Full Quantum Stack: Ket Platform

Evandro Chagas Ribeiro da Rosa[1], Eduardo Willwock Lussi[1], Jerusa Marchi[1], Rafael de Santiago[1], Eduardo Inacio Duzzioni[2]

[1]Departamento de Informática e Estatística, Universidade Federal de Santa Catarina, Florianópolis, Santa Catarina, Brazil.
[2]Departamento de Física, Universidade Federal de Santa Catarina, Florianópolis, Santa Catarina, Brazil.

Abstract

As quantum computing hardware continues to scale, the need for a robust software infrastructure that bridges the gap between high-level algorithm development and low-level physical qubit control becomes increasingly critical. A full-stack approach, analogous to classical computing, is essential for managing complexity, enabling hardware-agnostic programming, and systematically optimizing performance. In this paper, we present a comprehensive, end-to-end quantum software stack, detailing each layer of abstraction from user-facing code to hardware execution. We begin at the highest level with the Ket quantum programming platform, which provides an expressive, Python-based interface for algorithm development. We then describe the crucial multi-stage compilation process, which translates hardware-agnostic programs into hardware-compliant circuits by handling gate decomposition, qubit mapping to respect device connectivity, and native gate translation. To illustrate the complete workflow, we present a concrete example, compiling the Grover diffusion operator for a superconducting quantum processor. Finally, we connect the compiled circuit to its physical realization by explaining how native gates are implemented through calibrated microwave pulses. This includes the calibration of single- and two-qubit gates, frequency characterization, and measurement procedures, providing a clear picture of how abstract quantum programs ultimately map onto the physical control of a quantum processor. By providing a detailed blueprint of a complete quantum stack, this work illuminates the critical interplay between software abstractions and physical hardware, establishing a framework for developing practical and performant quantum applications.

Keywords: Quantum Software Stack, Quantum Compilation, High-Level Quantum Programming, Quantum Control, Quantum Hardware, Ket Platform

Contents

1 Introduction **2**

2 Quantum Programming Guide **4**
2.1 Classical-Quantum Model . . . 5
2.2 Quantum Process and Qubit Allocation . . . 6
2.3 Quantum Gates . . . 6
2.4 Extracting Information via Measurement . . . 9
2.5 KBW Simulator Configuration . . . 12

3 Quantum Compilation **12**
3.1 Quantum Gate Decomposition . . . 14
3.2 Quantum Circuit Mapping . . . 16
3.3 Native Gate Translation . . . 17
3.4 Expectation Value Engine . . . 18
3.5 Circuit Optimization . . . 20

4 Physical Realization of Quantum Programs **20**
4.1 Control Pulse Theory . . . 21
4.1.1 Control Hamiltonian . . . 21
4.1.2 Waveform Sampling . . . 22
4.1.3 Pulse Shaping . . . 23
4.2 Single-Qubit Gates Calibration . . . 26
4.2.1 Rabi Experiment . . . 27
4.2.2 Fine-Tuning . . . 28
4.2.3 Virtual Z Gates . . . 30
4.3 Finding the Qubit Frequency . . . 30
4.4 Multi-Qubit Gates Calibration . . . 32
4.4.1 iSWAP . . . 33
4.4.2 CPHASE . . . 35
4.4.3 Cross-Resonance . . . 36
4.5 Readout Calibration . . . 39

5 Final Remarks **40**

1 Introduction

Quantum computers have the potential to solve problems that are intractable for their classical counterparts [1]. While current quantum hardware has demonstrated a quantum advantage [2–8], the development of large-scale, practical quantum computers remains an open challenge [9, 10]. Achieving this objective requires significant improvements in quantum hardware engineering. However, a frequently overlooked aspect is the software infrastructure required to manage this hardware.

Although drag-and-drop interfaces for quantum gates provide an introductory approach to quantum computing, coding is essential for developing practical quantum

software, even for today's Noisy Intermediate-Scale Quantum (NISQ) computers [11]. Many quantum programming platforms [12–18], though sometimes accessible via high-level languages, primarily offer low-level quantum constructs, which are currently necessary to obtain meaningful results from noisy quantum devices. However, as the number of qubits increases, the time required for low-level programming and qubit-specific tuning becomes impractical; also, the resulting code often lacks hardware independence. Consequently, there is a strong motivation to adopt a higher-level programming model, where automated compiler optimizations replace manual, hardware-specific tuning.

The transition from low-level to high-level coding is not novel; a similar evolution occurred in classical computing. Although the C language was invented in 1972 [19] and standardized by the American National Standards Institute (ANSI) in 1989, assembly coding[1] was often used for performance-critical applications until the late 1990s. With advancements in compilers and the increasing complexity of processors, writing assembly code that could outperform compiler-generated code became increasingly challenging, as compilers could explore a wider range of optimizations than the average programmer. Consequently, high-level programming enhances both programmer productivity and execution performance. Just as classical computing performance has advanced through both hardware and software improvements, a similar trajectory is expected for quantum computing.

In this paper, we review the software infrastructure required to execute a quantum program written in Python on quantum hardware, presenting the components of a complete *quantum stack*. A software stack is a computer science concept that divides a solution into abstraction layers, starting with the problem domain at the highest level and becoming more specialized at lower levels. Our proposed quantum stack begins at a high level with Python code using the Ket quantum programming platform [20] and extends down to the microwave pulses that manipulate the physical qubits.

The Ket platform is an open-source project composed of three main components. The first is a Python programming interface, which serves as the high-level front-end for quantum software development. The second is Libket, the platform's runtime library, which is responsible for quantum compilation; this runtime is written in Rust, exposes a C API, and can be used independently of the Python interface. The third component is the Ket Bitwise Simulator (KBW), a noiseless simulator that enables the execution of quantum applications on ordinary hardware.

This review addresses the stack layers currently implemented in Ket, from the programming interface down to the low-level quantum circuit. The broader objective, however, is to expand this stack in both directions: upwards with domain-specific libraries for fields such as quantum chemistry [21] and finance [22], and downwards to include pulse-level programming [23] and direct Arbitrary Waveform Generator (AWG) management [24].

Structuring the software infrastructure into abstraction layers helps manage complexity, allowing developers to focus on one layer at a time. Furthermore, this layered approach facilitates the integration of new solutions and optimizations. This is because

[1] The human-readable language closest to machine code.

modifications typically require a new component to interact only with the immediately adjacent layers, rather than demanding a rewrite of the entire software stack, an aspect discussed in subsequent sections.

The remainder of this paper is structured as follows. Section 2 provides an overview of the Ket quantum programming platform, which enables the development of quantum software using Python. Within the quantum stack, Ket represents the highest abstraction layer, which is completely agnostic to the target quantum hardware and is used by programmers to develop domain-specific solutions. Section 3 describes how quantum code written in Ket is compiled for a target quantum computer, ensuring the final circuit adheres to the hardware's connectivity constraints and native gate set. The compilation process concludes by scheduling the pulses that physically manipulate the qubits, a step that depends on calibration parameters measured from the quantum device. Section 4 explores the methods and hardware aspects involved in calibrating a quantum computer to enable the execution of quantum software. We conclude in Section 5 with our final remarks.

2 Quantum Programming Guide

Ket is an open-source quantum programming platform designed to bring the expressivity of Python to quantum programming. Its core, *Libket*, is a high-performance runtime library written in Rust, a memory safety language known for its speed. This two-language architecture provides Python's ease of use for development and Rust's performance for computationally intensive tasks. The platform also includes a built-in quantum simulator, named *KBW*, which allows users to simulate small-scale quantum programs on a personal computer.

This section serves as a programming guide with Ket, demonstrating how to construct quantum programs in Python. It assumes the reader is familiar with both Python and the fundamentals of quantum computing, as our focus is on Ket's specific functions and programming model. We will cover the entire workflow, from setting up the execution environment and allocating qubits to applying quantum gates and extracting classical results through measurement.

For those wishing to execute the code examples, Ket can be installed from the Python Package Index (PyPI). It is compatible with Python 3.10 or newer and is available for Linux, Windows, and macOS (Intel and Apple silicon). Ket can be installed via pip:

```
pip install ket-lang==0.9.1
```

For users on non-x86_64 processors (such as ARM) under Linux or Windows, the Rust compiler is required to build Ket from source. The examples in this guide use Ket version 0.9, which was the latest version at the time of writing. For complete documentation, please visit https://quantumket.org.

2.1 Classical-Quantum Model

Quantum software solutions are inherently hybrid programs, where the quantum computer acts as a specialized accelerator, much like a Graphics Processing Unit (GPU) or a Field Programmable Gate Array (FPGA). As illustrated in Figure 1, when a quantum program is written in Ket, a clear division of labor occurs: classical instructions are treated as standard Python code executed primarily by the CPU, while quantum operations are passed to Libket, which is responsible for managing the quantum execution.

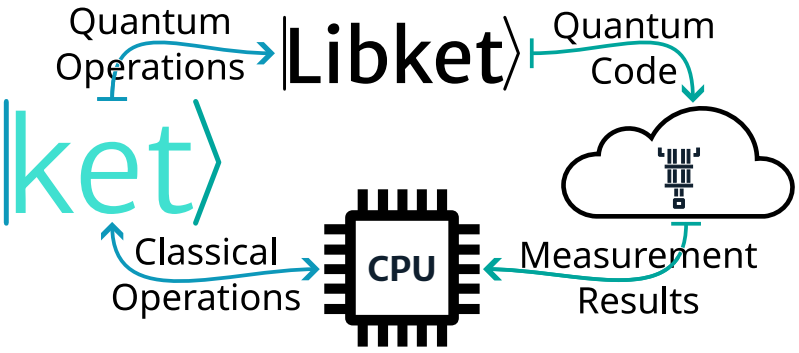


Fig. 1: The hybrid computational model in Ket. The programmer defines operations in Python. Classical logic is executed by the CPU, while quantum instructions are collected by the Libket runtime into a *quantum code*. This code is sent to a backend (a simulator or QPU), which returns classical measurement results. These results can then be used to guide the subsequent steps of the algorithm, forming a complete classical-quantum feedback loop.

Libket gathers these operations into a *quantum code*–an internal representation of a quantum circuit–and dispatches it to the target backend, which can be a simulator or a quantum computer. Once the execution is complete, the backend returns classical measurement results to the Python program. This allows the classical code to process the quantum output and determine the final answer.

We highlight the classical computer's role in interpreting the output of a quantum computation. A quantum device typically produces raw classical data, which must be post-processed to extract a final solution. Shor's algorithm serves as a prime example. After the quantum processor generates a bitstring related to a problem's period, the classical computer computes the continued fraction algorithm to determine the period.

To manage the classical-quantum interaction, Ket provides two execution modes that differ in how the two processors interact. In *batch execution* mode, all quantum operations are collected by Libket into a complete program before being sent to the target. This program is dispatched only when a result is requested. On the backend, batch execution allows requests to be managed by a queue, which reduces the quantum computer's idle time and optimizes resource usage. This model is also more efficient for NISQ devices as it is not impacted by classical-quantum communication latency.

In *live execution* mode, each quantum instruction is sent immediately to the execution target. Measurement results are available instantly, allowing classical logic to influence subsequent quantum operations. This dynamic control is essential for implementing protocols like quantum teleportation and error correction schemes [25, 26]. However, this interaction introduces a latency that is often impractical for current NISQ hardware. Consequently, this mode is primarily used with local simulators and is helpful for the study and debugging of quantum algorithms.

2.2 Quantum Process and Qubit Allocation

The entry point for any quantum program in Ket is the `Process` class, which manages the quantum execution context. It encapsulates all necessary information for execution, including the number of available qubits, the native gate set of the target device, qubit connectivity, and measurement capabilities.

Once a `Process` instance is created, its primary use is to allocate qubits via the `.alloc()` method. This method takes an optional integer to specify the number of qubits to allocate; if omitted, it allocates a single qubit. The method returns a *Quant* object, which is an opaque data type that acts as a list of qubit references. Newly allocated qubits are always initialized in the $|0\rangle$ state.

The default `Process()` constructor initializes the KBW simulator with 32 qubits available in sparse mode (see Section 2.5 for details). This provides a convenient starting point for developing and testing quantum programs. Figure 2 shows how to instantiate a `Process` and allocate 30 qubits to prepare a GHZ state. Multiple calls to `.alloc()` are allowed as long as qubits remain available.

```
process = Process()
qubits = process.alloc(30)
H(qubits[0])
ctrl(qubits[0], X)(qubits[1:])
dump(qubits).show()
```

Fig. 2: Example of qubit allocation using a `Process` instance and the subsequent preparation of a 30-qubit GHZ state.

The `Quant` object holds references to *logical qubits*, which are indexed from 0 to $N-1$ where N is the total number of allocated qubits. These logical indices do not necessarily correspond to the physical qubit labels on a Quantum Processing Unit (QPU). The compiler may map a single logical qubit to different physical qubits during execution to satisfy the hardware's connectivity constraints.

2.3 Quantum Gates

In Ket, quantum gates are represented as Python functions that take qubit references as arguments. Ket is flexible: any Python callable can act as a quantum gate, so long as it does not measure or allocate qubits[2]. This design simplifies the creation of

[2]This excludes the concept of ancillary qubits, which is not addressed in this paper.

new quantum operations and allows for the straightforward application of inverse and controlled modifiers to any gate-like function.

All quantum gates are ultimately constructed from a set of fundamental single-qubit gates provided by Libket [27], as detailed in Table 1. Multi-qubit gates are typically built by applying these fundamental gates in a controlled manner. To visualize a circuit without executing it, Ket provides the `qulib.draw` function, as shown in Figure 3.

Qubits in Ket are managed through the `Quant` data type. As a list-like object, `Quant` supports indexing, slicing, and concatenation, with the result of these operations also being a `Quant` instance. As demonstrated in Figure 3a, applying a single-qubit gate to a multi-qubit `Quant` object applies the gate to each qubit individually.

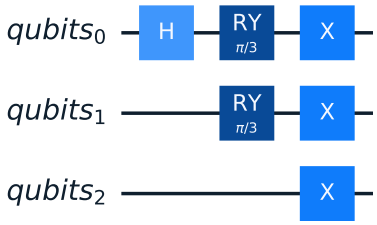
Table 1: Single-qubit gates provided by Libket.

Quantum Gate	Python Function	Gate Matrix
Pauli-X	`X(qubit)`	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli-Y	`Y(qubit)`	$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Pauli-Z	`Z(qubit)`	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard	`H(qubit)`	$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
X-Rotation	`RX(angle, qubit)`	$\begin{bmatrix} \cos(\frac{\theta}{2}) & -i\sin(\frac{\theta}{2}) \\ -i\sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{bmatrix}$
Y-Rotation	`RY(angle, qubit)`	$\begin{bmatrix} \cos(\frac{\theta}{2}) & -\sin(\frac{\theta}{2}) \\ \sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{bmatrix}$
Z-Rotation	`RZ(angle, qubit)`	$\begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix}$
Phase	`P(angle, qubit)`	$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix}$

Sequential and Composed Gates

Quantum gates are applied in the order they appear in the Python code. Ket also provides tools to compose gates into more complex operations. Figure 4 shows two ways to prepare a Bell state. In Ket, gate functions return the qubits they act upon, enabling a fluent interface where gates can be chained in a single line. Alternatively, the `kron` (tensor product) and `cat` (concatenation) functions can be used to build composite gates. Gate concatenation with `cat` is analogous to matrix multiplication, but the gates are listed in the order of their application in the circuit.

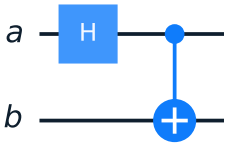
```python
def example(qubits: Quant):
    H(qubits[0])
    RY(pi / 3, qubits[:2])
    X(qubits)

# Args: function, num_qubits
qulib.draw(example, 3)
```



(a) Python code defining a quantum circuit. (b) Resulting quantum circuit diagram.

Fig. 3: Visualization of a quantum circuit defined in Ket using `qulib.draw`.

```python
# Method 1: Sequential application
def bell_seq(a: Quant, b: Quant):
    # More explicit qubit args
    CNOT(H(a), b)
# Method 2: Composition
HI = kron(H, I)
BELL = cat(HI, CNOT)
```



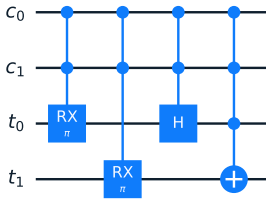
(a) Code for a Bell state preparation. (b) Circuit for a Bell state preparation.

Fig. 4: Implementation and visualization of a Bell state preparation using sequential application and gate composition.

Controlled Operations

The CNOT gate is built using Ket's `ctrl` function, a higher-order function that adds control qubits to any existing gate, including user-defined ones. As shown in Figure 5, when a user-defined function is controlled, the control logic is distributed to each underlying base gate within it.

```python
def example_cgate(c: Quant, t: Quant):
    # Controlled single gate
    ctrl(c, RX(pi / 2))(t)

    # Controlled user-defined gate
    ctrl(c, bell_seq)(t[0], t[1])
```



(a) Defining controlled operations. (b) Circuit of controlled operations.

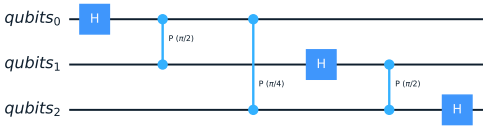
Fig. 5: Constructing controlled gates in Ket using the `ctrl` function.

Inverse Operations

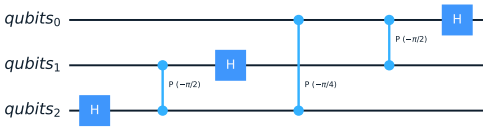
Every unitary gate U has an inverse, its adjoint $U^\dagger$. Ket provides the `adj` function to compute the inverse of any gate automatically. Figure 6 shows a recursive implementation of the Quantum Fourier Transform (QFT). Its inverse (`iqft`) is simply created by applying `adj` to the original `qft` function. The `adj` function reverses the sequence of operations and takes the adjoint of each individual gate.

```
def qft(qubits: Quant):
    if len(qubits) == 1:
        H(qubits[0])
        return
    head, *tail = qubits
    H(head)
    for i, c in enumerate(tail):
        angle = 2*pi/2**(i + 2)
        ctrl(c, P(angle))(head)
    qft(tail)
iqft = adj(qft)
```

(a) QFT implementation and its inverse.



(b) QFT circuit diagram.



(c) Inverse QFT (IQFT) circuit diagram.

Fig. 6: Implementation of QFT and its inverse using the `adj` function in Ket.

Conjugation by a Unitary

A common pattern in quantum algorithms is the application of a gate V conjugated by a unitary U, forming the structure $U^\dagger VU$. This is analogous to a change of basis. Ket provides the `with around` construct for this pattern, where the programmer defines U and V, and the inverse $U^\dagger$ is applied automatically.

Figure 7 shows two ways to implement an R_{XX} gate. The `with around` version is more concise and easier to maintain, as changes to U are automatically reflected in both its forward and inverse applications. This construct can also enable compiler optimizations, especially for controlled operations [27]. As shown in Figure 8, applying a control to the `with around` version of R_{XX} can result in a much more efficient circuit upon decomposition compared to controlling the explicit version, reducing the complexity and gate count required for hardware execution [28].

2.4 Extracting Information via Measurement

Measuring qubits is the fundamental method for extracting classical information from a quantum computation. Ket provides several functions for this purpose, whose availability can depend on the execution target.

Single-Shot Measurement

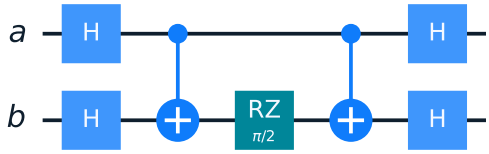
The `measure(qubits)` function performs a measurement in the computational basis, collapsing the quantum state, as illustrated in Figure 9. The result is an object whose

```
# Explicit implementation
def rxx_expl(angle, q0 , q1):
    H(q0)
    H(q1)
    CNOT(q0, q1)
    RZ(angle, q1)
    CNOT(q0, q1)
    H(q0)
    H(q1)
```

(a) Explicit R_{XX} gate implementation.

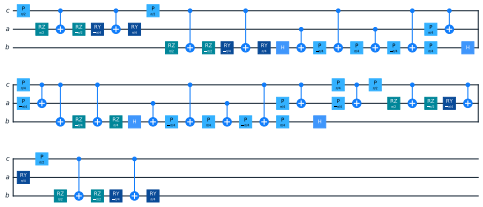
```
# Using 'with around'
def rxx_around(angle, q0, q1):
    H_both = kron(H, H)
    U = cat(H_both, CNOT)
    with around(U, q0, q1): # U
        RZ(angle, q1)       # V
```

(b) R_{XX} gate using `with around`.

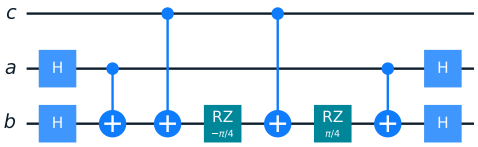


(c) Resulting R_{XX} circuit diagram (identical for both methods).

Fig. 7: R_{XX} gate implemented explicitly and using Ket's `with around` construct.



(a) Controlled R_{XX} from Figure 7a.



(b) Controlled R_{XX} from Figure 7b.

Fig. 8: Comparison of controlled R_{XX} implementations. The `with around` construct can lead to more optimized controlled operations, especially upon decomposition.

`.get()` method returns an unsigned integer representing the measured bit string. The ability to measure the same qubit multiple times depends on the backend; it is allowed in the KBW simulator but may not be on hardware.

Statistical Sampling

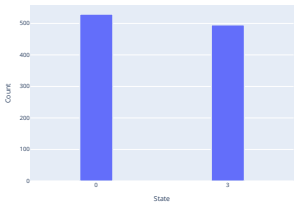
The `sample(qubits, shots)` function measures the qubits `shots` times and count the results. Its `.get()` method returns a dictionary mapping measured states to their counts, while `.histogram()` plots the results, as shown in Figure 10. In the KBW simulator, sampling does not alter the underlying state vector, allowing subsequent operations on the same state. On hardware, sampling consumes the state, requiring re-preparation for further operations.

```
def bell_measure(a: Quant, b: Quant) -> int:
    CNOT(H(a), b)
    # Measures a and b together
    m = measure(a + b)
    return m.get()
```

Fig. 9: Example of `measure`. Output is 0 ($|00\rangle$) or 3 ($|11\rangle$), each with approximately 50% probability.

```
def bell_sample(a: Quant, b: Quant):
    CNOT(H(a), b)
    s = sample(a + b, 1024)

    # Returns 'Samples' object
    return s
```

(a) `s.get()` example: `{0: 529, 3: 495}`.



(b) `s.histogram()` output.

Fig. 10: Example of statistical sampling using the `sample` function and its histogram output. Example with 1024 shots.

Expectation Value Calculation

The `exp_value(o)` function calculates the expectation value of an observable, which is defined using the `with obs()` context manager by summing weighted products of Pauli operators. Figure 11 illustrates observable construction and expectation value calculation. The result is a real number retrieved via `.get()`. Ket also supports automatic gradient calculation for variational algorithms [21].

```
def bell_expv(a: Quant, b: Quant) -> float:
    CNOT(H(a), b)
    with obs():
        A0 = Z(a)
        A1 = X(a)
        B0 = -(X(b)+Z(b))/sqrt(2)
        B1 = (X(b)-Z(b))/sqrt(2)
    # Hamiltonian for CHSH
    h = A0*B0 + A0*B1 + A1*B0 - A1*B1
    ev = exp_value(h)
    return ev.get()
```

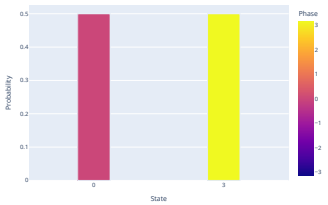
Fig. 11: Example of `exp_value` calculating $\langle h \rangle$ for the CHSH inequality. The expected result is $-2\sqrt{2} \approx -2.8284$.

Simulator State Vector Access

For simulations only, `dump(qubits)` provides access to the quantum state vector. This is a powerful debugging tool but is not physically possible on a real quantum computer. The returned object's `.show()` method displays the state in Dirac notation, and `.get()` returns a dictionary of basis states and their complex amplitudes. Figure 12 illustrates this function usage.

```
def bell_dump(a: Quant, b: Quant):
    CNOT(H(X(a)), b)
    d = dump(a + b)

    # Returns 'QuantumState' object
    return d
```

(a) `d.get()` outputs `{0:0.707, 3:-0.707}`. `d.show()` displays $\frac{1}{\sqrt{2}}|00\rangle - \frac{1}{\sqrt{2}}|11\rangle$.



(b) `d.histogram()` output.

Fig. 12: Example of state vector access in a simulator using the `dump` function.

2.5 KBW Simulator Configuration

The built-in KBW simulator is the default backend and can be configured via keyword arguments in the `Process` constructor. It has two simulation modes:

- The *Dense simulator* is a state-vector simulator that stores the state of n qubits as a complex vector of size 2^n. It is multithreaded and ideal for algorithms that create highly entangled states, but its memory usage grows exponentially.
- The *Sparse simulator* uses a hash map to store only the basis states with non-zero amplitudes. Its performance depends on the number of non-zero amplitudes, making it highly efficient for algorithms where the state remains sparse. For example, it can simulate the 30-qubit GHZ state (Figure 2), which has only two non-zero amplitudes, on a standard laptop.

The main configuration arguments for the `Process` constructor are:

- `num_qubits`: (Integer) Number of qubits. Defaults to 32 for sparse and 12 for dense.
- `simulator`: (String) The simulation mode: `"sparse"` (default) or `"dense"`.
- `execution`: (String) The execution mode: `"live"` (default) or `"batch"`.

3 Quantum Compilation

Analogous to classical high-level programming languages, whose code cannot be directly executed by a CPU, Ket produces high-level quantum circuits that must go through a compilation process to run on a Quantum Processing Unit (QPU). For instance, a program written in Ket may result in circuits with quantum gates that are

not natively supported by the QPU or are not in the set of calibrated gates. Furthermore, multi-qubit gates might be applied to qubits that are not physically connected on the device. The compilation process transforms the high-level quantum circuit into one that contains only QPU-supported gates and respects the device's qubit connectivity. This hardware-compliant circuit is then used for pulse scheduling, which generates the sequence of control pulses that physically manipulate the qubits.

Figure 13 illustrates the quantum software stack. At the top of the stack is the domain-specific application, which utilizes Ket for quantum development and acceleration. The high-level quantum circuit generated by Ket is then compiled by the platform's runtime library, Libket. This compilation process has three primary stages: multi-qubit gate decomposition (Section 3.1), where multi-controlled gates are broken down into single- and two-qubit gates; quantum circuit mapping (Section 3.2), where logical qubits are mapped to physical qubits respecting the device's connectivity; and native gate decomposition (Section 3.3), where the single- and two-qubit gates are further translated into the native gate set supported by the QPU.

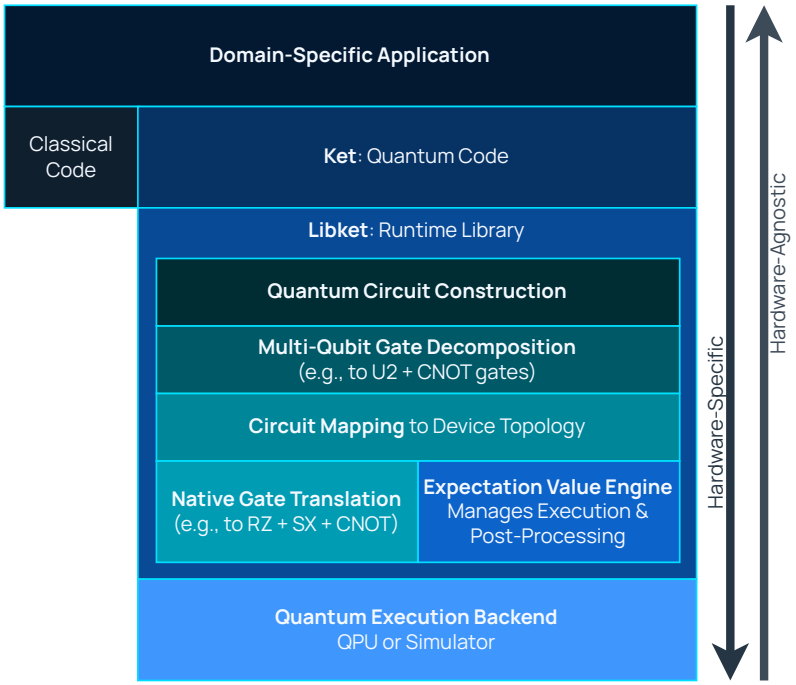


Fig. 13: Quantum software stack.

During compilation, additional processing is required if the desired computational result is an expectation value (Section 3.4), as different quantum circuit executions are often needed to compute the expected value of a Hamiltonian. Additionally, transverse

processes like error mitigation and circuit optimization can be applied throughout the stack to improve the final result.

In this paper, we focus on the quantum software stack for Noisy Intermediate-Scale Quantum (NISQ) computers; therefore, quantum error correction is not treated as a core part of the infrastructure. However, we argue that for future fault-tolerant quantum computers, the higher levels of the stack would remain largely the same. Many of the compilation steps could be repurposed for this new paradigm, with the major change being the addition of a quantum error correction encoding step right before sending the circuit to the quantum computer. Additional processing would also be needed on the quantum computer side for syndrome measurements and the general maintenance of the quantum error correction code.

In this section, we present a concrete example of the quantum compilation process. We will compile the Ket program for the Grover diffusion operator (Figure 14a) for a target QPU with the connectivity graph depicted in Figure 14b. This hardware, inspired by the IQM Garnet quantum computer, has a native gate set consisting of only CZ, R_Z, and $\sqrt{X}$ ($\equiv R_X(\frac{\pi}{2})$) gates.

Note that the Ket code is written in a qubit-agnostic manner; the specific number of qubits is a classical parameter that must be defined before compilation. For this example, we set the number of qubits to 10. This generates the high-level quantum circuit shown in Figure 14c, which serves as the input for our compilation workflow. The subsequent steps of this process are detailed in the remainder of this section.

3.1 Quantum Gate Decomposition

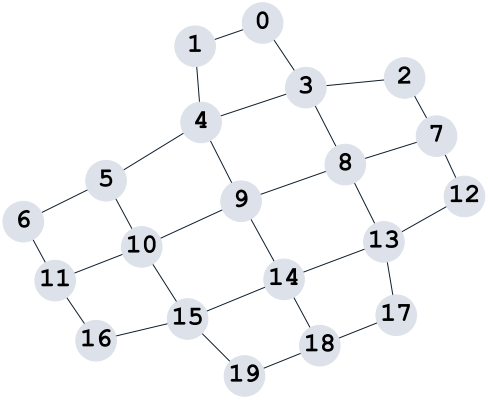
Despite the ability to apply controls to any gate-like callable in Ket, any multi-controlled gate is ultimately a multi-control, single-target gate. This limits the decomposition algorithms required by the compilation process to a finite set of efficient methods [28]. In Ket, decomposition algorithms are only required for multi-controlled Pauli gates, rotation gates ($SU(2)$ gates), Phase gates, and the Hadamard gate. In practice, this represents a small number of distinct gate classes.

Despite the small number of gate classes, a quantum compiler should implement different decomposition algorithms for the same class, as these algorithms have different trade-offs. Key metrics include the resulting circuit depth, the number of CNOTs required, and the need for ancillary qubits. For example, a highly efficient decomposition for multi-controlled Pauli gates results in a circuit with a linear number of CNOT gates and logarithmic depth relative to the number of controls. However, it requires roughly the same number of ancillary qubits as control qubits [28, 29], which may not be available. In such cases, an alternative decomposition algorithm is needed. The choice of the next-best option depends on whether circuit depth or CNOT count should be prioritized.

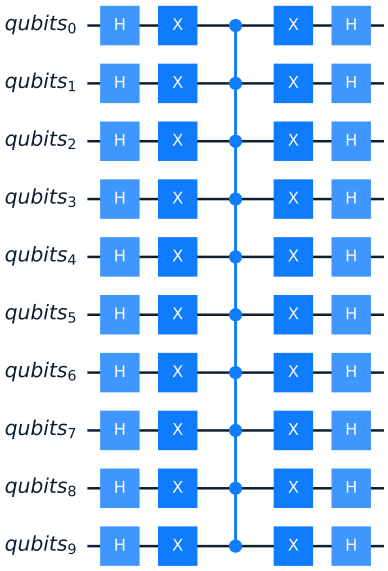
The choice of which decomposition algorithm to use in a given situation is made by the compiler; however, it is important for the programmer to be aware of the available algorithms and their associated complexities, as this can influence program performance. For example, although a single-qubit Phase gate is equivalent to a Z-rotation gate, this is not true for their controlled versions. The decomposition of

```
def diffusion(qubits: Quant):
    with around(cat(H, X), qubits):
        CZ(*qubits)
```

(a) Ket code for the Grover diffusion operator.



(b) Connectivity of the target QPU.



(c) Initial high-level 10-qubit circuit.

Fig. 14: Overview of the compilation example setup. (a) The hardware-agnostic Ket source code. (b) The target hardware connectivity. (c) The high-level circuit generated for a 10-qubit system, which serves as the input to the compiler.

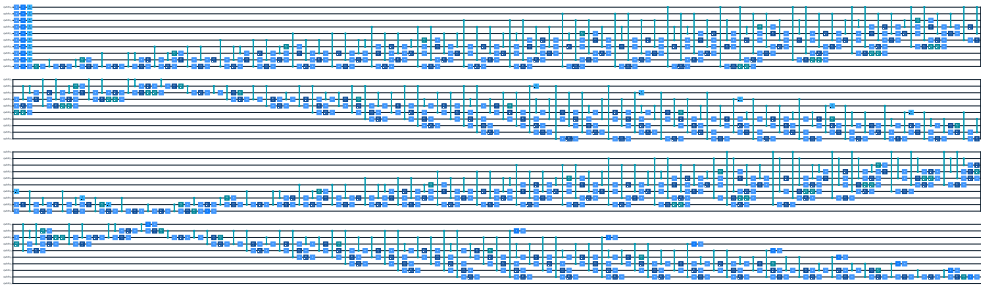
multi-controlled Phase gates requires more resources than that of multi-controlled Z-rotation gates. Therefore, whenever possible, the use of Z-rotation gates is preferable to Phase gates in controlled operations.

In the current NISQ era, with its limited resources, a programmer may be tempted to break abstraction and force the use of a particular decomposition algorithm. Although statically defining the decomposition can result in better performance on certain quantum computers, it may not be optimal on others, making the code's performance hardware-dependent. Conversely, if the compiler dynamically chooses the decomposition algorithm, it can select the appropriate optimization for different architectures. Furthermore, as new and better decomposition algorithms are developed and added to the compiler's pool, code performance can improve over time without manual changes. Code with a statically chosen algorithm will not benefit from these future compiler improvements.

Continuing with our compilation example, the first step is gate decomposition. While knowledge of the target QPU is not strictly necessary at this stage, it allows the compiler to select more efficient decompositions.

If the compiler has no information about the available hardware qubits, it must use a generic, resource-agnostic algorithm. For the multi-controlled Z gate in our example, this corresponds to a linear-depth decomposition that requires a quadratic number of two-qubit gates [30], as shown in Figure 15a.

However, when the compiler is aware of the target hardware–in this case, a 20-qubit QPU with 10 qubits currently unused–it can leverage these free qubits as auxiliary resources. This enables a more efficient decomposition that reduces the total gate count, as shown in Figure 15b. During this stage, the Ket compiler also translates all two-qubit operations into the target's native gate set; therefore, both circuits are expressed using only CZ and one-qubit gates.



(a) Generic decomposition without using auxiliary qubits.

(b) Optimized decomposition using 10 auxiliary qubits available on the target QPU.

Fig. 15: Effect of hardware awareness on gate decomposition. The compiler selects a more efficient strategy when it can use available hardware qubits as auxiliary resources.

3.2 Quantum Circuit Mapping

As compilation progresses, the quantum circuit becomes more hardware-specific. Although the use of only single- and two-qubit gates is a consensus in quantum computer design, the circuit mapping stage is highly dependent on the device's architecture. In this step of the compilation, the logical qubits–those allocated by the `Process` in the high-level program–are mapped to physical qubits. This mapping must respect the qubit connectivity of the QPU. Most often, this process also requires inserting SWAP gates to enable operations between qubits that are not directly connected, which can cause the effective position of a logical qubit to move around the QPU during computation.

Finding the optimal placement of SWAP gates is a computationally difficult problem that requires exponential time to solve exactly[31]; therefore, circuit mapping algorithms rely on heuristics. The state-of-the-art in circuit mapping is based

on the SWAP-based Bidirectional (SABRE) [32] and Dynamic Look-Ahead (DL) heuristics [33], both of which take the coupling graph of the QPU as input.

The quality of the final circuit is highly dependent on the initial mapping (the first assignment of logical-to-physical qubits). Since finding the optimal initial mapping is also a hard problem, heuristics are used to find a good starting point [34]. Many proposed heuristics rely on non-deterministic solvers, which can result in different mappings even for the same input circuit. The DL mapping algorithm, however, proposes deterministic heuristics for the initial mapping. Ket uses the Dynamic Look-Ahead mapping algorithm, as its deterministic nature aids in result reproducibility and helps in identifying performance improvements.

One strategy to improve the initial mapping is to use a forward-and-backward pass approach. The final mapping of a circuit (the position of logical qubits at the end) can be used as the initial mapping for the inverse of that circuit. The final mapping of this inverse circuit can then be used as a new, potentially better, initial mapping for the original circuit. This process can be iterated multiple times, with the best-performing initial mapping chosen from all passes.

Since not all qubits are created equal–the quality of qubits and their connections can vary–some techniques use calibration data to improve the final circuit. This optimization aims not just to reduce the number of SWAPs but to reduce the overall noise during circuit execution [35]. For example, qubits and connections with smaller error rates can be prioritized. For this purpose, single-qubit error rates are incorporated as weights on the nodes of the coupling graph, and two-qubit error rates are incorporated as weights on the edges.

Figure 16 shows the circuit from Figure 15b after circuit mapping, where all two-qubit gates now act exclusively between neighboring physical qubits, respecting the hardware's connectivity graph of Figure 14b. In this example, each SWAP gate is decomposed into three native CZ gates and Hadamard gates.

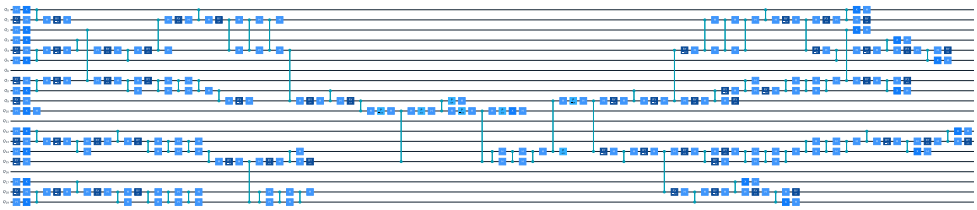


Fig. 16: The circuit after the qubit mapping stage. Logical qubits have been assigned to physical qubits.

3.3 Native Gate Translation

Once the circuit has been mapped to the device topology and decomposed into a basis of single- and two-qubit gates (*e.g.*, arbitrary U(2) and CNOT), the final compilation stage is to translate these gates into the native gate set supported by the quantum hardware. Up until this point, the compiler often works with a convenient intermediate

representation, such as arbitrary single-qubit unitary gates and CNOTs. However, due to physical implementation and calibration constraints, real quantum computers typically only support a limited, discrete set of single-qubit gates and one or two specific two-qubit entangling gates, which may not necessarily be the CNOT gate (*e.g.*, it could be an iSWAP or a controlled-Z gate). The physical aspects related to calibrating a gate set are addressed in Section 4.

The native gate translation step does not strictly need to be separate from the preceding compilation stages like gate decomposition and circuit mapping. However, performing this translation too early in the process can unnecessarily increase the complexity of those prior steps. For instance, mapping and optimization algorithms are often simpler to design and more effective when they can operate on a standardized intermediate gate set. In Ket, a mixed approach is taken: the translation of two-qubit gates may be handled during earlier steps, but the final translation of single-qubit gates into the native basis is typically performed as the last step of the decomposition process. A key optimization at this stage is to synthesize sequences of single-qubit gates. Instead of translating each gate in a sequence individually, their corresponding unitary matrices are first multiplied into a single arbitrary unitary. This resultant unitary is then synthesized into an optimal sequence of native gates, which often reduces the overall gate count and execution time.

After the gates are translated into the native set, the circuit is ready for execution. On a quantum simulator, this might be the final step. On physical quantum hardware, however, this native gate circuit is then passed to a pulse scheduler, which generates the precise sequence of analog control pulses (*e.g.*, microwave or laser pulses) that physically manipulate the qubits. Since pulse scheduling depends not only on the hardware's architecture but also on its calibration data (which can vary over time), this final step is typically performed by the hardware provider's control software.

To conclude the compilation example, the final stage involves translating all remaining single-qubit gates into the native instruction set of the target hardware. The circuit from the previous step (Figure 16) is processed, and each single-qubit gate is decomposed into sequences of R_Z and $R_X(\frac{\pm\pi}{2})$ gates. The final, fully compiled circuit is shown in Figure 17. It is now expressed entirely in terms of the QPU's native gates and is ready for execution.

3.4 Expectation Value Engine

When the desired result of a quantum computation is a one-shot measurement or a statistical sample of the final state, the compilation steps described thus far are sufficient to prepare the program for execution. However, when the goal is to compute the expectation value of a Hamiltonian ($\langle H \rangle$), as is common in variational algorithms, additional steps managed by the Expectation Value Engine are required.

Naively, to estimate the expectation value of a Hamiltonian expressed as a weighted sum of Pauli strings (*e.g.*, $H = \sum_i c_i P_i$), one would need to execute a different circuit for each term P_i in the sum. This involves adding basis-change rotations at the end of the original circuit to align the measurement basis with the Pauli operators in the term before measuring the qubits. As the number of terms in a Hamiltonian can grow significantly with the problem size, this approach can be prohibitively expensive.

Fig. 17: The final compiled circuit for the 10-qubit Grover diffusion operator. All logical operations have been translated into the native hardware gate set (CZ, R_Z, and $R_X(\frac{\pm\pi}{2})$), and all two-qubit gates respect the physical qubit connectivity.

To mitigate this, the engine employs various measurement strategies to reduce the number of distinct circuit executions needed. These techniques rely on identifying and grouping observables that can be measured simultaneously. Prominent strategies include:

- *Qubit-wise Commuting* (QWC) [36]: Groups Pauli strings that are composed of commuting Pauli operators on a qubit-by-qubit basis.
- *General Commuting* [37]: Identifies larger sets of mutually commuting Pauli strings, allowing more terms to be measured from a single circuit execution, though finding optimal groupings can be computationally intensive.
- *Classical Shadows* [38, 39]: A randomized measurement technique that constructs a concise classical description of the quantum state from a small number of measurements. This "shadow" can then be used to estimate the expectation values of many different observables in classical post-processing, drastically reducing the required number of quantum measurements for large Hamiltonians.

Each technique presents a trade-off between the number of quantum circuit executions and the amount of classical post-processing required. The Expectation Value Engine is responsible for executing a measurement strategy by: generating the necessary measurement circuits and reconstructing the final expectation value from the measurement outcomes. For efficiency, the engine typically starts from the circuit produced by the mapping stage, adds the required basis-change gates for a group of measurements, and then sends these new circuits to the final native gate translation stage.

Furthermore, variational quantum algorithms and quantum machine learning models benefit from calculating the gradient of expectation values with respect to parameterized gates. The engine also manages this process. Instead of relying on backpropagation, which is infeasible on quantum hardware, techniques like the

parameter-shift rule [40, 41] or stochastic parameter-shift [42] are used. These methods also require executing circuit variations and are managed by the engine. The parameter-shift rule provides a strong motivation for delaying native gate translation: it is far simpler to modify a single parameter θ in an abstract gate like $\mathrm{RZ}(\theta)$ than to track how that change propagates through a sequence of decomposed fixed native gates.

3.5 Circuit Optimization

Quantum circuit optimization is not a single compilation stage but a transversal process that can be performed at multiple layers of the software stack. The goal is to reduce circuit resources (such as gate count, depth, or CNOT count) to improve execution fidelity on noisy hardware. The available optimization techniques depend on the circuit representation at each layer.

Optimization can range from high-level logical reductions to low-level hardware-aware passes. For instance, at the top of the stack, high-level programming constructs can be used to reduce the number of controlled operations [27]. During compilation, common techniques include canceling adjacent inverse gates ($UU^\dagger \to I$) and merging consecutive single-qubit rotations. More advanced and computationally intensive techniques, such as resynthesizing entire circuit blocks using methods like the ZX-calculus [43], can provide significant reductions in gate count and are an active area of research.

4 Physical Realization of Quantum Programs

The quantum compilation process, detailed in Section 3, transforms a high-level quantum program into a circuit composed of native gates that respects the hardware's physical topology. However, this gate-based description is still an abstraction. A Quantum Processing Unit (QPU) does not execute gates directly; rather, it is a physical system manipulated by precisely timed analog control signals. The final step in running a quantum program is therefore to translate the sequence of native gates into a schedule of control pulses–typically microwave or laser pulses–that are sent to the QPU by classical control hardware.

This translation from the digital abstraction of gates to the analog reality of pulses is made possible by the careful definition and calibration of a native gate set. Each gate in this set, such as the CZ, R_Z, and $\sqrt{X}$ gates from our previous example, corresponds to a pre-characterized pulse shape, frequency, and duration that implements the desired unitary transformation. This section describes what is required at the hardware level to define, calibrate, and execute these pulse-level instructions.

While several technologies exist for building quantum computers, including trapped ions and photonics, we focus here on *superconducting transmon qubits*, one of the most prominent platforms [44, 45]. These qubits are engineered electrical circuits composed of capacitors, inductors, and Josephson junctions. The physics of superconductivity–where electrons form "Cooper pairs" and flow without resistance below a critical temperature–allows these circuits to maintain quantum coherence for long periods [46, Chapter 1]. The Josephson junction provides the necessary non-linearity,

creating unequally spaced quantized energy levels. The lowest two levels are chosen to encode the qubit states $|0\rangle$ and $|1\rangle$, which can then be manipulated by applying electromagnetic pulses at the appropriate transition frequency.

In this section, we describe how these superconducting qubits are programmed and manipulated at the pulse level, which constitutes the lowest layer of the quantum software stack. We will cover the calibration procedures required to characterize the hardware and ensure high-fidelity operations. We focus on the principles of quantum control, abstracting away details of low-level circuit design; for a more complete treatment of the physical implementation, we refer the reader to [47] and [48]. We begin by introducing the physical principles behind qubit control (Section 4.1), then present the essential calibration routines for single-qubit gates (Section 4.2), multi-qubit gates (Section 4.4), and measurement (Section 4.5).

4.1 Control Pulse Theory

As briefly introduced before, quantum gates are implemented using microwave pulses. From quantum mechanics, we know that a two-level quantum system undergoes coherent oscillations between its basis states, $|0\rangle$ and $|1\rangle$, when driven by a resonant oscillatory field [49, Section 7.5.3]. This coherent population transfer is known as Rabi oscillation. Constructing quantum gates essentially means precisely controlling the Rabi frequency (which sets how fast the state oscillates between $|0\rangle$ and $|1\rangle$) and the rotation axis on the Bloch sphere, e.g., x or y. By adjusting these parameters and the pulse duration, we steer the quantum state to the desired point.

A pulse is characterized by a waveform, typically represented by sine or cosine functions. In this context, we commonly utilize IQ modulation, which involves two distinct pulses: the in-phase (I) and the quadrature (Q), with a phase shift of $\pi/2$. The final pulse is obtained by combining the I and Q components, which results in a pulse with an amplitude and phase that depends on the two components. This can be visualized in Figure 18, where ω represents the frequency of the pulse, $A(t) = A_I(t) + iA_Q(t)$ denotes the dimensionless complex envelope (amplitude modulation), ϕ is the phase, t is time, and $V(t) = V_I(t) + V_Q(t)$ is the time-dependent voltage signal applied to the system.

When building control pulses, we modulate the attributes of the control pulse. The frequency ω is typically set to the qubit's resonance frequency to drive coherent $|0\rangle \leftrightarrow |1\rangle$ oscillations. The phase ϕ determines the rotation axis of the Rabi oscillation on the Bloch sphere. The amplitude A values indicate how fast the quantum state oscillates: the higher the value, the faster the quantum state oscillates. Therefore, to build faster gates, higher amplitude values are needed, and for slower gates, lower amplitude values suffice.

4.1.1 Control Hamiltonian

The complex envelope $A(t)$ defines the in-phase and quadrature components of the pulse as programmed in the control system. These are dimensionless quantities that specify the analog waveform to be delivered by the control electronics, typically constrained within a unit disk, $|A(t)| \leq 1$. However, the actual interaction strength

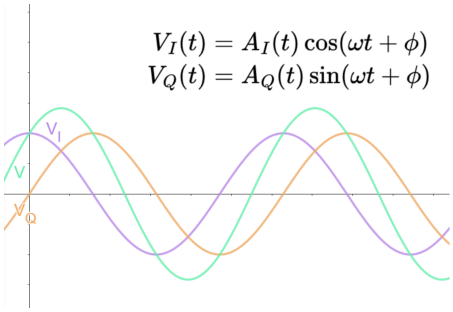


Fig. 18: Illustration of IQ modulation. The in-phase (I) and quadrature (Q) components, shifted by $\pi/2$, combine to form a pulse whose amplitude and phase are determined by their relative contributions.

between the control field and the qubit is described by the Hamiltonian (adapted from [50]):

$$H(t) = \frac{1}{2}\left(I(t)\sigma_x + Q(t)\sigma_y\right), \tag{4.1}$$

where $\gamma(t) = I(t) + iQ(t)$ is the control pulse waveform in Rabi frequency units. The physical quantities I and Q are proportional – though not exactly linear – to the voltages delivered to the qubit and determine the rotation rate and axis of the state on the Bloch sphere. In this representation, the I component drives rotations around the x-axis of the Bloch sphere, while the Q component drives rotations around the y-axis. Intermediate phase values drive rotations around other axes in the xy-plane.

It is possible to determine the rotation angle induced by a control pulse by integrating the control Hamiltonian over time. For an in-phase control pulse (with $Q(t) = 0$), the rotation angle Θ around x-axis is given by [47, Section IV D]:

$$\Theta = \int_0^t I(t')dt'. \tag{4.2}$$

This means that, for example, to implement a π-pulse around x-axis, one must choose the waveform $I(t)$ such that the total area under the pulse satisfies $\Theta = \pi$.

4.1.2 Waveform Sampling

In this theoretical model, we define the control pulse as a continuous complex-valued function $A(t)$. This function represents the envelope of the analog voltage signal that drives the qubit. However, in practice, the generation of such pulses is constrained by the limitations of the control hardware. Specifically, the control electronics rely on digital waveform generation, in which the continuous function $A(t)$ is represented by a discrete sequence of samples:

$$A[n] = A(n.dt) \tag{4.3}$$

where dt is the sampling time, a fixed temporal resolution determined by the sampling rate of the control system (*e.g.*, $1\,\mathrm{GS/s} = 1\,\mathrm{ns}$). This means that the pulse envelope is not specified as a smooth, continuous curve, but rather as a list of values at fixed intervals.

The discrete samples are then sent to a Digital-to-Analog Converter (DAC) [51], which transforms the digital sequence into an analog voltage signal $V(t)$. This analog signal is what is physically transmitted through the control line to the qubit. However, the DAC output is not a perfect stepwise voltage; instead, the actual signal $V(t)$ is an approximation of the ideal piecewise-constant waveform implied by the digital samples. From a signal processing perspective, the amplitude modulation of this signal can be analyzed using the Fourier transform, which reveals that the control pulse may contain a broad spectrum of frequency components beyond the target frequency ω.

An intuitive analogy can be made with musical instruments, such as a guitar. When a string is plucked to produce a note at a certain frequency, the sound emitted is not a pure sine wave, but rather a combination of multiple frequencies due to the finite duration and mechanical properties of the excitation. These additional frequency components may resonate with other strings, causing them to vibrate sympathetically even if they were not directly plucked. Similarly, in a quantum processor, a control pulse aimed at a specific qubit transition may unintentionally contain spectral components that resonate with other transitions–either within the same qubit (causing leakage to higher levels) or in neighboring qubits (leading to crosstalk).

Because hardware cannot perfectly reproduce pulses with abrupt transitions or extremely wide bandwidths, control sequences in experiments are often designed with these physical limitations in mind.

4.1.3 Pulse Shaping

The amplitude modulation of a control pulse plays a crucial role in determining the fidelity of quantum operations. The goal of pulse shaping is to modulate the amplitude in a way that enhances gate fidelity and increases robustness against various sources of error. There are three main approaches to pulse design: analytical techniques, quantum optimal control, and reinforcement learning.

In analytical techniques, optimal pulse shapes are derived based on a complete geometric understanding of the system dynamics [52]. These pulses are described by mathematical expressions with tunable parameters. Quantum optimal control, on the other hand, formulates pulse shaping as a numerical optimization problem. In reinforcement learning, control pulses are learned directly from interactions with the quantum hardware.

Here we describe some common pulse shaping methods utilized in these approaches.

Basic Pulse Shapes

The square pulse, shown in purple in Fig. 19, represents a control signal whose amplitude remains constant throughout its duration. Despite its simplicity, the square pulse has very sharp rise and fall edges, which introduce unwanted high-frequency content when analyzed in the frequency domain. These high frequencies can cause leakage errors, where the qubit unintentionally transitions to states beyond $|1\rangle$, such as $|2\rangle$

or $|3\rangle$. This happens because some of the frequency components of the pulse may match the transition frequencies to those higher energy levels, and can do so with significant strength. In addition, ideal square pulses cannot be physically implemented because creating perfectly sharp edges would require changing the signal infinitely fast–something that is not possible with real electronic equipment.

The effect of a square pulse on a qubit is straightforward to analyze, as it corresponds to the area under a constant amplitude waveform–*i.e.*, a rectangle. For example, to implement a $\pi/2$ rotation over a duration of 60 ns, we can compute the required Rabi frequency as $\frac{\pi/2}{60\,\mathrm{ns}} \approx 26\,179\,938\,\mathrm{rad\,s^{-1}} \approx 4.167\,\mathrm{MHz}$. The Rabi frequency defines the rate at which the qubit state oscillates between $|0\rangle$ and $|1\rangle$ and is directly related to the pulse amplitude. To determine the amplitude that produces this target Rabi frequency, one typically performs a Rabi oscillation experiment and may need to fine-tune the amplitude-to-frequency mapping. This calibration process is discussed in detail in Section 4.2. The rotation axis on the Bloch sphere is controlled by the phase of the pulse, which in turn depends on the in-phase (I) and quadrature (Q) components: applying the pulse through the I channel yields an x rotation, while using the Q channel results in a y rotation.

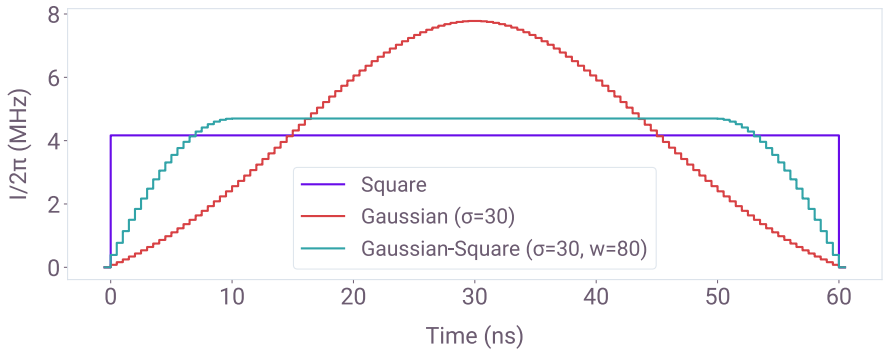


Fig. 19: Illustration of three basic pulse shapes in the in-phase (I) quadrature: square (purple), Gaussian (red), and Gaussian-square (green). Each pulse implements a $\pi/2$ rotation with a total duration of 60 ns and a sampling time of 0.5 ns.

To suppress the high-frequency components of a square pulse, Gaussian-shaped envelopes are commonly used (see Fig. 19). For the Gaussian-based pulse shapes considered below, we define a normalization function

$$\mathcal{N}[g](t) = A\,\frac{g(t) - g(-1)}{1 - g(-1)}\,, \tag{4.4}$$

where $g(t)$ is the unnormalized envelope and A is the amplitude scaling factor. The subtraction at $t = -1$ ensures that $\mathcal{N}[g](t)$ vanishes exactly one time step before ($t = -1\,dt$) and after ($t = d + 1\,dt$) the pulse duration.

The Gaussian envelope is defined as

$$g_{\text{gauss}}(t) = \exp\left[-\frac{(t-d/2)^2}{2\sigma^2}\right], \tag{4.5}$$

and the normalized waveform is simply $f_{\text{gauss}}(t) = \mathcal{N}[g_{\text{gauss}}](t)$. Here, d is the total pulse duration and σ is the standard deviation controlling the smoothness of the rise and fall of the pulse.

While Gaussian pulses offer smoother transitions and attenuate some high-frequency components caused by abrupt amplitude changes, maintaining the same area under the curve–or equivalently, the same rotation–requires higher amplitude values near the center of the Gaussian shape. When implementing faster Gaussian pulses, the issue of high-frequency components arise again, similarly to square pulses, but now it is a consequence of the pulse shape itself rather than its implementation.

An intermediary between Gaussian and square pulses is the Gaussian-square pulse (also shown in Fig. 19). This pulse shape combines Gaussian rise and fall sections with a square (constant) plateau at its center. The unnormalized envelope is given by

$$g_{\text{gsq}}(t) = \begin{cases} \exp\left(-\dfrac{(t-r)^2}{2\sigma^2}\right), & t < r, \\ 1, & r \le t < r+w, \\ \exp\left(-\dfrac{(t-(r+w))^2}{2\sigma^2}\right), & t \ge r+w\,, \end{cases} \tag{4.6}$$

and the normalized waveform is $f_{\text{gsq}}(t) = \mathcal{N}[g_{\text{gsq}}](t)$. Here, σ represents the standard deviation of the Gaussian rise and fall, w denotes the width of the plateau, and $r = d - w$ is the risefall factor, with d being the total pulse duration.

DRAG

To suppress leakage errors from fast Gaussian pulses, a technique called Derivative Removal by Adiabatic Gate (DRAG) [53] is often employed for implementing single-qubit gates. This approach involves a pulse shape combining a standard Gaussian envelope in the I component with an additional derivative component in the Q component. The pulse can be written as

$$f(t) = f_{\text{gauss}}(t)\left[1 - i\,B\left(\frac{t-d/2}{\sigma^2}\right)\right], \qquad 0 \le t < d\,, \tag{4.7}$$

$f_{\text{gauss}}(t)$ is the normalized Gaussian envelope. Here, all parameters retain the same meaning as previously defined, except for B, which determines the strength of the Q-quadrature correction.

The idea behind the DRAG waveform is to apply an additional quadrature component (Q) to the control pulse in order to mitigate leakage into higher energy levels. Without this correction, the I component of the pulse (typically shaped as a Gaussian envelope) exhibits Fourier components at frequencies close to the $|1\rangle \to |2\rangle$ transition,

which can inadvertently drive population out of the computational subspace. By introducing the properly scaled Q component, the DRAG pulse cancels out these unwanted high-frequency contributions, thereby reducing the spectral weight around the leakage transition.

Quantum Optimal Control

Quantum optimal control tackles the pulse design problem by directly formulating it as a numerical optimization task. The goal is to find control fields–typically piecewise-constant or smoothly varying waveforms–that drive the quantum system to implement a target unitary operation with high fidelity.

The challenge lies in the high dimensionality and nonlinearity of the control landscape. Control pulses must respect physical constraints, such as bandwidth and amplitude limits, while also compensating for hardware imperfections and environmental noise. Furthermore, the cost functions involved (*e.g.*, gate infidelity) are often highly non-convex, requiring advanced optimization strategies to avoid local minima and ensure convergence to effective solutions.

Within this framework, the Hamiltonian in Equation 4.1 serves as the control model, and can be extended to account for error processes. For instance, dephasing noise $\eta(t)$ and amplitude miscalibrations $\beta(t)$ can be incorporated as:

$$H_{\text{robust}}(t) = (1 + \beta(t))H(t) + \eta(t)\sigma_z, \tag{4.8}$$

which enables the synthesis of control pulses that are intrinsically robust to these types of noise.

Tools such as *Boulder Opal* by Q-CTRL exemplify this approach. By leveraging numerical optimization over realistic noise models [50], they produce high-fidelity, noise-robust pulse shapes. More recently, Boulder Opal has integrated reinforcement learning with optimal control, using autonomous agents to discover optimal pulses directly from hardware interaction [54], bridging model-based and data-driven control paradigms.

The techniques discussed so far are necessary for designing high-fidelity waveforms that implement quantum logic gates. However, achieving reliable quantum operations on actual hardware also requires a precise calibration of system parameters and control channels. The next sections present key calibration procedures that enable accurate control of quantum systems and ensure that the pulses designed theoretically can be faithfully implemented on physical devices.

4.2 Single-Qubit Gates Calibration

A fundamental goal of calibration is to enable the accurate implementation of a set of quantum gates that is universal for quantum computation. In this context, any quantum algorithm can be decomposed into sequences of single-qubit gates and at least one two-qubit entangling gate, such as the CNOT [49, Section 4.5.2]. Various universal gate sets are known, including $\{\text{CNOT}, H, T\}$ [55], $\{\text{CNOT}, R_y(\pi/4), S\}$ [56], and $\{\text{Toffoli}, H\}$ [57]. For example, IBMQ devices adopt a native gate set consisting of $\{\sqrt{X}, R_z, \text{CNOT}\}$.

For a universal set of single-qubit operations, a widely used choice is $\{\sqrt{X}, R_z\}$, motivated by the fact that arbitrary single-qubit unitaries can be efficiently decomposed using only two $\sqrt{X}$ (*i.e.*, $R_x(\pi/2)$) gates and three R_z rotations. As shown in [58] and, any unitary operation $U(\theta, \phi, \lambda)$ can be expressed as:

$$U(\theta, \phi, \lambda) = R_z(\lambda)\sqrt{X}R_z(\theta)\sqrt{X}^{-1}R_z(\phi). \tag{4.9}$$

This decomposition breaks down an arbitrary single-qubit gate with three Euler angles U into five other quantum gates: two $\sqrt{X}$ (or $R_x(\pi/2)$) gates and three R_z gates. Such decomposition enables the implementation of arbitrary single-qubit gates using just a $\sqrt{X}$ gate, given the virtual implementation of R_z gates (*i.e.*, through phase adjustments in the control system, as discussed in Section 4.2.3).

An alternative approach involves implementing a U gate using arbitrary x rotations. While this method reduces the number of quantum gates, it relies on the capability to execute arbitrary x rotations:

$$U(\theta, \phi, \lambda) = R_z(\lambda - \pi/2)\, R_x(\theta)\, R_z(\phi - 3\pi/2). \tag{4.10}$$

The calibration process of quantum gates consists of adjusting the control hardware parameters in order to implement the desired theoretical control pulse. For single-qubit gates, we need to calibrate the $(I, Q) \leftrightarrow (A_I, A_Q)$ mapping–that is, determine the hardware input amplitudes A_I and A_Q that produce the desired Rabi frequencies I and Q. In what follows, we describe a typical process for this calibration. In Section 4.2.1, we introduce a standard Rabi experiment that provides an overview of this mapping. Then, in Section 4.2.2, we show how to fine-tune it. Finally, in Section 4.2.3, we explain how virtual Z gates are implemented in practice.

4.2.1 Rabi Experiment

The first step in constructing quantum gates is to understand how the hardware responds to control pulses. Recall that the complex envelope $A(t) = A_I(t) + iA_Q(t)$ defines the pulse shape as programmed in the control system, while the physical effect on the qubit is governed by the components $I(t)$ and $Q(t)$ in the control Hamiltonian (Equation 4.1).

The Rabi experiment consists of calibrating the relation $(I, Q) \leftrightarrow (A_I, A_Q)$ by fixing the frequency and phase of the pulse and sweeping over different envelope amplitudes (A). For each amplitude, the qubit is driven with a constant pulse over a range of durations, and the resulting state populations are measured.

For a fixed drive amplitude, the population of the qubit excited state oscillates over time due to Rabi oscillations. This behavior can be modeled by

$$P_1(t) = \mathcal{A}\cos^2(\Omega\pi t + \Phi) + \delta, \tag{4.11}$$

where t denotes the evolution time, Ω is the Rabi frequency scaled to a full rotation (2π), and $\mathcal{A}$, Φ, and δ are fitting parameters.

To illustrate, Figure 20 shows the Rabi oscillation obtained from qubit 0 on the IBMQ Kyoto (retired) quantum computer using a pulse with an amplitude of 0.005. The data points represent the experimental results, while the continuous line corresponds to the fitted function $P_1(t)$. In this specific instance, the extracted Rabi frequency is approximately 668 584 Hz, indicating that the qubit state oscillates between $|0\rangle$ and $|1\rangle$ roughly every $1/668584 \approx 1496$ ns. Thus, to implement an X gate using a pulse of amplitude 0.005 on this qubit, a pulse duration of approximately $1496\,\text{ns}/2 \approx 748$ ns would be required.

The function fitting process is repeated for each selected amplitude value to derive the corresponding Rabi frequency Ω. As a result, we establish a correlation between the hardware input amplitude A and the Rabi frequency Ω by interpolating the experimental pairs (Ω, A). The final outcome of a Rabi experiment conducted on qubit 0 of Kyoto is shown in Figure 21. In this experiment, Rabi frequencies were determined for 18 amplitude values ranging from 0.005 to 0.45 (with results mirrored for negative values). This plotted function serves as a fundamental reference for implementing quantum gates, providing the pulse amplitude corresponding to a given Rabi frequency. When designing quantum gates, with the duration and rotation angle defined, we calculate the required Rabi frequency for the given time interval and use this function to obtain the appropriate amplitude.

4.2.2 Fine-Tuning

The interpolation of Rabi experiment data does not perfectly represent the relationship $\Omega \leftrightarrow A$. The exact Rabi frequencies are known only for the specific set of amplitudes used in the experiment. As a result, estimating frequencies for other amplitudes may introduce under- or over-rotations. To improve this mapping, the control pulse requires fine-tuning. This process typically involves sweeping the amplitude around those estimate by the interpolation to identify the amplitude that achieves the highest fidelity[3].

[3] Fidelity measures the closeness between quantum states. For pure states $|\psi\rangle$ and $|\phi\rangle$, it is defined as $F(|\psi\rangle, |\phi\rangle) = |\langle\psi|\phi\rangle|^2$ [49, Section 9.2.2].

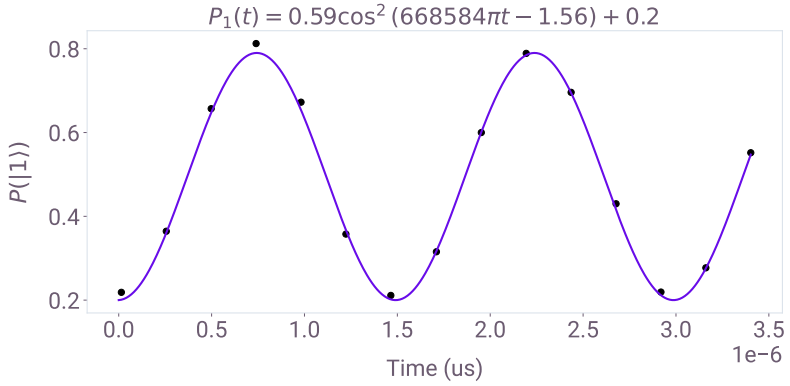


Fig. 20: Rabi oscillation obtained from qubit 0 on the IBMQ Kyoto quantum computer using a pulse with an amplitude of 0.005.

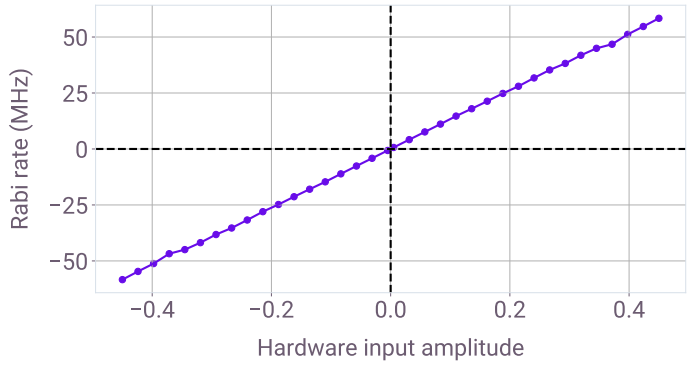


Fig. 21: Rabi experiment results conducted on qubit 0 of the IBMQ Kyoto quantum computer.

Specifically, the goal is to determine the parameters S_{amp} and S_{rel} in the following equation to maximize fidelity [50, 54]:

$$A_{\text{fine-tuned}}(t) = S_{amp}\left(S_{rel}A_I(t) + iA_Q(t)\right). \tag{4.12}$$

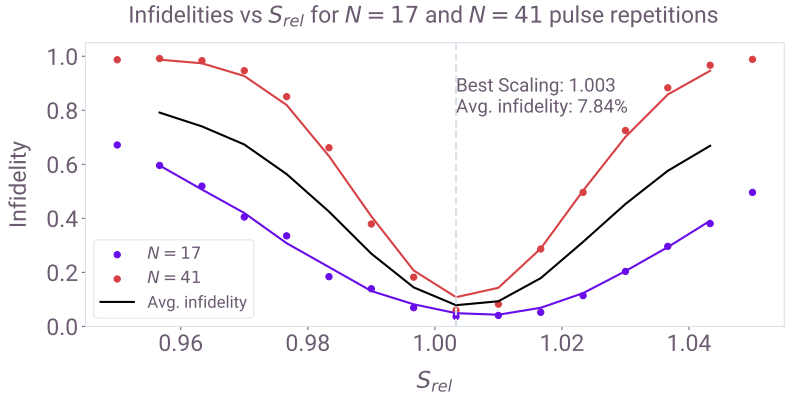


Fig. 22: Fine-tuning experiment for a $\pi/2$ pulse on IBMQ Brisbane, using a 256 ns Q-CTRL dephasing-robust control. The plot shows the infidelity as a function of the relative scaling S_{rel} for $N = 17$ and $N = 41$ repetitions. The black curve denotes the averaged infidelity, used to extract the optimal S_{rel}.

Figure 22 presents an example of a fine-tuning experiment. The control pulse amplitude A was swept around the value estimated by the interpolation, and the resulting fidelity was measured for two different pulse repetitions. Repeating the control pulse serves to amplify any systematic errors, making them more observable. The highest

measured value of S_{rel} was approximately 1.003, with an average infidelity of about 7.84% across the two repetitions.

4.2.3 Virtual Z Gates

Unlike x or y rotations, which require the application of shaped microwave pulses in the xy-plane, rotations around the z axis can be implemented virtually–without applying any physical pulse. These so-called virtual Z gates are realized by simply updating the phase reference frame of subsequent control pulses. As a result, z rotations have effectively zero duration and do not introduce additional errors or hardware overhead [58].

From the control pulse perspective, a rotation in the xy-plane of the Bloch sphere is implemented by modulating the relative phase ϕ of the IQ envelope. We can rewrite the Hamiltonian from Equation 4.1 as:

$$H(t) = \frac{\gamma(t)}{2} \left[\cos(\phi)\sigma_x + \sin(\phi)\sigma_y\right]. \tag{4.13}$$

The phase ϕ determines the rotation axis in the xy-plane. For a constant amplitude pulse γ_c over a duration T, the unitary operator becomes:

$$U = \exp\left(-i\frac{\gamma_c T}{2}\left[\cos(\phi)\sigma_x + \sin(\phi)\sigma_y\right]\right). \tag{4.14}$$

Thus, by changing only the phase ϕ of the control signal, one can rotate around any axis in the xy-plane without modifying the actual waveform. For instance, $\phi = 0$ corresponds to a rotation around the x axis, while $\phi = \pi/2$ corresponds to a rotation around the y axis.

Since ϕ sets the rotation axis, a Z rotation can be implemented by adjusting the phase. A rotation with an arbitrary phase ϕ_0 can be expressed in terms of a standard $R_x(\theta)$ rotation and virtual Z gates:

$$e^{-i\frac{\theta}{2}\left[\cos(\phi_0)\sigma_x + \sin(\phi_0)\sigma_y\right]} = R_z(-\phi_0)R_x(\theta)R_z(\phi_0), \tag{4.15}$$

which shows that a phase shift between pulses implements a virtual Z rotation [58]. This operation is performed entirely in software, without requiring any physical pulses, and is therefore considered noiseless and instantaneous.

We have described the theory and the calibration procedures necessary to implement single-qubit gates. Before addressing the more complex case of two-qubit gates, we first introduce a key preliminary step: determining the transition frequency of each qubit. This frequency calibration is a prerequisite for gate operations, as it determines the carrier frequency of control pulses.

4.3 Finding the Qubit Frequency

Superconducting qubits, such as transmons, are physical systems whose energy levels resemble those of a quantum harmonic oscillator–an idealized system with equally

spaced energy levels. However, transmons differ from the ideal case in a crucial way: the spacing between their energy levels is not exactly uniform. This deviation from perfect regularity is known as anharmonicity, and in the case of transmons, it is small, so we say they are weakly anharmonic.

Despite having multiple energy levels, the dynamics of a transmon are typically restricted to the two lowest ones, $|0\rangle$ and $|1\rangle$. The energy gap between these levels defines the qubit transition frequency ω, which determines the resonance condition for the microwave pulses used to implement single-qubit gates.

A commonly used model to describe this behavior is the Duffing oscillator, which captures the anharmonic nature of the energy level spacings. Its Hamiltonian in the rotating frame is given by [47, Section II.A]:

$$H = \omega a^\dagger a + \frac{\alpha}{2} a^\dagger a^\dagger a a, \tag{4.16}$$

where a ($a^\dagger$) are operators that lower (raise) the energy level of the system, ω is the transition frequency between $|0\rangle$ and $|1\rangle$, and $\alpha < 0$ is the anharmonicity. The negative sign ensures that transitions to higher levels like $|2\rangle$ are off-resonant, allowing the system to behave effectively as a qubit.

Precisely characterizing the frequency ω of each qubit is a crucial calibration step, as this frequency defines the carrier of the microwave pulses used to perform single-qubit rotations. Deviations from resonance result in phase accumulation errors and reduced gate fidelity.

Here we describe how to experimentally determine this frequency using spectroscopy techniques. The basic idea is to prepare the qubit in its ground state $|0\rangle$ and apply a low-amplitude microwave pulse of fixed duration at varying frequencies. After each pulse, a measurement is performed to determine the probability of the qubit being excited to the $|1\rangle$ state.

When the drive frequency is far detuned from the qubit transition frequency ω, the applied pulse does not effectively couple to the qubit, and the system remains in the ground state. This is because off-resonant drives fail to match the energy gap required for the $|0\rangle \to |1\rangle$ transition, resulting in negligible population transfer. As the drive frequency approaches resonance, the probability of excitation increases, reaching a maximum when the frequency precisely matches the energy difference $\hbar\omega$.

This behavior results in a frequency-dependent excitation profile, as illustrated in Figure 23, with a characteristic Lorentzian shape centered around the true qubit frequency. The measured excitation probability as a function of drive frequency f is typically fitted to a function of the form:

$$\mathcal{L}(f) = -A \cdot \frac{\Gamma^2}{(f-\omega)^2 + \Gamma^2} + B, \tag{4.17}$$

where A is the amplitude of the curve, B is an offset, ω is the resonance frequency, and Γ is the linewidth (related to the qubit's decoherence and energy relaxation times). The center of this Lorentzian peak provides an accurate estimate of the qubit transition frequency ω, which is then used to calibrate the carrier frequency of control pulses.

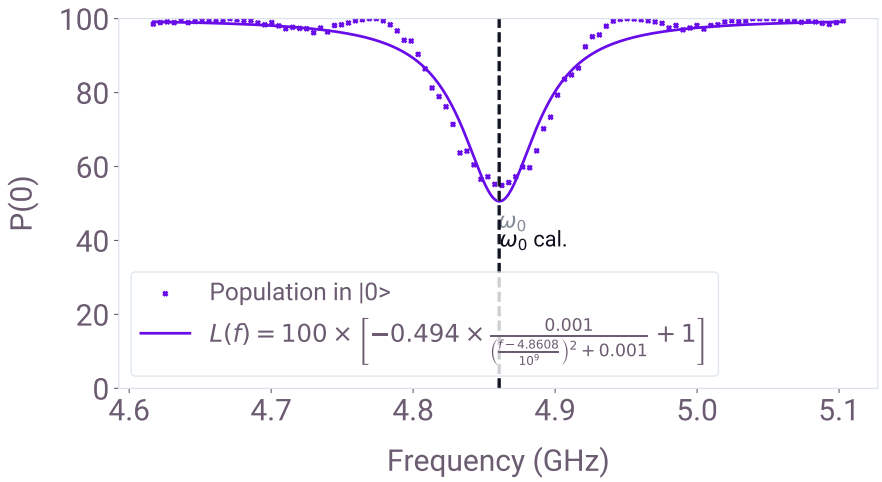


Fig. 23: Frequency spectroscopy of a superconducting qubit obtained with the Qiskit Pulse simulator. The probability of measuring the ground state $|0\rangle$ is plotted as a function of the drive frequency. The dip corresponds to the qubit resonance at ω_0 (gray dashed line), while ω_0^{cal} (black dashed line) indicates the calibrated frequency. The solid line is a Lorentzian fit.

Beyond enabling single-qubit control, knowledge of each qubit's transition frequency also plays a central role in multi-qubit gate calibration. In particular, the relative detuning between qubits determines the strength and nature of their effective coupling, whether mediated by fixed-frequency interactions–as in the cross-resonance gate–or by tunable elements, as in iSWAP and CPHASE gates. In the next section, we show how multi-qubit gates can be implemented.

4.4 Multi-Qubit Gates Calibration

Unlike single-qubit gates–which are typically implemented using direct microwave drives–there are multiple techniques available for realizing multi-qubit gates in superconducting qubits. The choice of technique depends mostly on the underlying hardware design, such as whether the qubits are fixed-frequency or frequency-tunable.

Up to this point, we have abstracted many low-level hardware details to focus on the software and control aspects of gate calibration. However, as we move to multi-qubit interactions, these hardware characteristics become more relevant and must be explicitly considered. In particular, the ability to tune the qubit frequency or exploit specific types of couplings determines which two-qubit gate protocols can be implemented.

Some qubits are equipped with a flux line, which allows dynamic tuning of this frequency by adjusting the magnetic flux threading the qubit loop. This tunability enables gates that rely on frequency detuning, such as the iSWAP (Section 4.4.1) and CPHASE (Section 4.4.2) gates.

On the other hand, qubits without flux lines–*i.e.*, fixed-frequency qubits–typically implement multi-qubit gates using the cross-resonance technique (Section 4.4.3). In this method, a microwave pulse resonant with one qubit is applied through the control line of another qubit, inducing a controllable interaction between them.

Each of these methods introduces its own calibration challenges, such as mitigating flux noise in tunable qubits or dealing with spurious interactions in cross-resonance schemes. In this section, we explore how these multi-qubit gates are implemented and calibrated in practice.

4.4.1 iSWAP

The iSWAP quantum gate [59] arises naturally from the coherent interaction between two coupled qubits. The dominant interaction term between two capacitively coupled qubits can be expressed as follows [47, Section IV.E.1]:

$$H = g\,\sigma_y^{(1)} \otimes \sigma_y^{(2)}, \tag{4.18}$$

where g is the coupling strength, and $\sigma_y^{(i)}$ denotes the Pauli-Y operator acting on qubit i.

To better understand the dynamics induced by this interaction, it is useful to express the Hamiltonian in terms of the ladder operators $\sigma_\pm = (\sigma_x \pm i\sigma_y)/2$. In the interaction picture and applying the rotating wave approximation (RWA), fast-oscillating terms are neglected under the assumption that they average out over time due to their high frequency. This leads to the simplified interaction Hamiltonian:

$$H = g\left(e^{i\delta\omega_{12}t}\,\sigma_+^{(1)}\sigma_-^{(2)} + e^{-i\delta\omega_{12}t}\,\sigma_-^{(1)}\sigma_+^{(2)}\right), \tag{4.19}$$

where $\delta\omega_{12} = \omega_1 - \omega_2$ is the frequency detuning between the two qubits. This expression describes an energy-conserving exchange of excitations between the two qubits.

If we tune the qubits such that their frequencies match, *i.e.*, $\omega_1 = \omega_2$, then $\delta\omega_{12} = 0$ and the Hamiltonian becomes time-independent:

$$H = g\left(\sigma_+^{(1)}\sigma_-^{(2)} + \sigma_-^{(1)}\sigma_+^{(2)}\right) = \frac{g}{2}\left(\sigma_x^{(1)}\sigma_x^{(2)} + \sigma_y^{(1)}\sigma_y^{(2)}\right). \tag{4.20}$$

This is the well-known XY-type interaction Hamiltonian, which preserves the total excitation number and is responsible for coherent population exchange between the qubits.

The time evolution under this Hamiltonian is governed by the unitary operator:

$$U(t) = \exp\left(-i\frac{g}{2}\left(\sigma_x \otimes \sigma_x + \sigma_y \otimes \sigma_y\right)t\right) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(gt) & -i\sin(gt) & 0 \\ 0 & -i\sin(gt) & \cos(gt) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{4.21}$$

By allowing this interaction to evolve for a time $t' = \pi/2g$, the unitary becomes:

$$U(\pi/2g) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -i & 0 \\ 0 & -i & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \equiv \text{iSWAP}. \tag{4.22}$$

Alternatively, evolving for half this time $t'' = \pi/4g$ leads to the so-called square-root of iSWAP gate:

$$U(\pi/4g) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{\sqrt{2}} & -\frac{i}{\sqrt{2}} & 0 \\ 0 & -\frac{i}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \equiv \sqrt{\text{iSWAP}}. \tag{4.23}$$

The iSWAP gate performs the following transformations on the two-qubit computational basis:

$$\begin{aligned} |00\rangle &\to |00\rangle\,, \\ |01\rangle &\to -i\,|10\rangle\,, \\ |10\rangle &\to -i\,|01\rangle\,, \\ |11\rangle &\to |11\rangle\,. \end{aligned}$$

This gate effectively swaps the $|01\rangle$ and $|10\rangle$ states up to a phase factor of $-i$, while leaving $|00\rangle$ and $|11\rangle$ unchanged. Because of this behavior, the iSWAP gate is often used as an entangling gate in superconducting qubit platforms. Furthermore, gates such as CNOT and SWAP can be constructed from iSWAP and single-qubit rotations, as illustrated in Figure 24.

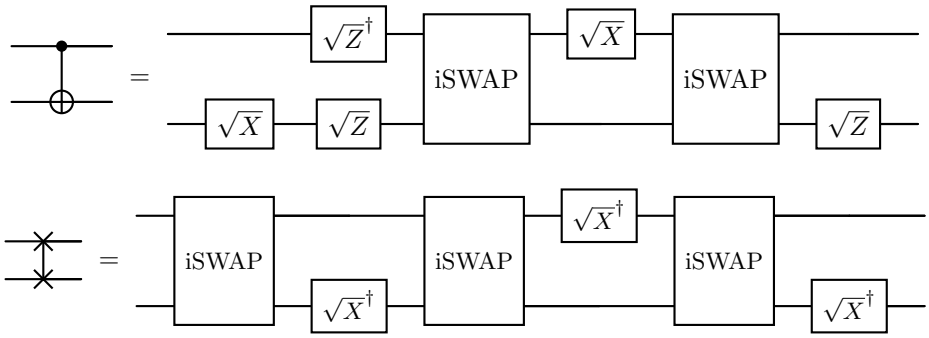


Fig. 24: CNOT and SWAP decomposition using iSWAP gates and single-qubit rotations [59].

4.4.2 CPHASE

To implement a controlled-phase (CPHASE) interaction, we use a technique conceptually similar to that used for the iSWAP gate, but now explicitly take into account the presence of higher energy levels. Recall that for flux-tunable qubits, the qubit frequency can be adjusted by applying an external magnetic flux, thereby changing the energy level spacings. Up to this point, however, we have considered only single-qubit effects and neglected qubit-qubit interaction.

In reality, when two coupled quantum states are brought near resonance, their energy levels repel each other due to hybridization. This phenomenon is known as an avoided level crossing. For example, when we increase the flux applied to qubit 1 to bring its frequency close to that of qubit 2 (*i.e.*, $\omega_1 \approx \omega_2$), the energy of $|10\rangle$ decreases until it approaches that of $|01\rangle$. However, due to the coupling, the two states do not cross but instead split, indicating a strong resonant exchange interaction. At this operating point, coherent oscillations between $|10\rangle$ and $|01\rangle$ can be used to implement the iSWAP gate. Moreover, because of the avoided crossing, further increasing the flux shifts the frequency of qubit 2 rather than that of qubit 1.

To implement a CPHASE gate, we exploit a different avoided crossing, this time between the $|11\rangle$ and $|20\rangle$ states. Starting from the computational state $|11\rangle$, we slowly – that is, adiabatically – vary the flux applied to qubit 1 toward the avoided crossing point at ϕ_{CPHASE}, where the states $|11\rangle$ and $|20\rangle$ hybridize. After waiting for a duration τ at this point, we return adiabatically to the initial flux bias $\phi_0 = 0$. The final state remains $|11\rangle$ but acquires an additional phase relative to the other computational basis states. This conditional phase enables the implementation of the CPHASE gate.

This behavior can be understood using the adiabatic theorem [60, Section 5.6.2]. According to the theorem, if a system is initially prepared in an eigenstate of a time-dependent Hamiltonian $H(t)$ and the Hamiltonian varies sufficiently slowly, the system will remain in the corresponding instantaneous eigenstate throughout the evolution, acquiring only a phase factor. In the computational basis, and assuming that each basis state remains approximately in its instantaneous eigenstate throughout the trajectory $l(\tau)$, this results in the following unitary operator:

$$U_{\text{ad}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{i\theta_{01}(l)} & 0 & 0 \\ 0 & 0 & e^{i\theta_{10}(l)} & 0 \\ 0 & 0 & 0 & e^{i\theta_{11}(l)} \end{bmatrix}, \tag{4.24}$$

where the phase acquired by each state is given by:

$$\theta_{ij}(l(\tau)) = \int_0^\tau \omega_{ij}[l(t)]\, dt, \tag{4.25}$$

with ω_{ij} denoting the instantaneous energy of state $|ij\rangle$ during the flux trajectory $l(t)$ of qubit 1.

Now define $\zeta = \omega_{11} - (\omega_{01} + \omega_{10})$ as the phase accumulation rate difference between the $|11\rangle$ state and the combined $|01\rangle + |10\rangle$ states. This detuning arises due to the repulsion of the $|11\rangle$ level caused by the nearby $|20\rangle$ level at the avoided crossing.

By choosing a flux trajectory l_π such that the integrated detuning equals π,

$$\int_0^\tau \zeta[l_\pi(t)]\, dt = \theta_{11}(l_\pi) - \theta_{01}(l_\pi) - \theta_{10}(l_\pi) = \pi,$$

the resulting unitary evolution becomes:

$$U_{\text{ad}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{i\theta_{01}(l_\pi)} & 0 & 0 \\ 0 & 0 & e^{i\theta_{10}(l_\pi)} & 0 \\ 0 & 0 & 0 & e^{i[\pi+\theta_{01}(l_\pi)+\theta_{10}(l_\pi)]} \end{bmatrix}. \tag{4.26}$$

We can eliminate the accumulated single-qubit phases by applying appropriate virtual-Z gates such that $\theta_{01}(l_\pi) = \theta_{10}(l_\pi) = 0$. This yields the final unitary:

$$U_{\text{ad}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\pi} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} = \text{CZ}. \tag{4.27}$$

More generally, by engineering the flux pulse shape and duration, one can calibrate a wide range of controlled-phase gates, not limited to CZ. In the next section, we explore a purely microwave-based two-qubit gate mechanism – Cross-Resonance – used primarily in fixed-frequency transmon architectures.

4.4.3 Cross-Resonance

When flux-tunable qubits are not available, an alternative method for implementing two-qubit entangling gates is the Cross-Resonance (CR) technique [61]. The CR gate is a microwave-only scheme commonly used in fixed-frequency transmon architectures. The basic idea is to apply a microwave pulse on the control qubit at the transition frequency of the target qubit. Due to the qubit-qubit coupling, this induces a Rabi oscillation in the target qubit that depends on the state of the control qubit.

Figure 25 shows the expectation value of $\langle \sigma_z \rangle$ for the target qubit as a function of time, conditioned on the control qubit being in state $|0\rangle$ (blue) or $|1\rangle$ (red). Notice that the Rabi oscillation frequencies differ depending on the control state. To implement a CNOT gate, we seek a moment in time where the target qubit remains in $|0\rangle$ when the control is in $|0\rangle$, and flips to $|1\rangle$ when the control is in $|1\rangle$–*i.e.*, when the expectation values are $+1$ and -1, respectively. In this example, such synchronization occurs at $t = 180\,\text{ns}$.

However, in practice, these Rabi frequencies do not always align so neatly. In many cases, several oscillations are required before a point of synchronization suitable for a CNOT occurs, which may increase the gate time and reduce fidelity.

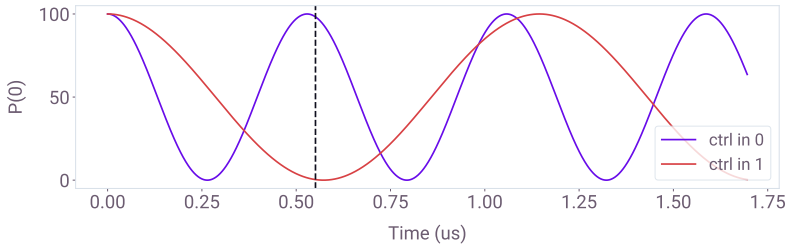


Fig. 25: Rabi oscillations of the target qubit induced by a cross-resonance pulse applied to the control qubit, obtained with the Qiskit Pulse simulator. The oscillation frequency depends on whether the control qubit is in $|0\rangle$ (purple) or $|1\rangle$ (red). The point at which the oscillations reach opposite extrema (here, around 0.55 μs) is suitable for implementing a CNOT gate.

To calibrate a cross-resonance gate, one should perform a Rabi experiment on the target qubit with the control qubit initialized in both $|0\rangle$ and $|1\rangle$, as described in Section 4.2.1. The result should be similar to that shown in Figure 26, where the oscillation curves allow estimation of the Rabi frequencies conditioned on the control qubit state and pulse amplitude. From this, an appropriate drive amplitude can be chosen to minimize the synchronization time and optimize the CR gate duration. Note that this amplitude is typically fine-tuned beyond the initial estimate for optimal performance.

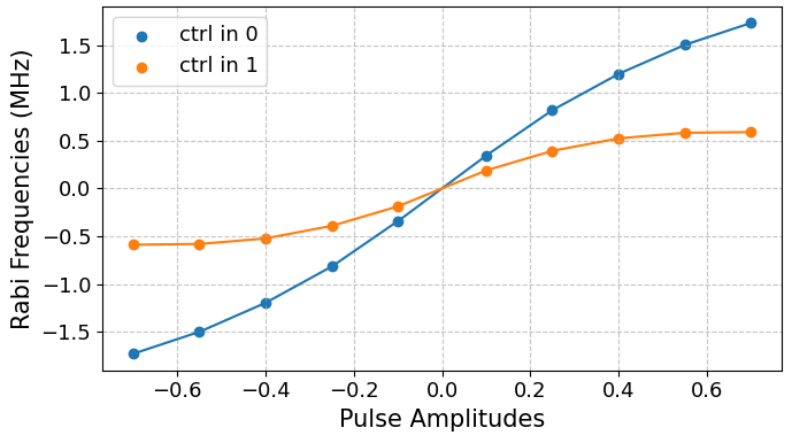


Fig. 26: Example of a cross-resonance Rabi calibration obtained with the Qiskit Pulse simulator. The target qubit undergoes Rabi oscillations induced by a cross-resonance pulse while the control qubit is prepared in $|0\rangle$ (blue) and $|1\rangle$ (orange). These curves are used to estimate the drive amplitude and gate time for implementing a CNOT.

To understand the functioning of the cross-resonance (CR) gate, we begin by modeling the system as two coupled transmon qubits, each truncated to its two lowest energy levels, $|0\rangle$ and $|1\rangle$. In this reduced basis, the effective two-qubit Hamiltonian takes the form:

$$H_{qq} = \sum_{i=1,2} \frac{\omega_i}{2} \sigma_z^{(i)} + g\, \sigma_y^{(1)} \otimes \sigma_y^{(2)}, \tag{4.28}$$

where ω_i is the transition frequency of qubit i, and g represents the strength of the coherent coupling between them.

We now include a drive applied to qubit 1, but at the frequency of qubit 2, which is the essence of the cross-resonance mechanism. The drive Hamiltonian is written as:

$$H_d(t) = V(t)\, \sigma_x^{(1)}, \tag{4.29}$$

where $V(t) = V_I(t) + V_Q(t)$, and the quadrature components are given by:

$$V_I(t) = A_I(t) \cos(\omega_2 t + \phi), \quad V_Q(t) = A_Q(t) \sin(\omega_2 t + \phi),$$

as introduced in Section 4.1. The total Hamiltonian of the driven system is:

$$H = H_{qq} + H_d(t). \tag{4.30}$$

To simplify the analysis and highlight the effective interaction induced on qubit 2, we apply the Schrieffer-Wolff transformation [62]. This perturbative method block-diagonalizes the Hamiltonian to first order, removing off-resonant couplings and yielding an effective interaction Hamiltonian of the form

$$\tilde{H}_d \approx V(t)\, \sigma_x^{(1)} + V(t)\, \frac{g}{\Delta_{12}}\, \sigma_z^{(1)} \otimes \sigma_x^{(2)}\,, \tag{4.31}$$

where $\Delta_{12} = \omega_1 - \omega_2$.

The second term, proportional to $\sigma_z^{(1)} \otimes \sigma_x^{(2)}$, corresponds to the effective interaction responsible for the cross-resonance mechanism [62]. When the drive is resonant with qubit 1 ($\omega_d = \omega_1$), the first term dominates and directly drives qubit 1. When the drive is tuned to qubit 2 ($\omega_d = \omega_2$), the second term becomes resonant, activating a conditional drive on qubit 2 that depends on the state of qubit 1. This conditional interaction enables the implementation of two-qubit gates such as the CNOT.

Consequently, the unitary of the CR gate is:

$$\mathrm{CR}(\theta) = e^{-i\frac{\theta}{2}\sigma_z \otimes \sigma_x} = \begin{bmatrix} \cos(\theta/2) & -i\sin(\theta/2) & 0 & 0 \\ -i\sin(\theta/2) & \cos(\theta/2) & 0 & 0 \\ 0 & 0 & \cos(\theta/2) & i\sin(\theta/2) \\ 0 & 0 & i\sin(\theta/2) & \cos(\theta/2) \end{bmatrix}. \tag{4.32}$$

This matrix reveals that simply calibrating the angle θ does not yield a standard CNOT gate. Additional single-qubit rotations are required to complete the transformation. In particular, when $\theta = -\pi/2$, a CNOT can be constructed using the CR gate together with $\sqrt{X}$ and $\sqrt{Z}$ gates:

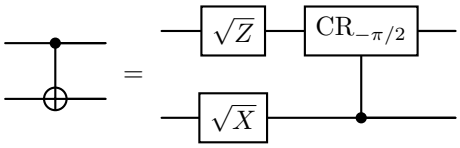


Fig. 27: Decomposition of a CNOT gate using the $\mathrm{CR}_{-\pi/2}$ gate. Here, the drive is applied to the top (control) qubit, inducing a conditional σ_x on the bottom (target) qubit.

We have now explored three key two-qubit gate mechanisms–iSWAP, CPHASE, and Cross-Resonance–each suited to different qubit architectures and coupling strategies. With these entangling gates in place, we now turn to the final step of quantum processing: how measurement is performed and calibrated in superconducting qubit systems.

4.5 Readout Calibration

To measure a qubit, we couple it to a readout resonator. Because of the dispersive interaction, the resonator frequency shifts depending on the qubit state. The basic idea of the dispersive readout is to apply a probe pulse to the resonator and analyze the reflected signal. Depending on the qubit state and the probe frequency, the amplitude and phase of the reflected signal are modified, allowing us to infer the qubit state.

In order to understand the behavior of the resonator coupled to the qubit [63], consider the following Hamiltonian:

$$H = \omega_r \left(a^\dagger a + \frac{1}{2} \right) + \frac{\omega_q}{2} \sigma_z + g \left(\sigma_+ a + \sigma_- a^\dagger \right), \tag{4.33}$$

where ω_r is the resonator frequency, ω_q is the qubit frequency, and g is the qubit–resonator coupling strength.

In the dispersive limit, where $\Delta = |\omega_q - \omega_r| \gg g$, the qubit and the resonator do not exchange energy resonantly, but instead cause small shifts in each other's frequencies. To analyze this regime, we apply the Schrieffer-Wolff transformation [63]. This leads to the effective Hamiltonian:

$$H_{\text{eff}} = (\omega_r + \chi \sigma_z) \left(a^\dagger a + \frac{1}{2} \right) + \frac{\tilde{\omega}_q}{2} \sigma_z + \text{const.}, \tag{4.34}$$

where $\chi = \frac{g^2}{\Delta}$ is the dispersive shift, and $\tilde{\omega}_q = \omega_q + \chi$ is the Lamb-shifted qubit frequency.

This effective Hamiltonian shows that the resonator frequency becomes conditional on the qubit state. When the qubit is in $|0\rangle$ or $|1\rangle$, the resonator effectively oscillates at $\omega_r \pm \chi$, respectively. This state-dependent frequency shift is the key principle behind dispersive readout: by sending a probe signal near ω_r and observing its phase and

amplitude response, one can infer the qubit state. At the same time, the qubit frequency is renormalized due to vacuum fluctuations of the resonator field, a correction known as the Lamb shift.

In practice, the measured quantity is the reflection coefficient S_{11} of the probe tone. As the probe frequency is varied across the resonance, both the amplitude and phase of S_{11} change in a way that depends on the qubit state. Typically, the reflected amplitude is reduced when the qubit is in one state and remains near unity in the other, while the corresponding phases differ significantly at specific detunings. The most common operating point is to fix the probe frequency at the bare resonator frequency ω_r, where the two qubit states produce the largest phase separation in the reflected signal.

Another way to visualize the information is in the complex plane, where S_{11} is represented as a vector with amplitude and phase. For each qubit state, repeated measurements form distinct clusters in this plane. The separation between these clusters quantifies the readout fidelity: the larger the separation, the easier it is to discriminate between $|0\rangle$ and $|1\rangle$.

Finally, to convert the analog readout signal into a digital qubit outcome, a classification algorithm is applied. A simple threshold on the measured quadrature components is often sufficient, but more sophisticated methods, such as maximum likelihood estimation or machine learning classifiers, can further improve readout accuracy.

5 Final Remarks

In this work, we have detailed the architecture and components of a comprehensive, full-stack quantum software platform, from the high-level programming interface provided by Ket down to the physical pulse-level control of superconducting qubits. The development of such a complete stack is crucial for bridging the significant gap between abstract quantum algorithms and the complex, noisy physical hardware of the NISQ era. Mastering the entire compilation and execution flow provides a powerful tool for co-designing hardware and software, optimizing performance, and accelerating the path toward practical quantum applications.

This work represents a significant step forward for the quantum computing ecosystem. The platform's potential for technological impact was recently underscored when it was awarded first place in the prestigious SBC Innovation Seal 2025, an award from the Brazilian Computer Society (SBC) that recognizes academic research with a high potential to become impactful technology. This positions the platform as a cornerstone of a sovereign, national quantum infrastructure in Brazil, empowering the nation's scientific community and industry.

Nevertheless, this platform is a foundation upon which much can be built. We identify several key avenues for future expansion and improvement. The immediate priorities include the integration of more advanced error mitigation techniques, which are essential for extracting meaningful results from current hardware. Looking further ahead, incorporating the principles of quantum error correction will be necessary for

the transition to fault-tolerant computing. The domain of quantum circuit optimization also offers vast room for enhancement, with opportunities to implement more sophisticated, hardware-aware optimization passes and explore advanced resynthesis techniques.

Looking ahead, our vision for the platform involves expanding the stack's capabilities at both the highest and lowest levels of abstraction. At the application level, we will develop domain-specific libraries to provide powerful tools for researchers in fields like quantum chemistry [21] and finance [22]. Concurrently, we will push downwards towards the hardware interface by integrating pulse-level programming capabilities [23] and support for direct Arbitrary Waveform Generator (AWG) management [24]. To foster a collaborative and transparent research environment, the core components of this stack are available at https://quantumket.org as an open-source platform, encouraging community contributions and accelerating the pace of innovation. This open model will not only benefit the research community but also serve as a valuable educational tool, helping to train the next generation of quantum scientists and engineers.

Acknowledgements. ECRR acknowledges the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - CAPES, Finance Code 001; EID, JM, and ECRR acknowledge the Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq through grant number 409673/2022-6; JM, EID, and ECRR acknowledge the Fundação de Amparo à Pesquisa e Inovação do Estado de Santa Catarina - FAPESC through Project FAPESC TR nº 2024TR002672; EID acknowledges financial support from the National Institute for Science and Technology of Quantum Information (CNPq, INCT-IQ 465469/2014-0); EWL acknowledges financial support from the Fundação de Amparo à Pesquisa e Inovação do Estado de Santa Catarina (FAPESC) under Public Call No. 24/2025.

References

[1] Shor, P.W.: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. SIAM Journal on Computing **26**(5), 1484–1509 (1997) https://doi.org/10.1137/S0097539795293172

[2] Arute, F., Arya, K., Babbush, R., Bacon, D., Bardin, J.C., Barends, R., Biswas, R., Boixo, S., Brandao, F.G.S.L., Buell, D.A., Burkett, B., Chen, Y., Chen, Z., Chiaro, B., Collins, R., Courtney, W., Dunsworth, A., Farhi, E., Foxen, B., Fowler, A., Gidney, C., Giustina, M., Graff, R., Guerin, K., Habegger, S., Harrigan, M.P., Hartmann, M.J., Ho, A., Hoffmann, M., Huang, T., Humble, T.S., Isakov, S.V., Jeffrey, E., Jiang, Z., Kafri, D., Kechedzhi, K., Kelly, J., Klimov, P.V., Knysh, S., Korotkov, A., Kostritsa, F., Landhuis, D., Lindmark, M., Lucero, E., Lyakh, D., Mandrà, S., McClean, J.R., McEwen, M., Megrant, A., Mi, X., Michielsen, K., Mohseni, M., Mutus, J., Naaman, O., Neeley, M., Neill, C., Niu, M.Y., Ostby, E., Petukhov, A., Platt, J.C., Quintana, C., Rieffel, E.G., Roushan, P., Rubin, N.C., Sank, D., Satzinger, K.J., Smelyanskiy, V., Sung, K.J., Trevithick, M.D., Vainsencher, A., Villalonga, B., White, T., Yao,

Z.J., Yeh, P., Zalcman, A., Neven, H., Martinis, J.M.: Quantum supremacy using a programmable superconducting processor. Nature **574**(7779), 505–510 (2019) https://doi.org/10.1038/s41586-019-1666-5

[3] Madsen, L.S., Laudenbach, F., Askarani, M.F., Rortais, F., Vincent, T., Bulmer, J.F.F., Miatto, F.M., Neuhaus, L., Helt, L.G., Collins, M.J., Lita, A.E., Gerrits, T., Nam, S.W., Vaidya, V.D., Menotti, M., Dhand, I., Vernon, Z., Quesada, N., Lavoie, J.: Quantum computational advantage with a programmable photonic processor. Nature **606**(7912), 75–81 (2022) https://doi.org/10.1038/s41586-022-04725-x

[4] Zhong, H.-S., Wang, H., Deng, Y.-H., Chen, M.-C., Peng, L.-C., Luo, Y.-H., Qin, J., Wu, D., Ding, X., Hu, Y., Hu, P., Yang, X.-Y., Zhang, W.-J., Li, H., Li, Y., Jiang, X., Gan, L., Yang, G., You, L., Wang, Z., Li, L., Liu, N.-L., Lu, C.-Y., Pan, J.-W.: Quantum computational advantage using photons. Science **370**(6523), 1460–1463 (2020) https://doi.org/10.1126/science.abe8770

[5] Wu, Y., Bao, W.-S., Cao, S., Chen, F., Chen, M.-C., Chen, X., Chung, T.-H., Deng, H., Du, Y., Fan, D., Gong, M., Guo, C., Guo, C., Guo, S., Han, L., Hong, L., Huang, H.-L., Huo, Y.-H., Li, L., Li, N., Li, S., Li, Y., Liang, F., Lin, C., Lin, J., Qian, H., Qiao, D., Rong, H., Su, H., Sun, L., Wang, L., Wang, S., Wu, D., Xu, Y., Yan, K., Yang, W., Yang, Y., Ye, Y., Yin, J., Ying, C., Yu, J., Zha, C., Zhang, C., Zhang, H., Zhang, K., Zhang, Y., Zhao, H., Zhao, Y., Zhou, L., Zhu, Q., Lu, C.-Y., Peng, C.-Z., Zhu, X., Pan, J.-W.: Strong Quantum Computational Advantage Using a Superconducting Quantum Processor. Physical Review Letters **127**(18), 180501 (2021) https://doi.org/10.1103/PhysRevLett.127.180501

[6] Zhong, H.-S., Deng, Y.-H., Qin, J., Wang, H., Chen, M.-C., Peng, L.-C., Luo, Y.-H., Wu, D., Gong, S.-Q., Su, H., Hu, Y., Hu, P., Yang, X.-Y., Zhang, W.-J., Li, H., Li, Y., Jiang, X., Gan, L., Yang, G., You, L., Wang, Z., Li, L., Liu, N.-L., Renema, J.J., Lu, C.-Y., Pan, J.-W.: Phase-Programmable Gaussian Boson Sampling Using Stimulated Squeezed Light. Physical Review Letters **127**(18), 180502 (2021) https://doi.org/10.1103/PhysRevLett.127.180502

[7] Pokharel, B., Lidar, D.A.: Demonstration of Algorithmic Quantum Speedup. Physical Review Letters **130**(21), 210602 (2023) https://doi.org/10.1103/PhysRevLett.130.210602

[8] Liu, H.-L., Su, H., Gong, S.-Q., Gu, Y.-C., Tang, H.-Y., Jia, M.-H., Wei, Q., Song, Y., Wang, D., Zheng, M., Chen, F., Li, L., Ren, S., Zhu, X., Wang, M., Chen, Y., Liu, Y., Song, L., Yang, P., Chen, J., An, H., Zhang, L., Gan, L., Yang, G., Xu, J.-M., He, Y.-M., Wang, H., Zhong, H.-S., Chen, M.-C., Jiang, X., Li, L., Liu, N.-L., Deng, Y.-H., Su, X.-L., Zhang, Q., Lu, C.-Y., Pan, J.-W.: Robust Quantum Computational Advantage with Programmable 3050-Photon Gaussian Boson Sampling. arXiv (2025). https://doi.org/10.48550/arXiv.2508.09092

[9] Preskill, J.: Quantum Computing and the Entanglement Frontier. arXiv (2012).

https://doi.org/10.48550/ARXIV.1203.5813

[10] Google Quantum AI and Collaborators, Acharya, R., Abanin, D.A., Aghababaie-Beni, L., Aleiner, I., Andersen, T.I., Ansmann, M., Arute, F., Arya, K., Asfaw, A., Astrakhantsev, N., Atalaya, J., Babbush, R., Bacon, D., Ballard, B., Bardin, J.C., Bausch, J., Bengtsson, A., Bilmes, A., Blackwell, S., Boixo, S., Bortoli, G., Bourassa, A., Bovaird, J., Brill, L., Broughton, M., Browne, D.A., Buchea, B., Buckley, B.B., Buell, D.A., Burger, T., Burkett, B., Bushnell, N., Cabrera, A., Campero, J., Chang, H.-S., Chen, Y., Chen, Z., Chiaro, B., Chik, D., Chou, C., Claes, J., Cleland, A.Y., Cogan, J., Collins, R., Conner, P., Courtney, W., Crook, A.L., Curtin, B., Das, S., Davies, A., De Lorenzo, L., Debroy, D.M., Demura, S., Devoret, M., Di Paolo, A., Donohoe, P., Drozdov, I., Dunsworth, A., Earle, C., Edlich, T., Eickbusch, A., Elbag, A.M., Elzouka, M., Erickson, C., Faoro, L., Farhi, E., Ferreira, V.S., Burgos, L.F., Forati, E., Fowler, A.G., Foxen, B., Ganjam, S., Garcia, G., Gasca, R., Genois, E., Giang, W., Gidney, C., Gilboa, D., Gosula, R., Dau, A.G., Graumann, D., Greene, A., Gross, J.A., Habegger, S., Hall, J., Hamilton, M.C., Hansen, M., Harrigan, M.P., Harrington, S.D., Heras, F.J.H., Heslin, S., Heu, P., Higgott, O., Hill, G., Hilton, J., Holland, G., Hong, S., Huang, H.-Y., Huff, A., Huggins, W.J., Ioffe, L.B., Isakov, S.V., Iveland, J., Jeffrey, E., Jiang, Z., Jones, C., Jordan, S., Joshi, C., Juhas, P., Kafri, D., Kang, H., Karamlou, A.H., Kechedzhi, K., Kelly, J., Khaire, T., Khattar, T., Khezri, M., Kim, S., Klimov, P.V., Klots, A.R., Kobrin, B., Kohli, P., Korotkov, A.N., Kostritsa, F., Kothari, R., Kozlovskii, B., Kreikebaum, J.M., Kurilovich, V.D., Lacroix, N., Landhuis, D., Lange-Dei, T., Langley, B.W., Laptev, P., Lau, K.-M., Le Guevel, L., Ledford, J., Lee, J., Lee, K., Lensky, Y.D., Leon, S., Lester, B.J., Li, W.Y., Li, Y., Lill, A.T., Liu, W., Livingston, W.P., Locharla, A., Lucero, E., Lundahl, D., Lunt, A., Madhuk, S., Malone, F.D., Maloney, A., Mandrà, S., Manyika, J., Martin, L.S., Martin, O., Martin, S., Maxfield, C., McClean, J.R., McEwen, M., Meeks, S., Megrant, A., Mi, X., Miao, K.C., Mieszala, A., Molavi, R., Molina, S., Montazeri, S., Morvan, A., Movassagh, R., Mruczkiewicz, W., Naaman, O., Neeley, M., Neill, C., Nersisyan, A., Neven, H., Newman, M., Ng, J.H., Nguyen, A., Nguyen, M., Ni, C.-H., Niu, M.Y., O'Brien, T.E., Oliver, W.D., Opremcak, A., Ottosson, K., Petukhov, A., Pizzuto, A., Platt, J., Potter, R., Pritchard, O., Pryadko, L.P., Quintana, C., Ramachandran, G., Reagor, M.J., Redding, J., Rhodes, D.M., Roberts, G., Rosenberg, E., Rosenfeld, E., Roushan, P., Rubin, N.C., Saei, N., Sank, D., Sankaragomathi, K., Satzinger, K.J., Schurkus, H.F., Schuster, C., Senior, A.W., Shearn, M.J., Shorter, A., Shutty, N., Shvarts, V., Singh, S., Sivak, V., Skruzny, J., Small, S., Smelyanskiy, V., Smith, W.C., Somma, R.D., Springer, S., Sterling, G., Strain, D., Suchard, J., Szasz, A., Sztein, A., Thor, D., Torres, A., Torunbalci, M.M., Vaishnav, A., Vargas, J., Vdovichev, S., Vidal, G., Villalonga, B., Heidweiller, C.V., Waltman, S., Wang, S.X., Ware, B., Weber, K., Weidel, T., White, T., Wong, K., Woo, B.W.K., Xing, C., Yao, Z.J., Yeh, P., Ying, B., Yoo, J., Yosri, N., Young, G., Zalcman, A., Zhang, Y., Zhu, N., Zobrist, N.: Quantum error correction below the surface code threshold. Nature **638**(8052), 920–926 (2025) https://doi.org/10.1038/s41586-024-08449-y

[11] Preskill, J.: Quantum Computing in the NISQ era and beyond. Quantum **2**, 79 (2018) https://doi.org/10.22331/q-2018-08-06-79

[12] Javadi-Abhari, A., Treinish, M., Krsulich, K., Wood, C.J., Lishman, J., Gacon, J., Martiel, S., Nation, P.D., Bishop, L.S., Cross, A.W., Johnson, B.R., Gambetta, J.M.: Quantum Computing with Qiskit. arXiv (2024). https://doi.org/10.48550/arXiv.2405.08810

[13] Svore, K., Geller, A., Troyer, M., Azariah, J., Granade, C., Heim, B., Kliuchnikov, V., Mykhailova, M., Paz, A., Roetteler, M.: Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In: Proceedings of the Real World Domain Specific Languages Workshop 2018, pp. 1–10. ACM, Vienna Austria (2018). https://doi.org/10.1145/3183895.3183901

[14] Steiger, D.S., Häner, T., Troyer, M.: ProjectQ: An open source software framework for quantum computing. Quantum **2**, 49 (2018) https://doi.org/10.22331/q-2018-01-31-49

[15] Bichsel, B., Baader, M., Gehr, T., Vechev, M.: Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 286–300. ACM, London UK (2020). https://doi.org/10.1145/3385412.3386007

[16] Smith, R.S., Curtis, M.J., Zeng, W.J.: A Practical Quantum Instruction Set Architecture. arXiv (2016). https://doi.org/10.48550/ARXIV.1608.03355

[17] Sivarajah, S., Dilkes, S., Cowtan, A., Simmons, W., Edgington, A., Duncan, R.: T|ket⟩: A retargetable compiler for NISQ devices. Quantum Science and Technology **6**(1), 014003 (2021) https://doi.org/10.1088/2058-9565/ab8e92

[18] Efthymiou, S., Ramos-Calderer, S., Bravo-Prieto, C., Pérez-Salinas, A., García-Martín, D., Garcia-Saez, A., Latorre, J.I., Carrazza, S.: Qibo: A framework for quantum simulation with hardware acceleration. Quantum Science and Technology **7**(1), 015018 (2022) https://doi.org/10.1088/2058-9565/ac39f5

[19] Kernighan, B.W., Ritchie, D.M.: The C Programming Language, 28. print edn. Prentice-Hall Software Series. Prentice-Hall, Englewood Cliffs, NJ (1991)

[20] Da Rosa, E.C.R., De Santiago, R.: Ket Quantum Programming. ACM Journal on Emerging Technologies in Computing Systems **18**(1), 1–25 (2022) https://doi.org/10.1145/3474224

[21] Tilly, J., Chen, H., Cao, S., Picozzi, D., Setia, K., Li, Y., Grant, E., Wossnig, L., Rungger, I., Booth, G.H., Tennyson, J.: The Variational Quantum Eigensolver: A review of methods and best practices. Physics Reports **986**, 1–128 (2022) https://doi.org/10.1016/j.physrep.2022.08.003

[22] Herman, D., Googin, C., Liu, X., Sun, Y., Galda, A., Safro, I., Pistoia, M., Alexeev, Y.: Quantum computing for finance. Nature Reviews Physics **5**(8), 450–465 (2023) https://doi.org/10.1038/s42254-023-00603-1

[23] Alexander, T., Kanazawa, N., Egger, D.J., Capelluto, L., Wood, C.J., Javadi-Abhari, A., C McKay, D.: Qiskit pulse: Programming quantum computers through the cloud with pulses. Quantum Science and Technology **5**(4), 044006 (2020) https://doi.org/10.1088/2058-9565/aba404

[24] Stefanazzi, L., Treptow, K., Wilcer, N., Stoughton, C., Bradford, C., Uemura, S., Zorzetti, S., Montella, S., Cancelo, G., Sussman, S., Houck, A., Saxena, S., Arnaldi, H., Agrawal, A., Zhang, H., Ding, C., Schuster, D.I.: The QICK (Quantum Instrumentation Control Kit): Readout and control for qubits and detectors. Review of Scientific Instruments **93**(4), 044709 (2022) https://doi.org/10.1063/5.0076249

[25] Devitt, S.J., Munro, W.J., Nemoto, K.: Quantum error correction for beginners. Reports on Progress in Physics **76**(7), 076001 (2013) https://doi.org/10.1088/0034-4885/76/7/076001

[26] Chatterjee, A., Phalak, K., Ghosh, S.: Quantum Error Correction For Dummies (2023) https://doi.org/10.48550/ARXIV.2304.08678

[27] Rosa, E.C.R., Duzzioni, E.I., De Santiago, R.: Optimizing Gate Decomposition for High-Level Quantum Programming. Quantum **9**, 1659 (2025) https://doi.org/10.22331/q-2025-03-12-1659

[28] Rosa, E.C.R., Marchi, J., Duzzioni, E.I., Santiago, R.: Quantum Gate Decomposition: A Study of Compilation Time vs. Execution Time Trade-offs. arXiv (2025). https://doi.org/10.48550/arXiv.2504.20291

[29] Maslov, D.: Advantages of using relative-phase Toffoli gates with an application to multiple control Toffoli optimization. Physical Review A **93**(2), 022311 (2016) https://doi.org/10.1103/PhysRevA.93.022311

[30] Da Silva, A.J., Park, D.K.: Linear-depth quantum circuits for multiqubit controlled gates. Physical Review A **106**(4), 042602 (2022) https://doi.org/10.1103/PhysRevA.106.042602

[31] Siraichi, M.Y., Santos, V.F.D., Collange, C., Pereira, F.M.Q.: Qubit allocation. In: Proceedings of the 2018 International Symposium on Code Generation and Optimization, pp. 113–125. ACM, Vienna Austria (2018). https://doi.org/10.1145/3168822

[32] Li, G., Ding, Y., Xie, Y.: Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating

Systems, pp. 1001–1014. ACM, Providence RI USA (2019). https://doi.org/10.1145/3297858.3304023

[33] Zhu, P., Guan, Z., Cheng, X.: A Dynamic Look-Ahead Heuristic for the Qubit Mapping Problem of NISQ Computers. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **39**(12), 4721–4735 (2020) https://doi.org/10.1109/TCAD.2020.2970594

[34] Chowdhury, S., Gill, R., Badhoutiya, A., Srivastava, A.P., Khan, A.K., Singh, R.: Qubit Allocation Strategies in Quantum Computing for Improved Computational Efficiency. In: 2024 4th International Conference on Innovative Practices in Technology and Management (ICIPTM), pp. 1–6. IEEE, Noida, India (2024). https://doi.org/10.1109/ICIPTM59628.2024.10563524

[35] Niu, S., Suau, A., Staffelbach, G., Todri-Sanial, A.: A Hardware-Aware Heuristic for the Qubit Mapping Problem in the NISQ Era. IEEE Transactions on Quantum Engineering **1**, 1–14 (2020) https://doi.org/10.1109/TQE.2020.3026544

[36] Verteletskyi, V., Yen, T.-C., Izmaylov, A.F.: Measurement optimization in the variational quantum eigensolver using a minimum clique cover. The Journal of Chemical Physics **152**(12), 124114 (2020) https://doi.org/10.1063/1.5141458

[37] Yen, T.-C., Verteletskyi, V., Izmaylov, A.F.: Measuring All Compatible Operators in One Series of Single-Qubit Measurements Using Unitary Transformations. Journal of Chemical Theory and Computation **16**(4), 2400–2409 (2020) https://doi.org/10.1021/acs.jctc.0c00008

[38] Huang, H.-Y., Kueng, R., Preskill, J.: Predicting many properties of a quantum system from very few measurements. Nature Physics **16**(10), 1050–1057 (2020) https://doi.org/10.1038/s41567-020-0932-7

[39] Bertuzzi, L., Engster, J.P., Da Rosa, E.C.R., Duzzioni, E.I.: Shadow measurements for feedback-based quantum optimization. Physical Review A **112**(2), 022419 (2025) https://doi.org/10.1103/snht-7jsf

[40] Mitarai, K., Negoro, M., Kitagawa, M., Fujii, K.: Quantum circuit learning. Physical Review A **98**(3), 032309 (2018) https://doi.org/10.1103/PhysRevA.98.032309

[41] Schuld, M., Bergholm, V., Gogolin, C., Izaac, J., Killoran, N.: Evaluating analytic gradients on quantum hardware. Physical Review A **99**(3), 032331 (2019) https://doi.org/10.1103/PhysRevA.99.032331

[42] Banchi, L., Crooks, G.E.: Measuring Analytic Gradients of General Quantum Evolution with the Stochastic Parameter Shift Rule. Quantum **5**, 386 (2021) https://doi.org/10.22331/q-2021-01-25-386

[43] Coecke, B., Duncan, R.: Interacting quantum observables: Categorical algebra and diagrammatics. New Journal of Physics **13**(4), 043016 (2011) https://doi.org/10.1088/1367-2630/13/4/043016

[44] Maslov, D., Nam, Y., Kim, J.: An outlook for Quantum computing [point of view]. Proceedings of the IEEE **107**(1), 5–10 (2019) https://doi.org/10.1109/JPROC.2018.2884353

[45] Gill, S.S., Kumar, A., Singh, H., Singh, M., Kaur, K., Usman, M., Buyya, R.: Quantum computing: A taxonomy, systematic review and future directions. Software: Practice and Experience **52**(1), 66–114 (2022) https://doi.org/10.1002/spe.3039

[46] Fossheim, K., Sudbø, A.: Superconductivity: Physics and Applications, 1st edn. John Wiley & Sons, Hoboken, NJ (2004). https://doi.org/10.1002/0470020784

[47] Krantz, P., Kjaergaard, M., Yan, F., Orlando, T.P., Gustavsson, S., Oliver, W.D.: A quantum engineer's guide to superconducting qubits. Applied Physics Reviews **6**(2) (2019) https://doi.org/10.1063/1.5089550

[48] Blais, A., Grimsmo, A.L., Girvin, S.M., Wallraff, A.: Circuit quantum electrodynamics. Reviews of Modern Physics **93**(2), 025005 (2021) https://doi.org/10.1103/RevModPhys.93.025005

[49] Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information, 10th anniversary edition edn. Cambridge university press, Cambridge (2010). https://doi.org/10.1017/CBO9780511976667

[50] Carvalho, A.R.R., Ball, H., Biercuk, M.J., Hush, M.R., Thomsen, F.: Error-Robust Quantum Logic Optimization Using a Cloud Quantum Computer Interface. Physical Review Applied **15**(6), 064054 (2021) https://doi.org/10.1103/PhysRevApplied.15.064054

[51] Razavi, B.: Principles of Data Conversion System Design. Wiley-IEEE Press, New York (1995)

[52] Koch, C.P., Boscain, U., Calarco, T., Dirr, G., Filipp, S., Glaser, S.J., Kosloff, R., Montangero, S., Schulte-Herbrüggen, T., Sugny, D., Wilhelm, F.K.: Quantum optimal control in quantum technologies. Strategic report on current status, visions and goals for research in Europe. EPJ Quantum Technology **9**(1), 19 (2022) https://doi.org/10.1140/epjqt/s40507-022-00138-x

[53] Motzoi, F., Gambetta, J.M., Rebentrost, P., Wilhelm, F.K.: Simple pulses for elimination of leakage in weakly nonlinear qubits. Physical Review Letters **103**(11), 110501 (2009) https://doi.org/10.1103/PhysRevLett.103.110501

[54] Baum, Y., Amico, M., Howell, S., Hush, M., Liuzzi, M., Mundada, P., Merkh,

T., Carvalho, A.R.R., Biercuk, M.J.: Experimental deep reinforcement learning for error-robust gate-set design on a superconducting Quantum computer. PRX Quantum **2**(4), 040324 (2021) https://doi.org/10.1103/PRX{Quantum}.2.0403

[55] Boykin, P.O., Mor, T., Pulver, M., Roychowdhury, V., Vatan, F.: A new universal and fault-tolerant quantum basis. Information Processing Letters **75**(3), 101–107 (2000) https://doi.org/10.1016/S0020-0190(00)00084-3

[56] Kitaev, A.Y.: Quantum computations: Algorithms and error correction. Russian Mathematical Surveys **52**(6), 1191 (1997) https://doi.org/10.1070/RM1997v052n06ABEH002155

[57] Shi, Y.: Both toffoli and controlled-NOT need little help to do universal Quantum computing. Quantum Info. Comput. **3**(1), 84–92 (2003)

[58] McKay, D.C., Wood, C.J., Sheldon, S., Chow, J.M., Gambetta, J.M.: Efficient Z gates for quantum computing. Physical Review A: Atomic, Molecular, and Optical Physics **96**(2), 022330 (2017) https://doi.org/10.1103/PhysRevA.96.022330

[59] Schuch, N., Siewert, J.: Natural two-qubit gate for quantum computation using the XY interaction. Physical Review A: Atomic, Molecular, and Optical Physics **67**(3), 032301 (2003) https://doi.org/10.1103/PhysRevA.67.032301

[60] Sakurai, J.J., Napolitano, J.: Modern Quantum Mechanics, 3rd edn. Cambridge University Press, Cambridge (2020). https://doi.org/10.1017/9781108587280

[61] Rigetti, C., Devoret, M.: Fully microwave-tunable universal gates in superconducting qubits with linear couplings and fixed transition frequencies. Physical Review B **81**(13), 134507 (2010) https://doi.org/10.1103/PhysRevB.81.134507

[62] Bravyi, S., DiVincenzo, D.P., Loss, D.: Schrieffer–Wolff transformation for quantum many-body systems. Annals of Physics **326**(10), 2793–2826 (2011) https://doi.org/10.1016/j.aop.2011.06.004

[63] Koch, J., Yu, T.M., Gambetta, J., Houck, A.A., Schuster, D.I., Majer, J., Blais, A., Devoret, M.H., Girvin, S.M., Schoelkopf, R.J.: Charge-insensitive qubit design derived from the Cooper pair box. Physical Review A: Atomic, Molecular, and Optical Physics **76**(4), 042319 (2007) https://doi.org/10.1103/PhysRevA.76.042319