DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

# A Quantum Software Experimentation Interface to a Superconducting Quantum Computer

Teodor-Adrian Mihaescu

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

# A Quantum Software Experimentation Interface to a Superconducting Quantum Computer

# Eine Quanten-Software-Experimentier-Schnittstelle zu einem supraleitenden Quantencomputer

| | |
|---|---|
| Author: | Teodor-Adrian Mihaescu |
| Examiner: | Prof. Dr. rer. nat. Christian Mendl |
| Assistant advisor: | M. Sc. Martin Knudsen |
| Submission Date: | July 9th, 2024 |

I hereby declare that this thesis is entirely the result of my own work except where other-
wise indicated. I have only used the resources given in the list of references.

July 9th, 2024 Teodor-Adrian Mihaescu

# Acknowledgments

I would like to thank the following people for helping out with the organizational, research, and implementation parts of this project:
My advisor, **Martin Knudsen**, for all his patience, guidance, and support throughout all the stages of this project.
**Prof. Dr. Christian Mendl**, for his valuable insights and feedback on the work done.
The **Walther-Meißner-Institute team**, for providing the necessary infrastructure and resources for testing the implementation on real research-grade quantum hardware.

*"If you are not completely confused by quantum mechanics, you do not understand it."*

*-John Archibald Wheeler*

# Abstract

In the context of quantum hardware needing to operate in highly controlled environments (such as laboratories and research centers), networking communication is essential. This information exchange usually takes place between the quantum development toolkits and the actual quantum hardware and simulators, that execute the declared quantum circuits and algorithms. This thesis proposes and explores a solution to this topic, a quantum software experimentation interface between the quantum development Python library qib (developed at the Technical University Munich) and the simulators and superconducting quantum hardware provided by the infrastructure available at the Walther-Meißner-Institute. It further offers analysis, testing, and architectural details about the implementation, providing insights into the challenges, proposed solutions, and potential future extensions of such an interface, to serve as a guide for future developers and researchers willing to implement or extend a similar infrastructure. As a proof of concept, the implemented interface was successfully used to run an iterative hybrid quantum-classical algorithm that converged on an actual quantum processor.

# Contents

Part I.

# Introduction and Background Knowledge

# 1. Introduction

Quantum Computing represents a significant leap from classical computing, harnessing the principles of quantum mechanics to process information. Unlike classical bits, quantum bits (qubits) can exist simultaneously in multiple states, enabling quantum computers to solve certain complex problems much faster than their classical counterparts.

In recent years, quantum computing has transitioned from a largely theoretical field to one with practical applications and active hardware development. An area of particular growth and challenge within quantum computing is quantum software – the development of algorithms, protocols, and tools for programming and utilizing quantum computers. Unlike classical software, quantum software must be compatible with the unique behaviors and constraints of quantum mechanics, making its development a complex and rapidly evolving field.

This thesis will provide some context-related background knowledge on the current state of quantum software development, particularly in the area of network communication between quantum software development frameworks and superconducting quantum computers or quantum simulators. It will explore the limitations, already-proposed solutions, and potential research trajectories in this area, and most importantly include detailed insights on the implementation process of a quantum software development backend interface.

## 1.1. Implementation Context

### 1.1.1. The Walther-Meißner-Institute (WMI)

The Walther-Meißner-Institute (WMI), with its focus on quantum computing research, primarily emphasizes the hardware aspect of this technology. One of its research fields focuses on the development aspect of superconducting quantum computers. WMI's facilities and expertise in quantum hardware provide a fruitful setting for exploring the practical aspects of quantum computing. They also provide the hardware environment for this thesis's implementation part, with their superconducting quantum computer setting, used for different kinds of quantum experiments, as well as quantum applications.

### 1.1.2. The qib Python Package

On the software side, the qib Python package facilitates an academic toolkit for quantum software development and experimentation. Developed at the Technical University of Munich, qib offers a software development and experimentation framework for quantum algorithms, providing researchers and developers with the resources to create and test quantum programs. This package aims to bridge the gap between theoretical quantum computing and practical quantum algorithm implementations. While still in its early development stages, it lacks certain features that such a toolkit should provide, integration with various quantum hardware or simulator backends being one of these missing features, and also representing the main focus for this thesis.

## 1.2. Thesis Goals

The core goal of this thesis involves extending the qib quantum development toolkit. The objective is to enable qib to interface with various backends, particularly focusing on one of the superconducting quantum computers available at the WMI, and other simulators. This extension aims to streamline the process of running and testing quantum algorithms on actual quantum hardware, thereby fostering a more cohesive integration of software and hardware in the quantum computing domain.

Further splitting this core goal, the thesis will be structured and defined by the following sub-goals:

1. **Interfacing Quantum Software with Hardware:** Investigating the interfaces between quantum software development toolkits and quantum hardware. A significant focus will be on implementing and performing these interactions over web APIs, an essential aspect of Quantum-as-a-Service (QaaS) platforms.

2. **Implementing a Quantum Software Interface for qib:** Expanding a Python quantum software toolkit to support communication with different quantum backends, thereby broadening its applicability and utility in the quantum computing ecosystem.

3. **Analyzing Implementation Results:** Testing, and analyzing results of the implementation at goal 2, comparing it with other existent solutions in the context of architecture, performance, and integration.

This first part (Part I) of the thesis sets the foundation for understanding the current developments in quantum computing and quantum software. It aims to provide a context-related overview, offering insights into the evolution, current state, and challenges in this field (i.e. goal 1). It also introduces theoretical concepts and practical tools, which will be instrumental in the implementation part of this thesis. The second part (Part II) delves deeper into the implementation aspect of the subject at hand (i.e. goal 2). In the end, part 3 (Part III) aims to test, analyze, and draw conclusions based on the resulting data, covering the 3rd, and final goal of this thesis.

# 2. Quantum Computing

## 2.1. Quantum Theory Background

Quantum Computing [40], an emerging field at the interface of quantum physics and computer science, aims to address complex problems using quantum algorithms that are challenging for classical computers to solve. A critical aspect of quantum computing's capability lies in specific quantum properties: qubits, superposition, entanglement, and quantum gate operations.

While quantum computers share the concept of bits with classical computers, a quantum bit (qubit) can exist in both states (0 and 1) simultaneously, thanks to superposition. This unlocks massive parallel processing capabilities. Entangled states, like Bell states [6], are another key concept where linked qubits are intricately connected, enabling features like quantum teleportation. While several other quantum computational models exist ([8], [1]), this thesis focuses on the most common and intuitive one: the qubit gate-based model [33]. This approach relies on sequences of quantum gates to perform computations (similar to how classical computers use gates in their circuits) and qubits (as the fundamental unit of quantum information, capable of existing in a superposition of both 0 and 1 states simultaneously, similar to how classical computers use bits for storing data in binary format).

Quantum gates, similar to logic gates in classical computing, are operations that can be applied on a set of qubits in order to change their quantum state. They are usually represented as matrices, and are reversible and unitary, meaning they preserve the total probability of a system. Quantum circuits represent a sequence of quantum gates that execute quantum computations, which are inherently parallel and reversibly modeled. Gate-based quantum circuits act on qubits, much like classical circuits act on normal bits, but their operations are uniquely influenced by the principles of quantum mechanics. Unlike classical logic gates which produce definitive outcomes, quantum gates manipulate the probabilities of a qubit's state. Fundamental quantum gates include the Pauli X, Y, Z gates, the Hadamard gate (H gate), and the controlled gates (CX, CY, CZ gates) for two-qubit operations. For some of the fundamental thesis-relevant quantum gate representations and their mode of operation see Table 2.1.

| **Gate** | **Circuit Representation** | **Matrix Representation** |
|---|---|---|
| $(I)$ **Identity Gate:** A single-qubit quantum gate that leaves the qubit state unchanged. | $-\boxed{I}-$ | $I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ |
| $(X)$ **Pauli-X Gate:** A single-qubit quantum gate that rotates the qubit state by $\pi$ radians (180°) about the x-axis. | $-\boxed{X}-$ | $X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ |
| $(Y)$ **Pauli-Y Gate:** A single-qubit quantum gate that rotates the qubit state by $\pi$ radians (180°) about the y-axis. | $-\boxed{Y}-$ | $Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$ |
| $(Z)$ **Pauli-Z Gate:** A single-qubit quantum gate that rotates the qubit state by $\pi$ radians (180°) about the z-axis. | $-\boxed{Z}-$ | $Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$ |
| $(H)$ **Hadamard Gate:** A single-qubit quantum gate that rotates the qubit state about the x+z-axis, changing the computation basis from $\|0\rangle, \|1\rangle$ to $\|+\rangle, \|-\rangle$ and vice-versa. | $-\boxed{H}-$ | $H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$ |
| $(SX)$ **Square Root of X Gate:** A single-qubit quantum gate that rotates the qubit state by $\sqrt{X}$ about the x-axis. | $-\boxed{\sqrt{X}}-$ | $SX = \begin{pmatrix} 1+i & 1-i \\ 1-i & 1+i \end{pmatrix}$ |
| $(RX)$ **Rotation-X Gate:** A single-qubit quantum gate that rotates the qubit state by a given angle $\theta$ (in radians) about the x-axis. | $-\boxed{RX(\theta)}-$ | $RX(\theta) = \begin{pmatrix} \cos(\frac{\theta}{2}) & -i\sin(\frac{\theta}{2}) \\ -i\sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{pmatrix}$ |
| $(RY)$ **Rotation-Y Gate:** A single-qubit quantum gate that rotates the qubit state by a given angle $\theta$ (in radians) about the y-axis. | $-\boxed{RY(\theta)}-$ | $RY(\theta) = \begin{pmatrix} \cos(\frac{\theta}{2}) & -\sin(\frac{\theta}{2}) \\ -\sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{pmatrix}$ |

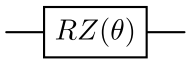
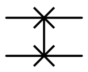
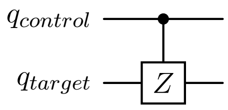
| | | |
|---|---|---|
| $(RZ)$ **Rotation-Z Gate:** A single-qubit quantum gate that rotates the qubit state by a given angle $\theta$ (in radians) about the z-axis. |  | $RZ(\theta) = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix}$ |
| $(iSWAP)$ **iSwap Gate:** A 2-qubit quantum gate that swaps the states of the qubits, and phases the $\lvert 01\rangle$ and $\lvert 10\rangle$ amplitudes by $i$. |  | $iSWAP = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & i & 0 \\ 0 & i & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$ |
| $(CZ)$ **Controlled-Z Gate:** A single-qubit quantum gate that applies a Z gate to the target qubit only when the state of the control qubit equals $\lvert 1\rangle$. |  | $CZ = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$ |

Table 2.1.: Context-relevant quantum gates representations and mode of operation. Also, refer to [40] and [49].

Hardware-wise there are many ways of realizing quantum information processing devices, also known as quantum computers. These include: optical techniques (i.e. electromagnetic radiation) [31], schemes based on methods of trapping atoms, such as ion traps [17], [34], neutral atom traps [21], or even nuclear magnetic resonance (NMR) [11]. However, one of the most promising and widely used approaches is based on superconducting circuits [32]. Superconducting quantum computing is a rapidly evolving field, with many companies and research labs working on developing quantum computers based on superconducting circuits. Some of the most prominent ones are IBM, Google, Rigetti, and Intel. This paper centers on superconducting quantum computing, as the development part of this thesis mainly revolves around building a communication interface for such a quantum computer.

## 2.2. Superconducting Quantum Computational Environments

As mentioned in section 2.1, there are many ways of realizing quantum hardware, and more possibilities are yet to be explored. Although not the main focus of this thesis, it is still good to get a general understanding of how the quantum hardware, which lies at the other end of the communication flow, is implemented and behaves. In the scope of this thesis, the actual WMI quantum hardware that has been used is implemented using superconducting quantum computational mechanisms [32].

This section will offer a quick overview of how this computational paradigm works, based on the 5 main DiVincenzo criteria [16] used for realizing quantum information processing hardware. Thus, a further explanation of how different components (such as qubits, gates, circuits, etc.) of a superconducting quantum computing environment work and are implemented will be provided.

Since the mathematical concepts and actual physical realization principles are out of the scope of this thesis, the information provided in this section will be rather superficial. For a deeper understanding of the presented topics, please refer to (P. Krantz, et. al.) [32], which was used as the basis for the provided explanations.

Although partially out-of-scope, this chapter is particularly relevant in light of the following two arguments:

1. Since this whole process requires a lot of precision, control, and advanced physical systems, **quantum devices need to be centrally available**, in highly controlled environments, such as laboratories. This is one of the main reasons why distributed infrastructures (i.e. cloud) are needed to access these quantum devices remotely, and why the communication between quantum software environments (usually realized on classical machines) and quantum hardware is so important.

2. Since the entire development pipeline for such a system undergoes multiple complex stages (from the development of the quantum algorithms to the actual execution, taking place on the quantum hardware), having a general undestanding of the hardware implementation of the system is beneficial for understanding the system and its limitations as a whole. For example, certain noise or calibration errors might propagate and lead to unexpected or unwanted results.

### 2.2.1. Qubits Characterization

A qubit, the fundamental unit of quantum information, requires clear definition of its states ($|0\rangle$ and $|1\rangle$) and well-characterized properties for scalability. Superconducting qubits exploit quantum effects in nanoscale circuits to encode information. These lithographically-defined circuits mimic atoms with quantized energy levels, controllable through Josephson junctions (see Figure 2.1). Different designs (charge, flux, etc.) offer trade-offs between noise sensitivity and operational parameters.

### 2.2.2. Qubits Initialization

Similar to classical computing, quantum systems should be able to initialize registers (i.e. qubits' state) to a known value before the start of the computation. This also guarantees that quantum error correction can be applied to a newly available set of qubits in a state of low entropy. There are two main approaches to performing this step:

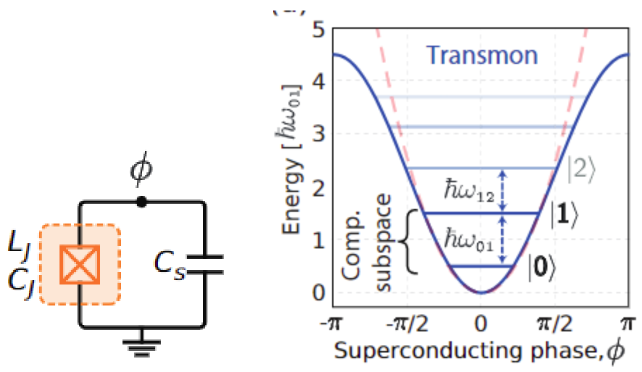


Figure 2.1.: (a) Josephson qubit circuit, in which the nonlinear inductance $L_J$ (depicted by the Josephson subcircuit within the dashed orange box) is paralleled by a capacitance, $C_s$.
(b) The Josephson inductance transforms the quadratic energy potential (dashed red) into a sinusoidal form (solid blue), resulting in non-equidistant energy levels.
***(Source: A Quantum Engineer's Guide to Superconducting Qubits, FIG 1, page 5 [32]).***

1. By "naturally" cooling the system down, such that it reaches the ground state of its Hamiltonian.

2. By measuring a system, which further projects its state into a desired initial state.

In superconducting qubit systems, initialization is achieved through active and passive methods. **Active initialization** involves applying a sequence of microwave pulses to drive the qubit into its ground state. This method is fast but requires precise control over the pulse parameters. **Passive initialization (cooling)**, on the other hand, relies on the natural relaxation of the qubit to its ground state over time, a process determined by the qubit's energy relaxation time $T_1$.

Superconducting qubits operate at very low temperatures, close to absolute zero, facilitated by dilution refrigerators. This extreme cooling is essential to reduce thermal energy that can cause excitations in the qubits, thus naturally bringing them to their lowest energy state (i.e. ground state). Techniques such as feedback control based on real-time qubit state measurement can also be used to enhance initialization fidelity.

As mentioned before, such a process requires highly advanced physical types of machinery and control systems (such as dilution refrigerators), which are usually available in specialized laboratories. This is one of the reasons why the communication between quantum software environments and quantum hardware is so important, as it allows for remote access to these systems.

### 2.2.3. Decoherence Times

Decoherence time describes the ability of a qubit (or any quantum system) to maintain its state (i.e. quantum coherence) long enough when interacting with the environment, a crucial property for performing quantum computations. It encompasses both the energy relaxation time $T_1$ and the phase coherence time $T_2$, representing the actual ability of the qubit to maintain its state and coherence. Decoherence time is a sensitive value in a quantum system, since if it is too long, the capability of the quantum computer tends to shift towards that of a classical one. On the other hand, if the decoherence time is too short, the unique quantum features for the specific computational style might not act as expected.

Superconducting qubits achieve extended decoherence times through engineering efforts aimed at minimizing interaction with the environment, which can introduce noise and loss. Materials science advancements, such as the development of purer superconducting materials and optimized fabrication processes, reduce sources of decoherence. As this decoherence time process can increase the noise and loss of a quantum system, this can also affect the actual development of quantum algorithms and the results of the computations.

### 2.2.4. Quantum Gates

Quantum gates, sequences of transformations, manipulate qubit states and dictate a quantum system's power. Implementing gates in quantum physical systems involves applying specific Hamiltonians ($H_1$, $H_2$, ...) at precise times to achieve desired transformations. In simpler terms, this translates to manipulating qubits with specific controls (Hamiltonians) at defined moments.

In the context of this paper and in the context of quantum software development in general, abstract language specifications such as QASM (see subsection 3.4.1) are used to describe quantum circuits, gates, and other quantum-related instructions. Since these specifications can not be directly understood or converted by the actual quantum hardware/device, to bridge this gap, compilers translate the QASM code into a series of control signals. These signals manipulate the qubits by precisely adjusting their Hamiltonians – essentially fine-tuning the system's energy landscape.

In superconducting quantum computers, these control signals come in the form of shaped microwave pulses and magnetic field adjustments. By carefully adjusting the intensity and duration of these pulses over time, the compiler creates the specific Hamiltonian sequences needed in order to precisely implement and execute the given gates. This allows the quantum computer to translate the abstract algorithm (e.g. from OpenQASM) into the physical manipulations required to achieve the desired quantum transformations. Qubit coupling (shared resonators, direct links) enables two-qubit gates (such as CNOT, CPHASE), crucial for quantum algorithms. Entanglement, on the other hand, is engineered through circuit design and tailored pulses. For a more technically-accurate representation of the above-mentioned process refer to Figure 2.2.

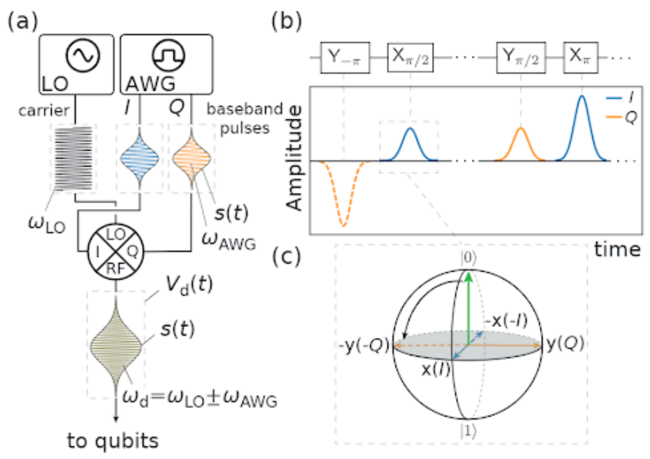


Figure 2.2.: (a) A typical qubit drive setup schematic.
(b) Example of how a gate sequence is converted into a waveform produced by the arbitrary waveform generator (AWG).
(c) The effect of a pulse on a $|0\rangle$ state.
***(Source: A Quantum Engineer's Guide to Superconducting Qubits, FIG 13, page 29 [32]).***

### 2.2.5. Measurement

Similar to classical computing, quantum computations require reading the final state. In the quantum world, this translates to probabilities ($|0\rangle$ with probability $p$ and $|1\rangle$ with probability $1-p$). Ideally, measurements shouldn't alter the qubit's state, which is crucial for error correction and certain algorithms. Therefore, fast and accurate (high-fidelity) measurements are essential.

Superconducting qubits leverage dispersive readout within circuit QED. Here, the qubit is detuned from a resonator, causing a state-dependent shift in the resonator's frequency. By measuring this shift, the qubit's state is revealed, converting the quantum information into a readable classical signal. Optimizing readout involves maximizing this signal compared to background noise.

This technique balances isolation (preventing decoherence) with rapid readout. While minimally disruptive (especially with few photons), the qubit can still decay ($T_1$-relaxation) during measurement, limiting fidelity.

The aspect of measurement is also a very relevant one in the context of quantum software development. Since the results of the quantum computations are usually read out through such measurements, the fidelity and accuracy of these operations are crucial for the overall success of the executed quantum algorithms. Since measurement results are subject to various noise and error inconsistencies, the quantum software development process usually involves multiple iterations of such operations (also known as shots), in order to minimize the influence of noise and errors and maximize the fidelity of the end-result.

# 3. Quantum Software

## 3.1. The Tools We Have at Hand

Quantum software is at the heart of making quantum computing accessible and practical. Serrano et al. (2022) [57] provide a comprehensive review of the main quantum software components and platforms, emphasizing the need for quantum programming languages and computing environments that abstract low-level technology details. This abstraction is crucial for advancing quantum computing technology by making it more accessible to programmers who may not have a deep understanding of the underlying quantum mechanics. The Talavera Manifesto for Quantum Software Engineering and Programming [45] also highlights the complexity of quantum computers and the need for the development of specialized programming languages and computing environments. According to Serrano et al. (2022), most existing environments lack features desired for the advancement of quantum computing, as outlined in the manifesto. The manifesto thus stresses the importance of developing quantum software engineering techniques and tools to ensure the practical feasibility of quantum software. It serves as a guideline and a call to action for the development of robust, user-friendly quantum programming environments and languages, which are essential for harnessing the full potential of quantum computing technology.

The manifesto goes into even more detail, by specifying some principles and criteria that quantum software development and quantum software engineering should follow, such as:

1. **It must be technology-agnostic in regards to quantum programming languages and technologies:** designs solutions to work across different quantum programming languages and technologies without preference.

2. **It must endorse the coexistence between quantum and classical computing:** facilitates the integration of new quantum algorithms using classical reverse engineering methods and supports the use of traditional techniques to help assimilate quantum programming.

3. **It must support quantum software project management effectively:** manages quantum software projects to meet both operational and business objectives, emphasizing the creation of accurate and adaptive estimation methods for quantum software development based on existing models.

4. **It must take into account the continuous development of quantum software:** emphasizes the need for quantum software to be routinely updated and refined throughout its entire lifecycle.

5. **It must aim to deliver quantum software with minimum defects:** creates quantum programs with as few issues as possible and establishes mechanisms for their early detection and resolution.

6. **It must prioritize high-quality maintenance of quantum software:** stresses the importance of both the process and the product in quantum software, with the goal of achieving and sustaining high-quality standards.

7. **It must take into consideration quantum software reusability:** encourages the development of quantum software in a way that allows for components to be reused, facilitating the creation of software libraries and architectural patterns for quantum computing.

8. **It must emphasize the importance of security and privacy from the outset:** commits to ensuring that quantum information systems are secure and protect user privacy from the beginning of the software development process.

9. **It must cover the importance of governance and management in quantum software:** underlines the need for awareness and adherence to organizational structures, processes, policies, and frameworks, as well as an understanding of the infrastructure and services associated with quantum software and the responsibilities of the providing organizations.

Although very hard/impossible to meet all these criteria at this point in time, quantum software engineering and development should strive to meet as many of them as possible. These very same principles were also considered in the architectural decisions and implementation work done on qib and its extensions (i.e. the implementation part of this thesis).

Despite the current limitations and apparent inaccessibility to quantum software development, progress has been made in the field in recent years. With multiple toolkits, platforms, libraries, simulators, and other similar tools emerging from different backgrounds (e.g. academic, commercial, etc..), and different stakeholders coming into play.

Quantum software technologies can be classified into several key categories, such as **(1) programming languages and compilers** [23] [10] [59], **(2) simulation platforms**, **(3) machine learning libraries** [61] [19] [43], **(4) optimization tools** [41] [67], **(5) cryptography and security libraries** [65] [14], and **(6) hardware control software** [9] [3].

In the scope of this thesis, the focus is on quantum software technologies from categories 1, 2, 3, and 6. Where, qib (chapter 5) and Qiskit (section 3.5) fall under the first category, both being toolkits that provide a quantum programming and compilation environment. The quantum simulators that have been used (such as the Qiskit Aer Simulator available at the WMI section 4.2) for testing purposes fall in the second category. While the hardware infrastructure currently available and developed at WMI (section 4.2) falls in the last one. Although Quantum ML libraries have not been directly used, some experiments involving quantum machine learning have been performed (section 7.3) in order to test the developed module, where ML-specific algorithms have been implemented, resembling some of the features or functionalities that tools under the 3rd category provide.

## 3.2. Limitations of Quantum Software

Quantum software faces several limitations, such as hardware dependence, limited application scope, and the lack of mature and accessible development tools and platforms. These limitations are actively being addressed through research and development. Some of the ones that also influenced the implementation and design decisions of this thesis are discussed below:

1. **Noise and Error Rates:** Quantum computers are highly susceptible to errors due to environmental noise, resulting in limitations in quantum software's reliability. Ball et al. (2020) [5] discuss the development of software tools for quantum control that improve quantum computer performance by mitigating noise and errors. These tools facilitate the efficient execution of quantum logic operations and algorithms with built-in robustness to errors, without complex logical encoding.

2. **Limited Qubit Connectivity:** The limited interaction distance between qubits in quantum hardware architectures is a significant constraint. Saeedi et al. (2011) [55] proposed methods to optimize quantum circuits for linear nearest-neighbor architectures, including template matching optimization and exact synthesis approaches. These methods aim to realize circuits for architectures with limited qubit interaction, reducing quantum cost significantly.

3. **Scalability Issues:** Quantum software's scalability is constrained by the limited number of qubits and coherence time in current quantum computers. Akbar et al. (2022) [2] highlight the need for software-intensive methodologies and tools for developing quantum software applications that can operate effectively on these emerging quantum hardware technologies.

4. **Complexity of Quantum Algorithms:** The complexity and unorthodox nature of quantum algorithms present a steep learning curve for programmers. To bridge the gap, García-Alonso et al. (2022) [18] proposed the Quantum API Gateway, an adaptation of the API Gateway pattern, recommending the best quantum computer for running specific quantum services at runtime. This helps in managing the complexity of combining quantum algorithms with traditional software.

5. **Integration with Classical Computing:** The integration of quantum computing with existing classical computing infrastructure is challenging. Perelshtein et al. (2022) [44] introduced a hybrid quantum cloud based on a memory-centric and heterogeneous multiprocessing architecture. This demonstrates how hybrid algorithms, integrating quantum and classical computing, can provide advantages in various fields.

Some of the challenges that quantum computation presents are especially visible in the deployment of quantum software. Unlike classical software, which can be executed on the same hardware it is deployed on, quantum software requires a specialized quantum processing unit (QPU) and further communication channels between this QPU and the development environment. This distinction leads to additional limitations, such as:

1. **Hardware Dependency** [30]: Quantum software is inherently dependent on quantum hardware. QPUs, necessary for executing quantum software, are not as universally accessible as classical computing resources. Their specialized nature, along with the requirement for extremely low temperatures, makes them less adaptable to conventional IT environments.

2. **Compatibility Issues** [7]: The diversity in quantum computing technologies (e.g., superconducting qubits, trapped ions) leads to compatibility issues. Quantum software developed for one type of QPU may not be directly executable on another, limiting the software's versatility and increasing development time and costs.

3. **Connectivity Challenges** [39]: Interfacing quantum software with quantum hardware typically requires high-speed, secure, and reliable communication channels. Furthermore, both ends of the interface should be compatible, accepting the same transpilation, serialization, and configuration standards. All these add an extra layer of complexity and communication overhead, that needs to be managed by all the stakeholders involved in the development of a quantum application.

4. **Single-Use Deployment** [35]: In contrast to classical applications, which are deployed once and can be invoked multiple times, quantum applications require new deployments for each invocation. This is a fundamental shift in the deployment lifecycle, necessitating a more dynamic and flexible approach. Different solutions have been proposed to this issue, but many of them still struggle with fully achieving the QA principles of the Talavera Manifesto [45].

5. **External Hosting of Code** [35]: Unlike traditional cloud models like Infrastructure-as-a-Service (IaaS), where the application code is hosted on the same environment it is executed on, Quantum-as-a-Service (QaaS) does not currently support on-site hosting of quantum applications. Consequently, the quantum application code must be hosted on an external, conventional computing resource. This separation adds complexity, as the quantum code must be compiled and managed externally before being deployed to a quantum computer.

Additionally, the quantum software ecosystem currently lacks the maturity seen in classical computing. Quantum programming languages, development environments, and debugging tools are still in their nascent stages. This lack of mature tooling further complicates the deployment process. Nevertheless, all these limitations create a fertile ground for further solution designs and research trajectories, such as:

1. **Enhanced Software Development Kits (SDKs)** [20]: To bridge the gap between classical hosting environments and quantum hardware, robust SDKs capable of efficiently compiling quantum algorithms and managing their deployment on quantum computers during runtime are essential. These SDKs must handle the complexities of translating high-level quantum algorithms into hardware-specific instructions, accounting for the nuances of different quantum computing platforms.

2. **Orchestration and Automation Tools** [18], [66]: Given the repeated deployment necessity, there's a pressing need for sophisticated orchestration tools that can automate the deployment process, making it as efficient and less resource-intensive as possible. These tools would handle the scheduling and management of quantum computations, optimizing the use of quantum resources.

3. **Hybrid Computing Models** [36]: Since quantum computers excel at specific types of problems, integrating them into hybrid models where they work alongside classical computers can optimize performance. This approach requires developing frameworks and platforms that seamlessly integrate quantum and classical computing resources.

4. **Quantum Cloud Integration Models** [22], [51]: Research into more advanced QaaS models, which might allow for closer integration between the quantum application code and the quantum hardware, can reduce the overhead and complexity of the current deployment model.

5. **Quantum Resource Management:** Exploring efficient ways to manage quantum resources, considering their unique constraints and capabilities, is crucial. This includes research into quantum job scheduling, error correction mechanisms, and optimizing quantum circuit compilation.

6. **Quantum-Ready Deployment Technologies:** Adapting existing deployment technologies to accommodate quantum applications or developing new technologies specifically designed for quantum computing environments will be a significant research area. This involves rethinking deployment models and possibly creating quantum-native deployment paradigms.

7. **Security and Reliability in Quantum Deployment** [15]: As quantum computing becomes more prevalent, ensuring the security and reliability of deployed quantum applications, especially in a cloud-based context, will be a major area of concern. This

includes research into quantum cryptography and secure quantum communication protocols.

In summary, while quantum computing presents significant opportunities, it also introduces a set of unique challenges in software development and deployment. Addressing these challenges requires a concerted effort in developing new tools, models, and frameworks, specifically tailored to the quantum computing paradigm.

Addressing these limitations is also one of the main concerns of this thesis. By implementing a quantum software interface for communicating with various quantum computers and quantum simulator backends, the aspect of quantum software-to-hardware interaction and deployment automation is particularly addressed.

## 3.3. The Generic Execution Flow of Quantum Computation

As one of the main topics and concerns of this paper, this section focuses on offering an overview of a generic execution flow of a gate-based quantum computation, starting from writing the source code and sending experiments over the network, to the execution and results measurement processes that take place on the actual quantum device, and within its quantum environment (also known as the quantum backend[1]). A deeper dive into each one of the steps of this process will be provided, as well as explanations of how each step is relevant to the context of this thesis. The reader is encouraged to refer to the following diagram (see Figure 3.1) for a general understanding and road-map of the explanations provided below.

### 3.3.1. Declaration and Transpilation

The first step encompasses the processes that a quantum algorithm undergoes within the library, SDK, or quantum framework where it is being developed [58]. As for now, this step takes place exclusively on a different machine, a classical one. It all starts with the quantum developer or researcher writing source code that defines the quantum algorithm in the specific quantum programming language of the tool that they are using (e.g. see Figure 3.2). This code is usually all about defining initial, human-readable circuits, that can be later processed by the development environment, and at the end sent to and executed by the quantum backend.

Once the source code becomes available, it can be optimized and transpiled [29] within the classical environment, so that both elements of system-dependent configurations and system-independent ones can be applied to the defined quantum algorithm. This part is the most complex one of this step [48], and it is usually system and tool-dependent, involving rewriting and optimization procedures, such as:

[1]Please note that this term might also refer to quantum simulators, or other quantum processor environments able to execute code, measure results, and send them back to the requester.

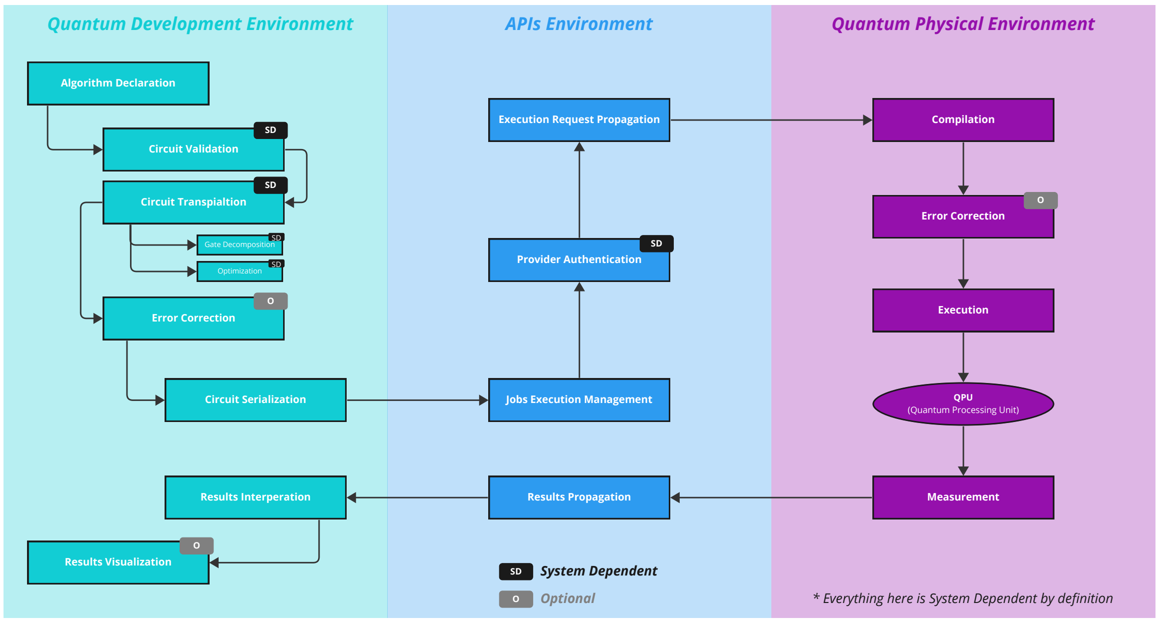
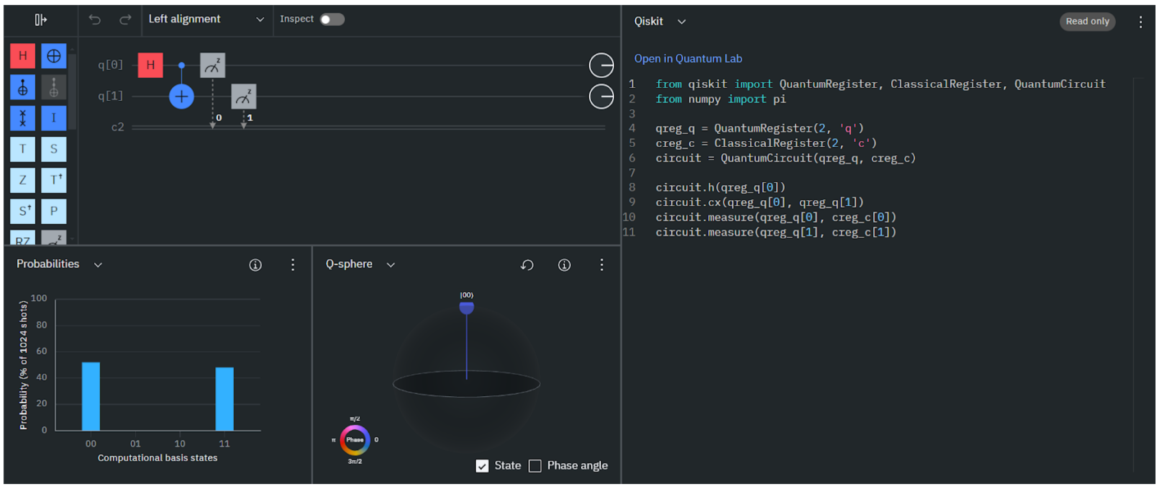


Figure 3.1.: A generic example of a quantum computation execution flow.
***First column:*** The execution steps that take place in the classical development environment of a quantum algorithm.
***Second column:*** The serialization and information exchange steps that take place between the classical and quantum environments (usually part of the logic provided by the APIs and the involved communication channels).
***Third column:*** The execution steps that take place in the quantum environment (quantum backend).

- **Mapping logical qubits to physical qubits** on the quantum device, considering connectivity constraints.
- **Optimizing the circuit** by reducing gate counts and depth through gate cancellations and merging, using various optimization passes.
- **Decomposing high-level gates** into the native gate set supported by the target quantum device.
- **Scheduling gates** to minimize the overall circuit execution time, taking into account the qubit coherence times.
- etc.

This whole step is called transpilation (and not compilation) because it accurately describes the process (an important distinction in quantum computing) because the aim is



```
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
from numpy import pi

qreg_q = QuantumRegister(2, 'q')
creg_c = ClassicalRegister(2, 'c')
circuit = QuantumCircuit(qreg_q, creg_c)

circuit.h(qreg_q[0])
circuit.cx(qreg_q[0], qreg_q[1])
circuit.measure(qreg_q[0], creg_c[0])
circuit.measure(qreg_q[1], creg_c[1])
```

Figure 3.2.: Defining a quantum algorithm in IBM Quantum Learning (for Qiskit)[25].

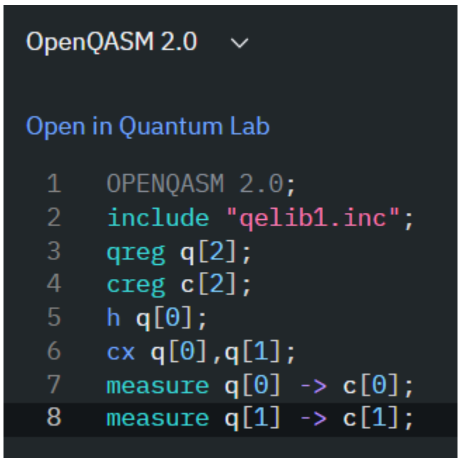
to adapt the circuit for execution on specific hardware without changing the circuit's fundamental behavior or the results it produces. The process involves translating an abstract quantum circuit into a functionally equivalent one that is tailored to the constraints and characteristics of a specific quantum device. This process results in a circuit that operates at the same level of the programming language as the original.

### 3.3.2. Serialization and Information Exchange

Once the source code (i.e. quantum algorithm, circuit, etc.) is written and transpiled, the result can be serialized [37] and sent over the network, to the targeted quantum backend (be it an actual quantum hardware machine, simulator, or other types of quantum processors). The serialization and de-serialization process usually takes place on both ends. Thus, it requires some kind of API implementation and intermediate common language declaration for the communication to be viable. More specifically, this communication takes place between the classical environment where the source code is defined and stored, and the actual quantum environment that is responsible for executing the provided algorithms/experiments.

Furthermore, active communication channels need to be implemented between the two environments, such that a response-reply mechanism can be established between the systems. In order to understand why the communication channels are required, one could take the following scenario as an example:

1. The classical system identifies itself and authenticates to the quantum system, such that execution of experiments on the system becomes available and the communica-



```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[2];
creg c[2];
h q[0];
cx q[0],q[1];
measure q[0] -> c[0];
measure q[1] -> c[1];
```

Figure 3.3.: The OpenQASM 2.0 version of the defined algorithm.

tion channel is established in a secure manner [2].

2. The classical system submits an experiment (a quantum algorithm), with specific configurations and options for it (e.g. the number of shots i.e. how many times the experiment should be repeated before the final measurement is performed and the results are sent back) to the quantum backend over the established communication channel. It further receives a reply from the backend that the request is valid (or invalid), and that its execution has been queued (or rejected).

3. (If valid) The quantum system further processes the received experiment, compiles it, and executes it on the actual quantum processor. It keeps track of the experiment (e.g. by storing it into a database) and updates its status, depending on the progress it made.

4. The classical system can query the quantum system for results, or the current execution status, over the same communication channel that was previously established.

5. Once the experiment finished its execution on the quantum processor, the quantum system can send the results back to the classical system over the communication channel.

6. The received experiment results, can be further interpreted and used within the classical system for all types of quantum applications (e.g. hybrid computation).

[2]This usually involves some kind of secret (e.g. an API Key) that gets exchanged between the two systems and that is later needed by the classical system for it to be authenticated and authorized by the backend environment of the quantum system.

The term ”communication channels” doesn’t necessarily refer to an active communication mechanism, such as message queues or active TCP connections, but rather to an establishment of a mechanism (e.g. a stateless one) that allows classical environments and quantum environments to identify, store, and refer to a common set of experiments, execution jobs, configurations, and various other relevant metadata, needed during the process. In other words, the two environments must be synchronized and keep a record of what, when, how, and with which results certain experiments were, are, or will be executed on the targeted quantum backend.

This whole step can usually be implemented in practice, by using intermediate representation specifications, such as OpenQASM (subsection 3.4.1)(see Figure 3.3) and common data structures definitions, like Qobj (subsection 3.4.2). As for the networking part, implementing classical RESTful APIs [52] that communicate over the HTTP protocol is usually enough to establish a well-defined set of rules and definitions for a viable communication between the environments. For more details regarding industry standards and actual data specifications, refer to section 3.4.

One of the biggest limitations and current subject of research in the field is how to realize this whole step by using a technology- and system-agnostic approach (see section 3.1). Various solutions have been proposed over the years [18]. Since different types of quantum systems require different types of interaction, it is hard to define a clear set of data specifications and intermediate representations that can be uniformly applied to any quantum computation in the world. Although difficult to ensure uniformity on a general worldwide scale, the path is being slowly paved, with language specifications such as OpenQASM (subsection 3.4.1) or OpenPulse [9] bridging the gap between the abstract aspects of quantum algorithms and the physical implementations of quantum systems.

### 3.3.3. Compilation and Execution

Once the information (i.e. experimentation request) reaches the quantum backend, the latter is responsible for validating, de-serializing, and compiling [34] the circuits to the physical representation required for the execution to be adequately performed on the quantum processor (e.g. pulses that should be applied in a specific order, for a specific time, to specific qubits). The backend is also responsible for identifying, storing, and being available to read or write data about the quantum experiments performed on its end. On request, it should offer the required information to its invokers (e.g. the classical system that requested the execution of the experiment).

Since the actual compilation and execution process is system-dependent and out of scope for this thesis, this step will be not explained in detail. But for a clearer understanding of how the actual compilation is performed within a superconducting quantum computer, such as the one available at the WMI, refer to section 2.2. Also, for further details about quantum simulator processors, refer to section 3.5 where an explanation on how the Qiskit simulator (such as the one at the WMI) is working was provided.

### 3.3.4. Measurement and Results Processing

At the end of the execution step, for most algorithms there is usually a measurement involved [53], a physical one, that is performed within the actual quantum processor. However in practice, multiple measurements are performed for such an experiment, and the result is then digitalized and interpreted as a statistical probability distribution of these measurements. The quantum system should be aware of these results and store them somewhere, such that when the classical system requires them, it is able to send them over the established network channel.

After the results have been received by the classical system, this system will be responsible for interpreting and using them accordingly, depending on the given quantum application scenario that is being executed.

The actual measurement operation that is performed within the quantum processor is again out of scope. But feel free to refer to subsection 2.2.5 for more details about how this operation is performed within a superconducting quantum computing environment.

As for processing and interpreting the results, this flow is similar to the one performed in the second step (Serialization and Information Exchange), just reverted: the quantum backend is the sender this time, with the classical system receiving the results, de-serializing them, and interpreting them accordingly.

### 3.3.5. Thesis Focus

This thesis (especially the implementation part) mostly focuses on the second step (Serialization and Information Exchange) since this is the main goal of expanding the qib framework with a backend module (a module able to exchange information between qib's definitions, and various quantum backends).

While step **three (Compilation and Execution)** is mostly out of scope, steps **one (Declaration and Transpilation)** and **four (Measurement and Results Processing)** are also within the scope of the thesis. This is because some declaration-, validation-, and optimization-related processes had to be performed, in order for qib to be compatible with and ready for a standardized intermediate representation and serialization. Tasks such as adding new gates, implementing control flow instructions, and extending existing components with serialization-ready functionalities, were needed in order for qib to be prepared for a coherent information exchange with various backends.

For more design decisions and implementation details, refer to Part II.

## 3.4. Industry Standards and Data Representations

This section mainly focuses on the thesis-scope relevant standard data representations and data models used in quantum software development and experimentation. Quantum interface languages will be discussed, such as OpenQASM, how circuits can be represented for such a language in Qobj notation, what are the benefits of these kinds of representations, why have they been chosen for the implementation part of this thesis, as well as other similar industry standards that have helped throughout the research and development stages of this thesis.

### 3.4.1. OpenQASM

**OpenQASM (Open Quantum Assembly Language)** [12] is a **quantum interface language**, inspired by the classical assembly languages, and based upon the quantum **assembly language (QASM)** [17], [60], [4]. It is designed to facilitate the representation and simulation of quantum algorithms on quantum computing devices. Its scope lies in the ability to provide a bridge between the abstract aspects of quantum computing and the practical implementation of quantum software. OpenQASM allows researchers and developers to describe quantum circuits in a way that can be executed on quantum computers, thereby enabling the exploration of quantum computing applications and the development of quantum algorithms.

As the source reference [12] describes, OpenQASM uses the key concept of **Intermediate Representation (IR)** for representing quantum computations. Thus, OpenQASM is neither a source language description of this computation, nor the actual instructions that will be performed on the quantum hardware itself, but it's rather something in-between, providing an interface between these two. See section 3.3 for more details about the execution flow of quantum computation, and where these IRs languages come into play.

The structure of OpenQASM is designed to be intuitive, allowing for the concise representation of quantum operations. It supports a variety of quantum gates and operations, including conditional operations based on classical logic, which are essential for implementing complex quantum algorithms. The language is also flexible, supporting the description of both idealized quantum circuits and those that take into account the physical constraints of real quantum hardware.

The human-readable form of OpenQASM, inspired by C and assembly languages, consists of multiple instruction sets that can be used in order to describe a circuit (see Table 3.1 for examples of the core instructions). Also, see Figure 3.4 and Source Code 3.1 for an example of such a circuit description.

The same syntax can be used in order to define new (user-defined) unitary gates. See Source Code 3.2 and Figure 3.5 for a specific example of this scenario.

Everything mentioned above mainly focuses on the features provided by OpenQASM 2.0, but in 2022 OpenQASM 3.0 [13] came up. It is more capable than its predecessor, being able to also express concepts on a more fine-grained physical layer. OpenQASM 3.0 introduces some valuable new features, such as:

- Syntax improvements
- New control flow statements, e.g. loops

| Statement | Description | Example |
|---|---|---|
| | ***Structural Statements*** | |
| `OPENQASM` **`version;`** | Denotes a file in Open QASM format (first line of the file) | `OPENQASM 2.0;` |
| `qreg` **`name`**`[`**`size`**`];` | Declare a named register of qubits | `qreg q[5];` |
| `creg` **`name`**`[`**`size`**`];` | Declare a named register of bits | `creg c[5];` |
| `include` **`"filename";`** | Open and parse another source file | `include \qib.inc";` |
| `gate` **`name`**`(`**`params`**`)` **`qargs body`** | Declare a unitary gate | See example in Source Code 3.2 |
| `// comment text` | Comment a line of text | `// this is a comment` |
| | ***Instructions*** | |
| `U(`**`theta,phi,lambda`**`)` **`qubit`**`\|`**`qreg;`** | Apply built-in single qubit gate(s) | `U(pi/2,2*pi/3,0) q[0];` |
| `CX` **`qubit\|qreg,qubit\|qreg`** | Apply built-in single qubit gate(s) | `CX q[0],q[1];` |
| `measure` **`qubit\|qreg`** `->` **`bit\|creg;`** | Make measurement(s) in Z basis | `measure q -> c;` |
| `reset` **`qubit\|qreg;`** | Prepare qubit(s) in $\|0\rangle$ | `reset q[0];` |
| `if(`**`creg`**`==`**`int`**`)` **`qop;`** | Conditionally apply quantum operations | `if(c==5) CX q[0],q[1];` |

Table 3.1.: OpenQASM instruction sets examples.

```
OPENQASM 2.0;
include "qelib1.inc";
// declare the quantum and classical registers that will be used
qreg q[5];
creg c[5];

// column 1
cx q[2],q[1];

// column 2
x q[1];
h q[2];
s q[3];
y q[4];

// column 3
t q[2];
z q[3];

// column 4
tdg q[2];
z q[3];

// column 5
x q[1];
h q[2];
sdg q[3];
y q[4];

// column 6
cx q[2],q[1];

// column 7
measure q[0] -> c[0];
measure q[1] -> c[1];
measure q[2] -> c[2];
measure q[3] -> c[3];
measure q[4] -> c[4];
```

Source Code 3.1.: OpenQASM circuit definition instructions example.

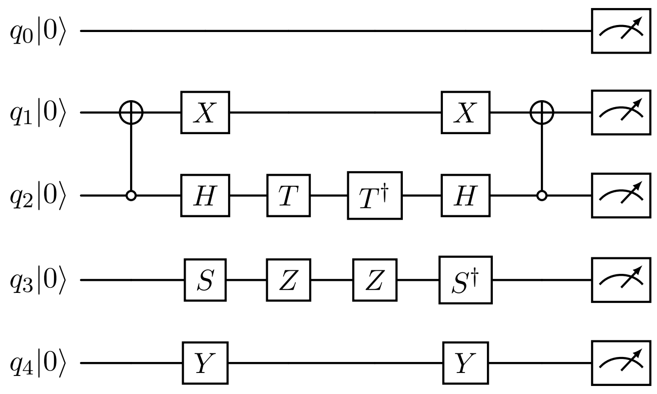


Figure 3.4.: Circuit visual representation of the OpenQASM instructions in Source Code 3.1.

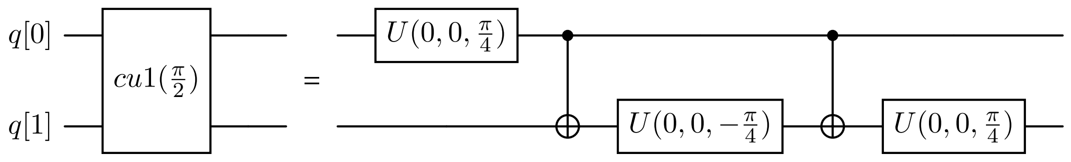


Figure 3.5.: Circuit visual representation of the unitary gate defined in Source Code 3.2.

- Timing and optimization statements, e.g. the delay statement
- Continuous gates, gate modifiers, and new non-unitary operations (i.e. non-destructive projective measurement and reset)

For the live specification of both OpenQASM 2.0 and 3.0, see: [27] and [26]

In the scope of this thesis, the OpenQASM 2.0 specification was mainly used for IRs of the quantum particles, gates, and circuits. All these, in the context of the performed experiments and data representation through the network. OpenQASM-serialized information, together with configurations and other metadata were packaged in Qobj data structures and sent over the network to quantum processors (i.e. backend instances) (See subsection 3.4.2 for more details on the Qobj data structure, and how it was used in the implementation part of this thesis.).

Although OpenQASM 2.0 was the reference point (fully supporting its syntax), some additional statements, like the delay instruction, only found in OpenQASM 3.0, have still been implemented.

The scope of OpenQASM extends beyond merely defining quantum circuits. It serves as a core tool in the quantum computing ecosystem, enabling the simulation of quantum circuits, the optimization of quantum algorithms, and the interfacing with quantum hardware, although further discussing this topic would be out of scope. With its open-source

```
gate cu1(theta) a,b
{
    U(0,0,theta/2) a;
    CX a,b;
    U(0,0,-theta/2) b;
    CX a,b;
    U(0,0,theta/2) b;
}
cu1(pi/2) q[0],q[1];
```

Source Code 3.2.: OpenQASM statements for defining a new unitary gate.

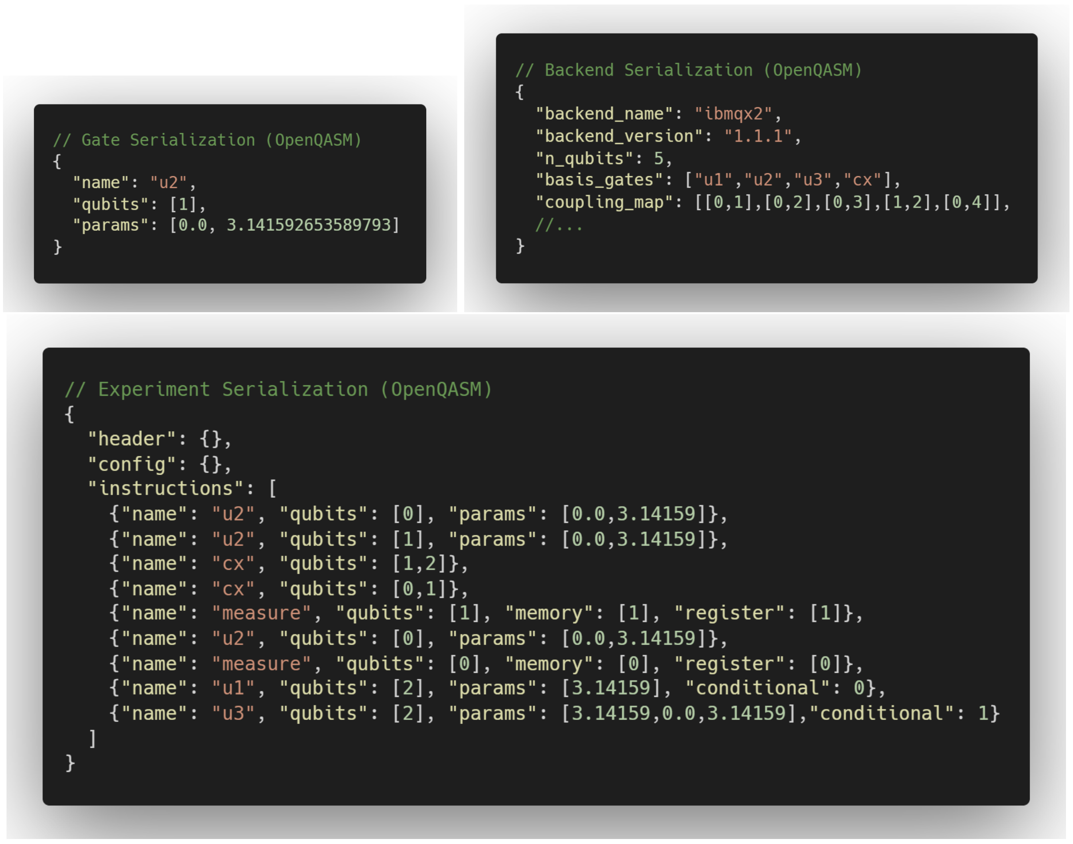
nature, OpenQASM encourages community contributions and the sharing of quantum algorithms, fostering collaboration and innovation within the field of quantum computing.

### 3.4.2. The Quantum Object (Qobj) Data Structure

Although, at its core, nothing more than a JSON-based data representation specification, Qobjs are particularly good for ensuring a standardized data format for quantum experiments. This data format is then used for transferring quantum computing specifications between quantum software development environments, where source code is defined, and quantum processor backends, which process, interpret, compile, and transfer the resulting instructions to the actual hardware, responsible for the actual execution.

In the context of this thesis, Qobjs were used in order to pack the data in a standardized format, that could be later sent over the wire (i.e. using HTTP requests) to the targeted quantum backend. The same Qobjs are used by backends to communicate experiment results and other necessary parameters to the quantum development kit in discussion. A Qobj may be structured in different modes, and different implementations are used out there. But for the scope of this thesis, IBM's Qobj representation [37] [50] was mainly followed, which is also one of the most popular data representation standards for quantum computing.

The OpenQASM specification was used for serializing data structures, such as particles, gates, and circuits, that were later packed in Qobjs, and sent over the network (see Figure 3.6).



```
// Gate Serialization (OpenQASM)
{
  "name": "u2",
  "qubits": [1],
  "params": [0.0, 3.141592653589793]
}
```

```
// Backend Serialization (OpenQASM)
{
  "backend_name": "ibmqx2",
  "backend_version": "1.1.1",
  "n_qubits": 5,
  "basis_gates": ["u1","u2","u3","cx"],
  "coupling_map": [[0,1],[0,2],[0,3],[1,2],[0,4]],
  //...
}
```

```
// Experiment Serialization (OpenQASM)
{
  "header": {},
  "config": {},
  "instructions": [
    {"name": "u2", "qubits": [0], "params": [0.0,3.14159]},
    {"name": "u2", "qubits": [1], "params": [0.0,3.14159]},
    {"name": "cx", "qubits": [1,2]},
    {"name": "cx", "qubits": [0,1]},
    {"name": "measure", "qubits": [1], "memory": [1], "register": [1]},
    {"name": "u2", "qubits": [0], "params": [0.0,3.14159]},
    {"name": "measure", "qubits": [0], "memory": [0], "register": [0]},
    {"name": "u1", "qubits": [2], "params": [3.14159], "conditional": 0},
    {"name": "u3", "qubits": [2], "params": [3.14159,0.0,3.14159],"conditional": 1}
  ]
}
```

Figure 3.6.: Gate, backend (configuration), and experiment serialization using the OpenQASM specification packed as Qobjs.

## 3.5. Qiskit - The Open-Source Quantum Software Development Kit

This section introduces Qiskit, offering a quick overview of its architecture, features, and capabilities. Explaining why it represented an important reference point in the scope of this thesis, as well as for experimenting and testing out features, by using its custom simulator capabilities.

As a leading quantum computing framework developed by IBM Qiskit facilitates the development and execution of quantum algorithms. It provides tools for creating quantum circuits, simulating them on classical computers, and running them on actual quantum hardware accessible through the IBM Quantum Experience platform. Qiskit is designed to support the full quantum computing lifecycle, from experimentation and education to research and application development.

Qiskit's main features include:

- **Circuit Composition:** Allows for the construction of quantum circuits using a comprehensive human-readable library of quantum gates and operations (by using Python as a general-purpose programming language).
- **Simulation:** Offers simulators to emulate the execution of quantum circuits on classical hardware, providing insights into their behavior and outcomes of different experiments.
- **Execution on Real Quantum Hardware:** Enables circuits to be run on IBM's quantum computers via the cloud, or other quantum hardware, via custom extensions of the framework.
- **Quantum Circuit Optimization:** Offers tools for optimizing circuits to improve performance and reduce quantum resource usage.

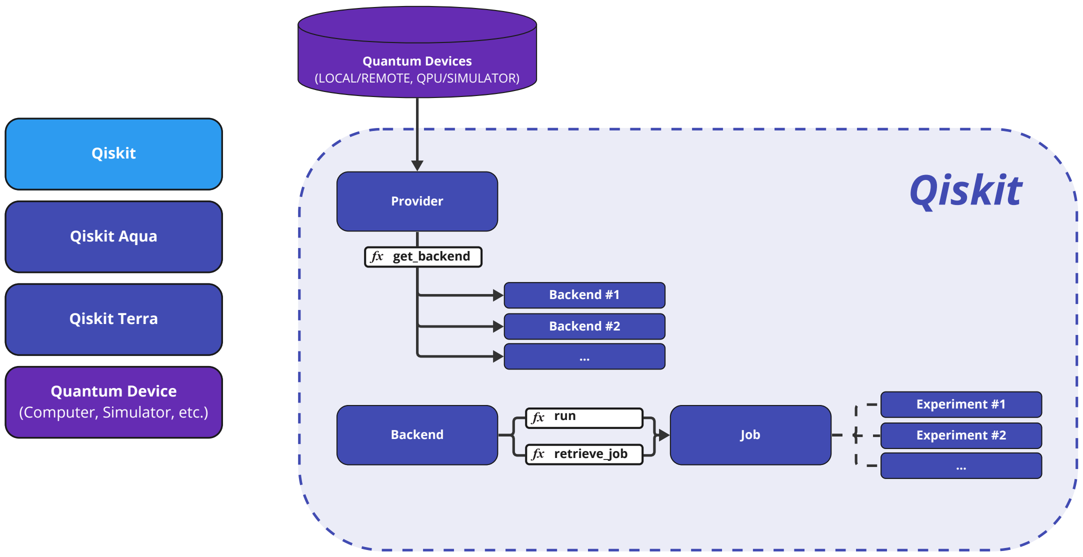


Figure 3.7.: Qiskit's General Architecture Stack.
*Left:* The architecture stack showing Qiskit's core components and their layering.
*Right:* The hierarchy of Qiskit's core classes within the components.

Qiskit's architecture is modular, comprising several components that work together seamlessly (see Figure 3.7):

- **Qiskit Terra:** The foundation of the framework, offering the basic building blocks for quantum circuits, and the interface for compilation and execution on different quantum backends.

- **Qiskit Aqua:** A library of quantum algorithms and applications, facilitating the direct application of quantum computing to real-world problems.
- ***Qiskit Aer:*** Provides high-performance simulators for testing and validating quantum circuits and algorithms on classical hardware (not part of the core architecture).
- ***OpenQASM:*** An implementation of the OpenQASM quantum assembly language (subsection 3.4.1), used by Qiskit (in general) and Qiskit Terra for specifying IRs for the quantum circuits.

These components interconnect through Qiskit Terra, which serves as the core. Terra translates high-level quantum circuit descriptions into executable code for simulators (Aer) and quantum processors, utilizing OpenQASM for circuit representation. Aqua extends Terra's capabilities by offering a higher-level abstraction for algorithm implementation.

In contrast, for a comparison with the architecture stack provided by qib, in the context of extending it for quantum backend interfaces and communications, see Figure 6.3.

See Source Code 3.3 for a simple example of defining the source code for a circuit in Qiskit and executing it on a Qiskit simulator, as a job (i.e. a package comprised of multiple experiments i.e. quantum circuits).

Besides its **ease of use**, Qiskit has other advantages too, such as its **open-source nature**, which allows for custom extensions (such as the custom simulator that was used in the implementation part of this thesis), or the **comprehensive and modular quantum tools** it provides which support a wide range of quantum computing activities. For a more in-depth understanding of Qiskit and its features, see the Qiskit documentation [24].

In the scope of this thesis, Qiskit has represented an important reference point, both in terms of its architecture as well as the standards it implements (e.g. OpenQASM, Qobj, etc.). Since qib is a custom-made open-source quantum development framework, it has many similar concepts or requirements to Qiskit, just on a different scale, and with a more specialized academically-focused approach. This factor has contributed to how aspects like data structures, serialization-methods, transpilation, validation, configuration, and network are and were implemented inside of qib. For more details about the actual implementation and a comparison with Qiskit, see Part II.

One of the most relevant user-facing aspects of a quantum development framework is how its workflow is designed, both in terms of architecture and User Experience (UX). Since both Qiskit and qib have an imperative workflow, they should be designed in a straightforward but configurable manner. See Figure 3.8 for a diagram representation of Qiskit's workflow [3]. The workflow needs to be straightforward, so that no matter the user types and their previous knowledge or background, they are able to perform experiments in an intuitive manner. On the other side, the workflow needs to be highly-configurable, so that experimentation can cover a vast majority of cases. Therefore, different parameters,

[3]Please note that this diagram reflects the workflow as it is structured at the moment of writing this paper. Since Qiskit is still in active development, this might change in the future.

```python
from qiskit import QuantumCircuit, Aer, execute
from qiskit.visualization import plot_histogram

# Create a Quantum Circuit acting on a quantum register of two qubits
circuit = QuantumCircuit(2)

# Add a Hadamard gate on qubit 0, putting it into superposition
circuit.h(0)

# Add a CNOTgate on control qubit 0 and target qubit 1, creating entanglement
circuit.cx(0, 1)

# Map the quantum measurement to the classical bits
circuit.measure_all()

# Execute the circuit on the qasm simulator
simulator = Aer.get_backend('qasm_simulator')
job = execute(circuit, simulator, shots=1000)

# Grab results from the job
result = job.result()

# Returns counts
counts = result.get_counts(circuit)
print("\nTotal count for 00 and 11 are:", counts)

# Plot a histogram
plot_histogram(counts)
```

Source Code 3.3.: A simple example demonstrating the creation of a quantum circuit with Qiskit, executing it on a simulator, and interpreting the results.

configuration properties (both static and dynamic), helper methods, and various metadata have to be implemented within the given framework. For how this specific aspect has been configured and treated within qib, refer to Part II, and specifically to the workflow diagram in Figure 6.2.

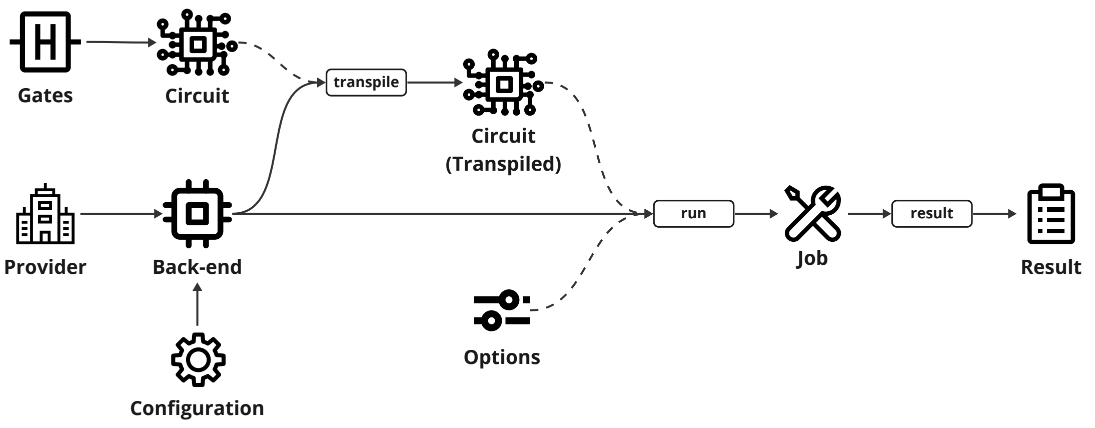


Figure 3.8.: Generic execution workflow for Qiskit experiments.

Moreover, Qiskit has played a crucial role in performing tests in safe and simulated environments within the implementation scope of the thesis. Custom Qiskit OpenQASM simulators have been defined at WMI, so that experimentation could be performed on them, as a first level of fallthrough. This offers big advantages and enhances productivity within the process of quantum applications and experiments development, since:

- It offers a **fail-safe layer** to test different quantum algorithms before they run on the actual physical quantum hardware.
- It offers a **transparent and controllable environment**, where different parameters can be defined and configured accordingly, and the execution flow can be logically understood and debugged.
- It **minimizes resource consumption**, by limiting the overhead of operating the actual physical hardware and cluttering the network with pending requests and waiting queues.
- It **leverages Qiskit's power** and its modules, within the custom system where it is implemented.
- It **eliminates the complex barriers** of communication and data interpretation that are standing between the classical system and the quantum one, for testing purposes.

# Part II.

# Designing a Software Development Interface for Quantum Backends

# 4. The Quantum Computational Environment at the WMI

This chapter will delve deeper into the actual quantum computational environment at the Walther-Meißner-Institute (WMI) and the superconducting quantum computer used for the implementation part of this thesis. It will provide an overview of the hardware, software, and tools available at the WMI for quantum computing research and experimentation. This chapter will also discuss the challenges and opportunities presented by the quantum computational environment at the WMI, setting the stage for the subsequent chapters that focus on interfacing quantum software with this environment.

## 4.1. The Walther-Meißner-Institute

The Walther-Meißner-Institute [64] for Low Temperature Research (WMI) is a research institute operated by the Bavarian Academy of Sciences and Humanities (BAdW). WMI focuses on exploring physics at extremely cold temperatures. They conduct both fundamental and applied research in areas like quantum mechanics and technologies, superconductivity, magnetism, and spintronics. Additionally, they develop specialized methods and tools for reaching and studying very low temperatures, measuring the properties of quantum systems and exotic materials, and growing crystals and thin films for research. WMI even runs a helium liquefier that provides liquid helium, essential for low-temperature research, to both universities in Munich.

## 4.2. The Quantum Backends available at WMI

As deduced from their activity area, the WMI owns various custom-built quantum computers and simulators for research and experimentation. These devices are known as ”quantum backends”, and are the actual quantum devices that execute quantum algorithms and experiments. For this thesis, two of their quantum backends have been mainly used: A Qiskit Simulator (Qiskit Aer) specifically configured for the WMI environment and an actual superconducting quantum chip, which is able to execute quantum circuits through various layers of abstraction, serialization, and communication protocols.

As follows, a brief characterization of these quantum backends, their configuration, differentiating aspects, and capabilities will be provided.

### WMI Qiskit Simulator

| | |
|---|---|
| **Type** | Quantum Simulator |
| **Technology** | Qiskit Aer Simulator |
| **Available Qubits** | 3 |
| **Gate-calibrated Qubits**[1] | all 3 |
| **Available Gates** | `Identity, PauliX, PauliY, PauliZ, Hadamard, SX, RX, RY, RZ, ISWAP, CZ` |
| **Maximum Shots** | $2^{13}$ |
| **Coupling Map** | all 3 qubits are coupled with each other |

Qiskit simulators are powerful tools that allow researchers and developers to test quantum algorithms and circuits without the need for actual quantum hardware (For more details about Qiskit in the context of this thesis, see page 27). This simulator is specifically configured for the WMI environment, providing a reliable and efficient platform for implementing, testing, and debugging various circuits, algorithms, and features, before executing them in a real quantum hardware-based environment.

The simulator at hand is a state vector simulator. This method tracks the probability of every possible state the qubits in the circuit could be in. For small circuits with few qubits, this is a straightforward approach. However, as the number of qubits grows, the number of possible states explodes, making this method impractical for larger circuits. Since only 3 qubits are available in this simulator, the state vector method is a viable option for simulating quantum circuits.

The simulator is also a noiseless one, assuming perfect qubits, and thus being ideal for obtaining clear results in an initial testing and debugging stage.

### WMI Superconducting Quantum Chip

| | |
|---|---|
| **Type** | Quantum Computer |
| **Technology** | Superconducting QPU |
| **Available Qubits** | 3 |
| **Gate-calibrated Qubits** | only the first one ($q[0]$) |
| **Available Gates** | `Identity, PauliX, PauliY, SX, RZ` |
| **Maximum Shots** | $2^{16}$ |
| **Coupling Map** | no qubits are coupled between one-another |

[1]Gate-calibrated qubits are qubits which are configured such that unitary quantum operations (i.e. gates and control instructions) can be performed on them

This quantum QPU (Quantum Processing Unit) is part of a superconducting quantum computer, which is a type of quantum computer that uses superconducting circuits to create and manipulate qubits (see section 2.2 for more details). This chip is a real quantum device that can execute quantum circuits and algorithms, providing a platform for testing and running quantum programs in a real quantum hardware environment.

This particular quantum computer was only calibrated for a limited number of gates and qubits, thus considerably limiting the number of circuits, algorithms, and experiments that could be executed and performed on it.

In comparison to a noiseless simulator, this quantum chip is a noisy quantum device. This means that the qubits are not perfect, and the operations performed on them are subject to errors. These errors can be due to various factors, such as qubit decoherence, gate errors, and readout errors. The presence of noise in the quantum chip can affect the results of quantum computations, making it more challenging to obtain accurate and reliable results. Thus, it was mostly used in the final implementation stages, in order to validate the results obtained in the simulator and to test the robustness of the entire system. The effects of the noise will also be clearly visible in Part III, where the results of the implementation will be analyzed and discussed.

## 4.3. The WMI Backend API and its Architecture

Although the architecture details of the WMI infrastructure are not openly available and are out of the scope of this thesis, the main point of information exchange between the quantum software toolkit (qib) and the Local Area Network (LAN) at WMI is the **WMI Backend API**. So, everything besides the API itself and the actual quantum backends (which have been previously discussed) will be treated as a black-box for the rest of this thesis.

The WMI Backend API is a RESTful API that provides a communication gateway for the quantum backend devices available at WMI. It allows users to interact with the quantum backends, submit quantum circuits for execution, verify their execution status, and retrieve the results of the performed quantum computations. The API is based on the OpenQASM language (subsection 3.4.1) for specifying quantum circuits and the Qobj (subsection 3.4.2) serialization standard for representing quantum circuits in a machine-readable form. Since Qobj is a JSON-based data format, this can be directly included in the body of the HTTP requests exchanged between the API endpoints and the client.

Architecture-wise the WMI Backend API is pretty straightforward, consisting of two main endpoints: one for submitting experiments (in the form of a HTTP PUT request) and one for querying the status and results of the already-submitted experiments (in the form of a HTTP POST request). Refer to Figure 4.1 for a more visual representation of the described API architecture.

As mentioned before, the experiments, submitted in OpenQASM format as circuit instructions, together with their options, and metadata are sent directly in the body of the

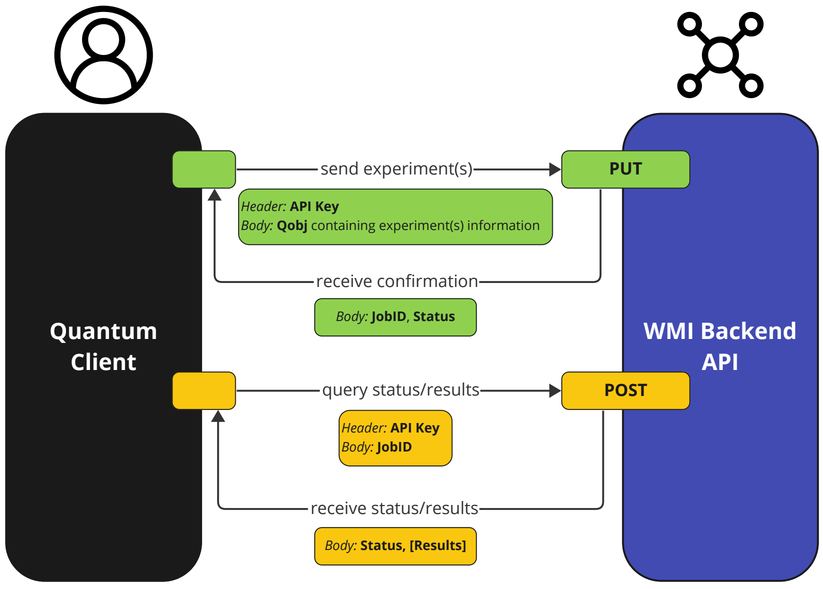


Figure 4.1.: The architecture of the WMI Backend API.

request, serialized as a Qobj (i.e. JSON) object. See Source Code 4.1 for such an example.

The WMI Backend API also provides authentication mechanisms to ensure secure access to the quantum backends. Users are required to authenticate themselves using a valid API key before they can interact with the quantum backends. This API key is generated by the WMI Backend API and is unique to each user. It is used to authenticate the user's identity and authorize their access to the quantum backends. The API key is passed as a header in the HTTP requests sent to the API endpoints, allowing the API to verify the user's identity and grant them access to the desired quantum backends.

```json
{
    "qobj_id": "67b6aac3-8421-4995-b8ff-ac7167fb4b24",
    "type": "QASM",
    "schema_version": "1.3.0",
    "experiments": [
        {
            "header": {
                "qubit_labels": { "qubits": [["q", 0]] },
                "n_qubits": 1,
                "qreg_sizes": { "q": 1 },
                "clbit_labels": { "clbits": [["c", 0]] },
                "memory_slots": 1,
                "creg_sizes": { "c": 1 },
                "name": "ExampleExperiment",
                "global_phase": 0.0,
                "metadata": {}
            },
            "config": { "n_qubits": 1, "memory_slots": 1 },
            "instructions": [
                { "name": "rz", "params": [90], "qubits": [0] },
                { "name": "measure", "qubits": [0], "memory": [0] }
            ]
        }
    ],
    "header": { "backend_name": "dedicated", "backend_version": "1.0.0" },
    "config": {
        "shots": 1024,
        "memory": true,
        "meas_level": 2,
        "init_qubits": true,
        "do_emulation": false,
        "memory_slots": 1,
        "n_qubits": 1,
        "chip": "dedicated"
    }
}
```

Source Code 4.1.: An example of a Qobj request body serialization for a simple quantum circuit.

# 5. qib - The Python Package for Quantum Software Experimentation

qib is a Python package developed at the Technical University of Munich, designed to facilitate quantum software development and experimentation. It provides a comprehensive toolkit for creating and testing quantum algorithms, circuits, and gates, enabling researchers and developers to explore the capabilities and limitations of quantum computing. qib is designed to bridge the gap between theoretical quantum computing concepts and practical implementations, offering a platform for developing and testing quantum programs.

As a quantum development toolkit, qib has multiple features and modules working together to achieve different goals and workflow scenarios, some of these components/features include:

- **Quantum Circuits Representation:** qib provides a module for creating and manipulating quantum circuits, which are the fundamental building blocks of quantum algorithms. Quantum circuits consist of quantum unitary operations (such as gates) that operate on fields (such as qubits), which could have different properties and be configured in different latices (i.e. coupling maps). This feature enables the execution of quantum computations. qib's quantum circuit module allows users to define quantum circuits, add quantum gates, and simulate their behavior.
- **Tensor Networks:** qib supports tensor network representations for quantum circuits, which are essential for simulating and optimizing quantum computations.
- **Quantum Algorithms:** qib includes some already-implemented algorithms that can be used out of the box, such as qubitization and variational quantum eigensolver algorithms.
- **Quantum Simulators:** qib also provides quantum simulators, such as a statevector simulator, as well as a tensor-network simulator. These simulators allow users to simulate quantum circuits and algorithms, providing insights into their behavior and performance.

Since qib's development is still in progress, various modules and features are yet to be added to the toolkit. One of these modules is the backend module, which is also the main focus of this thesis. For more information regarding its architecture and implementation details see chapter 6.

## 5.1. The Architecture of qib

As mentioned before, qib is structured in multiple Python modules, with different functionalities and purposes. The modules of qib at the point of writing this thesis are:

- **field:** This module contains information and definitions for different types of quantum particles, such as qubits, bosons, fermions, etc. As well as the `Field` class, which acts as a bridge between the type of a particle and its lattice configuration.
- **lattice:** The lattice, also known as the coupling map, is a configuration of a group of particles in a quantum system. This configuration describes how the particles are displaced in the system, and how they interact with each other. This information is especially important in the context of multi-qubit quantum operations. qib implements various lattice configurations, such as integer, layered, and triangular lattices.
- **operator:** This module contains definitions for all qib-implemented quantum operations that can be performed on particles in the scope of a quantum circuit. These operators include unitary operators, such as quantum gates, but also control or measurement operators, such as delay or barrier.
- **circuit:** This module mainly defines the `Circuit` class, which represents a quantum circuit. The `Circuit` class contains a list of order-dependent quantum operations that are applied to a set of particles (e.g. qubits) in a quantum system. The module also contains functions for creating and manipulating quantum circuits, such as appending gates, merging circuits, or converting the circuit to a tensor network.
- **transform:** This module mainly includes quantum encodings for fermionic field operators.
- **algorithms:** This module contains already-implemented algorithms that can be used within qib, such as the qubitization algorithm or the variational quantum eigensolver algorithm.
- **simulator:** This module contains quantum simulators implemented in qib, for testing and debugging purposes. The simulators currently include a statevector simulator and a tensor-network simulator.
- **tensor_network:** This module contains components used for the tensor network representation feature of qib.
- **backend:** This is the module that was implemented in the scope of this thesis, and it contains the interface module for quantum backend communications. For more information on this module, see chapter 6.
- **util:** This is an utilitarian module, containing auxiliary functions and parameters used across qib's modules, such as conversions, networking functions, or constants.

In terms of project structure, qib is organized as a Python package, with the following root directory structure:

- **doc:** Containing Sphinx-generated documentation for the qib package.
- **examples:** Containing various examples of qib's features and use cases.
- **src:** Containing the actual source code of the qib package, structured in the modules described above.
- **tests:** Containing unit tests for the qib package, ensuring the correctness of its functionalities.
- **Other files:** Containing setup and configuration files, version control specific files, the open-source license, etc.

For more details regarding qib, its architecture, source code, as well as some use-case examples, see the official qib documentation [47] as well as the open-source code available on GitHub [46].

## 5.2. Gate-based quantum computing in qib

Besides all the other features and modules that qib supplies, the main focus of this thesis is on the ones that provide the necessary tooling for enabling gate-based quantum computing. For the sake of simplicity, this bundle of modules will be called "$qib_{CORE}$" for the remainder of this thesis. This chapter will dive deeper into the respective architecture of $qib_{CORE}$, as well as provide some examples of how gate-based quantum computing can be performed using qib.

As seen in Figure 5.1, the main classes and components [1] of $qib_{CORE}$ interrelate with each other in a loosely coupled manner (e.g. by using interface segregation and dependency inversion principles), which allows for a high degree of extensibility, maintainability, modularity, and configurability, when it comes to developing and utilizing $qib_{CORE}$ for quantum software development. The most relevant classes of $qib_{CORE}$ are:

- **Field:** This class is responsible for defining the properties of a quantum field, capable of withholding quantum particles, such as qubits, bosons, fermions, etc. The field also contains information about the coupling (i.e. lattice configuration) of the respective particles. This setup makes $qib_{CORE}$ capable of modeling and simulating complex quantum systems with a high degree of fidelity and configurability.

[1]For the sake of simplicity, some out-of-scope classes, methods, and members will not be discussed in this context, or will be just shallowly mentioned.

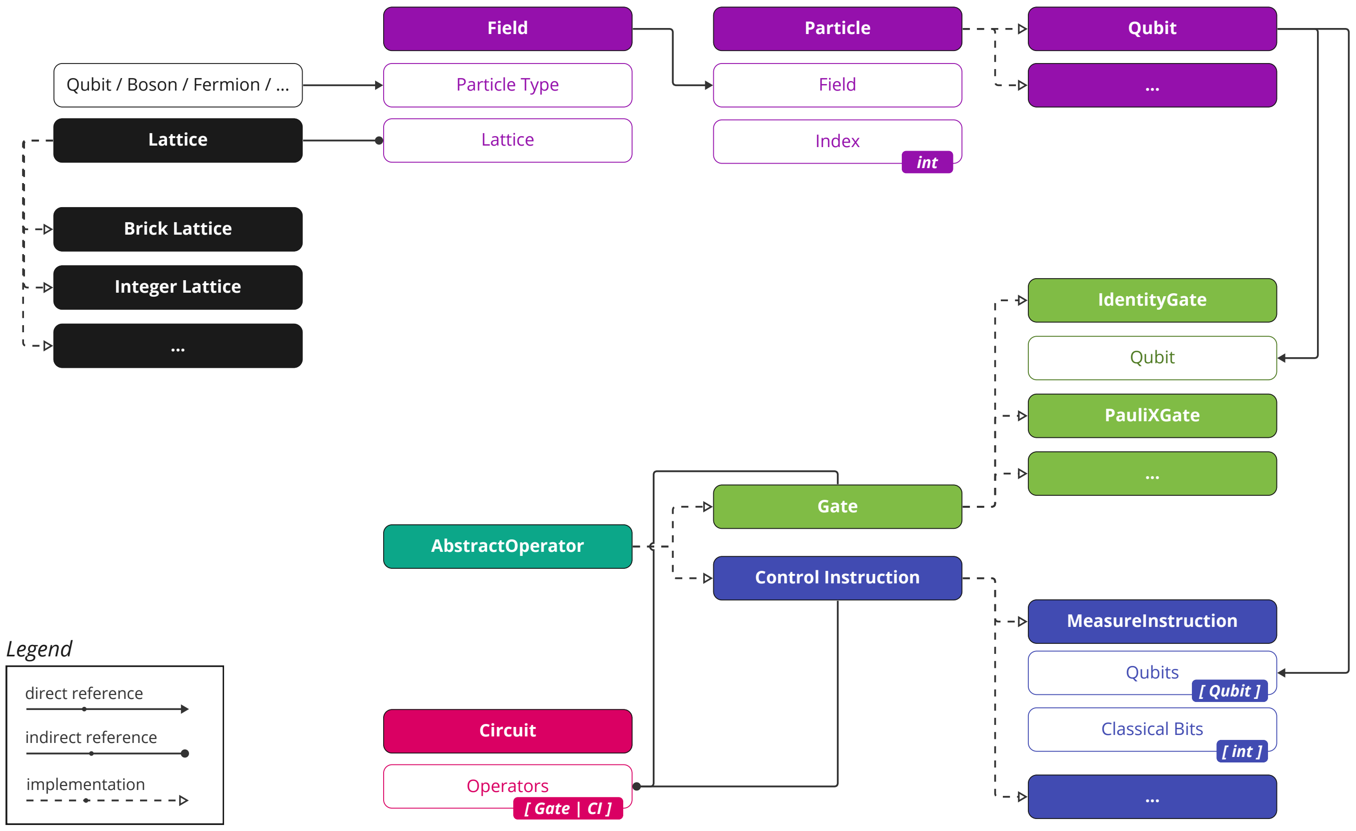


Figure 5.1.: The architecture of gate-based quantum computing classes in qib (part of $qib_{CORE}$).

- **Particle:** This is the base class that describes the properties of a particle (e.g. qubit) in a given quantum system. It mainly contains a reference to the field that withholds the given particle, as well as the particle index, which is later used as a unique identifier of the given particle.
- **Qubit:** This class is an implementation of the **Particle** base class, representing the most common type of particle found in a gate-based quantum computing system. This will also be solely used in the context of this thesis.
- **Gate:** This class is one of the implementations of the **AbstractOperator** base class, representing a base class of its own, for all the unitary quantum operations that could be performed in a quantum circuit (i.e. quantum gates). Further implementations of the **Gate** class, represent the actual quantum gates (such as the Identity, Pauli X, Hadamard, etc. gates). The actual gate classes also define the specific behavior of the gate (i.e. the matrix representation of the gate, the number of qubits it operates on, etc.).
- **Control Instruction:** This class is another implementation of the **AbstractOperator** base class, representing a base class for all control instructions of a quantum circuit [2]. Control instructions are used to control the flow of a quantum circuit, by adding barriers, delays, or other control operations. Moreover, in order to avoid unnecessary complexity, the measurement operation was also defined as an implementation of the **Control Instruction** base class.
- **Circuit:** This class stands at the core of gate-based computing in qib. It represents a quantum circuit, containing a list of quantum operations (such as gates and control instructions) that are applied to a set of particles (e.g. qubits) in a quantum system. The **Circuit** class also contains methods for creating and manipulating quantum circuits, such as appending gates, merging circuits, or converting the circuit to a tensor network.

As for a concrete example of how qib can be used for gate-based quantum computing, consider the Python code snippet in Source Code 5.1. This code snippet demonstrates a basic example of how to create a quantum circuit in qib, add quantum gates to it, and simulate its execution using the statevector simulator provided within qib.

[2]This class was added as part of the other extensions provided to qib in the scope of the backend module implementation. See section 6.5 for more details.

```python
import qib

# Define a qubit field with two sites
# (i.e. a quantum register containing two qubits)
field = qib.field.Field(qib.field.ParticleType.QUBIT,
                        qib.lattice.IntegerLattice((2,)))

# Define the two qubits of the field
qa = qib.field.Qubit(field, 0)
qb = qib.field.Qubit(field, 1)

# Define a standard Hadamard and CNOT gates and link them to the qubits
hadamard = qib.HadamardGate(qa)
cnot = qib.ControlledGate(qib.PauliXGate(qb), 1).set_control(qa)

# Construct a circuit out of the gates
my_circuit = qib.Circuit()
my_circuit.append_gate(hadamard)
my_circuit.append_gate(cnot)

# Simulate the circuit using the statevector simulator
statesim = qib.simulator.StatevectorSimulator()
result = statesim.run(my_circuit, [field], None)
```

Source Code 5.1.: A basic example of using qib for local/simulated gate-based quantum computing.

# 6. Extending qib by implementing an Interface Module for Quantum Backend Communications

After offering theoretical and pragmatical insights into quantum computing and quantum software development as a whole (Part I), and describing the setting of this thesis in chapter 4 and chapter 5, this chapter finally delves deeper into the implementation work that has been done in order to extend the qib toolkit to interface with quantum backends, and particularly with the quantum backends provided by the Walther-Meißner-Institute (WMI). This chapter covers the architecture, design choices, and implementation aspects of the backend interface module, as well as various workflow, architectural, and networking diagrams of the implemented solution.

As it was done with "$qib_{CORE}$" in the previous chapter, for simplicity reasons, the backend module of qib together with all its members will be referred to as "$qib_{BACK}$" for the remainder of this thesis.

## 6.1. Extending qib for Quantum Backends

As seen in Figure 6.1, $qib_{BACK}$ follows the same loosely-coupled architecture as $qib_{CORE}$. It uses the same design principles and patterns, in order to ensure uniformity and maintainability among the modules of qib. The architecture consists of some core classes, such as the `Experiment` and `QuantumProcessor` classes, which ensure the main functionality of quantum algorithms definition, initialization, validation, serialization, and execution flows; as well as some auxiliary classes, mostly used for configuring the executed experiments, or storing data, such as the `ExperimentResults` or the `Options` classes.

Since qib's source code is entirely written in Python, no abstract classes are available by default. However, the **`abc`** module was used in order to define such abstract classes, methods or properties. This way, the backend module can be easily extended to support other quantum backends and providers, just by implementing the abstract classes and methods defined in the backend module (see section 6.6).

In Figure 6.1 there is also a certain distinction made between ***Functional Components*** (i.e. mainly classes that ensure the functional/logical flow of executing quantum algorithms on various quantum backends over the network) and ***Data Components*** (i.e. auxiliary classes that ensure the configuration and (meta)data storage of the quantum algorithms and experiments). This distinction was made in order to ensure a clear separation

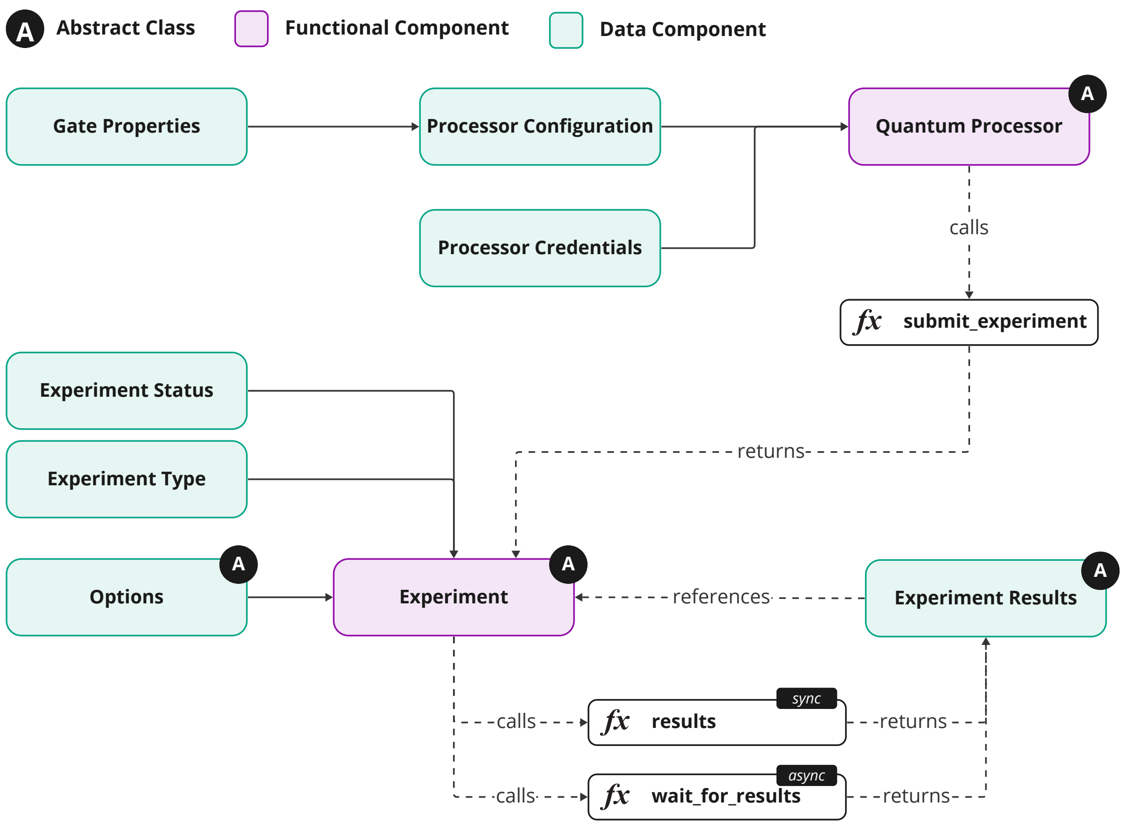


Figure 6.1.: The architecture diagram of the qib backend module.

of concerns and again to make the code more maintainable and extendable.

A short description (in the order of relevance) of each one of the components/classes of $qib_{BACK}$ and their role in the module will be provided, as follows:

- **Experiment:** This core class is responsible for modeling an actual quantum experiment that runs on a given quantum processor (i.e. backend). It has methods for initializing, validating, serializing, and querying results (in both blocking and non-blocking modes) for such an experiment. Although not a main feature of such an experiment, and not implemented for the WMI provider, at the moment of writing this thesis, the class also provides an abstract method for canceling an already-running experiment.

- **Quantum Processor:** This core represents a model of the actual quantum device/backend on which the experiment will run. It contains the configuration of the processor (see **Processor Configuration**), as well as provides methods for submitting a quantum circuit to the backend, generating an **Experiment** object in the process.

- **Processor Configuration:** This is a helper class, used for configuring a quantum processor/backend. It is meant to store configuration properties, that will be taken into consideration when executing experiments on the given processor. This class is not an abstract one, and it is not meant to be customizable by the different providers, but rather to contain core properties, that can be found and configured in any type of quantum backend (such as the name and version of the backend, the number of qubits and available gates, desired query frequency, etc.).

- **Processor Credentials:** This is a small auxiliary class used for initializing and storing the credentials of a given quantum processor at run time (credentials such as the `API Key` of the backend gateway of the provider).

- **Gate Properties:** This helper class, is mainly used within the **Processor Configuration** in order to define the properties of the gates that are available on a given quantum processor. These properties include (but are not limited to): the name of the gate, the number of qubits it acts on, as well as the optional parameters it takes.

- **Options:** This is an auxiliary class, used for configuring the execution of an experiment. In contrast to the **Processor Configuration**, this class is a client-facing one, and it is meant to be customizable by the user, in order to fine-tune the execution of an experiment (such as the desired number of shots, the initialization mode of the qubits, and other provider-specific properties that can be configured on-the-fly). It is composed of both required and optional properties, with the required ones being mandatory for the execution of an experiment (and being set by default with some pre-given values, which can be changed by the user), and the optional ones being also configurable by the user, but not mandatory for the execution of an experiment.

- **Experiment Results:** This data class mainly stores the results of a previously-executed quantum experiment, together with a reference to the given experiment, and some minimal functionality meant for post-processing/representing the results in the desired way.

- **Experiment Status:** This is an enumerator class, representing the execution state of a given experiment. The experiment can be in both non-terminal states (i.e. `INITIALIZING`, `QUEUED`, `RUNNING`), as well as terminal ones (i.e. `DONE`, `ERROR`, `CANCELLED`). For a full list and description of these states, see Table 6.2.

- **Experiment Type:** This is also an enumerator class, representing the type of the executed experiment (i.e. the standard it uses for representing the gates and serializing the information). By default, currently only **OpenQASM** (see subsection 3.4.1) is supported, with the future possibility of extending it for **OpenPulse** experiments, other vendor-specific or custom standards, or flavor alterations of existing standards.

In terms of functionality, the backend module exposes and requires an implementation for the following main methods:

- **QuantumProcessor.submit_experiment(NAME, CIRCUIT, OPTIONS):** Responsible for submitting a quantum circuit to the processor, by creating, initializing, and validating an **Experiment** object, and returning it. The experiment will be initialized using the given `NAME`, and its execution will be defined by the serialized `CIRCUIT` and `OPTIONS` provided to this method. After the serialization and validation steps, the experiment will be automatically sent to the given quantum backend over the network, as part of the submission process.

- **Experiment.results():** The blocking mode of attaining the results of an `Experiment` (or an error, if the experiment was unsuccessful). This method will wait for the experiment to finish, and then return the results in the form of a `ExperimentResults` object.

- **Experiment.wait_for_results():** Same as **Experiment.results()**, but in a non-blocking mode, utilizing Python's `asyncio` capabilities.

- **Experiment.query_status():** This method is responsible for querying the current status of an experiment, storing the results if the experiment has finished (or an error, if the experiment was unsuccessful), and returning the `ExperimentStatus` of the experiment. This method is also internally used by the **wait_for_results()** and **results()** methods. The two methods are essentially calling the **query_status()** method in a loop, with the given query frequency, until the experiment reaches a terminal state.

Some non-exposed, but still important methods of quantum backends that want to be enabled within the module are:

- **QuantumProcessor._send_request(EXPERIMENT):** This is the actual method that sends the given `EXPERIMENT` to the quantum backend over the network. It is backend-specific, and thus the logic needs to be defined in a custom manner for each individual backend.

- **QuantumProcessor._process_response(EXPERIMENT, RESPONSE):** Same as the **_send_request()** method, this method is also backend-specific, and it is responsible for processing the response of the quantum backend after the experiment has been submitted. The method will parse the given `RESPONSE` dictionary, and update the given `EXPERIMENT` object with the results, or an error, if the experiment was unsuccessful.

- **Options.optional():** This method is responsible for returning the optional properties of the `Options` class, in a dictionary format. This method is not mandatory, but it is recommended to be implemented, in order to provide a clear overview of the configurable properties of the `Options` class, and make sure they are only included in the serialization if explicitly specified by the user.

- **`Experiment._initialize()`:** This method is responsible for initializing the experiment, by setting the initial state of the experiment and preparing it for the validation and submission steps. These preparation steps include serializing the experiment to the given type (e.g. OpenQASM), generating a `UUID` value for the `QobjID`, as well as ensuring that properties and members are set to their corresponding initial value.

- **`Experiment._validate()`:** This method is responsible for validating the experiment. Various validations are performed as part of this step, such as checking if the number of shots does not exceed the maximum allowed one, checking if the number and coupling of qubits in the submitted circuit is compatible with the configuration of the processor (i.e. with the specification of the backend), checking if the utilized gates are available and adequately configured by the backend, etc.

- **`Experiment.cancel()`:** This method is responsible for canceling an already-running experiment. It is not mandatory to be implemented, but it is recommended to be implemented, in order to provide a clear way of canceling an experiment, in case it is needed.

- **`Experiment.as_qasm()`:** This method is responsible for serializing the experiment to the OpenQASM standard.

- **`Experiment.from_json(JSON)`:** This method is responsible for deserializing an experiment from a `JSON` object, and initializing it with the given properties.

- **`ExperimentResults.get_counts(BINARY)`:** This method is responsible for returning the counts of the results of the experiment, in a dictionary format. If the `BINARY` parameter is set to `True`, the counts will be returned in binary format, as opposed to its default behavior (`False`) that returns the counts in hexadecimal format.

- **`ExperimentResults.from_json(JSON)`:** This method is responsible for deserializing the results of an experiment from a `JSON` object, and initializing the `ExperimentResults` object with the given values.

## 6.2. The Workflow of executing Quantum Experiments in qib

This section offers an overview of the generic workflow of executing quantum experiments in qib, using the $qib_{BACK}$ module. The workflow is depicted in Figure 6.2, and it is grouped by modules, in order to provide a clear and structured view of the steps that need to be taken in order to run a quantum experiment in qib.

The quantum developer begins by declaring the fields, qubits, and gate configuration of the desired quantum circuit, steps that can be also seen in Source Code 5.1. As opposed

```python
import qib

# ... (see Source Code 6.1.)

# Initialize the desired quantum processor and experiment options
processor = qib.backend.wmi.WMIQSimProcessor(access_token = "AccessToken")
options = qib.backend.wmi.WMIOptions(
    shots = 1024,
    init_qubits = True,
    do_emulation = False)

# Submit the experiment
experiment = processor.submit_experiment(
    name = "MyFirstQuantumExperiment",
    circuit = my_circuit,
    options = options)

# Query and print the results of the experiment
results = experiment.results()
print(results.get_counts())
```

Source Code 6.1.: An example of running a quantum experiment in qib, using the $qib_{BACK}$ module.

to the source code in Source Code 5.1, now the developer uses the $qib_{BACK}$ module, and more specifically the `submit_experiment` method of the `QuantumProcessor` class, in order to submit the experiment to the backend (see Source Code 6.1). This method will automatically handle the initialization, serialization, and validation steps of the submitted quantum circuit and options, and will return an `Experiment` object, which can be further used for querying the results of the experiment.

The querying of the results can be realized in both a blocking or non-blocking mode, by using the `results` or `wait_for_results` methods of the `Experiment` object respectively. The results will be stored in an `ExperimentResults` object, which can be further processed, analyzed, and post-processed by the quantum developer.

In comparison to Qiskit's workflow (see Figure 3.8), qib's workflow (see Figure 6.2) has a more linear approach, that strives to simplify the execution of quantum experiments, by reducing the number of steps needed to be taken by the quantum developer, while still keeping the same degree of configurability and flexibility. One of the main differences is represented by the missing transpilation feature/step of qib, which is not yet implemented (at the moment of writing this thesis). Moreover, the initialization, serial-

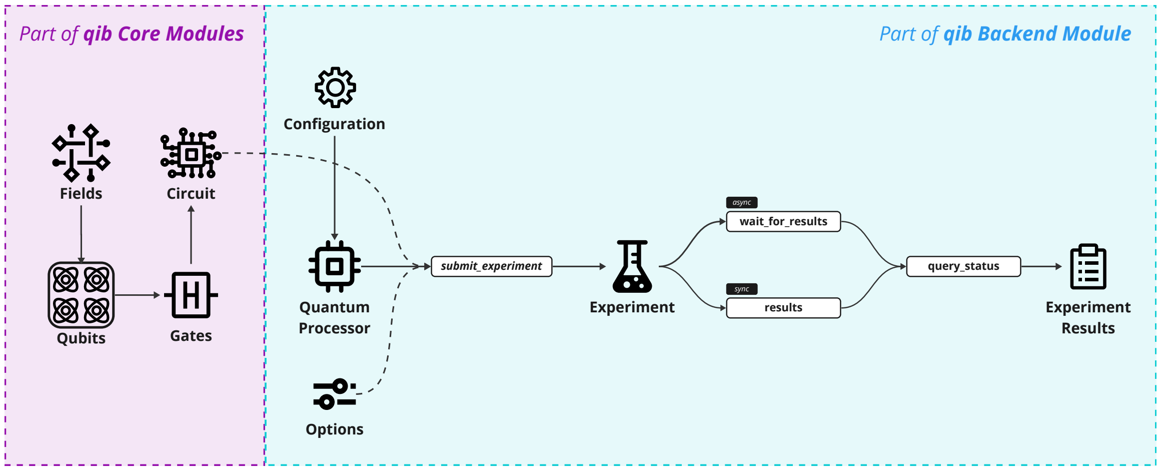


Figure 6.2.: The workflow of executing quantum experiments in qib (grouped by modules).

ization, validation, and the (future) transpilation steps are automatically handled by the `submit_experiment` method, thus providing a more streamlined and user-friendly experience for the quantum developer.

## 6.3. qib Implementation of the WMI Backends Interface

A full architecture stack covering 3 different scenarios can be seen in Figure 6.3, where the backend module is shown as an interface between the qib core modules and the quantum backends that it enables. As seen in the figure, contrary to Qiskit (see Figure 3.7), qib does not explicitly provide an intermediate `Provider` base class that manages the backends (`QuantumProcessor` classes in this case), but rather flattens this hierarchy, by directly declaring all the functionalities of the backends of a provider, in a separate submodule of $qib_{BACK}$. This architectural decision was made in order to simplify the hierarchy of the architecture while preserving the same sense of structure and modularity.

In this section, ***Scenario 1*** and ***Sceenario 2*** are particularly relevant, showing the architecture stack of qib, in the context of enabling it for the WMI backends (i.e. the quantum computer and the Qiskit simulator, see chapter 4). In this context, the backend module was extended with the following classes (see Figure 6.4):

- **`WMIQSimProcessor`:** This represents the actual implementation class of the quantum processor for the Qiskit simulator backend provided by WMI (see section 4.2). It extends the `QuantumProcessor` base class, and it is responsible for submitting experiments to the WMI Qiskit simulator backend, as well as for processing the responses of the backend. The class also contains the configuration of the processor,

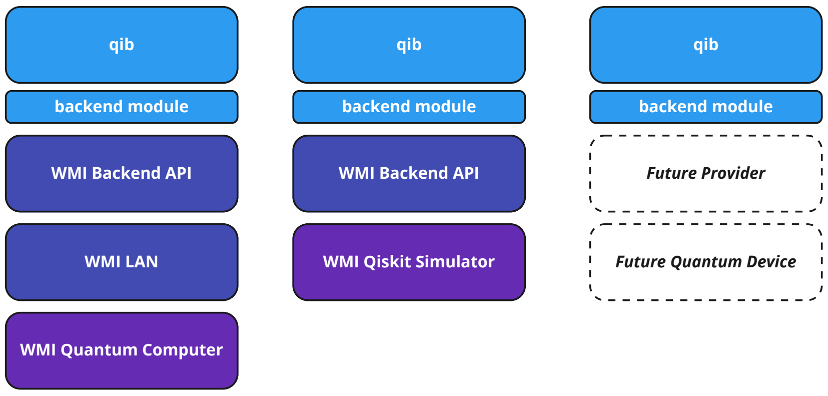


Figure 6.3.: The architecture stack of qib, with clear visibility on how the backend module works as an interface.
***Scenario 1** (1st **column):*** The stack architecture of qib interacting with the WMI quantum computer backend.
***Scenario 2** (2nd **column):*** The stack architecture of qib interacting with the WMI Qiskit simulator backend.
***Scenario 3** (3rd **column):*** The stack architecture of qib interacting with a future quantum provider/backend.

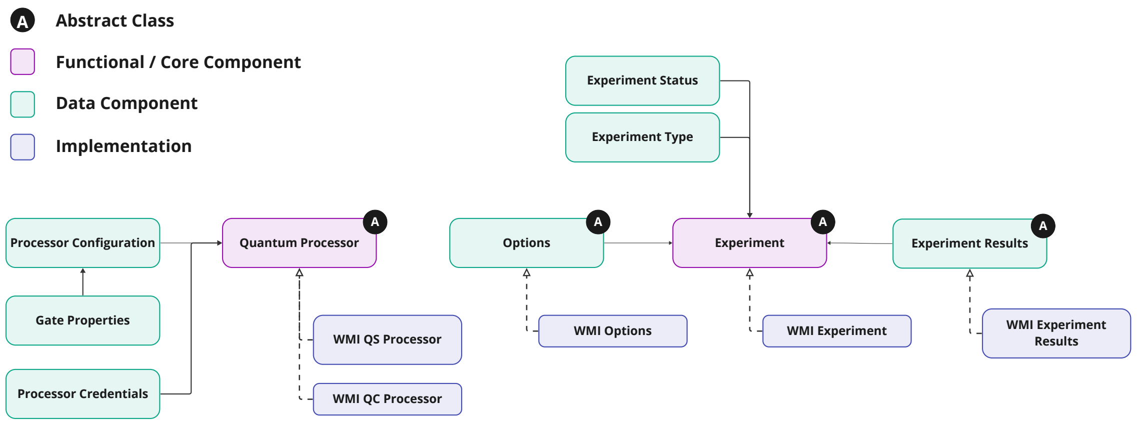


Figure 6.4.: The architecture diagram of the qib backend module, with the WMI implementations.

as well as the credentials structure needed for the authentication of the user (i.e. the hardcoded `URL` and the `API Key` that should be provided by the user at run-time).

- **`WMIQCProcessor`:** Same as the **`WMIQSimProcessor`**, this class represents the actual implementation class of the quantum processor for the WMI quantum computer backend. It has a similar logic and behavior as its counterpart, but its configuration is different, reflecting the slightly more restricted configuration of the actual quantum machine (see section 4.2).

- **`WMIExperiment`:** This class reflects and defines the experimentation model used among the backends provided by the WMI. In terms of functionality, it contains the most relevant and core logic of executing quantum experiments on the WMI backends, thus extending the architecture provided by the `Experiment` base class.

- **`WMIExperimentResults`:** This class extends the `ExperimentResults` base class, and it is meant to be used for post-processing, deserializing, and representing the results obtained by the experiments previously executed on the WMI backends.

- **`WMIOptions`:** This class extends the `Options` base class, and it is meant to be used for configuring and providing execution context for an experiment meant to be executed on the WMI backends. This implementation includes a multitude of custom-configurable WMI-specific properties, with some of the most relevant ones described in Table 6.1.

## 6.4. The Networking Architecture of Quantum Backend Communications

When it comes to the networking part, $qib_{BACK}$ uses a HTTP RESTful type of communication, in order to send and receive the quantum experimentation data to and from the quantum backends (i.e. from the APIs that the backends expose).

This section will further explore some networking scenarios that can occur when running quantum experiments in qib, using the WMI provider interface. The scenarios will cover the default case of running a successful quantum experiment, as well as the cases of running an invalid experiment, encountering a runtime error, and canceling a running experiment.

### Networking Flow of a Successful Quantum Experiment

Figure 6.5 shows the networking flow of running a successful quantum experiment in qib, using the WMI provider interface.

[1]Required options will be marked with a star symbol (*).

| Option Name[1] | Description |
|---|---|
| *int **shots** | The number of shots that the backend should perform for the given experiment. **Default value: 1024** |
| *bool **init_qubits** | If the qubits should be initialized to the ground state $\|0\rangle$, or not. **Default value: True** |
| *bool **do_emulation** | If emulation should be performed in the quantum system before running the experiment. Emulation ensures that the qubits will be in random states before execution. **Default value: False** |
| float **trigger_time** | It corresponds to the duration of an acquisition loop, essentially indicating how frequently experiments are repeated. **Default value: 0.001** |
| str **acquisition_mode** | Specifies the method to be used when gathering information about the quantum state. Different acquisition modes target different aspects of the system's properties. **Default value: 'spectroscopy'** |

Table 6.1.: The most relevant options of the `WMIOptions` class, used for configuring the execution of an experiment on the WMI backends.

The flow starts with the quantum developer initializing an object of its desired quantum processor (e.g. `WMIQCProcessor`), defining the circuit(s) to be executed, and submitting the experiment to the processor (via the `submit_experiment` method), together with the desired name and options. This gets then processed by $qib_{BACK}$'s internal logic, which sets the ***Experiment Status*** to `INITIALIZING`, initializes, validates, and serializes the experiment, and sends it to the ***WMI Backend API*** via an `HTTP PUT` request. The experiment is sent as a Qobj (see subsection 3.4.2) serialized JSON object directly in the body of the request.

If the submitted experiment is valid, this gets then acknowledged by the ***WMI Backend API*** with a `200 OK` response, and the experiment (internally called execution "Job") gets internally (i.e. within the WMI LAN) queued for execution. As mentioned before, everything that takes place within the WMI LAN will be treated as a black-box, thus on a general level, the main logic of the steps that follow (internally) include processes such as request validation, job storing and scheduling, network internal communications via various protocols, and finally the actual execution of the quantum experiment on the quantum device. Once the experiment is acknowledged on qib's side, the ***Experiment Status*** is set to `QUEUED`.

At this point, the quantum developer can use methods such as `get_results`, `wait_for_results`, or `query_results` in order to get the current status of the queued experiment. These methods will further trigger a `HTTP POST` request to the ***WMI Backend API***, querying the status of the experiment, by including its `job_id` in the body of

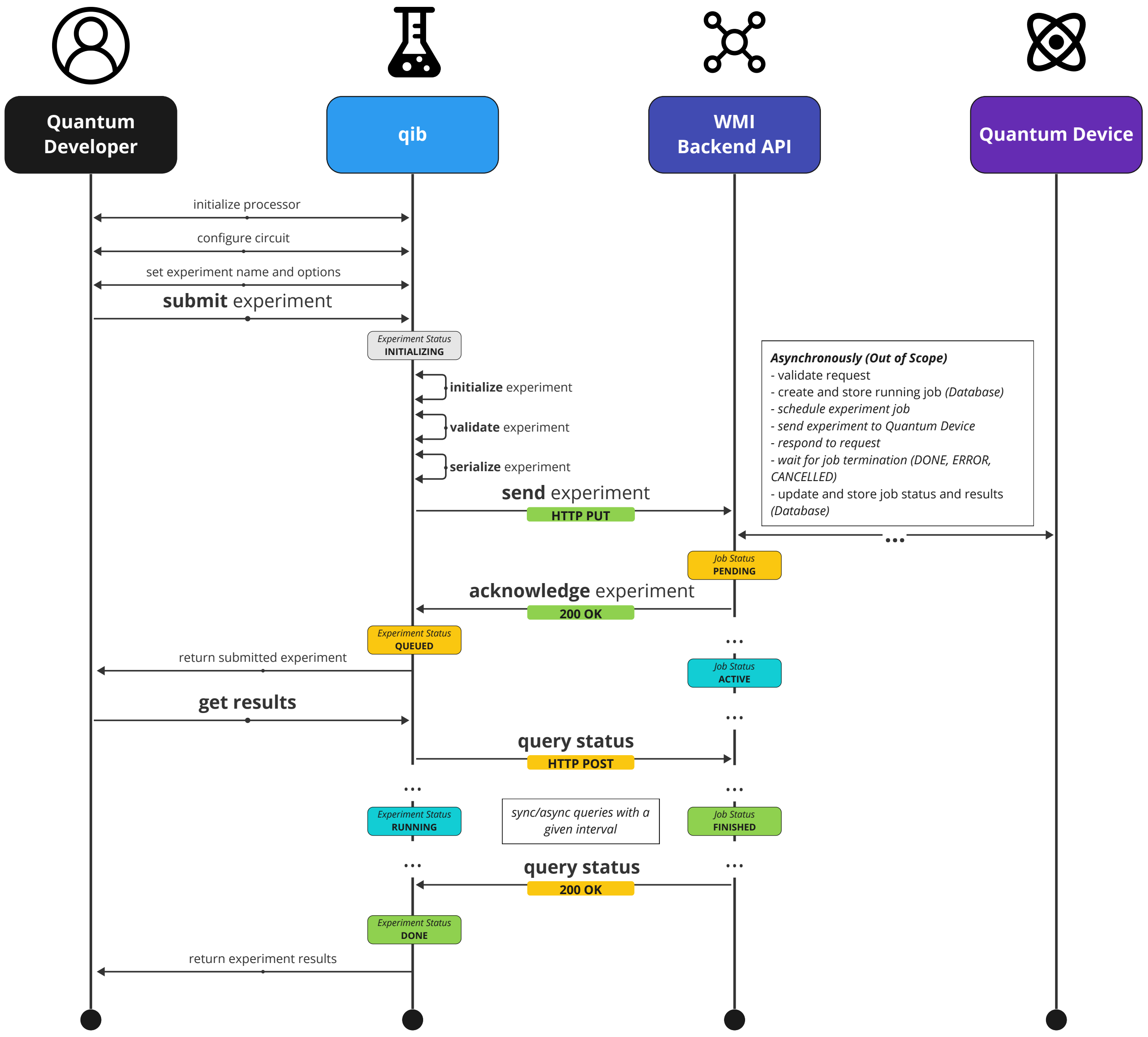

Quantum Developer
qib
WMI Backend API
Quantum Device
initialize processor
configure circuit
set experiment name and options
submit experiment
Experiment Status INITIALIZING
initialize experiment
validate experiment
serialize experiment
send experiment
HTTP PUT
Asynchronously (Out of Scope)
- validate request
- create and store running job (Database)
- schedule experiment job
- send experiment to Quantum Device
- respond to request
- wait for job termination (DONE, ERROR, CANCELLED)
- update and store job status and results (Database)
Job Status PENDING
acknowledge experiment
200 OK
Experiment Status QUEUED
return submitted experiment
Job Status ACTIVE
get results
query status
HTTP POST
Experiment Status RUNNING
sync/async queries with a given interval
Job Status FINISHED
query status
200 OK
Experiment Status DONE
return experiment results

Figure 6.5.: The default networking flow of successfully running a quantum experiment in qib, using the WMI provider interface.

the request. The API will then respond with a `200 OK` acknowledgment response, containing the current status of the experiment, and the ***Experiment Status*** will be updated accordingly.

| **Experiment Status (qib)** | **Job Status (WMI)** | **Description** |
|---|---|---|
| `INITIALIZING` | — | Experiment is being initialized and hasn't yet been sent to the quantum backend. |
| `QUEUED` | `PENDING` | Experiment has been successfully initialized, validated, serialized, and sent to the quantum backend, where is waiting for execution. |
| `RUNNING` | `ACTIVE` | Experiment is being executed. |
| `DONE` | `FINISHED` | Experiment was successfully executed, and results are available. |
| `CANCELLED` | `CANCELLED` | Experiment execution has been cancelled. |
| `ERROR` | `OFFLINE` or *`unknown`* | Experiment initialization or execution encountered an error, and a corresponding error message is available. |

Table 6.2.: All the possible states of an experiment in qib, and their corresponding states in the WMI provider.

As noticed from the figure, the status of the experiment (qib) and of the job (WMI) goes through different stages, these have different names and meaning for qib as they have for WMI, hence a more detailed description and one-to-one mapping can be found in Table 6.2.

Once the experiment is executed successfully, the ***WMI Backend API*** will respond with a `200 OK` acknowledgment response, containing the results of the experiment in the body of the response. The ***Experiment Status*** will be set to `DONE`, and the results will be stored in a `ExperimentResults` object, which will be returned to the quantum developer. The results can then be further processed, analyzed, and visualized by the developer.

## Networking Flow of an Invalid Quantum Experiment

As seen in Figure 6.6, if the submitted experiment is invalid or contains misconfigured serialization that still passes the $qib_{BACK}$'s internal logic and gets sent to the ***WMI Backend API***, the API will respond with a `500 Internal Server Error` acknowledgment response to the initial `HTTP PUT` request, indicating that the submitted experiment is invalid.

This scenario can also be reproduced if the backend is offline, but in this case, the acknowledgment response will contain the `status: 'offline'` property in the message body. In both cases, the ***Experiment Status*** will be set to `ERROR`, and the corresponding error message will be available for the quantum developer to query.

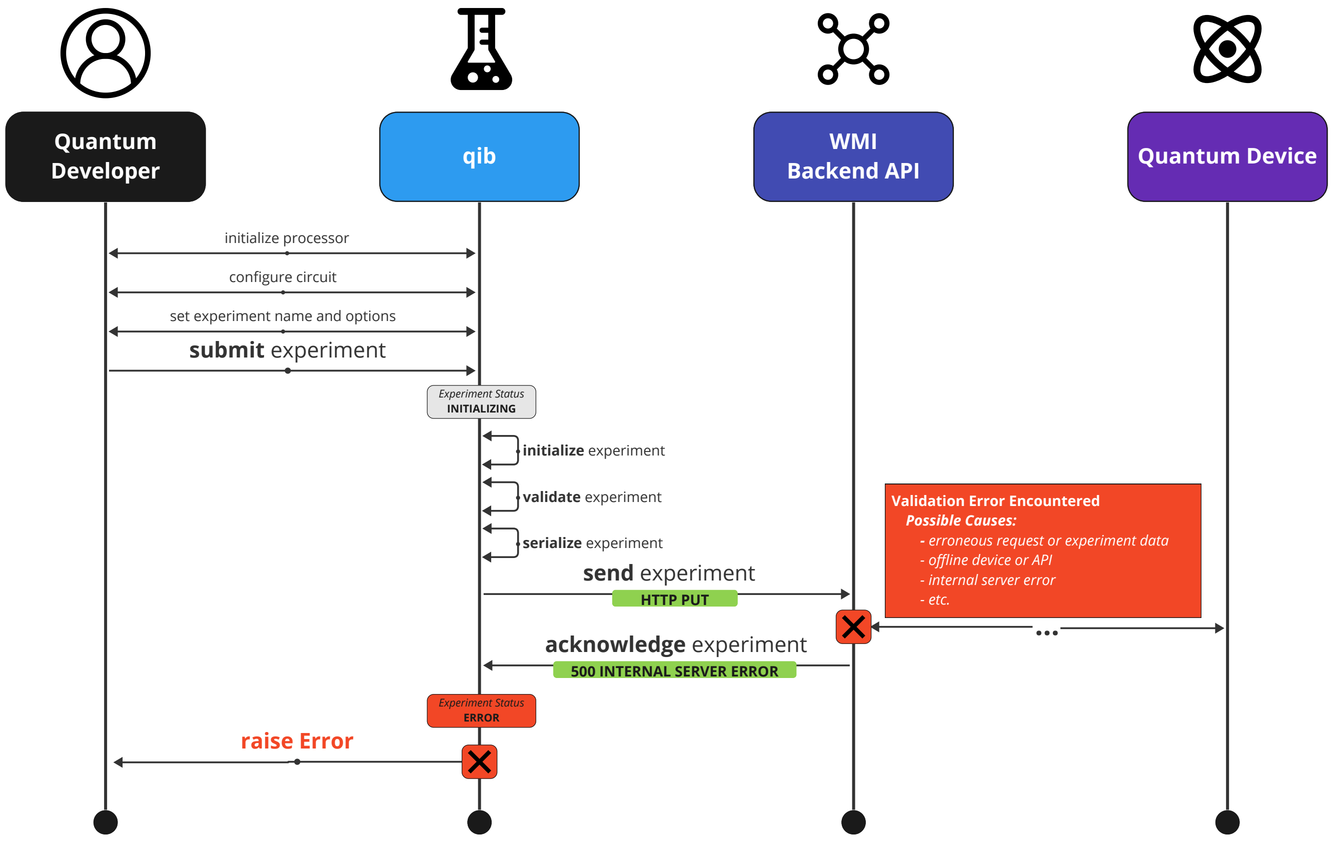


Figure 6.6.: The networking flow of running an invalid quantum experiment in qib, using the WMI provider interface.

### Networking Flow of a Quantum Experiment encountering a Runtime Error

However, if a runtime error is encountered, as seen in Figure 6.7, the ***WMI Backend API*** will (at some point) respond with a `500 Internal Server Error` acknowledgment response to the `HTTP POST` request used for querying the status of the (successfully) submitted experiment. The occurance of this scenario is rather rare, but it can happen due to various reasons, such as network issues, backend overload, or internal errors in the WMI LAN.

### Networking Flow of Canceling a Running Quantum Experiment

Although not yet implemented (i.e. at the moment of writing this thesis), Figure 6.8 shows the potential networking scenario of canceling an already running quantum experiment.

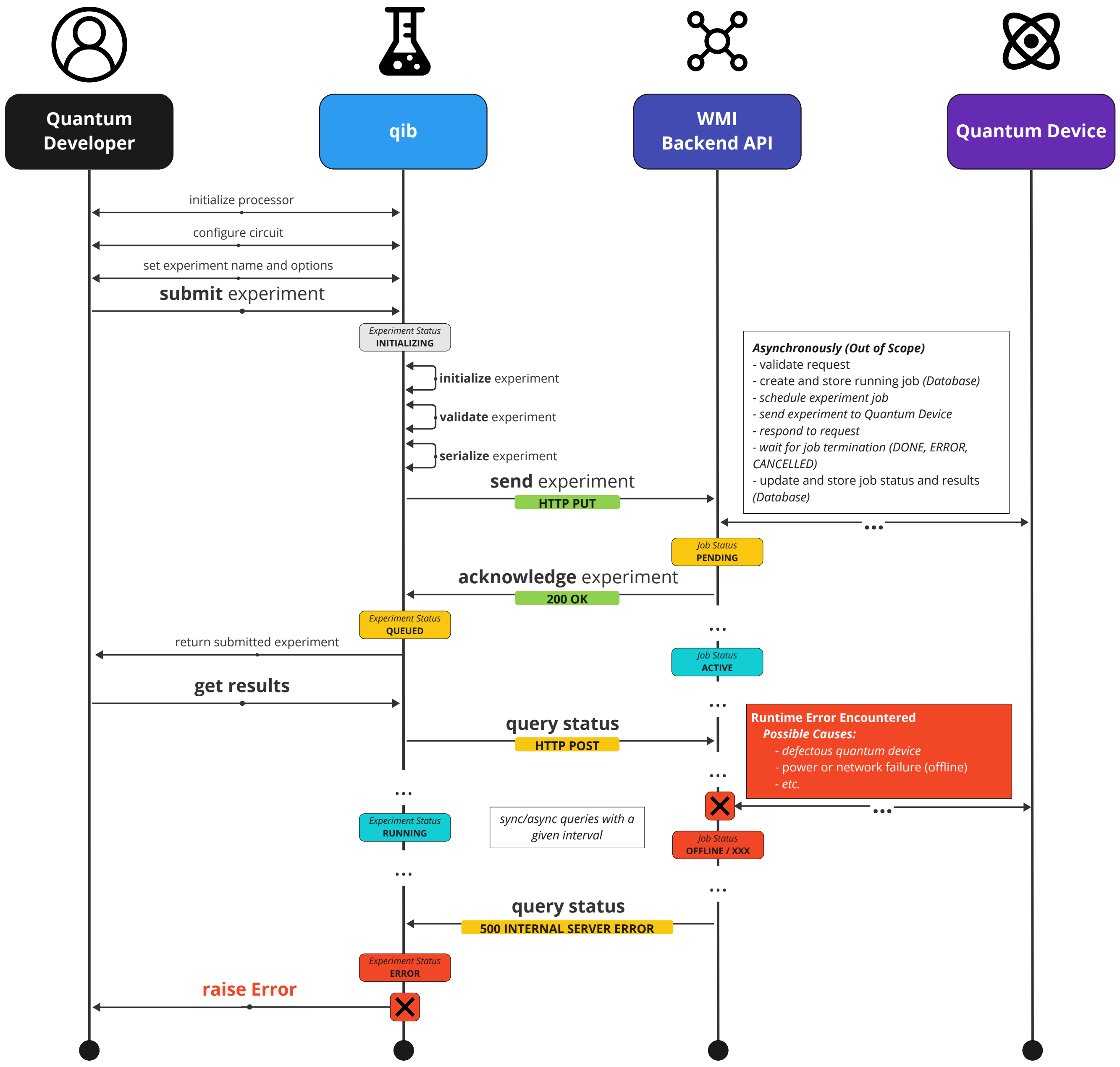

Quantum Developer
qib
WMI Backend API
Quantum Device
initialize processor
configure circuit
set experiment name and options
**submit** experiment
Experiment Status INITIALIZING
**initialize** experiment
**validate** experiment
**serialize** experiment
**send** experiment
HTTP PUT
*Asynchronously (Out of Scope)*
- validate request
- create and store running job *(Database)*
- *schedule experiment job*
- *send experiment to Quantum Device*
- *respond to request*
- *wait for job termination (DONE, ERROR, CANCELLED)*
- update and store job status and results *(Database)*
Job Status PENDING
**acknowledge** experiment
200 OK
Experiment Status QUEUED
return submitted experiment
Job Status ACTIVE
**get results**
**query status**
HTTP POST
Runtime Error Encountered
*Possible Causes:*
- *defectous quantum device*
- power or network failure (offline)
- *etc.*
Experiment Status RUNNING
*sync/async queries with a given interval*
Job Status OFFLINE / XXX
**query status**
500 INTERNAL SERVER ERROR
Experiment Status ERROR
raise Error

Figure 6.7.: The networking flow of running a quantum experiment that encounters a runtime error in qib, using the WMI provider interface.

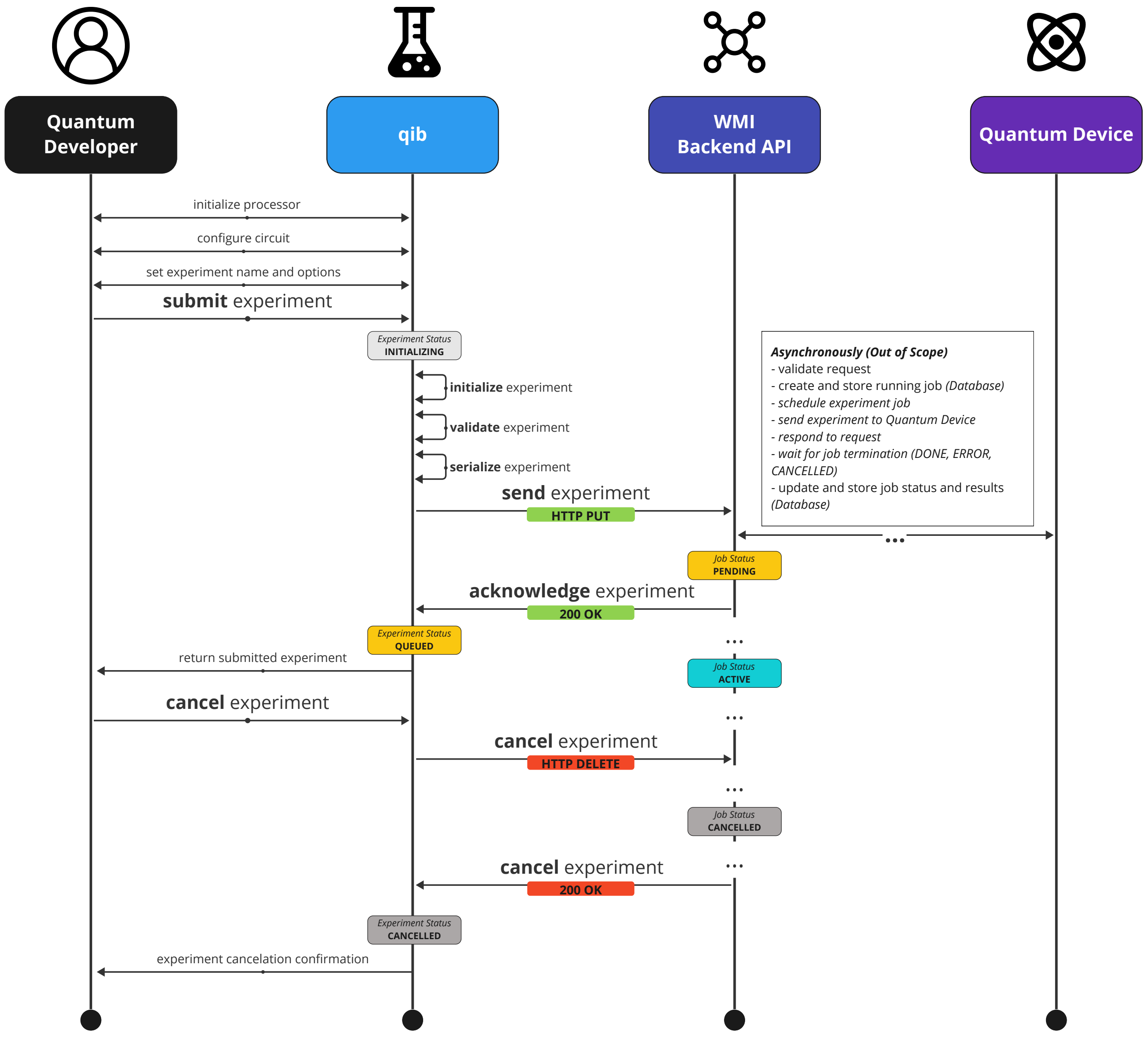


Figure 6.8.: The networking flow of canceling a running quantum experiment in qib, using the WMI provider interface.

This scenario could be reproduced by calling the `cancel` method of the `Experiment` object, which will send a `HTTP DELETE` request to the ***WMI Backend API***, with the `job_id` of the running experiment. The API will then respond with a `200 OK` acknowledgment response, and the ***Experiment Status*** will be set to `CANCELLED`. The results of the experiment will not be available, and the experiment will be removed from the queue of the quantum backend.

## 6.5. Other Extensions to qib

In the process of extending qib for backend communications, and more specifically for the WMI provider, some other extensions or features to the already existing $qib_{CORE}$ were also necessary. In this section, some of these extensions will be briefly described, as follows:

1. **New Gates:** Since the WMI backends provide some gates that were not implemented by default in $qib_{CORE}$, these had to be implemented as children of the `Gate` base class, in the `qib.operator` Python module of $qib_{CORE}$. The implemented gates are: `Identity Gate`, `iSwap Gate`, and the `SX Gate` (see Table 6.3).

2. **Control Instructions:** Although not part of the OpenQASM2 standard (see subsection 3.4.1), the `barrier` and `delay` control instructions (part of OpenQASM3) were also provided by the WMI backends, and thus had to be added to $qib_{CORE}$ (see Table 6.3). This was achieved by creating a new deriving base class `ControlInstruction` of the `AbstractOperator` base class. This newly created base class was also used in order to implement the `measure` control instruction, which is used in declaring a circuit, to specify where and what qubits should be measured (i.e. projective measurement), and in which classical register(s) should the result(s) be stored.

3. **OpenQASM Serialization:** Since OpenQASM serialization was necessary for gates, circuits, and experiments to be sent to the WMI backends, new `as_qasm()` methods have been implemented for the `Gate`, `Circuit`, and `Experiment` classes. These methods are responsible for serializing the given object to the corresponding OpenQASM standard (see Figure 3.6).

4. **Networking:** As a requirement, for interacting with the designated backends over the network, a simple `networking` submodule was added to the `util` module in $qib_{CORE}$, containing the necessary logic for performing HTTP requests and handling responses, as well as for retrying requests on failure and handling error messages. This submodule is used by the `QuantumProcessor` and `Experiment` classes in $qib_{BACK}$, in order to send and receive data to and from the quantum backends.

## 6.6. Extending qib for future providers

As both an academic as well as a research-focused toolkit, qib should be designed with future quantum providers in mind (and not solely rely on the WMI backends available in

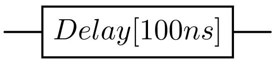
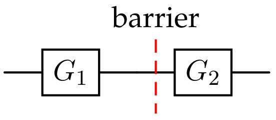
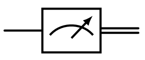
| Quantum Operation | Circuit Representation | Description |
|---|---|---|
| **Identity Gate**<br>**iSwap Gate**<br>**SX Gate** | See Table 2.1 | |
| **Delay Instruction** |  | A quantum instruction used to introduce a controlled delay in the execution of subsequent operations. |
| **Barrier Instruction** |  | A quantum instruction used to prevent certain qubits from being affected by subsequent operations, ensuring their state remains unchanged. |
| **Measure Instruction** |  | A quantum instruction that collapses the superposition of a qubit's state to a classical state, providing an outcome that can be observed and recorded. |

Table 6.3.: The newly implemented gates and control instructions in $qib_{CORE}$, as children of the `Gate` and `ControlInstruction` base classes.

the present). One more reason for this extensibility property is represented by the very nature of quantum computing itself, which is still undergoing a lot of changes, and advancements, as well as new technologies and providers emerging, that could potentially be integrated into qib.

This section will dive deeper into the topic of extending qib for such future quantum providers and more specifically enabling the current architecture of $qib_{BACK}$ for the backends of these future providers. In this context, ***Scenario 3*** from Figure 6.3 becomes particularly relevant. The section will explore this scenario, covering the architectural decisions as well as the implementations that need to be performed in order to properly extend the module in the future.

In order to extend $qib_{BACK}$ for a future quantum provider, the following steps must be taken:

1. **Create the provider sub-module.** Create a new folder with the provider name, under `src/backend`, and create a new `__init__.py` file within the folder, to initialize the Python module. All the following Python implementations should be also declared within this file (see Source Code 6.2).

2. **Implement $qib_{BACK}$'s base classes.** Implement the base classes provided by $qib_{BACK}$, `Options`, `Experiment`, `ExperimentResults`, for the corresponding provider (see

```
from qib.backend.myprov.myprov_options import MyProvOptions
from qib.backend.myprov.myprov_experiment import MyProvExperiment,
                                                 MyProvExperimentResults

# Backend A
from qib.backend.myprov.myprov_backend_a import MyProvBackendA

# Backend B
from qib.backend.myprov.myprov_backend_b import MyProvBackendB

# ...
```

Source Code 6.2.: Template content of the `__init__.py` file for a newly-added provider submodule in $qib_{BACK}$.

Source Code 6.3 and Source Code 6.4). Optionally, these classes can also be implemented for each backend separately (if this is necessary in the given context).

3. **Implement the corresponding processor classes for the given backends.** For each new backend of the provider, implement the corresponding processor class, extending the `QuantumProcessor` base class, and implementing the required methods for submitting the experiments to the given backend (see Source Code 6.5).

```python
from qib.backend.options import Options

class MyProvOptions(Options):
def __init__(self,
                option1: str = "default_value1", # required option
                option2: str = None # optional option
                # ...
            ):
    self.option1: str = option1
    self.option2: str = option2

def optional(self) -> dict:
    optional: dict = {}
    if self.option2: optional['option2'] = self.option2
    return optional
```

Source Code 6.3.: Template of the `Options` class implementation for a newly-added provider submodule in $qib_{BACK}$.

```python
from qib.circuit import Circuit
from qib.backend.myprov import MyProvOptions
from qib.backend import ExperimentStatus,
                        Experiment,
                        ExperimentResults,
                        ExperimentType

class MyProvExperiment(Experiment):
def __init__(self,
            circuit: Circuit,
            options: MyProvOptions,
            exp_type: ExperimentType)
    self.circuit: Circuit = circuit
    self.options: MyProvOptions = options
    self.exp_type: ExperimentType = exp_type
    self._initialize()
    self._validate()

def results(self) -> MyProvExperimentResults | None:
    # implement accordingly

# ...


class MyProvExperimentResults(ExperimentResults):
# ...
```

Source Code 6.4.: Template of the `Experiment` and `ExperimentResults` classes implementations for a newly-added provider submodule in $qib_{BACK}$.

```python
from qib.circuit import Circuit
from qib.backend import QuantumProcessor, ProcessorConfiguration
from qib.backend.myprov import MyProvOptions, MyProvExperiment

class MyProvBackendA(QuantumProcessor):
    def __init__(self):
        # implement accordingly

    @staticmethod
    def configuration() -> ProcessorConfiguration:
        return ProcessorConfiguration(
            backend_name = 'BackendA'
            backend_version = 'v1.5.6'
            # ...
        )

    def submit_experiment(circuit: Circuit,
                          options: MyProvOptions = MyProvOptions()
                          ) -> MyProvExperiment:
        # implement accordingly
```

Source Code 6.5.: Template of the `QuantumProcessor` class implementation for a newly-added backend of a newly-added provider submodule in $qib_{BACK}$.

# Part III.

# Experiments, Results, and Conclusions

# 7. Experimental Results and Tests

This chapter will dive deeper into the actual tests and experiments that have been performed on the implementation of $qib_{BACK}$. It will start by describing some automated tests and basic experiments that were initially performed while implementing the logic of $qib_{BACK}$. Concluding with the most relevant and complex experiment performed in the context of this thesis, a hybrid quantum-classical machine learning experiment. It was the first successful hybrid quantum-classical algorithm to be run on the cloud interface provided by the WMI infrastructure, and the results were promising, thus testing the potential and robustness of both qib and WMI architectures.

## 7.1. Testing the implementation

This section will shortly cover the automated tests performed on the $qib_{BACK}$ implementation. These tests mainly consist of automated Unit Tests (UTs), that were performed using the Python `unittest` framework. Within qib, UTs are categorized in separate files depending on the qib module that they are testing, containing separate test cases with separate scenarios, covering the (potential) utilization scenarios of the qib toolkit.

For testing the logic implemented within $qib_{BACK}$ the `test_back.py` file was extended with the following scenarios:

- Performing experiments on the **WMI Qiskit Simulator** processor:

  - Defining fields, qubits, gates, and circuits.

  - Initializing, validating, and submitting experiments.

  - Querying, retrieving, and validating mock results.

- Performing experiments on the **WMI Qiskit Quantum Computer** processor:

  *- Same scenarios as for the WMI Qiskit Simulator processor.*

- Testing the validation steps of various `Experiment` objects in different scenarios:

  - Validating the **"Shots exceeded"** scenario (i.e. the submitted number of shots does not exceed the maximum number of shots allowed on the processor).

  - Validating the **"Gate not supported"** scenario (i.e. used gates are supported by the processor).

- Validating the **"Gate qubits not configured"** scenario (i.e. used gates are configured for the used qubits in the system of the processor).

- Validating the **"Gate parameters not configured"** scenario (i.e. the parametrized gates used in the experiment use the correct parameters, configured by the processor).

- Validating the **"Number of qubits exceeded"** scenario (i.e. the number of qubits used in the experiment does not exceed the maximum number of qubits allowed on the processor).

For simulating the networking responses received by $qib_{BACK}$ in a real-life scenario, mock data was used, in the form of JSON object files pre-loaded into Python dictionaries, before performing the corresponding tests. Calls to mocked methods are then used in order to simulate the behavior (i.e. retrieve the mocked responses) of the actual backends (i.e. APIs). In this way, $qib_{BACK}$ can be tested without the need for an actual backend connection, or cumbersome integration tests, that would require the actual quantum backends (i.e. devices) to be up and running.

## 7.2. Basic Experiments

As part of the testing and experimentation stage, various basic experiments were performed using the implemented interface, in order to validate, fine-tune, and optimize the logic of $qib_{BACK}$. These experiments were performed on both the WMI Qiskit Simulator and the WMI Quantum Computer processors (with some of them being performed exclusively on the simulator, because of the restrictions imposed by the quantum processor).

One of the first basic experiments that were performed was trying to reproduce all 4 Bell States [6] on the WMI Qiskit Simulator backend.

Given:

$$|\beta_{xy}\rangle \equiv \frac{|0,y\rangle + (-1)^x|1,\hat{y}\rangle}{\sqrt{2}}, \text{ where } \hat{y} \text{ is the negation of } y. \tag{7.1}$$

The 4 Bell States are defined as follows:

$$\begin{aligned} |\Phi^+\rangle &= \frac{|00\rangle + |11\rangle}{\sqrt{2}} \qquad & |\Phi^-\rangle &= \frac{|00\rangle - |11\rangle}{\sqrt{2}} \\ |\Psi^+\rangle &= \frac{|01\rangle + |10\rangle}{\sqrt{2}} \qquad & |\Psi^-\rangle &= \frac{|01\rangle - |10\rangle}{\sqrt{2}} \end{aligned} \tag{7.2}$$

The corresponding circuits (see Figure 7.1) were successfully executed on the simulator, with the results being consistent with the expected Bell States (see Figure 7.2). This is exactly what one would expect on running these circuits on a perfect simulator, where no noise is introduced, but statistical error still exists. See Source Code 7.1 for the qib Python code that has been used for this experiment.

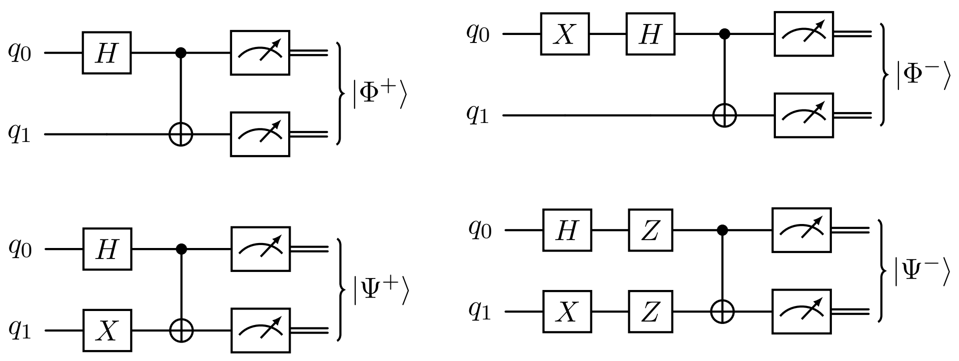


Figure 7.1.: Circuits for generating the four Bell states. For the decomposed versions, used in the experiments, see Figure A.1.

Since not all gates are configured for the WMI machines, some of the experiments and tests performed had to take advantage of the basic decomposition property of quantum gates (i.e. matrices), see Appendix A for more details on how this was specifically applied in the context of the necessary circuits and gates.

Unfortunately, this experiment could not be reproduced on the WMI Quantum Computer, since it requires 2 calibrated qubits to be used, and the WMI Quantum Computer only has one qubit that is calibrated and available for experiments.

As for a basic testing experiment performed on the WMI QPU, a simple implementation of a circuit containing a `SX Gate` was executed (see Figure 7.3). The goal of this experiment was to put the only qubit available and configured within the WMI QPU in a superposition state. The results were successful, but the error rate can be noticed, as opposed to its WMI Qiskit Simulator counterpart (see Figure 7.4). In addition to statistical errors, experimental errors can now be also observed as a direct consequence of the noise present in the quantum computer. This is a common issue in the quantum computing field, with active research and new solutions trying to mitigate it as much as possible. The error rate can be seen in the results of the experiment, as the probabilities of the states are not exactly 0.5, as they should be in the case of a perfect quantum computer. See Source Code 7.2 for the qib Python code that has been used for this experiment.

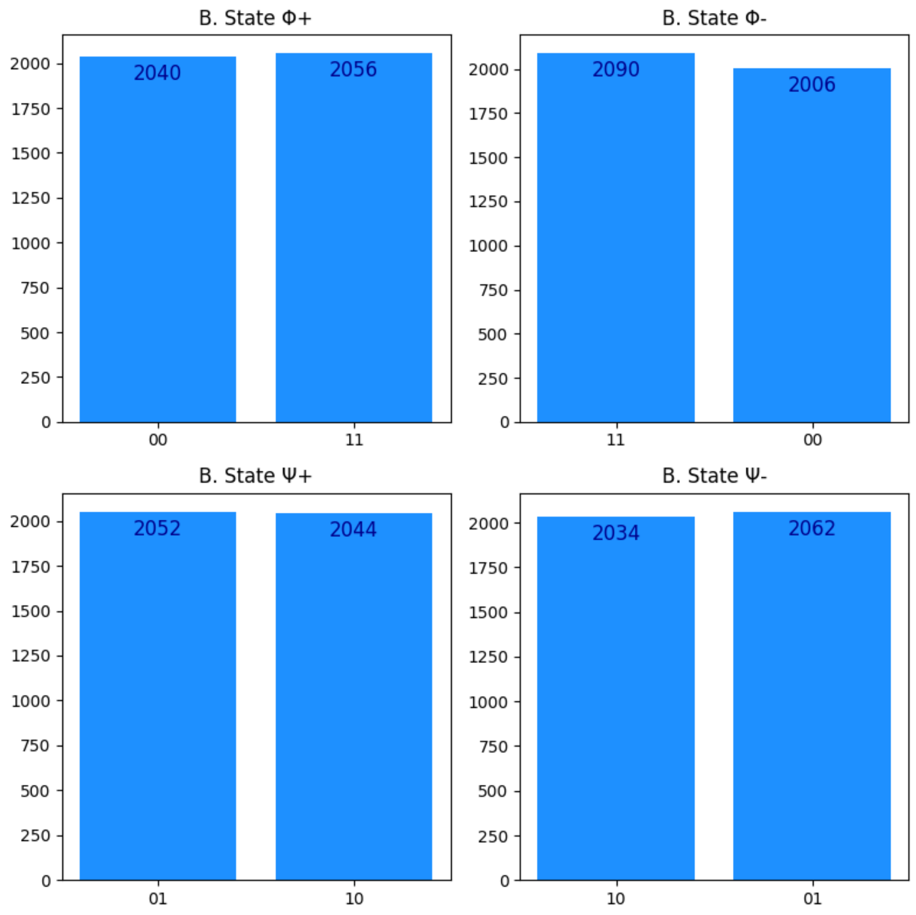


Figure 7.2.: The results (i.e. states distribution) of the Bell States experiment on the WMI Qiskit Simulator.

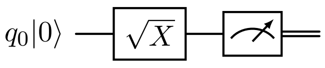


Figure 7.3.: The circuit for the SX Gate experiment on the WMI QPU.

```python
# Defines and returns the 4 Bell States circuits
def wmi_qs_bell_circuits() -> qib.Circuit[]:
    field = qib.field.Field(
    qib.field.ParticleType.QUBIT, qib.lattice.IntegerLattice((3,)))
    q0 = qib.field.Qubit(field, 0)
    q1 = qib.field.Qubit(field, 1)

    # ...

    return [circuit_phi_plus, circuit_phi_minus,
            circuit_psi_plus, circuit_psi_minus]

# Initializes the backend, submits the given circuit,
# and returns the results
def wmi_qs_experiment(circuit: qib.Circuit, access_token: str) ->
                                qib.backend.ExperimentResults:
    processor = qib.backend.wmi.WMIQSimProcessor(access_token)
    options = qib.backend.wmi.WMIOptions(shots=4096)
    experiment = processor.submit_experiment(
        name = 'qib-integration-test-qs',
        circ = circuit,
        options = options)
    results = experiment.results()
    return results

# Plots the state distribution of the given results
def wmi_qs_plot_states(results_counts: list[dict]) -> None:
    # ...

# Initialize the circuits and run the experiment
bell_circuits = wmi_qs_bell_circuits()
results_counts: list[dict] = []
for circuit in bell_circuits:
    results = wmi_qs_experiment(circuit, access_token)
    results_counts.append(results.get_counts(binary=True))

# Plot experiment results
wmi_qs_plot_states(results_counts)
```

Source Code 7.1.: Python code for running the Bell States experiment on the WMI Qiskit Simulator.

```python
# Defines and returns the Sx circuit
def wmi_qc_circuit():
    field = qib.field.Field(
        qib.field.ParticleType.QUBIT, qib.lattice.IntegerLattice((3,)))
    q0 = qib.field.Qubit(field, 0)

    circuit = qib.Circuit([
        qib.SxGate(q0),
        qib.MeasureInstruction([q0])
    ])
    return circuit

# Initializes the backend, submits the given circuit,
# and returns the results
def wmi_qc_experiment(circuit: qib.Circuit, access_token: str) ->
                                qib.backend.ExperimentResults:
    processor = qib.backend.wmi.WMIQCProcessor(access_token=access_token)
    experiment = processor.submit_experiment(
        name = 'qib-integration-test-qc',
        circ = circuit)
    results = experiment.results()
    return results

# Plots the state distribution of the given results
def wmi_qc_plot(counts: dict) -> None:
    # ...

# Initialize the circuits and run the experiment
circuit = wmi_qc_circuit()
results = wmi_qc_experiment(circuit, access_token).get_counts(binary=True)

# Plot experiment results
wmi_qc_plot(results)
```

Source Code 7.2.: Python code for running the SX Gate experiment on the WMI QPU.

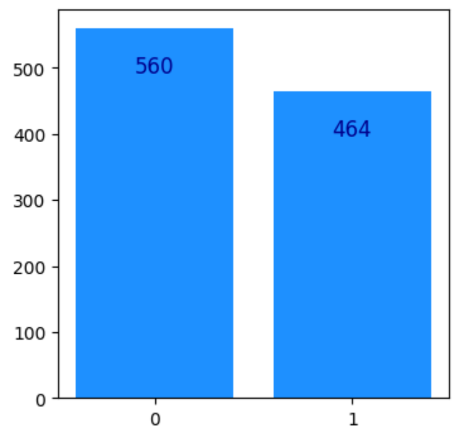


Figure 7.4.: The results (i.e. states distribution) of the SX Gate experiment on the WMI QPU. The error rate can be noticed in the count's difference of the states.

## 7.3. Hybrid Quantum-Classical Experiment

**Quantum State Tomography (QST)** is the procedure used to experimentally determine an unknown quantum state. Considering an unknown state $|\psi\rangle$ of a single qubit, how can one experimentally ascertain the state of $|\psi\rangle$? If only a single copy of $|\psi\rangle$ is available, it is impossible to fully characterize it. This is because the qubit state is hidden before measurement (i.e. any measurement could be the result of an infinite set of qubit states). However, if a large number of copies of $|\psi\rangle$ is available, it becomes possible to estimate the state. For example, if $|\psi\rangle$ is the quantum state produced by an experiment, one can repeat the experiment multiple times to generate many copies of the state $|\psi\rangle$.

QST is used to estimate what the state of a given quantum system is, an indispensable technique necessary for realizing **Quantum Process Tomography (QPT)**, that aims to characterize the dynamics of a quantum system. Together, QST and QPT offer a comprehensive toolkit for quantum system characterization, being closely interdependent in terms of the accuracy and reliability of the overall results.

This experiment aimed to implement a single-qubit Quantum State Tomography process on both WMI backends, by using a gradient descent algorithm for learning the quantum state of the system. The main process consisted of training on the parameters of a universal quantum rotation gate, in order to reproduce (as accurately as possible) the initial randomly pre-generated quantum state of the system.

The experiment was structured in 3 main stages (initialization, training, and testing), which will be described as follows, together with the set-up, algorithms, and mathematical concepts that were necessary for realizing each stage. After that, the results of the experiment will be presented, together with the conclusions, comments, and comparisons between the two backend systems.

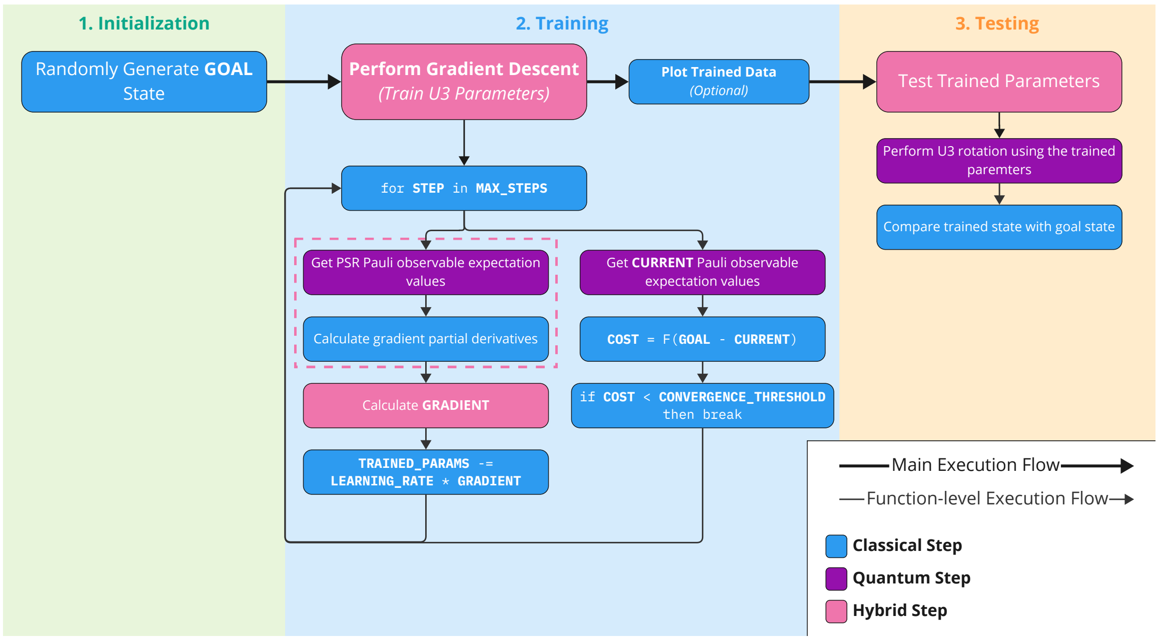

1. Initialization
Randomly Generate GOAL State
2. Training
Perform Gradient Descent (Train U3 Parameters)
Plot Trained Data (Optional)
for STEP in MAX_STEPS
Get PSR Pauli observable expectation values
Calculate gradient partial derivatives
Calculate GRADIENT
TRAINED_PARAMS -= LEARNING_RATE * GRADIENT
Get CURRENT Pauli observable expectation values
COST = F(GOAL - CURRENT)
if COST < CONVERGENCE_THRESHOLD then break
3. Testing
Test Trained Parameters
Perform U3 rotation using the trained paremters
Compare trained state with goal state
Main Execution Flow
Function-level Execution Flow
Classical Step
Quantum Step
Hybrid Step

Figure 7.5.: The steps performed in the implemented Hybrid Quantum-Classical Algorithm, and their nature.

The diagram in Figure 7.5 illustrates the nature of the Hybrid Quantum-Classical Algorithm implemented for this experiment, highlighting the clear distinction between the quantum, classical, and hybrid steps involved in the algorithm.

### Stage 1: Initialization

In this initial stage of the experiment, a random quantum state is generated, by generating two random rotation angles ($\theta$ and $\phi$) in spherical coordinates and converting them to their corresponding cartesian coordinates. This is the goal state that the algorithm should be able to learn and reproduce, by training on the parameters of a universal quantum rotation gate (see Figure 7.6).

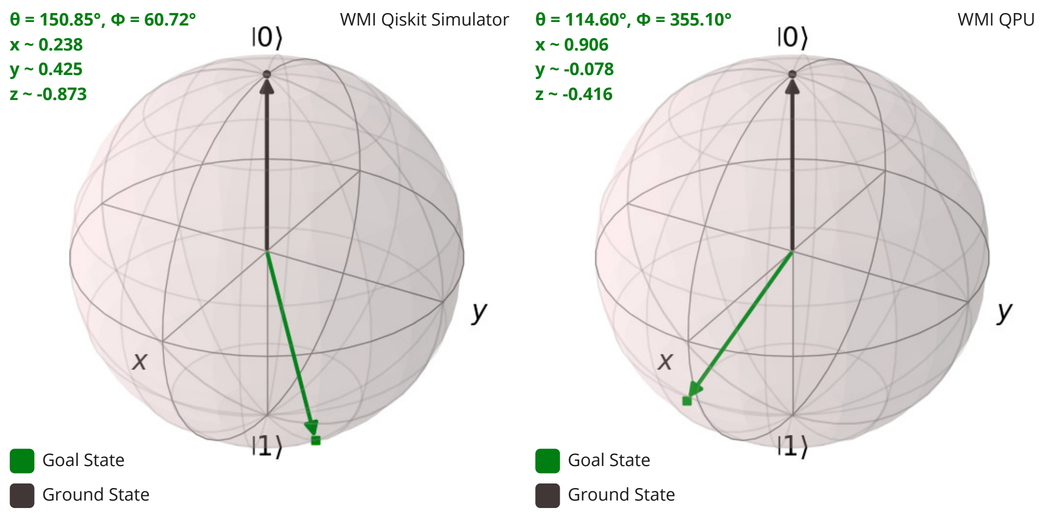


Figure 7.6.: An example of two randomly generated initial quantum states for the Hybrid ML experiment on the WMI Qiskit Simulator (left) and the WMI Qiskit Quantum Computer (right).

## Stage 2: Training

This is the main stage of the experiment, consisting of an implementation of the gradient descent algorithm [54] performed on the parameters $(\theta, \phi, \lambda)$ of a universal quantum rotation gate, more exactly the $U3$ gate (see Table 7.1). Since this gate is neither configured nor available on the WMI backends, gate decomposition was performed (see Figure 7.7), in order to decompose the $U3$ gate into a sequence of elementary gates that are supported by the backends (for more information see Appendix A).

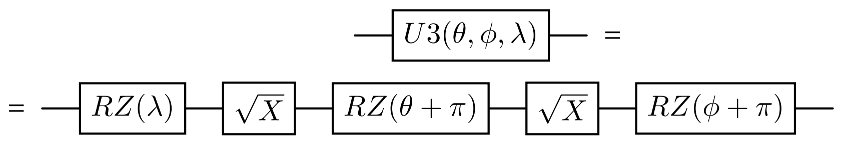


Figure 7.7.: Decomposing the $U3$ gate to WMI-available gates.

For the gradient descent algorithm, a simple **Squared Euclidean Distance** function was used as the cost function $F$, which measures the difference between the goal state and the current state of the system.

$$F = \sum_{k \in \{x,y,z\}} u_k^2 \quad \text{where} \quad u_k(k_{\text{goal}}, k_{\text{current}}) = k_{\text{goal}} - k_{\text{current}}, \quad \text{for } k \in \{x, y, z\} \tag{7.3}$$

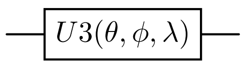
| **Circuit Representation** | **Matrix Representation** |
|---|---|
|  | $U3(\theta,\phi,\lambda) = \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -e^{i\lambda}\sin\left(\frac{\theta}{2}\right) \\ e^{i\phi}\sin\left(\frac{\theta}{2}\right) & e^{i(\phi+\lambda)}\cos\left(\frac{\theta}{2}\right) \end{pmatrix}$ |

Table 7.1.: The $U3$ universal single-qubit quantum rotation gate, with 3 Euler angles.

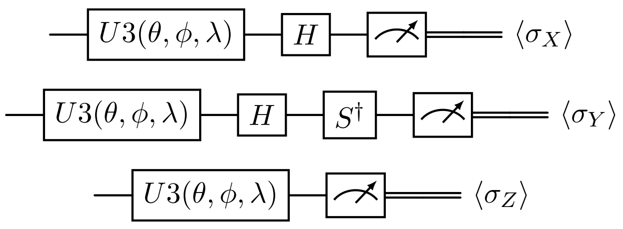


Figure 7.8.: The circuits for the Pauli X, Y, and Z post-rotation measurements.
***(Note: Decomposed versions have been used for PauliX and PauliY, in the actual experiment)***

Here, $k_{current}$ can be calculated as the expectation value of a Pauli observable, resulting in Equation 7.6. Because the expectation values of each Pauli observable depend on the circuit state, they are functions of $\theta, \phi, \lambda$.

The end-goal was to minimize this cost function $F$, and to get as close as possible to the goal state.

$$Min\{F(\theta_s,\phi_s,\lambda_s)\} \tag{7.4}$$

In order to achieve this, the gradient

$$\nabla F = \begin{pmatrix} \frac{\partial F}{\partial \theta} \\ \frac{\partial F}{\partial \phi} \\ \frac{\partial F}{\partial \lambda} \end{pmatrix} \tag{7.5}$$

was calculated at each iteration step of the algorithm by using repetitive rotations (i.e. the decomposed $U3$ gate circuit) and post-rotation measurements [1] (i.e. measurement statistics in all 3 Pauli bases, see Figure 7.8) of the current system state [2].

[1]For performing the necessary post-rotation measurements, gate decomposition was again necessary, see Appendix A.

[2]The results of the measurements were then scaled and converted from a $|0\rangle$ and $|1\rangle$ distribution to a scaled average of 1's and -1's, since that is the necessary context-relevant value of the Bloch Sphere's cartesian coordinate system.

$$\begin{pmatrix} x_{current} \\ y_{current} \\ z_{current} \end{pmatrix} = \begin{pmatrix} \langle \sigma_x \rangle \\ \langle \sigma_y \rangle \\ \langle \sigma_z \rangle \end{pmatrix} \tag{7.6}$$

The calculated gradient was then plugged into the gradient descent formula in order to update the trained parameters of the $U3$ gate.

$$\begin{pmatrix} \theta_i \\ \phi_i \\ \lambda_i \end{pmatrix} = \begin{pmatrix} \theta_{i-1} \\ \phi_{i-1} \\ \lambda_{i-1} \end{pmatrix} - \alpha * \nabla F, \text{ where } i \text{ represents the current iteration step} \tag{7.7}$$

In order to calculate the gradient, one must calculate the three partial derivatives of it in regards to the $U3$ gate parameters.

$$\frac{\partial F}{\partial \theta}, \frac{\partial F}{\partial \phi}, \text{and } \frac{\partial F}{\partial \lambda} \tag{7.8}$$

First, the chain rule was applied. By taking the formula of the cost function $F$ from Equation 7.3 and considering Equation 7.6, the first partial derivative of the gradient can be formulated as:

$$\frac{\partial F}{\partial \theta} = \frac{\partial F}{\partial u} \cdot \frac{\partial u}{\partial \theta} = \sum_{k \in \{x,y,z\}} \frac{\partial u_k^2}{\partial u_k} \cdot -\frac{\partial \langle \sigma_k \rangle}{\partial \theta} = \sum_{k \in \{x,y,z\}} -2u_k \cdot \frac{\partial \langle \sigma_k \rangle}{\partial \theta} \tag{7.9}$$

For simplicity reasons:

$$\text{Let } \frac{\partial \langle \sigma_k \rangle}{\partial \theta} = T_k \quad \text{,for } k \in \{x, y, z\} \tag{7.10}$$

Then the **Parameter Shift Rule (PSR)** [42] was necessary. PSR is a technique used in quantum machine learning for calculating the gradient of a variational quantum circuit with respect to its parameters. For more information on the Parameter Shift Rule in the context of this thesis, see Appendix B.

Since one seeks to minimize the cost function with respect to the parameters of this variational quantum algorithm, the PSR enables the computation of gradients, i.e. of the partial derivatives needed for the final gradient calculation.

For example, given a quantum circuit with a single parametrized gate $U(\theta)$ and an observable $X$. The expectation value of the observable, where the circuit is in a particular quantum state is given by:

$$E(\theta) = \langle 0|U^\dagger(\theta) X U(\theta)|0\rangle. \tag{7.11}$$

Using the Parameter Shift Rule, the gradient is:

$$\frac{\partial E(\theta)}{\partial \theta} = \frac{1}{2}\left(E(\theta + \frac{\pi}{2}) - E(\theta - \frac{\pi}{2})\right). \tag{7.12}$$

This gradient can be used in the gradient descent update rule to optimize $\theta$. This very same technique has been used to optimize the $\theta, \phi$, and $\lambda$ parameters of the $U3$ gate in the context of this experiment. Lastly, with the PSR constants $c_{PSR} = \frac{1}{2}$ and $s_{PSR} = \frac{\pi}{2}$ [3], $T_k$ from Equation 7.10 can be written as:

$$T_k = \frac{1}{2} \cdot [\langle \sigma_k(\theta_{current} + \pi/2, \phi_{current}, \lambda_{current}) \rangle - \langle \sigma_k(\theta_{current} - \pi/2, \phi_{current}, \lambda_{current}) \rangle], \quad \text{for } k \in \{x, y, z\} \tag{7.13}$$

By expanding the sum in Equation 7.9 and taking into consideration Equation 7.13 the resulting formula for the first partial derivative of the gradient can be written as:

$$\begin{aligned} \frac{\partial F}{\partial \theta} &= \frac{\partial F}{\partial u} \cdot \frac{\partial u}{\partial \theta} = (-2u_x \cdot T_x) + (-2u_y \cdot T_y) + (-2u_z \cdot T_z), \\ &\text{where} \quad T_k = \frac{1}{2} \cdot [\langle \sigma_k(\theta_{\text{current}} + \pi/2, \phi_{\text{current}}, \lambda_{\text{current}}) \rangle - \langle \sigma_k(\theta_{\text{current}} - \pi/2, \phi_{\text{current}}, \lambda_{\text{current}}) \rangle], \\ &\text{for } k \in \{x, y, z\} \end{aligned} \tag{7.14}$$

Concluding, the formulas for the other two partial derivatives of the gradient are:

$$\begin{aligned} \frac{\partial F}{\partial \phi} &= \frac{\partial F}{\partial u} \cdot \frac{\partial u}{\partial \phi} = (-2u_x \cdot P_x) + (-2u_y \cdot P_y) + (-2u_z \cdot P_z), \\ &\text{where} \quad P_k = \frac{1}{2} \cdot [\langle \sigma_k(\theta_{\text{current}}, \phi_{\text{current}} + \pi/2, \lambda_{\text{current}}) \rangle - \langle \sigma_k(\theta_{\text{current}}, \phi_{\text{current}} - \pi/2, \lambda_{\text{current}}) \rangle], \\ &\text{for } k \in \{x, y, z\} \end{aligned} \tag{7.15}$$

$$\begin{aligned} \frac{\partial F}{\partial \lambda} &= \frac{\partial F}{\partial u} \cdot \frac{\partial u}{\partial \lambda} = (-2u_x \cdot L_x) + (-2u_y \cdot L_y) + (-2u_z \cdot L_z), \\ &\text{where} \quad L_k = \frac{1}{2} \cdot [\langle \sigma_k(\theta_{\text{current}}, \phi_{\text{current}}, \lambda_{\text{current}} + \pi/2) \rangle - \langle \sigma_k(\theta_{\text{current}}, \phi_{\text{current}}, \lambda_{\text{current}} - \pi/2) \rangle], \\ &\text{for } k \in \{x, y, z\} \end{aligned} \tag{7.16}$$

As for the hyperpatameters of the algorithm, the values were adapted for each backend individually, in order to optimize the training process (this will be further discussed in section 7.3). The hyperparameters used for the training process are the following:

- **Learning Rate ($\alpha$):** The step size at each iteration of the algorithm.

[3] see section B to understand why

- **Max Steps:** The maximum number of iterations allowed for the algorithm.
- **Convergence Threshold:** The threshold value for the loss function, at which the algorithm should stop.

The training algorithm described above is also programmatically represented in the Python pseudocode at Source Code 7.3.

### Stage 3: Testing

Once the three Euclidean angles (i.e. parameters of the U3 gate) have been successfully trained, with satisfactory accuracy, they are used in stage 3 for testing the actual results, by performing a final rotation on the system and measuring the state in the Pauli X, Y, and Z bases. The results are then compared with the initial goal state, in order to evaluate the accuracy of the trained values. The goal is for the trained state (i.e. vector or cartesian coordinates) to be as close as possible to the initial (randomly generated) state, with a minimal error rate.

### Results

The obtained results were mostly as expected, with the noiseless WMI Qiskit Simulator backend (see Figure 7.9) providing more accurate results, as opposed to the noisy WMI QPU backend (see Figure 7.10). Thus, the hyperparameter values were adjusted accordingly, with a higher convergence criteria tolerance for the WMI QPU backend (0.1 instead of 0.01). After optimizing the hyperparameters, convergence was attained pretty fast, with the cost function usually reaching the convergence threshold in less than 10 iterations, without overshooting, for both backends.

One of the main observations, a notable difference between the two backends, and an unpredicted result was the length of the resulting trained quantum state vector in the experiments concerning the WMI Quantum Processor, which was always less than 1. This is a potential result of the noise present in the quantum system of the device, which can lead to a deviation from the expected results, and can be a direct consequence of such inaccurate/imperfect outcomes.

In terms of convergence steepness/speed, this is very much dependent on the learning rate and the convergence threshold hyperparameters. Thus, on average, the WMI QPU seems to be converging faster than the WMI Qiskit Simulator, this is mainly due to the lower convergence threshold expectation, with the simulator fine-tuning the trained state to a higher degree of accuracy.

```
goal = [RANDOM_X, RANDOM_Y, RANDOM_Z] # The randomly generated quantum state
initial_state = [0,0,0] # Ground State
learning_rate, max_steps, convergence_threshold # Hyperparameters

def F(goal, current) # Cost Function
def PAULI_X(theta, phi, lam) # Pauli X post-rotation measurement
def PAULI_Y(theta, phi, lam) # Pauli Y post-rotation measurement
def PAULI_Z(theta, phi, lam) # Pauli Z post-rotation measurement
# Partial derivative of the cost function with respect to theta
def D_THETA(goal, current)
# Partial derivative of the cost function with respect to phi
def D_PHI(goal, current)
# Partial derivative of the cost function with respect to lambda
def D_LAM(goal, current)

# Calculates and returns the gradient and cost of the current iteration step
def CALCULATE_GRADIENT(theta, phi, lam):
    current = [PAULI_X(theta, phi, lam),
               PAULI_Y(theta, phi, lam),
               PAULI_Z(theta, phi, lam)]
    gradient = [D_THETA(goal, current),
                D_PHI(goal, current),
                D_LAM(goal, current)]
    cost = F(goal, current)
    return gradient, cost

# Performs the gradient descent algorithm and returns the trained parameters
def GRADIENT_DESCENT(trained, learning_rate, max_steps, convergence_threshold
    for step in range(max_steps):
        gradient, cost = CALCULATE_GRADIENT(theta, phi, lam)
        trained -= learning_rate * gradient
        if cost < convergence_threshold:
            break
    return trained

trained_params = GRADIENT_DESCENT(initial_state,
                                  learning_rate,
                                  max_steps,
                                  convergenece_threshold)
```

Source Code 7.3.: Python pseudocode for the Quantum State Tomography training algorithm.

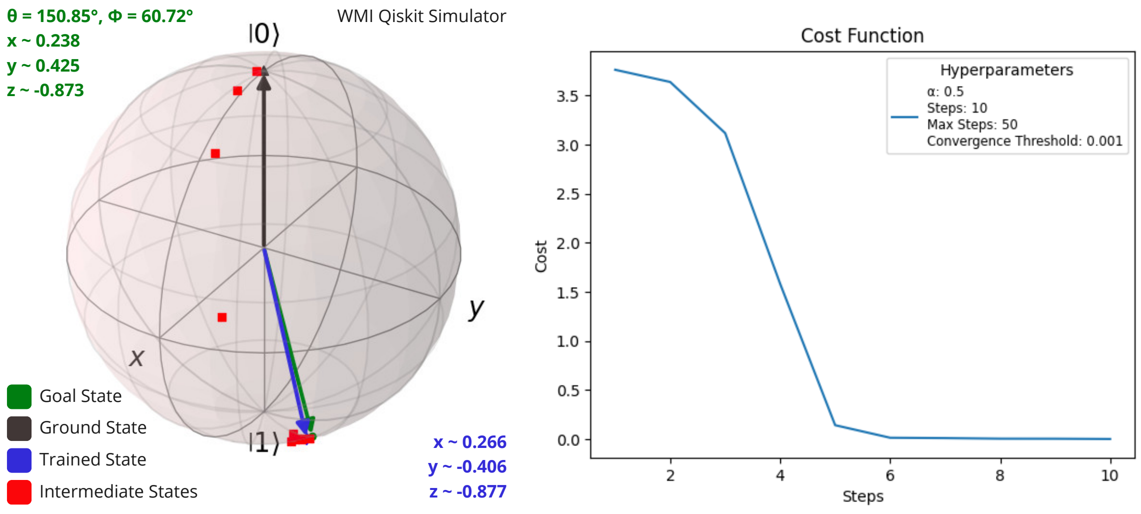


Figure 7.9.: The results of the Hybrid ML experiment on the **WMI Qiskit Simulator**.
**(left)** initial, goal, intermediate, and trained quantum states in Bloch Sphere representation.
**(right)** the evolution of the cost function during training.

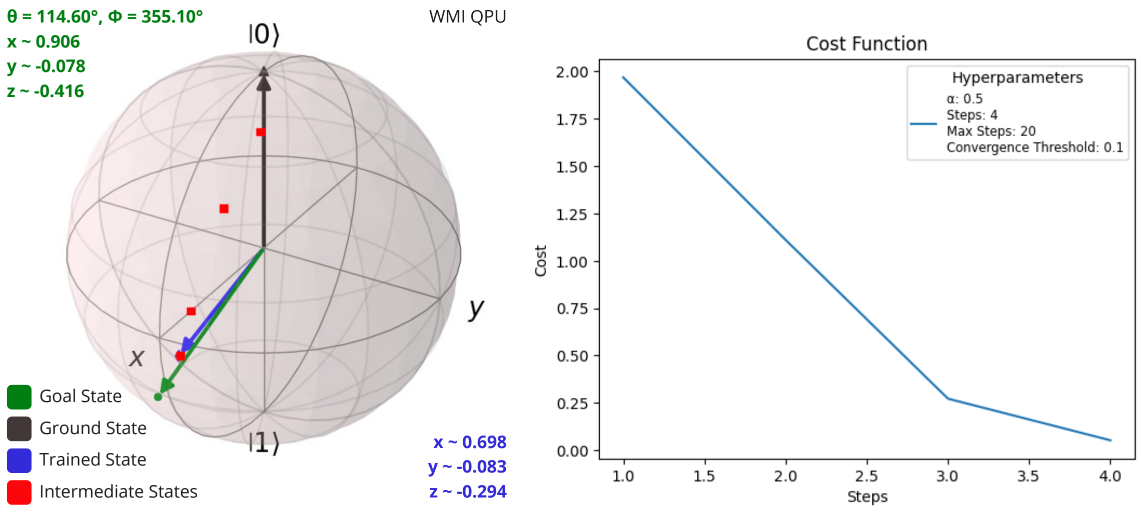


Figure 7.10.: The results of the Hybrid ML experiment on the **WMI QPU**.
**(left)** initial, goal, intermediate, and trained quantum states in Bloch Sphere representation.
**(right)** the evolution of the cost function during training.

### Discussion: Adjusting the Hyperparameters

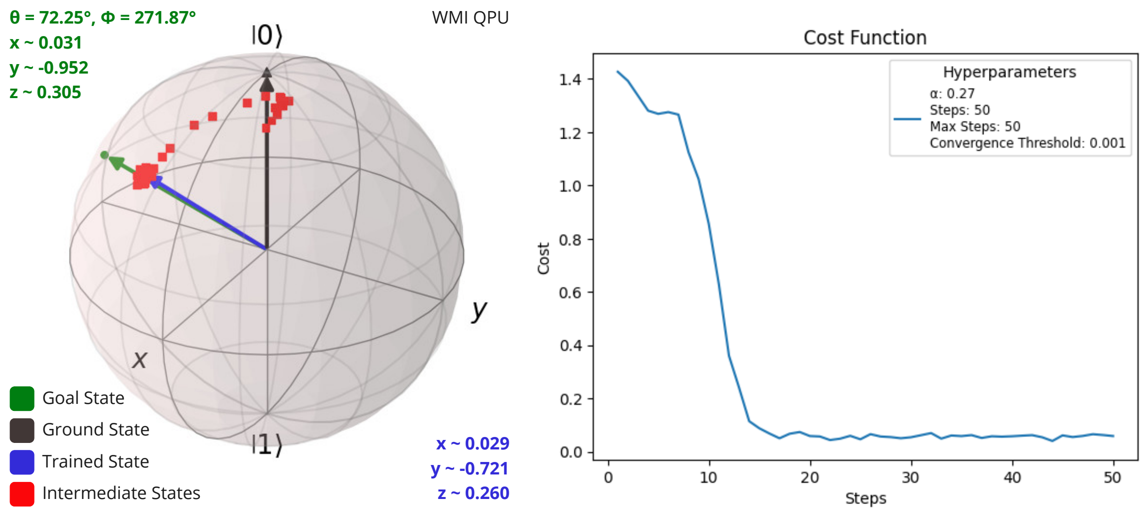


Figure 7.11.: The results of the Hybrid ML experiment on the **WMI QPU**, while still adjusting the hyperparameters.
**(left)** initial (gray), goal (green), intermediate (red), and trained (blue) quantum states in Bloch Sphere representation.
**(right)** the evolution of the cost function during training.

The process of adjusting the hyperparameters was not a straightforward task, especially for the WMI QPU backend, where the noise represented a major factor in the training process. Figure 7.11 depicts one of the runs performed on the WMI QPU backend, while still adjusting the hyperparameters. Here the main problem encountered during training for this backend can be observed, mainly the fact that (with a smaller learning rate and a stricter convergence threshold) the cost function did not converge properly. It attained more accurate results than the one from Figure 7.10, but still not accurate enough to reach the threshold of 0.001. The highest accuracy (i.e. lowest cost function value) reached on the WMI QPU backend was around **0.05**, while the WMI Qiskit Simulator backend was able to reach an accuracy of even **0.0002**.

Nonetheless, the experiment was considered a success, with the results being accurate enough in the provided context, and the infrastructure of $qib_{BACK}$ and WMI proving to be robust and reliable. Multiple iterations of this experiment have been performed on both WMI backends, with the end goal of stress-testing the implemented infrastructure of $qib_{BACK}$, while establishing a concrete, real-life scenario, of a valuable experiment that can be performed on the platform. The platforms proved themselves to be reliable enough for executing over $2^{12}$ shots for each of the 21 jobs performed in each iteration step of an average run, with over 860.160 shots and 210 jobs performed, for a run of 10 steps, in total. These numbers are an additional indicator of the scalability and robustness of the implemented module, and the potential of the WMI backend infrastructure. Moreover, this was the first time that such hybrid machine learning experiment was performed through the web API of the Walther-Meißner-Institute, thus marking another successful result for their infrastructure.

# 8. Conclusions

In conclusion, this thesis explores the topic of quantum software development using networking between various quantum actors through the proposed objective of defining and implementing such an interface for enabling a quantum software development environment between the qib Python library, and the already-available infrastructure of the Walther-Meissner Institute.

The implementation of the qib interface has been successful in providing a clear and concise way to interact with the quantum hardware and simulators at the Walther-Meißner-Institute. The interface has been designed to be as simple as possible, while still providing the necessary functionality and customizability for the user. It has been implemented in a way that allows for easy integration with existing quantum software development tools and providers, such as Qiskit simulators or the WMI superconducting QPUs, and can be easily extended to support even more providers and backends in the future.

Furthermore, the implementation was stress-tested through a hybrid iterative quantum-classical experiment, which ended up converging on a solution, with decent accuracy, on a real quantum hardware backend provided by the Walther-Meißner-Institute. The experiment was a success, proving that the implemented infrastructure is consistent and reliable, from a technical standpoint.

The infrastructure has been also unit- and integration-tested, undergoing multiple tests and experimentation procedures, in both real-life and mocked scenarios. The results have been consistent and the interface has been proven to be reliable and robust. Some statistical and experimental errors have been noticed, especially when executing quantum circuits on actual quantum hardware, which are normal and expected in the field. It is possible to still mitigate and correct a bigger proportion of them in the future, by implementing features such as quantum error correction or other transpilation/compilation methods that might improve the accuracy and reliability of the platform.

More specifically, $qib_{BACK}$ can be further extended with features, such as: auto-mapping qubits and scheduling gates based on the physical constraints of the backend, quantum error correction, proper transpilation (including circuit optimization, gate decomposition, routing on restricted topology, translation to basis gates, etc.), and various other optimization techniques (pulse-level optimization, resource estimation and management, dynamic circuit reconfiguration, etc.).

# Appendix

# A. Quantum Gate Decomposition

**Quantum Gate Decomposition** is the process of breaking down complex quantum operations into sequences of simpler, universal quantum gates, such as those from the standard set (e.g., $CNOT$, $Hadamard$, and single-qubit rotation gates). This decomposition is necessary because physical quantum computers typically only implement a limited set of basic gates. By decomposing more complex operations into these basic gates, one can execute any quantum algorithm on actual quantum hardware. This process is relevant for the practical implementation of quantum algorithms and for optimizing quantum circuits for individual quantum backends.

Although normally, part of the quantum transpilation process [62], which was not yet implemented within $qib_{BACK}$, quantum gate decomposition was still manually performed in the context of this thesis in order to perform quantum experiments on machines, that wouldn't normally support the gates used in the experiments [1]. This appendix dives deeper into the actual decompositions that have been performed and the logic that stands behind them.

For a list of the available gates and backends configuration of the WMI quantum environment, refer to section 4.2. Based on that, the following gate decompositions have been performed in the experiments conducted in this thesis:

## Basic Experiment: Bell States

For generating the four quantum Bell states, executing the circuits in Figure 7.1 is normally necessary, but since the $CX$ (or $CNOT$) and the $Z$ gates are not configured on the WMI backends, decomposing these two gates to WMI-available gates was necessary (see Figure A.1 for the performed decompositions).

The decompositions provided in Figure A.1 can also be validated mathematically. The $CX$ gate can be decomposed as follows:

$$CX = (H \otimes I) \cdot CZ \cdot (H \otimes I)$$

[1]Note that for some of the performed manual gate decompositions, the output generated by the Qiskit transpiler was still used for reference [28]

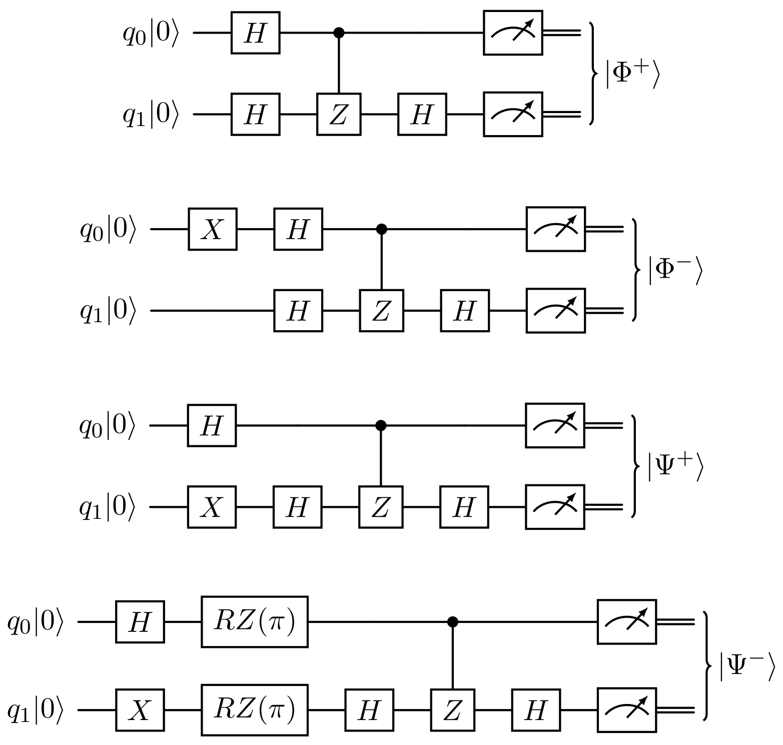


Figure A.1.: The decomposed circuits for the 4 Bell States experiment. For the original circuits see Figure 7.1.

Where:

$$CX = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \quad H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

The tensor product $H \otimes I$ is:

$$H \otimes I = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix}$$

The Controlled-Z gate $CZ$ is:

$$CZ = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

Combining them, one gets:

$$CX = \left( \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \right) \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \cdot \left( \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \right) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

As for the $Z$ gate, it can be simply decomposed as:

$$Z = RZ(\pi)$$

Where:

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \quad RZ(\theta) = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix}$$

For $\theta = \pi$:

$$RZ(\pi) = \begin{pmatrix} e^{-i\pi/2} & 0 \\ 0 & e^{i\pi/2} \end{pmatrix} = \begin{pmatrix} -i & 0 \\ 0 & i \end{pmatrix} = -i * \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Since the global phase has no influence on measurement statistics (which was what was needed in the context of the performed experiments), the assumption that global phase $-i = 1$ has been made. Thus:

$$RZ(\pi) = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = Z$$

## Hybrid Quantum-Classical Algorithm: The U3 Gate

One of the most crucial gates, for performing the QST experiment, was the $U3$ Gate [63], unfortunately, such a generic gate is not supported by the WMI backends, so a decomposition had to be used instead, after multiple trials and errors [2], the decomposition in Figure 7.7 has been chosen.

This gate decomposition can also be mathematically proven, as follows:

$$U3(\theta, \phi, \lambda) = RZ(\phi + \pi) \cdot SX \cdot RZ(\theta + \pi) \cdot SX \cdot RZ(\lambda)$$

Where:

$$U3(\theta, \phi, \lambda) = \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -e^{i\lambda}\sin\left(\frac{\theta}{2}\right) \\ e^{i\phi}\sin\left(\frac{\theta}{2}\right) & e^{i(\phi+\lambda)}\cos\left(\frac{\theta}{2}\right) \end{pmatrix}, \quad RZ(\alpha) = \begin{pmatrix} e^{-i\alpha/2} & 0 \\ 0 & e^{i\alpha/2} \end{pmatrix}, \quad SX = \frac{1}{\sqrt{2}}\begin{pmatrix} 1+i & 1-i \\ 1-i & 1+i \end{pmatrix}$$

Applying the gates step-by-step:

$$RZ(\lambda) = \begin{pmatrix} e^{-i\lambda/2} & 0 \\ 0 & e^{i\lambda/2} \end{pmatrix}$$

$$SX \cdot RZ(\lambda) = \frac{1}{\sqrt{2}}\begin{pmatrix} 1+i & 1-i \\ 1-i & 1+i \end{pmatrix}\begin{pmatrix} e^{-i\lambda/2} & 0 \\ 0 & e^{i\lambda/2} \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} (1+i)e^{-i\lambda/2} & (1-i)e^{i\lambda/2} \\ (1-i)e^{-i\lambda/2} & (1+i)e^{i\lambda/2} \end{pmatrix}$$

$$RZ(\theta + \pi) = \begin{pmatrix} -ie^{-i\theta/2} & 0 \\ 0 & ie^{i\theta/2} \end{pmatrix}$$

$$RZ(\theta + \pi) \cdot (SX \cdot RZ(\lambda)) = \frac{1}{\sqrt{2}}\begin{pmatrix} -ie^{-i\theta/2}(1+i)e^{-i\lambda/2} & -ie^{-i\theta/2}(1-i)e^{i\lambda/2} \\ ie^{i\theta/2}(1-i)e^{-i\lambda/2} & ie^{i\theta/2}(1+i)e^{i\lambda/2} \end{pmatrix}$$

$$SX \cdot (RZ(\theta+\pi) \cdot (SX \cdot RZ(\lambda))) = \frac{1}{\sqrt{2}}\begin{pmatrix} 1+i & 1-i \\ 1-i & 1+i \end{pmatrix} \cdot \frac{1}{\sqrt{2}}\begin{pmatrix} -ie^{-i(\theta/2+\lambda/2)}(1+i) & -ie^{-i(\theta/2-\lambda/2)}(1-i) \\ ie^{i(\theta/2-\lambda/2)}(1-i) & ie^{i(\theta/2+\lambda/2)}(1+i) \end{pmatrix}$$

$$RZ(\phi + \pi) = \begin{pmatrix} -ie^{-i\phi/2} & 0 \\ 0 & ie^{i\phi/2} \end{pmatrix}$$

$$RZ(\phi + \pi) \cdot (SX \cdot (RZ(\theta + \pi) \cdot (SX \cdot RZ(\lambda))))$$

[2]Initially the $U2$ gate was considered for this experiment, but a problem with the built implementation yielded a partial Bloch Sphere coverage, which was insufficient for correctly performing the QST experiment.

$$= RZ(\phi+\pi)\cdot\frac{1}{\sqrt{2}}\begin{pmatrix}1+i & 1-i\\ 1-i & 1+i\end{pmatrix}\cdot\frac{1}{\sqrt{2}}\begin{pmatrix}-ie^{-i(\theta/2+\lambda/2)}(1+i) & -ie^{-i(\theta/2-\lambda/2)}(1-i)\\ ie^{i(\theta/2-\lambda/2)}(1-i) & ie^{i(\theta/2+\lambda/2)}(1+i)\end{pmatrix}$$

Simplifying gives:

$$RZ(\phi+\pi)\cdot SX\cdot RZ(\theta+\pi)\cdot SX\cdot RZ(\lambda) = \begin{pmatrix}\cos\left(\frac{\theta}{2}\right) & -e^{i\lambda}\sin\left(\frac{\theta}{2}\right)\\ e^{i\phi}\sin\left(\frac{\theta}{2}\right) & e^{i(\phi+\lambda)}\cos\left(\frac{\theta}{2}\right)\end{pmatrix} = U3(\theta,\phi,\lambda)$$

## Quantum State Tomography: Post-rotation measurements

Another example of gate decomposition that had to be performed, was the post-rotation gates used for measuring the outcome state of the quantum system, after performing the $U3$ gate rotations, in all 3 Pauli bases (**PauliX, PauliY, PauliZ**). See Figure 7.8 for the actual quantum circuits of these post-rotation measurements. These had to be further decomposed to the circuits in Figure A.2, because the $H$ and $S^\dagger$ gates were not available on the quantum backend.

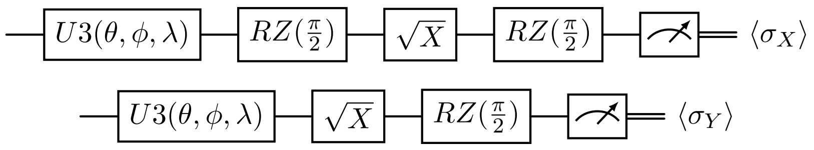


Figure A.2.: Decomposing the post-rotation PauliX and PauliY circuits to WMI-available gates.

The mathematical proof of these decompositions can be performed, as follows:

$$\textbf{PauliX}: H = RZ(\frac{\pi}{2})\cdot\sqrt{X}\cdot RZ(\frac{\pi}{2})$$
$$\textbf{PauliY}: S^\dagger\cdot H = RZ(\frac{\pi}{2})\cdot\sqrt{X}$$

Where:

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix},\quad RZ\left(\frac{\pi}{2}\right) = \begin{pmatrix}\frac{\sqrt{2}}{2}-i\frac{\sqrt{2}}{2} & 0\\ 0 & \frac{\sqrt{2}}{2}+i\frac{\sqrt{2}}{2}\end{pmatrix},\quad \sqrt{X} = \frac{1}{2}\begin{pmatrix}1+i & 1-i\\ 1-i & 1+i\end{pmatrix},\quad S^\dagger = \begin{pmatrix}1 & 0\\ 0 & -i\end{pmatrix}$$

For **PauliX**:

$$RZ\left(\frac{\pi}{2}\right)\cdot\sqrt{X} = \begin{pmatrix}\left(\frac{\sqrt{2}}{2}-i\frac{\sqrt{2}}{2}\right)\frac{1+i}{2} & \left(\frac{\sqrt{2}}{2}-i\frac{\sqrt{2}}{2}\right)\frac{1-i}{2}\\ \left(\frac{\sqrt{2}}{2}+i\frac{\sqrt{2}}{2}\right)\frac{1-i}{2} & \left(\frac{\sqrt{2}}{2}+i\frac{\sqrt{2}}{2}\right)\frac{1+i}{2}\end{pmatrix}$$

Simplifying yields:

$$RZ\left(\frac{\pi}{2}\right)\cdot\sqrt{X} = \frac{1}{\sqrt{2}}\begin{pmatrix} i & 1 \\ 1 & -i \end{pmatrix}$$

By multiplying with another $RZ(\frac{\pi}{2})$ matrix, one gets:

$$\begin{aligned}
&\frac{1}{\sqrt{2}}\begin{pmatrix} i & 1 \\ 1 & -i \end{pmatrix}\cdot\begin{pmatrix} \frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2} & 0 \\ 0 & \frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2} \end{pmatrix} \\
&= \frac{1}{\sqrt{2}}\begin{pmatrix} i\left(\frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2}\right) & 1\left(\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}\right) \\ 1\left(\frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2}\right) & -i\left(\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}\right) \end{pmatrix} \\
&= \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = H
\end{aligned}$$

And for **PauliY**:

$$\begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix}\cdot\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2} & 0 \\ 0 & \frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2} \end{pmatrix}\cdot\frac{1}{2}\begin{pmatrix} 1+i & 1-i \\ 1-i & 1+i \end{pmatrix}$$

$$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ -i & i \end{pmatrix} = \begin{pmatrix} \left(\frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2}\right)\frac{1+i}{2} & \left(\frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2}\right)\frac{1-i}{2} \\ \left(\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}\right)\frac{1-i}{2} & \left(\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}\right)\frac{1+i}{2} \end{pmatrix}$$

$$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ -i & i \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} -i & 1 \\ 1 & i \end{pmatrix}$$

Assuming global phase $-i = 1$ again, this equation (i.e. matrix decomposition) proves to be correct as well.

# B. Parameter-shift Rule

The **Parameter-shift Rule** [42] is a technique used in quantum computing, especially within the context of variational quantum algorithms. It allows for the computation of gradients of quantum functions with respect to their parameters, which is essential for gradient-based optimization methods. This appendix provides a detailed explanation of the parameter shift rule and its application in parameterized quantum gates.

## Parameterized Quantum Circuits

A parameterized quantum circuit consists of a sequence of quantum gates, each of which can depend on one or more parameters. The unitary operation implemented by such a circuit can be expressed as a product of unitary operators:

$$U(\theta) = U_N(\theta_N)U_{N-1}(\theta_{N-1}) \cdots U_1(\theta_1) \tag{B.1}$$

where each $U_i(\theta_i)$ is a unitary operator parameterized by $\theta_i$.

## Parameter Shift Rule

The parameter shift rule provides a way to compute the derivative of an expectation value of an observable with respect to a gate parameter. Given the parametrized quantum circuit scenario formulated in Equation B.1, by isolating a single parameter $\theta_i$ and its associated gate $U_i(\theta_i)$, one gets the following quantum circuit function [1]:

$$f(\theta_i) = \langle 0|U_i^\dagger(\theta_i) \cdot B \cdot U_i(\theta_i)|0\rangle \tag{B.2}$$

Where $B$ is a quantum operator. For simplicity reasons, the unitary conjugation can be rewritten as:

$$U_i^\dagger(\theta_i) \cdot B \cdot U_i(\theta_i) = L_{\theta_i}(B) \tag{B.3}$$

Where $L_{\theta_i}(B)$ is a linear transformation applied on operator $B$ with respect to parameter $\theta_i$. Given this new, rewritten equation, the gradient of the function can be defined as:

[1]The zeros here indicate that the quantum circuit starts in the ground state, which was a common assumption in the context of this paper if not explicitly stated otherwise.

$$\nabla f(\theta_i) = \langle 0 | \nabla L_{\theta_i}(B) | 0 \rangle \in \mathbb{R} \tag{B.4}$$

As has been proven in [56], in the special case of a gate $U$ that can be generated by a Hermitian operator with 2 unique eigenvalues[2], this gradient can be formulated as a linear combination of the transformation $L$ using different sets of parameters. Such that:

$$\nabla L_{\theta_i}(B) = c[L_{\theta_i+s}(B) - L_{\theta_i-s}(B)] \tag{B.5}$$

Thus, for a quantum function $f(\theta)$ representing the expectation value, the partial derivative with respect to a parameter $\theta$ can be expressed as:

$$\frac{\partial f(\theta)}{\partial \theta} = c\left[f(\theta + s) - f(\theta - s)\right], \tag{B.6}$$

where $c$ and $s$ are constants dependent on the specific gate used (independent of $\theta$). Typically, $c = 1/2$ and $s = \pi/2$ for common quantum gates [38].

[2]In the context of this thesis, all PSR applications involved such gates (e.g. the $RZ(\theta)$ gate)

# Bibliography

[1] Dorit Aharonov et al. *Adiabatic Quantum Computation Is Equivalent to Standard Quantum Computation*. Mar. 2005. DOI: 10.48550/arXiv.quant-ph/0405098. arXiv: quant-ph/0405098. (Visited on 02/06/2024).

[2] Muhammad Azeem Akbar, Arif Ali Khan, and Saima Rafi. "A Systematic Decision-Making Framework for Tackling Quantum Software Engineering Challenges". In: *Automated Software Engineering* 30.2 (July 2023), p. 22. ISSN: 1573-7535. DOI: 10.1007/s10515-023-00389-7. (Visited on 02/06/2024).

[3] *ARTIQ (M-Labs)*. URL: https://m-labs.hk/ (visited on 02/06/2024).

[4] Steven Balensiefer, Lucas Kreger-Stickles, and Mark Oskin. "QUALE: Quantum Architecture Layout Evaluator". In: *Quantum Information and Computation III*. Vol. 5815. SPIE, May 2005, pp. 103–114. DOI: 10.1117/12.604073. (Visited on 02/06/2024).

[5] Harrison Ball et al. *Software Tools for Quantum Control: Improving Quantum Computer Performance through Noise and Error Suppression*. July 2020. DOI: 10.48550/arXiv.2001.04060. arXiv: 2001.04060 [quant-ph]. (Visited on 02/06/2024).

[6] Saptashwa Bhattacharyya. *Quantum Computing: Bell State and Entanglement with Qiskit*. Nov. 2022. (Visited on 02/06/2024).

[7] S. Bravyi et al. "The Future of Quantum Computing with Superconducting Qubits". In: *Journal of Applied Physics* (2022). DOI: 10.1063/5.0082975. (Visited on 02/06/2024).

[8] H. Briegel et al. "Measurement-Based Quantum Computation". In: *Nat. Phys.* 5 (Oct. 2009), pp. 19–26. DOI: 10.1038/nphys1157.

[9] Lauren Capelluto and Thomas Alexander. "OpenPulse: Software for Experimental Physicists in Quantum Computing". In: *American Physical Society (March Meeting)*. Mar. 2020. (Visited on 02/06/2024).

[10] *Cirq*. URL: https://quantumai.google/cirq (visited on 02/06/2024).

[11] David G. Cory, Mark D. Price, and Timothy F. Havel. "Nuclear Magnetic Resonance Spectroscopy: An Experimentally Accessible Paradigm for Quantum Computing". In: *Physica D: Nonlinear Phenomena*. Proceedings of the Fourth Workshop on Physics and Consumption 120.1 (Sept. 1998), pp. 82–101. ISSN: 0167-2789. DOI: 10.1016/S0167-2789(98)00046-3. (Visited on 04/07/2024).

[12] Andrew W. Cross et al. *Open Quantum Assembly Language*. July 2017. arXiv: 1707.03429 [quant-ph]. (Visited on 02/06/2024).

[13] Andrew W. Cross et al. "OpenQASM 3: A Broader and Deeper Quantum Assembly Language". In: *ACM Transactions on Quantum Computing* 3.3 (Sept. 2022), pp. 1–50. ISSN: 2643-6809, 2643-6817. DOI: 10.1145/3505636. arXiv: 2104.14722 [quant-ph]. (Visited on 02/06/2024).

[14] dev. *OpenQKD*. URL: https://openqkd.eu/ (visited on 02/06/2024).

[15] E. Diamanti et al. "Practical Challenges in Quantum Key Distribution". In: *npj Quantum Information* 2 (2016). DOI: 10.1038/npjqi.2016.25. (Visited on 02/06/2024).

[16] David P. DiVincenzo and IBM. "The Physical Implementation of Quantum Computation". In: *Fortschritte der Physik* 48.9-11 (Sept. 2000), pp. 771–783. ISSN: 00158208, 15213978. DOI: 10.1002/1521-3978(200009)48:9/11<771::AID-PROP771>3.0.CO;2-E. arXiv: quant-ph/0002077. (Visited on 02/09/2024).

[17] Mohammad Javad Dousti, Alireza Shafaei, and Massoud Pedram. *Squash 2: A Hierarchical Scalable Quantum Mapper Considering Ancilla Sharing*. Dec. 2015. DOI: 10.48550/arXiv.1512.07402. arXiv: 1512.07402 [quant-ph]. (Visited on 02/06/2024).

[18] Jose Garcia-Alonso et al. "Quantum Software as a Service Through a Quantum API Gateway". In: *IEEE Internet Computing* 26.1 (Jan. 2022), pp. 34–41. ISSN: 1941-0131. DOI: 10.1109/MIC.2021.3132688. (Visited on 02/06/2024).

[19] *Google Quantum AI*. URL: https://quantumai.google/ (visited on 02/06/2024).

[20] Thomas Häner et al. "A Software Methodology for Compiling Quantum Programs". In: *Quantum Science and Technology* 3.2 (Apr. 2018), p. 020501. ISSN: 2058-9565. DOI: 10.1088/2058-9565/aaa5cc. arXiv: 1604.01401 [quant-ph]. (Visited on 02/06/2024).

[21] Loïc Henriet et al. "Quantum Computing with Neutral Atoms". In: *Quantum* 4 (Sept. 2020), p. 327. DOI: 10.22331/q-2020-09-21-327. (Visited on 04/07/2024).

[22] *IBM Quantum Computing*. URL: https://www.ibm.com/www.ibm.com/quantum (visited on 02/06/2024).

[23] *IBM Quantum Computing — Qiskit*. URL: https://www.ibm.com/quantum/www.ibm.com/quantum/qiskit (visited on 02/06/2024).

[24] *IBM Quantum Documentation — Qiskit*. URL: https://docs.quantum.ibm.com/ (visited on 02/10/2024).

[25] *IBM Quantum Learning — Qiskit*. URL: https://quantum.ibm.com/composer (visited on 02/10/2024).

[26] *Introduction — OpenQASM 3.0 Specification Documentation*. URL: https://openqasm.com/versions/3.0/intro.html (visited on 02/06/2024).

[27] *Introduction — OpenQASM Live Specification Documentation*. URL: https://openqasm.com/intro.html (visited on 02/06/2024).

[28] Ali Javadi-Abhari et al. *Quantum computing with Qiskit*. 2024. DOI: 10.48550/arXiv.2405.08810. arXiv: 2405.08810 [quant-ph].

[29] Yanjun Ji, Sebastian Brandhofer, and I. Polian. "Calibration-Aware Transpilation for Variational Quantum Optimization". In: *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)* (2022), pp. 204–214. DOI: 10.1109/QCE53715.2022.00040. (Visited on 02/10/2024).

[30] Peter J. Karalekas et al. "A Quantum-Classical Cloud Platform Optimized for Variational Hybrid Algorithms". In: *Quantum Science and Technology* 5 (2020). DOI: 10.1088/2058-9565/ab7559. (Visited on 02/06/2024).

[31] Changsoon Kim et al. "Integrated Optics Technology for Quantum Information Processing in Atomic Systems". In: *Conference on Lasers and Electro-Optics/Quantum Electronics and Laser Science Conference and Photonic Applications Systems Technologies (2007), Paper JTuA24*. Optica Publishing Group, May 2007, JTuA24. (Visited on 04/07/2024).

[32] Philip Krantz et al. "A Quantum Engineer's Guide to Superconducting Qubits". In: *Applied Physics Reviews* 6.2 (June 2019), p. 021318. ISSN: 1931-9401. DOI: 10.1063/1.5089550. arXiv: 1904.06560 [cond-mat, physics:physics, physics:quant-ph]. (Visited on 02/09/2024).

[33] Ryan LaRose. "Overview and Comparison of Gate Level Quantum Software Platforms". In: *Quantum* 3 (Mar. 2019), p. 130. ISSN: 2521-327X. DOI: 10.22331/q-2019-03-25-130. arXiv: 1807.02500 [quant-ph]. (Visited on 02/06/2024).

[34] D. Maslov. "Basic Circuit Compilation Techniques for an Ion-Trap Quantum Machine". In: *New Journal of Physics* 19 (2016). DOI: 10.1088/1367-2630/aa5e47. (Visited on 02/10/2024).

[35] A. McCaskey et al. "A Language and Hardware Independent Approach to Quantum-Classical Computing". In: *SoftwareX* 7 (2017), pp. 245–254. DOI: 10.1016/J.SOFTX.2018.07.007. (Visited on 02/06/2024).

[36] A. McCaskey et al. "XACC: A System-Level Software Infrastructure for Heterogeneous Quantum–Classical Computing". In: *Quantum Science and Technology* 5 (2019). DOI: 10.1088/2058-9565/ab6bf6. (Visited on 02/06/2024).

[37] David C. McKay et al. *Qiskit Backend Specifications for OpenQASM and OpenPulse Experiments*. Sept. 2018. arXiv: 1809.03452 [quant-ph]. (Visited on 02/06/2024).

[38] Kosuke Mitarai et al. "Quantum Circuit Learning". In: *Physical Review A* 98.3 (Sept. 2018), p. 032309. ISSN: 2469-9926, 2469-9934. DOI: 10.1103/PhysRevA.98.032309. arXiv: 1803.00745 [quant-ph]. (Visited on 06/25/2024).

[39] Thien Nguyen and A. McCaskey. "Enabling Pulse-Level Programming, Compilation, and Execution in XACC". In: *IEEE Transactions on Computers* 71 (2020), pp. 547–558. DOI: 10.1109/TC.2021.3057166. (Visited on 02/06/2024).

[40] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, Dec. 2010. ISBN: 9780511976667. DOI: 10.1017/CBO9780511976667. (Visited on 02/06/2024).

[41] *Ocean™ Developer Tools — D-Wave*. URL: https://www.dwavesys.com/solutions-and-products/ocean/ (visited on 02/06/2024).

[42] *Parameter-Shift Rules — PennyLane*. URL: https://pennylane.ai/qml/glossary/parameter%5C_shift/ (visited on 02/06/2024).

[43] *PennyLane*. URL: https://pennylane.ai/ (visited on 02/06/2024).

[44] Michael Perelshtein et al. *Practical Application-Specific Advantage through Hybrid Quantum Computing*. May 2022. DOI: 10.48550/arXiv.2205.04858. arXiv: 2205.04858 [quant-ph]. (Visited on 02/06/2024).

[45] Mario Piattini et al. "The Talavera Manifesto for Quantum Software Engineering and Programming". In: ().

[46] *Qc-Tum/Qib: Python Package for Quantum Circuits and Algorithms, Focusing on Quantum Simulation*. URL: https://github.com/qc-tum/qib (visited on 05/04/2024).

[47] *Qib Documentation (Qib 0.1.0)*. URL: https://qib.readthedocs.io/en/latest/ (visited on 05/04/2024).

[48] Qiskit. *How Does The Qiskit Transpiler Work?* July 2021. (Visited on 02/10/2024).

[49] *Qiskit's Circuit Library*. en. URL: https://docs.quantum.ibm.com/api/qiskit/circuit_library (visited on 07/03/2024).

[50] *Qobj*. URL: https://docs.quantum.ibm.com/api/qiskit/qobj (visited on 02/06/2024).

[51] *Quanten-Cloud-Computing-Service – Amazon Braket – AWS*. URL: https://aws.amazon.com/de/braket/ (visited on 02/06/2024).

[52] Leonard Richardson, Mike Amundsen, and Sam Ruby. *RESTful Web APIs: Services for a Changing World*. 1st edition. Beijing Cambridge Farnham Köln Sebastopol Tokyo: O'Reilly and Associates, Oct. 2013. ISBN: 978-1-4493-5806-8.

[53] S. Ritter et al. "An Elementary Quantum Network of Single Atoms in Optical Cavities". In: *Nature* 484 (2012), pp. 195–200. DOI: 10.1038/nature11023. (Visited on 02/10/2024).

[54] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2017. arXiv: 1609.04747.

[55] Mehdi Saeedi, Robert Wille, and Rolf Drechsler. "Synthesis of Quantum Circuits for Linear Nearest Neighbor Architectures". In: *Quantum Information Processing* 10.3 (June 2011), pp. 355–377. ISSN: 1570-0755, 1573-1332. DOI: 10.1007/s11128-010-0201-2. arXiv: 1110.6412 [quant-ph]. (Visited on 02/06/2024).

[56] Maria Schuld et al. "Evaluating Analytic Gradients on Quantum Hardware". In: *Physical Review A* 99.3 (Mar. 2019), p. 032331. ISSN: 2469-9926, 2469-9934. DOI: 10.1103/PhysRevA.99.032331. arXiv: 1811.11184 [quant-ph]. (Visited on 02/06/2024).

[57] Manuel A. Serrano et al. "Quantum Software Components and Platforms: Overview and Quality Assessment". In: *ACM Computing Surveys* 55.8 (Aug. 2023), pp. 1–31. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3548679. (Visited on 02/06/2024).

[58] Brian N. Siegelwax. *What Is "Transpilation?"* Dec. 2021. URL: https://levelup.gitconnected.com/what-is-transpilation-4d12d51e2aa4 (visited on 02/10/2024).

[59] SoniaLopezBravo. *Introduction to Q# & Quantum Development Kit - Azure Quantum*. Jan. 2024. URL: https://learn.microsoft.com/en-us/azure/quantum/overview-what-is-qsharp-and-qdk (visited on 02/06/2024).

[60] K.M. Svore et al. "A Layered Software Architecture for Quantum Computing Design Tools". In: *Computer* 39.1 (Jan. 2006), pp. 74–83. ISSN: 1558-0814. DOI: 10.1109/MC.2006.4. (Visited on 02/06/2024).

[61] *TensorFlow*. URL: https://www.tensorflow.org/ (visited on 02/06/2024).

[62] *transpiler*. en. URL: https://docs.quantum.ibm.com/api/qiskit/transpiler (visited on 06/16/2024).

[63] *U3Gate*. en. URL: https://docs.quantum.ibm.com/api/qiskit/qiskit.circuit.library.U3Gate (visited on 06/18/2024).

[64] *Walther-Meißner-Institut*. URL: https://www.wmi.badw.de/home (visited on 05/09/2024).

[65] *What Is Quantum Key Distribution (QKD) and How Does It Work?* URL: https://www.techtarget.com/searchsecurity/definition/quantum-key-distribution-QKD (visited on 02/06/2024).

[66] Karoline Wild et al. "TOSCA4QC: Two Modeling Styles for TOSCA to Automate the Deployment and Orchestration of Quantum Applications". In: *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*. Eindhoven, Netherlands: IEEE, Oct. 2020, pp. 125–134. ISBN: 978-1-72816-473-1. DOI: 10.1109/EDOC49727.2020.00024. (Visited on 02/06/2024).

[67] Leo Zhou et al. "Quantum Approximate Optimization Algorithm: Performance, Mechanism, and Implementation on Near-Term Devices". In: *Physical Review X* 10.2 (June 2020), p. 021067. ISSN: 2160-3308. DOI: 10.1103/PhysRevX.10.021067. arXiv: 1812.01041 [cond-mat, physics:quant-ph]. (Visited on 02/06/2024).