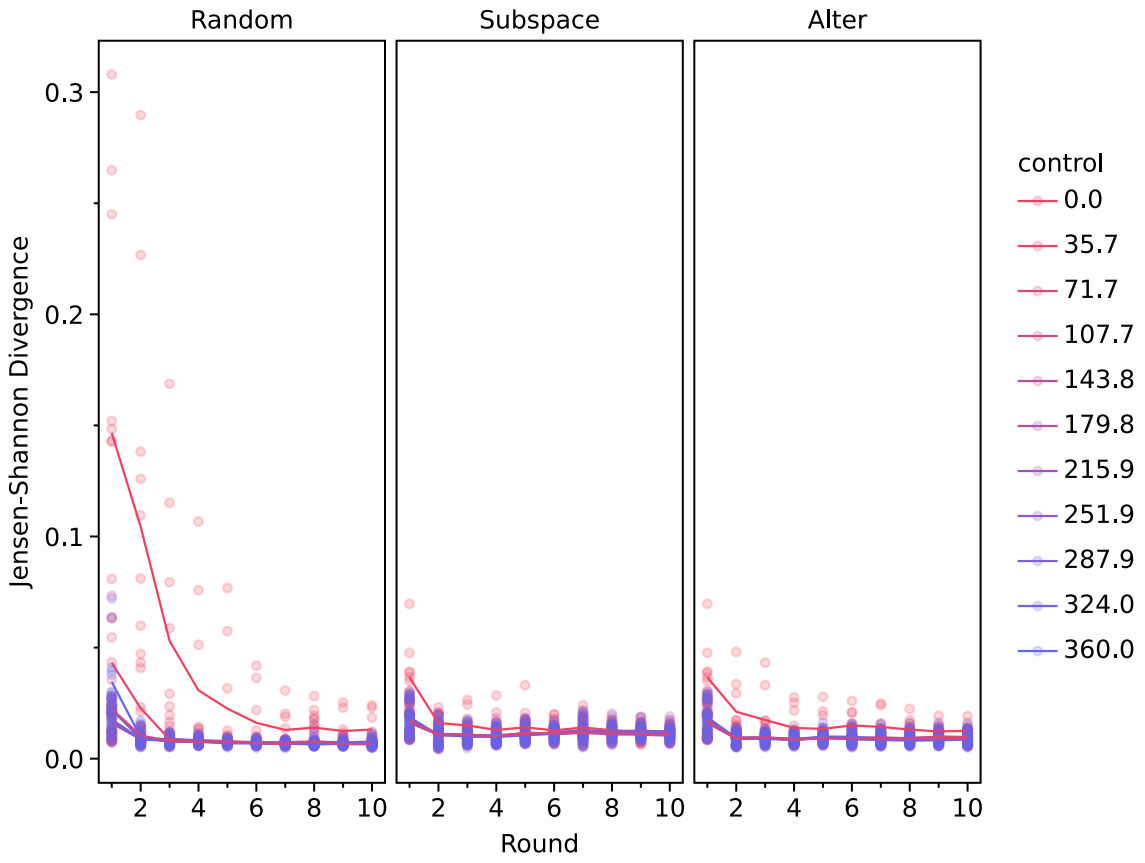


Figure 6.10: The JSD at the difference sample size. We perform 10 realizations of random, subspace, and alternative strategies and plot the mean as colored solid lines. Each realization uses a different random seed. We can see that the JSD values for the random strategy are more widely distributed than those for the alternative and subspace strategies, especially for the control parameters near 0°. The results suggest that, using alternative and subspace strategies, the predictive model can better predict the target distributions than a random strategy in the first few rounds.

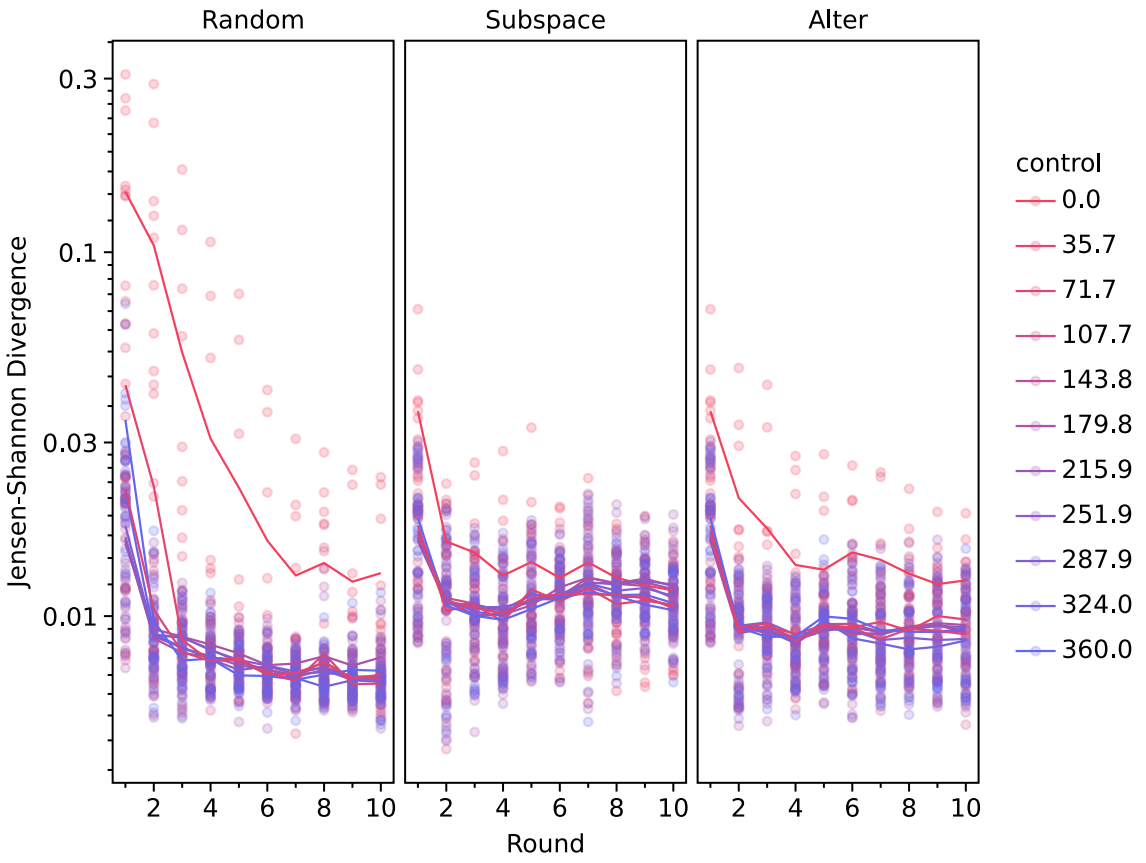


Figure 6.11: This figure use the same data as in Figure 6.10, but with log-scale on the Y-axis. With log-scale, we can observe that the JSD values corresponding to the random strategy quickly catches up the alternative and subspace strategies. We can also see that the JSD values of alternative strategy are smaller than the subspace strategy, suggesting a new strategy where we use subspace selection in the first few rounds and following by random selection.

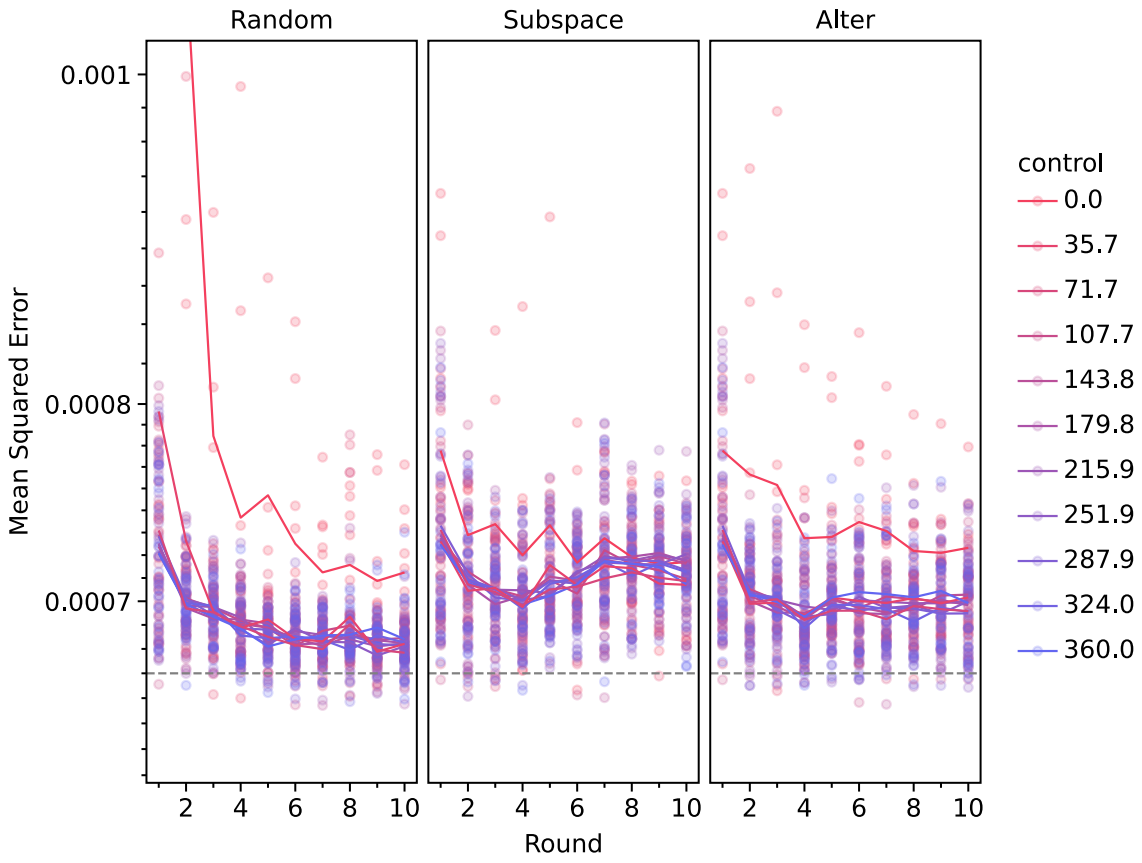


Figure 6.12: In a manner similar to Figure 6.11, we plot the test mean squared error $\mathbb{E}_{\text{test}}\left[\mathcal{L}_{\text{SE[E]}}\right]$ at different sample sizes in this figure, using a log scale. We can see that the performance of the random strategy is better than the EIG based strategies in the later rounds on average, and it approaches the data variance plot as a dashed gray line.

Round	Random	Subspace	Alter
1	0.000791	0.000735	0.000735
2	0.000736	0.000712	0.000708
3	0.000703	0.000707	0.000705
4	0.000693	0.000702	0.000696
5	0.000691	0.000711	0.000702
6	0.000687	0.000710	0.000702
7	0.000685	0.000720	0.000701
8	0.000688	0.000718	0.000700
9	0.000683	0.000717	0.000701
10	0.000683	0.000715	0.000701

Table 6.2: The table shows the average $\mathbb{E}_{\text{test}}\left[\mathcal{L}_{\text{SE[E]}}\right]$ over control parameters of each strategy at different characterization rounds.

We can see from the results in Figure 6.10 that the subspace strategy consistently selects control parameters (i.e., experimental design) that enable the model to perform better than a random strategy in the first few batches. From Figure 6.8 and Figure 6.9, we can observe that the subspace strategy actively selects control parameters near $\theta = \{0^\circ, 90^\circ, 180^\circ, 270^\circ, 360^\circ\}$. This subspace dataset is a good starter dataset that allows the model to start with good initial model parameters. However, since the landscape of the EIG does not change significantly, the choice of experimental designs is concentrated at the same points. Consequently, there is no significant improvement in performance with additional samples, as can be observed with log-scale in Figure 6.11 compared to the random strategy. On the other hand, the random strategy catches up and performs in terms of both JSD and $\mathbb{E}_{\text{test}}\left[\mathcal{L}_{\text{SE[E]}}\right]$ better after more data are collected, especially, that the $\mathbb{E}_{\text{test}}\left[\mathcal{L}_{\text{SE[E]}}\right]$ approach the data variance (dashed gray line). However, this strategy does not perform well at the $\theta = 0^\circ$ compared to the subspace, which deliberately chooses $\theta = 0^\circ$. So we consider the subspace alternative as the first in the sequence. We observe improved model performance compared to the pure subspace strategy. The alternative strategy performs well at $\theta = 0^\circ$, similar to the subspace strategy, and we can improve the model's performance with more samples. Although the improvement is not significant, and the random strategy can produce a model that catches up with more samples, the subspace strategy offers a better start than the random strategy. From an optimistic point of view, a random strategy yields good performance even with a small sample-size dataset, without the computational overhead of experimental design selection, as in the EIG based strategies. This observation could be advantageous in situations where reducing

experiment duration is more important than collecting a few data points. Regarding the use of EIG based strategies, there are several opportunities for improvement. For instance, multiple configurations of the strategies are possible with access to the EIG. One of the example is that we can warm-start the characterization with the subspace strategy on the first trial and then use the random strategy for the remaining trials. With warm-starting, we essentially give the random strategy a head start while also gaining the advantages of faster computation and potentially better JSD values in later trials. It is also an interesting open question whether the dataset bias due to EIG actually allows the model to learn faster with fewer data points, or whether the model performs better simply because there are fewer aspects of the dataset (i.e., the non-uniformity of the input features) for the model to learn.

To roughly assess the effectiveness of the BOED approach, we compare the standard deviation of the predictive model's weights and biases across sequential rounds. For each model parameter, we average the standard deviation across realizations (different random keys) and then calculate the difference between strategies, and plot the results in Figure 6.13. We note that this evaluation method is simple and implicitly assumes that the distribution of model parameters will concentrate at some true distributions. This assumption is reasonable for the physics-based model, but might not be strictly true for a BNN. Nonetheless, we use this method for evaluation because we apply the same prior distributions, model architectures, and training pipelines across all strategies. Consequently, we expect that the posterior distribution of the model parameters for each strategy will approach a similar local optimum. Furthermore, our primary interest lies in the relative reduction of the standard deviation between strategies, rather than evaluating its absolute value. We observe that substantial portions of the standard deviation values for the EIG -based strategies are lower than those for the random strategies. This observation suggests that BOED's approach has successfully improved confidence in parameter estimation compared to random strategies. Moreover, the confidence in the estimation of the subspace strategy is better than that of the alternative strategy, which is a mixed subspace-random strategy. The results from Figure 6.11 and Figure 6.13 suggest that being confident in model parameters does not directly imply a better predictive performance for the case of PGM. Moreover, there is also a possibility that the marginal guide used in this experiment is too simple[7]. It is also possible that using an EIG informed dataset may yield a highly biased model that does not generalize well to unseen data. Sometimes, we have observed datasets containing outliers that cause a model to perform worse (i.e., predict a very high JSD). We want to note that if the EIG informed strategy degrades model performance, we can abort the operation early. One possible preventive measure

[7]Please see https://github.com/PorametPat/thesis-code/blob/main/chapter-5-strategies-v4/model.py for more details

is to mix EIG informed designs with random designs as done in the alternative strategy. Nonetheless, even with the random strategy alone, we can see that PGM achieves strong predictive performance with few data points, suggesting that we should be careful when building an automated system to avoid unnecessary computation.

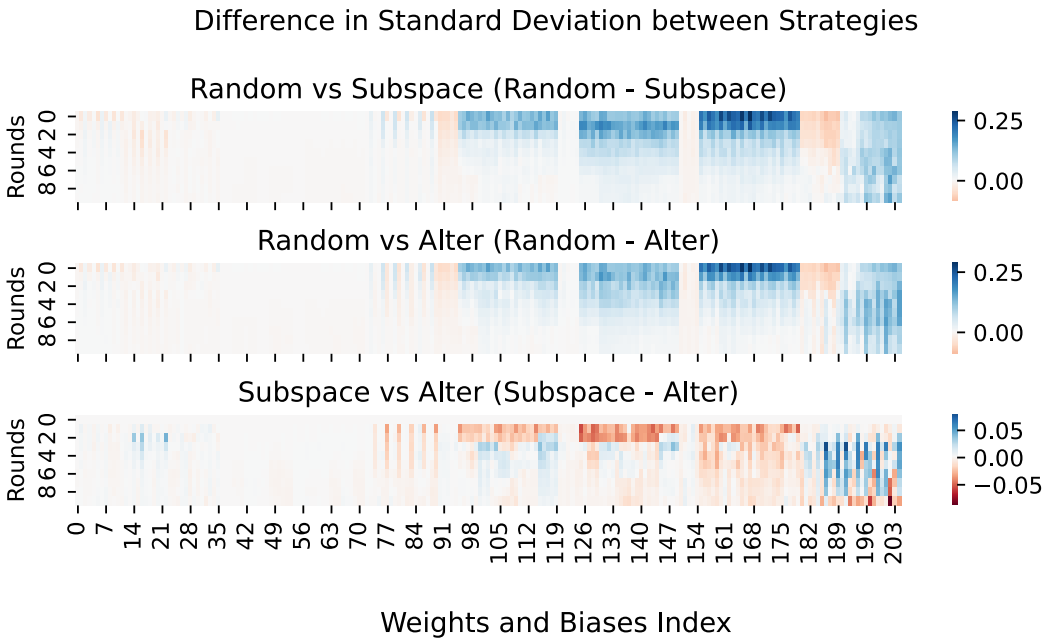


Figure 6.13: The heat map of the differences in the standard deviation of the weights and biases of the model between strategies. We can see that the standard deviation of the weights and biases for the subspace and alternative strategies is lower than that of the random strategy in earlier rounds, indicating that the algorithm is more confident about the weights and biases of the EIG based strategies than the random strategy.

Chapter 7. Next Steps Forward

Reflecting on the *journey so far*, we find ourselves at a pivotal moment, ready to revisit our discussions as we enter this final, decisive chapter on building a framework for C&C of the quantum device. Now is also the perfect time to cast our eyes forward and imagine the possibilities that await on the horizon.

7.1. Captain's Log: The Journey So Far

Let us review the contribution of this dissertation from the beginning. One of the most important questions that motivates our design of the framework is how we can efficiently utilize the information from the experimental data from the quantum system, which we discussed in Chapter 1. Let us review our question again by considering the typical scenario where we would like to characterize the quantum device by some model. This model can be motivated by a physical system of interest, or a platform-agnostic model such as a machine learning-based model. With the model abstraction, the characterization problem has been reduced to a model parameter estimation problem instead. Suppose that the experimental data contain a certain amount of information we need to estimate the model parameters that allow the model to predict the experiment to some accuracy. If consuming the resource is not costly, we can collect "more than enough" data to estimate the model parameters. However, in practice, consuming the resource is very expensive. So, we need to collect "just enough" data to estimate the model parameters.

In one simple way, we addressed the data efficiency question by answering another question. Naturally, we know that our collected data is sufficient when the predictive model performs well on the test dataset, ideally achieving optimal accuracy. However, in practice, it is very hard (if not impossible) for the model to achieve ideal accuracy due to noise in the training dataset itself, especially for datasets from the quantum system. In Section 3.1.3, we used the decomposition of mean squared error to derive the expected minimum value achievable in terms of the number of shots used to estimate the expectation value. We refer to this expected minimum value as the data variance. Moreover, we found that the data variance serves as the upper bound for the absolute difference between the predicted and exact AGF. With these results, it is now possible to iteratively collect a training dataset so that the predictive model constructed from it performs well on the test dataset to a desired accuracy.

To further optimize data utilization, we considered the BOED approach to select the data points that maximize EIG. But to use BOED, it is necessary to upgrade the statistical model to the probabilistic model. In Section 3.2.3, we propose Probabilistic Graybox Model (PGM), the probabilistic upgrade of Statistic Graybox Model (SGM). Our analysis suggests

that we cannot infer the variance of the intermediate expectation value. However, our numerical results show that PGM captures the statistics of the observed data better than SGM. Nonetheless, PGM is a necessary path forward for the Graybox characterization method to be used with BOED.

A quantum system is not limited to a single device. To truly harness the power of quantum computation, we need to connect multiple quantum devices so they can work together via the quantum Internet. At the core of the quantum internet is yet another quantum device, namely a quantum repeater, which also requires routine C&C. The availability of the device is essential, especially in the quantum network case, where a single down quantum repeater can easily become a network bottleneck. In Chapter 4, we discussed swapping-based and measurement-based approaches to distributing entanglement in quantum networks. We also emulate the quantum network protocol (MQNC) on the IBM Q device. Although the distributed entangled pairs do not violate the CHSH inequality, we are able to model the failure in terms of the noise that exists in the quantum operation. Thus, highlighting the importance of the C&C in this context. We then discuss the possible ways to apply the Graybox characterization method to the experiments to improve the connection fidelity.

In Chapter 5, we discussed the design and implementation of `inspeqtor` , which is the name of the data-efficient C&C framework and the Python library that implements the framework. At the heart of the framework, the central part is the design of the control module. We develop a way to structure the control action that can seamlessly be used with the solver of the quantum system, as well as in the training of a machine learning-based model. The solution is elegant since the developer can access the control parameter precisely using a key-value map, while also converts the control into the array representation, which is the data structure typically expected in training a machine learning model. Data module offers an opinionated way to store the control action and the corresponding experimental data. Taking advantage of the ease of transforming the control data structure, it is straightforward to serialize and deserialize the control to and from the file system. By abstracting the physics module, developers have finer control over the behavior of the quantum device, from the Hamiltonian to the measurement. On the other side of the coin, the model module offers a "starter pack" for modeling the system using parametrized operators, such as Hermitian and unitary operators, via a neural network. Moreover, we provide a modified unitary-based model to handle SPAM noise, which is very important in realistic settings.

When the need for a more advanced model arises, developer can define their own custom model and an appropriate adapter for it to work with `inspeqtor` . Data, control, physics, model, and probabilistic modules (including a custom BNN built with the proba-

bilistic module) are designed to be fully compatible with BOED, implemented in the BOED module.

Library / Tool	Data-efficient Approach	Flexibility
`inspeqtor` (this dissertation)	EIG, [55]	**High**
Qibocal [116]	**No**	**Low** (predefined experiments)
JAXQuantum [117]	**No**	**Moderate** (Based on jax, but main for superconducting qubit)
C3 [115] *(last updated Jan 2024)*	**No**	Unknown
QCtrl Boulder Opal [56] (closed-source)	OBSID [57]	**High**

Table 7.1: Comparison of software libraries and tools for quantum characterization.

On a technical level, implementing the framework in Python makes it easier for researchers to use. On the other hand, we use JAX as a numerical framework that comes with several benefits. For instance, JAX offers automatic differentiation and vectorization in a functional style, making composition, extension, reusability, and testing easy. Furthermore, we can benefit from Just-In-Time compilation offered by JAX, which significantly improves the execution speed. Even more, JAX makes it easy to deploy the program across CPUs, GPUs, and TPUs, which infrastructure engineers can greatly benefit from. We list the libraries and the approach to the data utilization efficiency in Table 7.1.

Finally, we demonstrated the characterization of a transmon qubit using statistical, probabilistic, and BOED approaches with `inspeqtor` in Chapter 6. We observed that EIG -based strategies offer a head start in predictive model construction, as their models can achieve lower JSD values than the random strategy in the first few rounds. Although the random strategy can catch up in later rounds, the results also suggest that it can benefit from the warm start provided by EIG -based strategies.

7.2. To boldly go where no one has gone before

From the hints in the contributions, we now discuss the open questions and fertile grounds for future research that our framework unlocks.

Synchronization signal and automating recalibration: A persistent challenge in operating quantum processors is the drift in system parameters over time. While we established that data variance serves as an upper bound for the absolute difference between predicted and exact AGF, this variance holds untapped potential as a "synchronization signal". By continuously monitoring the empirical test MSE loss of the model, we could detect when the predictive model has desynchronized from the actual physical device due to parameter drift. Once the preset threshold is reached, the system can trigger a maintenance routine, becoming an automated recalibration system. Nonetheless, this data variance approach is not the only way to achieve an automated system. For example, F. Mazurek *et al.* [183] uses statistical model checking to construct a maintenance system for a quantum device. Other approaches [78], [184], [185], [186], [187], [188] rely on additional qubits called "spectator qubit" to monitor the noise in the system. Regardless of the approach, implementing the necessary tools to build the automated recalibration system is one of the next important steps forward.

Alternative Data-efficient methods: As discussed in Section 2.4 and Section 3.3, one-step optimal BOED demonstrated in Section 6.5 is not the only way to achieve data-efficient characterization. Implementing and integrating those methods into Inspeqtor is an enormous amount of work, but it is also an interesting future direction. Especially, the sequential BOED approaches, which are direct successors to the One-step BOED, may require a large amount of computational memory during the calculation. Much work remains to be done on both theoretical and numerical aspects.

Structural identifiability analysis: Although BOED and other data-efficient approaches allow us to select data points that are more informative about the system than others, we briefly discussed in Section 2.4 that, if the control is poorly chosen or there are too few data points, it may not be possible to uniquely identify the model parameters. It would therefore be beneficial to implement tools for structural identifiability analysis for the quantum system, enabling users to assess whether their chosen model and control are appropriate before performing actual computations and experiments.

Implementation in other Programming Languages: Python is obviously a famous choice when it comes to data science or quickly sketching an idea. However, when a task becomes crucial where mistakes and errors have to be prevented or caught as soon as possible, or before they even occur, Python might not be the answer. Furthermore, when the need to maximize performance and minimize memory footprint arises, there are more suitable programming languages that meet the requirements. For example, Julia is a relatively new dynamic programming language designed to be high-performance yet easy to program, unlike a complex language such as C++. If type strictness, error handling, and performance are important, Rust may be a solution.

At the time of writing this dissertation, the question of how quantum computation will change our lives is raised with increasing frequency. We are currently in a highly competitive era where industries are rushing to translate research into practical commercial products, though a clear winner has yet to emerge. In this race for profitability, many technological advancements are likely to be kept secret. While this is understandable from a business perspective, it creates barriers for collaboration across broader scientific communities, ultimately slowing the overall pace of human progress.

For humanity to reach the stars and go beyond our current capabilities, we must build upon a solid, shared foundation. We hope that our open-source framework will serve as a tool for the community to strive toward a better future.

Bibliography

[1] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, Oct. 1997, doi: 10.1137/S0097539795293172.

[2] R. Van Meter, W. J. Munro, K. Nemoto, and K. M. Itoh, "Arithmetic on a Distributed-Memory Quantum Multicomputer," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 3, no. 4, pp. 1–23, Jan. 2008, doi: 10.1145/1324177.1324179.

[3] G. H. Low and I. L. Chuang, "Hamiltonian Simulation by Qubitization," *Quantum*, vol. 3, p. 163, Jul. 2019, doi: 10.22331/q-2019-07-12-163.

[4] A. Gilyén, Y. Su, G. H. Low, and N. Wiebe, "Quantum Singular Value Transformation and beyond: Exponential Improvements for Quantum Matrix Arithmetics," in *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, Phoenix AZ USA: ACM, Jun. 2019, pp. 193–204. doi: 10.1145/3313276.3316366.

[5] L. Clinton, J. Bausch, and T. Cubitt, "Hamiltonian Simulation Algorithms for Near-Term Quantum Hardware," *Nature Communications*, vol. 12, no. 1, p. 4989, Aug. 2021, doi: 10.1038/s41467-021-25196-0.

[6] J. M. Martyn, Z. M. Rossi, A. K. Tan, and I. L. Chuang, "Grand Unification of Quantum Algorithms," *PRX Quantum*, vol. 2, no. 4, p. 40203, Dec. 2021, doi: 10.1103/PRXQuantum.2.040203.

[7] E. Farhi, J. Goldstone, S. Gutmann, and L. Zhou, "The Quantum Approximate Optimization Algorithm and the Sherrington-Kirkpatrick Model at Infinite Size." Accessed: Mar. 17, 2026. [Online]. Available: http://arxiv.org/abs/1910.08187

[8] Y. Cao, S. Jin, and N. Liu, "Unifying Framework for Quantum Simulation Algorithms for Time-Dependent Hamiltonian Dynamics," *Physical Review Research*, vol. 7, no. 4, p. 43186, Nov. 2025, doi: 10.1103/fkh5-b669.

[9] M. Gili, P. San Sebastian, and A. Blázquez-García, "Optimization of Flight Routes: Quantum Approximate Optimization Algorithm for the Tail Assignment Problem," *Quantum Machine Intelligence*, vol. 8, no. 1, p. 54, Jun. 2026, doi: 10.1007/s42484-026-00387-4.

[10] C. Gidney and M. Ekerå, "How to Factor 2048 Bit RSA Integers in 8 Hours Using 20 Million Noisy Qubits," *Quantum*, vol. 5, p. 433, Apr. 2021, doi: 10.22331/q-2021-04-15-433.

[11] J. Lee *et al.*, "Even More Efficient Quantum Computations of Chemistry Through Tensor Hypercontraction," *PRX Quantum*, vol. 2, no. 3, p. 30305, Jul. 2021, doi: 10.1103/PRXQuantum.2.030305.

[12] R. Babbush *et al.*, "The Grand Challenge of Quantum Applications." Accessed: May 20, 2026. [Online]. Available: https://arxiv.org/abs/2511.09124

[13] C. H. Bennett and G. Brassard, "Quantum Cryptography: Public Key Distribution and Coin Tossing," *Theoretical Computer Science*, vol. 560, pp. 7–11, Dec. 2014, doi: 10.1016/j.tcs.2014.05.025.

[14] Ö. Erkılıç *et al.*, "Enhanced Continuous-Variable Quantum Key Distribution Protocol via Adaptive Signal Processing," *Communications Physics*, vol. 8, no. 1, p. 406, Oct. 2025, doi: 10.1038/s42005-025-02317-5.

[15] P. Andriolo, E. Vasquez, E. Agudelo, M. Riegler, M. Pivoluska, and G. Murta, "Quantum Key Distribution with Imperfections: Recent Advances in Security Proofs," *Brazilian Journal of Physics*, vol. 56, no. 4, p. 157, Apr. 2026, doi: 10.1007/s13538-026-02062-2.

[16] B.-W. Lu *et al.*, "Device-Independent Quantum Key Distribution over 100 Km with Single Atoms," *Science*, vol. 391, no. 6785, pp. 592–597, Feb. 2026, doi: 10.1126/science.aec6243.

[17] A. Trabesinger, "Quantum Simulation," *Nature Physics*, vol. 8, no. 4, p. 263, Apr. 2012, doi: 10.1038/nphys2258.

[18] M. Reiher, N. Wiebe, K. M. Svore, D. Wecker, and M. Troyer, "Elucidating Reaction Mechanisms on Quantum Computers," *Proceedings of the National Academy of Sciences*, vol. 114, no. 29, pp. 7555–7560, Jul. 2017, doi: 10.1073/pnas.1619152114.

[19] T. Proctor *et al.*, "Detecting and Tracking Drift in Quantum Information Processors," *Nature Communications*, vol. 11, no. 1, p. 5396, Oct. 2020, doi: 10.1038/s41467-020-19074-4.

[20] B. Li, F. A. Calderon-Vargas, J. Zeng, and E. Barnes, "Designing Arbitrary Single-Axis Rotations Robust against Perpendicular Time-Dependent Noise," *New Journal of Physics*, vol. 23, no. 9, p. 93032, Sep. 2021, doi: 10.1088/1367-2630/ac22ea.

[21] S. Dasgupta, T. S. Humble, and A. Danageozian, "Adaptive Mitigation of Time-Varying Quantum Noise," in *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, Bellevue, WA, USA: IEEE, Sep. 2023, pp. 99–110. doi: 10.1109/QCE57702.2023.00020.

[22] J. Benestad, J. A. Krzywda, E. van Nieuwenburg, and J. Danon, "Efficient Adaptive Bayesian Estimation of a Slowly Fluctuating Overhauser Field Gradient," *SciPost Physics*, vol. 17, no. 1, p. 14, Jul. 2024, doi: 10.21468/SciPostPhys.17.1.014.

[23] X. Yang *et al.*, "Quantum Control for Time-Dependent Noise by Inverse Geometric Optimization," *Science China Physics, Mechanics & Astronomy*, vol. 67, no. 9, p. 290312, Sep. 2024, doi: 10.1007/s11433-024-2411-y.

[24] B. Gulácsi and G. Burkard, "Temporally Correlated Quantum Noise in Driven Quantum Systems with Applications to Quantum Gate Operations," *Physical Review Research*, vol. 7, no. 2, p. 23073, Apr. 2025, doi: 10.1103/PhysRevResearch.7.023073.

[25] B. J. Brown, N. H. Nickerson, and D. E. Browne, "Fault-Tolerant Error Correction with the Gauge Color Code," *Nature Communications*, vol. 7, no. 1, p. 12302, Jul. 2016, doi: 10.1038/ncomms12302.

[26] R. Chao and B. W. Reichardt, "Flag Fault-Tolerant Error Correction for Any Stabilizer Code," *PRX Quantum*, vol. 1, no. 1, p. 10302, Sep. 2020, doi: 10.1103/PRXQuantum.1.010302.

[27] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, "Surface Codes: Towards Practical Large-Scale Quantum Computation," *Physical Review A*, vol. 86, no. 3, p. 32324, Sep. 2012, doi: 10.1103/PhysRevA.86.032324.

[28] N. R. Dash, S. Dutta, R. Srikanth, and S. Banerjee, "Concatenating Quantum Error-Correcting Codes with Decoherence-Free Subspaces and Vice Versa," *Physical Review A*, vol. 109, no. 6, p. 62411, Jun. 2024, doi: 10.1103/PhysRevA.109.062411.

[29] Google Quantum AI and Collaborators *et al.*, "Quantum Error Correction below the Surface Code Threshold," *Nature*, vol. 638, no. 8052, pp. 920–926, Feb. 2025, doi: 10.1038/s41586-024-08449-y.

[30] H.-J. Briegel, W. Dür, J. I. Cirac, and P. Zoller, "Quantum Repeaters: The Role of Imperfect Local Operations in Quantum Communication," *Physical Review Letters*, vol. 81, no. 26, pp. 5932–5935, Dec. 1998, doi: 10.1103/PhysRevLett.81.5932.

[31] L. Jiang, J. M. Taylor, K. Nemoto, W. J. Munro, R. Van Meter, and M. D. Lukin, "Quantum Repeater with Encoding," *Physical Review A*, vol. 79, no. 3, p. 32325, Mar. 2009, doi: 10.1103/PhysRevA.79.032325.

[32] K. Azuma *et al.*, "Quantum Repeaters: From Quantum Networks to the Quantum Internet," *Reviews of Modern Physics*, vol. 95, no. 4, p. 45006, Dec. 2023, doi: 10.1103/RevModPhys.95.045006.

[33] N. Benchasattabuse, M. Hajdušek, and R. Van Meter, "Bridging All-Photonic and Memory-Based Quantum Repeaters," *IEEE Transactions on Quantum Engineering*, vol. 7, pp. 1–15, 2026, doi: 10.1109/TQE.2026.3653126.

[34] H. J. Kimble, "The Quantum Internet," *Nature*, vol. 453, no. 7198, pp. 1023–1030, Jun. 2008, doi: 10.1038/nature07127.

[35] R. Van Meter, *Quantum Networking*, 1st ed. Wiley, 2014. doi: 10.1002/9781118648919.

[36] S. Wehner, D. Elkouss, and R. Hanson, "Quantum Internet: A Vision for the Road Ahead," *Science*, vol. 362, no. 6412, p. eaam9288, Oct. 2018, doi: 10.1126/science.aam9288.

[37] S. Muralidharan, L. Li, J. Kim, N. Lütkenhaus, M. D. Lukin, and L. Jiang, "Optimal Architectures for Long Distance Quantum Communication," *Scientific Reports*, vol. 6, no. 1, p. 20463, Feb. 2016, doi: 10.1038/srep20463.

[38] S. Bravyi, O. Dial, J. M. Gambetta, D. Gil, and Z. Nazario, "The Future of Quantum Computing with Superconducting Qubits," *Journal of Applied Physics*, vol. 132, no. 16, p. 160902, Oct. 2022, doi: 10.1063/5.0082975.

[39] Y. Kim *et al.*, "Evidence for the Utility of Quantum Computing before Fault Tolerance," *Nature*, vol. 618, no. 7965, pp. 500–505, Jun. 2023, doi: 10.1038/s41586-023-06096-3.

[40] R. D. Delaney *et al.*, "Scalable Multispecies Ion Transport in a Grid-Based Surface-Electrode Trap," *Physical Review X*, vol. 14, no. 4, p. 41028, Nov. 2024, doi: 10.1103/PhysRevX.14.041028.

[41] A. Ransford *et al.*, "Helios: A 98-Qubit Trapped-Ion Quantum Computer." Accessed: May 03, 2026. [Online]. Available: https://arxiv.org/abs/2511.05465

[42] M. Liu *et al.*, "Certified Randomness Using a Trapped-Ion Quantum Processor," *Nature*, vol. 640, no. 8058, pp. 343–348, Apr. 2025, doi: 10.1038/s41586-025-08737-1.

[43] K. Seki, Y. Kikuchi, T. Hayata, and S. Yunoki, "Simulating Floquet Scrambling Circuits on Trapped-Ion Quantum Computers," *Physical Review Research*, vol. 7, no. 2, p. 23032, Apr. 2025, doi: 10.1103/PhysRevResearch.7.023032.

[44] R. Haghshenas *et al.*, "Digital Quantum Magnetism on a Trapped-Ion Quantum Computer," *Nature*, Apr. 2026, doi: 10.1038/s41586-026-10445-3.

[45] F. Tripier *et al.*, "Fault-Tolerant Quantum Computing with Trapped Ions: The Walking Cat Architecture." Accessed: May 03, 2026. [Online]. Available: https://arxiv.org/abs/2604.19481

[46] A. Salavrakos, T. Sedrakyan, J. Mills, S. Mansfield, and R. Mezher, "Error-Mitigated Photonic Quantum Circuit Born Machine," *Physical Review A*, vol. 111, no. 3, p. L030401, Mar. 2025, doi: 10.1103/PhysRevA.111.L030401.

[47] R. Mezher and S. Mansfield, "Assessing the Quality of Near-Term Photonic Quantum Devices." Accessed: May 03, 2026. [Online]. Available: https://arxiv.org/abs/2202.04735

[48] H. Aghaee Rad *et al.*, "Scaling and Networking a Modular Photonic Quantum Computer," *Nature*, vol. 638, no. 8052, pp. 912–919, Feb. 2025, doi: 10.1038/s41586-024-08406-9.

[49] K. Alexander *et al.*, "A Manufacturable Platform for Photonic Quantum Computing," *Nature*, vol. 641, no. 8064, pp. 876–883, May 2025, doi: 10.1038/s41586-025-08820-7.

[50] D. Sommers *et al.*, "Open-Source Highly Parallel Electromagnetic Simulations for Superconducting Circuits." Accessed: May 03, 2026. [Online]. Available: https://arxiv.org/abs/2511.01220

[51] N. Heurtel *et al.*, "Perceval: A Software Platform for Discrete Variable Photonic Quantum Computing," *Quantum*, vol. 7, p. 931, Feb. 2023, doi: 10.22331/q-2023-02-21-931.

[52] A. Youssry, G. A. Paz-Silva, and C. Ferrie, "Characterization and Control of Open Quantum Systems beyond Quantum Noise Spectroscopy," *npj Quantum Information*, vol. 6, no. 1, p. 95, Dec. 2020, doi: 10.1038/s41534-020-00332-8.

[53] A. Youssry *et al.*, "Experimental Graybox Quantum System Identification and Control," *npj Quantum Information*, vol. 10, no. 1, p. 9, Jan. 2024, doi: 10.1038/s41534-023-00795-5.

[54] T. Rainforth, A. Foster, D. R. Ivanova, and F. Bickford Smith, "Modern Bayesian Experimental Design," *Statistical Science*, vol. 39, no. 1, Feb. 2024, doi: 10.1214/23-STS915.

[55] A. Foster *et al.*, "Variational Bayesian Optimal Experimental Design," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, no. 1259, Red Hook, NY, USA: Curran Associates Inc., 2019, pp. 14059–14070.

[56] "Boulder Opal." Accessed: Jul. 01, 2025. [Online]. Available: https://docs.q-ctrl.com/boulder-opal

[57] T. M. Stace *et al.*, "Optimized Bayesian System Identification in Quantum Devices," *Physical Review Applied*, vol. 21, no. 1, p. 14012, Jan. 2024, doi: 10.1103/PhysRevApplied.21.014012.

[58] J. P. Marceaux, "Efficient Characterization, Calibration, and Correction of Quantum Information Processors," Doctoral dissertation, 2025. [Online]. Available: https://escholarship.org/uc/item/969239sw

[59] L. Sarra and F. Marquardt, "Deep Bayesian Experimental Design for Quantum Many-Body Systems," *Machine Learning: Science and Technology*, vol. 4, no. 4, p. 45022, Dec. 2023, doi: 10.1088/2632-2153/ad020d.

[60] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, 10th anniversary ed. Cambridge ; New York: Cambridge University Press, 2010.

[61] S. Majidy, C. Wilson, and R. Laflamme, *Building Quantum Computers: A Practical Introduction*. Cambridge New York, NY Port Melbourne New Delhi Singapore: Cambridge University Press, 2025.

[62] J. F. Clauser, M. A. Horne, A. Shimony, and R. A. Holt, "Proposed Experiment to Test Local Hidden-Variable Theories," *Physical Review Letters*, vol. 23, no. 15, pp. 880–884, Oct. 1969, doi: 10.1103/PhysRevLett.23.880.

[63] N. Lambert *et al.*, "QuTiP 5: The Quantum Toolbox in Python." Accessed: Nov. 15, 2025. [Online]. Available: https://arxiv.org/abs/2412.04705

[64] D. Manzano, "A Short Introduction to the Lindblad Master Equation," *AIP Advances*, vol. 10, no. 2, p. 25106, Feb. 2020, doi: 10.1063/1.5115323.

[65] A. Javadi-Abhari *et al.*, "Quantum Computing with Qiskit." Accessed: May 07, 2025. [Online]. Available: https://arxiv.org/abs/2405.08810

[66] X. Laforgue, G. Dridi, and S. Guérin, "Optimal Quantum Control Robust against Pulse Inhomogeneities: Analytic Solutions," *Physical Review A*, vol. 106, no. 5, p. 52608, Nov. 2022, doi: 10.1103/PhysRevA.106.052608.

[67] A. Hashim *et al.*, "Practical Introduction to Benchmarking and Characterization of Quantum Computers," *PRX Quantum*, vol. 6, no. 3, p. 30202, Aug. 2025, doi: 10.1103/PRXQuantum.6.030202.

[68] 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*, vol. 6, no. 2, p. 21318, Jun. 2019, doi: 10.1063/1.5089550.

[69] S. Günther, N. A. Petersson, and J. L. Dubois, "Quandary: An Open-Source C++ Package for High-Performance Optimal Control of Open Quantum Systems." Accessed: Nov. 14, 2025. [Online]. Available: http://arxiv.org/abs/2110.10310

[70] E. Paladino, Y. M. Galperin, G. Falci, and B. L. Altshuler, "1 / f Noise: Implications for Solid-State Quantum Information," *Reviews of Modern Physics*, vol. 86, no. 2, pp. 361–418, Apr. 2014, doi: 10.1103/RevModPhys.86.361.

[71] T. Chalermpusitarak, B. Tonekaboni, Y. Wang, L. M. Norris, L. Viola, and G. A. Paz-Silva, "Frame-Based Filter-Function Formalism for Quantum Characterization and Control," *PRX Quantum*, vol. 2, no. 3, p. 30315, Jul. 2021, doi: 10.1103/PRXQuantum.2.030315.

[72] D. Hocker *et al.*, "Characterization of Control Noise Effects in Optimal Quantum Unitary Dynamics," *Physical Review A*, vol. 90, no. 6, p. 62309, Dec. 2014, doi: 10.1103/PhysRevA.90.062309.

[73] G. A. Álvarez and D. Suter, "Measuring the Spectrum of Colored Noise by Dynamical Decoupling," *Physical Review Letters*, vol. 107, no. 23, p. 230501, Nov. 2011, doi: 10.1103/PhysRevLett.107.230501.

[74] L.-L. Yan, J.-Q. Zhang, J. Jing, and M. Feng, "Suppression of Dissipation in a Laser-Driven Qubit by White Noise," *Physics Letters A*, vol. 379, no. 39, pp. 2417–2423, Oct. 2015, doi: 10.1016/j.physleta.2015.06.023.

[75] F. Motzoi, J. M. Gambetta, P. Rebentrost, and F. K. Wilhelm, "Simple Pulses for Elimination of Leakage in Weakly Nonlinear Qubits," *Physical Review Letters*, vol. 103, no. 11, p. 110501, Sep. 2009, doi: 10.1103/PhysRevLett.103.110501.

[76] L. S. Theis, F. Motzoi, S. Machnes, and F. K. Wilhelm, "Counteracting Systems of Diabaticities Using DRAG Controls: The Status after 10 Years^{(a)}," *EPL (Europhysics Letters)*, vol. 123, no. 6, p. 60001, Oct. 2018, doi: 10.1209/0295-5075/123/60001.

[77] E. Hyyppä *et al.*, "Reducing Leakage of Single-Qubit Gates for Superconducting Quantum Processors Using Analytical Control Pulse Envelopes," *PRX Quantum*, vol. 5, no. 3, p. 30353, Sep. 2024, doi: 10.1103/PRXQuantum.5.030353.

[78] S. Majumder, L. Andreta De Castro, and K. R. Brown, "Real-Time Calibration with Spectator Qubits," *npj Quantum Information*, vol. 6, no. 1, p. 19, Feb. 2020, doi: 10.1038/s41534-020-0251-y.

[79] A. Ash- Saki, M. Alam, and S. Ghosh, "Experimental Characterization, Modeling, and Analysis of Crosstalk in a Quantum Computer," *IEEE Transactions on Quantum Engineering*, vol. 1, pp. 1–6, 2020, doi: 10.1109/TQE.2020.3023338.

[80] B. Undseth *et al.*, "Nonlinear Response and Crosstalk of Electrically Driven Silicon Spin Qubits," *Physical Review Applied*, vol. 19, no. 4, p. 44078, Apr. 2023, doi: 10.1103/PhysRevApplied.19.044078.

[81] P. Zhao, "Mitigation of Quantum Crosstalk in Cross-Resonance-Based Qubit Architectures," *Physical Review Applied*, vol. 20, no. 5, p. 54033, Nov. 2023, doi: 10.1103/PhysRevApplied.20.054033.

[82] K. Heya, Y. Suzuki, Y. Nakamura, and K. Fujii, "Variational Quantum Gate Optimization." Accessed: Apr. 25, 2026. [Online]. Available: http://arxiv.org/abs/1810.12745

[83] S. S. Tannu and M. K. Qureshi, "Mitigating Measurement Errors in Quantum Computers by Exploiting State-Dependent Bias," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, Columbus OH USA: ACM, Oct. 2019, pp. 279–290. doi: 10.1145/3352460.3358265.

[84] M. R. Geller, "Rigorous Measurement Error Correction," *Quantum Science and Technology*, vol. 5, no. 3, p. 03LT01, Jun. 2020, doi: 10.1088/2058-9565/ab9591.

[85] L. Funcke, T. Hartung, K. Jansen, S. Kühn, P. Stornati, and X. Wang, "Measurement Error Mitigation in Quantum Computers through Classical Bit-Flip Correction," *Physical Review A*, vol. 105, no. 6, p. 62404, Jun. 2022, doi: 10.1103/PhysRevA.105.062404.

[86] M. A. Nielsen, "A Simple Formula for the Average Gate Fidelity of a Quantum Dynamical Operation," *Physics Letters A*, vol. 303, no. 4, pp. 249–252, Oct. 2002, doi: 10.1016/S0375-9601(02)01272-0.

[87] C. P. Koch *et al.*, "Quantum Optimal Control in Quantum Technologies. Strategic Report on Current Status, Visions and Goals for Research in Europe," *EPJ Quantum Technology*, vol. 9, no. 1, p. 19, Dec. 2022, doi: 10.1140/epjqt/s40507-022-00138-x.

[88] M. Bukov, A. G. R. Day, D. Sels, P. Weinberg, A. Polkovnikov, and P. Mehta, "Reinforcement Learning in Different Phases of Quantum Control," *Physical Review X*, vol. 8, no. 3, p. 31086, Sep. 2018, doi: 10.1103/PhysRevX.8.031086.

[89] X.-M. Zhang, Z. Wei, R. Asad, X.-C. Yang, and X. Wang, "When Does Reinforcement Learning Stand out in Quantum Control? A Comparative Study on State Preparation," *npj Quantum Information*, vol. 5, no. 1, p. 85, Oct. 2019, doi: 10.1038/s41534-019-0201-8.

[90] Yuval Baum *et al.*, "Experimental Deep Reinforcement Learning for Error-Robust Gate-Set Design on a Superconducting Quantum Computer," *PRX Quantum*, vol. 2, no. 4, p. 40324, Nov. 2021, doi: 10.1103/PRXQuantum.2.040324.

[91] M. Y. Niu, S. Boixo, V. N. Smelyanskiy, and H. Neven, "Universal Quantum Control through Deep Reinforcement Learning," *npj Quantum Information*, vol. 5, no. 1, p. 33, Apr. 2019, doi: 10.1038/s41534-019-0141-3.

[92] M. F. Lazin, C. R. Shelton, S. N. Sandhofer, and B. M. Wong, "High-Dimensional Multi-Fidelity Bayesian Optimization for Quantum Control," *Machine Learning:*

Science and Technology, vol. 4, no. 4, p. 45014, Dec. 2023, doi: 10.1088/2632-2153/ad0100.

[93] S. A. Renganathan, J. Larson, and S. M. Wild, "Lookahead Acquisition Functions for Finite-Horizon Time-Dependent Bayesian Optimization and Application to Quantum Optimal Control." Accessed: Apr. 02, 2025. [Online]. Available: http://arxiv.org/abs/2105.09824

[94] F. Sauvage and F. Mintert, "Optimal Quantum Control with Poor Statistics," *PRX Quantum*, vol. 1, no. 2, p. 20322, Dec. 2020, doi: 10.1103/PRXQuantum.1.020322.

[95] A. N. Sloss and S. Gustafson, "2019 Evolutionary Algorithms Review," in *Genetic Programming Theory and Practice XVII*, W. Banzhaf, E. Goodman, L. Sheneman, L. Trujillo, and B. Worzel, Eds., Cham: Springer International Publishing, 2020, pp. 307–344. doi: 10.1007/978-3-030-39958-0_16.

[96] M. J. Kochenderfer and T. A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019.

[97] N. Khaneja, T. 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*, vol. 172, no. 2, pp. 296–305, Feb. 2005, doi: 10.1016/j.jmr.2004.11.004.

[98] R. Porotti, V. Peano, and F. Marquardt, "Gradient-Ascent Pulse Engineering with Feedback," *PRX Quantum*, vol. 4, no. 3, p. 30305, Jul. 2023, doi: 10.1103/PRXQuantum.4.030305.

[99] S. Machnes, E. Assémat, D. Tannor, and F. K. Wilhelm, "Tunable, Flexible, and Efficient Optimization of Control Pulses for Practical Qubits," *Physical Review Letters*, vol. 120, no. 15, p. 150401, Apr. 2018, doi: 10.1103/PhysRevLett.120.150401.

[100] U. Güngördü and J. P. Kestner, "Robust Quantum Gates Using Smooth Pulses and Physics-Informed Neural Networks," *Physical Review Research*, vol. 4, no. 2, p. 23155, May 2022, doi: 10.1103/PhysRevResearch.4.023155.

[101] M. H. Goerz, K. B. Whaley, and C. P. Koch, "Hybrid Optimization Schemes for Quantum Control," *EPJ Quantum Technology*, vol. 2, no. 1, p. 21, Dec. 2015, doi: 10.1140/epjqt/s40507-015-0034-0.

[102] D. J. Egger and F. K. Wilhelm, "Adaptive Hybrid Optimal Quantum Control for Imprecisely Characterized Systems," *Physical Review Letters*, vol. 112, no. 24, p. 240503, Jun. 2014, doi: 10.1103/PhysRevLett.112.240503.

[103] T. Evans *et al.*, "Fast Bayesian Tomography of a Two-Qubit Gate Set in Silicon," *Physical Review Applied*, vol. 17, no. 2, p. 24068, Feb. 2022, doi: 10.1103/PhysRevApplied.17.024068.

[104] R.-B. Wu, B. Chu, D. H. Owens, and H. Rabitz, "Data-Driven Gradient Algorithm for High-Precision Quantum Control," *Physical Review A*, vol. 97, no. 4, p. 42122, Apr. 2018, doi: 10.1103/PhysRevA.97.042122.

[105] J. Combes *et al.*, "Forest Benchmarking: QCVV Using PyQuil." Accessed: May 29, 2026. [Online]. Available: https://zenodo.org/record/3455847

[106] A. V. Rodionov *et al.*, "Compressed Sensing Quantum Process Tomography for Superconducting Quantum Gates," *Physical Review B*, vol. 90, no. 14, p. 144504, Oct. 2014, doi: 10.1103/PhysRevB.90.144504.

[107] E. Nielsen, J. K. Gamble, K. Rudinger, T. Scholten, K. Young, and R. Blume-Kohout, "Gate Set Tomography," *Quantum*, vol. 5, p. 557, Oct. 2021, doi: 10.22331/q-2021-10-05-557.

[108] S. T. Flammia and Y.-K. Liu, "Direct Fidelity Estimation from Few Pauli Measurements," *Physical Review Letters*, vol. 106, no. 23, p. 230501, Jun. 2011, doi: 10.1103/PhysRevLett.106.230501.

[109] C. J. Wood and J. M. Gambetta, "Quantification and Characterization of Leakage Errors," *Physical Review A*, vol. 97, no. 3, p. 32306, Mar. 2018, doi: 10.1103/PhysRevA.97.032306.

[110] Z. Chen *et al.*, "Measuring and Suppressing Quantum State Leakage in a Superconducting Qubit," *Physical Review Letters*, vol. 116, no. 2, p. 20501, Jan. 2016, doi: 10.1103/PhysRevLett.116.020501.

[111] K. Huang, D. Farfurnik, A. Seif, M. Hafezi, and Y.-K. Liu, "Random Pulse Sequences for Qubit Noise Spectroscopy," *Physical Review Applied*, vol. 23, no. 5, p. 54090, May 2025, doi: 10.1103/PhysRevApplied.23.054090.

[112] C. Tornow, N. Kanazawa, W. E. Shanks, and D. J. Egger, "Minimum Quantum Run-Time Characterization and Calibration via Restless Measurements with Dynamic Repetition Rates," *Physical Review Applied*, vol. 17, no. 6, p. 64061, Jun. 2022, doi: 10.1103/PhysRevApplied.17.064061.

[113] D. T. Lennon *et al.*, "Efficiently Measuring a Quantum Device Using Machine Learning," *npj Quantum Information*, vol. 5, no. 1, p. 79, Sep. 2019, doi: 10.1038/s41534-019-0193-4.

[114] L. J. Fiderer, J. Schuff, and D. Braun, "Neural-Network Heuristics for Adaptive Bayesian Quantum Estimation," *PRX Quantum*, vol. 2, no. 2, p. 20303, Apr. 2021, doi: 10.1103/PRXQuantum.2.020303.

[115] N. Wittler *et al.*, "Integrated Tool Set for Control, Calibration, and Characterization of Quantum Devices Applied to Superconducting Qubits," *Physical Review Applied*, vol. 15, no. 3, p. 34080, Mar. 2021, doi: 10.1103/PhysRevApplied.15.034080.

[116] A. Pasquale, S. Efthymiou, S. Ramos-Calderer, J. Wilkens, I. Roth, and S. Carrazza, "Towards an Open-Source Framework to Perform Quantum Calibration and Characterization." Accessed: May 07, 2025. [Online]. Available: http://arxiv.org/abs/2303.10397

[117] S. R. Jha, S. Chowdhury, G. Rolleri, M. Hays, J. A. Grover, and W. D. Oliver, "JAXQuantum: An Auto-Differentiable and Hardware-Accelerated Toolkit for Quantum Hardware Design, Simulation, and Control." [Online]. Available: https://jaxquantum.org/

[118] Yanjun Ji, M. Roth, D. A. Kreplin, I. Polian, and F. K. Wilhelm, "Data-Efficient Quantum Noise Modeling via Machine Learning," *Physical Review Applied*, vol. 25, no. 3, p. 34051, Mar. 2026, doi: 10.1103/5r9m-y6z6.

[119] F. Li, R. Baptista, and Y. Marzouk, "Expected Information Gain Estimation via Density Approximations: Sample Allocation and Dimension Reduction." Accessed: Jun. 30, 2025. [Online]. Available: http://arxiv.org/abs/2411.08390

[120] E. Goan and C. Fookes, "Bayesian Neural Networks: An Introduction and Survey," in *Case Studies in Applied Bayesian Data Science*, vol. 2259, K. L. Mengersen, P. Pudlo, and C. P. Robert, Eds., Cham: Springer International Publishing, 2020, pp. 45–87. doi: 10.1007/978-3-030-42553-1_3.

[121] L. V. Jospin, H. Laga, F. Boussaid, W. Buntine, and M. Bennamoun, "Hands-On Bayesian Neural Networks—A Tutorial for Deep Learning Users," *IEEE Computational Intelligence Magazine*, vol. 17, no. 2, pp. 29–48, May 2022, doi: 10.1109/MCI.2022.3155327.

[122] J. Arbel, K. Pitas, M. Vladimirova, and V. Fortuin, "A Primer on Bayesian Neural Networks: Review and Debates." Accessed: Apr. 18, 2025. [Online]. Available: http://arxiv.org/abs/2309.16314

[123] I. Kobyzev, S. J. Prince, and M. A. Brubaker, "Normalizing Flows: An Introduction and Review of Current Methods," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 11, pp. 3964–3979, Nov. 2021, doi: 10.1109/TPAMI.2020.2992934.

[124] A. Wehenkel and G. Louppe, "You Say Normalizing Flows I See Bayesian Networks." Accessed: Jun. 28, 2025. [Online]. Available: http://arxiv.org/abs/2006.00866

[125] A. Foster, M. Jankowiak, M. O'Meara, Y. W. Teh, and T. Rainforth, "A Unified Stochastic Gradient Approach to Designing Bayesian-Optimal Experiments." Accessed: May 27, 2025. [Online]. Available: http://arxiv.org/abs/1911.00294

[126] S. Kleinegesse and M. U. Gutmann, "Bayesian Experimental Design for Implicit Models by Mutual Information Neural Estimation." Accessed: Jun. 15, 2026. [Online]. Available: https://arxiv.org/abs/2002.08129

[127] A. Foster, D. R. Ivanova, I. Malik, and T. Rainforth, "Deep Adaptive Design: Amortizing Sequential Bayesian Experimental Design." Accessed: May 27, 2025. [Online]. Available: http://arxiv.org/abs/2103.02438

[128] D. R. Ivanova, A. Foster, S. Kleinegesse, M. U. Gutmann, and T. Rainforth, "Implicit Deep Adaptive Design: Policy-Based Experimental Design without Likelihoods." Accessed: May 27, 2025. [Online]. Available: http://arxiv.org/abs/2111.02329

[129] R. S. Gupta, C. L. Edmunds, A. R. Milne, C. Hempel, and M. J. Biercuk, "Adaptive Characterization of Spatially Inhomogeneous Fields and Errors in Qubit Registers," *npj Quantum Information*, vol. 6, no. 1, pp. 1–10, Jun. 2020, doi: 10.1038/s41534-020-0286-0.

[130] J. H. Guillaume *et al.*, "Introductory Overview of Identifiability Analysis: A Guide to Evaluating Whether You Have the Right Type of Data for Your Modeling Purpose," *Environmental Modelling & Software*, vol. 119, pp. 418–432, Sep. 2019, doi: 10.1016/j.envsoft.2019.07.007.

[131] F.-G. Wieland, A. L. Hauber, M. Rosenblatt, C. Tönsing, and J. Timmer, "On Structural and Practical Identifiability," *Current Opinion in Systems Biology*, vol. 25, pp. 60–69, Mar. 2021, doi: 10.1016/j.coisb.2021.03.005.

[132] A. Raue *et al.*, "Structural and Practical Identifiability Analysis of Partially Observed Dynamical Models by Exploiting the Profile Likelihood," *Bioinformatics*, vol. 25, no. 15, pp. 1923–1929, Aug. 2009, doi: 10.1093/bioinformatics/btp358.

[133] L. Wanika *et al.*, "Structural and Practical Identifiability Analysis in Bioengineering: A Beginner's Guide," *Journal of Biological Engineering*, vol. 18, no. 1, p. 20, Mar. 2024, doi: 10.1186/s13036-024-00410-x.

[134] P. Henning, "Probabilistic Machine Learning," technical report, 2023. [Online]. Available: https://github.com/philipphennig/Probabilistic_ML

[135] K. P. Murphy, *Probabilistic Machine Learning: An Introduction*. MIT Press, 2022. [Online]. Available: http://probml.github.io/book1

[136] K. P. Murphy, *Probabilistic Machine Learning: Advanced Topics*. MIT Press, 2023. [Online]. Available: http://probml.github.io/book2

[137] Andreas Fyrillas, O. Faure, N. Maring, J. Senellart, and N. Belabas, "Scalable Machine Learning-Assisted Clear-Box Characterization for Optimally Controlled Photonic Circuits," *Optica*, vol. 11, no. 3, pp. 427–436, Mar. 2024, doi: 10.1364/OPTICA.512148.

[138] A. Youssry, R. J. Chapman, A. Peruzzo, C. Ferrie, and M. Tomamichel, "Modeling and Control of a Reconfigurable Photonic Circuit Using Deep Learning," *Quantum Science and Technology*, vol. 5, no. 2, p. 25001, Jan. 2020, doi: 10.1088/2058-9565/ab60de.

[139] P. Pathumsoot, A. Chantasri, M. Hajdušek, and R. Van Meter, "Graybox Characterization and Calibration with Finite-Shot Estimation on Superconducting-Qubit Experiments," *New Journal of Physics*, May 2026, doi: 10.1088/1367-2630/ae6df2.

[140] C. M. Bishop, *Pattern Recognition and Machine Learning*. in Information Science and Statistics. New York: Springer, 2006.

[141] A. Youssry and H. I. Nurdin, "Multi-Axis Control of a Qubit in the Presence of Unknown Non-Markovian Quantum Noise," *Quantum Science and Technology*, vol. 8, no. 1, p. 15018, Jan. 2023, doi: 10.1088/2058-9565/aca711.

[142] D. C. McKay, C. J. Wood, S. Sheldon, J. M. Chow, and J. M. Gambetta, "Efficient Z Gates for Quantum Computing," *Physical Review A*, vol. 96, no. 2, p. 22330, Aug. 2017, doi: 10.1103/PhysRevA.96.022330.

[143] M. AbuGhanem, "IBM Quantum Computers: Evolution, Performance, and Future Directions," *The Journal of Supercomputing*, vol. 81, no. 5, p. 687, Apr. 2025, doi: 10.1007/s11227-025-07047-7.

[144] J. M. Chow *et al.*, "Implementing a Strand of a Scalable Fault-Tolerant Quantum Computing Fabric," *Nature Communications*, vol. 5, no. 1, p. 4015, Jun. 2014, doi: 10.1038/ncomms5015.

[145] T. Caneva, T. Calarco, and S. Montangero, "Chopped Random-Basis Quantum Optimization," *Physical Review A*, vol. 84, no. 2, p. 22326, Aug. 2011, doi: 10.1103/PhysRevA.84.022326.

[146] Y. Sung *et al.*, "Non-Gaussian Noise Spectroscopy with a Superconducting Qubit Sensor," *Nature Communications*, vol. 10, no. 1, p. 3715, Sep. 2019, doi: 10.1038/s41467-019-11699-4.

[147] L. M. Norris, G. A. Paz-Silva, and L. Viola, "Qubit Noise Spectroscopy for Non-Gaussian Dephasing Environments," *Physical Review Letters*, vol. 116, no. 15, p. 150503, Apr. 2016, doi: 10.1103/PhysRevLett.116.150503.

[148] G. A. Paz-Silva, L. M. Norris, F. Beaudoin, and L. Viola, "Extending Comb-Based Spectral Estimation to Multiaxis Quantum Noise," *Physical Review A*, vol. 100, no. 4, p. 42334, Oct. 2019, doi: 10.1103/PhysRevA.100.042334.

[149] P. Pathumsoot, M. Hajdušek, and R. Van Meter, "Probabilistic Graybox Characterization of Quantum Devices with Bayesian Neural Networks." Accessed: Feb. 22, 2026. [Online]. Available: http://arxiv.org/abs/2509.24232

[150] J. Tan, L. Xiao, D. Qiu, L. Luo, and P. Mateus, "Distributed Quantum Algorithm for Simon's Problem," *Physical Review A*, vol. 106, no. 3, p. 32417, Sep. 2022, doi: 10.1103/PhysRevA.106.032417.

[151] S. Muralidharan, "The Simulation of Distributed Quantum Algorithms," *The Journal of Supercomputing*, vol. 81, no. 5, p. 645, Mar. 2025, doi: 10.1007/s11227-025-07125-w.

[152] D. Main *et al.*, "Distributed Quantum Computing across an Optical Network Link," *Nature*, vol. 638, no. 8050, pp. 383–388, Feb. 2025, doi: 10.1038/s41586-024-08404-x.

[153] D. Gottesman, T. Jennewein, and S. Croke, "Longer-Baseline Telescopes Using Quantum Repeaters," *Physical Review Letters*, vol. 109, no. 7, p. 70503, Aug. 2012, doi: 10.1103/PhysRevLett.109.070503.

[154] R. Valivarthi *et al.*, "Teleportation Systems Toward a Quantum Internet," *PRX Quantum*, vol. 1, no. 2, p. 20317, Dec. 2020, doi: 10.1103/PRXQuantum.1.020317.

[155] Y.-F. Huang, X.-F. Ren, Y.-S. Zhang, L.-M. Duan, and G.-C. Guo, "Experimental Teleportation of a Quantum Controlled-NOT Gate," *Physical Review Letters*, vol. 93, no. 24, p. 240501, Dec. 2004, doi: 10.1103/PhysRevLett.93.240501.

[156] K.-C. Chang *et al.*, "Quantum Teleportation of a Silicon Nanophotonic CNOT Gate," *Optica Quantum*, vol. 3, no. 4, p. 381, Aug. 2025, doi: 10.1364/OPTICAQ.554577.

[157] T. Matsuo, T. Satoh, S. Nagayama, and R. Van Meter, "Analysis of Measurement-Based Quantum Network Coding over Repeater Networks under Noisy Conditions," *Physical Review A*, vol. 97, no. 6, p. 62328, Jun. 2018, doi: 10.1103/PhysRevA.97.062328.

[158] P. Pathumsoot, T. Matsuo, T. Satoh, M. Hajdušek, S. Suwanna, and R. Van Meter, "Modeling of Measurement-Based Quantum Network Coding on a Superconducting Quantum Processor," *Physical Review A*, vol. 101, no. 5, p. 52301, May 2020, doi: 10.1103/PhysRevA.101.052301.

[159] S. Hill and W. K. Wootters, "Entanglement of a Pair of Quantum Bits," *Physical Review Letters*, vol. 78, no. 26, pp. 5022–5025, Jun. 1997, doi: 10.1103/PhysRevLett.78.5022.

[160] N. Brunner, D. Cavalcanti, S. Pironio, V. Scarani, and S. Wehner, "Bell Nonlocality," *Reviews of Modern Physics*, vol. 86, no. 2, pp. 419–478, Apr. 2014, doi: 10.1103/RevModPhys.86.419.

[161] T. Vértesi, "More Efficient Bell Inequalities for Werner States," *Physical Review A*, vol. 78, no. 3, p. 32112, Sep. 2008, doi: 10.1103/PhysRevA.78.032112.

[162] K. Azuma, K. Tamaki, and H.-K. Lo, "All-Photonic Quantum Repeaters," *Nature Communications*, vol. 6, no. 1, p. 6787, Apr. 2015, doi: 10.1038/ncomms7787.

[163] X. Dong, X. Cao, W.-L. Li, G. Zhang, Z. Peng, and R.-B. Wu, "Quantum Optimal Control Theory for the Shaping of Flying Qubits," *Physical Review Applied*, vol. 23, no. 4, p. 44045, Apr. 2025, doi: 10.1103/PhysRevApplied.23.044045.

[164] P. Pathumsoot *et al.*, "Boosting End-to-End Entanglement Fidelity in Quantum Repeater Networks via Hybridized Strategies," in *2024 International Conference on Quantum Communications, Networking, and Computing (QCNC)*, Kanazawa, Japan: IEEE, Jul. 2024, pp. 49–56. doi: 10.1109/QCNC62729.2024.00054.

[165] P. Pathumsoot, "Inspeqtor." Accessed: May 03, 2026. [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.17748401

[166] "JAX: Composable Transformations of Python+NumPy Programs." [Online]. Available: http://github.com/jax-ml/jax

[167] J. Ansel *et al.*, "PyTorch 2: Faster Machine Learning through Dynamic Python Bytecode Transformation and Graph Compilation," in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*, ACM, Apr. 2024. doi: 10.1145/3620665.3640366.

[168] M. Abadi *et al.*, "TensorFlow, Large-scale Machine Learning on Heterogeneous Systems." Nov. 2015. doi: 10.5281/zenodo.4724125.

[169] R. Vink *et al.*, "Pola-Rs/Polars: Python Polars 1.41.1." Accessed: May 29, 2026. [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.7697217

[170] P. Kidger, "On Neural Differential Equations," Doctoral dissertation, 2021.

[171] DeepMind *et al.*, "The DeepMind JAX Ecosystem." [Online]. Available: http://github.com/google-deepmind

[172] T. Pinder and D. Dodd, "GPJax: A Gaussian Process Framework in JAX," *Journal of Open Source Software*, vol. 7, no. 75, p. 4455, Jul. 2022, doi: 10.21105/joss.04455.

[173] A. Alonso, "Alonfnt/Bayex." Accessed: Oct. 04, 2025. [Online]. Available: https://github.com/alonfnt/bayex

[174] Jonathan Heek *et al.*, "Flax: A Neural Network Library and Ecosystem for JAX." [Online]. Available: http://github.com/google/flax

[175] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A Research Platform for Distributed Model Selection and Training." Accessed: May 29, 2026. [Online]. Available: https://arxiv.org/abs/1807.05118

[176] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A Next-generation Hyperparameter Optimization Framework," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Anchorage AK USA: ACM, Jul. 2019, pp. 2623–2631. doi: 10.1145/3292500.3330701.

[177] E. Bingham *et al.*, "Pyro: Deep Universal Probabilistic Programming," *Journal of Machine Learning Research*, vol. 20, p. 28:1–28:6, 2019, [Online]. Available: http://jmlr.org/papers/v20/18-403.html

[178] D. Phan, N. Pradhan, and M. Jankowiak, "Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro." Accessed: May 05, 2025. [Online]. Available: http://arxiv.org/abs/1912.11554

[179] A. Alexanderian, "A Brief Note on the Bayesian D-optimality Criterion." Accessed: May 14, 2026. [Online]. Available: https://arxiv.org/abs/2212.11466

[180] L. Kühmichel *et al.*, "BayesFlow 2: Multi-Backend Amortized Bayesian Inference in Python." Accessed: May 29, 2026. [Online]. Available: https://arxiv.org/abs/2602.07098

[181] A. Olivier, D. G. Giovanis, B. Aakash, M. Chauhan, L. Vandanapu, and M. D. Shields, "UQpy: A General Purpose Python Package and Development Environment for Uncertainty Quantification," *Journal of Computational Science*, vol. 47, p. 101204, Nov. 2020, doi: 10.1016/j.jocs.2020.101204.

[182] F. Nielsen, "On the Jensen–Shannon Symmetrization of Distances Relying on Abstract Means," *Entropy*, vol. 21, no. 5, p. 485, May 2019, doi: 10.3390/e21050485.

[183] F. Mazurek *et al.*, "Tailored Quantum Device Calibration with Statistical Model Checking." Accessed: Aug. 06, 2025. [Online]. Available: https://arxiv.org/abs/2507.12323

[184] R. S. Gupta, L. C. G. Govia, and M. J. Biercuk, "Integration of Spectator Qubits into Quantum Computer Architectures for Hardware Tune-up and Calibration," *Physical Review A*, vol. 102, no. 4, p. 42611, Oct. 2020, doi: 10.1103/PhysRevA.102.042611.

[185] A. Youssry, G. A. Paz-Silva, and C. Ferrie, "Noise Detection with Spectator Qubits and Quantum Feature Engineering," *New Journal of Physics*, vol. 25, no. 7, p. 73004, Jul. 2023, doi: 10.1088/1367-2630/ace2e4.

[186] N. Kamjam, P. Pathumsoot, and A. Chantasri, "Random-Telegraph Noise Mitigation and Qubit Decoherence in Solid-State Experiments," *Journal of Physics: Conference Series*, vol. 2431, no. 1, p. 12102, Jan. 2023, doi: 10.1088/1742-6596/2431/1/012102.

[187] S. Rojanasirivanit, P. Pathumsoot, and A. Chantasri, "Estimating Unknown Qubit Phase under Telegraph Noises Using Recurrent Neural Network," *Journal of Physics: Conference Series*, vol. 2431, no. 1, p. 12103, Jan. 2023, doi: 10.1088/1742-6596/2431/1/012103.

[188] K. Singh, C. E. Bradley, S. Anand, V. Ramesh, R. White, and H. Bernien, "Mid-Circuit Correction of Correlated Phase Errors Using an Array of Spectator Qubits," *Science*, vol. 380, no. 6651, pp. 1265–1269, Jun. 2023, doi: 10.1126/science.ade5337.

Appendix A Detailed Derivation

A.1 Data variance: An alternative derivation

From Bishop's Equation (1.87) [140], the model prediction of input x is denoted as $y(x)$, and the observation from experiment is a random variable t. The expectation of $(y(x)-t)^2$ with respect to the all possible input and output is

$$\mathbb{E}[L] = \int\int (y(x)-t)^2 p(x,t)dxdt. \tag{142}$$

From Bishop's Equation (1.89), the expected (true) value of the observation t given x is,

$$h(x) = \mathbb{E}[t|x] = \int tp(t|x)dt. \tag{143}$$

Now, consider the expansion of the square error,

$$\begin{aligned}(y(x)-t)^2 &= (y(x)-\mathbb{E}[t|x]+\mathbb{E}[t|x]-t)^2 &(144.1)\\ &= (y(x)-\mathbb{E}[t|x])^2 + 2(y(x)-\mathbb{E}[t|x])(\mathbb{E}[t|x]-t) &(144.2)\\ &\quad +(\mathbb{E}[t|x]-t)^2, &(144.3)\end{aligned}$$

then compute the expectation with respect to t conditioned on a single input x,

$$\begin{aligned}\mathbb{E}(t)[(y(x)-t)^2] &= \int (y(x)-t)^2 p(t|x)dt, &(145.1)\\ &= (y(x)-\mathbb{E}[t|x])^2 &(145.2)\\ &\quad +2(y(x)-\mathbb{E}[t|x])\left(\underbrace{\int (\mathbb{E}[t|x]-t)p(t|x)dt}_{\text{A}}\right) &(145.3)\\ &\quad + \int (\mathbb{E}[t|x]-t)^2 p(t|x)dt. &(145.4)\end{aligned}$$

Note that $y(x)$ and $\mathbb{E}[t|x]$ are independent of the random variable t by definition. From Equation 144, consider the term A,

$$\int (\mathbb{E}[t|x]-t)p(t|x)dt = \mathbb{E}[t|x] - \int tp(t|x)dt = 0, \tag{146}$$

where we use the definition of expected value of t in Equation 143. Therefore, the Equation 145 is reduced to

$$\mathbb{E}(t)[(y(x)-t)^2] = (y(x)-\mathbb{E}[t|x])^2 + \underbrace{\int (\mathbb{E}[t|x]-t)^2 p(t|x)dt}_{\text{Conditional Variance}}. \tag{147}$$

Note that the last term is the conditional variance of t given the value of x by definition. So, we denote the conditional variance as

$$\text{Var}[t|x] = \int (\mathbb{E}[t|x]-t)^2 p(t|x)dt. \tag{148}$$

We then take the expectation over random variable x, and define $h(x) = \mathbb{E}[t|x]$ as was done by Bishop,

$$\begin{aligned}
\mathbb{E}_x[\mathbb{E}_t[L]] = \mathbb{E}_{x,t}[L] &= \int \mathbb{E}_t[(y(x)-t)^2]p(x)dx && (149.1)\\
&= \int (y(x)-h(x))^2 p(x)dx && (149.2)\\
&\quad + \int \text{Var}[t|x]p(x)dx, && (149.3)
\end{aligned}$$

which results in equation (3.37) in Bishop. Next, we know that the model is trained on a particular dataset $\mathcal{D}$, so in practice, the model depends on $\mathcal{D}$ as $y(x;\mathcal{D})$. Let's consider the following,

$$\begin{aligned}
(y(x;\mathcal{D})-h(x))^2 &= (y(x;\mathcal{D})-\mathbb{E}_{\mathcal{D}}[y(x;\mathcal{D})] + \mathbb{E}_{\mathcal{D}}[y(x;\mathcal{D})]-h(x))^2 && (150.1)\\
&= (y(x;\mathcal{D})-\mathbb{E}_{\mathcal{D}}[y(x;\mathcal{D})])^2 && (150.2)\\
&\quad +(\mathbb{E}_{\mathcal{D}}[y(x;\mathcal{D})]-h(x))^2 && (150.3)\\
&\quad +2(y(x;\mathcal{D})-\mathbb{E}_{\mathcal{D}}[y(x;\mathcal{D})])(\mathbb{E}_{\mathcal{D}}[y(x;\mathcal{D})]-h(x)), && (150.4)
\end{aligned}$$

and the last term will be zero when we take the expectation over the dataset $\mathcal{D}$, we get,

$$\begin{aligned}
\mathbb{E}_{\mathcal{D}}[(y(x;\mathcal{D})-h(x))^2] &= \underbrace{(y(x;\mathcal{D})-\mathbb{E}_{\mathcal{D}}[y(x;\mathcal{D})])^2}_{\text{Bias}^2} && (151.1)\\
&\quad +\underbrace{\mathbb{E}_{\mathcal{D}}\left[(\mathbb{E}_{\mathcal{D}}[y(x;\mathcal{D})]-h(x))^2\right]}_{\text{Variance}}. && (151.2)
\end{aligned}$$

We now take the expectation over dataset $\mathcal{D}$ on Equation 149,

$$\mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_{x,t}[L]\right] = \int \mathbb{E}_{\mathcal{D}}[(y(x)-h(x))^2]p(x)dx + \mathbb{E}_{\mathcal{D}}\left[\int \text{Var}[t|x]p(x)dx\right]. \tag{152}$$

We then substitute the result from Equation 151 and note that the integral of conditional variance over x does not depend on the choice of dataset $\mathcal{D}$,

$$\mathbb{E}_{\mathcal{D},x,t|x}[L] = \int \text{Bias}^2 p(x)dx \tag{153.1}$$

$$+ \int \text{Variance } p(x)dx \tag{153.2}$$

$$+ \int \text{Var}[t|x]p(x)dx, \tag{153.3}$$

where the expectations on the LHS are over the possible choice of dataset $\mathcal{D}$, test input x, and the observation t given the test input x. We can consider only a particular value of input x by taking the derivative of x,

$$\mathbb{E}_{\mathcal{D},t|x}[L] = \text{Bias}^2 + \text{Variance} + \text{Var}[t|x], \tag{154.1}$$

$$0 = \mathbb{E}_{\mathcal{D},t|x}[L] - \text{Bias}^2 - \text{Variance} - \text{Var}[t|x], \tag{154.2}$$

which recovers the result that agrees with other references, except that the variance of the data has now become the conditional variance instead.

Consider our context, the minimum value of expected $\mathcal{L}_{\text{SE[E]}}$ is,

$$\mathbb{E}_{\mathcal{D},t|x}\left[\mathcal{L}_{\text{SE[E]}}\right] \geqslant \frac{1}{18}\sum_{\rho_0,k}\text{Var}\left[\mathbb{E}\left[\hat{O}_k\right]_{\rho_0} \mid \mathbf{\Theta}\right] \tag{155.1}$$

$$\geqslant \frac{2}{3n}, \tag{155.2}$$

which is the usual result we expected.

A.2 The performance upper bound in terms of the data variance

From the Cauchy-Schwarz inequality,

$$\left(\sum_i v_i u_i\right)^2 \leqslant \left(\sum_i u_i^2\right)\left(\sum_i v_i^2\right), \tag{156}$$

We define the following,

$$u_i = \langle\hat{O}\rangle^{\text{model}}_{j,k,m} - \langle\hat{O}\rangle^{\text{exp}}_{j,k,m}, v_i = \alpha_{j,k}\beta_{m,j}, \tag{157}$$

to get

$$\sum_i v_i^2 = 6, \tag{158}$$

and

$$\sum_i v_i u_i = 12(\bar{F}_{\text{model}} - \bar{F}_{\text{exp}}). \tag{159}$$

Note that $\langle \hat{O} \rangle^{\text{exp}}_{j,k,m}$ is the true (infinite-shot) expectation value of observable, where the finite-shot expectation value is denoted as $\mathbb{E}^{\text{exp}}_{j,k,m}$ which is a random variable conditioned on input $\boldsymbol{\Theta}$. From the $\mathcal{L}_{\text{SE[E]}}$ definition, we reindex j, k, m to i and rewrite it in terms of u_i as follows,

$$\mathcal{L}_{\text{SE[E]}} = \frac{1}{18} \sum_{j,k,m} \left(\langle \hat{O} \rangle^{\text{model}}_{j,k,m} - \mathbb{E}^{\text{exp}}_{j,k,m} \right)^2 \tag{160.1}$$

$$= \frac{1}{18} \sum_i \left(\langle \hat{O} \rangle^{\text{model}}(i) - \langle \hat{O} \rangle^{\text{exp}}(i) + \langle \hat{O} \rangle^{\text{exp}}(i) - \mathbb{E}^{\text{exp}}(i) \right)^2 \tag{160.2}$$

$$= \frac{1}{18} \sum_i \left(u_i + \underbrace{\langle \hat{O} \rangle^{\text{exp}}(i) - \mathbb{E}^{\text{exp}}(i)}_{\varepsilon_i} \right)^2 \tag{160.3}$$

$$= \frac{1}{18} \sum_i (u_i^2 - 2u_i\varepsilon_i + \varepsilon_i^2). \tag{160.4}$$

Rearranging the above in the more useful form, we get,

$$\sum_i u_i^2 = 18\mathcal{L}_{\text{SE[E]}} + \sum_i (2u_i\varepsilon_i - \varepsilon_i^2). \tag{161}$$

Then, we substitute the results back into the CS inequality.

$$\left(\bar{F}_{\text{model}} - \bar{F}_{\text{exp}}\right)^2 \leqslant \frac{6}{12^2} \left(18\mathcal{L}_{\text{SE[E]}} + \sum_i (2u_i\varepsilon_i - \varepsilon_i^2) \right). \tag{162}$$

Now we take the expectation over the distribution of the observed (finite-shot) expectation value y given input x, we simplify the cross term first as follows,

$$\mathbb{E}_{y|x} \left[\sum_i 2u_i\varepsilon_i \right] = \sum_i 2u_i \mathbb{E}_{y|x}[\varepsilon_i] \tag{163.1}$$

$$= \sum_i 2u_i \mathbb{E}_{y|x} \left[\langle \hat{O} \rangle^{\text{exp}}(i) - \mathbb{E}^{\text{exp}}(i) \right] \tag{163.2}$$

$$= \sum_i 2u_i \left(\langle \hat{O} \rangle^{\text{exp}}(i) - \mathbb{E}_{y|x}[\mathbb{E}^{\text{exp}}(i)] \right) \tag{163.3}$$

$$= 0, \tag{163.4}$$

where note that u_i is constant with respect to the experimental noise. So the inequality becomes,

$$\left(\bar{F}_{\text{model}} - \bar{F}_{\text{exp}}\right)^2 \leqslant \frac{6}{12^2}\left(18\mathbb{E}_{y|x}\left[\mathcal{L}_{\text{SE[E]}}\right] - \sum_i \underbrace{\mathbb{E}_{y|x}\left[\varepsilon_i^2\right]}_{\text{conditional variance}}\right), \tag{164.1}$$

$$\leqslant \frac{6}{12^2}\left(18\mathbb{E}_{y|x}\left[\mathcal{L}_{\text{SE[E]}}\right] - \sum_i \text{Var}[\mathbb{E}^{\text{exp}}(i) \mid \boldsymbol{\Theta}]\right), \tag{164.2}$$

$$\leqslant \frac{3}{4}\left(\mathbb{E}_{y|x}\left[\mathcal{L}_{\text{SE[E]}}\right] - \frac{2}{3n}\right). \tag{164.3}$$

we now compute the expectation over the choice of dataset $\mathcal{D}$ to recover the definition of expected loss as follows,

$$\mathbb{E}_{\mathcal{D}}\left[\left(\bar{F}_{\text{model}} - \bar{F}_{\text{exp}}\right)^2\right] \leqslant \frac{3}{4}\left(\mathbb{E}_{\mathcal{D},y|x}\left[\mathcal{L}_{\text{SE[E]}}\right] - \frac{2}{3n}\right). \tag{165}$$

From the Jensen inequality, consider $|\bar{F}_{\text{model}} - \bar{F}_{\text{exp}}|$ as a random variable, we get,

$$\mathbb{E}_{\mathcal{D}}[|\bar{F}_{\text{model}} - \bar{F}_{\text{exp}}|] \leqslant \sqrt{\frac{3}{4}\left(\mathbb{E}_{\mathcal{D},y|x}\left[\mathcal{L}_{\text{SE[E]}}\right] - \frac{2}{3n}\right)} \tag{166}$$

A.3 Rotating Frame

In this section, we derive the Hamiltonian H in the rotating frame using the Unitary operator U. From the unitary transformation,

$$H \to UHU^\dagger + i\dot{U}U^\dagger \tag{167}$$

Let us define $a_0 = 2\pi\omega_q$ and $a_1 = 2\pi\Omega$

$$H(t) = \frac{a_0}{2}\hat{\sigma}_z + a_1 s(t)\hat{\sigma}_x \tag{168}$$

With the frame

$$U_0 = \exp\left(i\frac{a_0}{2}\hat{\sigma}_z t\right) \tag{169}$$

We have

$$\dot{U}_0 = \frac{\partial}{\partial t}U_0 = i\frac{a_0}{2}\hat{\sigma}_z U_0, \tag{170}$$

and

$$i\dot{U}_0 U_0^\dagger = -\frac{a_0}{2}\hat{\sigma}_z. \tag{171}$$

And also

$$U_0 H U_0^\dagger = U_0\Big(\frac{a_0}{2}\hat{\sigma}_z + a_1 s(t)\hat{\sigma}_x\Big)U_0^\dagger \tag{172.1}$$

$$= \frac{a_0}{2}\sigma_z + a_1 s(t) U_0 \hat{\sigma}_x U_0^\dagger \tag{172.2}$$

Consider the term $U_0\hat{\sigma}_x U_0^\dagger$ and define $a' = \frac{a_0}{2}t$, we have,

$$U_0\hat{\sigma}_x U_0^\dagger = (\cos(a')I + i\sin(a')\hat{\sigma}_z)\hat{\sigma}_x(\cos(a')I - i\sin(a')\hat{\sigma}_z) \tag{173.1}$$

$$= \cos^2(a')\hat{\sigma}_x - i\cos(a')\sin(a')\hat{\sigma}_x\hat{\sigma}_z \tag{173.2}$$

$$+ i\sin(a')\cos(a')\hat{\sigma}_z\hat{\sigma}_x + \sin^2(a')\hat{\sigma}_z\hat{\sigma}_x\hat{\sigma}_z \tag{173.3}$$

We use the fact that $\hat{\sigma}_y = i\hat{\sigma}_x\hat{\sigma}_z$ and $\hat{\sigma}_x\hat{\sigma}_z = -\hat{\sigma}_z\hat{\sigma}_x$, to simplify above to

$$U_0\hat{\sigma}_x U_0^\dagger = \cos^2(a')\hat{\sigma}_x - 2\cos(a')\sin(a')\hat{\sigma}_y - \sin^2(a')\hat{\sigma}_x \tag{174.1}$$

$$= (\cos^2(a') - \sin^2(a'))\hat{\sigma}_x - 2\cos(a')\sin(a')\hat{\sigma}_y. \tag{174.2}$$

Then with the following identities,

$$\cos(2x) = \cos^2(x) - \sin^2(x) \tag{175.1}$$

$$\sin(2x) = 2\cos(x)\sin(x), \tag{175.2}$$

we get

$$U_0\hat{\sigma}_x U_0^\dagger = \cos(2a')\hat{\sigma}_x - \sin(2a')\hat{\sigma}_y \tag{176}$$

Finally we arrive with the Hamiltonian H in the rotating frame,

$$H_{\text{rot}} = U_0 H U_0^\dagger + i\dot{U}_0 U_0^\dagger = a_1 s(t)\big(\cos(a_0 t)\hat{\sigma}_x - \sin(a_0 t)\hat{\sigma}_y\big) \tag{177}$$

Appendix B List of Papers and Presentations

B.1 Peer-Reviewed Journals

1. Pathumsoot, P., Matsuo, T., Satoh, T., Hajdušek, M., Suwanna, S., & Van Meter, R. (2020). Modeling of measurement-based quantum network coding on a superconducting quantum processor. Physical Review A, 101(5), 052301. https://doi.org/10.1103/PhysRevA.101.052301
2. Pathumsoot, P., Chantasri, A., Hajdušek, M., & Van Meter, R. (2026). Graybox characterization and calibration with finite-shot estimation on superconducting-qubit experiments. New Journal of Physics. https://doi.org/10.1088/1367-2630/ae6df2

B.2 Conference Papers (Peer-Reviewed)

1. Chianvichai, P., Pathumsoot, P., & Suwanna, S. (2024). Enhancing Quantum Network Establishment Through Multi-Objective Genetic Algorithm. 2024 International Conference on Quantum Communications, Networking, and Computing (QCNC), 85–90. https://doi.org/10.1109/QCNC62729.2024.00022
2. Kamjam, N., Pathumsoot, P., & Chantasri, A. (2023). Random-telegraph noise mitigation and qubit decoherence in solid-state experiments. Journal of Physics: Conference Series, 2431(1), 012102. https://doi.org/10.1088/1742-6596/2431/1/012102
3. Pathumsoot, P., Benchasattabuse, N., Satoh, R., Hajdusek, M., Van Meter, R., & Suwanna, S. (2021). Optimizing Link-Level Entanglement Generation in Quantum Networks with Unequal Link Lengths. 2021 25th International Computer Science and Engineering Conference (ICSEC), 179–184. https://doi.org/10.1109/ICSEC53205.2021.9684634
4. Pathumsoot, P., Tansuwannont, T., Benchasattabuse, N., Satoh, R., Hajdušek, M., Chaiwongkhot, P., Suwanna, S., & Van Meter, R. (2024). Boosting End-to-End Entanglement Fidelity in Quantum Repeater Networks via Hybridized Strategies. 2024 International Conference on Quantum Communications, Networking, and Computing (QCNC), 49–56. https://doi.org/10.1109/QCNC62729.2024.00054
5. Rojanasirivanit, S., Pathumsoot, P., & Chantasri, A. (2023). Estimating unknown qubit phase under telegraph noises using recurrent neural network. Journal of Physics: Conference Series, 2431(1), 012103. https://doi.org/10.1088/1742-6596/2431/1/012103

6. Satoh, R., Hajdusek, M., Benchasattabuse, N., Nagayama, S., Teramoto, K., Matsuo, T., Metwalli, S. A., Pathumsoot, P., Satoh, T., Suzuki, S., & Meter, R. V. (2022). QuISP: A Quantum Internet Simulation Package. 2022 IEEE International Conference on Quantum Computing and Engineering (QCE), 353–364. https://doi.org/10.1109/QCE53715.2022.00056
7. Supasil, J., Pathumsoot, P., & Suwanna, S. (2021). Simulation of implementable quantum-assisted genetic algorithm. Journal of Physics: Conference Series, 1719(1), 012102. https://doi.org/10.1088/1742-6596/1719/1/012102

B.3 Manuscripts under Review and In Preparation

1. Pathumsoot, P., Hajdušek, M., & Meter, R. V. (2025). Probabilistic Graybox Characterization of Quantum Devices with Bayesian Neural Networks (arXiv:2509.24232). arXiv. https://doi.org/10.48550/arXiv.2509.24232

Appendix B.3 Glossary

AGF – **Average Gate Fidelity**: A metric that measure the closeness of quantum channel to target Unitary operator. 18, 18, 18, 18, 37, 37, 37, 37, 104, 109, 109, 112, 138, 146, 147, 147, 147, 147, 147, 147, 149, 150, 151, 163, 166

BNN – **Bayesian Neural Network**: A Neural Network that model parameters are sample from distributions instead of being fixed values. 25, 25, 35, 61, 61, 61, 61, 61, 69, 69, 69, 120, 120, 120, 121, 121, 122, 122, 153, 154, 161, 164

BOED – **Bayesian Optimal Experimental Design**: An approach to design an experiment ahead of time such that the experimetnal design maxmize information gain. 24, 25, 27, 28, 66, 69, 69, 69, 69, 121, 125, 129, 129, 153, 161, 161, 163, 163, 164, 165, 166, 166, 166, 166

DNN – **Deep Neural Network** 25, 25, 25, 25, 25, 31, 61, 61, 61, 69, 121, 123

EIG – **Expected Information Gain**: The expected inforamtion gain is an expected reduction in entropy of the system after updating with the experiment with some design. 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 69, 69, 69, 69, 69, 69, 69, 69, 69, 69, 69, 125, 125, 125, 125, 125, 125, 126, 126, 126, 126, 126, 151, 151, 152, 153, 153, 153, 153, 154, 155, 155, 159, 160, 160, 161, 161, 161, 161, 161, 161, 162, 162, 163, 165, 165, 165

JSD – **Jensen-Shannon Divergence**: A symmetric version of KL divergence. 69, 125, 125, 125, 149, 151, 156, 157, 160, 161, 161, 165

MLP – **Multi-Layer Perceptron**: A simple Neural Network architecture that apply multiple Dense layers to the input. 36, 121

PGM – **Probabilistic Graybox Model**: Probabilistic version of Graybox predictive model. 61, 61, 63, 66, 66, 66, 66, 139, 139, 140, 140, 146, 147, 147, 147, 147, 148, 148, 149, 149, 149, 149, 149, 150, 151, 151, 151, 151, 152, 152, 153, 161, 162, 163, 164, 164

PML – **Probabilistic Machine Learning**: A machine learning approach that learn from the observed data by using Bayesian Inference. 28, 28, 28, 28, 28, 28

SGM – **Statistic Graybox Model**: Statistical version of Graybox predictive model. 61, 63, 66, 66, 66, 67, 114, 138, 139, 140, 144, 147, 147, 147, 147, 148, 148, 149, 149, 149, 149, 149, 150, 151, 151, 151, 151, 151, 163, 164

SML – **Statistical Machine Learning**: A machine learning approach that learn from the observed data by minimizing some loss function. 28, 28, 28, 28, 28, 28, 28

“Peace cannot be kept by force; it can only be achieved by understanding.”

— Albert Einstein